cDB: SCPU-enabled Secure SQL Engine

Rajarshi Agnihotri, Radu Sion
Computer Science Deptartment, Stony Brook University, Stony Brook, NY, USA
rajarshi, sion(@cs.sunysb.edu)

Setups for outsourcing databases work on the assumption that the hosting server is trusted. Existing efficient 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. In cDB, we 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 sCPU). 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.

The IBM 4764 secure coprocessor is validated at the highest level under the stringent FIPS PUB (Federal Information Processing Standards Publication) 140-2 standard. It has a 266MHz Motorola PowerPC-405 CPU with a 66MHz PCI bus clock and contains specialized hardware to perform AES, DES, TDES, RSA, and SHA-1 cryptographic processes. It also protects cryptographic keys and sensitive custom applications.

The untrusted database service provider will try to alter or steal the data and will try to infer all information about the data, the metadata, the queries, and the query patterns among other things. Our aim is to prevent or at least detect these activities.

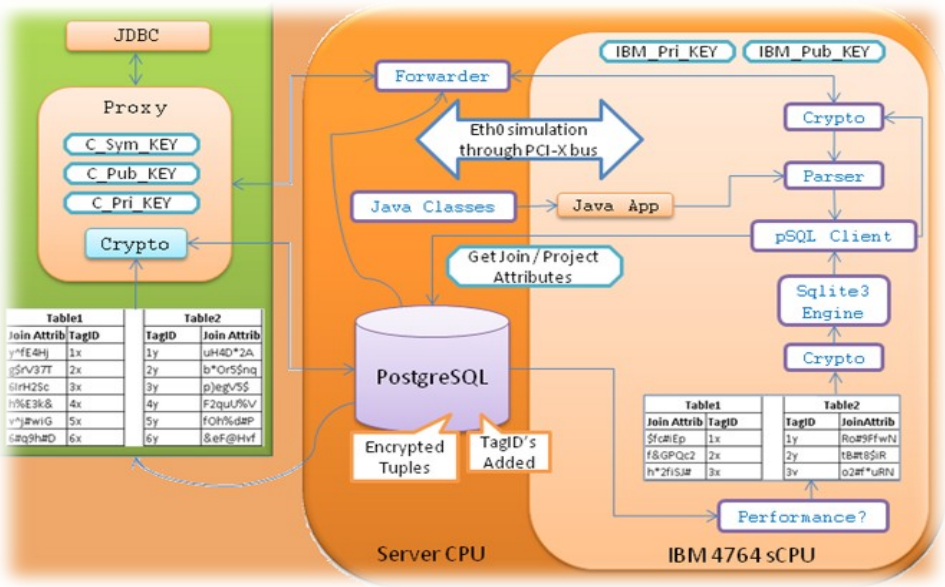
The client has the ability to decrypt the data. It has a public-private key pair and it is aware of the sCPU's public key.

For resolution of a query, the client submits the query to the database service provider (host) server. The server forwards the query to the sCPU. The sCPU decrypts the query, parses it and converts it into two or more queries. Some of these queries are sensitive and some are non-sensitive. The sCPU then requests from the server, the data required to resolve the sensitive queries. After the sensitive queries are resolved, it sends the non-sensitive queries to the server. The server resolves the non-sensitive queries and sends the results back to the client. The sCPU also

sends some signature on the result to the client so that the client can verify if the server returned correct results. The client decrypts the final result and verifies its integrity using the signature sent by the sCPU.

To make the query process transparent to the user, we run an application with a standard database interface on the client. User submits the queries to this application instead of submitting them to the server. This application signs the query with user's public key and encrypts it with sCPU's public key and sends it over to the server. Similarly, upon receiving the final encrypted results from the server, it verifies the sCPU signature, decrypts the results and returns them to the user.

The main database is stored on the server and



the tuples and attribute names are encrypted by a key known only to the client and the sCPU. Each table in the database has a different encryption key. In each table of the database, we introduce an additional un-encrypted attribute that serves as an identification. The key to each tuple of the table is derived from the table specific key and this un-encrypted attribute (TagID). This way each row of a table has a different key and any two tuples in different tables have different keys. So the same value in different tables or in different tuples of the same table, will be encrypted differently.

A stored procedure at the server listens for encrypted queries sent by the client and transfers them to the sCPU. It maintains one connection with the sCPU and one with the client for the duration of query resolution. It fulfills all data requests of the sCPU. It receives the authentication signature on the result from the sCPU and computes final encrypted results and transfers them to the client. The data never stays in plain on the server.

The sCPU Cryptographic Module (incoming/outgoing) decrypts and authenticates queries received from the client via the server. The module also provides authentication through signature on outgoing data (query results) to the client via the server. This module also decrypts the data required to resolve the sensitive queries.

The parser parses the query in order to identify sensitive and non-sensitive parts of a query. This identification is used by the rest of

the system to resolve sensitive queries on the sCPU and insensitive queries on the server.

We employ a two pronged strategy for resolution of the sensitive queries on the sCPU. For simple queries, we use custom C programs to resolve them. For complex queries, we use the SQLite library. This gives us the best combination of performance and ease of implementation. The results can then be sent to the server along with non-sensitive queries for the execution of non-sensitive queries.

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

We clock timings for special 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 inside 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. In the prototype, we are not including the costs for encryption/decryption operations.

The times taken by the custom C code implementation show dramatic improvement over the SQLite library implementation. The same query when run on the custom C code, takes less than a second for a join resulting in 10,000 output rows. The times are taken at the server.

Size(kb)	Rows	Total Time (s)	Misc Time (s)	Fetch Time (s)	Join Time (s)
140	10000	0.935	0.934	0.080	0.212
1400	99999	4.723	1.897	0.610	2.216

We have implemented the prototype for the basic queries, all data required for which can fit in the sCPU. We still have to devise strategies to resolve the queries for which all data does not fit in the sCPU. We also need to implement a RAM file system to be used as a swap disk for sCPU. We also need to devise strategies to be able to import a non-deterministic superset of a range of encrypted data based on some condition on the plaintext.