Azure SQL Database Always Encrypted

Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramaratnam Venkatesan, Mike Zwilling
Microsoft Azure and Microsoft Research

ABSTRACT

This paper presents *Always Encrypted*, a recently released feature of Microsoft SQL Server that uses column granularity encryption to provide cryptographic data protection guarantees. Always Encrypted can be used to outsource database administration while keeping the data confidential from an administrator, including cloud operators. The first version of Always Encrypted was released in Azure SQL Database and as part of SQL Server 2016, and supported equality operations over deterministically encrypted columns. The second version, released as part of SQL Server 2019, uses an *enclave* running within a *trusted execution environment* to provide richer functionality that includes comparison and string pattern matching for an IND-CPA-secure (randomized) encryption scheme. We present the security, functionality, and design of Always Encrypted, and provide a performance evaluation using the TPC-C benchmark.

ACM Reference Format:
Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramaratnam Venkatesan, Mike Zwilling. 2020. Azure SQL Database Always Encrypted. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3386141

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3386141

1 INTRODUCTION

There are many reasons to move workloads to the cloud. The cloud provides elasticity in a pay-as-you-go model which allows customers to pay only for computational resources that they really need. Furthermore, the cloud promises high availability and a fast time-to-market as additional computational resources can be provisioned within minutes as opposed to weeks or months in a non-cloud, on-premise setting.

Security and compliance are an important, if not the most important, driver for cloud adoption. Due to economies of scale, a large cloud provider can afford the best security technologies and experts that continuously evolve and apply the state-of-the-art. Specifically, the cloud provider can continuously patch the infrastructure whenever vulnerabilities become known and make large security investments because these are amortized over a huge computational infrastructure.

The cloud, however, also exposes additional attack vectors as the cloud provider itself or rogue employees of the cloud provider may compromise sensitive client data and cause breaches. Unlike a traditional on-premise deployment, the user of a cloud service has no control over system administrators who might have administrative privileges to the machines that host and process the data [5, 14, 20].

This paper presents *Always Encrypted* a unique feature of Microsoft SQL Server that provides confidentiality controls for databases in the cloud and on-premise. Current database systems provide sophisticated access control mechanisms [23, 29] and encryption support for data-at-rest, but they do not protect the data against attackers with administrative privileges on the database or on the server that hosts the database. Microsoft SQL Server is the first and to date only industrial-strength database system that provides this kind of protection.

A note on Microsoft SQL Server offerings: SQL Server is available in two related offerings: (1) as a traditional "box-product" that customers can deploy on infrastructure of their choice (private datacenter or a cloud VM); (2) as a cloud database *Platform-as-a-Service (PaaS)* offering called *Azure SQL Database*. Both offerings rely on the same codebase with relatively minor version and feature differences. In the

following, unless qualified otherwise, when we refer to SQL Server, we refer to both offerings.

1.1 Always Encrypted

Always Encrypted (AE) endows the database system with cryptographic data protection using encryption. AE allows data owners to use encryption at a column granularity to outsource database administration while keeping data confidential from the administrators, including cloud operators.

Operationally, these security guarantees are achieved by keeping data identified as sensitive, encrypted at all times: *at rest* when stored on disk, in SQL Server's internal memory while *in use* (except within the memory of Trusted Execution Environments (TEEs) as we will see), and *in transit* during backups and result communication. SQL Server is *untrusted* for these guarantees and they continue to hold if the SQL Server instance is compromised. Encryption keys are generated by clients in a "bring-your-own-key" model and they are never exposed to SQL Server. This property of AE is the central difference as compared to traditional data protection mechanisms such as *transparent data encryption (TDE)* [22, 27, 30, 31] and role-based access controls [23, 29], where the database system needs to be trusted for the protections. TDE, for example, keeps data encrypted at rest but decrypts it when loaded into memory during query processing. The TDE design requires the database system to hold encryption keys, so a malicious administrator can recover the keys and data using simple memory scraping attacks.

The fundamental challenge introduced by data encryption is computation over ciphertext. One approach to address this challenge is to use specialized encryption schemes that allow computation over ciphertext. The first version of AE (*AEv1*) uses *deterministic encryption* which, as the name suggests, deterministically produces the same ciphertext for a given plaintext. This property allows equality operations over ciphertext and AEv1 relies on this to support database operations based on equality such as point lookups, equijoins, and equality-based grouping. Functionality restrictions aside, this approach suffers from a serious usability painpoint: since SQL Server does not have the encryption key, turning on encryption for the first time (*initial encryption*) and rotating encryption keys both require a roundtrip to client systems possessing the encryption key(s), which can be prohibitively expensive. For terabyte large databases, this roundtrip can result in latencies as long as a week for initial encryption and key rotation, which was a nonstarter for many customers with such large databases.

The main innovation of the second version of Always Encrypted (*AEv2*) released as part of SQL Server 2019 (and soon on Azure SQL Database) is to use a *trusted execution environment (TEE)* to address some of the functionality and usability restrictions of AEv1. A TEE is an emerging security technology that provides a way for a small amount of trusted code called an *enclave* to be run as part of a larger untrusted host process. The TEE hides the enclave computation and state from the host process and host OS and therefore, administrators of the host system. AE currently supports Windows *Virtualization-based Security (VBS)* enclaves [35]. We are also working on supporting Intel SGX enclaves [12]. (In the following, unless qualified otherwise, AE refers to the latest v2 version which also includes all of the v1 functionality as discussed in Section 2.)

AE uses the TEE to temporarily store encryption keys and perform computations on decrypted, plaintext data. While conceptually simple, this approach introduces significant engineering and technical challenges: (1) We need additional services to *attest* the trustworthiness of the enclave code and such attestation services need to work for a variety of SQL Server deployment scenarios (on-premise, Azure, other clouds); (2) TEE raises the question of how query processing is divided between the trusted enclave and untrusted SQL Server? The simple strawman of pushing all of SQL Server querying functionality into the enclave inherits any vulnerabilities in the large SQL Server code base. There are also subtle *information leakage* attacks where an attacker can learn plaintext information from data movement patterns to and from the enclave; (3) There are *devops* challenges such as how to handle failures within the enclave and how to debug customer errors originating within the enclave while respecting customer data confidentiality requirements.

While we subscribe to a northstar goal of supporting most of SQL functionality, AEv2 represents a first step towards richer querying using TEEs. In AEv2, we support general comparisons (beyond equality) and string pattern matching operations. In addition, initial encryptions and key rotations go through the TEE and avoid the roundtrip to client systems. Accordingly, AE today is not designed to be applicable to the entire database, but to high-sensitivity columns, e.g. personally identifiable identifiers such as social security numbers, credit card numbers, names, and addresses.

1.2 Customer Impact and Experiences

Despite its functionality limitations, AEv1 is used by a wide variety of customers ranging from financial institutions (e.g., Financial Fabric, Produbanco) to insurance companies (e.g., Progressive Insurance) and health care organizations (e.g., Fullerton Health Care). These customers use AE mostly for OLTP applications and encrypt only *personally identifiable identifier (PII)* columns such as SSNs, names, email addresses, and credit card numbers.

The AEv2 feature was designed based on feedback from these and other customers. Customer applications suggest

that many types of sensitive information such as names, phone numbers, and location data require richer operations going beyond equality. Further, the data sizes are large enough to make the client-based initial encryption and key rotation impractical. Although only recently released, we are seeing an increase in interest from a broader set of customers including those in manufacturing and retail sectors who feel that Always Encrypted will simplify their ability to meeting regulatory compliance requirements, such as the EU General Data Protection Regulation (GDPR).

1.3 Related Work

AE builds on a rich body of research in the area of encrypted database systems. Without this foundational work, it would not have been possible to build AE. While all that work is relevant, it took many additional innovations to build AE as a full-fledged commercial relational database system with end-to-end data protection guarantees using encryption. AE is also one of the first commercial server-side platforms to leverage emerging TEE technology.

The idea of using encryption for data protection for cloud outsourcing was first proposed by Hacigumus et al. [16]. The main challenge of using encryption is query processing over data obscured by encryption. All prior research prototypes [4] use some combination of *homomorphic encryption* that allows computation over ciphertext and TEE-based processing.

Fully homomorphic encryption (FHE) schemes that allow arbitrary computation are inefficient for database querying, significant advances [10, 17, 28] in recent years notwithstanding. FHE also works with fixed-size inputs and outputs and therefore suffers from an *abstraction mismatch* with database querying where the input and output sizes can vary arbitrarily. Prior systems [15, 24, 26, 33] have relied on specialized encryption schemes such as *property-preserving encryption* that preserve some plaintext property such as order when encrypted, *partial homomorphic encryption* that support limited operations such as addition over ciphertext but with better performance characteristics than FHE, and *garbled circuits*. Many industrial NoSQL systems such as Google Encrypted BigQuery [15] and Ciphercloud [11] rely on such specialized encryption schemes as well.

The first research prototype based on TEEs was TrustedDB [7] which uses a coarse-grained approach of running a full query processing stack within the TEE. The fine-grained architecture of Always Encrypted was first described in [2, 3]. AE adopted and significantly expanded that design and showed for the first time that it can be deployed commercially and at scale in the cloud and on premise. Recent work [13, 36] has focused on expensive Oblivious RAM techniques for TEEs to hide access patterns going beyond the operational security provided by AE.

2 OVERVIEW

In this section, we discuss how Always Encrypted is configured, the query functionality that it provides and introduce the notion of an enclave.

2.1 Enclave

An *enclave* is a part of the virtual address space of a process and includes code and data that is shielded from the rest of the process and the operating system (OS), and hence from actors with administrative privileges to the machine. We use the term *host* to refer to part of the process outside the enclave. The host initializes the enclave by loading a specially compiled dynamically linked library (dll) and invokes the enclave code using function calls. The enclave code can access the entire address space of the process while by design the host cannot access the enclave memory.

AE currently supports software based enclaves using *Windows Virtualization-based Security (VBS)* [35]. We are in the process of adding support for *Intel SGX* [12]. For SGX, the enclave is protected by the CPU, while for VBS, the protection comes from the hypervisor (Windows Hyper-V). This implies that the Intel processor is a trusted component for SGX enclaves and Hyper-V (and the underlying processor), a trusted component for VBS enclaves. A detailed discussion of enclaves is beyond the scope of this paper and we refer the reader to these papers [12, 35].

Enclave platforms support a protocol called *attestation* using which a remote system (e.g., client) can verify the authenticity of the initial code and data in an enclave. This protocol relies on a trusted external service called the *attestation service*. We describe the details of attestation as used in Always Encrypted in Section 4.2.

2.2 Encryption Keys

AE uses a two-level key hierarchy to encrypt data. Data is encrypted using symmetric encryption based on a *column encryption key (CEK)*, a 32 byte AES key. A CEK is stored within the database encrypted using a second-level key called the *column master key (CMK)*. A CMK is stored in a separate *key provider* and SQL Server AE stores only a URI reference to the key in the key provider, without having access to the key material. Since all key metadata is stored in SQL Server, except for the CMK, this ensures that the database is the single source of truth, and that key metadata is replicated and backed up along with SQL Server data. The client controls and configures the key provider(s) used in an AE instance. We support the following key providers out of the box: Azure Key Vault [6], Windows Certificate store, Java Key Store, and

```
CREATE COLUMN MASTER KEY MyCMK WITH (
 KEY_STORE_PROVIDER_NAME = N'AZURE_KEY_VAULT_PROVIDER',
  KEY_PATH = N'https://vault.azure.net/...',
  ENCLAVE_COMPUTATIONS (SIGNATURE = 0x6FCF...))

CREATE COLUMN ENCRYPTION KEY MyCEK
  WITH VALUES (COLUMN_MASTER_KEY = MyCMK,
  ALGORITHM = 'RSA_OAEP', ENCRYPTED_VALUE = 0x0170...)

CREATE TABLE T(id int,
 value int ENCRYPTED WITH
 (COLUMN_ENCRYPTION_KEY = MyCEK,
  ENCRYPTION_TYPE = Randomized,
  ALGORITHM = 'AEAD_AES_256_CBC_HMAC_SHA_256') )
```

Figure 1: SQL DDL to specify a column master key (CMK), column encryption key (CEK) and column encryption

Key Stores rooted in Hardware Security Modules (HSMs). We also have an extensible interface that lets customers plug in key providers of their choice.

Figure 1 shows the DDL statement for provisioning a CEK and a CMK. The illustrated CMK refers to a key stored in Azure Key Vault [6], specified using the Key_Store_Provider_Name attribute. The Key_Path property specifies the URI of the key. We allow enclave computations to be configured at the granularity of a CMK; it follows that all CEKs encrypted with the above CMK are permitted to be sent to SQL Server's enclave. We call a CMK *enclave-enabled* if it is configured to allow enclave computations. We call a CEK *enclave-enabled* if its corresponding CMK is enclave-enabled. We call non-enclave-enabled keys as *enclave-disabled*. We sign the CMK metadata with the key material in the CMK (specified in the Signature property) to prevent SQL from tampering with it; without this protection, SQL Server can alter the metadata to use a CEK in an enclave even when a client disallows such use. The DDL statement used to provision a CEK specifies the CMK used to encrypt it, the encrypted CEK, and a signature to protect the encrypted CEK. Our DDL syntax is designed to be extensible: for example, while we currently only support RSA-based encryption with OAEP padding for CEKs, we require the DDL statement to explicitly specify the `Algorithm` as `RSA_OAEP`. This design allows potential future extensions to other encryption algorithms.

2.3 Data Encryption

Always Encrypted is a column-level encryption feature. The encryption configuration of a column consists of an *encryption scheme*, the encryption and decryption algorithms, and a CEK used to encrypt values in the column, i.e. encryption of data is at the *cell* level. Figure 1 shows an example of a

AcctID	AcctBal	Branch
1	100	Seattle
2	200	Seattle
3	200	Zurich

AcctID (plaintext)	AcctBal (RND)	Branch (DET)
1	0xa24	0x123
2	0x3b7	0x123
3	0xdf2	0x363

Figure 2: Example Plaintext and Encrypted Database

`CREATE TABLE` DDL that specifies an encrypted column. Always Encrypted supports two encryption schemes discussed below:

1. *Deterministic (DET)*: In deterministic encryption, there is no randomization during encryption, so there is a one-to-one correspondence between ciphertext and plaintext. Informally, it is AES CBC mode encryption with SHA hash of the plaintext as the Initialization Vector (IV). We remark that our approach is more secure than using the ECB mode of AES. The ECB mode of AES is deterministic at the level of plaintext *blocks*; if a block is repeated, then the corresponding ciphertext blocks would be identical. On the other hand, our implementation of deterministic encryption only preserves equality at the level of the whole value.

Deterministic encryption allows the database engine to evaluate an equality of values using their ciphertexts but it weakens confidentiality since it leaks the frequency distribution of values in the column. We refer to columns encrypted with deterministic encryption as *DET* columns.

2. *Randomized (RND)*: Randomized encryption uses the standard Cipher Block Chaining (CBC) mode of AES encryption with a random Initialization Vector (IV). If the CEK is enclave-disabled, then no operations are supported over a RND column, so queries can reference this column only in the `SELECT` clause to fetch it as part of the result. If the CEK is enclave-enabled, then we do support equality, range and pattern matching queries (LIKE predicate) on the column using the enclave for computation.

Both modes of encryption also include an HMAC per encrypted value. While AE makes no guarantees about data integrity, we use HMACs as a usability feature; absent HMACs, there is no way for a client to tell apart legitimate ciphertext from garbage, which could lead to undetectable data corruption, for instance if a client erroneously inserts random byte sequences as ciphertext.

Notice that the table creation DDL specifies an encryption algorithm explicitly. Just like CEK encryption, our data encryption algorithm is also extensible. While we only support AES based encryption as described above today, we could extend them to newer algorithms in the future.

Figure 2 illustrates columnar encryption for an example instance of the Account table of the TPC-C benchmark. The

table on the left is the plaintext, and the one on the right is the corresponding encrypted table using a policy that specifies that the AcctID column is stored in plaintext, the AcctBal column is encrypted using randomized encryption, and the Branch column is deterministically encrypted.

2.4 Functionality

AE maintains encryption end-to-end. The client issues encrypted queries, i.e. queries where the input parameters are encrypted and SQL returns encrypted results (we discuss application transparency in Section 2.5.)

2.4.1 **Key Provisioning.** While we provide DDL to create key metadata, our DDL expects clients to configure the CMK and compute the encrypted value of CEKs. In order to ease the burden for clients, we automate the above steps in our tools.

2.4.2 **Data Encryption.** To turn on encryption (*initial encryption*) for a column using an enclave-enabled CEK, we rely on an `ALTER TABLE ALTER COLUMN` DDL statement. To encrypt columns using an enclave-disabled CEK, we need a roundtrip to a client system, and we provide client-side tools for this purpose.

An important operation with encryption is *key rotation*. A CMK rotation does not require re-encrypting data, but merely CEKs encrypted with it. To prevent downtime we allow CEKs to be encrypted with two CMKs temporarily for the duration of the CMK rotation. A CEK rotation does require re-encrypting data. Just like initial encryption, we use an `ALTER TABLE ALTER COLUMN` DDL statement when both prior and new-CEKs are enclave-enabled; otherwise, we rely on client-side tools to manage the client round-trip. All operations discussed here are *online*, meaning the client sees no downtime during key rotation or initial encryption.

2.4.3 **Select and Update Queries.** AE restricts querying functionality for encrypted columns. For columns encrypted with enclave-disabled keys, AE supports only equality operations on DET columns, i.e. point lookups, equi-joins and equality grouping, and no scalar operations on RND columns. For columns with enclave enabled keys, AE uses the enclave to support equality, range comparisons, and `LIKE` pattern-matching predicates, even for RND columns.

2.4.4 **Indexing.** On DET columns, we support point indexing. On RND columns with enclave-enabled keys, we also support range indexing using SQL Server's B+-Trees. Range indexing is not supported on deterministically encrypted columns, where enclave-enabled keys can only be used for in-place encryption and key rotation; we see deterministic encryption strictly as a way to support equality based comparisons.

2.5 Application Transparency

Always Encrypted is designed for *application transparency* meaning applications do not need to be modified to use AE functionality, modulo the functionality restrictions above and some fine-print discussed next. In order to achieve transparency, we only support parameterized queries (including stored procedures and functions). This is not a serious functional limitation since good programming practices already recommend parameterization, and any ad hoc query can be rewritten to be parameterized. We achieve transparency by enhancing the SQL client drivers to be aware of AE, such that (a) query parameters are encrypted on their way to SQL, and (b) results are decrypted when returned to the application. When a query requires computations in the enclave, then the driver also transparently sends CEKs to the enclave.

2.6 Threat Model

Our goal is to ensure data confidentiality from entities with privileged OS and database access. To characterize this threat, we introduce the *strong adversary*, who has unbounded power over the SQL Server process and can not only view the contents of the server's memory/disk at every instant, along with all external and internal communication, but also tamper with it, for instance by attaching a debugger to SQL. A strong adversary however cannot observe state or computations within the enclave because it is specifically designed for this purpose. As an emerging technology, TEEs are undergoing an arms race between side channel attacks and the corresponding patches [34]. A similar dynamic of hardware attacks has been observed in the past and continues to be observed in the development of Hardware Security Modules [19]. Current enclave side channel attacks are specific to the TEE implementations, not the promised security goals. The design of AE is not dependent on a specific TEE implementation allowing us to transition to a more secure implementation if necessary. We therefore exclude enclave side-channel attacks from our scope. Our adversary is stronger than the *honest-but-curious* adversary assumed in most prior work [3, 7, 15, 24, 26, 33], who can only observe, but not tamper with the processing.

3 ARCHITECTURE

Figure 3 describes the architecture of Always Encrypted. We show the SQL Server component shaded to illustrate that it is untrusted. All other components—the SQL client, the enclave and the attestation service are trusted. The keys visible in the respective trusted components are also illustrated—the key provider stores the CMK, and the CEKs are visible in the client driver and the enclave. Data is stored encrypted at column granularity within SQL Server; Figure 3 shows encrypted data corresponding to the schema introduced in

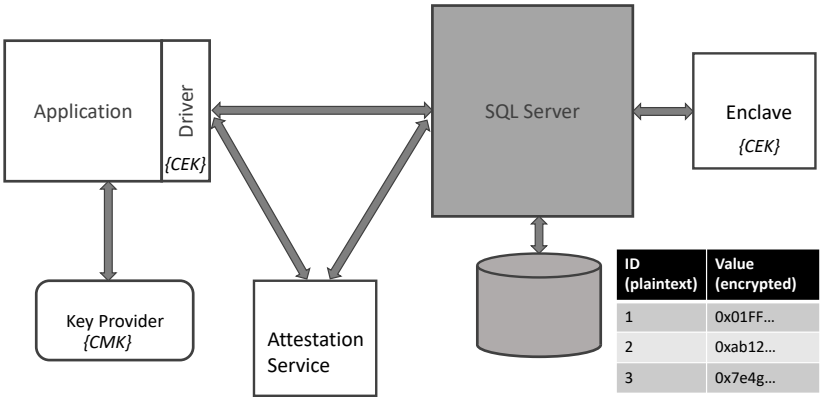

Figure 3: Architecture of SQL Server Always Encrypted

Figure 1. SQL Server also stores CEKs encrypted with the corresponding CMKs.

Since encryption is transparent, the application issues a parameterized query with plaintext parameters and expects results in plaintext. The driver therefore needs to deduce the encryption "type" information, the encryption keys to encrypt parameters and decrypt results. Implementing this functionality entirely within the driver is a substantial engineering challenge since it involves duplicating full SQL Server parsing in the driver. Further, this step also requires access to metadata stored in SQL Server for "binding", attaching semantic interpretation to the parsed query. Our design instead relies on implementing the encryption type deduction functionality within SQL Server and making it available to the driver using a new api called `sp_describe_parameter_encryption`. The output of this call for a parameterized query contains: the encryption type information (CEK) for each parameter; if the evaluation of the query requires an enclave, the output also contains the set of CEKs required within the enclave. For each CEK above, the output contains the encrypted CEK and the CMK metadata. Since SQL Server is untrusted, it could return incorrect output for the `sp_describe_parameter_encryption` call and undermine security. We discuss client controls that mitigate this risk in Section 4.1.

If the query requires enclave computations, then SQL Server also makes a call to a trusted *attestation* service and returns attestation information to the client, which is included in the above output. The attestation information if returned is used to establish a shared secret between the driver and the enclave. Overall, in our approach, SQL Server acts as the untrusted "man-in-the-middle" mediating communication between the driver and the enclave.

The driver uses the output of `sp_describe_parameter_encryption` to obtain CMKs, decrypt CEKs, and issue a query to SQL Server with encrypted parameters. If the query evaluation requires an enclave, the driver installs the necessary (decrypted) CEKs in the enclave using the shared-secret based secure channel. SQL executes the query (using the enclave if needed) and returns encrypted results to the client, along with key metadata needed to decrypt the results. The driver decrypts the results and presents them in the clear to the application.

We now briefly outline how query execution inside SQL Server uses the enclave. The SQL Server engine implementation of AE inherits many of the design elements from Cipherbase [2]. Our design relies on the observation that most components of a database engine do not directly compute on column-granularity data values, but rather move or copy data values between different locations (disk, buffer pool, log, and lock table). Their functionality is unaffected whether the values are encrypted or in plaintext. SQL Server code localizes all computations on columnar data values to a module called *expression services (ES)*. Our enclave runs a subset of ES needed to implement the functionality that we support. While we have made changes outside the enclave, those changes are not extensive.

For example, consider the data shown in Figure 3. It indicates an instance of Table `T` whose schema is defined in Figure 1. Column value is encrypted using randomized encryption with an enclave enabled key. Suppose that the application issues the (parameterized) query `select * from T where value = @v`; the driver encrypts the value of parameter `@v` before forwarding the query to SQL Server. Suppose that the filter predicate is evaluated using a table scan. In our architecture, data would be fetched from the storage engine encrypted, into the filter operator which would, for each row, invoke the enclave to evaluate the filter. Inside the enclave, values are decrypted and the filter is evaluated on the corresponding plaintext. The boolean result of the

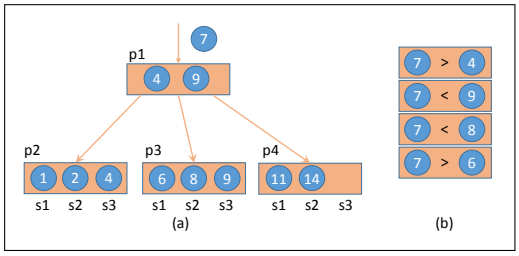

Figure 4: A single-column range index. Ciphertext (using RND encryption) is shown with a blue circle over the plaintext (not visible to SQL Server). Page ids ($p1$-$p4$) and slot ids ($s1$-$s3$) within pages are also shown. Also shown are comparisons performed in the enclave and their plaintext results (using > or <) while inserting (encrypted) key 7. The results of comparisons ensure that key 7 is inserted between keys 6 and 8.

predicate is returned to SQL Server in the clear (causing information leakage that we will analyze below) using which SQL Server continues its execution as usual. As this example suggests, the same expression is typically invoked multiple times. To accommodate this usage pattern, an expression is *registered* once in the enclave and invoked subsequently using the handle returned by the registration.

3.1 Indexing

AE supports equality and range indexes, which differ in terms of supported functionality and information leakage. We do not support clustered range indexes for reasons that we will discuss in Section 4.5.

3.1.1 **Equality indexing.** An equality index is implemented on DET columns by building a standard B+-Tree. The index keys are ordered in the B+-Tree using ciphertext, not plaintext values. By also looking up the index using ciphertext based ordering, we support equality-based lookups, but not range lookups.

3.1.2 **Range indexing.** Range indexing is supported on RND columns using a B+-Tree, using the enclave for comparisons. Unlike equality indexes, the indexed keys are now ordered by their plaintext values (the index still stores only ciphertexts). When building the index or performing operations such as lookups or updates, comparisons are routed to the enclave; the enclave decrypts its inputs and returns the comparison result in the clear, which is used by SQL Server for further processing as usual. Figure 4 shows a sample range index and the comparisons performed on ciphertext values while inserting a new key. Note that the vast majority of index processing, including latching/locking for concurrency and managing page splits/merges during updates, remains unaffected by encryption.

Operation	Leakage to strong adversary
Comparison (DET)	Frequency distribution over values
Comparison (RND)	Ordering over values
LIKE predicate using scans	Unknown predicate over values
LIKE predicate using an index (i.e. prefix matches)	Ordering over values plus some information about proximity
DDL to encrypt data	Limited access to encryption oracle only with client authorization

Figure 5: Operation leakage in AE

3.2 Data Confidentiality

Always Encrypted is designed for data confidentiality, and other security guarantees such as data integrity and protection from denial of service are non-goals. Further, AE does not provide metadata confidentiality and reveals table and column names, the number of tables, the number of columns in each table, (primary) key properties, the cardinalities of tables and the lengths of data values.

Ideally, we desire *semantic security* for encrypted data where a strong adversary does not learn anything beyond metadata and coarse statistical properties listed above. Unfortunately, semantic security is impractical since even communicating encrypted results over an untrusted network leaks some information and breaks semantic security.

Always Encrypted is instead designed to provide *operational* data confidentiality where the information that an adversary learns is a function of data operations performed when evaluating client-authorized queries. We note that since an adversary does not have access to the keys, it cannot generate arbitrary queries of its choice. Operational data confidentiality has been discussed in prior work, e.g. Cipherbase [3] and CryptDB [26]. Figure 5 shows the leakage of various operations. The leakage of DET encryption has been studied before, so we focus on the leakage associated with enclave based processing. Using the enclave for range queries reveals ordering; as a special case, an index build requires sorting of data that reveals the data ordering. Similar confidentiality guarantees are offered by order preserving encryption (OPE) [9]. However, existing constructions of OPE either leak higher-order bits of plaintext values or require a client-side component and do not guarantee encryption immutability [25]. (Informally, in a mutable encryption scheme, the encryption of a value could change when other values are inserted.) Our range indexes do not have these limitations. Evaluating LIKE predicates using an index (i.e. prefix matches) reveals some information about the proximity of values in addition to ordering, e.g. the fact that a subset

of values share a prefix. However, even if we did not support LIKE predicates, a client that reduces LIKE predicates to range predicates would reveal the same information to an adversary that has the background knowledge that the in-coming range queries emanate from prefix matches.

Secure Compilation For AE DDL. Always Encrypted uses DDL (`ALTER TABLE ALTER COLUMN`) statements to rotate keys and initialize encryption. When the CEKs involved in the operation are enclave-enabled, AE uses the enclave to avoid a client roundtrip. In particular, initial encryption requires the enclave to encrypt plaintext values. We seek to restrict exposing such an encryption *oracle* at the enclave where an adversary can generate ciphertexts for plaintexts of its choosing. Accordingly, we only allow the enclave Encrypt function to be called if the client explicitly authorizes it. We check for client authorization by having the driver *sign* the query text using the session secret; we compute a SHA256 hash of the query text and include it along with the CEKs that are encrypted with the shared secret. The enclave seeks a *proof* of client authorization when SQL Server requests access to the Encrypt function [1]. SQL Server supplies a proof using the parse tree of the Alter Table Alter Column query to the enclave, which uses the parse tree, the raw query text and its SHA256 hash to validate that the client is authorizing the type conversion needed for the DDL.

3.3 Design Alternatives

Always Encrypted uses the enclave only for expression evaluation. An alternate design explored in prior work [7, 8] is to run the entire database engine inside the enclave. We decided against the latter design due to considerations in Azure Sql Database, the Platform-as-a-Service (PaaS) cloud offering of SQL Server. In a PaaS setting, the cloud organization is tasked with administrative tasks such as configuring backups, replication for high-availability, and troubleshooting in the event of failures. To perform these tasks cloud operators need to be able to perform operations such as examining query plans to troubleshoot performance related problems, collecting dumps during a crash, connecting to the database server running in production to query the system runtime, e.g. the transaction conflict graph, and in rare scenarios, attaching a debugger. Supporting such operations requires full access to the SQL Server process, and running the process inside an enclave does not add any security.

In contrast, with the design of AE described above, all management tasks can be carried out with full access to the SQL Server process, but without access to the enclave memory. Enclave memory is automatically stripped from crash dumps, but since the amount of code running inside the enclave is small, dump information inside the enclave is almost never necessary. Furthermore, connecting to the database returns only encrypted data. Since cloud administrators cannot access the CEKs, they cannot decrypt the data. We see the above advantages as a side-effect of having a small *Trusted Computing Base* (TCB). While SQL Server is a complex system with millions of lines of code, our enclave on which the security of AE relies, is a tiny fraction of SQL Server's code base. The benefits of having a small TCB are widely recognized [2].

[1] Our authorization check is generalized to all type conversions using the enclave. We use Encrypt as an illustrative example.

4 IMPLEMENTATION

This section details the implementation of the AE feature.

4.1 Client Driver

We updated recent versions of various SQL Server drivers including ADO.Net, ODBC, and JDBC drivers to include client-side AE functionality. As described in Section 3, when the application issues a parameterized query to the driver, the driver invokes a SQL Server api, `sp_describe_parameter_encryption`, to retrieve encryption type information to encrypt query parameters and install enclave CEKs.

Example 4.1. Consider the parameterized query `select * from T where value = @v` over the running example table in Figures 1 and 3. The `value` column of Table `T` uses RND encryption with an enclave enabled key. The output of the call to `sp_describe_parameter_encryption` indicates that: (1) parameter `@v` should be encrypted with randomized encryption with CEK `MyCEK`, and (2) the CEK `MyCEK` should be sent to the enclave for evaluating the query. It further contains the metadata for CEK `MyCEK` and its corresponding CMK, `MyCMK` shown in the DDL in Figure 1. Since the query requires enclave computations, then SQL Server also makes a call to the attestation service and returns attestation information, which is included in the above output.

The driver constructs an encrypted query by encrypting parameters and issues it to SQL for execution. If CEKs are needed in the enclave, it first checks that the CEKs are authorized to be sent to the enclave using the corresponding CMK signature, and then encrypts them with the shared secret and sends it to SQL Server along with the query.

In order to avoid frequent CEK decryptions, which in the case of external key providers like Azure Key Vault could involve a network call, the driver caches the decrypted CEKs for a duration that can be controlled by clients. Further, the shared secret obtained as an outcome of attestation is cached in the driver in order to avoid frequent invocations of the attestation protocol. All of the above caches are shared across the entire client process. Note that the above architecture incurs two round-trips to SQL. In order to not force every application to incur two round trips, we add a property to the

connection string indicating its use for AE. In the absence of the property, the driver does not invoke the special API described above.

The fact that SQL is the source of truth for type deduction and key metadata introduces the following vulnerabilities in the system. First, SQL could maliciously alter the output of `sp_describe_parameter_encryption` to claim that parameters corresponding to encrypted columns are not encrypted. In order to address this issue, we allow an application to explicitly force a parameter to be encrypted. Second, instead of returning the key metadata for keys provisioned by the client, SQL could return metadata corresponding to keys provisioned maliciously. In order to prevent this attack, we allow the application to restrict the key paths of the CMK to a list of trusted paths. The driver checks that the CMK metadata returned by SQL belongs to the list of trusted paths.

4.2 Attestation and Shared Secret

The goal of attestation is for the client to check the health of the enclave before releasing keys. The attestation protocol is invoked at query time on a signal from the client during the call to `sp_describe_parameter_encryption` (and only when needed, i.e., there is enclave computation involved.) We build upon attestation to establish a shared secret between the driver and the enclave.

Attestation breaks down into two portions: the health of the enclave platform, and the health of the code running inside the enclave. The details are specific to the enclave platform. We describe the details for the VBS enclave. The attestation service we support is a feature in Windows Server known as the Host Guardian Service (HGS) [18]. HGS measures the health of a host machine using Trusted Platform Module (TPM) measurements. TPMs measure the boot sequence of a host and the measurement is returned in the form of a log called the *TCG* log. In an offline step, the TCG log obtained from the machine hosting SQL is registered with the HGS service to be included in its white-list. For VBS enclaves, we only trust the hypervisor, but not the host kernel. Therefore, we are only interested in the measurement of the boot sequence until the hypervisor is loaded. In order to attest the VBS enclave, SQL invokes Windows to send the current TCG log to HGS, which looks up its white-list and responds with a *health certificate* in the event of a match in the white-list. The health-certificate is signed by a signing key possessed by HGS, that we refer to as the *HGS signing key* and contains a signing key possessed by the host hypervisor, which we refer to as the *host signing key*.

SQL then issues a call to Windows to measure the enclave. The measurement is called an enclave *report* and it contains attributes of the enclave including the *author ID* that refers to the signing key used to sign the enclave binary, the hash of the enclave binary, and version numbers of the enclave and the host hyper-visor. Our VBS enclave creates an RSA public/private key pair when it is loaded. In addition, the enclave report contains a hash of the enclave's public key. We use Diffie-Hellman (DH) key exchange to establish a shared secret between enclave and driver, and fold it into the attestation protocol to save client-server roundtrips. When the attestation protocol is invoked (as part of the call to `sp_describe_parameter_encryption`), the client passes its DH public key. As part of its attestation information, SQL returns:

(1) The host health certificate containing the host signing key.
(2) The enclave report signed by the host signing key.
(3) The enclave's public key and DH public key, signed by the enclave's public key. Since the client sends its DH-public key as input, at this point, it follows from the DH protocol that the enclave already holds the shared secret.

On receipt of the above information, the client checks the chain of trust as follows.

(1) Check that the health certificate is signed by the HGS signing key. It obtains the HGS signing key by querying HGS (all HGS APIs are exposed using http(s)).
(2) Check that the enclave report is signed by the host signing key embedded in the health certificate.
(3) Check that the enclave is healthy. In our current implementation, we base this check on: (1) the signing key; we build the enclave binary using a specially provisioned signing key, and use it to check the enclave health. Using the binary hash is a possibility, but would break even with minor modifications to the enclave code, and (2) version numbers; in the event of a security update to our enclave, we build the enclave with an updated version number and would release a client that checks for the updated version number.
(4) Check that the enclave public key returned is consistent with the hash embedded in the report, and that the enclave DH public key is signed by the enclave public key. At this point, the attestation process is complete, and the client can derive the shared secret.

As noted above, the shared secret is used by the driver to encrypt CEKs and is sent on the TDS stream when executing the query. We note that one limitation of using the shared secret as-is, is that SQL could replay the TDS stream to send keys to the enclave. In order to address replay attacks, the driver adds a nonce to the encrypted CEKs being sent to the enclave. The enclave checks that the nonce is not repeated (for a given session, i.e. shared secret value). A simple strawman for nonce checking is to use a counter at the driver to generate nonces. The enclave checks if a new nonce

is greater than the most recent previous nonce. While this strawman requires $O(1)$ enclave state per session, it does not work correctly when driver-enclave communication is out of order, which is possible since both the client application and SQL Server are multi-threaded.

Our implementation of nonce checking is a generalization of the above idea: the driver still uses a counter to generate nonces, however the enclave now tracks all historical nonces. The enclave encodes all historical nonces using compact ranges. For example, the contiguous set of nonce values 0, ..., 100 are encoded using a range $[0, 100]$. The overall idea behind this design is that, since the driver generates sequential nonce values, the sequence of nonce values that the enclaves sees is still close to sequential with some local reorderings, which translates to a very compact encoding of historical nonces.

4.3 SQL Engine: Metadata and Type System

Figure 6: Encryption Type Lattice

SQL stores information about encryption in its metadata. This includes key metadata in new system tables we introduce, and also encryption information associated with each column. We enhance the SQL type system to reason about encryption. Encryption information is incorporated as an additional attribute of SQL types; for instance, there is an encrypted integer, an encrypted string, an encrypted datetime, etc. The type information of a column, parameter, or variable includes not only the encryption type, but also the identifier of the corresponding CEK. Type deduction on a query consists of checking not only plaintext types (e.g. can a string be used to lookup an integer column) but also encryption types. Therefore, we introduce an additional phase of type deduction in SQL, namely encryption type deduction. Furthermore, unlike plaintext types that are fully declared, owing to our transparent API, encryption types are not declared in the input query. Hence, encryption type deduction needs to operate on unknown types.

For this purpose, we use the observation that our encryption types form a lattice and we setup a constraint solving system using the lattice to infer encryption types. For ease of exposition, we describe the lattice structure without enclaves in the picture. The extension to include enclaves is straightforward. In the absence of enclaves, there are three generalized encryption types — Plaintext, Deterministic and Randomized — that form a lattice shown in Figure 6 (with enclaves, there are more generalized types but still maintain a lattice structure). Operations decrease strictly as we go from Plaintext to Deterministic to Randomized. The arrows in the Figure indicate lattice order. We note that the above are generalized types since they do not refer to any specific CEK. As we compile the query, we add constraints on the encryption type of each literal (parameter/variable) and solve for them. We illustrate with an example.

Example 4.2. Consider the data shown in Figure 3 and the query `select * from T where value = @v`. Since this discussion does not include enclaves, assume that column value is encrypted with Deterministic encryption. As we compile the query, when we encounter parameter `@v`, we add it to our constraint system with an unknown encryption τ with the constraint $\tau \leq$ Randomized, where $\leq$ is a reference to the lattice order. When we encounter the predicate value = @v, we add two constraints: (1) $\tau \leq$ Deterministic since equality is not allowed on Randomized encryption (without enclaves), and (2) $\tau =$ `EncryptionType(value)` since equality is only allowed if both operands have the same encryption type (this is also true with enclaves). Since the encryption type of the column value is known, solving for the above system yields the result that the encryption type of parameter `@v` is the same as the encryption type of the column value.

In our implementation, we do not explicitly use a constraint solver. We use a Union-Find algorithm to solve the constraints implicitly. We have equivalence classes to represent all operands with the same (potentially unknown) encryption type, merging them when we encounter an equality constraint if no constraint violation is introduced. Inequality constraints are processed by adjusting the encryption type of an equivalence class to account for the inequality (e.g., from Randomized to Deterministic in the above example), again only if no constraint is violated in doing so. The above encryption type deduction also tracks all CEKs needed in the enclave for query processing. There could be cases where we have multiple solutions to the constraint system. In such cases, our preference is to solve using the Plaintext type.

The above type deduction is invoked during the call to `sp_describe_parameter_encryption`, but it is also invoked during query execution as part of normal type deduction. The results of type deduction are cached in the plan cache with the query plan, to avoid recomputing them on every execution.

4.4 SQL Engine: Expression Services

As noted in Section 3, we run a subset of expression services (ES) inside the enclave. In SQL, ES is implemented as a stack machine. All expressions required for query processing

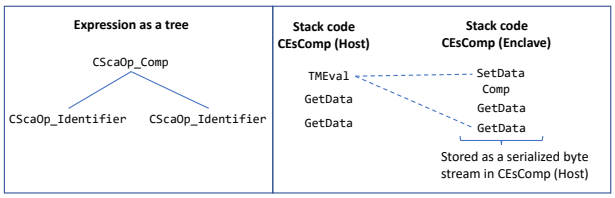

Figure 7: Illustration of Expression Compilation

are compiled into ES objects of a class called `CEsComp` and stored in the plan cache. At runtime, SQL generates an executable version of `CEsComp`, called `CEsExec` which exposes an `Eval()` method to run the stack program on provided inputs. All scalar operations of query operators in SQL Server are encapsulated within `CEsExec` objects they own: evaluating filter predicates, computing hashes for hash join probing, and checking join equality all translates to `Eval()` calls of these objects.

When running ES in the enclave, we faced the following challenge. Like all other components in SQL, ES does not call the operating system (OS) directly for resource management, and instead uses SQL's own internal OS abstraction called SQL OS. SQL OS itself calls the OS through a platform abstraction layer that lets SQL run on operating systems other than Windows. However, the enclave runtime explicitly excludes the OS, since the OS is untrusted. The enclave runtime does support all the resource management that an OS provides (memory, threading and synchronization, exception handling) but it is provided through restricted non-OS interfaces.

Given the above challenge, we faced three options for running ES within the enclave: (1) reimplement ES, (2) port ES with SQL OS, and (3) port ES without SQL OS. We rejected the option of reimplementing ES (a departure from the Cipherbase project); we wanted to inherit all the benefits of ES, e.g. its handling of strings, specifically collations, NULL values and exceptions. Since our goal is to eventually support a larger fraction of SQL, we chose to *port* ES to run inside the enclave. We also rejected the option of porting SQL OS, since SQL OS is a large component written to support all of SQL, whereas ES requires only a small subset of SQL OS. Therefore, we wrote a small SQL OS layer (that we will refer to as the *enclave SQL OS* to distinguish it from the host SQL OS) that supports the SQL OS abstractions needed for ES in addition to cryptographic operations needed within the enclave, and is implemented on top of the enclave runtime. The above layering also lets us easily port our enclave code between enclave platforms; by re-implementing the enclave SQL OS against different enclave runtimes, we let most of the enclave code remain unchanged.

Under the above approach, we compile the source code of ES to generate two binaries — the standard ES binary running outside the enclave, and the enclave binary. As noted in Section 4.2, we sign the enclave binary with a specially provisioned key.

Expressions in SQL are compiled from their tree representation to a stack machine, specifically the `CEsComp` object referred to above. We add a new instruction called `TMEval` to the ES stack machine instruction set. This instruction invokes an enclave computation. We represent the enclave computation using another `CEsComp` object, one that is evaluated within the enclave. The expression object running in the enclave is serialized and stored inlined in the host object. We serialize the object as a way to implement a deep copy. During execution, the entire `CEsComp` object is reconstructed inside the enclave. We reconstruct the object since the compile-time `CEsComp` object is used at execution time as well, and having the enclave reference an object stored in the host would introduce an attack vector where SQL could interfere with the enclave evaluation by tampering with the `CEsComp` object.

We note that the `TMEval` instruction is used only for enclave computations. Equality operations on deterministically encrypted columns are simply treated as `VARBINARY` or `BINARY` equality operations.

Example 4.3. Consider our running example query from Example 4.2, select * from T where value = @v. Figure 7 shows the tree representation of the comparison, and the output of compiling it to a stack machine. We generate two `CEsComp` objects, one running at the host with a stub `TMEval` for the enclave call, and another running in the enclave.

4.4.1 Expression Evaluation within Enclave. The enclave exposes an interface `Eval(expr, inputs, outputs)` to evaluate a scalar expressions within the enclave. The parameter `expr` in the `Eval` call is a serialized representation which deserializes to a `CEsExec` object within the enclave. The parameter `inputs` is an array of data values that form the input to the expression evaluation; the parameter `outputs` is an array of data value buffers for storing the outputs of the evaluation.

The enclave `Eval` method is called from within the `TMEval` stack instruction with inputs popped from the stack during host-side expression evaluation. The enclave enforces security checks that ensures for instance that encrypted and plaintext values cannot be compared.

We now describe how encryption and decryption is handled during expression evaluation within the enclave. The ES stack instruction set contains two instructions `GetData` and `SetData` to move data to and from the stack, respectively. For enclave expression evaluations, the source of any

`GetData` instruction is one of the values in the inputs parameter, and the destination of any `SetData` instruction is one of the buffers in the `outputs` parameter.

These instructions are annotated with the type of data, which includes the CEK identifier and encryption scheme when the data is encrypted. Using this information, the `GetData` instruction automatically decrypts any encrypted data before it is placed on the stack; Similarly, the `SetData` instruction automatically encrypts the data before moving it off the stack if its type information indicates it should be encrypted. In other words, all decryptions and encryptions happen at ingress and egress points, and the stack program evaluation itself is oblivious to the encryption details.

Enclave memory is stripped from dumps and not visible in the debugger. This does not affect our supportability since the enclave code is small. We leverage structured exception handling to obtain coarse-grained information in the case of hardware faults such as access violations.

4.5 Indexing

The core idea behind indexing was presented in Section 3. One of the main challenges we encounter with indexing is recovery. In SQL Server, redo recovery is physical, but undo recovery of indexes is logical; for instance, aborted inserts are undone by navigating the B+-Tree and deleting the record. This poses a problem for indexes on encrypted columns (henceforth referred to as encrypted indexes). Encrypted indexes require keys in the enclave, and the client only sends keys when running queries. Hence, we have to consider the possibility that the client never runs any query using the encrypted index, which could potentially block recovery. In order to address this problem, we mark any recovery transaction that finds the key missing to be *deferred*, leveraging a pre-existing mechanism of deferred transactions available in SQL. When the client connects and sends keys to the enclave, the deferred transactions are resolved. Since deferred transactions hold locks, the above approach could lead to large parts of the database being unavailable. For instance, if SQL crashes during a bulk load on a table with an encrypted index, then the deferred transactions could lock up a large portion of the table; while clients can connect to the database, they would be unable to perform update operations on most of the table.

To mitigate the above problem, we leverage the *constant-time recovery* (CTR) feature also available in SQL Server 2019 [1]. Briefly, CTR makes the database fully available with all locks released in constant time in the event of a crash. It does so by persisting versions of the data; when the database recovers from a crash, clients only get access to the latest committed version (with all locks released), while uncommitted versions are cleaned in the background. In the presence of an encrypted index, the database is fully available for clients, but the version cleaner that performs index traversals could potentially not find keys in the enclave, in which case it keeps retrying. When the client connects to provide keys, the version cleaner completes successfully. When the database is configured to use CTR, then even though the above availability issue is not eliminated (there are corner cases where we could end up with deferred transactions), the overall database availability is improved.

The above approach ensures database availability without relying on the client to supply keys. However, if the client never supplies keys, then other database administration tasks such as log truncation are blocked. The same issue also arises if we are restoring a database backup in a machine that has no enclave configured. In order to address such problems, we introduce the mechanism of forcing resolution of deferred transactions by skipping recovery of index pages and marking the index as invalid in the metadata. If an enclave is not configured, then index invalidation is automatic, whereas if an enclave is configured, then index invalidation could be initiated using explicit policies based on timeout or resource consumption, e.g. log space consumption. Since invalidating a clustered index can lead to data loss, we do not support clustered indexes on encrypted data.

4.6 Performance Optimizations

Section 4.1 lists performance optimizations relevant to the driver such as caching of CEKs. We now discuss optimizations in the SQL engine. Our optimizations focus on the use of the enclave. Calling the enclave incurs an overhead since it resides in a different security boundary; and since expression evaluation constitutes the inner loop of query processing, by moving it into the enclave, we make the inner loop expensive. Instead of calling the enclave as a function, i.e. synchronous execution, we spool up an enclave worker thread and pin it to a core. Host workers submit work to the enclave using a queue, and the enclave worker consumes and performs the work. After completing its work, the enclave worker spins for a fixed duration polling for work before exiting the enclave and going to sleep. In this way, if the system is making heavy use of the enclave, then the expectation is that host workers are constantly feeding the enclave work, keeping it busy, owing to which we avoid the enclave context switch cost. On the other hand, if the workload does not heavily use the enclave, then the enclave goes to sleep freeing up resources for SQL Server host.

Further, the enclave is multi-threaded and each enclave thread processes ES requests as described in section Section 4.4.1. To simplify synchronization issues all state changes such as adding CEKs are handled by a single enclave thread. The other threads only read the current state. As state changes

are rare operations this design allows for efficient scaling of enclave resources. Our performance optimizations are still work in progress and we are in the process of implementing some of the optimizations identified in [3].

5 PERFORMANCE EVALUATION

This section contains results of preliminary performance experiments with AE using the TPC-C benchmark [32].

5.1 Hardware configuration

Our experiments were run on virtual machines on Microsoft Azure [21]. The SQL Server AE instance was run on a standard *DS15 v2* virtual machine with 20 cores and 140 GB of main memory. The VM was equipped with two separate P30 premium SSD disks to store data and log files on separate drives. We used VBS enclaves [35] and we allocated four threads to run the enclave.

We used a Microsoft internal SQL Server TPC-C driver called *Benchcraft* to run the benchmark. Benchcraft was run on a standard *D4 v2* virtual machine with 8 cores and 28 GB of main memory. The Host Guardian Service (Section 4.2) was run on a standard *DC4s* VM with 4 cores and 16 GB of main memory to attest the VBS enclave.

5.2 Systems Compared

We compared the performance of three SQL Server configurations described below.

1. SQL-PT: This configuration runs SQL Server on TPC-C data with no encryption. Further, the TPC-C client driver connects to SQL Server using a non-AE connection string. This configuration serves as the baseline to measure various AE-related overheads.

2. SQL-PT-AEConn: This configuration again runs SQL Server with no data encryption. However, the client driver now connects using an AE connection string. While basic transaction processing remains unchanged, this configuration introduces one additional client-server roundtrip for `sp_describe_parameter_encryption` as described in Section 4.1.

3. SQL-AE: This configuration runs SQL Server with data encryption and therefore relies on Always Encrypted for transaction processing. By default, we use *RND* encryption with enclave enabled keys. We also vary the number of enclave threads. *SQL-AE-RND-1* and *SQL-AE-RND-4* specify 1 and 4 enclave threads respectively. We also consider a variant of this configuration with *DET* encryption and non-enclave enabled keys which does not use the enclave. We prefix these variants as *SQL-AE-RND-1*, *SQL-AE-RND-4* and *SQL-AE-DET*; *SQL-AE* refers to *SQL-AE-RND-4* unless explicitly qualified.

5.3 Benchmark and Encryption Configuration

We use the TPC-C benchmark [32] with some minor changes for our performance evaluation. The benchmark consists of nine tables and five types of transactions over these tables that simulate the business activities of a wholesale supplier.

We encrypt the personally identifiable columns of the benchmark which are the `C_FIRST`, `C_LAST`, `C_STREET_1`, `C_STREET_2`, `C_CITY`, and `C_STATE` columns in the `Customer` table. As noted above, we use `RND` encryption with enclave-enabled keys for *SQL-AE-RND* configurations, and `DET` with enclave-disabled keys for *SQL-AE-DET*. Since the column to CEK mapping does not impact performance we choose the simplest configuration of using the same CEK for all encrypted columns. All other columns in the database remain unencrypted.

We made minor changes to the TPC-C stored procedures to reflect current functionality limitations of Always Encrypted. We modify the `Payment` and `Order Status` transactions to remove the `ORDER BY` on `C_FIRST` (since we do not, at this point support `ORDER BY` using enclaves); both transactions select a subset of customers using a filter predicate, order these customers by their first names (`C_FIRST`) and use this ordered list to identify the median customer. We also create a `NONCLUSTERED` (non-unique) index `CUSTOMER_NC1` on `CUSTOMER(C_W_ID, C_D_ID, C_LAST, C_FIRST, C_ID)` deviating from the benchmark specification which requires a unique constraint on these columns.

With this encryption configuration, the only scalar operation over encrypted data is the equality `C_LAST = @c_last` of values in the `C_LAST` column against a provided parameter `@c_last`, and this operation is used by both `Payment` and `Order Status` transactions. Together, these two transaction types account for around 47% of the workload; in terms of expressions evaluated, the equality predicate is invoked for 60% of transactions of each type (the other 40% involve an equality of `C_ID` and not over `C_LAST`). In summary, around 28% of the workload (in terms of expression evaluation) involves computation on encrypted data.

The benchmark includes a scaling factor `W` representing the number of warehouses. For our experiments, we used `W = 800`; consistent with the benchmark specification, this is the smallest scaling factor that maximized cpu usage for *SQL-PT*, our baseline configuration. At `W = 800` warehouses the `CUSTOMER` table has 24 million rows.

5.4 Results

5.4.1 AE vs. Baselines. Figure 8 shows the relative (normalized) performance of the three configurations. We varied the number of TPC-C client driver threads (shown on the horizontal axis) and for each setting show the normalized

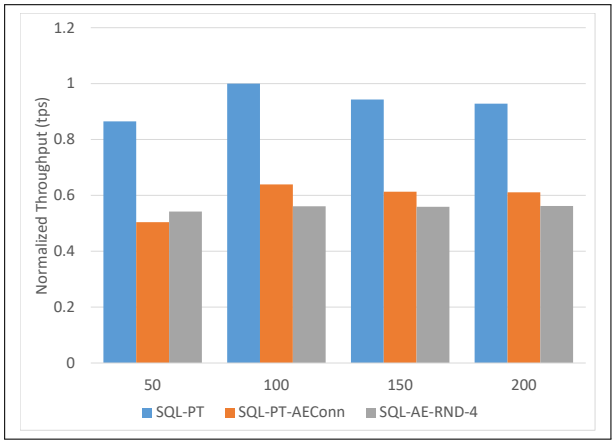

Figure 8: Normalized TPCC benchmark transaction processing rates for the three systems compared for different number of TPCC client driver threads. The benchmark scaling factor was `W = 800`.

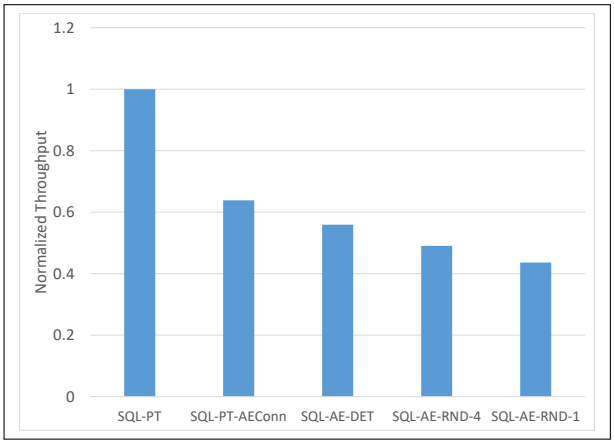

Figure 9: Normalized TPCC benchmark transaction processing rates comparing enclave-based Always Encrypted processing and non-enclave-based processing using *DET* with 100 client driver threads and `W = 800`

.

throughput for the three configurations. At 100 client driver threads the throughput on all three configurations is close to or at their respective maximums. Under this load, AE currently achieves roughly half the throughput of the baseline plaintext SQL Server.

The *SQL-PT-AEConn* system that runs SQL Server with no encryption but with an AE connection achieves 64% of the throughput of the plaintext baseline. This suggests that the bulk of the drop of performance happens due to the additional roundtrip introduced by the `sp_describe_parameter_encryption` call to provide transparency. However, we believe this overhead is not fundamental and can be reduced with client-side caching of the results of `sp_describe_parameter_encryption`.

5.4.2 Enclave vs. Deterministic Encryption. Figure 9 compares the performance of enclave-based processing with *RND* encryption (*SQL-AE-RND*) against non-enclave based processing with *DET* encryption for 100 client driver threads and 800 warehouses.

The performance of *SQL-AE-DET* is roughly in between those of *SQL-PT-AEConn* and *SQL-AE-RND*. With the optimizations we currently support, enclave based computation is 12.3% slower than DET encryption. We expect this gap to narrow as we add further optimizations. Since enclaves provide further functionality, we view this result as promising and an acceptable overhead for the additional functionality.

6 CONCLUSIONS AND FUTURE DIRECTIONS

This paper presented Always Encrypted, a feature of Microsoft SQL Server that offers end-to-end data confidentiality guarantees using encryption. The main challenge of computing on encrypted data is addressed by AE using property-preserving deterministic encryption in its initial release, and going ahead, with a trusted execution environment in the form of an enclave running within the SQL Server process. Our design takes a first step towards supporting richer functionality on encrypted data while providing confidentiality guarantees against a strong adversary that can compromise SQL Server, in the process enabling complex administrative tasks to be undertaken without access to sensitive data. The AE system is the first of its kind in the industry.

We are still in early days of understanding and exploiting the full potential of the enclave-based AE architecture. While our initial performance improvements are promising, we are working on further performance improvements. In its current form, AE restricts query functionality and is not intended as a data protection mechanism for the entire database, but rather for a small subset of sensitive columns such as personally identifiable identifiers. The main avenue for future work is to make AE a general-purpose solution for all data without restricting query functionality.

ACKNOWLEDGMENTS

AE is the result of a large scale collaboration among SQL, Windows and Microsoft Research. We acknowledge the contributions of Manuel Costa, Elnata Degefa, Kedar Dubashi, Benjin Dubishar, Raul Garcia, Istvan Haller, Joachim Hammer, Ajay Manchepalli, Bala Neerumalla, Aditya Nigam and Nikhil Vithlani, who contributed to the design and implementation of AE.

REFERENCES

[1] Panagiotis Antonopoulos, Peter Byrne, et al. 2019. Constant Time Recovery in Azure SQL Database. *PVLDB* 12, 12 (2019), 2143–2154.

[2] Arvind Arasu, Spyros Blanas, Ken Eguro, et al. 2013. Orthogonal Security with Cipherbase. In *CIDR*.

[3] Arvind Arasu, Ken Eguro, Manas Joglekar, et al. 2015. Transaction processing on confidential data using Cipherbase. In *ICDE*. 435–446.

[4] Arvind Arasu, Ken Eguro, Raghav Kaushik, et al. 2014. Querying encrypted data. In *SIGMOD*. (tutorial).

[5] Michael Armbrust, Armando Fox, Rean Griffith, et al. 2010. A view of cloud computing. *Commun. ACM* 53, 4 (2010), 50–58.

[6] Azure Key Vault 2019. (2019). https://azure.microsoft.com/en-us/services/key-vault/.

[7] Sumeet Bajaj and Radu Sion. 2011. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *SIGMOD*. 205–216.

[8] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.* 33, 3 (2015), 8:1–8:26.

[9] A. Boldyreva, N. Chenette, and A. O'Neill. 2011. Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In *CRYPTO*.

[10] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2016. Homomorphic Encryption for Arithmetic of Approximate Numbers. Cryptology ePrint Archive, Report 2016/421. (2016). https://eprint.iacr.org/2016/421.

[11] Ciphercloud 2019. (2019). http://www.ciphercloud.com.

[12] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86.

[13] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *PVLDB* 13, 2 (2019), 169–183.

[14] European Network and Information Security Agency 2009. Cloud computing risk assessment. (November 2009).

[15] Google Encrypted BigQuery client 2019. (2019). https://github.com/google/encrypted-bigquery-client.

[16] Hakan Hacigümüs, Balakrishna R. Iyer, Chen Li, et al. 2002. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*. 216–227.

[17] Shai Halevi, Yuriy Polyakov, and Victor Shoup. 2019. An Improved RNS Variant of the BFV Homomorphic Encryption Scheme. In *CT-RSA*, Mitsuru Matsui (Ed.). Springer, 83–105.

[18] Host Guardian Service (HGS) 2016. (2016). https://docs.microsoft.com/en-us/windows-server/security/guarded-fabric-shielded-vm/guarded-fabric-manage-hgs.

[19] Yehuda Lindell. 2019. Lessons Learned from Recently-Discovered Major Vulnerabilities in Hardware Security Modules. (June 2019).

[20] Peter Mell and Tim Grance. 2009. Effectively and Securely using the Cloud Computing Paradigm. (October 2009).

[21] Microsoft Azure 2019. (2019). https://azure.microsoft.com/en-us/.

[22] Oracle Transparent Data Encryption (TDE) 2019. (2019). https://docs.oracle.com/database/121/ASOAG/introduction-to-transparent-data-encryption.htm.

[23] Oracle Virtual Private Database (VPD) 2019. (2019). https://docs.oracle.com/cd/B28359_01/network.111/b28531/vpd.htm.

[24] Rishabh Poddar, Tobias Boelter, and Raluca A. Popa. 2019. Arx: An Encrypted Database using Semantically Secure Encryption. *PVLDB* 12, 11 (2019), 1664–1678.

[25] Raluca A. Popa, Frank H. Li, and Nickolai Zeldovich. 2013. An Ideal-Security Protocol for Order-Preserving Encoding. In *IEEE Symposium on Security and Privacy*.

[26] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, et al. 2011. CryptDB: protecting confidentiality with encrypted query processing. In *SOSP*. 85–100.

[27] SAP HANA Client-Side Data Encryption 2019. (2019). https://blogs.sap.com/2018/11/20/sap-hana-client-side-data-encryption-by-the-sap-hana-academy/.

[28] SEAL 2019. Microsoft SEAL (release 3.4). https://github.com/Microsoft/SEAL. (Oct. 2019). Microsoft Research, Redmond, WA.

[29] SQL Server Row-Level Security 2019. (2019). https://docs.microsoft.com/en-us/sql/relational-databases/security/row-level-security.

[30] SQL Server Transparent Data Encryption (TDE) 2019. (2019). https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/transparent-data-encryption.

[31] Sybase Adaptive Server Enterprise 2019. (2019). https://blogs.sap.com/2013/04/23/sap-sybase-ase-keeping-private-data-private/.

[32] TPC-C Benchmark 2019. (2019). http://www.tpc.org/tpcc/.

[33] Stephen Tu, M. Frans Kaashoek, Samuel Madden, et al. 2013. Processing Analytical Queries over Encrypted Data. *PVLDB* 6, 5 (2013), 289–300.

[34] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. https://cacheoutattack.com/. (2020).

[35] Virtualization-based Security 2019. (2019). https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs.

[36] Wenting Zheng, Ankur Dave, Jethro G. Beekman, et al. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI*. 283–298.