EnclaveDB: A Secure Database using SGX

Christian Priebe
Imperial College London
christian.priebe13@imperial.ac.uk

Kapil Vaswani
Microsoft Research
kapilv@microsoft.com

Manuel Costa
Microsoft Research
manuelc@microsoft.com

***Abstract*—We propose EnclaveDB, a database engine that guarantees confidentiality, integrity, and freshness for data and queries. EnclaveDB guarantees these properties even when the database administrator is malicious, when an attacker has compromised the operating system or the hypervisor, and when the database runs in an untrusted host in the cloud. EnclaveDB achieves this by placing sensitive data (tables, indexes and other metadata) in *enclaves* protected by trusted hardware (such as Intel SGX). EnclaveDB has a small trusted computing base, which includes an in-memory storage and query engine, a transaction manager and pre-compiled stored procedures. A key component of EnclaveDB is an efficient protocol for checking integrity and freshness of the database log. The protocol supports concurrent, asynchronous appends and truncation, and requires minimal synchronization between threads. Our experiments using standard database benchmarks and a performance model that simulates large enclaves show that EnclaveDB achieves strong security with low overhead (up to 40% for TPC-C) compared to an industry strength in-memory database engine.**

I. Introduction

Modern data processing services hosted in cloud environments are under constant attack from malicious entities such as database administrators, server administrators, hackers who exploit bugs in the operating system or hypervisor, and even nation states. This results in frequent data breaches that reduce trust in online services. Semantically secure encryption can provide strong and efficient protection for data at rest and in transit, but this is not sufficient because data processing systems decrypt sensitive data in memory during query processing. Systems such as CryptDB [1], Monomi [2] and Seabed [3] use property-preserving encryption to allow query processing on encrypted data. This approach has been adopted in several products [4], [5], but suffers from limited querying capabilities and is prone to information leakage [6], [7], [8].

Another approach to enable secure query processing is to use trusted execution environments or *enclaves*. Enclaves (e.g., Intel Software Guard Extensions (SGX) [9]) can protect sensitive data and code, even from powerful attackers that control or have compromised the operating system and the hypervisor on a host machine. While enclaves can mitigate several attacks, using them requires careful refactoring of applications into trusted and untrusted components to achieve desired security and privacy goals. Furthermore, ensuring high level security properties such as confidentiality, integrity, and freshness requires additional logic to protect secrets when they leave the enclave and verify their integrity when they are read. This task is relatively simple in applications such as password checkers, key management systems and simpler

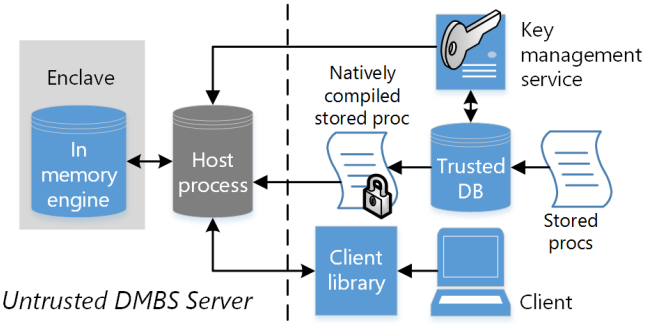

Fig. 1: Overview of EnclaveDB's architecture. EnclaveDB hosts sensitive data along with natively compiled queries and a query engine in an enclave.

data processing frameworks. For example, researchers have proposed the use of enclaves for building secure versions of streaming and batch processing frameworks in Opaque [10] and VC3 [11]. However, redesigning more complex systems such as databases to use enclaves and offer strong security properties remains an open problem. Previous work that places small pieces of the query engine in trusted hardware, such as Cipherbase [12] and TrustedDB [13], does not provide confidentiality for queries or integrity and freshness for data. Alternatively, hosting the whole database service inside an enclave [14] results in a large trusted computing base (TCB) and increased performance overheads, and does not provide protection from the database administrator.

In this paper, we propose EnclaveDB, a database that ensures confidentiality, integrity, and freshness for queries and data. EnclaveDB has a programming model similar to conventional relational databases – authorized users can create tables and indexes, and query the tables using stored procedures expressed in SQL. However, unlike a conventional database, EnclaveDB provides security from hackers, malicious server administrators and database administrators. As shown in Figure 1, EnclaveDB protects database state by hosting all *sensitive* data (tables, indexes, queries and other intermediate state) in enclave memory. This design choice is feasible due to rapidly decreasing DRAM costs and the expected availability of systems which support large enclaves (of the order of several hundred gigabytes).

Unlike a conventional database, EnclaveDB compiles queries on sensitive data to native code using an ahead-of-time compiler on a trusted client machine. Pre-compiled queries are

signed, encrypted and deployed to an enclave on the untrusted database server. Decoupling compilation from execution allows components such as the query parser, compiler and optimizer to be hosted in a trusted environment, thereby reducing the attack surface available to the adversary. EnclaveDB clients execute pre-compiled queries by establishing a secure channel with the enclave and sending requests with encrypted parameters. The enclave authenticates requests, decrypts parameters, executes the pre-compiled query, encrypts query results, and sends the results back to the client.

In addition to confidentiality of data and queries, EnclaveDB also guarantees integrity and freshness of data. Integrity and freshness are critical properties for many applications such as banking, auctions, voting and control systems. The integrity of in-memory tables and indexes is guaranteed by the enclave hardware. The integrity of queries is ensured by hosting queries and a transaction manager within the enclave. In addition, EnclaveDB employs a number of checks to detect and prevent integrity violations during query processing and database recovery. This includes checks to detect invalid API usage and *Iago* attacks [15] caused by a malicious database server/operating system that violates its specification. A key component of EnclaveDB is an efficient protocol for ensuring confidentiality, integrity and freshness of the database log. The protocol supports concurrent appends and truncation of the log and requires minimal synchronization between threads.

We have built a prototype of EnclaveDB using Hekaton, SQL Server's in-memory database engine. Our prototype has a small TCB (over 100X smaller than a conventional database server). We evaluate the performance of EnclaveDB using industry-standard database benchmarks and a performance model that accounts for the overheads of enclaves. Our evaluation shows that EnclaveDB delivers high performance (up to 31,000 tps for TPC-C) and has low overheads (up to 40% lower throughput compared to an insecure baseline).

This paper makes the following contributions.

- We propose EnclaveDB, an in-memory database that uses enclaves to provide strong security properties.
- EnclaveDB guarantees confidentiality, integrity and freshness using a combination of encryption, native compilation and a scalable protocol for checking integrity and freshness of the database log.
- EnclaveDB has a small TCB - over 100X smaller than a conventional database server.
- We evaluate EnclaveDB using standard benchmarks and a performance model that simulates enclave overheads. The evaluation shows that EnclaveDB delivers security with high performance.

The rest of this paper is organized as follows. We start with an overview of enclaves and the Hekaton engine in Section II. In Section III, we discuss our threat model. Section IV explores EnclaveDB's architecture. We then describe the protocol for checking integrity of checkpoints and the log in Section V. Section VI describes several optimizations and Section VII discusses multi-party support, followed by a detailed evaluation of EnclaveDB in Section VIII. Finally, we present related work in Section IX and conclude with Section X.

II. Background

A. Enclaves

Trusted execution environments or *enclaves* such as Intel SGX protect code and data from all other software in a system. With OS support, an untrusted hosting application can create an enclave in its virtual address space. Once an enclave has been initialized, code and data within the enclave is isolated from the rest of the system, including privileged software. Application threads can however switch into enclave mode at pre-defined entry points and execute user-mode instructions.

Intel SGX enforces isolation by storing enclave code and data in a data structure called the Enclave Page Cache (EPC), which resides in a preconfigured portion of DRAM called the Processor Reserved Memory (PRM). The processor ensures that any software outside the enclave cannot access the PRM. However, code hosted inside an enclave can access both non-PRM memory and PRM memory that belongs to the enclave. SGX includes a memory encryption engine which encrypts and authenticates enclave data evicted to memory, and ensures integrity and freshness using a merkle-tree structure over the EPC. SGX also protects enclaves against a variety of hardware/software attacks including attempts to access enclave memory via DMA or by reusing cached TLB translations.

In addition to isolation, enclaves also support *sealing* and *remote attestation*. Sealing allows an enclave to securely persist and retrieve secrets on the local host. Sealing keys can be bound to a specific enclave identity or a signing authority, e.g. the enclave owner. Sealed data is confidentiality- and integrity-protected, but sealing does not provide freshness guarantees. Remote attestation allows a remote challenger to establish trust in an enclave. In Intel SGX, code hosted in an enclave can request for a *quote*, which contains a number of enclave attributes including a measurement of the enclave's initial state. The quote is signed by a processor-specific attestation key. A remote challenger can use Intel's attestation verification service to verify that a given quote has been signed by a valid attestation key. The challenger can also verify that the enclave has been initialized in an expected state. Once an enclave has been verified, the challenger can set up a secure channel with the enclave (using a secure key exchange protocol) and provision secrets such as encrypted code or data encryption keys to the enclave.

B. Hekaton

Hekaton [16] is a database engine in SQL Server optimized for OLTP workloads where data fits in memory. The engine's design is based on the observation that memory prices are dropping and machines with over 1TB of memory are already commonplace. As a result, datasets for many OLTP workloads can fit entirely in memory.

Hekaton allows users to host selected tables in memory and create one or more memory-resident indexes on the table. Hekaton tables are durable - the Hekaton engine logs

transactions on memory resident tables to a persistent log shared with SQL Server. Periodically, Hekaton *checkpoints* the log by compressing log records into a more compact representation on disk. On failure, the database state can be recovered using checkpoints and the tail of the log. To further optimize performance, Hekaton supports a mode of execution where table definitions and SQL queries over in-memory tables are compiled to efficient machine code. Native compilation is restricted to queries where decisions typically made by the query interpreter at runtime can be made at compile time, e.g., queries where the data types of all columns and variables are known at compile time. These restrictions allow the Hekaton compiler to generate efficient code with optimized control flow and no runtime type checks.

In this paper, we show that the principles behind the design of a high performance database engine are aligned with security. Specifically, in-memory tables and indexes are ideal data structures for securely hosting and querying sensitive data in enclaves. In-memory tables eliminate the need for expensive software encryption and integrity checking otherwise required for disk-based tables. Query processing on in-memory data minimizes the leakage of sensitive information and the number of transitions between the enclave and the host. Finally, native compilation allows query compilation and optimization to be decoupled from query execution, enabling a mode of compilation where queries are compiled on a trusted database and deployed to an enclave on an untrusted server, significantly reducing the attack surface available to an adversary.

III. Threat model

We consider a strong adversary that controls the entire software stack on the database server, except the code inside enclaves. This represents threats from an untrusted server administrator, the database administrator, and attackers who may compromise the operating system, the hypervisor or the database server. The adversary can access *and tamper* with any server-side state in memory, on disk and over the network. This includes attacks that tamper with database files such as logs and checkpoints e.g. overwriting, dropping, duplicating and/or reordering log records. The adversary can mount replay attacks by arbitrarily shutting down the database and attempting to recover from a stale state. The adversary can attempt to fork the database e.g. by running multiple replicas of the database instance on the same or different machines and sending requests from different clients to different instances. The adversary can also observe and arbitrarily change control flow e.g. make an arbitrary sequence of calls to any of the predefined entry points in the enclave. However, denial of service and side channels attacks (e.g., access patterns and timing) are outside the scope of this paper. Side channels are a serious concern with trusted hardware [17], [18], [19], [20], and building efficient side channel protection for high performance systems like EnclaveDB remains an open problem.

We trust the processor and assume the adversary cannot physically open the processor package and extract secrets or corrupt state inside the processor. We assume that the code placed inside the enclave is correct and does not leak secrets intentionally. Recent research has shown that it is possible to automatically enforce and verify confidentiality for reasonably sized applications at low runtime overheads [21]. We also assume that all client-side components such as SQL clients and the key management service are trusted. This is a common assumption in cloud-based systems and often realized by hosting the client in a trusted on-premises environment (e.g. behind firewalls controlled by the user) or in enclaves.

For encryption, we rely on a scheme that provides authenticated encryption with associated data (AEAD). We write $Enc[k]\{ad\}(text)$ to represent the encryption of $text$ using key k and authentication data ad and assume that the result contains the authenticated data. We write $Dec[k](enc)$ to represent authentication and decryption of ciphertext containing authentication data. We assume that our scheme is both IND-CPA and IND-CTXT. Our implementation uses AES-GCM [22], a high-performance AEAD scheme.

Even under this threat model, we wish to guarantee both confidentiality and *linearizability* [23]. In a linearizable database, transactions appear to execute sequentially in an order consistent with real time. Therefore, clients do not have to reason about concurrency or failures. In our context, a linearizable database frees the clients from having to reason about an active attacker. In addition to linearizability, we would also like to ensure *liveness* i.e. the database should always be able to recover from unexpected shutdowns at any time. Note that liveness does not imply availability; an attacker can always prevent progress e.g. by not allocating resources.

IV. Architecture

An EnclaveDB service consists of an untrusted database server that hosts public data and an enclave that contains sensitive data. Figure 2 shows the server-side components in EnclaveDB. The enclave hosts a modified Hekaton query processing engine, natively compiled stored procedures, and a trusted kernel which provides a runtime environment for the database engine and security primitives such as attestation and sealing. The untrusted host process runs all other components of the database server, including a query compiler and processor for public data, and the log and storage managers. The untrusted server supports database administration tasks – the database administrator may login to the database and perform maintenance operations (e.g., backups, troubleshooting of server problems, configuration of storage options), but does not get access to sensitive data. This is critically important, since database administration is often outsourced (e.g., to a cloud provider or third parties).

The query processor on the untrusted database server supports generic queries on public data; in cases where such data exists, it can be kept out of the enclave, e.g., as a performance optimization. The query processor is also responsible for receiving requests to execute stored procedures on secret data and handing them over to the Hekaton engine. EnclaveDB currently does not support queries that operate over both public and secret data; guaranteeing security in the presence of such

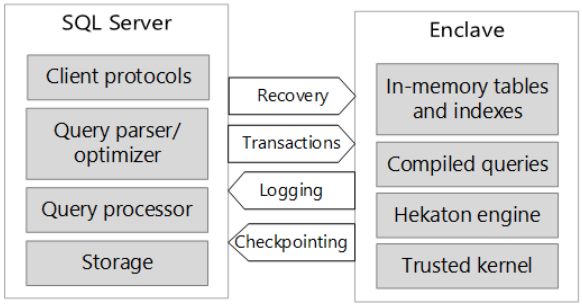

Fig. 2: Server-side components of EnclaveDB

queries is a challenging problem and left for future work. We now describe these components in detail.

A. Trusted kernel

Conventional databases rely on OS services such as threading, memory management and storage. However, this layering is not secure under a threat model where the OS may be compromised. One way of protecting applications is to use a library OS that runs within the enclave [14], [24]. However, this introduces a large amount of trusted code in the software stack. In EnclaveDB, we adopt the principle of least privilege – we introduce a thin layer called the trusted kernel that provides the Hekaton engine with the minimal set of services it requires. The trusted kernel implements some enclave-specific services such as management of enclave memory and enclave threads (described below), and delegates other services such as storage to the host operating system with additional logic for ensuring confidentiality and integrity. We now describe services that have implications for EnclaveDB.

Threading: SGX supports a mechanism for protecting the state of threads from the host when in enclave mode. Developers can reserve a part of enclave memory for an array of thread control structures (TCS). SGX uses a TCS to save and restore the host thread's context when the thread enters or exits the enclave. As required by SGX, the trusted kernel allocates a stack for each thread in enclave memory when a thread enters the enclave, and then transfers control to the application. This ensures that the host cannot observe or tamper with the thread's state.

From an application's perspective, this threading model is best viewed as a *pool of enclave threads*. When the host calls into the enclave, the trusted kernel effectively 'suspends' the host thread and switches to an unused enclave thread. When the call completes (or an exception occurs), the trusted kernel reclaims the enclave thread and execution resumes on the host thread. Therefore, the size of the thread pool, which is fixed on enclave creation, determines the maximum degree of concurrency available to the application.

Thread-local storage: The trusted kernel supports thread-local storage (TLS), which is used extensively by the Hekaton engine for efficient access to performance-critical data structures such as transaction read/write sets. However, the threading model described above leads to a subtle change in the semantics of TLS. The trusted kernel does not guarantee that TLS is preserved across multiple calls into the enclave from the same host thread. Preserving TLS across calls would require the kernel to trust a host assigned thread identifier, thereby introducing a new attack vector. This change in semantics did not mandate a code change in Hekaton since the engine already re-establishes TLS from the heap on every entry into the engine, except in the case of re-entrancy i.e. nested calls into the engine. As described in Section V, Hekaton components such as the log use re-entrancy (via callbacks) and assume that TLS is preserved on re-entrant calls. Since the trusted kernel does not guarantee these semantics, we modified the Hekaton engine to save TLS on the heap before enclave exits and restore TLS state on re-entrant calls.

```
Tx* TxAlloc()
bool TxExecute(Tx* tx, BYTE* name,
               BYTE** params, BYTE** ret)
bool TxPrepare(Tx* tx, TxPrepareCallback prepareCb)
void TxCommit(Tx* tx)
void TxAbort(Tx* tx)
```

Table 1: Hekaton's transaction processing API

B. Query compilation and loading

In a conventional database, the database server compiles, optimizes and executes queries. Therefore, the entire query processing pipeline is part of the attack surface. EnclaveDB reduces the attack surface by relying on client-side, native query compilation. The client packages all pre-compiled queries (expressed as stored procedures) along with the query engine and the trusted kernel, and deploys the package into an enclave. This design offers strong security because the queries are part of enclave measurement. However, the design also implies that any change in schema e.g., adding or removing queries, requires taking the database offline and redeploying the package. Online schema changes can be supported using a trusted loader; we leave this extension for future work.

C. Transaction processing

Hekaton uses a two-phase protocol for transaction processing (Table 1, see Figure 3 for the complete protocol). When the host receives a request to execute a stored procedure, it first creates a new transaction using `TxAlloc` and assigns a *logical* start timestamp to the transaction using a monotonically increasing counter stored in memory. It then executes the stored procedure in the context of the transaction by calling `TxExecute` along with the name of the stored procedure, buffers containing parameter values, and buffers for storing return values. `TxExecute` loads the natively compiled binary corresponding to the stored procedure and transfers control to a well-defined function within the binary. The binary calls into the Hekaton engine to perform operations on Hekaton tables and update transaction state (e.g., read and write sets, start timestamps).

After executing the stored procedure, the host prepares the transaction for committing by calling `TxPrepare`. The

prepare phase validates the transaction by checking for conflicts, assigns a logical end timestamp to the transaction, waits for transactions it depends on to commit, and logs the transaction's write set. Since logging involves expensive I/O operations, `TxPrepare` is asynchronous - it registers a callback and returns after initiating the log I/O. Once the I/O has completed, the host calls `TxCommit` to commit the transaction, which releases any resources associated with the transaction, unblocks all dependent transactions, and writes return values to be sent to the client. However, if the prepare phase fails (e.g., due to conflicts), the host calls `TxAbort`.

This protocol is vulnerable to a number of attacks from a malicious host even if the Hekaton engine is hosted in an enclave. The host can pass arbitrary transaction handles (`Tx*`), tamper with the incoming request (e.g., by changing parameter or return values), and invoke the protocol methods out of order. We make the following client-side modifications to ensure integrity of client-server interactions.

- We extend the query compiler to embed metadata such as the stored procedure name and the position and type of parameters in a dedicated section in the native binary. This metadata is used to validate requests.
- EnclaveDB clients connect to EnclaveDB by creating a secure channel with the enclave and establishing a shared session key $\mathcal{S}_{\mathcal{K}}$. During session creation, clients authenticate the enclave using a quote that contains the enclave's measurement. The enclave can authenticate clients using certificates or tokens issued by a trusted authority; our implementation uses certificates embedded in the EnclaveDB engine binary.
- EnclaveDB clients encrypt parameter values using $\mathcal{S}_{\mathcal{K}}$. Each parameter value is encrypted using authenticated encryption with the parameter position, type and a nonce as authentication data to prevent replay attacks.

We also make the following server-side modifications.

- The transaction processing APIs verify that transaction objects passed as a parameter are allocated in enclave memory, and procedure names, parameters and return values are buffers in untrusted memory.
- `TxExecute` authenticates all incoming requests. If authentication succeeds, EnclaveDB loads the native binary and checks if the procedure name, parameter positions and types are consistent with metadata stored in the binary. If validation succeeds, EnclaveDB decrypts parameters, allocates buffers in enclave memory for return values, and forwards the request to the Hekaton engine.
- After a stored procedure has executed, EnclaveDB encrypts return values (or error messages as described below) and writes them to buffers allocated by the host.
- The engine maintains additional state to ensure that the host does not attempt to commit a transaction if the prepare phase failed.

Observe that once a request has been validated, the stored procedure executes entirely within the enclave on tables hosted in the enclave. This prevents the host from tampering with query processing and reduces information leakage. We now discuss cases where sensitive information is generated during query processing.

Errors: Error conditions that occur during query processing can be classified as *secret dependent* and *secret independent*. Secret dependent errors directly or indirectly depend on sensitive values e.g. values stored in the tables or passed as parameters. This includes violations of database integrity constraints such as uniqueness, invalid cast/conversion, and invalid arithmetic operations. Clearly, revealing these errors to the host leaks information. However, this information is required by SQL Server since the occurrence of an error (or lack thereof) drives execution along different code paths e.g. the code path where no results are sent to the client. EnclaveDB addresses this problem by translating secret dependent errors into a generic error, and packaging the actual error code and message into a single message which is encrypted and delivered to the client. The client extracts the message and relays the error code to the application. In this process, the adversary learns that *some* error occurred during query processing but does not learn the cause of the error. This leakage can easily be prevented by always relaying an error code (`SUCCESS` if no error) and the result set (containing garbage on error) back to the client. However, this requires changes to the SQL client-server protocol and is outside the scope of this work.

Statistics: During execution, EnclaveDB collects a number of statistics that are useful for profiling and optimizing performance. For example, EnclaveDB collects frequency and execution time of each query, which can help identify slow running queries. The optimizer utilizes cardinalities of values in columns to determine efficient query plans. Some of these statistics are sensitive because they reveal properties of sensitive data. Therefore, EnclaveDB maintains all profiling information in enclave memory. EnclaveDB exposes an API to export this information (in encrypted form) and import it into a trusted client database, where it can be decrypted, analyzed, and used during native compilation.

D. Key management

EnclaveDB supports a much simpler model for key management compared to existing systems [1], [12] which requires users to associate and manage encryption keys for each column containing sensitive data. In EnclaveDB, sensitive columns are hosted in enclave memory, and data in these columns is encrypted and integrity protected by the memory encryption engine when it is evicted from the processor cache. Users only need to create and manage a single database encryption key $\mathcal{D}_{\mathcal{K}}$, which is used to encrypt all persistent database state. Users provision the key to a trusted key management service (KMS), along with a policy that specifies the enclave (identified using the enclave's measurement) that the key can be provisioned to. When an EnclaveDB instance starts or resumes from a failure, it remotely attests with the KMS and receives $\mathcal{D}_{\mathcal{K}}$.

Algorithm 1 Specification of the logging interface exposed by the host to Hekaton

```
1:  L ← ∅
2:  procedure LogAppend(tx, size, serializeCb, commitCb)
3:      buf ← alloc(size)
4:      serializeCb(tx, buf)
5:      L ← L ∪ buf
6:      startLogIO(tx, buf, commitCb)
7:  procedure LogTruncate(buf)
8:      L ← L \ {b ∈ L | b < buf}
9:  procedure GetLogIterator(start, end)
10:     return {b ∈ L | b ≥ start ∧ b ≤ end}
```

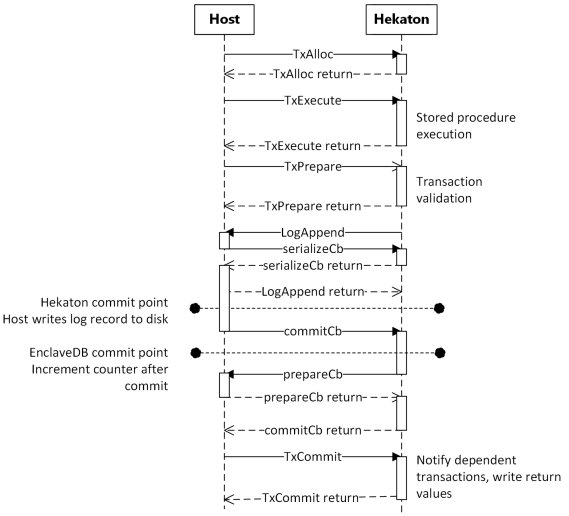

Fig. 3: Transaction commit protocol in EnclaveDB

V. Logging and recovery

The Hekaton engine makes in-memory tables durable by writing transactions on secret data to a persistent log managed by the host (i.e. SQL Server). Since the host cannot be trusted, EnclaveDB must ensure that a malicious host cannot observe or tamper with the contents of the log. In this section, we present a high-level specification of the logging and checkpointing APIs and then describe protocols to ensure integrity.

As shown in Algorithm 1, the log can be abstracted as a stream of bytes and a set L that contains indexes in the stream where individual log records begin. The log supports operations for appending log records, truncating the log, and iterating over the log. The append operation allocates space in the stream for the log record and returns an index *buf* in the stream, which also serves as the address of the buffer used for reading or writing the log record. The append operation invokes a callback $serializeCb$, which writes the log record in the allocated buffer, and schedules another callback $commitCb$, which is invoked when the log record has been flushed to disk. The truncation operation deallocates all buffers preceding the given index. The iterator returns the indexes of all log records between any two indexes $start$ and end.

The Hekaton engine uses the log as follows (Figure 3). After a transaction has been validated (in `TxPrepare`), the engine serializes the transaction's write set into a log record by calling $LogAppend$. Each log record includes the start and end timestamp of the transaction. The host writes the log record to disk and then invokes the commit callback. At this point, the transaction enters the commit state. During `TxCommit`, Hekaton unblocks dependent transactions and notifies the client submitting the transaction. Hekaton is designed to write multiple log records concurrently to avoid scaling bottlenecks with the tail of the log. This is possible because the serialization order of transactions is determined solely by end timestamps and not by the ordering in the log. Also note that failures can occur at any point e.g. after a log record has been flushed to disk but before the client is notified or dependent transactions are unblocked. We refer to transactions for which Hekaton *may* have unblocked dependent transactions or notified the client as *visible transactions*, and transactions whose dependent transactions remain blocked and no notifications have been generated as *invisible transactions*.

Algorithm 2 Hekaton operations for creating checkpoints and restoring the database after a failure

```
1:  sys ← ∅
2:  procedure CreateCheckpoint(start, end)
3:      {datafile, deltafile} ← SerializeLog(start, end)
4:      sys ← sys ∪ {datafile, deltafile}
5:      WriteFile(ROOT_FILE, sys)
6:      LogTruncate(end)
7:  procedure RestoreDatabase(start, end)
8:      sys ← ReadFile(ROOT_FILE)
9:      for {data, delta} ∈ sys do
10:         RestoreCheckpoint(data, delta)
11:     ReplayLog(start, end)
```

To avoid unbounded growth of the log, Hekaton periodically creates checkpoints and then truncates the log (Figure 2). Each checkpoint is a pair of append-only files, a data file and a delta file. The data file contains all records that have been inserted or updated since the last checkpoint and the delta file contains all deleted records. Checkpoints are created using an in-memory cache of log records, which is updated during commit. The names of data and delta files are saved in a special table called the *system table* (sys), which is persisted in a file called the *root file*. To avoid data loss, Hekaton truncates the log at a carefully selected index called the *truncation index*, which satisfies the invariant that all transactions committing after the truncation operation will be allocated indexes higher than the truncation index. During recovery (see $RestoreDatabase$ in Algorithm 2), Hekaton retrieves the truncation index from the database master file and the list of checkpoint file pairs from the root file. It restores tables and indexes from checkpoints, and replays the tail of the log from the truncation index.

A. Log Integrity

One way of ensuring integrity of the log and checkpoints is to use an encrypted file system [14]. An encrypted file system encrypts files with a key stored in enclave memory and checks integrity using a merkle tree [25]. However, maintaining a merkle tree for highly concurrent and write-intensive workloads such as a database log can be expensive.

A merkle tree introduces contention because the log is an append-only structure with a large number of threads writing close to the tail of the file. These threads will update roughly the same set of nodes in the merkle tree, and contend for locks protecting these nodes [26], [27]. The merkle tree also introduces contention for any monotonic counter(s) used to protect the tree against replay attacks. Finally, the size of a merkle tree is proportional to the size of the log, which can grow to several 100 GBs. If (a part of) the merkle tree is maintained in enclave memory, it reduces the amount of enclave memory available to the database. On the other hand, if the merkle tree is maintained on disk, a single log append can translate into multiple updates on disk.

In this paper, we propose a new and efficient protocol (Algorithm 3) for checking integrity and freshness of the log. The protocol is based on the following observations.

- Correctness of database recovery does not depend on the order of log records in the log. Instead, the ordering of transactions is determined by start and end timestamps embedded in the log records. Therefore, the log can be viewed as a *set* of log records instead of a raw file.
- Hekaton can ensure state continuity as long as all checkpoints, and log records of all visible transactions that have not been truncated are read during recovery.

Our protocol uses monotonic counters (Section V-C) to track sets of log records and identify log records that must exist in the log on recovery. We assume a monotonic counter service that exposes an API with functions $CreateCounter$, $GetCounter$, $IncCounter$, and $SetCounter$. $CreateCounter$ creates counters bound to the TCB of the enclave and the platform, $IncCounter$ atomically increments the counter and returns the previous value of the counter, and $SetCounter$ sets a counter to a given value if it is higher than the counter's current value or fails otherwise. The protocol uses three counters: W tracks log records that have been written to the log, V tracks log records generated by visible transactions, and R tracks truncated log records. To ensure that the protocol does not introduce new synchronization bottlenecks, we track these sets separately for each thread using per-thread monotonic counters. In other words, W, V, and R are k-dimensional vectors, where k is the number of enclave threads, which is fixed on enclave creation.

The counters are updated as follows. Each thread t processing a transaction increments the counter W_t after transaction validation but *before* sending the log record to the host (Line 17). All log records are encrypted with $\mathcal{D}_\mathcal{K}$ using authenticated encryption, with authentication data consisting of the thread identifier t, the counter value W_t and the log record's index in the log (Line 18); the authentication data is stored in the log record's header. This ensures that each log record can be uniquely identified by the pair of attributes (t, w) embedded in the log record. The counter V_t is incremented during the commit callback for a log record generated by thread t *before* unblocking dependent transactions and notifying the client (Line 25). At any point in time, the difference between pairs of counters W_t and V_t represents log records of transactions that are not yet visible.

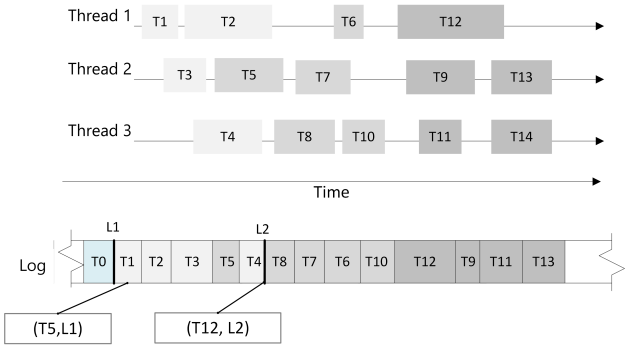

Fig. 4: Serialization points during transaction processing.

Tracking truncated log records is more challenging because at any given point, there are multiple log records being written to the log at indexes assigned by the host, and a malicious host can mount attacks by assigning indexes arbitrarily e.g. in portions of the log that have already been truncated. In order to detect malicious behavior, we establish a *contract* between EnclaveDB and the host based on the notion of *serialization points*. A serialization point is defined as follows.

Definition 5.1: Let t be a transaction and l be an index in the log. Let $log(t)$ represent the first index in the log where a log record for transaction t has been written. A serialization point is a pair (t, l) such that all transactions committing after t should be written to the log after the index l. Formally, let $\leq$ represent the *happens before* relationship between transactions.

$$serialization\_point(t, l) \Rightarrow \forall t' \mid t \leq t', log(t') > l$$

It also follows that given a serialization point (t, l), the log can be safely truncated at l once all transactions that have committed before t have been written to a checkpoint. From the perspective of integrity checking, serialization points have the following implications.

- Given a serialization point (t, l), it follows that once transaction t commits, a correct log implementation never returns an index l' such that $l' \leq l$ in any subsequent calls to $LogAppend$. A violation of this property indicates an attack.
- Once a serialization point (t, l) is established, we can safely compute the expected number of log records that have written before the index l because no more log records should be written at any index $l' \leq l$.

There are many ways of establishing serialization points. For example, Hekaton establishes serialization points by grouping transactions based on end timestamps and *waiting* for all transactions in group g_{n-1} to commit (and hence be written to the log) before committing any transactions in g_{n+1}. Figure 4 illustrates this mechanism using a sample execution. Transactions are color-coded according to groups. $(T5, L1)$ is a valid serialization point because all transactions committing after $T5$ are written to the log after $L1$. Our protocol is inde-

Algorithm 3 Protocol for checking integrity of the log

```
1:  Monotonic counters
2:  ∀t ∈ Threads, W_t ← 0, V_t ← 0, R_t ← 0
3:  E ← 0
4:  Volatile enclave state
5:  ∀t ∈ Threads, l_t ← ∅, s_t^0 ← 0
6:  b_0 ← 0, p ← 1
7:
8:  procedure EnclaveLogAppend(tx, size)
9:      size ← size + HEADER_SZ
10:     LogAppend(tx, size, enclaveSerializeCb, enclaveCommitCb)
11:
12: procedure enclaveSerializeCb(tx, buf, size)
13:     t ← GetCurrentThreadId()
14:     assert(buf > b_p)
15:     tmp ← malloc(size − HEADER_SZ)
16:     serializeCb(tx, tmp)
17:     w ← IncCounter(W_t)
18:     buf ← Enc[D_K]{GetCounter(E), t, w}(tmp)
19:     l_t ← l_t ∪ buf
20:     free(tmp)
21:
22: procedure enclaveCommitCb(tx, enc)
23:     {e, t, c}, buf ← Dec[D_K](enc)
24:     assert(e == GetCounter(E) ∧ c == GetCounter(V_t))
25:     IncCounter(V_t)
26:     commitCb(tx, buf)
27:
28: procedure OnSerializationPoint(buf)
29:     ∀t, s_t^p ← s_t^{p−1} + | {b ∈ l_t | b < buf} |
30:     ∀t, l_t ← l_t \ {b ∈ l_t | b < buf}
31:     b_p ← buf
32:     p ← p + 1
33:
Require: (∃ i | i ≤ p ∧ b_i == buf)
34: procedure EnclaveLogTruncate(buf)
35:     j ← (i | i ≤ p ∧ b_i == buf)
36:     ∀t, SetCounter(R_t, s_t^j)
37:     LogTruncate(buf)
38:
39: procedure ReplayLog(start, end)
40:     ∀t, c_t ← GetCounter(R_t)
41:     ∀t, s_t^p ← GetCounter(R_t)
42:     I = GetLogIterator(start, end)
43:     for enc ∈ I do
44:         {e, t, c}, buf ← Dec[D_K](enc)
45:         assert(e == GetCounter(E))
46:         if GetCounter(R_t) ≤ c ≤ GetCounter(V_t) then
47:             assert (c == c_t + 1)
48:             c_t ← c_t + 1
49:             l_t ← l_t ∪ buf
50:             ApplyLogRecord(buf)
51:     ∀t, assert(c_t == GetCounter(V_t))
52:     if ∃t | c_t ≠ GetCounter(W_t) then
53:         OnSerializationPoint(end)
54:         CreateCheckpoint(start, end)
55:         IncCounter(E)
56:     ∀t, SetCounter(V_t, GetCounter(W_t))
```

pendent of the way and the frequency with which serialization points are established; it only requires that the client of the log (i.e. Hekaton) periodically establish serialization points, and notify the protocol when a serialization point is established.

Our protocol uses serialization points for tracking truncated log records as follows. We maintain (in volatile enclave memory) the sequence of serialization points b, a per-thread list of indexes l_t at which thread t has written log records since the most recent serialization point, and a sequence s of sets, one for each serialization point, where each element s_t^p is a number of log records written by thread t before the serialization point p. When a new log record is created, we add its index to the list l_t after checking that the host does not violate the serialization point contract (line 14). We introduce a new operation $OnSerializationPoint$ which is invoked by Hekaton when it establishes a serialization point. For each thread t, this operation computes a summary s_t^p and removes all indexes preceding the serialization point from l_t.

We also modified Hekaton to invoke $EnclaveLogTruncate$ after creating a checkpoint, passing in a truncation index, which must be a previously declared serialization point. This operation updates the vector clock R (line 36) to reflect the set of truncated log records using a previously computed summary before calling out to truncate the log.

During recovery ($ReplayLog$), EnclaveDB reads the counters R and scans the tail of the log. While scanning, we check that log records have not been tampered with and that the log records generated by each thread appear in order of their counter value with no gaps (Line 47). After scanning the tail, we check if *all* visible log records (tracked by V) have been read (Line 51), and report a freshness violation otherwise.

Note that the recovery protocol excludes log records generated by invisible transactions i.e. log records with counter values greater than V_t. Excluding these log records is safe because unlike Hekaton, the increment of the counter V_t (line 25) is the commit point in EnclaveDB - clients and dependent transactions are notified only after the increment. However, simply excluding these log records allows the adversary to mount a 'replay' attack by withholding log records belonging to invisible transactions and adding them to the log in a later execution, thereby creating non-linearizable executions. We prevent these attacks by invalidating all such log records using an additional monotonic *epoch counter* E. This counter is included in each log record's authentication data, and incremented (line 55) when log records belonging to invisible transactions are detected in the log (line 52). The increment, coupled with the additional check that all log records read during recovery must belong to the current epoch (line 45) invalidates all invisible log records that the adversary may have withheld. We create a checkpoint and truncate the whole log before incrementing the epoch counter to ensure that visible transactions are not lost. We also update the counters V, setting them equal to W before resuming execution.

B. Checkpoint Integrity

As discussed, EnclaveDB periodically truncates the log after creating checkpoints. To avoid data loss, EnclaveDB must ensure that before log records are truncated from the log, they have been included in checkpoint files that are guaranteed to be read during recovery. Furthermore, EnclaveDB must also ensure that any tampering with the checkpoint files is detected.

EnclaveDB achieves these properties as follows. First, EnclaveDB maintains a cryptographic hash for each data and delta file. The hash is updated as blocks are added to the

Algorithm 4 Protocol for checking integrity and freshness of checkpoints.

```
1:  S ← 0 //Monotonic counter
2:  procedure EnclaveWriteFile(name, data)
3:      WriteFile(name, Enc[D_K]{GetCounter(S) + 1}(data))
4:      IncCounter(S)
5:
6:  procedure EnclaveReadFile(name)
7:      enc ← ReadFile(name, GetCounter(S))
8:      s, data ← Dec[D_K](enc)
9:      assert(s == GetCounter(S))
10:     WriteFile(name, Enc[D_K]{GetCounter(S) + 1}(data))
11:     IncCounter(S)
12:     WriteFile(name, Enc[D_K]{GetCounter(S) + 1}(data))
13:     IncCounter(S)
14:     return data
```

file. Once all writes to a checkpoint file have completed, the hash is saved in the system table along with the file name. EnclaveDB checks the integrity of all checkpoint file pairs read during recovery by comparing the hash of their contents with the hash in the root file.

Next, EnclaveDB uses a state-continuity protocol based on Ariadne [28] to save and restore the system table within the root file while guaranteeing integrity, freshness and liveness. The protocol (shown in Algorithm 4) uses a monotonic counter S to track versions of the root file. The protocol binds the contents of each file with the counter value (by adding the counter value to the file and generating a keyed MAC). Then the file is written to disk and the counter is incremented. The protocol allows the adversary to obtain a file with a counter value one more than the current counter (by introducing a failure after the file has been written but before the increment). However, all such versions are invalidated by writing two versions of the current root file (i.e. with enclosed counter value matching S) with counter values $S+1$ and $S+2$ before using the root file to reconstruct the system table. Refer to [28] for a proof of correctness.

C. Monotonic Counters

The protocol described above relies on monotonic counters to ensure state continuity. There are many ways of implementing monotonic counters. For instance, SGX uses wear-limited NVRAM available in the management engine [9]. Our experiments confirmed prior results [28] which show that accessing these counters is slow (~100ms per counter update), and not sufficient for the latency and throughput requirements of EnclaveDB. In EnclaveDB, we use a dedicated monotonic counter service implemented using replicated enclaves [29]. The service stores counters in enclaves replicated across different fault domains and uses a consensus protocol to order operations on counters. This approach is more flexible and efficient than SGX counters, and can tolerate failures as long as a quorum of replicas is available.

D. Forking attacks

The protocol described above ensures that any database enclave recovers to the latest state. However, it permits the adversary to launch forking attacks by creating multiple enclaves with the same package on one or more servers, and directing different clients to different enclaves. EnclaveDB prevents forking attacks by ensuring that at any point in time, only one enclave (and therefore one database instance) is 'active'. On creation, each EnclaveDB enclave generates a 128-bit GUID. This GUID is encrypted using the public key of the KMS and included in the enclave's quote. The KMS maintains a mapping from database instances to GUIDs and a *black list* of all GUIDs it has previously received in quotes. When it receives a quote containing a new GUID, it adds the current GUID to the blacklist and updates the GUID associated with the database instance. Each database enclave also includes its GUID (encrypted using the session key) in all communication with clients. EnclaveDB clients verify that they are establishing a session with or receiving a response from the most recent incarnation of the database by validating the response with the KMS, and retrying if validation fails.

E. Proof Sketch

In this section, we present an informal proof that the logging protocol described above guarantees integrity, continuity and liveness. Integrity and continuity are critical for establishing that EnclaveDB guarantees linearizability (proof beyond the scope of this paper), and liveness ensures that EnclaveDB makes progress in the absence of an attacker.

Claim 5.1: **Checkpoint continuity.** *Once EnclWriteFile completes writing a root file, all log records included in checkpoint file pairs referenced in the root file are guaranteed to be read during any subsequent recovery.*
This follows from the freshness guarantees of Ariadne's protocol which prevents replay attacks, and integrity checks on the root file and checkpoint file pairs.

Claim 5.2: **Continuity.** *Log records generated by visible transactions are either contained in the log or included in a checkpoint file pair contained in the root file.*
Consider a visible transaction T which generates log record l. Let c be the counter value embedded in the log record, t be the identifier of the enclave thread generating the log record, and e be the epoch in which the log record is generated.

First, observe that each log record is uniquely identified by the pair (t, c). This follows from the fact that c is obtained from the tamper-proof, monotonic counter W_t, which is atomically incremented every time a log record is generated.

Now consider any recovery that occurs following a failure after the transaction T became visible. It follows that $c < V_t$. There are two possible cases we encounter during recovery.

- $R_t \leq c < V_t$. In this case, the log record l must be read from the log since the recovery protocol reads all records from R_t to V_t for each thread t. If the log record is missing or has been duplicated, either the assert at line 47 or 51 fails. If the log record has been tampered with, authentication fails. In either case, the database fails to recover.

- $c < R_t$. In this case, we show that the log record must belong to a checkpoint included in the root file. Observe that the counters R are updated *after* a checkpoint has been created and *before* the log is truncated. This implies that the counters always under-approximate the set of log records that have been included in checkpoints. Therefore, if $c < R_t$, then the log record must have been included in a checkpoint. This, in conjunction with checkpoint continuity (Claim 5.1) ensures that either the log record is included in a checkpoint read during recovery, or the database fails to recover.

Claim 5.3: **Integrity.** *Invisible transactions have no effect on database state.*

We show that log records generated by invisible transactions are ignored while reconstructing database state during recovery. Consider a log record l generated by an invisible transaction T. Let c and e be the counter and epoch values associated with l, and t be the enclave thread that generated the log record. By definition, the commit callback was not invoked for l. Since the counter V_t is incremented in the commit callback, it must be true that $W_t \geq c$ and $V_t < c$ until a failure occurs. We consider two cases.

- If no failure occurs, then clearly the log record l is never used during recovery. Furthermore, since the commit callback is never invoked for l, any transactions dependent on T continue to remain blocked, and the client issuing T is not sent a signed notification.
- If a failure occurs, then in the next recovery step, l is not used to reconstruct state since $c > V_t$. Furthermore, since $V_t < W_t$, EnclaveDB creates a checkpoint that does not include l and increments the epoch counter before updating V_t and resuming transaction processing. Incrementing the epoch counter invalidates l.

Claim 5.4: **Liveness.** *EnclaveDB does not introduce new states where execution terminates in the absence of an active attacker.*

We show that none of the assertions introduced by EnclaveDB fail in the absence of an active attacker.

- *Authentication failures.* If requests, responses, the log and checkpoints are not tampered with, and the database is not forked, none of the authentication checks fail.
- *Thread-level commit ordering failure* (line 47). Each thread in EnclaveDB generates a log record with counter value c only after a previous log record with counter value $c-1$ has been saved to storage. Therefore, in the absence of an active attacker, the recovery protocol should receive log records in a sequence that respects this ordering.
- *Mismatched epoch* (line 45). We prove this using induction. It is easy to see that this assertion cannot fail with $E = 0$. Assume this assertion does not fail during recovery when $E = i$. Consider any subsequent recovery which increments the epoch counter. In the absence of an active attacker, EnclaveDB initiates recovery with $start$ and end indexes equal to the most recent truncation index and the tail of the log. This ensures that the entire log is truncated before the epoch counter is incremented. Since new log records are generated only after the counter is incremented and recovery completes, all subsequent log records will be generated with epoch counter $i+1$. Therefore, there will be no epoch mismatch when $E = i + 1$.

VI. Optimizations

In EnclaveDB, the use of enclaves introduces several sources of overheads. These include the cost of context switching (saving and restoring thread context and invalidating hardware TLB), memory encryption and integrity checking, encrypting and decrypting data, and copying data in and out of enclaves. Compared to prior work that uses trusted hardware for evaluating individual expressions of a query in trusted hardware [12], [13], EnclaveDB has a much smaller number of context switches because the entire transaction is evaluated in enclave mode. Furthermore, the number of context switches is fixed and independent of the amount of data being processed. Also, EnclaveDB incurs the cost of software encryption and decryption only at transaction boundaries (parameters and return values), and for log records and checkpoints, which is significantly more efficient than encrypting and decrypting individual values. When compared to designs such as Haven [14], EnclaveDB achieves better performance because of more efficient protocols for ensuring integrity and freshness. We implemented a number of optimizations to further improve performance.

- We refactored the Hekaton engine to move state and logic that does not depend on secrets to the host. This includes state for tracking whether log IO for a transaction has been completed.
- SQL Server uses a co-operative thread scheduler where all threads are expected to periodically yield control. In EnclaveDB, every yield results in a context switch. We modified the Hekaton engine to reduce the frequency with which threads yield (by 50%).
- We use *prefetching* to reduce the number of context switches. Prefetching involves speculatively calling an enclave API as part of a previous enclave invocation, caching its results on the host and returning the cached result when the API is subsequently called. This optimization only applies to side-effect free APIs.
- We cache values of per-thread monotonic counters in thread-local state. Therefore, we incur the cost of context switches only on writes.

These optimizations resulted in a significant reduction in the number of context switches (from an average of 110 to 10 context switches per transaction). This includes 5 calls into the enclave and an additional callback to write the log record if the transaction performs any writes, and 4 calls out of enclave mode, which includes 2 monotonic counter updates.

VII. Multi-party SQL

In this section, we show how EnclaveDB can be extended to support a scenario where *mutually untrusting users* can host a

shared database while guaranteeing strong security properties. For simplicity, we consider a fixed set of mutually untrusting users $\mathcal{U} = \{U_1, \ldots U_k\}$ who share a database hosted in an untrusted environment. Each user U_i is associated with a public-private key pair $({\mathcal{P}_\mathcal{K}}^i, {\mathcal{S}_\mathcal{K}}^i)$. A subset of these users may collude with the database administrator. For this paper, we assume that the table definitions and stored procedures are pre-defined and known to all users. We would like to guarantee the following properties (in addition to protection from the administrator).

- **Access control.** Only authorized users can execute stored procedures.
- **Integrity.** Authorized users can only execute one among the set of pre-defined stored procedures.
- **Confidentiality.** Authorized users learn no information about the state of the database apart from the results of stored procedures they execute.

We achieve these properties as follows. Each user U_i creates her own copy of the enclave package by compiling all table definitions and stored procedures in her own trusted environment. Additionally, U_i embeds the public keys of all authorized users in a well-known section of the trusted kernel binary. The DBA (who is not trusted) repeats this process and deploys his version of the package on an untrusted server. On enclave creation, EnclaveDB generates a new $\mathcal{D}_\mathcal{K}$ instead of retrieving the key from the KMS, and seals $\mathcal{D}_\mathcal{K}$ to the platform for future recovery. This key is used to encrypt log records and checkpoints and shared with all users using a secret sharing algorithm [30].

Once the database is initialized, any user U_i can use remote attestation to check that the database enclave has been correctly initialized by comparing the enclave's measurement with the measurement of her package, and initiate creation of a secure channel. The enclave authenticates requests for creating a secure channel using public keys embedded in the trusted kernel binary. Once a secure channel is established, the user can send requests to execute any of the pre-compiled stored procedures. EnclaveDB ensures that users learn nothing more than the response of transactions they execute.

VIII. Evaluation

We have developed and tested EnclaveDB on Intel SGX. Our implementation uses an in-house SGX SDK and Intel SGX PSW v1.1.28151. Since the current generation of Intel Skylake CPUs restricts the EPC to 128MB, we can only deploy small databases using SGX hardware. To evaluate performance for realistic database sizes, we use a performance model that simulates large enclaves and accounts for the main sources of overheads. In this section, we describe the performance model and present results from an evaluation using two standard database benchmarks.

Performance model: To model SGX performance with larger enclaves, we assume that code in SGX will have the same performance as current CPUs except for (1) the additional cost of enclave transitions and (2) the additional cost incurred by last level cache (LLC) misses due to memory encryption and integrity checking while accessing the EPC. We model enclave transitions by introducing a system call to change protection of a pre-allocated page. This call flushes the TLB and adds a delay of ~10000 cycles, which is approximately the cost of a transition measured on SGX hardware.

For modeling the cost of memory encryption, we considered the option of artificially reducing DRAM frequency and hence available memory bandwidth (similar to [14]). However, reducing frequency affects both enclave and non-enclave code, and does not accurately reflect the slowdown caused by memory encryption. Instead, we model the cost of memory encryption by using binary instrumentation to 'penalize' all memory accesses generated by code within the enclave. Our instrumentation tool injects a fixed delay before every memory access within the enclave (excluding accesses to the stack since they are likely to hit in the caches). The amount of delay is obtained using a process of calibration on current SGX hardware. We first measure the overhead of running a set of micro-benchmarks using SGX enclaves. The delay is the lowest number of cycles such that running the same benchmarks after injecting the delay before every access on the same hardware *without enclaves* results in overhead higher than the overhead of running the application within the enclave. Our microbenchmarks consists of a simple key-value store [31] and a set of machine learning applications [32].

Using this calibration process, we find that our performance model with a delay of 10 cycles always over-approximate overheads of current SGX hardware. For example, SGX incurs an overhead of 37% for the key-value store whereas our performance model estimates the overhead to be 42%. On processors of older generations, we introduce this delay by injecting a single `pause` instruction. On Skylake processors and CPUs from newer generations, a pause has much higher latency ($\sim$140 cycles). We therefore inject a sequence of 20 NOPs instead, which delays the access by 10 cycles. This delay is implementation dependent so that the model should be re-calibrated if the implementation changes.

Benchmarks and Setup: We evaluate the performance of EnclaveDB using two standard database benchmarks, TPC-C [33] and TATP [34]. The TPC-C benchmark represents the activity of a wholesale supplier which manages and sells products. We use a database with 256 warehouses. This database has an in-memory size of 32GB. The workload consists of a client driver running on two machines simulating 64 concurrent users each; each user executes five stored procedures in accordance with the TPC-C specification [33]. The in-memory size grows by ~6GB/min during execution of this workload. The TATP benchmark simulates a typical location register database used by mobile carriers to store information about valid subscribers, including their mobile phone number and their current location. The database consists of 4 tables and 7 stored procedures. We create a database with 10 million subscribers. The client driver simulates 100 active subscribers querying and updating the database.

We optimized these benchmarks for use with an in-memory database by creating appropriately sized indexes to optimize

query performance. The workload we generate drives CPU utilization close to 100% in the baseline configuration while leaving just enough slack for the checkpointing process to keep up with the rate of transaction commits. We run each benchmark 5 times for 20 minutes each and measure performance every minute. We performed the experiments on Intel Xeon E7 servers running at 2.1 Ghz. The servers have 4 sockets with 8 cores each (hyper-threading disabled). Each CPU has an integrated memory controller with 4 memory channels attached to 8 32GB DDR4 DIMMs (2 DIMMs per channel), with a total capacity of 512GB. The storage subsystem consists of 8 256GB SSDs and is used for storing both the log and checkpoints. The servers run Windows Server 2016.

We evaluated each benchmark in four configurations. BASE is the configuration with Hekaton running outside enclaves. CRYPT is a configuration with EnclaveDB running in simulated enclave mode. The model emulates enclaves within the application's address space by allocating a region of virtual memory and loading all enclave binaries in that part of the address space. In this configuration, all software security features such as log/checkpoint encryption and integrity checking are enabled. CRYPT-CALL is a configuration that adds the cost of context switching to CRYPT. Finally, CRYPT-CALL-MEM adds the cost of memory encryption to CRYPT-CALL. In all configurations except BASE, we simulate enclaves of size 192GB. We configured the trusted kernel to use 128 threads; for each thread, we allocate a stack of size 64K. For both benchmarks, we consider all tables and stored procedures as sensitive and host them in enclaves.

Trusted computing base: We measured the size of the TCB for both these benchmarks. The Hekaton engine is 300K LOC and the trusted kernel is 25K LOC. The queries and table definitions are 41K and 18K lines of auto-generated code in TPC-C and TATP respectively. In comparison, the SQL Server OLTP engine is 10M LOC and Windows is >100M LOC. Thus, the TCB of EnclaveDB is over two orders of magnitude smaller than a conventional database and over an order of magnitude smaller than systems such as Haven [14] whose library OS has >5M LOC [35]. The main components that contribute to the TCB in EnclaveDB are checkpointing and recovery (~100K LOC) and the transaction manager (~50K LOC). Reducing the TCB further, either by refactoring the engine, or using verification remains an open problem.

Context switches: The number of context switches is an important indicator of overheads for applications using enclaves. We measured the number of context switches per transaction for both benchmarks. On average, TPC-C incurs 5 context switches into the enclave (4 for the commit protocol, and 1 call to serialize the log record). TPC-C also incurs 3 call outs per transaction on average, a call out to notify the host of the outcome of `TxPrepare`, a call out to create a log record, and a number of other less frequent call outs that occur periodically. TATP has a similar profile, with the difference that context switches due to logging are less frequent since 80% of the transactions are read-only. In either case, the number of context switches is independent of the amount of

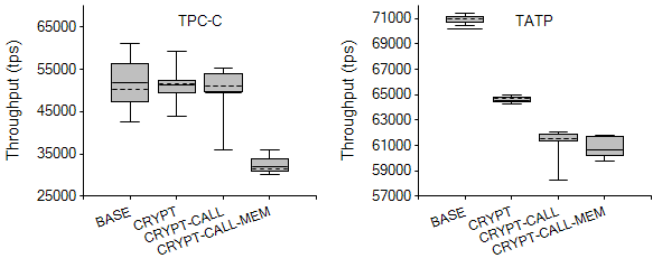

Fig. 5: TPC-C and TATP throughput for different configurations. The plot shows the maximum, 95th percentile, mean, median and minimum throughput values over a period of 20 minutes across 5 executions.

state in the enclave and the transaction logic (except in the case of read-only transactions).

TPC-C: Figure 5 shows the variation in TPC-C throughput for different configurations. In the baseline configuration, EnclaveDB achieves a mean throughput of 52,000 tps (which translates to 1.35 million new order transactions per minute or tpmC), with a peak of over 1.6 million tpmC. The throughput in both CRYPT and CRYPT-CALL configurations is statistically similar to the baseline. This suggests that the additional overheads of running the database with the trusted kernel, switching thread contexts, copying data in and out of the enclave, and encryption and integrity checking are negligible. We attribute this to our design which minimizes the number of context switches, amortizes the cost of encryption/decryption, and the efficient protocol for checking integrity of the log.

We also find that the variability in throughput is lower in the CRYPT configuration compared to baseline. This is due to two aspects of our design. First, virtual memory for the enclave is allocated at enclave creation and never returned to the host operating system. In contrast, the in-memory engine running outside the enclave periodically returns unused memory back to the SQL buffer pool, and must re-allocate memory when required. Secondly, the enclave thread scheduler yields control to the host less often compared to the baseline scheduler to reduce switching costs.

Finally, we observe that the CRYPT-CALL-MEM configuration, which models the cost of memory encryption, has a mean throughput of 31,000 tps, a drop of ~40% compared to baseline. Even with these overheads, throughput is over two orders of magnitude higher than prior work [14], which achieved a throughput of ~80 tpsE for the TPC-E benchmark on a 4-core machine using a similar performance model.

To further understand these overheads, we compared the configurations CRYPT and CRYPT-CALL-MEM with baseline across a number of other performance metrics. Figure 6 shows the comparison over a 20-minute window with samples collected every 10 seconds and averaged every minute. All configurations have high CPU utilization (over 90%) on average, which suggests that CPU remains the main bottleneck. The periodic changes in CPU, memory, and disk bandwidth utilization in BASE (and to a lesser extent in CRYPT) are caused by

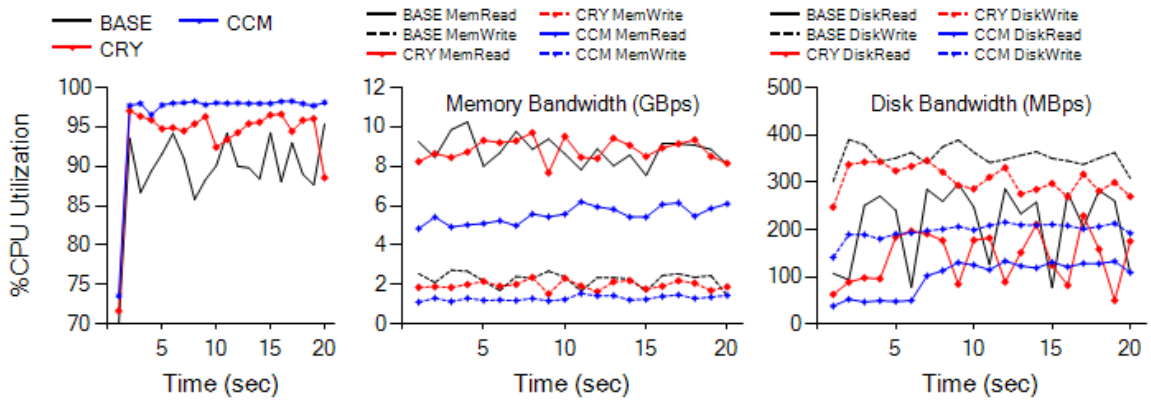

Fig. 6: Profiles comparing CPU, memory bandwidth and disk bandwidth utilization for the TPC-C benchmark. BASE (unmarked lines), CRY (red lines marked with diamonds) and CCM (blue lines marked with stars) represent the baseline, CRYPT and CRYPT-CALL-MEM configurations.

checkpointing. The similarity of memory and disk utilization between BASE and CRYPT also confirms that the the integrity checking protocol does not introduce new bottlenecks. We also observe that memory bandwidth utilization is high (reaching 10GBps), and memory reads dominate writes, whereas disk writes dominate reads. This is expected since most of the transaction processing occurs in-memory, whereas disk traffic is dominated by writes to the log. Finally, we observe that both memory and disk bandwidth utilization in CRYPT-CALL-MEM configuration are significantly lower than baseline and CRYPT, caused by pause instructions which model memory encryption. Based on these observations, we conclude that if the next generation SGX hardware can deliver an effective memory read/write bandwidth of ~6GBps and 2GBps respectively with large enclaves, we can expect overheads of ~40%.

TATP: We deployed a scaled down version of the TATP benchmark (with 1000 subscribers) in EnclaveDB using SGX hardware. We were only able to run this benchmark for 40,000 transactions (4 client threads issuing 10,000 transactions each) before running out of enclave memory. For this scaled down workload, Hekaton achieves a peak throughput of 7,900 tps whereas EnclaveDB achieved a peak throughput of 7,700 tps, an overhead of 2.5%. This is however, not a representative workload because of the small size of the database and short duration of the workload.

Figure 5 shows the throughput for the full TATP workload. EnclaveDB achieves a higher throughput of 71,000 tps with low variability compared to TPC-C because it is a predominantly read only workload (with 80% read transactions). The mean throughput reduces to ~65,000 tps after switching to the CRYPT configuration. Accounting for the costs of context switching and memory encryption reduces the throughput further; we observe a mean throughput of 60,500 tps in the CRYPT-CALL-MEM configuration, an overhead of 15% relative to baseline.

As shown in Figure 7, CPU remains the main bottleneck in the CRYPT-CALL-MEM configuration. Also, much like TPC-C, memory reads dominate writes and disk writes dominate reads. However, both memory and disk bandwidth utilization are lower compared to TPC-C, reflecting the predominantly read-only nature of the workload. The memory read bandwidth for the CRYPT-CALL-MEM configuration is higher than baseline. This is because the additional memory traffic generated by EnclaveDB (due to context switching, encryption/decryption and integrity checking) dominates the reduction in utilization caused by `pause` instructions. We also measured the end-to-end latency for this benchmark. We find an increase in average latency by 10%, 18% and 22% for CRYPT, CRYPT-CALL, and CRYPT-CALL-MEM configurations respectively over BASE.

Based on these experiments, we can conclude that EnclaveDB achieves a very desirable combination of strong security (confidentiality and integrity) and high performance, a combination we believe should be acceptable to most users.

IX. Related Work

Existing research on secure databases falls into two broad categories, one based on homomorphic encryption, and the other using trusted hardware.

Homomorphic encryption: Homomorphic encryption [36] refers to encryption schemes that permit operations on encrypted values. While fully homomorphic encryption is not practical yet [37], a number of partially homomorphic encryption schemes have been proposed [38], [39], [40] that permit specific operations on encrypted data, potentially at the cost of weaker security. Systems such as CryptDB [1], Monomi [2], and Seabed [3] use these schemes to provide secure query processing while protecting the confidentiality of data. However, the types of queries supported by these systems are limited by the availability of corresponding encryption schemes or need to be augmented, e.g. by offloading parts of the query execution to clients [41], [2]. Arx [42] introduces two new types of database indices for range and equality queries based on garbled circuits [43]. With these, it can support a similar set of queries as previous systems while using semantically-secure encryption. In

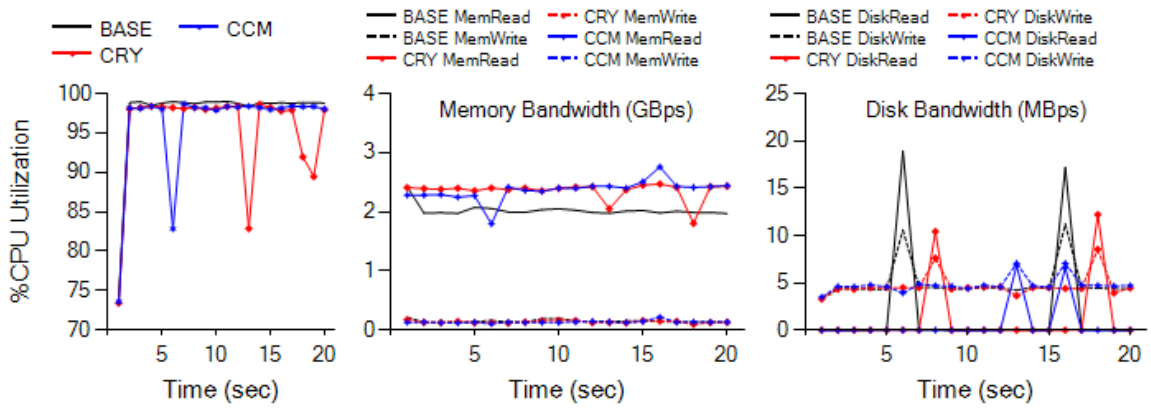

Fig. 7: Profiles comparing CPU, memory bandwidth and disk bandwidth utilization for the TATP benchmark.

contrast, EnclaveDB supports a broader set of queries, including arbitrary arithmetic, string manipulation, grouping and sorting, and uses strong probabilistic encryption. In addition, EnclaveDB provides not only confidentiality but also integrity and freshness guarantees for both stored data and query results.

Trusted hardware: Several database designs have been proposed that incorporate secure coprocessors [44], [45], [46] to securely process sensitive data. The data is stored in encrypted form on the host system and is only decrypted on the coprocessor as part of the query execution. However, currently available coprocessors are limited in terms of processing speed, storage, and bandwidth. TrustedDB [13] and Cipherbase [12] outsource computations on sensitive data to secure co-processors and FPGAs. These approaches require additional hardware and focus only on confidentiality, but do not guarantee the integrity or freshness of computation. They also suffer from high overheads due to the cost of data transfer over PCIe.

Haven [14] and Graphene [24] use Intel SGX for isolation and run unmodified applications by bundling them with an in-enclave library OS. Haven can run an unmodified version of SQL server but its library OS alone has over 5M LOC [35]. To avoid the large TCB overhead of a whole library OS, SCONE [47] places the C standard library inside the enclave and delegates all system calls performed by the application to the untrusted host. Compared to EnclaveDB, it still runs the full application inside the enclave while EnclaveDB further minimises the TCB by running large parts of the database server outside of the enclave. Panoply [48] allows applications to be split into multiple compartments and to be run across multiple enclaves following the principle of least privilege. Similarly, Glamdring [49] semi-automatically partitions applications to only run security-sensitive code within enclaves. However, these approaches are not easily applicable to complex applications such as databases.

X. Conclusions

In this paper, we proposed EnclaveDB, a database that uses trusted execution environments such as SGX enclaves to guarantee confidentiality and integrity with low overhead. EnclaveDB makes a careful set of design choices that reduce the TCB to a small set of security critical components such as the query engine and the transaction manager, and removes trust from the DBA. EnclaveDB also supports a multi-party mode where multiple, mutually distrusting users host sensitive data and execute queries in a shared database instance. A key component of EnclaveDB is an efficient protocol for ensuring the integrity and freshness of the database log. There are many ways EnclaveDB can be improved, such as support for online schema changes, dynamically changing the set of authorized users, and further reducing the TCB. But we believe that EnclaveDB lays a strong foundation for the next generation of secure databases.

References

[1] R. A. Popa, C. M. S. 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*, ser. SOSP '11, 2011, pp. 85–100.

[2] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," in *Proceedings of the 39th international conference on Very Large Data Bases*, ser. PVLDB'13. VLDB Endowment, 2013, pp. 289–300.

[3] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, and A. Modi, "Big Data Analytics over Encrypted Datasets with Seabed," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16, 2016.

[4] Microsoft. (2016) Always Encrypted Database Engine. [Online]. Available: https://msdn.microsoft.com/en-us/library/mt163865.aspx

[5] Google. (2017) Encrypted BigQuery client. [Online]. Available: https://github.com/google/encrypted-bigquery-client

[6] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart, "Leakage-Abuse Attacks against Order-Revealing Encryption," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, ser. SP '17. San Jose, California, USA: IEEE Computer Society, 2017.

[7] F. B. Durak, T. M. DuBuisson, and D. Cash, "What else is revealed by Order-Revealing Encryption?" in *ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria: ACM, 2016.

[8] M. Naveed, S. Kamara, and C. V. Wright, "Inference attacks on Property-Preserving Encrypted Databases," in *ACM Conference on Computer and Communications Security (CCS)*. Denver, CO, USA: ACM, 2015.

[9] 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, ser. HASP '13, 2013.

[10] W. Zheng, A. Dave, J. Beekman, R. A. Popa, J. Gonzalez, and I. Stoica, "Opaque: A Data Analytics Platform with Strong Security," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017.

[11] 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*. IEEE, 2015, pp. 38–54.

[12] 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*, ser. SIGMOD '13, 2013, pp. 1033–1036.

[13] S. Bajaj and R. Sion, "TrustedDB: A Trusted Hardware Based Database with Privacy and Data Confidentiality," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 205–216.

[14] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 8:1–8:26, Aug. 2015.

[15] S. Checkoway and H. Shacham, "Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13, 2013.

[16] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: SQL Server's memory-optimized OLTP engine," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 1243–1254.

[17] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing Page Faults from Telling Your Secrets," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 317–328.

[18] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 640–656.

[19] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17, 2017.

[20] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," *CoRR*, vol. abs/1702.07521, 2017.

[21] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. Rajamani, S. A. Seshia, and K. Vaswani, "A Design and Verification Methodology for Secure Isolated Regions," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16, 2016.

[22] D. McGrew and J.Viega, "The Galois countermode of operation (GCM)," Submission to NIST Modes of Operation Process, 2004.

[23] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[24] C. che 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)*. Santa Clara, CA: USENIX Association, 2017, pp. 645–658.

[25] R. C. Merkle, "Secrecy, Authentication, and Public Key Systems," Ph.D. dissertation, Stanford University, 1979, aAI8001972.

[26] R. Elbaz, D. Champagne, R. B. Lee, L. Torres, G. Sassatelli, and P. Guillemin, "Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks. cryptographic hardware and embedded systems-ches," 2007.

[27] H. Pang and K.-L. Tan, "Authenticating query results in edge computing," in *Proceedings of the 20th International Conference on Data Engineering*, 2004.

[28] R. Strackx and F. Piessens, "Ariadne: A Minimal Approach to State Continuity," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 875–892.

[29] 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)*. Vancouver, BC: USENIX Association, 2017, pp. 1289–1306.

[30] A. Shamir, "How to Share a Secret," *Communication of ACM*, vol. 22, no. 11, Nov. 1979.

[31] Kissdb. [Online]. Available: https://github.com/ftes/kissdb-sgx

[32] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious Multi-Party Machine Learning on Trusted Processors," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 619–636.

[33] TPC. (2017) TPC-C Homepage. [Online]. Available: http://www.tpc.org/tpcc/

[34] TATP. (2017) Telecom Application Transaction Processing Benchmark. [Online]. Available: http://tatpbenchmark.sourceforge.net/

[35] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the Library OS from the Top Down," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, 2011, pp. 291–304.

[36] C. Gentry, "Fully Homomorphic Encryption Using Ideal Lattices," in *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, ser. STOC '09, 2009, pp. 169–178.

[37] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic Evaluation of the AES Circuit," in *Advances in Cryptology–CRYPTO 2012*. Springer, 2012, pp. 850–867.

[38] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology Eurocrypt 1999*. Springer-Verlag, 1999, pp. 223–238.

[39] M. Bellare, A. Boldyreva, and A. O'Neil, "Deterministic Encryption and Efficiently Searchable Encryption," in *Proceedings of the International Symposium on Cryptography*, 2007.

[40] G. Amanatidis, A. Boldyreva, and A. O'Neil, "New Security Models and Provably-Secure Schemes for Basic Query Support in Outsourced Databases," in *Proceedings of the International Symposium on Cryptography*, 2007.

[41] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra, "Executing SQL Over Encrypted Data in the Database-Service-Provider Model," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. ACM, 2002, pp. 216–227.

[42] R. Poddar, T. Boelter, and R. A. Popa, "Arx: A strongly encrypted database system," *IACR Cryptology ePrint Archive*, vol. 2016, p. 591, 2016.

[43] A. C.-C. Yao, "How to generate and exchange secrets," in *Foundations of Computer Science, 1986., 27th Annual Symposium on*. IEEE, 1986, pp. 162–167.

[44] S. W. Smith and D. Safford, "Practical Server Privacy with Secure Coprocessors," *IBM Systems Journal*, vol. 40, no. 3, pp. 683–695, 2001.

[45] L. Bouganim and P. Pucheral, "Chip-secured Data Access: Confidential Data on Untrusted Servers," in *Proceedings of the 28th International Conference on Very Large Data Bases*, ser. VLDB '02, 2002, pp. 131–142.

[46] E. Mykletun and G. Tsudik, "Incorporating a secure coprocessor in the database-as-a-service model," in *Innovative Architecture for Future Generation High-Performance Processors and Systems, 2005*. IEEE, 2005, pp. 7–pp.

[47] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16, 2016, pp. 689–703.

[48] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB Linux Applications with SGX Enclaves," in *24th Annual Network and Distributed System Security Symposium, San Diego, California, USA, February, 2017*, 2017.

[49] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch, "Glamdring: Automatic Application Partitioning for Intel SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 285–298.