Orthogonal Security With Cipherbase

Arvind Arasu[1], Spyros Blanas[2] [*], Ken Eguro[1], Raghav Kaushik[1],
Donald Kossmann[3], Ravi Ramamurthy[1], and Ramaratnam Venkatesan[1]
[1]Microsoft Research [2]UW-Madison [3]ETH-Zurich

ABSTRACT

This paper describes the design of the Cipherbase system. Cipherbase is a full-fledged SQL database system that achieves high performance and high data confidentiality by storing and processing strongly encrypted data. The Cipherbase system incorporates customized trusted hardware, extending Microsoft's SQL Server for efficient execution of queries using both secure hardware and commodity servers. This paper presents the design of the Cipherbase secure hardware and its implementation using FPGAs. Furthermore, this paper shows how we addressed hardware / software co-design in the Cipherbase system.

1. INTRODUCTION

1.1 Problem Statement

The goal of many organizations today is to push as much data and computation into the cloud as possible. The cloud promises several cost advantages; e.g., increasing the utilization of an IT infrastructure. Furthermore, cloud computing can reduce the time to market for new data services and help organizations focus on their core business by outsourcing mundane IT tasks.

One of the most important concerns in the adoption of cloud computing is security; in particular, confidentiality. An organization may trust a cloud provider to properly operate provisioned services, but may not trust the employees of the cloud provider to keep its data confidential. In a public cloud, organizations may not trust co-tenants of shared cloud resources nor the isolation mechanisms put forth by virtual machines and hypervisors. In fact, in some scenarios, an organization may not even trust its own employees who operate a private cloud. As an example of a confidentiality breach, recently database administrators of several Swiss banks sold customer information to German and French tax authorities [23].

The goal of the Cipherbase project is to develop a novel cloud computing platform that allows organizations to leverage the advantages of cloud computing and at the same time achieve data confidentiality. The Cipherbase system provides the same features as traditional database systems (e.g., support for full SQL, transactions, and recovery) with almost the same performance and scalability characteristics, at little additional infrastructure cost. As will be shown, Cipherbase is based on an architecture with *secure co-processors*. In such an architecture, the goal is to decompose computation between traditional (insecure) hardware and trusted hardware.

In terms of confidentiality, the Cipherbase system supports various levels of encryption (from no encryption to strong encryption) and different end-to-end security settings so that the right level of confidentiality can be selected for all data.

More concretely, Cipherbase has the following properties:

- *Completeness:* Cipherbase is a full-fledged (SQL) database system. Thus, new applications can leverage the full richness of SQL and legacy applications need not be rewritten.
- *User-defined Confidentiality:* Users can specify encryption and end-to-end security for their data at a column granularity. Some data requires strong confidentiality guarantees; for other data, weaker guarantees suffice.
- *Efficiency:* Cipherbase executes queries and transactions efficiently, while meeting the user's confidentiality requirements for all data.

We call this set of features *orthogonal security* because it allows organizations to develop their applications and set their data security goals relatively independently of any performance, scalability, or cost considerations. Security comes at a cost and Cipherbase's performance degrades if all data requires strong confidentiality. So, organizations should carefully specify the level of confidentiality for data. For instance, public data such as *country names* should be marked as public and highly confidential data such as *customer names* should be marked as such. Cipherbase exploits these settings by optimizing queries and transactions taking the encryption and security of all data into account.

1.2 Design Space and Related Work

In general, there are three ways to build secure database systems:

- *Encryption at Rest:* Data is stored encrypted on a commodity storage system (e.g., hard disks) and shipped to a trusted domain when it needs to be processed. In this trusted domain the data is decrypted, processed, and possibly cached. The trusted domain may be located on the same premises (or even within the same box) as the (untrusted) storage system or remotely.
- *Secure Servers:* Data is stored and processed in the cloud on specially designated secure nodes.
- *Fully Homomorphic Encryption:* The data is encrypted in such a way that any operation (e.g., addition, multiplication, comparisons) can be performed directly on the data without decrypting

[*]Supported in part by the Microsoft Jim Gray Systems Lab.

6th Biennial Conference on Innovative Data Systems Research (CIDR '13)
January 6-9 2013, Asilomar, CA, USA

it. In this way, data can be processed using conventional untrusted servers. The encrypted results of a query from the untrusted system are shipped to the (trusted) client.

Encryption at Rest is the principle used in mainstream database products such as Oracle and Microsoft SQL Server. Users (or database administrators) can specify that certain data (tables or partitions) be stored encrypted (e.g., using AES) on disk. In these systems, the disks are assumed to be untrusted while other system components (main memory, CPU) are assumed to be trusted. Thus, the data is decrypted in main memory to process queries and updates. While these systems protect data against media theft or from attackers with access to the storage system, they do not protect data against attacks from administrators with super-user privileges. For instance, these systems did not provide any protection for the German and French customers of Swiss banks [23].

Dropbox [4] is an example of a remote *Encryption at Rest* system. Dropbox keeps files encrypted at its servers and ships data to client machines for processing. Secure databases that use the client for query processing over encrypted data also belong to this category [13]. Although this approach addresses the threat of administrator attacks, in a database context, shipping encrypted data (e.g., full tables) to off-cloud machines may be prohibitively expensive.

Secure Servers address the limitations of *Encryption at Rest* systems by offering special high-security processing nodes within the cloud. Some implementations such as AWS GovCloud [10] are built from commodity servers, but enhance their security by running specially vetted software stacks, highly restrictive security policies, and isolating the machines, both in terms of network connectivity and physical placement. However, this approach creates several practical problems. For example, the physical and logical isolation of these machines limits the types of applications that can run and the clients that may connect. Furthermore, this approach fragments the cloud. As a consequence, it increases the cost of building and maintaining the facility and complicates key cloud features such as seamless failover and dynamic scalability.

Other *secure server* arrangements employ devices such as IBM secure coprocessors (SCPs) [16] or Hardware Security Modules (HSMs) [15]. These devices are self-contained, forming a trusted computing region that can be readily deployed as an expansion card inside a commodity server. However, these devices have their own drawbacks. For example, secure co-processors are heavily limited in terms of memory and processing capabilities. This makes it impractical to run an industrial-strength database system, such as SQL Server, entirely in a secure co-processor. Similarly, HSMs are highly application-specific devices, effectively locking a specific application (perhaps even a specific instance of an application) to a particular piece of hardware. This again fragments the cloud, severely impacting adaptability and scalability.

Fully Homomorphic Encryption [6] can avoid the disadvantages of *Encryption at Rest* and *Secure Servers* since it allows processing of encrypted data in-situ on untrusted machines. Unfortunately, it is well-known that the state-of-the-art in fully homomorphic encryption is prohibitively expensive and therefore not practical [2]. There are many partially homomorphic encryption schemes that limit the kinds of operations (e.g., support multiplication but not addition) or the number of certain kinds of operations (e.g. at most N multiplications). These are less expensive, but cannot be used to achieve the *completeness* requirement of orthogonal security. CryptDB [19] is a recent system that exploits partially homomorphic properties of various encryption schemes to support a significant (but not full) subset of database functionality. Also, CryptDB does not provide orthogonal security since one cannot set a security policy independent of the query workload.

All three pure approaches have completeness, security, or performance limitations. TrustedDB [2] is an approach that combines the *Secure Servers* and *Encryption at Rest* approaches. TrustedDB has a novel architecture that combines an IBM secure co-processor (SCP) and a commodity server. It runs a lightweight SQLite database on the SCP and a more feature-rich MySQL database on the commodity server; as mentioned earlier, it is not practical to run an industrial strength database on an SCP. Query processing is distributed between two databases: Encrypted data is processed using SQLite in the SCP and plaintext data is processed using MySQL in the commodity server. This approach makes the best use of the available building blocks (secure hardware, commodity hardware, SQLite, MySQL).

Cipherbase adopts from TrustedDB the idea of combining trusted hardware and commodity servers in a single box. However, Cipherbase has a more sophisticated and fine-grained hardware-software co-design: Operators that involve encrypted data are executed on both the trusted (special-purpose) and untrusted (commodity) hardware. This thereby exploits plentiful commodity cloud servers as much as possible and uses comparatively limited trusted hardware as little as necessary. Another advantage of Cipherbase over TrustedDB is that Cipherbase extends a single industrial-strength database system (Microsoft SQL Server) to run all queries. This way, Cipherbase provides rich query-processing capabilities for all data; in contrast, TrustedDB can only provide rich SQL features for public data that is not encrypted.

1.3 Contributions

Cipherbase combines all three approaches discussed earlier to achieve orthogonal security. The key idea is to *simulate* fully homomorphic encryption on top of non-homomorphic encryption schemes (e.g., AES in CBC-mode) by integrating trusted hardware. In particular, Cipherbase uses trusted hardware to implement a core set of basic primitives to operate on encrypted data. An extended SQL Server database ships data to trusted hardware, invoking functionality within to implement query processing over encrypted data. Further, limiting the footprint of the trusted hardware enables Cipherbase to use inexpensive and secure dedicated circuits to implement trusted hardware functionality.

This paper reports on our most important design decisions, the Cipherbase architecture, and initial experimental results. Specifically, we present:

- *Cipherbase Architecture:* A new architecture that tightly integrates custom-designed, trusted hardware and commodity servers for a high performance and secure database system.
- *Trusted Hardware:* We discuss why it is advantageous to custom-design trusted hardware for secure database systems and give the design of our FPGA-based implementation.
- *Extended Database System:* We show how a commercial SQL database system (Microsoft SQL Server) can be extended to achieve orthogonal security. In particular, we show how queries can be processed across conventional servers and trusted hardware.
- *Security Models:* We identify a subtle dimension of data confidentiality: *runtime information leakage*. We show that any secure database system that ships data back and forth between trusted and untrusted domains reveals information even if data in the untrusted domain is strongly encrypted at all times. We also present novel ways to implement query operators in order to support different levels of runtime security.

The remainder of this paper is structured as follows: Section 2 describes the Cipherbase architecture. Section 3 defines the differ-

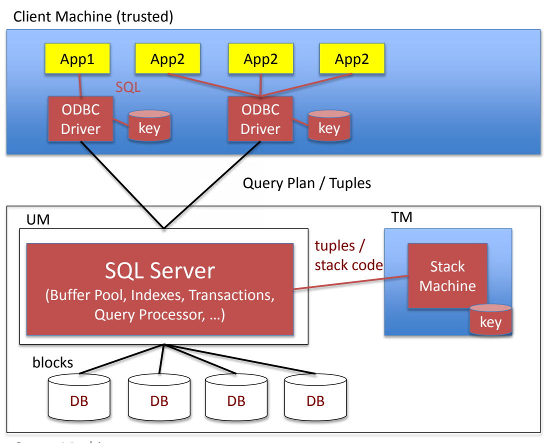

Figure 1: Cipherbase Architecture

ent levels of confidentiality that can be achieved with Cipherbase. Section 4 explains why we chose FPGAs to implement the trusted hardware and how we can make FPGAs secure. Section 5 gives details of how queries are executed across trusted and untrusted hardware. Section 6 lists several optimization techniques. Section 7 contains conclusions and possible avenues for future research.

2. OVERVIEW

Figure 1 gives an overview of the Cipherbase system. Applications (or end users) at client machines issue SQL queries and updates using, e.g., ODBC, embedded SQL, or a console just like in any other database system. These SQL statements are handled by the database driver at the client (e.g., an ODBC driver) and then processed by the server; again, just like in any other database system. Furthermore, Cipherbase has the same components as traditional database systems such as a storage manager (buffer pool, indexes, etc.) that reads/writes data in blocks from/to disks, a transaction manager (concurrency control, write-ahead logging, etc.), and a query processor (optimizer, runtime system, etc.).

What makes Cipherbase special is that it extends the main components of a database system; in particular, the ODBC driver at the client and the query processor. Specifically, Cipherbase extends all these components of Microsoft's SQL Server database system. Furthermore, Cipherbase integrates a special secure database processor that implements a stack machine to evaluate expressions on encrypted data.

The Cipherbase ODBC driver extends basic ODBC functionality in the following ways: First, the ODBC driver persists a (128-bit AES) key for each application. It uses this key to encrypt data, constants and parameter settings of queries and updates. Furthermore, it uses this key to decrypt results from the server. This way, encryption and security is transparent to the application with the only noticeable exception that the application needs to declare security requirements for data through column-level settings (Section 3). Second, the Cipherbase ODBC driver performs query optimization. Query optimization and statistics (e.g., histograms) needed for query optimization reveal information about underlying data, so we perform query optimization at the client, which is assumed to be trusted. Finally, for presentation in our examples, we assume that only data is stored encrypted and schema is in clear-text. In practice, the schema is also anonymized using opaque identifiers for table and column names with the Cipherbase ODBC driver managing the mappings.

The ODBC driver is assumed to run in a trusted computing environment. In many cloud computing environments, application servers also run in an (untrusted) public cloud. As part of future work, we plan to explore extending the architectural principles underlying Cipherbase to application servers to enable securely pushing even more functionality into the cloud.

To improve compilation performance, the Cipherbase ODBC driver locally caches meta-data and statistics, but original meta-data and statistics are stored encrypted in the server, respecting the confidentiality requirements of the application. The Cipherbase ODBC driver also supports caching and persistent storage of query plans, just like many other database products.

On the server side, Cipherbase receives a plan from the client's ODBC driver, interprets it using iterators, applies updates, and returns results to the client. The results are encrypted according to the application's confidentiality requirements. Even when data is strongly encrypted, the bulk of the query and transaction processing is carried out at the server using conventional main memory, processor caches, and CPUs, denoted as *UM* (for *Untrusted Machine*) in Figure 1. Furthermore, all data (including meta-data, statistics, logs, etc.) are stored on conventional disks (e.g., hard drives or flash). In Figure 1, conventional, untrusted hardware is denoted with a white background.

In addition to the UM, Cipherbase integrates a secure co-processor which is represented with a blue background and denoted as *TM* (for *Trusted Machine*) in Figure 1. The TM is used as a sub-module for core operations over encrypted data. The TM uses an FPGA and is placed inside the UM, connected via a high-speed 8x PCI Express bus. Since the bandwidth of the PCI Express bus is lower than the bandwidth of the UM's memory system, one of the design goals of Cipherbase is to minimize data transfer between the UM and the TM. In Cipherbase, the TM runs a *stack machine* that evaluates expressions such as predicates of SQL `WHERE` clauses. Other architectures that push more functionality into the TM are conceivable, but we try to limit the computations carried out in the TM. This is because it is comparatively more expensive to implement operations in the TM than on a commodity server. To provide the best scalability, we would like push as much of the computational load as possible to commodity cloud servers (while still observing the security policy). Cipherbase uses a stack machine for expression evaluation because this is the internal processing abstraction used in the SQL Server product. (More details on the evaluation of expressions are given in Section 5). Finally, the portions of the query plan that are to be run in the TM are signed by the client in order to prevent the adversary from issuing an instruction stream that sends plaintext back to UM.

The interpretation of a query plan involves shipping (encrypted) tuples from the UM to the TM and then decrypting, processing, and re-encrypting these tuples in the TM, before shipping the (encrypted) results back from the TM to the UM. The query plan specifies when and which tuples are shipped to the TM and which functions (i.e., stack code) the TM applies to the tuples. In order to decrypt and re-encrypt tuples, the application-specific secret key is known to the TM; it is not available and visible anywhere in the UM. More precisely, each TM has its own secret key which is burnt into hardware. Using this secret key, a TM stores the secret keys of applications in an encrypted format on the disks of the UM. It is important to note that just like the UM, the TM can have many "cores" (i.e., many FPGAs) in order to exploit intra-query parallelism for complex queries and/or inter-query parallelism for short transactions.

There are many challenges to extend a complex system such as SQL Server in order to implement orthogonal security using the architecture shown in Figure 1. For instance, we had to devise new protocols to establish new keys for new applications and to recover such keys in case they are lost at client machines. Furthermore, we are investing a great deal of engineering into extending the transaction manager and the implementation of indexes and materialized views. In this paper, however, we would like to highlight only a few extensions that we felt were particularly critical. Section 3 defines the security model and extensions to the SQL DDL that allow users to annotate whether data is confidential or public. Section 4 explains why we choose to implement the TM using FPGAs and how we make the TM secure. Section 5 shows how we extend the SQL Server runtime system to run operators on encrypted data both in the UM and TM. Section 6 explains how we extend the SQL Server query optimizer to generate plans that decide which data to ship to the TM, when to ship that data to the TM, and which functions to execute on that data in the TM.

Before describing these aspects, a final note on the Cipherbase architecture. This architecture makes use of all three design principles listed in the introduction. It uses the *Encryption at Rest* principle to store data, meta-data, statistics, and application keys persistently on disks in an encrypted form according to the confidentiality needs of the application. It uses secure hardware to evaluate expressions as part of the stack machine in the TM. Furthermore, it *simulates* fully homomorphic encryption by executing functions on encrypted data in the TM; that is, with the help of the TM, Cipherbase can apply a function efficiently on strongly encrypted data as if it were fully homomorphically encrypted. Whenever possible, Cipherbase also makes use of homomorphic properties of an encryption scheme. For instance, if the data is encrypted in an order-preserving way [3] and if the security model allows to do so, then the order-preserving encryption of the data is exploited to carry out as much computation as possible on encrypted data in the untrusted server. Likewise, a (partially) homomorphic encryption technique may support additions on encrypted data but no multiplications; again, Cipherbase will exploit this property for all queries that involve only additions and process those in the UM.

3. DATA SECURITY

In this section, we introduce two dimensions of security identified in Cipherbase: *static* security and *runtime* security. We also discuss their implications for the design of Cipherbase. We begin by describing our adversary model.

3.1 Adversary Model

There is a variety of threats in migrating to the cloud. One important category is the threats to data confidentiality. Information about sensitive data could get breached to cloud administrators who manage the server or hackers who can gain the equivalent of root access. Another category is threats to data integrity where an adversary actively tries to tamper with the data: e.g. deleting or inserting records, and modifying query results. It is also possible for an adversary to disrupt the database-as-a-service in other ways, e.g. selective or total denial of service.

This paper focuses primarily on threats to data confidentiality. In Cipherbase, this is accomplished primarily by restricting access to keys and plaintext to the TM. We note that the adversary does not need to have access to the key to tamper with the database. Integrity protection can be added on top of Cipherbase by using techniques from prior work [20, 24]. The main adversary we study is an eavesdropping cloud administrator. We assume an attack model where the adversary has complete control of the UM's software and hardware through super-user privileges, and can monitor the contents of disk, memory, the traffic between the UM and TM, the traffic

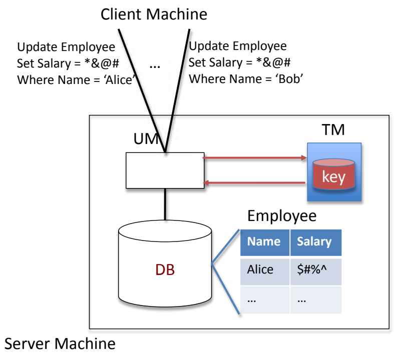

Figure 2: Information Leakage Through Query Access Patterns

between the client and the server, and all operations in the UM. We assume that the TM is secure and that the adversary does not have access to the internal state within the TM. We believe that, when properly built, the TM will only be vulnerable to physical attacks to the silicon itself. Protecting the TM against physical attacks is beyond the scope of this paper. However, there is a multitude of potential protective measures that can be applied to the TM's hardware to mitigate attacks such as eavesdropping, side-channel analysis or fault-based manipulation [21, 17]. We will discuss the security of TM in more detail in Section 4.

3.2 Static Security

We use the term *static* security to refer to the security of data at rest, i.e., when stored on disk. Currently, Cipherbase supports the following column-level static security options: *no encryption* (for public data), *deterministic* and/or *order-preserving* (weaker, partially homomorphic encryption), and *strong* (using AES in CBC mode). The information leaked by weaker encryption has been analyzed in prior work. We note that information about a strongly encrypted column can be revealed from the data at rest through its *relationship* to other columns that are encrypted differently. We illustrate through an example. Suppose that we have a table `Emp` (`Name`, `Age`, `Salary`) of employee records where `Salary` is sensitive, so it is strongly encrypted, and `Name` and `Age` are public and not encrypted. Suppose that the adversary knows that older employees earn more on the average. Then the adversary gains some information about the ordering of employee salaries even though the `Salary` column is strongly encrypted. The above information leakage is an inevitable consequence of the user-specified options and therefore the only way to address it in Cipherbase is to encrypt more columns.

3.3 Runtime Security

Data confidentiality overall depends not only on the security of data at rest but also the information revealed during query processing. One aspect of query processing that affects data confidentiality is query encryption. As noted in Section 2, we encrypt only query constants. The encryption of constants is the same as the static security option of the corresponding column, for example constants corresponding to public columns are public. We now illustrate another example of how information about encrypted columns is leaked through their relationship to non-encrypted (or weakly encrypted) columns.

EXAMPLE 3.1. *Consider an employee table* `Emp` *(*`Name`*,* `Salary`*) where* `Salary` *is strongly encrypted and* `Name` *is not. Suppose that we have a workload of queries shown in Figure 2 that essentially treat the* `Salary` *column as a blob; every*

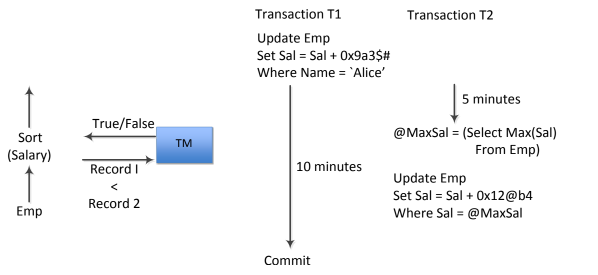

Figure 3: Information Leakage Through Computations on Encrypted Columns

query references the employee by name and updates its salary to a new strongly encrypted value (we show queries instead of plans for ease of exposition). Suppose that the adversary's knows as part of background knowledge that (1) salaries of contracted employees are generally larger than salaries of hourly-wage employees, and (2) salaries of hourly-wage employees are updated more often (since the number of hours can vary from week to week) than those of full-time employees. By observing the frequency with which the salaries of different employees are updated, the adversary learns about the ordering of employee names by salary.

In Cipherbase, we treat the information leakage as illustrated above also as an inevitable consequence of the user-defined static security options.

We now turn to information leakage through *computation* on encrypted columns which is our main focus in developing protective mechanisms. The most straightforward illustration of the above leakage is the disk encryption technique discussed in Section 1 which is the state of the art data encryption technique in the industry. Here, data is decrypted when it is read from disk into the buffer pool and the rest of the engine proceeds as usual. This technique is insecure against the adversary we are studying since the data is kept encrypted only on disk; everything else including the contents of main memory and the communication with the client is in the clear. It is possible to improve significantly upon the above technique by keeping the data encrypted across the stack — data is kept encrypted in the buffer pool, query intermediate results are encrypted and the final query results communicated to the client are encrypted. Computations over encrypted data are performed by using the TM. While encrypting data across the stack is sufficient to protect the data from casual intrusions, it reveals information to a sophisticated adversary who observes the *access patterns* of data movement. We illustrate through examples.

EXAMPLE 3.2. *We continue with the employee table, `Emp` (`Name`, `Salary`) where `Salary` is sensitive, so it is strongly encrypted, and `Name` is public and not encrypted. Consider a query that orders employees by salary. The execution plan on the left of Figure 3 illustrates an execution plan for this query. Since the data is encrypted both on disk and in memory, the TM is used to compare records. An adversary who observes the sequence of events during query execution can infer the ordering of employee names by salary even though the `Salary` column is strongly encrypted, both on disk and in memory.*

Now we present a more subtle form of information leakage. Consider the two concurrent transactions, T_1 and T_2, shown on the right in Figure 3 (we show queries instead of plans for ease of exposition). T_1 updates the salary of `Alice`, waits for 10 minutes, and then commits (we note that relevant constants are encrypted since the `Salary` column is encrypted). T_2 starts after T_1's update and attempts to update the salary of the highest paid employee. If T_2 is running in a read-committed isolation level, T_1 blocks T_2, and an attacker has access to T_1's and T_2's execution plan, then the attacker can infer that Alice is the highest paid employee.

3.4 Implications For System Design

Similar to static security, we develop more than one option for the runtime. We first present the design of a system that: (1) has user-defined confidentiality (and hence, orthogonal), and (2) significantly improves upon TDE by encrypting data across the stack, but at the same time does *not* hide access patterns, thereby permitting information leakage of the form illustrated in Example 3.2. We refer to the above setting as *basic* security. Basic security does protect the data from casual intrusions and we believe that there are many applications for which this setting is acceptable. We note that engineering the DBMS to efficiently support the above setting is challenging. We discuss the challenges and our design in Section 5.

We also develop the mechanism of an *oblivious* operator that hides access patterns and hence lets us control the leakage from computations on encrypted columns. For example, we can prevent the information leakage illustrated in the sort example using an oblivious sort operator. We design oblivious implementations of all relational operators. Section 5 presents oblivious operators and discusses higher security settings that utilize them. While the notion of an oblivious operator is a promising direction in helping increase the security of the system, we note that higher security does come at a potentially greater cost and restricts the functionality that can be wholly supported in the server. In particular, we address the information leakage through concurrent execution illustrated above by incorporating client-side processing.

4. FPGA BASED TRUSTED MACHINE

This section describes: (1) the potential security and practical benefits of reconfigurable hardware devices such as FPGAs for the Trusted Machine (TM), (2) techniques to secure FPGAs, and (3) our methodology for establishing a strong trusted compute resource in the FPGA.

4.1 Platform Alternatives

There are multiple potential platforms that can be used to implement the TM. To be suited for an orthogonally secure system like Cipherbase, such a platform should meet three requirements. First, the platform must be secure. By design, some hardware platforms are more difficult to attack than others. Second, the platform should offer high performance, both in terms of computational speed and communication bandwidth. This is because we would like to minimize the effect of shipping computations and data between the TM and UM. Third, the platform should be programmable and re-configurable. This allows us to support future releases of Cipherbase with more powerful features and optimizations. The remainder of this sub-section assesses four different hardware platforms with regard to these three requirements. As mentioned in the introduction, we chose FPGAs for Cipherbase because it best meets these three requirements.

General-Purpose Processors (GPP): Although versatile, available conventional processor-based systems such as a Single Board Computer have two characteristics that make them fundamentally more difficult to secure as compared to more specialized hardware platforms. First, GPPs are built with a single, physically unified memory space for both program and data. Although this makes these systems more adaptable when the intended use is not known *a priori*, this feature is also specifically exploited by attacks such as buffer overruns and rootkits that can defeat or sidestep memory protection mechanisms. Custom hardware solutions can be specialized to an application, completely separating the specification of the computation and the data flowing through that computation.

A second issue with GPPs is that they are inherently sequential with centralized control. This means that even basic operations such as reliably handling system I/O while computing requires multi-tasking. This feature implies an operating system and a certain level of complexity to implement context switching and time-slicing. Complexity does not necessarily lead to poor security, but it increases the potential surface area for coding mistakes and attack. Furthermore, time-multiplexing a single resource introduces the possibility of undesired side-effects and the intermingling of data. Custom hardware on the other hand, can be inherently parallel. Although more specific to a given operation (e.g. an I/O protocol), these circuits can be relatively simple, running completely independently from one another. Any communication between these circuits will be explicit and well-formed.

Secure Co-Processors (SCP): Secure co-processors have been used in prior work (including TrustedDB [2]) in settings requiring secure state and computation. SCPs can offer better security than general-purpose processor systems because they offer a small amount of secure non-volatile memory (e.g. to securely store keys within the processor) and come pre-installed with a restricted OS [16]. That said, some of the same characteristics discussed for GPPs can hold true for SCPs.

More importantly for the purposes of Cipherbase, existing SCPs are built for non-performance critical applications such as use in cash machines. Thus, they are unsuitable for high-throughput systems such as Cipherbase. As a concrete example, TrustedDB [2] reports AES decryption (a common operation in Cipherbase) rates on the order of tens of MB/s. At the same time, dedicated hardware implementations are easily capable of multiple GB/s [8].

Hardware Security Modules (HSM): Similar to the above platforms, HSMs are also small, self-contained expansion cards. Unlike SCPs, though, they typically use high-speed dedicated logic for computation rather than embedded processors. This dedicated logic addresses many of the performance issues. Furthermore, platforms built from dedicated logic also have security advantages. For example, processor-based systems traditionally operate with a program counter that iterates though a program held in memory to implement computation. This memory-based instruction execution opens the door to attack since malware may be able to manipulate the contents of this memory. Dedicated logic on the other hand is typically built from individual hardware state machines, hardwired to implement a computation. This makes modifying the execution of dedicated logic fundamentally more difficult, particularly as we are primarily concerned with network-based attacks in which the adversary does not have the physical access to the device in order to manipulate electrical connections.

At the same time though, existing HSMs are built as specialized, essentially black-box appliances. Although this may be appropriate for specific common operations such as generic encryption offload, different workloads will generally require different HSMs. This creates many practical issues. For example, the simple cost of multiple different cards is a non-trivial barrier to entry. Also, since cloud machines will generally have very few expansion slots, any given server may be unable to support all necessary cards simultaneously. This limits the migration of applications between machines, compromising cloud scaling and failover. Furthermore, since server farms often operate lights-out, installing new cards presents a logistical problem. This makes it difficult to support future customer applications.

Field Programmable Gate Array (FPGA): FPGAs are readily available programmable hardware devices that can combine the flexibility of processor-based systems with the security and performance of dedicated hardware. Like HSMs, we can implement the TM as hardware state machines. At the same time, the logic itself is built from volatile configuration memories and the binary that actually defines the computation is loaded at power-on from an external non-volatile memory. This allow us to change the computation quickly and easily.

4.2 Securing FPGAs

Since the behavior of an FPGA is dynamically defined by the binary that is loaded, the binary (also known as a bitstream) must be protected from alteration, reverse-engineering or duplication. This holds true for virtually all commercial applications, beyond specific security-oriented systems such as Cipherbase. Towards this end, FPGA device manufacturers have developed binary protection schemes. These techniques have been used for years and have become industry standard.

Specifically, as shown in Figure 4, hardware developers can create a unique AES key for each device and program this key into a small non-volatile, externally write-only memory inside the FPGA. This is performed after they receive it from the manufacturer but before they ship the finished product to the customer. The hardware developer then encrypts and signs the binary with this key and a hash-based message authentication code (HMAC), creating a bitstream unique to the specific device. The binary is then loaded into a non-volatile memory external to the FPGA and the system is deployed.

When the FPGA is powered on, the device will attempt to load a binary from the external memory. The FPGA decrypts and authenticates the binary using onboard dedicated decryption logic and the previously programmed AES key. This key can only be read by the decryption logic and is not accessible to any other circuitry, internal or external to the FPGA. If the binary can be properly decrypted and authenticated, the programming succeeds and the FPGA is ready for use. If not, the device enters an error state and will not function until provided with a valid bitstream. Since the encryption key and the decrypted binary are held only within the device, this process is generally considered to offer very high protection. Notice that since the binary is held encrypted and signed in external memory, the hardware developer (or other authorized party with access to the key) can easily distribute new binaries without compromising security.

Also note that despite the programmability of the platform, like conventional hardwired logic, FPGAs can maintain a strict separation between the "program" address space that defines the computation and the "data" address space that contains the values that are computed upon. Discounting special programming ports that must be explicitly instantiated in a binary that is loaded, the configuration memory and the programming pins for the FPGA are disjoint from normal I/O pins and the logic fabric itself. As Cipherbase does not use such ports, this is beyond the scope of this paper and we consider the two systems completely isolated from one another.

4.3 Building the TM

As shown in Figure 4, setting up the TM begins with a trusted authority (TA). This is a third party that both potential clients and the cloud operator trust. The TA generates and maintains the FPGA binary encryption keys. They also vet and compile the hardware code associated with the TM and create the encrypted and signed binaries for each device. As mentioned earlier, the TM also needs

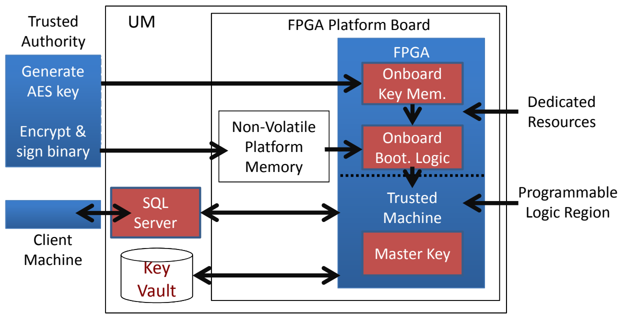

Figure 4: Trusted FPGA TM

a separate encryption key to perform database operations. In the simplest scenario, the TA creates and embeds a "master key" into the TM binary and distributes it to the database client. For example, the client could use this master key to encrypt their data with AES. The TM can then use its copy of the key to directly process data. While technically functional, this simple arrangement has several limitations. First, the client is locked into using a single key for all database operations. The client may want to encrypt different columns with different keys or they may have multiple databases. Second, the TA will need to generate separate binaries, each loaded onto the device independently for every set of potential database clients.

A more sophisticated system instead uses this master key to allow clients to bootstrap their own keys. In this case, the TM would embed an RSA public/private master key pair. The FPGA's public key is then published via standard public key infrastructure techniques, allowing clients to uniquely identify a given FPGA. Clients could then negotiate specific AES "session" keys for different database fields or different database applications. These session keys could be re-negotiated by the client before each transaction, or they could be cached locally in the cloud in a "key vault", maintained by the UM. In this case, the FPGA would encrypt each session key with its master key (either the aforementioned RSA key or another key defined by the TA) such that only it could recover the contents of the key vault. When the UM receives a query for a given database, the UM would transfer the encrypted session key to the TM along with the encrypted data. Additional information regarding the secure bootstrapping and operation of FPGAs can be found in [5]. Beyond this, additional information regarding the use of secure hardware can be found in [11].

5. QUERY EXECUTION RUNTIME

This section presents the Cipherbase runtime system. We start by presenting in Section 5.1 our system design for basic security. Higher security that hides access patterns is discussed in Section 5.2.

5.1 Basic Security

As discussed in Section 3, basic security improves upon encryption at rest by keeping data encrypted through the DBMS stack, while at the same time doing all processing in the server. The basic idea is to use the TM to run computations on the data. Engineering an efficient infrastructure that supports basic security involves two broad challenges. The first challenge is architectural — how is the processing to be divided between the UM and TM? The second inter-related challenge is performance. This section discusses each of the above challenges. We finally discuss the security offered by the basic system.

Plaintext Operation	*Primitive in TM*
$\sigma_{A=5}$	Dec($\overline{A}$) =Dec($\overline{5}$)
π_{A+B}	Enc(Dec($\overline{A}$) + Dec($\overline{B}$))
$\bowtie^{hash}_{A=B}$	Hash(Dec($\overline{A}$)); Dec($\overline{A}$) =Dec($\overline{B}$)
AGG(SUM(B))	Enc(Dec($\overline{B}$) + Dec($\overline{partialsum}$))
INDEX OPERATIONS	FINDPOS(Dec($\overline{k}$), $\langle$ Dec($\overline{k_1}$), …, Dec($\overline{k_n}$) $\rangle$)
RANGELOCK	Dec($\overline{v}$) $\in$[Dec($\overline{l}$), Dec($\overline{h}$)]

Figure 5: Common plaintext operations and corresponding primitives in TM to support the same operation in ciphertext

5.1.1 System Architecture

The UM and the TM constitute an asymmetric distributed system: the TM is secure but since it is based on specialized hardware is resource limited, while the UM is (potentially) insecure but powerful. This asymmetry holds independent of whether we use FPGAs or other alternatives, and argues for a design that minimizes the TM footprint.

One possibility is a loosely coupled architecture studied in prior work in the TrustedDB system [2]. TrustedDB runs a full database system SQLite in the TM for query processing over encrypted data and another, MySQL, in the UM for query processing over plaintext data. The loosely coupled approach uses limited computational resources at the TM for functionality (e.g., disk spills in hash join) that does not depend on encryption. If all columns are encrypted *all* query processing in TrustedDB happens within the TM.

In contrast, Cipherbase adopts a tightly coupled design. We revisit each module of the database system and identify core *primitives* that need to operate on encrypted data and factor these out to be implemented in TM. The goal is to minimize the work that is to be done in the TM. Interestingly, a small class of primitives involving encryption, decryption, and expression evaluation suffices to support query processing, concurrency control, and other database functionality.

Figure 5 lists, for a few typical operations, the primitives that are invoked in the TM during the execution of the operation. Here, $\overline{A}$ indicates the ciphertext of A and Enc, Dec, and Hash represent encryption, decryption, and hashing, respectively. For example, we use the following (single) primitive to implement indexing over encrypted columns: given an encrypted (index) key, find its position in an array of encrypted (index) keys.

The Cipherbase server runs an extended database system (SQL Server) with some components modified to make round-trips to the TM for operations over encrypted data. The main advantage of this design is that we are able to leverage the relatively powerful UM for operations that do not depend on encryption, even when data is strongly encrypted.

We illustrate through two examples. One is (B$^+$-Tree) indexing. The index is stored encrypted in the UM. An index lookup is broken down into index page searches. The index page search requires access to clear-text and hence is performed in TM. Index lookup invokes the FINDPOS primitive for each index page traversed to identify the next page to visit. All other indexing logic including concurrency control, recovery and also index update are performed almost wholly in the UM (the solitary exception is checking range predicates on index keys in order to perform key value range locking in order to address phantom reads). Similarly, the hash join operator uses the TM for computing hashes and checking equality. The rest of hash join logic—memory management, writing hash buckets to disk, and reloading them—runs in the UM.

The second advantage is software engineering: by attaching hooks to a small number of places in the SQL Server code and routing data to the TM, we are able to get rich functionality of an industrial strength database system. The third advantage, as discussed earlier, is that by keeping TM functionality simple, we are able to leverage the power of FPGAs.

Design of the TM: While the set of primitives that we support in the TM is small, involving encryption, decryption, and expression evaluation in the TM, Cipherbase is a complete general-purpose database system on strongly encrypted data; e.g., expressions in a projection could be arbitrarily complex and involve a variety of types. To be able to support these we design the TM as a *stack machine* that can be programmed to evaluate complex expressions. In fact, we adopted this design from the (existing) SQL Server product because this design is useful even in traditional database systems. Figure 6 illustrates the stack machine instructions used to evaluate the (parameterized) expression $\mathsf{Dec}(\$0) = \mathsf{Dec}(\$1)$. The parameterization in this example also highlights the separation of program and data. The program needed to evaluate instances of TM primitives is derived at query compilation time; the runtime merely supplies parameters to the program. In the FPGA, we directly implement in hardware each instruction supported by the stack machine. The details include dedicated circuitry for encryption, decryption, and basic data operations, and various parallelization optimizations such as those in [18].

5.1.2 Optimizations

One efficiency challenge we face is to ensure that the TM is not a bottleneck. Since the latency to the TM is large, one strategy for meeting the above challenge is to batch requests to the TM. We explore both inter-query and intra-query batching. In an OLTP workload with multiple concurrent transactions, we could batch requests from multiple transactions to the TM. In an OLAP workload where large amounts of data are scanned, we explore intra-query batching where, for example, every operator sends multiple records (typically with the same instructions) in a single round trip to the TM. Another general strategy to improve performance is to exploit the programmability and parallelism of the FPGA as studied in prior work [18]. We note that the design of the TM as discussed above is stateless. Extending the TM to be stateful opens up new optimizations — for instance, aggregation can be made faster by storing partial aggregates in the TM. Saving state in the TM not only reduces round trips but also reduces the number of encryption and decryption operations to be performed.

Another important set of optimizations is centered around the physical design of the database in order to reduce the storage overheads of encrypted data. Consider the case where every fixed length column (say 32 bits) is encrypted independently to a value that is 128 bits, this would lead to a storage overhead and a reduction of scan bandwidth for these columns by a factor of 4. In order to improve the effective bandwidth, we also consider optimizations to "batch" data tuples before encryption. We study multi-row encryption techniques where multiple values in a column are concatenated and then encrypted. While multi-row encryption would eliminate the storage overhead, extending the DBMS stack to support multi-row encryption is clearly challenging.

5.1.3 Security

We now briefly discuss the overall security implications of basic security for strongly encrypted columns. We can think of basic security as follows. The DBMS uses various operations in order to run queries — does a given record satisfy a given predicate, do these two records join, do these records share the same value in the grouping columns, etc. The TM is the oracle that answers the above questions without revealing the clear-text. Therefore, the information revealed is the results of operations used during query processing. A filter operator reveals identities of records satisfying the filter predicate. A sort operator reveals the ordering of records and a join operator reveals the join graph. If no operations are performed on a column, no information is revealed (besides what is inevitable, as discussed in Section 3). We note that concurrency makes the information leakage no worse; for example, in Figure 3, the execution of transaction T2 itself reveals the employee who has the maximum salary. The security of our basic system is similar to the security offered by CryptDB [19]. However, there are key differences. Even when computation is performed on a column, we only reveal information about the subset over which computation happens. For instance, if a subset of a table is sorted, we only reveal the ordering for the subset; in contrast, CryptDB reveals the ordering of the whole column. Second, we do not change the data storage, whereas CryptDB changes the data on disk to use weaker encryption and hence reveals information even to a weaker adversary who only has access to the disk. Finally, we note that in terms of functionality our basic system is complete whereas CryptDB is not.

Id	*Instruction*
1	`GetData` $0
2	`Decrypt`
3	`GetData` $1
4	`Decrypt`
5	`Compare`

Figure 6: Stack Machine for $\mathsf{Dec}(\$0) = \mathsf{Dec}(\$1)$

5.2 Higher Security

While we believe basic security is an acceptable level of data confidentiality for a wide class of applications, as noted before it does not hide access patterns. In this section, we first introduce our key mechanism to hide data access patterns namely the notion of an *oblivious* operator, and then discuss security settings that utilize this mechanism.

5.2.1 Oblivious Operators

We define an implementation of a relational operator to be *oblivious* if it reveals nothing about the data other than its input and output sizes, thereby hiding access patterns. We can adapt previous work [9] to design oblivious implementations of some relational operators such as sort and filter. An oblivious sort for example performs the same set of comparisons independent of the input collection to be sorted, hence hiding access patterns, and can be performed with the same external memory complexity as a standard sort [9]. It is possible to design an oblivious filter using oblivious sort as follows. In the first pass over the data, we tag every record with an encrypted boolean flag indicating whether the record satisfied the filter. We then perform an oblivious sort by the boolean flag and return the prefix of records satisfying the predicate. We can show that the above algorithm yields an oblivious filter.

In Cipherbase, we have developed oblivious implementations of all scan-based relational operators including joins, anti-joins and grouping-aggregation [1]. We illustrate an oblivious implementation of the groupby-aggregation operator for the special case of a single grouping attribute and `COUNT(*)` aggregate (our algorithm can be generalized to handle more general grouping and all standard aggregation functions.) Traditional sort-based grouping and aggregation with the sort step replaced with an oblivious sort step is not oblivious since it reveals the size of each group. Our imple-

Algorithm 1 Grouping and `COUNT(*)` aggregation of $R = \overline{r}_1, \ldots, \overline{r}_n$ with a single grouping attribute A.

```
1: procedure ObliviousGroupAggr($R$, $A$)
2:     $R_{sort} \leftarrow$ oblivious sort of $R$ on $A$
3:     $curA \leftarrow$ null
4:     $curCount \leftarrow 0$
5:     $\mathcal{G}_i \leftarrow \phi$                    ▷ Output with dummy records
6:     for all $\overline{r}$ in $R_{sort}$ do
7:         if $r[A] = curA$ then
8:             $curCount \leftarrow curCount + 1$
9:             Append $\overline{\langle dummy \rangle}$ to $\mathcal{G}_i$
10:        else
11:            Append $\overline{\langle curA, curCount \rangle}$ to $\mathcal{G}_i$
12:            $curA \leftarrow r[A]$
13:            $curCount \leftarrow 0$
14:        end if
15:    endfor
16:    Output $\mathcal{G}_i$ with $\overline{\langle dummy \rangle}$ removed (oblivious filter)
17: end procedure
```

mentation is a slight modification and is shown in Algorithm 1. We begin by obliviously sorting input stream R on grouping attribute A (Step 2). As in traditional aggregation, we scan the sorted stream and compute the counts of each group. The traditional aggregation produces one output tuple per group after the last input tuple belonging to the group has been processed (Step 11). Our modification is to produce dummy output tuples for the other input tuples of a group as well (Step 9). An oblivious filter is used to remove dummy tuples and get the final output (Step 16). The obliviousness of the operator follows since the input output pattern of the operator is independent of the contents of R, and the overall time and data complexity is the same as traditional sort-based group by aggregation. Similarly, our oblivious implementations of other relational operators also have almost the same data complexity as the traditional counterparts. The impact on the instructions supported by the TM is modest. For example, in order to implement oblivious sorting, the TM needs to be able to sort a block of records.

Finally, we note that the problem of oblivious indexing reduces to previously proposed oblivious RAM technology [7] that leads to a loss of spatial and temporal locality of reference and hence has potentially serious performance ramifications. We are investigating ways of making oblivious indexing practical along the lines of recent work [22].

5.2.2 *Security Settings*

We now explore security settings that utilize the mechanism of oblivious operators. By running a query with oblivious operators, we obtain an execution plan where the only information that may be leaked is aggregate statistics such as the intermediate result sizes. We therefore investigate a security setting corresponding to the above execution strategy. The meaning of the setting at a column-level is that the query evaluation can only reveal result sizes for all sub-expressions involving the column.

Another interesting security setting is where we are required to leak *no* information about a column, in addition to the inevitable leakage that happens from the static security setting as discussed in Section 3. While the above setting is essentially *equivalent* in security terms to treating the column as a blob on which no computation is performed, we can support an interesting class of queries wholly in the server by combining oblivious operators so long as the output sizes are determined either by the schema (which we assume to be known for the purposes of analyzing information leakage formally) or by the execution plan of the query. For example, consider the sort example in Figure 3. The output size of sorting is the same as the input size and hence reveals no additional information about the data. In general, we can support a larger class of queries at this setting including foreign-key joins, grouping, sorting and top-k.

Finally, we note that the both of the above higher security settings restrict the functionality that is wholly supported in the server. In particular, we address updates of the form shown in Figure 3 currently by incorporating client-side processing.

6. QUERY COMPILER

The Cipherbase query compiler extends the SQL Server query compiler. Just like the SQL Server (and most other compilers), it is composed of three main phases: (a) parsing, (b) (cost-based) optimization, and (c) code generation. Cipherbase extends all three phases; in particular, it uses the extensibility of the Cascades framework on which the SQL server optimizer is based [12]. What makes the Cipherbase compiler different are the following requirements:

- *Secure optimization and statistics:* As shown in Figure 1, the ODBC driver at client machines carries out query compilation and query optimization because this process can leak information such as statistics about the distribution of values in the database. A number of engineering challenges arise to effect such client-side query compilation. One issue is that the API of the server needs to be extended to run a plan shipped from the client instead of a query. Another issue is the maintenance of database statistics to be used as part of the cost model. Cipherbase stores statistics persistently at the server in an encrypted form and caches them in client machines in clear text. Furthermore, client machines generate statistics based on samples.

- *Enforcing security:* In Cipherbase, not every query plan is legal. Query plans that leak more information than allowed by the user-defined options are not legal and must be precluded even if they are feasible and return correct results. The Cipherbase optimizer, therefore, must be *security-aware*. We achieve this goal by providing additional *plan annotations* that keep track of the encryption of each column generated by the results of a plan and extending the enumeration rules to respect the client's requirements.

- *Optimizations:* In a system like Cipherbase, several optimizations are useful that are not useful in a traditional database system. Examples are specific join methods and reorderings that try to avoid multiple trips back and forth from the TM. To effect these optimizations, we choose to model the Cipherbase server as a distributed system with two nodes, the UM and the TM. This modeling does not reflect reality because the execution of most query operators involves both the UM and TM in a collaborative way. This abstraction, however, helps to reason about all the new optimizations and extend the SQL Server cost model for Cipherbase.

To get a feeling for the extensions implemented in Cipherbase, the remainder of this section describes the Cipherbase plans and some of the Cipherbase-specific optimization techniques.

6.1 Canonical Plan

The first step in implementing the Cipherbase compiler is to come up with a new *canonical query plan* that is generated by the Cipherbase SQL parser. Figure 7 shows an example canonical plan for a three-way join query with a group-by. The canonical query plan of Figure 7 specifies that all operators are executed in the TM and it explicitly models the movement of tuples from the UM to the TM using *decrypt* operators and back from the TM to the UM

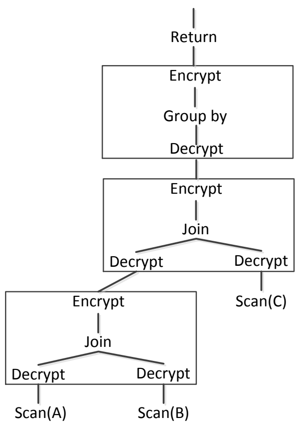

Figure 7: Canonical Plan

using *encrypt* operators. The canonical plan is secure because it works even for a database that is strongly encrypted and for a TM with limited main memory. However, it may not be the optimal plan because it relies heavily on operations in the expensive TM. So such a canonical plan is only the starting point for further optimizations.

6.2 Example Optimizations

There are a number of optimization techniques that make sense for Cipherbase and that typically do not make sense in a traditional relational database system. We list a few below.

- *Traditional:* Traditional optimizations such as join ordering and different join methods are just as useful in Cipherbase as in a traditional database system.
- *Move to UM:* If data is public or encrypted in a partially homomorphic way that matches the operation (e.g., order-preserving encryption), then the operation can be executed wholly by UM.
- *Exploiting Security Annotations:* The optimizer uses the security annotation to make decisions such as encryption of any intermediate results as well as the choice of physical operators.
- *Merge:* If there is sufficient memory available in the TM, then two operators can be executed in the TM without shipping tuples back and forth from the TM to the UM.
- *Semi-joins:* Since memory is a scarce resource in the TM, it is often better to project out only the relevant columns for an operation to the TM and/or to even apply semi-join programs, just as in a distributed database system.
- *Predicate Migration:* Executing predicates on encrypted data is expensive. So, all work on optimization of queries with expensive predicates or UDFs is applicable [14].

It is often the combination of applying these optimizations that give the biggest performance boosts. Furthermore, all these optimizations and the generation of security-aware plans can impact the join order. In a traditional database, for instance, it might be best to choose the following join order for a three-way join query: $(A \bowtie B) \bowtie C$. If $A \bowtie C$ can be carried out entirely in the UM (because the join attributes are public) and the join with B involves the TM (because B's join attribute is private), then Cipherbase may choose the following join order: $(A \bowtie C) \bowtie B$. Since SQL Server already enumerates all join orders, we do not have to extend join-ordering logic for Cipherbase: We only have to extend the cost model that costs out the different options to enable the Cipherbase optimizer to find the best join order and apply the best combination of optimizations.

7. CONCLUSIONS

In this paper, we presented an overview of Cipherbase which is a complete SQL database system that allows organizations to leverage the advantages of cloud computing and at the same time maintain the confidentiality of sensitive data. It achieves orthogonality (the system provides full functionality independent of the chosen security policy) with a tightly coupled hardware / software co-design that integrates FPGAs as trusted hardware into Microsoft's SQL Server database system. This paper described the architecture of Cipherbase and some of the most important design considerations. Cipherbase is still under active research and development at Microsoft Research.

8. REFERENCES

[1] A. Arasu, R. Kaushik, and R. Ramamurthy. On the security of querying encrypted data. Under submission.

[2] S. Bajaj and R. Sion. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *SIGMOD*, 2011.

[3] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *CRYPTO*, 2011.

[4] Dropbox. http://www.dropbox.com.

[5] K. Eguro and R. Venkatesan. FPGAs for trusted cloud computing. In *FPL*, 2012.

[6] C. Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3), 2010.

[7] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[8] T. Good and M. Benaissa. AES on FPGA from the fastest to the smallest. In *CHES*, 2005.

[9] Michael T. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *SPAA*, pages 379–388, 2011.

[10] Amazon Web Services LLC. AWS GovCloud. http://aws.amazon.com/govcloud-us/.

[11] V. Goyal, Y. Ishai, A. Sahai, R. Venkatesan, and A. Wadia. Founding cryptography on tamper-proof hardware tokens. In *TCC*, 2010.

[12] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3), 1995.

[13] H. Hacigümüs, B. R. Iyer, et al. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*, 2002.

[14] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *SIGMOD*, 1993.

[15] Hardware Security Module. http://en.wikipedia.org/wiki/Hardware_Security_Module.

[16] IBM Corporation. IBM Systems cryptographic hardware products. http://www-03.ibm.com/security/cryptocards/.

[17] National Institute of Standards Information Technology Laboratory and Technology. Security requirements for cryptographic modules, 2001.

[18] R. Müller, J. Teubner, and G. Alonso. Data Processing on FPGAs. *PVLDB*, 2(1), 2009.

[19] R. A. Popa, C. M. S. Redfield, N. Zeldovich, et al. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.

[20] Radu Sion. Query execution assurance for outsourced databases. In *VLDB*, 2005.

[21] Danil Sokolov, Julian Murphy, Alexander Bystrov, and Alex Yakovlev. Design and analysis of dual-rail circuits for security applications. *IEEE Transactions on Computers*, 54:449–460, 2005.

[22] E. Stefanov, E. Shi, and D. Song. Towards Practical Oblivious RAM. *CoRR*, abs/1106.3652, 2011.

[23] Germany Tackles Tax Evasion. Wall Street Journal, Feb 7 2010.

[24] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. In *SIGMOD*, 2009.