Dhinakaran Vinayagamurthy*, Alexey Gribov, and Sergey Gorbunov

StealthDB: a Scalable Encrypted Database with Full SQL Query Support

Abstract: Encrypted database systems provide a great method for protecting sensitive data in untrusted infrastructures. These systems are built using either special-purpose cryptographic algorithms that support operations over encrypted data, or by leveraging trusted computing co-processors. Strong cryptographic algorithms (e.g., public-key encryptions, garbled circuits) usually result in high performance overheads, while weaker algorithms (e.g., order-preserving encryption) result in large leakage profiles. On the other hand, some encrypted database systems (e.g., Cipherbase, TrustedDB) leverage non-standard trusted computing devices, and are designed to work around the architectural limitations of the specific devices used.

In this work we build StealthDB – an encrypted database system from Intel SGX. Our system can run on any newer generation Intel CPU. StealthDB has a very small trusted computing base, scales to large transactional workloads, requires minor DBMS changes, and provides a relatively strong security guarantees at steady state and during query execution. Our prototype on top of Postgres supports the full TPC-C benchmark with a 30% decrease in the average throughput over an unmodified version of Postgres operating on a 2GB unencrypted dataset.

Keywords: Encrypted databases, Intel SGX

DOI 10.2478/popets-2019-0052
Received 2018-11-30; revised 2019-03-15; accepted 2019-03-16.

1 Introduction

Over the last decade, storing and processing of enterprise data for a lot of companies has moved from the company's data center to third party public cloud infrastructure or service providers like AWS, Microsoft Azure and Google Cloud. These infrastructures are operated and maintained by potentially untrusted operators. Also, the infrastructure is shared between numerous clients. For instance, a single AWS physical instance may co-locate a number of virtual client instances. Given these *features*, protecting the confidentiality and integrity of user's data from administrators, co-tenants, and other attackers is a major challenge.

To tackle this problem, research has been done to build "encryption-in-use" mechanisms that greatly improve security by preventing the attackers and even the cloud operators from ever seeing the data in clear. A lot of work has been done on improving the security and performance on a subset of SQL operations as systematized in the survey by [22], but only a handful of systems are complete and evaluated at scale. The state of art encryption-in-use database systems which have been evaluated at scale can be divided into two main categories:

(A) systems built using advanced encryption schemes that allow to perform operations over the ciphertexts [45–47], and
(B) systems that leverage a trusted processing device (e.g., FPGA, IBM secure co-processor) to perform operations [2, 4, 18].

A practical encrypted database design is evaluated in terms of the following four aspects:

– security: leakage profile and security assumptions. Leakage profile characterizes the amount of data leakage introduced by the design. Security assumptions include the mathematical assumptions for the cryptography and the trusted computing base (TCB) and other trust assumptions for the trusted hardware.
– functionality: the SQL operations and DBMS functions supported.
– performance: throughput, latency and scalability to large datasets.
– intrusiveness level: amount of changes to the underlying DBMS.

***Corresponding Author: Dhinakaran Vinayagamurthy:** IBM Research India, E-mail: dvinaya1@in.ibm.com. Work done while at University of Waterloo.
Alexey Gribov: Symbiont.io, E-mail: gribov.alesha@gmail.com. Work done while at Stealthmine Inc.
Sergey Gorbunov: University of Waterloo and Algorand, E-mail: sgorbunov@uwaterloo.ca

CryptDB [47] is a seminal work in this area using property-preserving encryption schemes to execute queries over encrypted data. But, these schemes do not offer strong security and when used in multiple columns they are found to leak extensive information for real-world datasets [29, 41]. Also, [47] requires extensive computations (re-encryption of entire columns) on a trusted proxy or the client to support all the SQL queries. The other systems using advanced encryption schemes either have a very limited functionality [11, 34, 45] or incur heavy computational and storage overheads [46].

Cipherbase [2] offers a scalable design for transactional workloads with a strong leakage profile and complete SQL support, by leveraging on trusted hardware. But, the system uses FPGAs as its trusted hardware and hence has the following security implications: (i) an initial trusted and on-premise key loading phase is required for every FPGA device used, (ii) a huge trust is placed on the FPGA“shell” layer [1] implemented by the cloud operators which monitors the user operations on the FPGA to ensure the safety of the device. As such, significant research is required to use FPGAs as *trusted* hardware in cloud-based applications. The other trusted hardware based systems [4, 18] offer improved leakage profile but only at the cost of extensive DBMS changes, much larger TCB and huge performance overheads for large transactional workloads.

In this work, we study how to build an encrypted database system from a standard CPU leveraging the Intel Software Guard Extensions (SGX) instruction set [39]. SGX enables the creation of a small encrypted memory container (enclave) that can be accessed only by a predefined trusted code. The content of the enclave is protected from untrusted applications and even the system administrators, OS and hypervisor. Also, SGX is available in all the recent and future releases of Intel CPUs. Hence, SGX offers a great direction for protecting applications in cloud environments. But, SGX has its own set of restrictions. It requires rewriting of applications by partitioning code into trusted and untrusted segments. Also, there is a 90 MB bound on “secure” memory to run the trusted enclave code, which is not nearly enough for even medium size database workloads.[1] Additionally, SGX is vulnerable to memory, cache and other side-channel leakages, lacks syscalls and IO support, and incurs high overheads for switching between enclave and non-enclave modes, which further limit the complexity and functionality of the trusted enclave code. As such, one cannot take a DBMS system and naively try to “run it in an enclave”. But, it is important for an encrypted database design to get around these limitations without having to make extensive changes to the underlying DBMS, while still achieving the performance, security and functionality goals. Also, it is not clear whether a design that works well for another trusted hardware can be ported to SGX while preserving the end-to-end security guarantees, since each hardware has its unique set of security and usability requirements.

1.1 Our contributions

Design choices with SGX

We first investigate three possible design choices for an encrypted database with SGX in Section 4.2 by varying the DBMS components run inside an enclave. Through a set of benchmarking experiments, we identify a design that works best for our design goals (Section 5). We develop on that to get the StealthDB design.

StealthDB

The StealthDB system provides a complete SQL support, strong end-to-end security guarantees and performance with minimal changes to the underlying DBMS. A high-level overview of our system is presented in Figure 1. StealthDB uses AES-CTR, a semantically secure encryption scheme to encrypt all the data items in the database. During query execution, the client encrypts the query string and sends the ciphertext to the server. We implement a query parser inside an enclave, which first decrypts the ciphertext to get the query and parses the query to output a version with all the constants encrypted. For example, when a client sends ENC(select * from item where name = ‘John’), it is converted to select * from item where name = ‘ENC(John)’ by our enclave parser. To support queries of this form, we define

1 Although various SGX extensions are promised by Intel in future releases with larger secure memory, they are not available in the market yet and unclear when they will be. We also argue in the paper that these extensions should not affect our conclusions on the architecture of an encrypted DBMS with SGX.

encrypted datatypes and implement the operators over these datatypes inside an enclave. We make the operators data-oblivious [42] to protect against SGX side-channel attacks. We also encrypt the index file pages before they are written to disk. These changes are not intrusive and hence enable StealthDB inherit the functionality of the underlying DBMS completely.

Security

StealthDB offers a stronger leakage profile compared to the prior complete encrypted database systems. A snapshot adversary [5, 8, 16, 23, 50] learns only the "shape" of the database which includes the dimensions of the data structures maintained by the DBMS, along the recently collected query log information. An adversary with persistent access to memory and disk learns the inequalities ($<, >, =$) between the encrypted values in the indexes which are compared during the query execution, along with the query access pattern which includes the position of the result records in the database. In general, the enclave code can be thought of as providing a black-box access to the DBMS to perform the computations on encrypted data values and obtain the output (encrypted or unencrypted depending on the specification), without leaking any other information about the input data values. We explain our leakage profile in more detail in Section 6, and this profile matches the state-of-art (the strongest version in [2]) when providing either reasonable performance[2] or intrusiveness levels for large transactional workloads. Also, our TCB just includes the processor, the enclave code along with the SGX hardware and the attestation procedure. Our clients use the SGX attestation procedure to attest the correctness of the enclave code before issuing queries. This combined with the simplicity of the enclave code reduces the trust to be placed on the enclave code.

Evaluation

We implement our design on top of an existing Postgres DBMS. Our new encrypted datatypes and the corresponding UDFs are added as extensions in Postegres [48]. The only component that needs modifying the Postgres code is to encrypt/decrypt the index files when they are stored to/accessed from disk and this just needs a three lines change in the Postgres codebase. None of these changes are intrusive, or specific to Postgres. Hence, this design principle lets StealthDB benefit directly from any performance or feature improvements to the underlying DBMS engine. Performance-wise, StealthDB scales to large datasets with a similar complexity to an unmodified DBMS engine working on unencrypted data, adding only a tiny overhead for each query. Our evaluation results in Section 7 show that the system can process OLTP queries with a 30% reduction in throughput and ≈ 1 ms overhead in latency over an unencrypted DBMS with > 10M total rows (or 2 GB plaintext) of a TPC-C warehouse database for scale factor $W = 16$.

2 From our experience talking to the industry on the possible adoption of StealthDB, 50% to 2× overhead in *performance* is a reasonable penalty for the benefit of security against untrusted cloud operators.

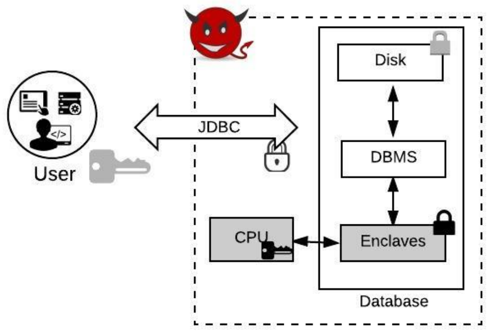


Fig. 1. High-level architecture overview of StealthDB

2 Background on Intel SGX

In this section we give a brief introduction to Intel Software Guard Extensions (SGX). We refer the reader to [14, 39] for more details on SGX. Intel SGX is a set of new x86 instructions that enable code isolation within virtual containers called enclaves. In the SGX architecture, developers are responsible for partitioning the application into enclave code and untrusted code, and to define an appropriate I/O communications interface between them. In SGX, security is bootstrapped from an underlying trusted processor, but not trust in a remote software stack. On the high level, the SGX hardware presents the following two functionalities to a user:

– $\mathsf{Load}(\mathsf{P}) \rightarrow (\mathsf{E}_\mathsf{P}, \phi)$: creates an enclave with an identifier E_P and loads the program P into it. It then produces a proof ϕ that the intended program P (and initial data) has been loaded into the enclave.

– $\mathsf{Execute}(\mathsf{E_P}, \mathsf{input}) \rightarrow (\mathsf{out}, \psi)$: given an enclave handle $\mathsf{E_P}$ (corresponding to an enclave with a program P), $\mathsf{Execute}$ runs it on an input input and produces a tuple constituting of the output out and a proof ψ. A client can use ψ to verify that out was produced by the enclave $\mathsf{E_P}$ executing with input.

There are three main functionalities that enclaves achieve: isolation, sealing and attestation. We provide a high-level description here. Please refer to [14, 20] for more detailed and formal descriptions.

Isolation: code and data inside the enclave protected memory cannot be read/modified by any process external to the enclave.

Sealing: data passed to the host environment is encrypted and authenticated with a *Seal Key* that is specific to the enclave identity and derived from a hardware-resident *Root Seal Key*. SGX uses AES-GCM to encrypt msg using the Seal key of the enclave calling the function.

Attestation: a special signing key and instructions are used to provide an unforgeable report attesting to code, static data, and (hardware-specific) metadata of an enclave, as well as outputs of computations performed inside the enclave. There are two forms of attestation: *local* and *remote*.

– *Local attestation.* An enclave A uses local attestation procedure to generate a *report* and attest to another enclave B on the same platform.
– *Remote attestation.* Remote attestation procedure generates a report specific to an enclave called *quote* that can be verified by any remote party.

Key establishment during attestation. Key establishment between two enclaves or between an enclave and a remote party can be accomplished on top of the local/remote attestation process. An enclave can send the key shares (for eg., a Diffie-Hellman key share g^a) and include them as the *additional authentication data* to MAC. Thus attestation provides authenticity and integrity to the key share from the enclave. In our system, we will very often run the key establishment phase on top of local/remote attestation to establish a secure channel for communication between two enclaves or between an enclave and a remote party using the established shared secret key.

SGX TCB. SGX stands out in that its TCB consists only of the CPU microcode and privileged containers, however it also requires the user to trust in Intel's key management infrastructure for signing microcode and various service enclaves. In particular, we must trust that the root seal keys embedded into devices are not leaked from the manufacturing facility, and that the Intel Provisioning Server safely manages root provisioning keys as well as other master secret keys.

Although SGX prevents an adversary from directly inspecting/tampering with the contents of the EPC, it does not protect against multiple software-based side channels. Correspondingly, the literature has demonstrated attacks that extract sensitive data through hardware resource pressure (e.g., cache [7, 9, 15, 52], thread scheduling [54] and branch predictor [37]) and the application's page-level access pattern [55]. Many of these works also provide fixes for their attacks with varying overheads and need to be patched by Intel. For the application's page-level access pattern though, it is up to the application developer to design data-independent memory accesses for the data to be secure.

3 Platform Overview

3.1 Usage Model.

We work with the following setting. A data owner aims to store and process data securely on a remote untrusted SQL database server. She authorizes clients by issuing them *credentials*, and wants to support the authorized clients to issue queries to the server. The server maintains a *credential database* for the authorized clients in an encrypted form. Each client authenticates to the server using its credentials, which will enable the client to issue its permitted queries to the database. The server in our model is equipped with a secure processor, such as Intel SGX. Hence, the server can be identified with some "platform-key" established by Intel SGX. The data owner and clients engage in the attestation of SGX enclaves in the server and on successful attestations, transfer any secret or sensitive material (master key, credentials, queries, etc.) to those enclaves via secure channels.

3.2 Threat Model

StealthDB provides security against passive adversaries. A passive adversary does not inject malicious code or alter the program execution in any way. But, it can read the contents of the memory, disk and all the communication, and hence may *passively* attempt to learn additional information from the data they observe.

There are two dimensions in which we analyze the threat model for our system. The first dimension is about the extent of access: adversaries restricted to monitoring the disk accesses versus the adversaries monitoring both the memory and disk accesses in the system. The second dimension is about the duration: adversaries getting snapshot accesses to memory or disk versus the much stronger ones which get persistent access to memory and disk. A snapshot attack might be due to a memory dump or some cold-boot attack by a malicious cloud provider or by a co-located client running on the same cloud server as the victim process which gets occasional access to the memory of the entire system due to access control bugs. SQL injection attacks [16, 17, 31], VM attack leaks [5, 8, 23, 50], disk theft and a “smash-and-grab” after a full system compromise [16] are some real-world examples of snapshot attacks [30].

4 Designing an Encrypted DB

In this section, we describe a few design goals we set out to achieve for our system. Then, we discuss and experiment with a few possible design choices possible when building an encrypted database from SGX.

4.1 Design Goals

The focus of StealthDB is on building a scalable encrypted database system that can support arbitrary query types, with a reasonable leakage. Construction of an encrypted DBMS with a complete SQL support under any meaningful notion of security is an uphill task in this world where the proposed attacks [28–30, 35] completely dismantle the security of even the constructions with limited functionality (like searchable encryption) which had, what was thought to be, minimal leakage (reveal just the locations of the results of each query). There has been extensive research to secure subsets of SQL operations [22], but a proposal can be included in a real world DBMS only if it is compatible with or provides a complete support of the DBMS tasks. For instance, the CryptDB design was part of or inspired many real-world systems [26, 27, 40, 47] due to an almost complete DBMS support. In this regard, we set our design goals as follows:

- Functionality goal: complete support to the SQL functionality of the underlying DBMS.
- Non-intrusiveness goal: minor modifications to the core DBMS operations of the underlying DBMS, for the encrypted database to retain the DBMS properties. If the underlying DBMS is ACID compliant, supports triggers and stored procedures, so should the encrypted database.
- Performance goal: high throughput and low latency when scaling to large datasets.
- Security goal: We will start by stating the security goals informally:
  - a snapshot adversary on both memory and disk should learn no information about the individual data items.
  - a persistent adversary on both memory and disk learns no information about the encrypted data that are not compared when the queries are processed, other than that they are not part of the query processing. Even for the data of the query execution, the leakage should match or be stronger than the previous works supporting complete SQL.

  We will later study the security for each proposed design. And, the leakage profile of the chosen StealthDB design will be detailed in Section 6.

There is an inherent trade-off here between security and performance which will influence our design choices. There is a lower bound of logarithmic overhead in performance [12, 25], just to support encrypted search without any leakage. This also translates to the trade-off between efficiency and the information leakage during the index building and usage. Moreover, we also aim to design secure versions of arithmetic and other operators to support SQL completely. Hence in this work, we lean towards achieving a good performance for large transactional workloads, while trying to achieve the best security possible for that performance.

4.2 Designing an Encrypted DB from SGX

We consider three design choices and evaluate them on a few micro experiments to help us understand how to build an encrypted database system with SGX. The design choices are summarized in Figure 2. We envision that in all three design choices data is encrypted on disk using a semantically secure encryption scheme. The designs differ in how queries are executed over the data.

The first, most obvious design would be to run the entire DBMS inside an enclave (left figure in 2). The data would be read from disk, decrypted transparently

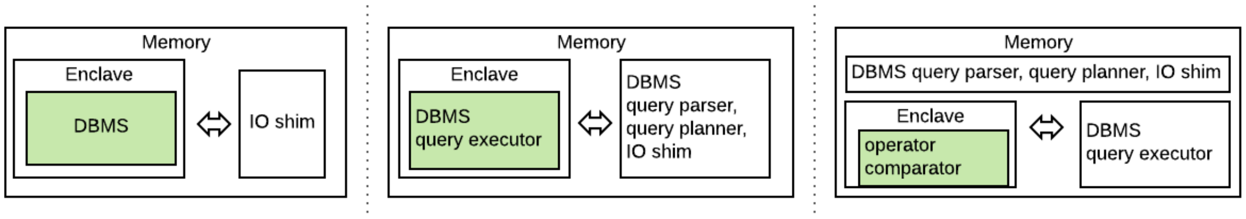


Fig. 2. Three alternative design choices for an encrypted database with SGX.

and then the DBMS would perform all necessary operations inside an enclave. However, SGX is not well suited for this task for a few reasons that we outlined earlier. The first issue is that SGX does not support IO or syscalls, so an additional outside shim layer would need to be exposed to talk to the kernel level, and the application dependencies need to be loaded inside (or outside via shim) an enclave. It is *feasible* to get around this issue using recent works such as Haven [6], Scone [3] and Graphene [13, 53]. They initiate the research in loading unmodified executables into enclaves. The second issue is that SGX is currently limited to 90 MB of working memory and significant penalties appear when going beyond that limit [43]. Future releases of SGX promise larger enclave sizes. However, the Merkle tree integrity protection for each memory page to prevent replay attacks does not scale well to larger enclaves. These two issues would result in heavy performance overheads on transactional workloads for this design.[3] But, this design can have better confidentiality guarantees when the SGX-based side-channels are addressed.

The second design we consider (middle figure in 2) keeps most of the DBMS in the untrusted zone. However, it places the query execution logic in the enclave. That is, when a query needs to be executed, individual tables can be brought in to the enclave to perform selections, projections, joins, etc. The query plan, I/O and other DBMS parts remain in the untrusted memory. In terms of scalability, this design suffers from the same problems as the previous choice due to limited secure memory. Also, tables and indexes need to be read from disk, deserialized and then loaded into enclave. In Figure 3 we show that the performance overhead for performing just this step (read and deserialize) inside an enclave is around 3× when the dataset fits within an enclave, and goes up to 9× for large datasets. In terms of security, the query processing logic would still need to do the non-trivial task of addressing the SGX side-channels. Finally, partitioning a DBMS to support this architecture is also a challenging task.

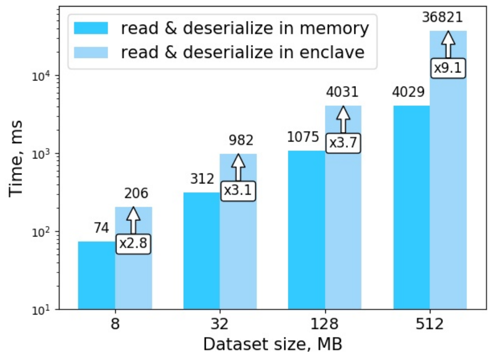
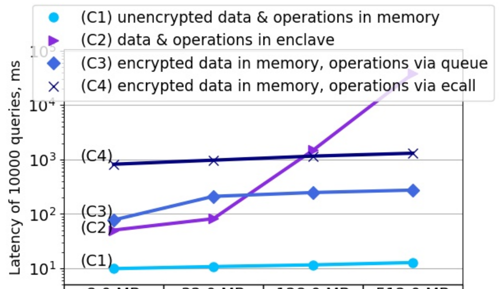


Fig. 3. Initialization time comparing in memory and in enclave deserialization for different dataset sizes.

In the third design, we keep most of the DBMS in the untrusted zone, and the dataset would reside in the untrusted memory with the data items encrypted individually. At the lowest level of the parsed query tree, each query is eventually broken down into some *primitive* operators (e.g., $<=, >=, +, *$) over individual data values. To perform operations over encrypted data in this design, we transfer individual data item(s) to an enclave, followed by the decryption of input, the operator function and the encryption of output inside the enclave. The advantage of this design is that the communication with the disk and network layers would remain unchanged. Overall, minimal changes to the DBMS are needed – one only needs to change how primitive operators on data values are performed. Also, the amount of code/data inside an enclave will remain a very small constant. This keeps the TCB very small, and it is easy to make it data-oblivious. Hence, we build on this design idea in Section 5. However, this design leaks relationship

3 We do not do a direct performance evaluation for this design, but the design that we discuss next which runs much less operations inside an enclave already has high overheads.



	8.0 MB	32.0 MB	128.0 MB	512.0 MB
(C1)	9.9	10.8	11.6	12.8
(C2)	50.5	81.8	1480.1	37590.2
(C3)	77.5	210.4	247.5	274.1
(C4)	819.2	972.8	1155.1	1304.9

Fig. 4. Latency to execute random binary tree searches comparing different approaches. Two different implementations of the partial approach: comparison function as trusted ecalls and the exit-less communication via a queue for transferring data to/from an enclave.

between encrypted data values during query execution in this design as discussed in Section 6.

In Figure 4, we compare performance of performing B-tree searches over database indexes in later two design choices. As expected, one can see that performing a search when an entire B-tree is loaded inside an enclave does not scale to larger datasets. (However, it performs well when the tree size is very small and can be fit entirely into an enclave.) In the third design, when the B-tree is kept encrypted in the untrusted memory but individual comparisons are executed in an enclave, we see up to 100× overheads compared to performing the search over unencrypted data. This can be explained by high switching costs for ecall/ocall functions, which are used for enclave entry/exit. Using an *exit-less* communication mechanism via a shared queue [43], we can reduce this overhead by $5\times -10\times$.

5 Architecture

The architecture of StealthDB is presented in Figure 5. As discussed in our third design, StealthDB makes minimal changes to the underlying DBMS, with most of our components augmented on top of an unmodified DBMS. We will now go through the flow of database creation and query life-cycle, and explain each of our components in detail as needed.

5.1 Database creation

When a database is created, the database owner designs a database schema to define the structure of the database. During the schema creation, StealthDB allows the owner to identify the columns of the tables in the database which have sensitive information and use our *encrypted datatypes* for those columns. An encrypted datatype is used to represent values which are the encrypted versions of its corresponding plaintext datatype. For instance, encrypted integers are represented by the encrypted datatype *enc_int4*.

```
CREATE TYPE enc_int4 (
    INPUT          = enc_int4_in,
    OUTPUT         = enc_int4_out,
    INTERNALLENGTH = 45,
    ALIGNMENT      = int4,
    STORAGE        = PLAIN
);
```

Fig. 6. Definition of *enc_int4*

And, a database owner can issue the following command to create a table *item* with two columns of types encrypted integers and encrypted strings:

```
CREATE TABLE item (
        id enc_int4 NOT NULL,
        name enc_text NOT NULL
);
```

StealthDB will encrypt the data values in an encrypted datatype using AES-CTR which is an encryption scheme providing confidentiality of the data values. We will discuss about the key(s) used by this encryption during the DBMS initialization.

5.2 DBMS Initialization

When the DBMS is started, the following additional steps are performed for StealthDB.

Enclaves creation

StealthDB creates three enclaves on the database server: the client authentication enclave Auth, the query preprocessing enclave PreProcessor and the operation enclave Ops. These enclaves are loaded by an *untrusted* DBMS runtime, but our system will later allow to *attest* that the correct code has been loaded into the enclaves. The clients use the remote attestation process and the publicly available measurements (hash) of the enclave

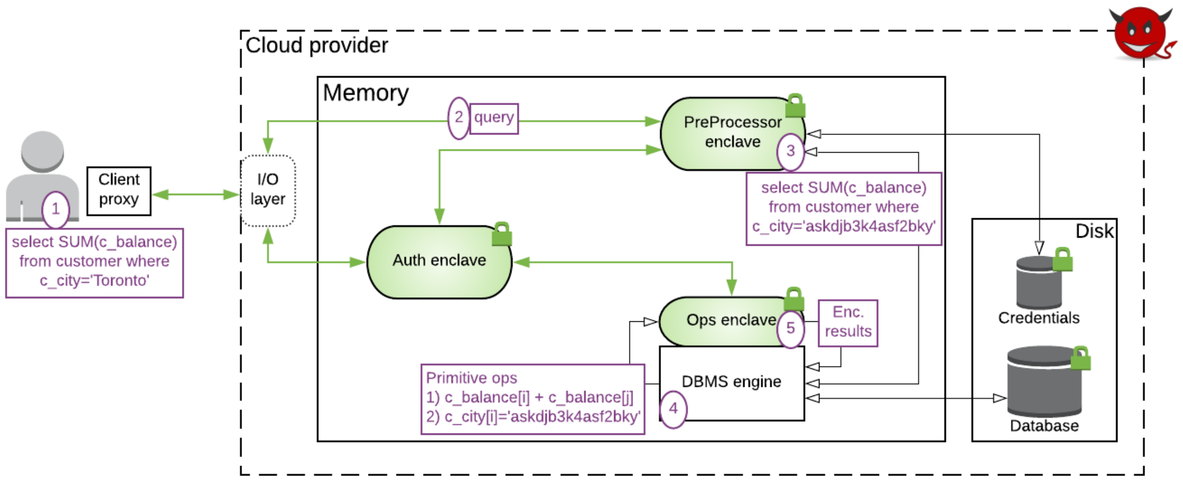

Cloud provider
Memory
Client proxy
1
select SUM(c_balance) from customer where c_city='Toronto'
I/O layer
2 query
PreProcessor enclave
3
select SUM(c_balance) from customer where c_city='askdjb3k4asf2bky'
Auth enclave
Ops enclave
5
Enc. results
DBMS engine
4
Primitive ops
1) c_balance[i] + c_balance[j]
2) c_city[i]='askdjb3k4asf2bky'
Disk
Credentials
Database

Fig. 5. StealthDB architecture. The life cycle of a query initiating from a client can be traced from steps 1 to 5. The lines with shaded arrows represent encrypted communication between those entities.

code to ensure the correctness of the loaded programs in the enclaves. We will defer the explanation of this step and the functionality of these enclaves to the sections below.

To facilitate the communication between the users and the enclaves, StealthDB introduces an I/O layer on the server side. Its job is to simply redirect requests between the appropriate enclaves and the DBMS. This will also act as the *wrapper* program for the enclaves helping in processing their I/O requests and system calls. Note that this layer is outside the SGX TCB, hence it is untrusted and can be controlled by an adversary.

Key generation

The initialization phase also involves generating a master secret key. StealthDB performs key generation inside the Auth enclave. Auth runs the KeyGen() function to sample a 128 bit secret key K at random for the AES encryption/decryption operations. In the current design, this master key K will be used to encrypt all the data values in the database. We do this for simplicity and our design can be extended to support an integration with a key management service to enable the usage of different keys for different clients or for different columns in the database.

Figure 7 outlines the key generation and transfer procedures. The master key K is then transferred to the PreProcessor and the Ops enclaves as follows. When the PreProcessor and Ops enclaves are created, they individually perform a local attestation with Auth and establish a secure channel with Auth. When the attestations succeed and after the secure channels are established, Auth's KeyTransfer() function uses the channels to send the master key K to PreProcessor and Ops. (On the other end, PreProcessor and Ops will run their KeyReceive() functions to complete these steps and receive K). On obtaining K, PreProcessor and Ops use SGX's sealing property to encrypt and store K for future use.

Transfer of credentials

The final task of the initialization phase involves transferring the client credentials and access policies to Auth. A client (proxy) will authenticate to Auth. And, from the point of view of the DBMS, Auth (and PreProcessor) will act as a client who has complete access to the database. To facilitate this, the data owner first engages in a remote attestation protocol with Auth along with a secure channel establishment and if it succeeds, she sends the *master* credentials along with the database of client credentials and access policies to Auth through the established channel. On obtaining these, Auth uses the SGX seal operation to encrypt and store them.

5.3 Client authentication

One of the challenges we need to address is to make sure that only the authorized users can query the encrypted database system. For this, we design an authentication method built on top of an existing DBMS.

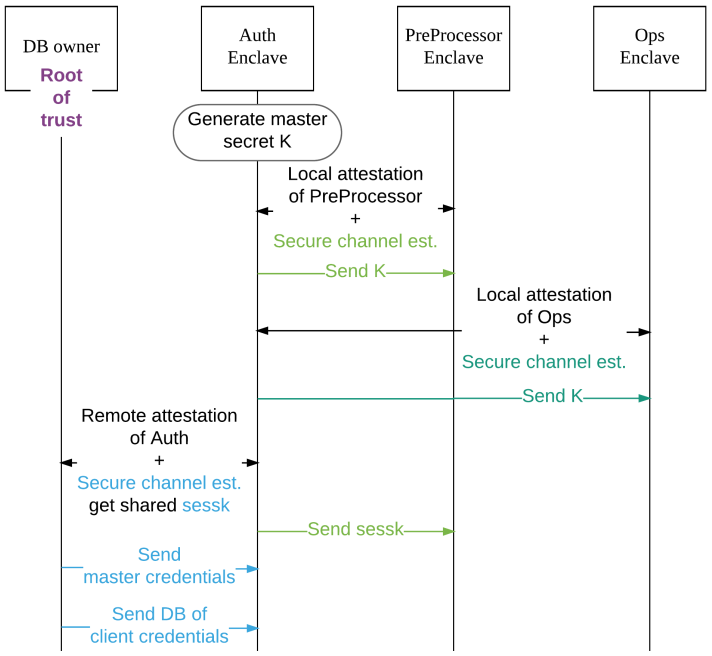

DB owner
Root of trust
Auth Enclave
PreProcessor Enclave
Ops Enclave
Generate master secret K
Local attestation of PreProcessor + Secure channel est.
Send K
Local attestation of Ops + Secure channel est.
Send K
Remote attestation of Auth + Secure channel est. get shared sessk
Send sessk
Send master credentials
Send DB of client credentials

Fig. 7. The authentication protocol of StealthDB

After the database server is started, it is now ready to accept connections from the clients. Here, StealthDB adds an authentication mechanism for the clients to authenticate to the Auth enclave. This works as follows.

First, the client proxy verifies that the DBMS has loaded the *correct* code into Auth, by performing the remote attestation (plus secure channel establishment) protocol with Auth as described in Section 2. Let sessk be the shared secret key obtained after its successful completion. The client will then authenticate to the Auth enclave using its credentials, say its password or its SSH key, through the established secure channel. On the server side, the I/O layer directs the client authentication requests to the CompleteClientAuth() function in Auth. CompleteClientAuth() unseals the client credentials database and uses it to verify the client credentials. If the client authentication completes successfully, the shared secret key sessk will be used as the *session key* for the client.

Once the client authentication is completed, the interaction with the client for query processing will be performed by the PreProcessor enclave. To facilitate this, the I/O layer will now invoke the TokenTransfer(ID, sessk) function in Auth to transfer the client "ID" and sessk to PreProcessor. This transfer will use the secure channel established between these enclaves during the master key transfer.

5.4 Query execution

Now we will explain the working of query processing and execution in StealthDB for a client which has completed its authentication successfully. The design of StealthDB permits the use of an unmodified query driver (e.g. JDBC, ODBC, etc.).

When a client issues a query, the client proxy encrypts the entire query string using the session key sessk with its ID included in the additional authenticated data. On the server side, the I/O layer directs the client queries to PreProcessor. The QueryPreProcessing function first decrypts the query ciphertext using the session key sessk for ID. Then, it checks whether this client is permitted to run this query. Typically, a DBMS allows the DB owners to specify access control policies for the clients. In StealthDB, we rewrite the access control monitor inside PreProcessor. If the checks are passed, QueryPreProcessing identifies the data values in the query which correspond to the columns in the database using encrypted datatypes using our query parser, and AES-encrypts these data values using the master secret key K. The output of this step, encquery, is given to the DBMS for execution.

Note that the DBMS is oblivious to the changes made to the query. The *structure* of encquery is same as that of the query issued by the client. This lets the DBMS use an unmodified query parser to parse this query. But after the query is parsed and a query plan is obtained, we need to augment the DBMS with functions to operate on the encrypted datatypes. We do this as follows.

We first identify the set of *primitive* operators used by the underlying DBMS. Primitive operators are those further-indivisible operators used in query plans:

- *Arithmetic operators* such as $+, -, \%, *$, etc.
- *Relational operators* such as $<, >, <=, >=, <>$, etc.
- *Logical operators* such as AND, OR, NOT, etc.
- *Hash functions* that are used to build some indexes.
- *Advanced math functions* such as *sin, cos, tan*, etc.

Traditionally, DBMSs define a functionality for each input datatype tuple supported by a primitive operator. StealthDB augments these with their functionalities when used with the corresponding encrypted datatypes as in Figure 8. Our implementation on Postgres implements primitive operator functionalities over the encrypted datatypes and include them as *extensions*.

For every possible input datatype tuple, we define a function inside the Ops enclave. Suppose that we are

```
CREATE OPERATOR = (
  LEFTARG = enc_int4,
  RIGHTARG = enc_int4,
  PROCEDURE = enc_int4_eq,
  COMMUTATOR = '=',
  NEGATOR = '<>',
  RESTRICT = eqsel,
  JOIN = eqjoinsel,
  HASHES, MERGES
);
```

Fig. 8. Operator = for *enc_int4*. Here, *enc_int4_eq* will call the Ops enclave to decrypt the input, check their equality and output the result.

given two encrypted data values (e_1, e_2) and an operator $\oplus$, the corresponding function inside Ops will perform:

1. *decryption* of the inputs e_1, e_2 using the master key to get plaintext values p_1, p_2,
2. *perform the operator function* to get $p_{\text{out}} = p_1 \oplus p_2$,
3. *encrypt* the result p_{out} to get a ciphertext e_{out} using the master key (if specified by the design).

The number of inputs and outputs may of course vary depending on operator. Moreover, datatype conversions are also allowed in our model. For example, an encrypted integer may be converted to an encrypted string, and so on. Overall, we only perform a few basic operations (decrypt, primitive operator, encrypt) during the query execution inside the enclave.

Finally, once the final result of the query is obtained, PreProcessor re-encrypts the results using the session key sessk and send them back to the client proxy.

Standard SGX *ocall/ecall* communication mechanism with enclaves is too slow when many calls are needed. To solve this, we implement an *exit-less* mechanism [43] for communicating with Ops. In [43], there is always one thread running inside an enclave listening for operator jobs. The DBMS uses our I/O layer to send jobs and receive replies via a communication queue. This method greatly improves performance by avoiding context switch for each call to the operator between trusted and untrusted zones, as we discussed earlier in Section 4.2.

There are also other inherent advantages with our design.

– When a client issues a query only involving unencrypted datatypes, the query processing and execution proceeds in the *native* way and hence with no overheads.
– A very interesting property is that our design also allows for computations between encrypted datatypes and unencrypted datatypes. The database owner here can also specify that the output of such computations should be encrypted to avoid leaking information about the encrypted inputs.
– Since our design implements only the primitive operators, it is easy for us to implement them inside Ops using data-oblivious methods [42] with a small performance overhead to counter the side-channel attacks of SGX.

5.5 Encrypting indexes

The indexed columns, unlike the other columns in the table, need extra layers of protection. When the column is indexed into a B-tree, for example, the structure of the tree reveals the inequalities with respect to the values in the column even though the individual values in the tree are encrypted. The inequalities are available even to a snapshot adversary after index creation before any query is made to the database. We provide two modifications to reduce this leakage. First, we re-encrypt the individual values in the column when placing these encrypted values in an index structure. This unlinks the connection between the values in the table and the index. This unlinking is maintained for an adversary obtaining only a snapshot of the table and the indexes. Even for a slightly weaker persistent adversary which does not observe the system during the index creation, the inequalities observed from the index structure can be connected to the table values only when a query accesses the corresponding table row as part of its result. For an adversary persistent throughout the index creation and usage, the security reduces to that provided by order-revealing encryption (ORE) [38] on the indexed columns. This change does not incur a performance overhead during the query execution in StealthDB.

The second change deals with this leakage on disk. StealthDB encrypts every page that is written to the files on disk corresponding to the indexes. We do this by encrypting the data right before it is written to the index files on disk, and decrypting the data read from the index files right after it is read from disk. In our implementation for Postgres, our changes to the codebase involve adding three lines of code to do this task. We create and run a fourth enclave Index_OP during the DBMS initialization which performs the encryption and decryption of the index data pages. And the three new lines are for retrieving the enclave ID, calling the encryption function inside Index_OP right before a FileWrite() of Postgres and for calling the decryption function inside Index_OP right after a FileRead(). The key used for these routines is generated and stored by Index_OP, and Auth attests the correct loading of Index_OP during the DBMS initialization.

5.6 Extensions

Encrypting logs

Some of the log files reveal sensitive information about the queries even for a snapshot adversary on disk [30]. We can protect against an adversary accessing disk by encrypting the log files on disk in a way similar to our encryption of index files on disk. Perhaps, one could ask why we do not encrypt every page written to disk, not just indexes and logs. But the individual data items in the tables are already encrypted and we get no concrete security improvements by encrypting the individual disk pages containing those data items.

Key management

In the current implementation, we use a single master key K to encrypt all the data values. K is sealed and stored on the disk by PreProcessor or Ops enclave when obtained from Auth. If and when the system is restarted, the enclaves are created again and a valid PreProcessor or Ops enclave can unseal the corresponding sealed components to obtain K. During this process, the AES-GCM encryption used in the SGX sealing provides confidentiality and integrity for the sealed component of K against any adversary. Also, when replicating the database across multiple machines, we can let the Auth in one of the machines to generate K and do a remote attestation to transfer it to the Auth enclaves in the other machines.

6 Security evaluation

The tradeoff between security, functionality, performance and the intrusiveness level decided by our design results in the leakage profile that we explain in this section.

First, we will discuss the effect of the SGX side-channel attacks on StealthDB. SGX is subject to various side-channel attacks as described in Section 2. The side-channel due to the application's page-level access pattern is a significant one and it is up to the application developer to design data-independent memory accesses for the application data to be secure. Our design addresses this side-channel by performing only primitive operations inside an enclave (Sections 4.2 and 5) and by using oblivious operators [42] for these primitive operations. We obviate the other software side-channels (except the cache-based ones) by simplifying the code inside the enclaves; running the primitive operations obliviously prevents these side-channels. The cache-based side channels [9, 15] though, are inherent to the x86 architecture and requires patching from Intel. (Also, these are instances of active attacks, which in general StealthDB does not protect against).

Now, let us discuss the leakage profile of StealthDB. As mentioned in our threat model in Section 3.2, StealthDB protects against semi-honest or passive adversaries. It does not provide integrity guarantees to the clients on the correctness of the query results. Neither does it provide confidentiality guarantees against an actively malicious adversary with side-information on the plaintext values encrypted in DB. We will first detail the leakage profile of StealthDB for different variants of semi-honest adversaries and through a series of security claims we will argue that StealthDB does not leak any more information than what is part of the leakage profile. Our evaluation is with respect to the architecture we propose, and hence independent on the specific underlying DBMS engine.

6.1 Leakage profile

StealthDB encrypts the individual data items, rather than an entire column or table at once, and hence this mandates a thorough leakage profiling. We classify the admissible adversaries as in [22] and quantify leakage profiles during the high level operations, Init and Query, of a DBMS for those adversaries. Init involves loading the database in the untrusted server to be ready for querying, and Query involves the client querying the database to get the required results. Note that a query in StealthDB can involve any operator supported by the underlying DBMS (for eg., relational, arithmetic and logical operators for a transactional DBMS).

We analyze the security of StealthDB against passive or semi-honest adversaries. We further classify the adversaries into snapshot and persistent adversaries. A snapshot adversary gets a snapshot to the memory of the system whereas a persistent adversary observes the memory of the system throughout its execution. We motivate these adversarial types in Section 3.2. A formal security definition is provided in Appendix A.1.

Let DB denote the database that we try to securely operate. DB includes all the data structures used by a database (for eg., tables, indexes, views, foreign tables) along with their contents. We will now define the leakage entities to understand the security of StealthDB. To understand the security of our system, we study the leakage profile during different phases of database execution: during the steady state and query execution. The leakage entities of interest to StealthDB are as follows:

- Let $\mathcal{S}_t$ indicate the *shape* of the database at time[4] $t \geq 0$ which includes
  - the database schema,
  - the shape of the tables and (database) views i.e., the number of rows and columns in the tables and views,
  - the shape of the indexes (for eg. the shape of a B-tree index reveals the number of keys in each internal node of the tree).

  More importantly, $\mathcal{S}_t$ does not include the contents of any of the data structures in the database. This entity varies with time depending on the queries run on DB.
- Let $\mathcal{Q}$ denote the leakage associated with a query execution. In StealthDB, $\mathcal{Q}$ is upper bounded by the union of the plaintext outputs of the Ops enclave invocations.
- Let $\mathcal{M}_t$ denote the leakage associated with the logs and the miscellaneous data structures maintained by a DBMS at time t to aid in its operations (including various profiling activities and recovery from unexpected failures).

In StealthDB, the entities $\mathcal{Q}$ and $\mathcal{M}$ are dependent on the underlying DBMS that StealthDB builds on. In Section B, we discuss the information that can be inferred from $\mathcal{S}$, $\mathcal{Q}$ and $\mathcal{M}$ for some real-world data structures and queries.

Note that $\mathcal{S}$, $\mathcal{Q}$ and $\mathcal{M}$ are leakages with respect to DB. We now define the leakage entity Π with respect to a query. In StealthDB, before the query is executed (after output by $\mathsf{PreProcessor}$), the query structure is revealed but not the constants in the query which are encrypted with the semantically secure encryption. With $\mathcal{Q}$ being the leakage during the execution of this query, the total leakage of a client query to the server is upper-bounded by the union of Π and $\mathcal{Q}$ for this query.

- Let Π indicate the leakage about the query before the DBMS begins processing it.

Typically, Π will be a subset of the DB-based leakages. In a real-world DBMS, Π might just be a subset of $\{\mathcal{M}_t\}$ since the details about input queries are usually logged and checkpointed.

We will now argue the leakage profile of StealthDB during different phases of its execution. All the following claims rely on the fact that no information (other than its length) about the key K used to encrypt the data is revealed to an adversary (Claim A.1). We would rely on the following security properties:

1. Remote and local attestation provided by SGX are secure according to Section 2.
2. The confidentiality of the intermediate values of the computation and the integrity of the computation from SGX.
3. The confidentiality and integrity provided by the secure channels established.
4. The confidentiality and integrity of the SGX sealing procedure.

Init phase

StealthDB only leaks the initial shape $\mathcal{S}_0$ during the Init phase. This is better than the OPE or ORE based designs [44, 47] which leak the '$<$' relation between all the values in the OPE/ORE encrypted columns.

Claim 6.1. *After the completion of* Init *and before any call to* Query *is made, StealthDB leaks at most* $\mathcal{S}_0$.

The high-level idea of the correctness of this claim is as follows. Sim obtains $\mathcal{S}_0$ from the leakage oracle $\mathcal{L}$ and outputs encryption of zeros according the shape $\mathcal{S}$ as EDB. An adversary Adv_2 that distinguishes the simulated EDB from a real EDB will break the semantic security of the encryption scheme.

Query phase

We will first argue the leakage of StealthDB for adversaries which obtain snapshot access to the memory. A snapshot adversary in StealthDB learns at most the shape $\mathcal{S}$ and the leakage $\mathcal{M}$ due to the miscellaneous information maintained at the time of the snapshot.

4 "Time" t refers to the epoch at which the data-structure is observed or collected from the system

$\mathcal{M}$ is further upper-bounded by the union of $\mathcal{Q}$ from the queries executed recently. More formally, we have the following claim. The correctness arguments for the claims in this section are in the Appendix.

Claim 6.2. *Consider a polynomial-time snapshot adversary on StealthDB obtaining the snapshot at time t. Let $t' \leq t$ be the latest time epoch before t for which the logs and miscellaneous data structures remain in memory and not written to disk. The adversary learns at most $\mathcal{S}_{t'}$ of the* DB *being operated and $\mathcal{Q}$ of the queries executed between t' and t. If the log items are encrypted in memory and assuming that the size of logs do not reveal sensitive information, the adversary learns at most $\mathcal{S}_t$.*

We will now argue the leakage for a persistent adversary. A persistent adversary in StealthDB learns the plaintext outputs of the Ops enclave invocations throughout its observance. More formally, we have the following claim.

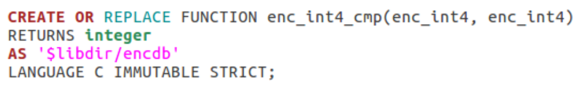
Claim 6.3. *A polynomial-time semi-honest adversary that has persistent access to the memory during the StealthDB execution on a* DB *learns at most the shape $\{\mathcal{S}_t\}_{t\geq 0}$ of* DB *and the query-execution associated leakage $\mathcal{Q}$ for all the queries executed, where $\mathcal{Q}$ is the union of the plaintext outputs of* Ops *invocations during the execution of the query.*

Note that this claim implies that the miscellaneous data structures $\mathcal{M}$ maintained or the parts of DB accessed during query execution do not leak more information than $\{\mathcal{S}_t\}$ and $\{\mathcal{Q}\}$ to a persistent adversary.

7 Implementation and Performance

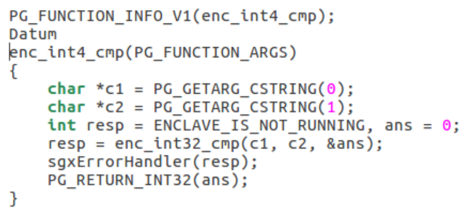
7.1 Implementation details

We implement StealthDB in C and C++ on top of Postgres 9.6 as an extension that loads new SQL functions, encrypted data types and operators and index support methods for the encrypted datatypes. The command *CREATE EXTENSION stealthdb* loads the files *stealthdb.so* (the main library), *enclave_stealthdb.so* (part of the code which is executed in enclaves), *stealthdb.control* (the version control file), *stealthdb.sql* (definitions of new defined functions) into the system. For instance, the function *enc_int4_cmp* in Figure 9 compares two *enc_int4* values and returns {-1, 0, 1}.



```
CREATE OR REPLACE FUNCTION enc_int4_cmp(enc_int4, enc_int4)
RETURNS integer
AS '$libdir/encdb'
LANGUAGE C IMMUTABLE STRICT;
```

Fig. 9. Example of a new function definition in stealthdb.sql



```
PG_FUNCTION_INFO_V1(enc_int4_cmp);
Datum
enc_int4_cmp(PG_FUNCTION_ARGS)
{
    char *c1 = PG_GETARG_CSTRING(0);
    char *c2 = PG_GETARG_CSTRING(1);
    int resp = ENCLAVE_IS_NOT_RUNNING, ans = 0;
    resp = enc_int32_cmp(c1, c2, &ans);
    sgxErrorHandler(resp);
    PG_RETURN_INT32(ans);
}
```

Fig. 10. Example of new defined function implementation in stealthdb.c

The function *enc_int4_cmp* in Figure 10 is executed in an enclave. We implement our query *pre-parser* in the PreProcessor enclave on the server side to encrypt the data values in queries and this design helps in avoiding changes to the client JDBC or ODBC drivers of the system. Our approach can be extended to other SQL-like database using user-defined functions. Though database systems like MySQL do not allow creating independent extensions like Postgres to include our changes, these changes are not intrusive and completely independent of the improvements to the core database operations. To protect against the side-channel attacks on SGX, we make every operation inside an enclave oblivious by leveraging AES-NI and CMOV instructions. The source code of Postgres 9.6 has about 700k lines of code while StealthDB has about 5k lines of code with 1.5k lines run in enclaves.

7.2 Performance evaluation

To measure StealthDB's performance, we use an Intel Xeon E3 3.60 GHz server with 8 cores and 16 GB of RAM. In our experiments, we measure the throughput and latency of StealthDB using the TPC-C trace and compare the results with an unmodified Postgres 9.6 which works with unencrypted data. The results were obtained by averaging multiple 1000 second runs with check-pointing turned off. We ran our experiments with the number of clients varying from 1 to 10 and with a single-threaded enclave used by all the client connec-

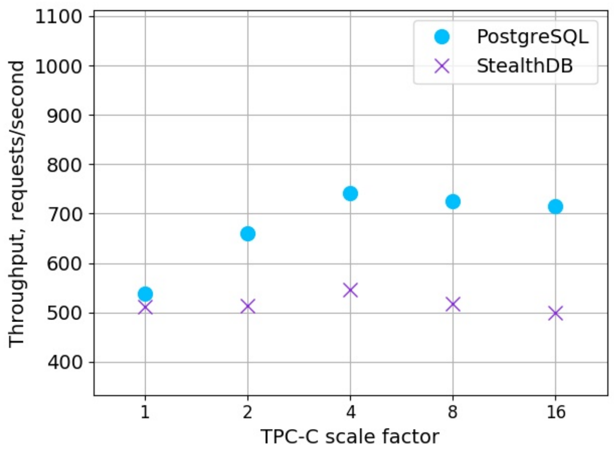


Fig. 11. TPC-C benchmarking throughput for running under Postgres and StealthDB with different scale factors

	Median	90th percentile
PostgreSQL	1.6	5.9
StealthDB	2.8	7.2

Table 1. Latency statistics of TPC-C requests, ms

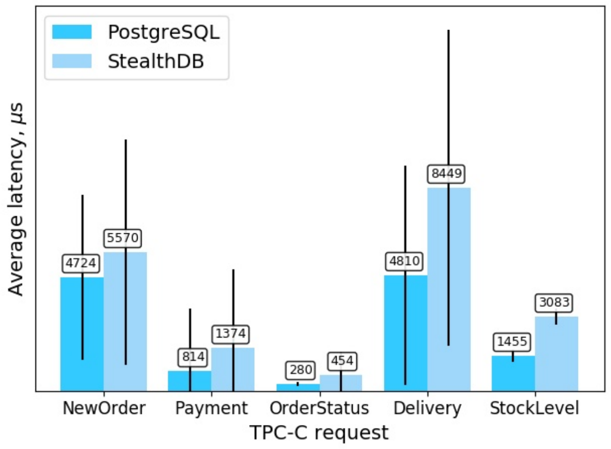


Fig. 12. Average latency and standard deviation for TPC-C requests under Postgres and StealthDB.

tions. The number of clients can be further increased if a multi-threaded enclave is used. Our first set of experiments leave the IDs in the TPC-C tables (e.g. w_id, o_w_id, etc.) unencrypted. The tested database includes nine tables with about 10 million rows in total. This is about 2GB of unecrypted data and when encrypted for StealthDB gives an encrypted database of size 7GB.

Throughput

Figure 11 shows the throughput for the TPC-C benchmarking for different scale factors. StealthDB incurs an 4.7% overhead over the unmodified Postgres for the scale factor $W = 1$ and around 30% overhead for $W = 16$. This is sufficient for many real-world transactional systems for the security advantages.

Latency

We measure the end-to-end TPC-C transaction latency for StealthDB with the scale factor $W = 16$. This includes the time for our query pre-parser.

Table 1 and Figure 12 compare the median and average latency for StealthDB with the unmodified Postgres. The 90th percentile of the latency of StealthDB system is 7.2 milliseconds which results in a 22% overhead over the unmodified version.

We also test the performance of StealthDB when the IDs are encrypted with AES-CTR. That results in about 3x throughput decrease over StealthDB with unencrypted IDs. And the latency is 3.6 times of that of the version with unencrypted IDs. The IDs in the TPC-C tables are just counters, hence encrypting them do not offer any concrete security advantages.

8 Related Work

This section builds on the comparisons from the introduction. The work most similar to ours is Cipherbase [2]. But the trusted on-premise key loading phase for every FPGA device, and cloud operator controlled “shell” monitor [1] inside an FPGA make FPGAs unsuitable for being used as a *trusted hardware* in the cloud. In terms of performance, [2] achieves about 10% better throughput than ours, but they skip two TPC-C transactions in their evaluation. Our evaluation with the complete TPC-C benchmark finds that these two transactions have the highest latency overheads. Similar bottlenecks are expected for Cipherbase with FPGAs. And, as expected, we achieve much lower latency (4×) over the FPGA implementation. TrustedDB [4] uses the IBM secure co-processor to perform operations, but with large portions of the DBMS engine executed inside the trusted zone. The IBM co-processor incurs high overheads for transactional workloads and also, this design is not suitable for SGX for both security and performance reasons as we discussed in Section 4.2.

CryptDB [47] uses a hybrid of encryption schemes to support subset of SQL functionality. Their underlying large leakage profiles often result in data compromise [29, 41]. Performance-wise, [47] achieves a similar throughput decrease as ours, but only when evaluated with the individual queries from the TPC-C transactions over a 20× smaller dataset. Arx replaces OPE scheme with a special garbled-circuit based searching method [46]. Garbled circuits however introduce large computational and storage overheads.

A few works studied how to build versions of encrypted databases with SGX. VC3 system proposes an architecture for analytical MapReduce jobs in cloud settings [51]. Opaque studies how to leverage SGX to secure distributed analytical workloads in Spark systems [56]. A concurrent work of ours, ObliDB [18], obtains an oblivious database supporting both transactional and analytical workloads. But, their solution involves extensive changes to the underlying DBMS engine, and does not scale well for transactional workloads. Another concurrent work, EnclaveDB [49], provides strong security guarantees against persistent and active adversaries. However, this is achieved by placing larger components of DBMS inside enclaves assuming the existence of large enclaves, in the order of gigabytes, which is much greater than the 128 MB available today.[5] They also ignore the access pattern and other side-channel attacks. In summary, [49] focuses on a different design space assuming how future trusted hardware designs may look, while our work focuses on building encrypted database from standard trusted hardware available today.

HardIDX [21] investigates how to perform range queries obliviously over B+ tree indexes inside an enclave, leaking only the parts of the database accessed per query. But, they only consider a static database, and the client should generate the full B+ tree index locally and store it in the server only for the querying. We can incorporate their ideas in StealthDB if we were to only support static databases and powerful clients. Also, [21] just prototypes index searches, whereas we architecture and build a complete encrypted database system.

A number of works study how to load unmodified applications into enclaves [3, 6, 33, 53]. These approaches work well for applications that process small data sizes, but do not scale well to larger workloads due to SGX limitations. Also, increasing the complexity of the codebase inside the enclaves aggravates the security risks associated with SGX [36].

OSPIR-OXT [10, 11, 19], SisoSPIR [34] and BLIND SEER [45] build encrypted database systems from scratch with provable security guarantees for a subset of functionality based on different cryptography tools. There are also multitude of other works which provide improvements over security or specific functionalities of a database, but they are not implemented or integrable with a mature DBMS. A recent systematization work by Fuller et al. [22] provides are great summary of the state-of-art research in encrypted database systems. Fully homomorphic encryption [24] is another powerful cryptographic primitive which enables an untrusted user to perform arbitrary computations on encrypted data without learning any information about the underlying data. But the current constructs for doing this are very far from being practical [32]. In general, while theoretical security of systems built based on cryptographic methods can be high, the *real-world* security of the system relies on the multitude of factors: correct implementations of non-trivial crypto algorithms, meta-data contents, information in log files, etc. Hence, it is not possible to argue their security just from the security of the crypto protocols used.

9 Conclusion

StealthDB offers a scalable encrypted cloud database system with full SQL query support with a modest 30% throughput decrease and ≈ 1 ms latency increase while providing strong end-to-end security guarantees. StealthDB can be implemented in any newer generation Intel CPUs. Supporting analytical workloads, reducing the leakage profile and protecting against active adversaries (i.e., providing integrity to the system) while maintaining our design principles are interesting open questions in this space. The source code of our implementation is also open-sourced.

5 It is an open question to achieve larger enclaves efficiently while providing security against physical attacks. SGX enclaves use Merkle-trees for integrity which adds logarithmic overhead to every access.

References

[1] Amazon. AWS shell interface specification. https://github.com/aws/aws-fpga/blob/master/hdk/docs/AWS_Shell_Interface_Specification.md, 2017. Accessed: 2017-10-01.

[2] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *CIDR*, 2013.

[3] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyers, R. Kapitza, P. R. Pietzuch, and C. Fetzer. SCONE: secure linux containers with intel SGX. In *OSDI*, pages 689–703, 2016.

[4] S. Bajaj and R. Sion. Trusteddb: A trusted hardware based database with privacy and data confidentiality. In *SIGMOD*, pages 205–216, 2011.

[5] M. Balduzzi, J. Zaddach, D. Balzarotti, E. Kirda, and S. Loureiro. A security analysis of amazon's elastic compute cloud service. In *SAC*, pages 1427–1434, 2012.

[6] A. Baumann, M. Peinado, and G. C. Hunt. Shielding applications from an untrusted cloud with haven. In *OSDI*, pages 267–283, 2014.

[7] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT*, 2017.

[8] S. Bugiel, S. Nürnberger, T. Pöppelmann, A. Sadeghi, and T. Schneider. Amazonia: when elasticity snaps back. In *CCS*, pages 389–400, 2011.

[9] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018.

[10] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS*, 2014.

[11] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO I*, pages 353–373, 2013.

[12] D. Cash and S. Tessaro. The locality of searchable symmetric encryption. In *EUROCRYPT*, pages 351–368, 2014.

[13] C. che Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *USENIX ATC*, pages 645–658, 2017.

[14] V. Costan and S. Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[15] F. Dall, G. D. Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom. Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):171–191, 2018.

[16] V. data breach incident report. https://regmedia.co.uk/2016/05/12/dbir_2016.pdf, 2016.

[17] M. Dzulfakar. Advanced mysql exploitation. Black Hat Las Vegas, 2009.

[18] S. Eskandarian and M. Zaharia. An oblivious general-purpose SQL database for the cloud. *CoRR*, abs/1710.00458, 2017.

[19] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich queries on encrypted data: Beyond exact matches. In *ESORICS II*, pages 123–145, 2015.

[20] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov. IRON: functional encryption using intel SGX. In *CCS*, pages 765–782, 2017.

[21] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A. Sadeghi. Hardidx: Practical and secure index with SGX. In *DBSec*, pages 386–408, 2017.

[22] B. Fuller, M. Varia, A. Yerukhimovich, E. Shen, A. Hamlin, V. Gadepally, R. Shay, J. D. Mitchell, and R. K. Cunningham. Sok: Cryptographically protected database search. In *IEEE SP*, pages 172–191, 2017.

[23] T. Garfinkel and M. Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *HotOS*, 2005.

[24] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[25] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[26] Google. Encrypted BigQuery client. https://github.com/google/encrypted-bigquery-client, 2017.

[27] P. Grofig, I. Hang, M. Härterich, F. Kerschbaum, M. Kohler, A. Schaad, A. Schröpfer, and W. Tighzert. Privacy by encrypted databases. In *Annual Privacy Forum*, pages 56–69. Springer, 2014.

[28] P. Grubbs, M. Lacharité, B. Minaud, and K. G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *CCS*, pages 315–331, 2018.

[29] P. Grubbs, R. McPherson, M. Naveed, T. Ristenpart, and V. Shmatikov. Breaking web applications built on top of encrypted data. In *ACM CCS*, pages 1353–1364, 2016.

[30] P. Grubbs, T. Ristenpart, and V. Shmatikov. Why your encrypted database is not secure. In *HotOS*, pages 162–168, 2017.

[31] B. D. A. Guimaraes. Advanced sql injection to operating system full control. Black Hat Europe, 2009.

[32] S. Halevi and V. Shoup. Algorithms in helib. In *CRYPTO I*, pages 554–571, 2014.

[33] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *OSDI*, pages 533–549, 2016.

[34] Y. Ishai, E. Kushilevitz, S. Lu, and R. Ostrovsky. Private large-scale databases with distributed searchable symmetric encryption. In *CT-RSA*, pages 90–107, 2016.

[35] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. Generic attacks on secure outsourced databases. In *CCS*, pages 1329–1340, 2016.

[36] J. Lee, J. S. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security*, pages 523–539, 2017.

[37] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, pages 557–574, 2017.

[38] K. Lewi and D. J. Wu. Order-revealing encryption: New constructions, applications, and lower bounds. In *CCS*,

pages 1167–1178, 2016.
[39] 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 *HASP*, page 10, 2013.
[40] Microsoft SQL Server 2016. Always encrypted database engine. https://msdn.microsoft.com/en-us/library/mt163865.aspx, 2017.
[41] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *ACM CCS*, pages 644–655, 2015.
[42] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, pages 619–636, 2016.
[43] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. Eleos: Exitless OS services for SGX enclaves. In *EuroSys*, pages 238–253, 2017.
[44] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan. Big data analytics over encrypted datasets with seabed. In *OSDI*, pages 587–602, 2016.
[45] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. D. Keromytis, and S. M. Bellovin. Blind seer: A scalable private DBMS. In *IEEE SP*, pages 359–374, 2014.
[46] R. Poddar, T. Boelter, and R. A. Popa. Arx: A strongly encrypted database system. *IACR Cryptology ePrint Archive*, 2016:591, 2016.
[47] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.
[48] PostgreSQL 9.5.10 Documentation. Extensions. https://www.postgresql.org/docs/9.5/static/external-extensions.html, 2018. Accessed: 2018-01-29.
[49] C. Priebe, K. Vaswani, and M. Costa. Enclavedb: A secure database using SGX. In *IEEE SP*, pages 264–278, 2018.
[50] T. Ristenpart and S. Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *NDSS*, 2010.
[51] 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 *IEEE SP*, pages 38–54, 2015.
[52] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *DIMVA*, pages 3–24, 2017.
[53] C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. Cooperation and security isolation of library oses for multi-process applications. In *EuroSys 2014*, pages 9:1–9:14, 2014.
[54] N. Weichbrodt, A. Kurmus, P. R. Pietzuch, and R. Kapitza. Asyncshock: Exploiting synchronisation bugs in intel SGX enclaves. In *ESORICS I*, pages 440–457, 2016.
[55] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE SP*, pages 640–656, 2015.
[56] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, pages 283–298, 2017.

Acknowledgements

S.G. was supported by grants from NSERC and University of Waterloo. The authors would like to thank the reviewers and the shepherd for their great comments and suggestions on improving the quality of the paper. D.V. would also like to thank Hemant Saxena for discussions on the internals of Postgres..

A Security addendum

A.1 Formal definition of security

Figure 13 provides the formal simulation security definition for an encrypted database system using trusted hardware definition. This definition is inspired by [20] who define simulation security for functional encryption using trusted hardware HW. An EncDB construction is secure if, for all admissible adversaries, there exists an efficient Sim such that:

$$|\Pr[\mathsf{Adv}(\mathsf{Real}_{\mathsf{EncDB}}) = 1] - \Pr[\mathsf{Adv}(\mathsf{Ideal}_{\mathsf{EncDB}}) = 1]| < \mathrm{negl}(\lambda)$$

where $\mathsf{Adv} = (\mathsf{Adv}_1, \mathsf{Adv}_2)$. Adv_1 runs the Real or the Ideal experiment, whereas Adv_2 obtains information about the experiment from Adv_1 depending on the adversarial type being studied and produces the output 0 or 1. A snapshot Adv_2 obtains a snapshot of the system, when desired, from Adv_1, whereas a persistent Adv_2 completely observes the EncDB system while Adv_1 is running the experiment. Adv_1 is tasked with just running the EncDB system; a semi-honest Adv_1 will run as per the specifications, and an actively malicious Adv_1 will run the system as desired to maximize the information obtained by Adv_2. The access to HW is treated as an oracle as in [20] and Sim simulates the oracle in the Ideal experiment. The HW oracle provides interfaces to the enclaves used (in StealthDB, they are $\mathsf{Auth}()$, $\mathsf{PreProcessor}(\mathsf{encquery})$ and $\mathsf{Ops}(\{\mathsf{input}\}, \mathsf{op})$). When Query is invoked on a query, Sim will obtain the leakage $\mathcal{Q}$ corresponding to a query from $\mathcal{L}$.

$\mathsf{Real}_{\mathsf{EncDB}}(1^\lambda)$:	$\mathsf{Ideal}_{\mathsf{EncDB}}(1^\lambda)$:
$(K, \mathbf{EDB}) \leftarrow \mathsf{Init}(1^\lambda, \mathbf{DB})$	$\mathbf{EDB} \leftarrow \mathsf{Sim}^{\mathcal{L}}(1^\lambda)$
$\mathbf{encres} \leftarrow \mathsf{Query}^{\mathsf{HW}(\cdot)}(\mathbf{EDB}, \mathbf{encquery})$	$\mathbf{encres} \leftarrow \mathsf{Query}^{\mathsf{Sim}^{\mathcal{L}(\cdot)}(\cdot)}(\mathbf{EDB}, \mathbf{encquery})$

Fig. 13. Security definition for an encrypted database system using trusted hardware.

A.2 Security of K during StealthDB execution

Outline. We will argue here that no information about the master key K is revealed; also that only the *permitted* clients can make the DBMS execute queries. This will be a precursor to the leakage profile analysis in Section 6.1

Claim A.1. *The confidentiality and integrity of the master key* K *is ensured throughout the StealthDB execution.*

The database owner forms the *root of trust* as in Figure 7. The owner is involved in a remote attestation protocol with Auth to check the correctness of the code and the constants loaded into Auth against the publicly available *expected measurement* of Auth. (The constants loaded into Auth include the expected measurements of PreProcessor and Ops). The master credentials for the database is transferred to a valid Auth. And, the security of SGX remote attestation guarantees the validity of Auth. From this point, the trust is transferred to Auth. Auth generates the master key K.

The master K is then transferred to the other enclaves PreProcessor and Ops by Auth through the secure channels established on top of local attestation. The security of local attestation ensures that Auth establishes secure channels with only those PreProcessor and Ops whose measurements match the *expected* hardcoded ones. Hence, K is transferred only to the correct instances of PreProcessor and Ops. Here, the confidentiality and integrity provided by the secure channel ensure that no information about K except its length is leaked to an adversary during the transfers.

Now, there are only two more operations which involve K. First, when K is used to AES encrypt and decrypt data values, the SGX security guarantees combined with the use of a data-oblivious implementation of the AES-NI instructions ensure that no intermediate values about K are leaked. Finally, K is also sealed and stored on the disk for later retrieval. Here, the SGX sealing process provides confidentiality and integrity to K.

Claim A.2. *During the query execution phase, a query which reaches the DBMS for execution satisfies the access control policies for the client requesting the query.*

The security of remote attestation also ensures that the database owner transfers the client credentials database only to a valid Auth. When a client proxy initiates a connection with the DBMS, a valid Auth establishes a session with the client only if the client has valid credentials. Next, Auth transfers the session key sessk (shared with the client) only to a valid PreProcessor. This is ensured by the security of local attestation. Now, when the client issues a query, the I/O layer relays it to PreProcessor and PreProcessor parses the query and proceeds only if the query satisfies the access policies of this client. Since there is no other interface for the client to issue a query to the semi-honest DBMS, StealthDB ensures that the semi-honest DBMS only executes a query from a valid client satisfying the access policies provided by the database owner.

A.3 Correctness of Claim 6.2

Adv_2 would query the snapshot of the system at time t. Sim sets up EDB as encryption of zeros of arbitrary shape $\mathcal{S}_0$ and answers the Ops queries arbitrarily till t'. At time t', Sim obtains $\mathcal{S}_{t'}$ from $\mathcal{L}$ and rewrites EDB with encryption of zeros according to $\mathcal{S}_{t'}$. For each query run between t' and t, Sim obtains $\mathcal{Q}$ from the oracle $\mathcal{L}$ and answers the Ops queries accordingly. This way, the execution of the Real and Ideal experiments and the corresponding shapes of EDB are consistent at time t assuming a deterministic order of execution for EDB.

We will now argue that the Real and the Ideal experiments are indistinguishable. When Adv_2 obtains the snapshot of the system at time t, it obtains EDB along with the logs and miscellaneous data structures maintained at time t. Given that the shape of EDB is consistent between the two experiments at time t, semantic security ensures that a real EDB is indistinguishable from the encryption of zeros. Logs, etc. for queries before time t' are encrypted and written to disk. Hence, they do not reveal any information about the data items in DB. The logs maintained in between t' and t are also consistent between the two experiments and are consistent.

If the logs and the other data structures are encrypted in memory, Sim can behave arbitrarily till t and just rewrite EDB according to $\mathcal{S}_t$ at time t. Following the assumption that the size of logs do not reveal sensitive information, the Real and the Ideal experiments are indistinguishable to Adv_2.

A.4 Correctness of Claim 6.3

We again give the high-level idea here. During Init, Sim obtains the shape $\mathcal{S}$ from the leakage oracle $\mathcal{L}$ and encrypts zeros as EDB according to $\mathcal{S}$. This EDB is indistinguishable from a real EDB by the semantic security of the encryption scheme. Further, during the execution of Query, the values in DB are only used inside the Ops enclave. With a deterministic execution of EDB, Sim uses $\mathcal{Q}$ obtained from $\mathcal{L}$ to answer the plaintext outputs. For the encrypted outputs, Sim produces encryption of zeros as Ops output and this is again indistinguishable from the encryption of the real values by the semantic security of the encryption scheme.

B Concrete leakage profiles

The discussion above provided an upper bound on the leakage in terms of abstract leakage entities. The definition of the shape $\mathcal{S}$ is concrete from the definition. But, $\mathcal{Q}$ and $\mathcal{M}$ depend on the underlying DBMS that StealthDB builds on. We will now concretize this for the different operations performed on encrypted data.

– *Arithmetic operations*: Some examples of arithmetic operators include +, -, %, * and advanced ones like sin, cos, log. For these operators, we provide the same security as a fully-homomorphic encryption (FHE) on the computation performed on individually encrypted data items. As in FHE, StealthDB does not reveal any information to a semi-honest adversary about the intermediate values of an arithmetic computation involving encrypted inputs and outputs, other than their length (as multiples of 128 for AES). Consider a simple example query from a TPC-C transaction: update table_warehouse set w_ytd = w_ytd + constant where w_id = constant2. StealthDB reveals no information about the values in the column w_ytd during the execution of this query.

– *String operations*: String operations like substring and wildcards have no leakage, other than the length of inputs and outputs (up to a multiple of 128), with them being encrypted.

– *Relational operations*: A real-world DBMS uses indexes to perform the relational operations like comparisons and joins efficiently. The $\mathcal{Q}$ for a query using an index, say a B-tree, includes the comparison results of the parts of the B-tree explored by the query. As the values in the index are re-encrypted versions of the values in the table, the comparison results are useful only when the corresponding values are accessed in the table. When a row becomes part of query results, an adversary can link it to the corresponding value in the index. From this, it can use the Ops output history to obtain the comparison results between the indexed value in this row with the indexed values from the other accessed rows. Hence, the information revealed by $\mathcal{Q}$ in StealthDB is the comparison results for indexed values in the rows accessed by the queries. In the worst case, our leakage against persistent adversaries reduces to the guarantees provided by ORE for the parts of the indexes explored by the queries.
There is also a non-trivial information leakage to a persistent adversary that only has access disk, and not memory. The index pages on disk that are modified during checkpointing reveal some inequalities within the data being inserted or modified. In Postgres, for instance, the index file stores data as 8 KB pages. When a new value is inserted into the table, only the pages that need to be changed are marked as dirty in the memory and eventually changed on disk.
For any other DBMS, the precise information revealed by $\mathcal{Q}$ varies based on its query execution and log maintenance procedures.