Stony Brook University

cDB: STRONG REGULATORY COMPLIANT DATABASES

A THESIS PRESENTED

BY

ASHISH ANAND

TO

THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

STONY BROOK UNIVERSITY

MAY 2009

Stony Brook University
The Graduate School

Ashish Anand

We, the thesis committee for the above candidate for the
Master of Science degree,
hereby recommend acceptance of this thesis.

Professor Radu Sion, Thesis Advisor
Computer Science Department

Professor Scott Smolka, Chairman of the Thesis Committee
Computer Science Department

Professor Rob Johnson
Computer Science Department

This thesis is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Thesis

cDB: STRONG REGULATORY COMPLIANT DATABASES

By

Ashish Anand

Master of Science
In
Computer Science
Stony Brook University

2009

Setups for outsourcing databases work on the assumption that the hosting server is trusted. Existing mechanisms require decryption of the database at the server, which may not be trusted. In this thesis we talk about how trusted hardware can be used to solve the problem of untrusted service providers. We will achieve this by designing mechanisms that allow clients to fully access and benefit from an encrypted database residing on a database controlled by an untrusted admin. A query is divided into sensitive and non-sensitive components. Sensitive query operations such as evaluation of join predicates are performed inside a trusted enclosure of the IBM 4764 secure coprocessor. The untrusted server hardware is relied upon for non-sensitive I/O such as data fetches and database management. A simple wrapper is deployed at the client to enable standard database operations. Traditional queries are transformed, encrypted and forwarded to the trusted hardware by the untrusted server. To support the above design we also cross-compile and deploy additional code for the sCPU including a Java virtual machine as well as a modified network stack that is able to run over the PCI system bus to comply with the IBM 4764's security standards (FIPS 140-2 Level 4) and load classes over HTTP from the untrusted server. Standard cryptographic primitives are used to ensure the untrusted server does not 'cheat' while passing data between the client and the sCPU, and back. The elements deployed in this instance are the Kaffe JVM, the SQLite3 engine and the PostgreSQL client in the sCPU (minimal embedded Linux on PowerPC-405) and PostgreSQL on the server.

Contents

List of Figures

List of Tables

Chapter 1: Introduction

My effort in the Trusted Hardware / Network Security & Applied Cryptography lab has been to gain experience and specialize in trusted hardware in general and the IBM 4764 PCI-X Cryptographic Coprocessor in particular with the ultimate goal of enabling for the first time, custom applications to be written for this platform.

“The IBM 4764 PCI-X Cryptographic Coprocessor is a state-of-the-art secure subsystem that is supported for use in certain IBM server systems to perform DES and public-key cryptography in a highly secure environment. You can also load software for highly sensitive processing, such as the minting of electronic postage, which must perform its intended function even when under the physical control of a motivated adversary.” **[1]**

We have achieved our goal and identified key components of the IBM 4764 that allow us to run custom software. We cross-compiled multiple application environments including a Java virtual machine and an Sqlite3 engine for the IBM 4764. We next applied our skills in deploying the IBM 4764 in the outsourced database context in the presence of an untrusted host. In this scenario, clients wish to access and process remotely hosted data while not requiring assumptions of trust in their provider.

“Minting of electronic money and electronic postage are examples of critical functions that must run in a highly trustworthy environment. Using toolkits available from IBM under custom contract, you can implement your own applications for the coprocessor, or extend IBM's CCA application. You can make a fast start on your custom application development when you extend CCA using its flexible access-control system and many existing services.

IBM will issue you a unique identifier and certify your code-signing key so that you can sign your own custom coprocessor software. You develop your software using conventional IBM or Microsoft C-language compilers and use the toolkit-provided debugging programs. You or your customers can then load coprocessor software in a normal server environment. Using the PKI-based outbound authentication capabilities of the coprocessor's control program, you can securely administer the coprocessor environment, even from remote locations. Auditors can inspect the coprocessor's digitally signed status response to confirm that the coprocessor remains untampered and running uniquely identified software.” **[2]**

1.1 Commercial Outsourced Computation/Storage Providers

A plethora of commercial outsourced computation/service and storage providers exist. The common adversarial model assumes clients trust the providers. We introduce just a few.

Traditional out-sourcing: IBM Blue Cloud (www.ibm.com/ibm/cloud)

IBM re-branded its traditional outsourcing paradigm Blue Cloud, added virtualization functionality based on Linux, Hadoop, Xen, or PowerVM and provides administration functionality through several third party vendors, including 3Tera.com and oppsource.net.

Clouds: Amazon AWS (www.aws.amazon.com)

Amazon offers a set of storage and computation outsourcing facilities structured and deployed as web services. These include a number of additional mechanisms and information portals such as queuing, website statistics, e-commerce, payments, billing, structure database, authentication, shipment, as well as a content delivery network infrastructure. The main computation outsourcing unit is a virtual machine image that is networked and loaded transparently. Third parties such as rightScale.com provide access and effective control.

Clouds: Google Apps (www.google.com/apps)

The Google App engine allows the transparent deployment of python applications in its infrastructure. It features dynamic web serving, persistent storage, load balancing, authentication and mail apps. Apps run in sandboxed platform-independent environments.

Clouds: Windows Azure (www.microsoft.com/azure)

Recently, Microsoft has also introduced its version of cloud, dubbed Windows Azure aimed at hosting .Net apps in managed as well as un-managed modes. It deploys Windows Server as a base operating system and Hyper-V as a virtualization layer.

Virtually all of the above operate in the trusted model. Clients are offered written contractual guarantees for the cloud's compliance with certain integrity policies (and very rarely confidentiality assurances) without any technological safeguards and illicit behavior detection mechanisms (except, as noted above, traditional security against outsiders).

The reasons are multiple. In corporate markets, traditional business models result in companies being often stuck with sizable price tags for the luxury of dedicated outsourcing and management, provided by large vendors such as IBM. These settings do not benefit from any of the advantages of the cloud-level economies of scale and consolidation that make them so promising. Service guarantees rely on mutual non-disclosure and security policy agreements. Clients often receive dedicated on-site 24-hour service.

In consumer markets on the other hand, where competitive pricing is available, current revenue and business models are heavily advertisement-driven and require direct access to client data (emails, documents), access patterns (visited sites, searches, buying patterns), health (see Google Health), and social networks. The data mining market's direct and indirect worth (through advertisements) are evalued in the billions and increasing, with no end in sight. Not too long ago, Google spent over $3 billion to acquire DoubleClick, a data mining and web-click tracking company among others.

A second related reason for the use of the trusted model in consumer markets is direct cost, especially in the case of free services. This is illustrated best by the cost-saving behavior of (email) service providers such as Google and Yahoo, that have been (and still are) operating for years with no in-transit confidentiality of traffic, mainly to avoid the significantly increased loads that would ensue due to enabling SSL connections. This led to the development of simple session hi-jacking tools. Ironically, the un-secured traffic is preceded by a SSL-secured password authentication step.

Secure Outsourcing

Yet, hundreds of millions of users embrace free insecure web apps. This shows that today's (mostly personal) cloud clients are willing to trade their privacy for (free) service. This is not necessarily a bad thing, especially at this critical-mass building stage, yet raises questions of clouds' viability for commercial, regulatory-compliant deployment, involving sensitive data and logic. And, from a bottom-line cost-perspective, is it worth even trying? This is what we aim to understand here.

Additional assurances will come at extra cost, mainly due to the network traffic, storage requirements and CPU cycles incurred by the security logic. Thus, a first step in understanding the financial viability of secure outsourcing is to evaluate whether the overall picture allows for this additional cost, while still resulting in end-to-end cost benefits.

1.2 Technical Overview of the IBM 4764 Secure Coprocessor

A secure coprocessor is a general-purpose computing environment that withstands physical attacks and logical attacks. The device must run the programs that it is supposed to, unmolested. You must be able to (remotely) distinguish between the real device and application, and a clever impersonator. The coprocessor must remain secure even if adversaries carry out destructive analysis of one or more devices.

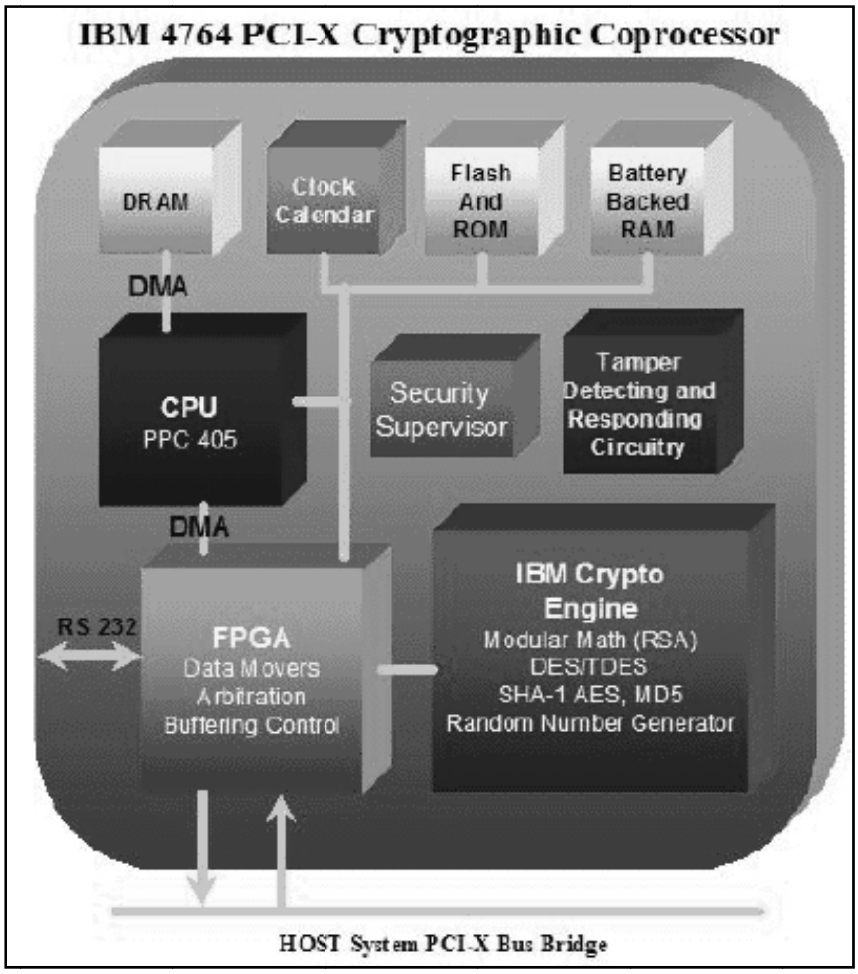

Figure 1: IBM 4764 sCPU Architecture

Its features include Data Encryption Standard (DES) and Triple-DES encryption, widely used hashing algorithm, SHA-1, in hardware, common RSA public key infrastructure algorithms, advanced 2048-bit modular-exponentiation operations, true hardware random number generation designed to improve performance, Pseudo Random Number Generation (PRNG), IBM offers a Common Cryptographic Architecture (CCA)

support program for the PCI-X cryptographic coprocessor. Below are some system throughput statistics with increasing record sizes.

Function	Context	IBM 4764	P4 @ 3.4Ghz
RSA sig.	512 bits	4200/s (est.)	1315/s
	1024 bits	848/s	261/s
	2048 bits	316-470/s	43/s
SHA-1	1KB blk.	1.42 MB/s	80 MB/s
	64 KB blk.	18.6 MB/s	120+ MB/s
DMA xfer	end-to-end	75-90 MB/s	1+ GB/s

Table 1: IBM 4764 vs. iP4@3.4Ghz/OpenSSL 0.9.7f

The IBM 4764 hardware is basically a 266MHz Motorola PowerPC-405 CPU with a 66MHz PCI bus clock. Our host server used in the project has 4 3.3GHz Intel processors.

Chapter 2: Related Work

Sun S. Chung & Gultekin Ozsoyoglu **[4]** talk about anti-tamper databases where they perform queries directly on encrypted databases using homomorphic encryption. But this type of encryption cannot be applied to complex or nested queries. Einar Mykletun, Maithili Narasimha & Gene Tsudik **[5]** talk about authentication and integrity for outsourced databases. They use cryptographic primitives to ensure query results have not been tampered with and are authentic. But this is based on the assumption that the database server is trusted and so, the database itself could be altered, if not the replies to queries.

Ernesto Damiani, et. Al **[6]** provide a hash based method for database encryption and use indexing on encrypted databases. But they return a super-set of the query which the client has to query again. Hakan H, et. Al **[7]** talk about executing SQL queries over encrypted data in the database-service-provider model. These two works are close to our work, and were published six to seven years ago while our implementation is using the new IBM 4764 trusted hardware.

George Davida, David Wells & John Kam **[8]** use the Chinese Remainder Theorem to generate sub-keys for encryption/decryption of fields within a record. We also use different keys for each row, and each table. In a 2005 paper by G. Aggarwal et. Al. **[9]**, a distributed architecture to partition the database is suggested with a reference to alternately using tamper-proof hardware the way we are. In a 2004 paper, Clifton & Murat K **[10]** suggest (in theory) an implementation like ours. S. W. Smith & D. Safford **[11]** talk about theoretical performance bottlenecks in practical server privacy with secure coprocessors. E. Mykletun & G. Tsudik **[12]** suggest a similar theoretical framework.

Hakan H, Bala Iyer, Chen Li & Sharad Mehrotra **[13]** talk about having a coarse index at the server side, next to the encrypted table, and use it to perform the initial join. The SELECT and decryption portion is then taken care of at the server. However, in their model, the client assumes integrity of the data stored at the database. In our implementation, we aim to ensure that the server does not manipulate the database itself. Also, in their model, they always send entire (encrypted) tables at the client which could translate into high network costs. We always send entire (encrypted) tables only to the 4764 through the system bus of the server and the 4764 has the client's encryption key.

Chapter 3: Verifying Integrity

Using outbound authentication functions, the 4764 can prove to the client that the applications running on it have not been tampered with. The 4764 non volatile memory is partitioned into four segments, each of which can contain program code and sensitive data. Segment 0 contains a ROM-based BIOS equivalent known as Miniboot. Segment 1 is the FLASH-based portion of the Miniboot. Segment 2 contains the operating system. Segment 3 contains the application (FLASH-based).

During manufacture, the 4764 generates a random RSA keypair (Device Keypair), puts the public key in a certificate, signs it with IBM's private key (IBM Class Root Keypair). The coprocessor saves this certificate and another certificate containing the IBM Class Root public key (which is also signed by IBM's private key). Using these certificates, a chain of trust is built in regards to updating higher memory segments (for example, to load new applications on the 3rd segment). Thus, an adversary cannot do much unless he gets access to either the IBM Cass Root private key (assumed impossible since only IBM possesses this) or the private key part of the Device Keypair (assumed impossible because of the tamper-proof model of the 4764).

3.1 Verifying Integrity of the IBM 4764 Coprocessor

The 4764 has its own public-private RSA device key pair. The private key is retained within the 4764 and IBM ships certificates for each coprocessor while the private key is hard-coded in the Coprocessor Load Utility (CLU). The public exponent and modulus of this pair is published on the IBM website **[3]**. It is possible to determine that a coprocessor is a legitimate untampered IBM 4764 using a validation command via the CLU.

3.2 Verifying Integrity of the Card Application

For the application code running inside the 4764, we use another public-private key pair. When the client wishes to talk to the 4764 code (aka send an SQL query etc.), it is encrypted with the public key of the card application. When the card sends replies back out, it can certify and sign them with its private key that can then be verified by the client using the conventional certification mechanisms. The 4764's public key is published.

Chapter 4: Design

We divide our model into three main components, the client, the main CPU (where the database, web server and Java classes reside) and the secure CPU (the IBM 4764 is plugged into the motherboard of the main CPU via the PCI-X slot).

4.1 Client Infrastructure

4.1.1 Database Driver Integration

The client runs a Java application which contains standard code to connect to a database server using JDBC or other similar database drivers. Instead of connecting to the actual database server, we require the application to connect to localhost with the same authentication credentials that would be used to connect to the actual database. The localhost acts as a proxy/protocol wrapper for the remote database. In theory, this step is equivalent to simply submitting the query and presenting the result. This is achieved by using a standard SQL client binary and having it connect to the actual database server instead.

4.1.2 The Server-like Proxy

When the JDBC driver tries to connect locally to the database, the server-like proxy pretends to be the actual server, takes the authentication credentials, verifies them from the actual database server and upon success, further takes the SQL query from the application via the JDBC driver. When the proxy receives any results after query processing, it also returns them to the JDBC driver following the standard reply protocol. Alternately, if the client is using the SQL client binary, we require the use of our modified SQL client that catches the input query and passes it to the cryptographic module, and listens for results from the cryptographic module instead of listening for the actual server to reply back.

4.1.3 Cryptographic Module

Upon receiving a query from the proxy, it is encrypted with the IBM 4764 secure coprocessor's public key. Now the encrypted query is ready to be sent over to the server. Upon receiving a signed result from the server, it is verified with the coprocessor's public key and decrypted with the client's private key and passed back to the proxy.

4.1.4 Optimization of Data Transfer

When the resulting data set of the TagID-JOIN attribute pairs gets large, it might be more efficient to fetch the SELECT attributes from the main CPU by the client directly rather than going through the secure CPU. We deploy another Sqlite3 engine as part of the client wrapper and use it for the same purpose as is the one inside the secure CPU used for.

4.2 Server Infrastructure

4.2.1 Forwarder

The forwarder listens for encrypted queries sent by the client and transfers them to the secure coprocessor. The forwarder listens for encrypted results sent by the secure coprocessor and transfers them to the client. Thus, we ensure that this untrusted server is not able to learn anything.

4.2.2 Database Repository

The main database is stored on the server and the tuples and attribute names are encrypted by the client's private key. In each table of the database, we introduce an additional attribute that serves simply as an un-encrypted row number. This 'TagID' is encoded: each row of a table will have a different key derived from the root private key, and this is used in generating a unique TagID for each row.

4.2.3 Java Class Repository

The server also hosts Java class files for any Java application that is run inside the IBM 4764 secure coprocessor. This is more so intended for future use when we would be using pre-written Java apps for optimizations, somewhat parallel to the concept of stored procedures in SQL.

4.2.4 Ethernet Simulation Logic

Communication between the server/main CPU (host) and the IBM 4764 secure coprocessor (card) take place over TCP/Ethernet. A LAN cable plugs into the IBM 4764. This mechanism is not is permitted for a production environment. Therefore in order to comply with the IBM 4764's security standards (FIPS 140-2 Level 4) it is required that all card-host communication takes place over the system bus. We modify standard network calls to route data over the bus instead of the network [Credit to Peter Williams].

4.3 Secure Coprocessor Infrastructure

4.3.1 Cryptographic Module (incoming/outgoing)

i. This module decrypts queries received from the client via the server. Since the queries were encrypted with the IBM 4764's public key, this decryption is trivial with the private-public key pair. The module also encrypts outgoing data (query results) to the client via the server using the client's public key.

ii. This module decrypts the tuples corresponding to the JOIN attributes in the received query based on knowledge of the client's private key. This is needed to be able to identify the qualifying TagID values for a pair of tables on which the JOIN operation is being performed.

4.3.2 Parser

The parser determines the JOIN attributes that are to be involved. Towards this, it examines the WHERE clause of the query (after decryption). This information is then relayed to the pSQL client. Later it examines the SELECT attributes to create final query results.

4.3.3 Performance Evaluation Logic

It would be inefficient to transfer n^2 rows when say a large number of attributes are requested as part of the SELECT clause and say the JOIN of two n-sized tables results in an n^2 sized result. In such cases, we send the resultant TagID's to the client and latter part of the protocol takes place at the client end. For smaller results, we process within the sCPU as per the protocol. We are also considering an alternate mechanism where we simply save the resultant 'interesting TagIDs' as a temporary (or even permanent, so that subsequent SQL sessions can access it) table to eliminate the need to send the 'interesting TagIDs' to the client or back to the 4764.

4.3.4 Secure JOIN Logic

The requested JOIN attributes with their entire set of values is fetched from the database server, along with the TagID's. This is done for every table (and every JOIN attribute) that is in the WHERE clause of the initial query from the client. Note the names of the JOIN attributes are encrypted, and so are the tuple values, each row being encrypted with a different key. Therefore, the sCPU must know the logic used in generating the keys for each row from the root key.

4.3.5 Sqlite3 Engine

This is a light implementation of an SQL engine written entirely in C **[14]**. We deploy this in the sCPU to import the fetched tables from the main CPU into a small database inside the secure CPU and to perform the JOIN operation and get the resultant TagID's for the tables involved in the initial query. The result can then either be used to fetch the SELECT attributes specified in the query or just be forwarded at the client end via the main CPU so the client may fetch the SELECT attributes based on the resultant TagID's itself. This decision is made by the performance evaluation module.

4.3.6 pSQL Client

We deploy a standard pSQL client binary inside the secure CPU that will be used to connect to the main database and fetch the SELECT attributes from it after knowing the TagID's needed to be looked at for the tables that are part of the JOIN. Note, these fetched results are encrypted with the client's private key.

4.3.7 Secure Java Application

We modified a standard open source Java Virtual Machine to be able to run Java applications from inside the secure CPU. Due to memory constraints inside the IBM 4764, we fetch the class files from the main CPU over HTTP.

4.3.8 Ethernet Simulation Logic

Peter Williams in the NSAC Lab also developed a custom TCP stack that allows communication with the IBM 4764 over the PCI-X bus transparently. Despite certain development crypto cards from IBM also featuring Ethernet ports, such methods of connection are not permitted in production environments. Therefore in order to comply with the IBM 4764's security standards (FIPS 140-2 Level 4) it is required that all card-host communication takes place over the system bus. We modify standard network calls to route data over the bus instead of the network.

Chapter 5: Implementation

5.1 Information Flow

First we initialize a receptor script in the 4764, a forwarder script at the server and a proxy script at the client. The Java application at the client end tries to connect locally to the (remote in reality) database. The proxy script receives the connection string and uses these credentials to authenticate to the remote database. Alternately, the client connects to the remote database using a modified pSQL client binary which after authentication, fetches the SQL query and sends it to the proxy instead of letting it go to the remote database server.

Once the proxy receives a query, it encrypts it using the 4764's public key and sends it to the forwarder on the server. The forwarder is designed to do nothing but send data from the client to the 4764 and vice versa. Thus even if a malicious server administrator deploys his own code at this point, he would be unable to decipher any of the incoming queries and outgoing results. The receptor script inside the 4764 receives this encrypted query and decrypts it with the 4764's private key. This private key is hardcoded in the application. In a production environment, a malicious system administrator would not have access to the (third) memory segment of the 4764 where our application bundle resides.

The decrypted query is sent to a parser script [Credit to Rajarshi Agnihotri] that fetches the WHERE clause of the query. For example for the query, *"SELECT table1.name, table2.dept FROM table1,table2 WHERE table1.ssn=table2.ssn;"* we know that we need to work on *table1* and *table2* inside the 4764. The parser then asks the pSQL client inside the 4764 to fetch these tables into the 4764. The entire table is not fetched: only the JOIN attributes along with the 'TagID' are fetched.

When the encrypted database is submitted to the server, the client adds an additional attribute 'TagID' in each table. This acts as an un-encrypted primary key for each table. The tuples in each row are encrypted based on a different key for each row, which in turn is derived from the client's root private key. The database server thus returns the requested tables containing the JOIN attributes and 'TagID' to the 4764. Since the JOIN attribute tuples are encrypted, a malicious administrator does not gain any knowledge by this transfer.

Upon receiving these tables inside the 4764, a cryptographic module takes the client's primary key to decrypt the tuples. Since each row was encrypted using a different key, two similar (plain text) values would have different resultant cipher texts when they are present on different rows in the two tables on which we are performing the JOIN operation. Thus the decryption is essential. The client's primary key is cached in the 4764 flash file system at the start of the protocol. To ensure the malicious administrator does not find out the client's primary key, the client sends this over to the 4764 via the 'forwarder' after encrypting it with the 4764's public key.

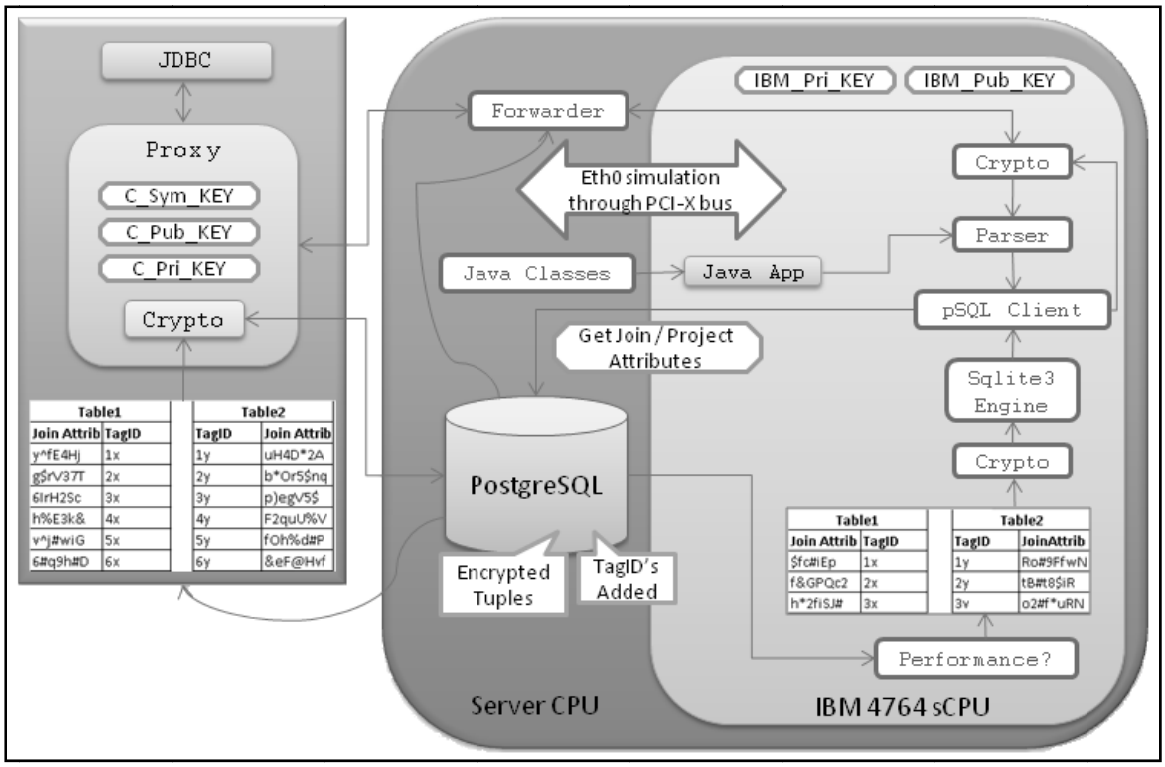

Figure 2: Model Design

The decrypted tables containing the JOIN attribute and 'TagID' are now sent to an Sqlite3 Engine in the 4764. This is a standard light weight implementation of an SQL server with basic capabilities. The decrypted tables are imported into an Sqlite3 compatible database, and the actual JOIN occurs here. For our simplistic query, *"SELECT table1.name, table2.dept FROM table1,table2 WHERE table1.ssn=table2.ssn;"*, the Sqlite3 now performs a *"SELECT table1.TagID,table2.TagID from table1,table2 WHERE table1.ssn=table2.ssn;"*. Thus we attain the 'TagID' attributes for both the tables that are the 'interesting' tuples resulting after the actual JOIN.

These are saved as temporary tables at the main database. Now the pSQL client (either from the client end or from the 4764) needs to know the SELECT attributes of the initial query that the 'parser' received. The pSQL client now basically just does a PROJECT like operation. It tells the database, *"SELECT table1.name,table2.dept FROM table1,table2,temp_table WHERE table1.TagID=temp_table.TagID_1 AND table2.TagID=temp_table.TagID_2;"* where temp_table.TagID_1 and temp_table.TagID_2 are the 'interesting TagIDs' returned by the Sqlite3 JOIN operation previously.

The results are sent to the cryptographic module so they can be encrypted with the client's public key, and are finally signed by the 4764's private key and sent to the forwarder. The forwarder simply sends these to the proxy at the client. Note once again, that a malicious administrator is not able to gain knowledge about these results. Once the result is received at the proxy it is either sent to the modified pSQL client or returned to the JDBC driver that initiated the connection.

5.2 Performance Module for Optimizing Data Transfer

For large datasets, it is possible to send sufficient information about the result back to the client that enables it to fetch the remaining attribute data from the server itself. Thus we deploy a performance decision module in the 4764 that makes the decision between either sending back the final result through the 4764 or sending only the 'interesting TagIDs' to the client.

In the case when the partial result is going to be processed at the client, once the Sqlite3 Engine determines the 'interesting TagIDs' for the two tables of the JOIN attributes, the pSQL client creates the temporary tables in the main database as before. Now the pSQL client at the client end requests a query such as *"SELECT table1.name,table2.dept FROM table1,table2,temp_table WHERE table1.TagID=temp_table.TagID_1 AND table2.TagID=temp_table.TagID_2;"* where temp_table.TagID_1 and temp_table.TagID_2 are the 'interesting TagIDs' returned by the Sqlite3 JOIN operation. When it arrives at the proxy, it is decrypted with the client's private key and verified etc. After decryption by the cryptographic module at the client, the result is returned to the JDBC driver.

Depending on the common workload, the branching decision threshold can be adjusted.

5.3 Ethernet Simulation

As mentioned before, to comply with the FIPS security standard, IBM does not permit the secure coprocessor to be hooked to Ethernet ports in production environments. In the NSAC Lab, Peter Williams wrote a full TCP stack replacement for the network socket calls and deployed logic at the server and in the secure coprocessor to simulate network data transfer over the system bus. Williams wrote a *socketoverride* library and we require all applications inside the secure coprocessor to link to it and modify their network calls to use our functions instead (the convention used is to append *call_* to the standard socket calls. This is just a simple rename operation (example, *connect()* would now be called *call_connect()* and so on). This is typically done for the JVM, the 'receptor' code, the pSQL client residing on the secure coprocessor and any secure Java application that runs inside the sCPU that needs network access to communicate to the main server.

5.4 JVM Cross-Port

To enable client driven Java logic to be run inside the IBM 4764, we also modified and cross-compiled the JVM source to load the class files over HTTP: the database server on the main CPU is also configured as a web server to host the class files for any Java programs that are needed to run from the secure coprocessor. While it appears to the JVM that the class files need be loaded over Ethernet, we also hack the network calls and have data transfer take place over the system bus. While this was needed to stick to security standards in production environments, it also enhances throughput.

We first tried to cross-compile a JVM called JamVM on the PowerPC-405 that the IBM 4764 runs on. Unfortunately we realized (after around 3 weeks) that the PowerPC-405 does not support floating point operations, and the native code in JamVM required this. There are hacks to emulate this in software through GCC flags etc., but we did not succeed. We then explored another JVM called Kaffe and managed to cross-compile it with the GCC floating point emulation.

5.5 Sub-key & TagID Encoding

In our prototype implementation, the TagIDs are uniquely generated integers per data tuple. It is generated as a function of the primary key attribute of the corresponding tuple, the table name and a client secret. The client's private key next must be used to generate different sub-keys for all the tables in the database. This is important because if two tables have the same cell values for the same attribute, at the same row numbers,

they must lead to different cipher texts after encryption. Similarly within the same table, each row needs to have a unique row key. Thus, we break down the root private key into unique table keys, and further breakdown a table key into unique row keys. The cryptographic modules at the client proxy and the secure coprocessor's 'receptor' are made aware of the (same) sub-key generation logic.

5.6 (Existing) Libraries Used

We use PostgreSQL (libpq 8.3) and Sqlite3 (libsqlite 3.6.13) API's for C directly in our code to connect to the database server/engine and for sending/receiving queries/results. This proves to be more efficient than using the standard client interfaces that ship with these packages. For cryptographic operations inside the secure coprocessor, we use the card side API (xcrypto) since that allows us to perform conventional crypto operations using the hardware accelerators build on the IBM 4764. For the crypto operations at the client end we use the industry standard (openssl 0.9.8).

Chapter 6: IBM 4764 Setup/Installation/Quick Overview

As part of this work, we also compiled a quick setup procedure for writing custom applications for the crypto card. In the following, we will highlight some of the problems we faced that are not directly mentioned on the IBM 4764 manuals available online.

6.1 Outline

0. Order hardware with PCI-X compatible motherboard
0. Download Novell's SUSE Linux (Enterprise Server 9)
1. CCA Support Program (for coding using verbs, not used), API Developer's Toolkit and Device Drivers (for custom programming)
2. Install cross linker, cross compiler
3. Writing applications
    a. Host side and card side code
    b. Compiling, making JFFS2 image etc.
4. Configure card for custom program loading
    a. Remove CCA Support program (not documented anywhere!)
5. Loading and running custom applications
6. Bandwidth testing

In a nutshell, the following steps are required to build and load applications:

1. Write the host and card-side toolkit applications that you want in C, using the Developer's Toolkit headers as necessary
2. Compile the host-side code using one of the supported native compilers.
3. Link the host-side code using one of the supported native linkers.
4. Compile the card-side code using a cross compiler.
5. Link the card-side program using the linker shipped with the cross compiler.
6. Use mkfs.jffs2, along with the Makefile and other files provided in *xctk/<version>/build* to create a */user0* JFFS2 image.
7. Load the JFFS2 filesystem image into the coprocessor using DRUID.

6.2 CCA API Developer's Toolkit, Device Drivers

- Release 3.25a of the software comes with a straightforward binary executable setup4764_3.25.0_date.bin which installs without any complications.
- Run /opt/IBM/4764/clu/./csulcu next. This is the coprocessor load utility, the basic interface used for most actions such as loading code onto the processor.
- Load CCA Support Program onto Segment 3 using the clu.

- Run the ST command to obtain segment ownership details. They would be:
  Segment 2: Runnable, Owner2: 2
  Segment3: Runnable, Owner3: 2
- Validate the coprocessor segment contents using the provided key file:
  VA 12r8565v.clu

The CCA Support Program provides an API in the form of 'verbs' for all crypto operations that can be included in typical C style programming. Thus, this limits you to run only the support program on the card on Segment 3, while allows easier coding on the host side program. If you wish to run arbitrary code on the card side as well, you must remove the support program from Segment 3 and load your own code instead. This code is linked with a number of available libraries (listed later).

6.3 Install cross linker, cross compiler

This must be done manually. There is a utility called 'crosstools' [http://www.kegel.com/crosstool] for building 'gcc' and 'ld' across other architectures. We need one to support the PowerPC 405. Please follow the procedure mentioned on the crosstools documentation, it is straightforward except for a couple of dependencies that must be installed manually. You should see the following message upon successful completion of the build process that takes a couple of hours:

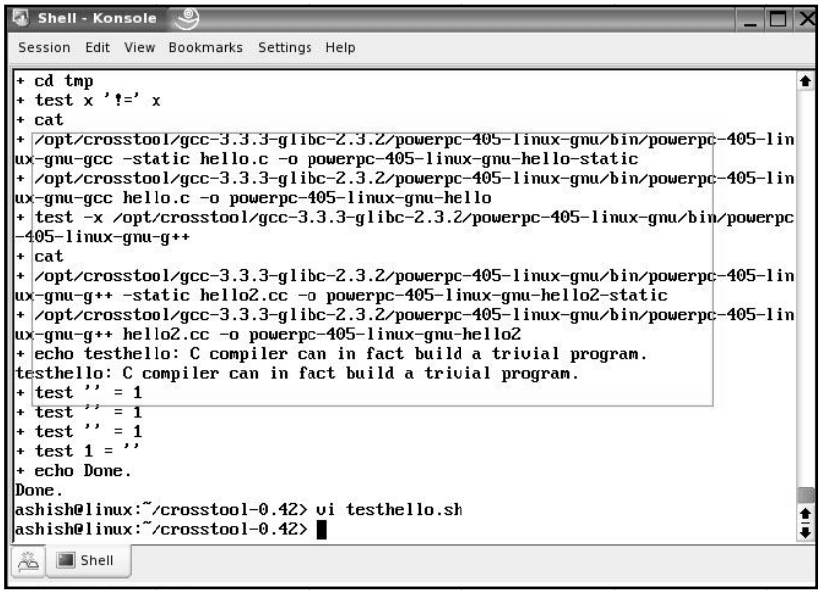

Figure 3: Successful Crosstools Build

Note: You must specify the paths used here in the makefile of your custom application.

6.4 Writing Applications

A card side program runs in Segment 3 over Linux running inside the coprocessor in Segment 2. It must be compiled using the cross compiler for the PowerPC 405. A host side program runs on the machine like conventional C programs.

Library files which may be linked with a card-side application:

- xctk/<version>/lib/card/gcc/libxccomapi_stub.so: If the application will communicate with the host
- xctk/<version>/lib/card/gcc/libxcmgrapi_stub.so: If the application will use configuration functions
- xctk/<version>/lib/card/gcc/libxcoa_stub.so: If the application will use Outbound Authentication functions
- xctk/<version>/lib/card/gcc/libxcrandom_stub.so: If the application will use the random number generator, or will generate DES or RSA keys
- xctk/<version>/lib/card/gcc/libxcrsalnx_stub.so: If the application will use Large Integer Modular Math, or RSA or DSA functions
- xctk/<version>/lib/card/gcc/libxcskch_stub.so: If the application will use DSE or Hashing functions

The development process requires the creation of a JFFS2 filesystem image that can be loaded into the coprocessor using DRUID. Use the makefile provided in */xctk/<version>/build ,* edit it to copy the desired application to the build directory, and make changes in init.sh required for the application (such as file names, paths etc.).

The issue faced here was that mkfs.jffs2 isn't part of default 2.6 kernel (support was removed). Now if you try to build mtd-utils which includes this application, it says you need a newer version of GLIBC_2.4 but default SUSE Linux 9 ES comes with GLIBC_2.3 so you need to rebuild glibc which is a long process. So instead I found a ready to run binary at:

http://www.mirrorservice.org/sites/sources.redhat.com/pub/jffs2/mkfs.jffs2

6.5 Configure coprocessor for custom program loading

These files are provided by IBM and must be used in the right sequence to make things work:

- **CR1rrrss.CLU**, which loads release rrr revision ss of IBM's system software into a coprocessor. CR1rrrss.CLU can only be loaded into an xCrypto card in the factory-fresh state
- **CE1rrrss.CLU**, which updates the system software in a coprocessor
- **TDVrrrss.CLU**, which prepares a coprocessor for use as a development platform. TDVrrrss.CLU can only be loaded into an xCrypto card that contains release rrr revision ss of IBM's system software
- **TE3rrrss.CLU**, which enables a coprocessor to accept coprocessor applications downloaded by the DRUID utility. TE3rrrss.CLU can only be loaded into an xCrypto card that has been prepared for use as a development platform using TDVrrrss.CLU
- **TL3rrrss.CLU**, which clears any state an application under development has saved in nonvolatile memory (so that the application will start next time with a clean slate). TL3rrrss.CLU also loads the "reverse-then-echo" application into the coprocessor. TL3rrrss.CLU can only be loaded into an IBM 4764 PCI-X that has been prepared for use as a development platform using DVrrrss.CLU and which has been prepared to accept downloaded applications using TE3rrrss.CLU
- **TR3rrrss.CLU**, which reloads the "reverse-then-echo" application into the coprocessor. TR3rrrss.CLU can only be loaded into an xCrypto card that has been prepared for use as a development platform using TDVrrrss.CLU and which has been prepared to accept downloaded applications using TE3rrrss.CLU and TL3rrrss.CLU
- **TRSrrrss.CLU**, which prepares an xCrypto card that has been used for development to be used in a production setting. TRSrrrss.CLU essentially restores the coprocessor to the state it is in immediately after CR1rrrss.CLU or CE1rrrss.CLU has been loaded. TRSrrrss.CLU can only be loaded into an xCrypto card that has been prepared for use as a development platform using TDVrrrss.CLU
- **ESTOWN2.E2T, EMBURN2.L2T, REMBURN2.R2T, and SUROWN2.S2T**, which are used to generate a version of an application suitable for release
- **tdvRRRLL.l2t** is the Toolkit EMBURN2 unpackaged CLU file. This file should only be used when segment2 has ownerID = 3 but is not reliable. In rare instances, such as a power or system failure during a load of tdvRRRLL.clu, it is possible for the card to have segment2 as owned, but unreliable. This file performs an EMBURN2 command on segment 2 with segment 2 ownerID = 3 which restores the card to a usable state after a failed toolkit CLU load for segment 2

Check the ownership identifiers now. The goal is to have '3' for Segment 2 and '6' for Segment 3 in order to run a custom application. By default, these were both '2. Here's what we tried initially:

1. Loaded ce132500.clu - updates the system software in segment 1.

2. tdv32500.clu - loads production version of coprocessor operating system into segment 2. (Error 844000dc: ownership of this segment 2 was already established)

3. te332500.clu - sets owner identifier for segment 3 which makes it possible to load s/w in segment 3. (Error 844000dd: for 3 also, already established)

4. tl332500.clu - sets public key associated with segment 3 and loads reverse/echo application. (Error 844000b1 - no message)

5. trs32500.clu - surrenders ownership of segment 2 (Error 84400eba - no message)

6. cex32500.clu - reloads segments 2 and 3. (Reload segment 2 successful but didn't change ownership identifiers)

It turned out that the .clu file to be used to first remove the CCA Support Program from Segment 3 was not listed in the manual. Here's the sequence of commands to issue to fix this:

1. crs32500.clu (Remove CCA)
2. tdv32500.clu (Load production version in segment 2)
3. te332500.clu (Set ownership id for segment 3)
4. tl332500.clu (Clear state and load sample application)

After this, any new application can be loaded using the PL command in the CLU once an image file has been created for the same.

6.6 Loading and running custom applications

Once a JFFS2 filesystem containing the image of the application has been generated, the filesystem may be downloaded to the coprocessor using DRUID. It prompts you for the binary file and the coprocessor adapter number (default 0 if only 1 present).

The issue we faced here was the following error:

ERROR: DRUID not permitted: seg2 not owned by development

If you encounter this, make sure to follow the exact sequence of .clu files mentioned in the configuration section.Return to the coprocessor load utility after this to reboot the card, after which the loaded application automatically runs on Segment 3.

6.7 Bandwidth Testing

Used the echo/reverse application that sends packets of specified maximum buffer size to the card and the card sends them back to the host for measuring the bandwidth. The observed bandwidth of this link averaged to ~43 MB/sec varying between 38 – 47 MB/sec.

Chapter 7: Performance Evaluation

We clock timings for worst case joins, implying that two tables having n rows each would result in a join result having n rows as well. The total time taken is clocked at the client end. Network and bus costs include time to send the query to the server, send the result to the client, miscellaneous trigger messages between the client, server and the sCPU. Other costs include time taken to parse the query and for other internal processing insude the sCPU. The other major cost incurred is the time taken to fetch the TagID and join attributes from the server into the sCPU. Then there is a cost for performing the initial join and the final join.

Size (kb)	Rows	Total Time (s)	Misc. Time (s)	Fetch Time (s)	Join time (s)
0.14	10	0.464216	0.413012975	0.046043	0.005126
1.4	100	0.581063	0.427546155	0.127042	0.026445
7	500	1.192257	0.4300471	0.619162	0.142979
14	1000	1.889178	0.314359219	1.289532	0.285114
17.5	1250	2.546543	0.495302614	1.660557	0.390459
21	1500	2.966557	0.519147475	1.969331	0.477823
28	2000	3.774773	0.568858291	2.587984	0.617778
70	5000	9.319711	0.938494898	6.728803	1.651994
112	8000	15.066052	1.459599118	10.659347	2.947101
140	10000	19.365546	1.872648298	13.844471	3.647962

Table 2: Protocol Timing Details

In this prototype we exclude the time for encryption and decryption of the query and results. We also exclude bus costs: the sCPU and main server are talking over the network.

We expect performance to get better after integrating the Ethernet simulation logic since it will be faster to communicate over the PCI-X bus than over the network.

The query used was “SELECT table1.name,table2.climit FROM table1,table2 WHERE table1.ssn=table2.ssn;”. Table1 and Table2 were generated to have the same number of rows and the JOIN result would result in the same number of rows as well since each key attribute in one table would match one in the other.

As evident, the bottleneck is in fetching the TagID and join attributes from the server to the sCPU. Due to this, the prototype scales for join results under 5000 rows. It takes 19 seconds end-to-end with two 10,000 row tables.

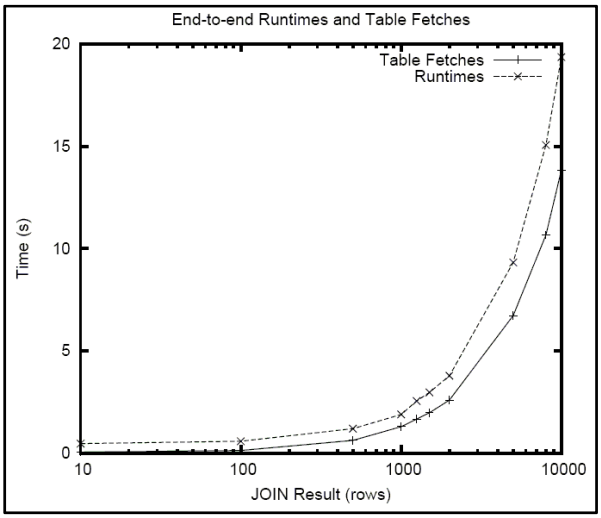

Figure 4: End-to-end Runtimes and Table Fetches

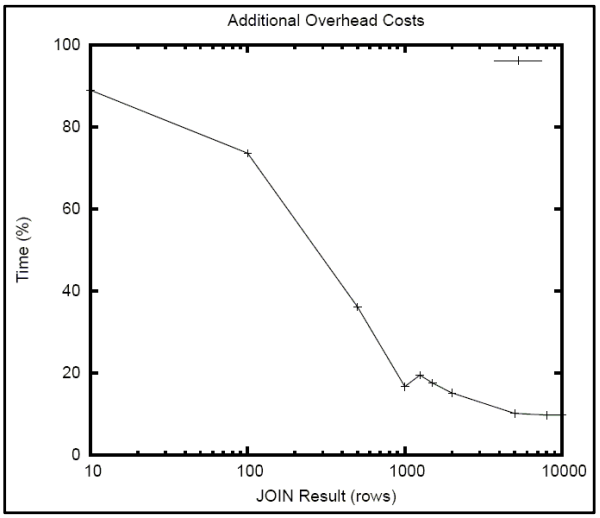

Figure 5: Additional Overhead Costs

When the tables have under 500 rows we observe that over 35% of the total time taken to produce results goes in tasks other than fetching information in the sCPU and performing the initial join. This includes network costs and the cost of performing the final query by the sCPU. Therefore the prototype does not scale for tables whose join results are under 500 rows.

Observe some numbers on the times taken to perform the initial joins in the sCPU using Sqlite3 (see Table2 – Join Times). As before, both tables consisted of n rows having unique primary key attributes and thus their join resulted in n rows too. The time to perform the same joins outside the IBM 4764 on a regular server will, of course, be lesser because of much more processing power on conventional servers.

We also present a comparison between server and sCPU performance with respect to performing joins in SQLITE3 independent of our framework. We clocked the times taken to process two 10,000 and two 100,000 row tables and compute a join. We observe that it is 35-45 times slower to do the same join inside the sCPU. This is attributed primarily to differences in CPU speed: 266MHz on the sCPU vs. 3.3GHz on the server. Then there are differences due to architectures. Whilst the sCPU runs on RISC based PowerPC-405, the server runs on Intel. The memory buses, the cache sizes, RAM etc. are also lesser on the sCPU.

We argue that it is still economical to deploy the outsourcing framework since we could use more than one secure coprocessor to try and make up for these trade-offs.

Chapter 8: Conclusions & Future Work

In terms of scalability, we argue that our model finds applications that may involve numerous tables and where joins would result in medium sized results. Because of the general overhead of the framework that wouldn't be there in an ordinary query-response scenario, we do not recommend the use of this model for very small databases where joins would *always* result in less than 100 rows.

The first prototype supports multithreading at the server side of the code. We need to extend this to inside the sCPU. While multiple clients can connect to the system and issue queries, the processing inside the sCPU can only handle one query at a time. We need to add code to handle this.

We propose to replace the forwarder with a stored procedure running on the server integrated with a psql client on the server. This will make the mechanism even more transparent: the client would submit an encrypted query directly though its psql interface (or the proxy in case of JDBC connections). This would also eliminate the need to open new connections as we do now (twice), for processing a single query.

Secondly, we need to get rid of the main bottleneck being encountered in our prototype: the time taken to transfer join attributes and TagIDs from the server into the sCPU. In our current implementation we are performing file I/O for this purpose in the form of the following query:

./psql -h server_ip -p server_port -U postgres -c "COPY some_table(tagid,join_attribute) TO STDOUT WITH CSV" some_db > some_file

With the implementation of our stored procedure we should be able to complete the same task in memory without redirecting output to a file. This should lead to a more efficient approach.

One other thing to find out is how much faster transferring information on the bus is versus the network. In theory transferring over the bus is indeed faster than network transfers. In Section 6.7 we show the throughput during bus transfer as well. But since the main server and the sCPU reside under the same network per say, with no other hosts interfering in communication and since our Ethernet simulation code aims to keep intact the basic network socket calls used in the code, we need to figure out if transfers over the bus will be largely faster or not. In the worst case, they will be the same if not faster no doubt, but we need to quantify this difference.

Bibliography

[1] Hardware overview of the IBM 4764 and PCIXCC feature
http://www-03.ibm.com/security/cryptocards/pcixcc/overhardware.shtml

[2] IBM 4764 product and PCIXX feature summary
http://www-03.ibm.com/security/cryptocards/pcixcc/overproduct.shtml

[3] Validating the IBM 4764 PCI-X Cryptographic Coprocessor
http://www-03.ibm.com/security/cryptocards/pcixcc/validation.shtml

[4] Anti-Tamper Databases: Processing Aggregate Queries over Encrypted Databases
22nd International Conference on Data Engineering Workshops '06

[5] Authentication and Integrity in Outsourced Databases
ACM Transactions on Storage, Vol. 2, No. 2, May '06

[6] Balancing Confidentiality and Efficiency in Untrusted Relational DBMSs
Conference on Computer and Communications Security '03

[7] Executing SQL over Encrypted Data in the Database-Service-Provider Model
ACM SIGMOD '02

[8] A Database Encryption System with Subkeys
ACM Transactions on Database Systems, Vol. 6, No. 2, June '81

[9] Two Can Keep a Secret: A Distributed Architecture for Secure Database Services
Conference on Innovative Data Systems Research '05

[10] Security Issues in Querying Encrypted Data
Purdue Computer Science Technical Report 04-013

[11] Practical Server Privacy with Secure Coprocessors
IBM Systems Journal, Vol. 40, No. 3, '01

[12] Incorporating a Secure Coprocessor in the Database-as-a-Service Model
Innovative Architecture for Future Generation High-Performance Processors and Systems '05

[13] Executing SQL over Encrypted Data in the Database-Service-Provider Model
ACM SIGMOD '02

[14] Sqlite3 Database Engine
http://www.sqlite.org