# Incorporating a Secure Coprocessor in the Database-as-a-Service Model

Einar Mykletun and Gene Tsudik
Computer Science Department
School of Information and Computer Science
University of California, Irvine
{mykletun,gts}@ics.uci.edu

## Abstract

*In this paper we suggest an extension to the Database-as-a-Service (DAS) model that intrdocues a secure coprocessor (SC) at an untrusted database service provider in ordet to overcome drawbacks in the plain DAS model. The processor serves as a neutral party between the clients and service providers with the goal of increasing security of outsourced data. Additionally, it supports a much broader range of queries performed and reduces both bandwidth and computational burdens on the client. We expect these improvements to make the DAS model more viable and attractive from a client's perspective.*

## 1 Introduction

The Database-as-a-Service (DAS) model [7] is a manifestation of the more general Software-as-a-Service trend which is becoming increasingly popular. In the DAS model, the client's database is stored at the service provider. The provider is responsible for provisioning adequate software, hardware and network resources for the client's database as well as support the various system administration tasks.

This model, depicted in figure 1, has several benefits to offer, which are all characteristic of the Application Service Provider model. (1) By outsourcing databases, clients can reduce the total cost of operation since they are no longer required to invest in the infrastructure as well as the personnel required to maintain the complex databases. (2) The service provider can offer the service to multiple clients and thus can amortize the cost (3) The clients can be assured of the quality of service that has been guaranteed by the service provider.

There are several new research challenges that are posed by the Database-as-a-Service model which influence the overall performance, usability and scalability of the system. A fundamental challenge associated with the DAS model is the security of the stored data. The clients store their data, which is arguably the most valuable asset of any organization, with an external, potentially untrusted, service provider. Therefore, it is essential to guard the privacy of the data, not only from malicious outside attackers but also from the service provider itself, with the use of adequate security measures.

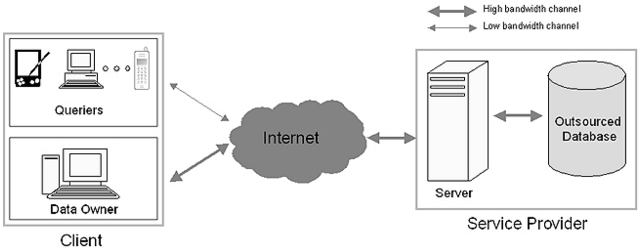

**Figure 1. DAS Model Overview**

In [6], the authors propose a novel architecture to run SQL queries on the encrypted data directly. In this model, the client stores the encrypted database at the DAS server and locally stores some information which is referred to as *metadata*. When the client needs to run an SQL query, it transforms the query, based on the metadata, into two separate queries, $Q_s$ and $Q_c$. $Q_s$ is executed at the server directly over encrypted data and $Q_c$ is run by the client locally on the data received from the server as a result of the first query.

Although the DAS approach seems promising, it also suffers from 3 major limitations, namely bandwidth overhead, limited server-side query support and overloaded clients. These stem from that the server is only able to execute restricted portions of the database queries over the encrypted data, which in turn results in a superset of tuples to be sent to the client who consequently needs to fulfill the remainder of the query. This places a bandwidth and computational responsibility upon the client that may be too great, especially if the client device is limited in its computation and energy resources.

In this paper we specify how the use of a Secure Coprocessor at the server can help overcome current problems in DAS and make the model more attractive. A secure coprocessor (SC) is a general-purpose computer that can be trusted to carry out its computations unmolested, even if an adversary has direct physical access to the device [3]. Such a device can be placed at the server to act as a neutral computing unit that can assist in providing the client with credibility that queries are being run correctly, while reducing the bandwidth and computational requirements at the client by executing a larger range of query types.

**Contribution**: With this paper we propose a framework for improving the DAS model by incorporating a secure coprocessor at the untrusted database service provider such as to overcome current limitations of DAS. Although current coprocessors do not meet the necessary technical specifications required for a truly efficient solution, we feel that our extension makes the whole DAS model more viable and attractive (due to advantages in performance and usability, i.e., wider range of supported queries).

**Organization**: Section 2 provides an overview of the DAS model. In Section 3 we introduce the concept of secure coprocessors, while the next section describes how to incorporate them in the DAS model. We review related work in Section 5 before concluding the paper in Section 6.

## 2 Database-as-a-Service

As described in the introduction, DAS is a specific instance of an outsource database model whereby clients who do not have the necessary resources to manage their own databases choose to outsource them to database service providers. The providers take on the responsibility for provisioning adequate software, hardware and network resources for the client's database as well as support the various system administration tasks. However, providers who gain complete access to the clients' data may not be trustworthy as they might store databases belonging to competing clients or simply have their own malicious intentions. In order for clients to have a high enough level of confidence in the privacy and integrity of outsourced data, they need to be assured that data is protected from both malicious outsiders and the database service provider itself.

One natural choice for ensuring data privacy is to use a strong encryption algorithm. The client encrypts the database using a symmetric-key encryption algorithm – such as AES[11] which is ideal for bulk data encryption – and stores the it at the service provider. Each time the client needs to execute a SQL query, it first obtains required tables from the server, decrypts the data and runs the query locally. While this guarantees security, it is extremely expensive both in terms of computation (from the client's perspective) as well as communication, thus rendering the model utterly inefficient. Ideally, what we need is a way to store the data in encrypted form, while still enabling the bulk of the SQL processing to be performed by the server, and, at the same time, protect the data from both inside and outside adversaries.

### 2.1 Query Processing

The authors of [7] propose a novel architecture to run SQL queries on the encrypted data directly. In this model, a client pre-processes its data by bucketizing it according to certain metadata. A bucket identifier is appended to each sensitive numeric attribute that is to be encrypted. These tags identify the range of values that the attribute belongs to, i.e. a salary attribute of $42,000 may be classified as belonging to the range $40,000 to $45,000. The attributes (but not the buckets id's) are encrypted with the desired cryptosystem to which only the client knows the decryption key, and the resulting database is transmitted to the service provider. Note that the provider is not given the metadata and is therefore unaware of the specific bucketization that has been used, and is therefore somewhat oblivious to the meaning of the bucket id's that accompany the database attributes.

When the client needs to run an SQL query, it first uses its metadata to transform the query into *server side* and *client side* queries, denoted by $Q_s$ and $Q_c$ respectively. $Q_s$ is represented in terms of the bucket identifiers, enabling the server to perform comparative operations, such as $\leq, \geq, =, \neq$, directly on the encrypted data, although at a coarse level. Due to the limited granularity that results from the bucketization, the server selects false positives and produces a superset of tuples while executing $Q_s$. These are returned (in encrypted form) to the client. Upon receiving these tuples, the client decrypts the result set and runs $Q_c$ on the plaintext data to complete the original SQL query, which may involve filtering out the false positives.

### 2.2 Limitations in DAS

The 3 major limitations of the DAS model are (1) the bandwidth overhead between the server and clients, (2) the risk of overloading clients with computations (query processing) and (3) the limited server-side query support. These stem from that the server is only able to execute restricted portions of the database queries over the encrypted data, which in turn results in a superset of tuples that are sent to the client who consequently needs to fulfill the remainder of the query, placing possibly large bandwidth and computational responsibilities.

The transmission of the superset of tuples results in a large computational and communication overhead at the client. The types of queries that can be run at the server over the encrypted data are limited to logical comparisons, thereby greatly reduces the usefulness of the server in the query processing. Specifically, operations such as data aggregation and pattern matching are not supported in full[1], resulting in a larger workload at the clients. This might be acceptable if the client is using a desktop/laptop with a high-speed network connection, but not so in case the client is a weak device such as a cell phone or low-end PDA, where battery power and computational resources are limited. Additionally, when communication is a premium, the bandwidth overhead described above becomes of even greater significance.

As can be seen from above, all the mentioned problems stem from that the server is unable to be of great use during the query processing, and the reasons for this are two-fold. Firstly, the server is working with encrypted tuples and the corresponding bucket identifiers, which limits both the accuracy of comparison operands and other query operation such as pattern matching. Secondly, the granularity of the bucketization greatly affects the number of false positives selected by the server during the execution of $Q_s$, which directly affects the bandwidth overhead and amount of necessary post-processing at the client. There are several possible ways to bucketize the data and it becomes a balance between accuracy and security. A more granular bucketization results in a data set that leaks a greater amount of information and can lead to attacks based on statistical inferences. Specifically, if the number of buckets are too numerous, it may be possible for an attacker who knows the type of data stored (for example, employees records) to map out the encrypted database according to an expected distribution. Too few buckets results in larger supersets as more false positives get included. The subject of bucketization and privacy-preserving indexing for databases is further investigated in [8].

## 3 Secure Coprocessors

A secure coprocessor (SC) is a general-purpose computer that can be trusted to carry out its computations unmolested, even if an adversary has direct physical access to the device [3]. It is equipped with a processor, non-volatile secure memory, input devices, a backup battery and is fully enclosed in a tamper-proof container (shielding any type of penetration) that can not be opened without triggering sensors that alert about an attack taking place. A SC can be installed on a computer to provide a secure perimeter in which sensitive data may be stored and processed. The security of such a device results from it being equipped with a multitude of sensors that can detect a variety of physical attacks. In the case that a penetration of the device is detected through the signaling by sensors that are monitoring possible attack venues, an alarm triggers and all contents in the secure memory is erased, possibly by destroying the physical memory chip by leaking acid onto it.

In addition to its security features, a SC may be equipped with hardware support for cryptographic operations, such as encryption, hashing and random-number generation. A logical approach to incorporating an SC into a system is therefore to store cryptographic keys and other sensitive data within the secure on-board memory and use it as a encryption/decryption device. Such solutions have already been proposed in the context of secure web-servers in which the goal is to reduce the level of trust placed on the server [10], and in secure auctions such that the buyers and sellers can avoid having to fully trust the auctioneer [12] . An SC is somewhat limited in its computing resources, both in processor speed and the amount of on-board memory available. This is a consequence of a variety of factors, but heat dissemination is one that stands out. Because the enclosure is designed to be tamper-proof, it is restricted in surface area and difficult to keep cool (because of no room for fans). The IBM 4758 SC, which is the first of its kind, is equipped with a 99 Mhz processor and 2 MB on-board memory [3], although technological advances have been realized since its introduction and are expected to continue.

Since the SC is a programmable unit, it is possible to provide it with a query execution engine. This has

[1] The authors of [6] suggest the use of homomorphic encryption functions, which allow for certain arithmetic operations to be performed directly over ciphertexts. However, due to the bucketization strategies used, it is only possible for the service provider to compute aggregates without false positives if the query range is exactly bounded by the bucket boundaries.

already been realized in [2] where the authors implement a query processor on a smartcard, a device even more restricted in its resources than the IBM 4758. One of the biggest hurdles involved with implementing a database engine on such a processor is the limited onboard storage space, resulting in that the device only can handle small databases or require an external storage device to interact with.

# 4 Secure Co-Processors in DAS

In this section we describe our proposed framework which consists of incorporating a secure coprocessor into the DAS model. The approach consists of installing a SC at the untrusted database service provider (server) to aid with the processing of confidential client queries. The client no longer bucketizes its data prior to storing it at the server. As described in section 2.2, the use of bucketization is inherently insecure (to some level) as information is leaked through bucket identifiers, and we therefore wish to completely remove this component. It is also assumed that the client can communicate with the SC, via the server, and that all such communication is sent over a secure channel, such as a VPN tunnel or an SSL connection. To set up such a channel, the client uses the public key of the SC to set up any additional necessary session keys. The client's database is encrypted with a symmetric-key algorithm well suited for bulk encryption, such as AES, and transmitted to the server. The encryption key is transmitted to and stored at the secure coprocessor, and thereby shared between it and the client.

## 4.1 Query Processing

When posing a query, a client splits it into a SC side ($Q_{sc}$) and a client side ($Q_c$) query, initiates a handshake protocol with the SC in which a session key is established and transits $Q_{sc}$ and $Q_c$ to the SC encrypted under this key. The SC, who has access to the encrypted database tuples stored at the server, runs the query $Q_{sc}$. Due to its restricted resources, and in particular its limited RAM size, it is necessary for it to use the server's storage space as its virtual memory in which it stores intermediary query results. After executing $Q_{sc}$ the SC either encrypts the results with its session key and sends the ciphertext to the client, or performs the additional $Q_c$ query and then transmits the encrypted results. In the former case the client is required to run $Q_c$ on the decrypted results, while in the latter it simply decrypts the received data.

As discussed below, the SC may be limited in the complexity of SQL queries that it can process efficiently, especially when executing operations that require several passes through large data sets (such as sorting). It may therefore be appropriate for the client to execute portions of the query, denoted by $Q_c$, upon receiving the query results from the SC. This is especially true if involving the client in the computation reduces the total query execution time (including transmission of the results). Whether or not $Q_c$ should be executed at the client depends upon upon the computational resources and bandwidth available to the client. For example, if the client is a laptop computer with a high-speed internet connection, then it may be quicker to have the SC transmit (via the server) a larger data set on which the client could execute $Q_c$ than pushing the extra computations on the SC. On the other hand, if the client is a cell phone using a GPRS connection to connect to the internet (GPRS has a theoretical maximum speed of 171.2 kbps, but much lower speeds are achieved in practice), it is probably best to send it as small a result set as possible as well as perform all computations at the SC. The client can identify its resource constraints and bandwidth capabilities at the time of posing the query, and with this information, the SC can determine whether or not to execute $Q_c$ itself.

Figure 2 depicts the envisioned DAS model with the SC and queries $Q_{cs}$ and $Q_{cs}$.

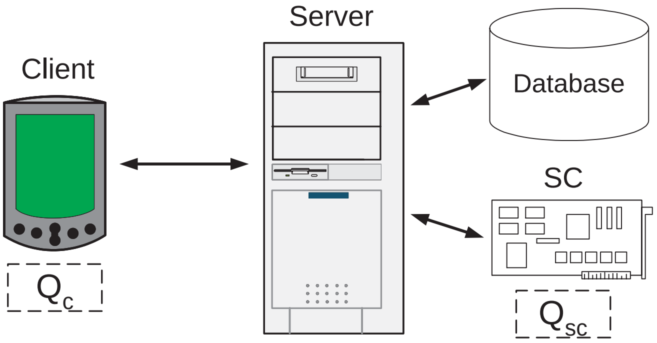

**Figure 2. The DAS Model with a Secure Coprocessor.**

## 4.2 Tuple Access Privacy

Because today's secure coprocessors are limited in on-board memory, it becomes necessary for them to use a *crypto paging* technique, which is identical to the concept of virtual memory, only that the contents are encrypted prior to being temporarily stored at the server. Crypto paging is required because the SC has insufficient memory to store intermediary query results, such as during a *Join* operation. However, as the SC utilizes the server's storage space for its crypto paging needs, a new subtle security threat arises which in turn requires a counter security service which we label as *tuple access privacy.* This refers to the technique of hiding what tuples (or blocks thereof) are fed as input to the SC and later included in query results. The frequency, contents and context of page swaps can be observed by the sever, which means that for every input to the SC the server can observe the output. We wish to hide from the server any correspondence between the data input to the SC and content of data stored during crypto paging, as this could reveal whether or not the input tuples are to be part of the query result.

Tuple access privacy might seem identical to the problem of *private information retrieval* (PIR), a topic that has received considerable attention in the cryptographic research community [5, 1, 4]. From a database perspective, the PIR problem can be stated as: "What does it take to implement a server that provides access to records in a large database, in a way that ensures access privacy and, potentially, the privacy of contents of the records themselves, even to the operator of this server?" [14]. There seems to be now way of solving this problem without using multi-round algorithms that involve obscuring the selection process by passing through the entire database. Such a solution may prove too costly for our model.

One possible solution for the tuple access privacy problem is for the SC to hide the relationship between incoming and outgoing data. This can be (partially) achieved by keeping a large enough buffer at the SC to store multiple blocks read from disk. It then becomes possible for the SC to use some randomness in selecting which blocks to write back to disk (as part of crypto paging) during query execution. For example, instead of reading one block of tuples from disk and immediately writing back those tuples that meet the query predicaments, the SC can instead delay the action of outputting tuples based on a probabilistic function or once its buffer becomes full. In addition, if the data returned to disk is encrypted non-deterministically, it becomes more difficult for an observer to match input blocks with output blocks. An extension is to incorporate more than one SC at the server, allowing for parallel execution.

We intend to pursue the issue of tuple access privacy in future work by identifying exactly what information the server can learn by observing the input and output to the SC, and what can be done to prevent any leakage of damageable information.

## 4.3 Performance

The performance of our approach very much depends upon the amount of data that is transmitted between the server and the SC, and consequently the number of calls to the encryption engine onboard the SC. It is important to identify the granularity at which to encrypt the database tuples, as it can be performed at an attribute-, tuple- or page-level. The choice affects the the overhead incurred due to encryption and the speed at which data can be encrypted and decrypted. For each run of an encryption or decryption algorithm, there is a cost associated with the initialization of key schedules and other states related to the algorithm. From a performance perspective, it is therefore beneficial to minimize the number of encryption and decryption operations performed, and this can be achieved by encrypting data in larger chunks (bulk encryption).

The work in [9] proposed a partitioned plaintext and ciphertext (PPC) secure database storage model that attempts to optimize encryption and decryption speed, while taking into account factors such as ciphertext expansion and non-deterministic encryption[2]. The main idea is to split disk pages into two minipages: one representing plaintext attributes and another – ciphertext attributes. This limits the number of encryption operations per accessed page and also limits ciphertext expansion to the size of one ciphertext block (i.e., 16 bytes for AES). An additional benefit of this approach is the separation of sensitive and non-sensitive data, allowing two categories to be queried independently and avoiding over-encryption (encrypting data that is not sensitive). The PPC approach might be an appropriate choice to speed up encrypted data access for the SC.

Query processing performance also becomes a concern when using a SC. In [14] the authors describe how the IBM 4758 SC suffers from a relatively large latency which stems from the slow bus speed on the coprocessor. Specifically, the communication between the onboard encryption engine and the input/output components proved to be a bottleneck. This is the reason why multi-pass operations on data, such as sorting and joins, are relatively inefficient to run at the SC. Temporary data structures need to be encrypted before they are placed on disk during crypto paging, and this adds to the number of calls to the encryption engine on the SC. We expect that future generation of SC's will have improved architectural features and will be able to offer much better throughput.

[2] Non-deterministic encryption refers to that multiple encryptions of the same plaintext should result in different ciphertexts.

## 5 Related Work

In [6] the authors focus on how to allow an untrusted server to run SQL queries over encrypted data in the DAS model. They make use of a homomorphic encryption scheme by Rivest et al. [13] with the property that the addition and multiplication of ciphertexts results in the identical arithmetic operations on the decrypted plaintexts, and with this scheme they are able to run certain aggregation queries. However, this cryptosystem's security relies upon an attacker's limited knowledge, specifically no knowledge about the data domain (plaintext values). Such an assumption is unrealistic in commercial settings whereby such information is standard and predictable, thereby greatly limiting their approach.

Instead of relying upon homomorphic encryption techniques, the authors in [2] propose to use tamper proof smartcards as mediators between clients and encrypted databases stored at untrusted DBA's. Data encryption, query evaluation and access right management are insulated within the smartcard and protected from tampering. Queries are split into server-side, smartcard-side and client-side portions, requiring the server to perform any portion of the query where predicates are equality comparators (=, ≠). This assumes that database entries are encrypted deterministically – encryption of the same plaintext always yields the same ciphertext – which makes the stored data vulnerable to statistical attacks by the DBA and intruders. However, if non-deterministic encryption were used, the above query spitting could not be realized and the resulting work load placed on the smartcard would be too great.

Similar our work, Smith and Safford investigate using secure coprocessors for providing access to database records while maintaining privacy with respect to both DBAs and outsiders [14]. Their approach is relatively theoretical as they aim to hide which encrypted tuples are selected as part of query response (which requires algorithms that touch upon every tuple in a database for every query posed). This differs from our goal, since we are not necessarily interested in completely hiding which of the encrypted tuples are selected. The performance of their solution is measured (with the IBM 4758 coprocessor [3]) and a conclusion is reached that current technology is inadequate to realize a full-fledged database querying model. The main bottleneck encountered is the direct memory access speed between the coprocessor's 3DES engine and internal RAM. We acknowledge that this limitation applies to our extension to the DAS model; however, the near-future advances in SC technology will most likely overcome obviate this problem.

## 6 Conclusion

This paper introduced a framework for including a secure coprocessor in the outsourced database model (also known as: *Database-as-a-Service*). The SC acts as a trusted intermediary between the database service provider and its clients, who store their databases at these untrusted service providers. Weaknesses existing in the plain DAS model – such as: limited server side query support, bandwidth overhead and computationally overloaded clients – can be overcome with the inclusion of an SC at the server side.

We describe how the query model changes with the incorporation of the SC, while still utilizing the client for certain parts of database queries, such as sorting and other operations that require multiple scans through a large number of tuples. To increase performance of query processing, we suggest the use of a database storage model optimized to handle encrypted tuples.

Due to the current state of secure coprocessor technology, it is necessary for SC's to have access to external storage when storing large intermediary results. This leads to concerns about tuple access privacy, whereby a server can observe which (encrypted) tuples are included in query responses. As part of future work, we intend to: (1) pursue this topic in greater detail to identify the exact threats and possible defense mechanisms and (2) implement (and experiment with) our framework using a real SC, such as the IBM 4758.

## References

[1] A. Beimel, Y. Ishai, E. Kushilevitz, and T. Malkin. One-way Functions are Essential for Single-Server Private Information Retrieval. In *Symposium on Theory of Computing*, 1999.

[2] L. Bouganim and P. Pucheral. Chip-Secured Data Access: Confidential Data on Untrusted Servers. In *International Conference on Very Large Data Bases*, pages 131–142, 2002.

[3] J. G. Dyer, M. Lindemann, R. S. R. Perez, L. van Doorn, and S. W. Smith. Building the IBM 4758 Secure Coprocessor. In *EEE Computer*, pages 57–66, 2001.

[4] Y. Gertner, S. Goldwasser, and T. Malkin. A Random Server Model for PIR. In *RANDOM*, 1998.

[5] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin. Protecting Data Privacy in Private Information Retrieval Schemes. In *Symposium on Theory of Computing*, 1998.

[6] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over Encrypted Data in the Database-Service-Provider Model. In *ACM SIGMOD Conference on Management of Data*, pages 216–227. ACM Press, June 2002.

[7] H. Hacigümüş, B. Iyer, and S. Mehrotra. Providing Database as a Service. In *International Conference on Data Engineering*, March 2002.

[8] B. Hore, S. Mehrotra, and G. Tsudik. A Privacy-Preserving Index for Range Queries. In *International Conference on Very Large Databases*, 2004.

[9] B. Iyer, S. Mehrotra, E. Mykletun, G. Tsudik, and Y. Wu. A Framework for Efficient Storage Security in RDBMS. In *International Conference on Extending Database Technology*, pages 169–179, 2004.

[10] S. Jiang, K. Minami, and S. Smith. Securing Web Servers against Insider Attack. In *Annual Computer Security Applications Conference*, 2001.

[11] N. I. of Standards and Technology. Advanced encryption standard. *NIST FIPS PUB 197*, 2001.

[12] A. Perrig, S. Smith, D. Song, and J. Tygar. SAM: A Flexible and Secure Auction Architecture using Trusted Hardware. In *The Electronic Journal for E-Commerce Tools and Applications*, volume 1, 2002.

[13] R. Rivest, L. Adleman, and M. Dertouzous. On Data Banks and Privacy Homomorphisms. In *Foundations of Secure Computation*, Academic Press, pages 169–179, 1978.

[14] S. W. Smith and D. Safford. Practical server privacy with secure coprocessors. In *IBM Systems Journal*, pages 683–695, 2001.