TrustedDB: A Trusted Hardware based Database with Privacy and Data Confidentiality

Sumeet Bajaj
Stony Brook Computer Science
Stony Brook, New York, USA
sbajaj@cs.stonybrook.edu

Radu Sion
Stony Brook Computer Science
Stony Brook, New York, USA
sion@cs.stonybrook.edu

ABSTRACT

TrustedDB is an outsourced database prototype that allows clients to execute SQL queries with privacy and under regulatory compliance constraints without having to trust the service provider. TrustedDB achieves this by leveraging server-hosted tamper-proof trusted hardware in critical query processing stages.

TrustedDB does not limit the query expressiveness of supported queries. And, despite the cost overhead and performance limitations of trusted hardware, the costs per query are orders of magnitude lower than any (existing or) potential future software-only mechanisms. TrustedDB is built and runs on actual hardware, and its performance and costs are evaluated here.

Categories and Subject Descriptors

H.2.0 [**DATABASE MANAGEMENT**]: General—*Security, integrity, and protection*; H.2.4 [**DATABASE MANAGEMENT**]: Systems—*Query Processing*

General Terms

Security

Keywords

Database, Trusted Hardware

1. INTRODUCTION

Outsourcing has finally arrived, due in no small part to the availability of cheap high speed networks, storage and CPUs. Clients can now minimize their management overheads and virtually eliminate infrastructure costs[1].

Virtually all major "cloud" providers today offer a database service of some kind as part of their overall solution. Numerous startups also feature more targeted data management and/or database platforms.

Yet, significant challenges lie in the path of large-scale adoption. Such services often require their customers to inherently trust the provider with full access to the outsourced datasets. But numerous instances of illicit insider behavior or data leaks have left clients reluctant to place sensitive data under the control of a remote, third-party provider, without practical assurances of *privacy* and *confidentiality* – especially in business, healthcare and government frameworks. And today's privacy guarantees of such services are at best declarative and subject customers to unreasonable fine-print clauses – e.g., allowing the server operator (or malicious attackers gaining access to its systems) to use customer behavior and content for commercial, profiling, or governmental surveillance purposes [15, 16].

Existing research addresses several such outsourcing security aspects, including access privacy, searches on encrypted data, range queries, and aggregate queries. To achieve privacy, in most of these efforts data is encrypted before outsourcing. Once encrypted however, inherent limitations in the types of primitive operations that can be performed on encrypted data lead to fundamental expressiveness and practicality constraints.

Recent theoretical cryptography results provide hope by proving the existence of universal homomorphisms, i.e., encryption mechanisms that allow computation of arbitrary functions without decrypting the inputs [43]. Unfortunately actual instances of such mechanisms seem to be decades away from being practical [19].

Ideas have also been proposed to leverage tamper-proof hardware to privately process data server-side, ranging from smartcard deployment [29] in healthcare, to more general database operations [8, 28, 31].

Yet, common wisdom so far has been that trusted hardware is generally impractical due to its performance limitations and higher acquisition costs. As a result, with very few exceptions [29], these efforts have stopped short of proposing or building full - fledged database processing engines.

However, *recent insights* [14] *into the cost-performance trade-off seem to suggest that things stand somewhat differently.* Specifically, at scale, in outsourced contexts, *computation inside secure processors is orders of magnitude cheaper than any equivalent cryptographic operation performed on the provider's unsecured common server hardware*[2], despite the overall greater acquisition cost of secure hardware.

[1] Discussing the merits or faults of outsourcing and "clouds" is beyond the scope of this paper. Others have done and continue to do so [7, 40, 41].

[2] e.g., it is much cheaper to add numbers privately inside expensive cryptographic coprocessors than to perform the ex-

SIGMOD'11, June 12–16, 2011, Athens, Greece.

This is so because cryptographic overheads (for cryptography that allows some processing by the server) are extremely high even for simple operations, a fact rooted not in cipher implementation inefficiencies but rather in fundamental cryptographic hardness assumptions and constructs (such as trapdoor functions – the cheapest we have so far being at least as expensive as modular multiplication [35]). This is unlikely to change anytime soon (none of the current primitives have, in the past half-century). New mathematical hardness problems (e.g., elliptic curve cryptography – which unfortunately is only a bit more efficient) will need to be discovered to allow hope of more efficient cryptography.

As a result, we posit that a full-fledged strong-privacy enabling secure database leveraging server-side trusted hardware can be built and run at a fraction of the cost of any (existing or future) cryptography-enabled private data processing on common hardware. We validate by designing and implementing TrustedDB, an SQL database processing engine that makes use of IBM 4764/5 [5, 6] cryptographic coprocessors programmed to run custom components securely.

Tamper resistant designs however are significantly constrained in both computational ability and memory capacity which makes implementing fully featured database solutions using secure coprocessors (SCPUs) very challenging. TrustedDB achieves this by utilizing common unsecured server resources to the maximum extent possible. For example TrustedDB enables the SCPU to transparently access external storage while preserving data confidentiality with on-the-fly encryption. This eliminates the limitations on the size of databases that can be supported. Moreover, client queries are pre-processed to identify sensitive components to be run inside the SCPU. Non-sensitive operations are off-loaded to the untrusted host server. This greatly improves performance and reduces the cost of transactions.

Overall, despite the overheads and performance limitations of trusted hardware, the costs of running TrustedDB are orders of magnitude lower than any (existing or) potential future cryptography-only mechanisms. The TrustedDB design provides strong data confidentiality assurances. Moreover, it does not limit query expressiveness.

The contributions of this paper are two-fold: (i) the introduction of new cost models and insights that explain and quantify the advantages of deploying trusted hardware for data processing, and (ii) the design, development, and evaluation of TrustedDB, a trusted hardware based relational database with full data confidentiality.

2. MODEL AND TOOLS

Deployment. We will consider the following concise yet representative deployment model. Sensitive data is placed by a client with a remote (untrusted) *service provider*. For confidentiality, parts of the data are encrypted before outsourcing. Later, the client or an authorized third party queries the outsourced datasets through an interface exposed by the server. It is imperative for the client that certain sensitive portions of the database are never revealed. Queries will need to be performed by the outsourcing server with full computational privacy. Moreover, the client prefers to not be restricted in the nature of queries. Network layer confidentiality is assured by mechanisms such as SSL/IPSec. Other security issues such as authentication and authorization have been extensively addressed in existing research and therefore are not the main focus here.

Adversary. For strong assurances, we consider the server to be untrusted and curious. Given the possibility to get away undetected, it will attempt to compromise data confidentiality. Overt denial of service attacks are not of interest.

Trusted Hardware. TrustedDB leverages the existence of trusted hardware such as the IBM 4758 PCI [4] and the newer IBM 4764 PCI-X [5] cryptographic coprocessors (SCPUs) in close data proximity. The 4764 is a PowerPC 405 based board, runs embedded Linux and can be custom programmed. Both SCPUs feature tamper resistant and responsive designs [27] and thus provide a secure execution environment. In the eventuality of illicit physical handling, the devices destroy their internal state and shut down.

However, tamper resistant designs face major challenges in heat dissipation and as a result, SCPUs are significantly constrained in both computation ability and memory capacity. This requires careful consideration in achieving efficient protocols. Efficient solutions can be achieved by balancing a trade-off in which cheap untrusted main CPUs perform unsensitive query components while operations on private sensitive data are performed inside the SCPUs.

Cryptography. The deployed SCPUs offer several cryptographic primitives including symmetric key encryption (AES, 3DES), public key encryption on key lengths up to 4096 bits (RSA), pseudo random number generation and cryptographic hash functions (e.g., in the SHA-family).

We will denote by $E(M, K)$ the encryption of message M with key K. PK_{ALICE} denotes a public key that belongs to Alice while SK_{ALICE} represents her private key. Digital signatures are written as $S(M, K)$ – a signature of message M with private key K. The cryptographic hash of message M is denoted by $H(M)$. $||$ represents concatenation. $\oplus$ denotes the bit-XOR operation.

3. THE REAL COSTS OF SECURITY

Traditional wisdom suggests that deploying trusted hardware is generally feasible only in niche-markets such as banking and ATM security while generally impractical due to performance limitations and high acquisition costs.

This idea has propagated also to the secure outsourcing realm mainly due to the belief that practical data confidentiality and protection against curious and malicious insiders can be achieved on untrusted common hardware in "software" only, e.g., by wisely applied combinations of cryptography and data security protocols.

Further, confirmation has been achieved, for *correctness* assurances (no data confidentiality) where this seems to be indeed the case at least for simple types of queries. Several results provide relatively efficient mechanisms that guarantee query correctness e.g., for range queries [25, 30].

Yet, as soon as confidentiality becomes a concern, data needs to be encrypted before outsourcing. Once encrypted, solutions can be envisioned that: (A) straightforwardly transfer data back to the client where it can be decrypted and queried, (B) deploy cryptographic constructs server-side to process encrypted data directly or indirectly, and (C) process the encrypted data server-side inside tamper-proof enclosures of trusted hardware (which the client trusts).

In the remainder of this section we will compare the per-transaction costs of each of these cases. This is possible

pensive cryptography needed to add the numbers encrypted on plain server hardware with privacy.

	H, S	M	L
monthly	$44.90	$95	$13
bandwidth (d/u)	15/5 Mbps	per 1Mbps	per 1Mbps
dedicated	No	Yes	Yes
picocent/bit	115/345	3665	500

Figure 1: Network service costs [12–14].

in view of novel results of Chen et al. [12–14] that allow such quantification. We will show that, at scale, in outsourced contexts, (C) *computation inside secure hardware processors is orders of magnitude cheaper than any equivalent cryptographic operation performed on the provider's unsecured common server hardware* (B). Moreover, due to the extremely high cost of networking when compared with computation, the overhead of transferring (even a small subset of the data) back to the client (for decryption and processing) in (A) is overall significantly more expensive than (C).

The main intuition behind this has to do with the amortized cost of CPU cycles in both trusted and common hardware, as well as the cost of transferring network bits. Due to economies of scale, provider-hosted CPU cycles are 1-2 orders of magnitude cheaper than that of clients, for CPU cycles in traditional hardware but most importantly also for *trusted* hardware. The cost of a CPU cycle in trusted hardware (56+ picocents[3], discussed below) becomes thus of the same order as the cost of a traditional CPU cycle at client sites (e.g., 14-27 picocents for small businesses). These include both acquisition and running components.

Additionally, when data is hosted far from its accessing clients, the extremely expensive network traffic often dominates; transferring *a single bit* of data over a network costs upwards of 3500 picocents!

Finally, cryptography that would allow processing on encrypted data demands extremely large numbers of cycles even for very simple operations such as addition. And, this limitation is rooted in fundamental cryptographic hardness assumptions and constructs (such as cryptographic trapdoors – the cheapest we have so far being at least as expensive as modular multiplication [35] – which comes at a pricetag of upwards of *tens of thousands of picocents per operation* [14]) and is unlikely to change anytime soon (none of the current primitives have, in the past half-century).

The above insights lead to (C) being a significantly more cost-efficient solution than (A) and (B). We now detail.

3.1 Cost of Primitives

Compute Cycles and Networks. In [12–14] Chen et al. derived the cost of compute cycles for a set of environments ranging from individual homes with a few PCs (H) to large enterprises and compute clouds running upwards of tens of thousands of CPUs (L) (M,L=medium,large sized business). These costs include a number of factors, such as hardware (server, networking), building (floor space leasing), energy (electricity), service (personnel, maintenance, administration) etc.

Their main thesis is that, because of the economies of scale and associated favorable operating parameters, per-cycle costs decrease dramatically when run in large compute providers' infrastructures (Figure 2). The resulting costs range from 27 picocents for a small business environment to less than half of a picocent for large cloud computing providers with infrastructures of tens of thousands of servers.

[3]1 US picocent = 10^{-14} USD

Parameters	H	S	M	L
Scale	<10	<1000	<10k	>10k
CPU utilization	5-8%	10-12%	15-20%	40-56%
server:admin ratio	N.A.	100-140	140-200	800-1000
Space ($/sqft/month)	N.A.	0.5	0.5	0.25
PUE	N.A.	2-2.5	1.6-2	1.2-1.5
Hardware ($/CPU)	750	500	500	350
Electricity ($/KW)	0.09	0.07	0.07	0.06
CPU Cycle (picocent)	5	14-27	2	<0.5

Figure 2: Summarized costs and key parameters for different computing environments [12–14].

Network service costs (Figure 1) range from a few hundred picocents per bit for *non-dedicated* service to thousands of picocents in the case of medium sized businesses.

Validation. We chose to validate these analytical numbers by exploring today's offerings and their pricing points. Results are surprisingly close. Amazon for e.g., charges 1-2.5 picocents per cycle (this price includes their markup as well as additional resources such as local storage and some networking traffic), whereas rackspace.com's CPU cycles range from 0.3-2.4 picocents. Similarly, amazon.com's network service can be had at price points ranging from 800 to 2500 picocents per bit, depending on source and destination region.

Trusted Hardware. We now evaluate the cost of SCPU cycles. Two main cost-impacting differences between general purpose trusted hardware SCPUs and traditional hardware are: reduced computation speeds and increased pricing points. The considered IBM 4764/5 SCPUs are connected to the world through PCI-X/PCIe buses and motherboards controlled by traditional CPUs. The main CPUs act only as a communication conduit for the SCPUs and do not need to spend time to service the SCPU otherwise.

For simplicity we consider only one SCPU per main CPU. For tighter packing of multiple SCPUs per motherboard (up to 4), the overall efficiency of the system is even higher.

SCPU energy consumption peaks by design at only 25W, yet, we conservatively estimate the deployment of SCPUs effectively doubles energy and service costs i.e., to defend against any critique claiming that in effect the main CPU will need to be dedicated as a communication conduit. In reality message forwarding takes only a few percentage points of the main CPU's compute capacity.

SCPUs are expensive (roughly one order of magnitude higher than traditional server hardware. At the time of writing the IBM 4764 could be purchased for around $8,000 at retail which is the number we will deploy in our estimation (unofficial bulk pricing is around $5,000 excluding support services). Finally, the 4764 runs a Motorola PowerPC 405 RISC CPU at 233 MHz. For consistency, we would like to normalize this to CISC x86 cycles. We performed extensive benchmarking (detailed results are out of scope here) and have identified that branching, integer ops and memory access performance (essential for DBMS query processing) is almost identical to an x86 AMD K6 model 6 at 200 MHz with 512 KB L2-cache – which in turn performs equivalently to a Pentium II core at 166 MHz [2].

Considering these numbers in equation (1) in [12–14] we derive the cost of (x86-equivalent) CPU cycles inside cloud-hosted SCPUs to be approximately 56 picocents. We note that while this is indeed much higher than the < 0.5 picocent cost of a cycle on cloud commodity hardware, it is comparable to the cost of cycles in CPUs hosted in small sized enterprises (14-27 picocents).

We validated these costs also by different avenues, e.g., by considering the core SCPU power consumption, acquisition costs, mean time between failure etc. For 56% utilization we arrived at the more favorable estimate of 48 picocents per cycle. Nevertheless, for consistency, in the following we use the above conservative 56 picocents value.

3.2 Comparison

Given these data points we now compare the (A), (B) and (C) alternatives discussed above. We consider the following simple scenario: a client outsources a dataset composed of integers encrypted to a service provider. The encrypted data is then subjected to a simple aggregation (SUM) query in which the server is to add all the integers and return the result to the client. We chose this mechanism not only for its illustrative simplicity but also because SUM aggregation is one of the very few types of queries for which non-hardware solutions have been proposed in existing research. This allows us to directly compare with existing work. Later we also generalize for arbitrary queries. Figure 3 summarizes the cost analysis that follows.

Querying un-encrypted data. No confidentiality. As a baseline consider the most prevalent scenario today, in which the client's data is stored unencrypted with the service provider. Client queries are executed entirely on the provider's side and only the results are transferred back. Although this is the most cost - effective solution it offers no data confidentiality. The lower bound cost of query execution in this case is as follows[4]:

$Cost_{unencrypted}$ = Cost of addition operations on server + Cost of transmitting results.

$$Cost_{unencrypted} = 2 \cdot D \cdot C_{bit\_transmit} + \left(\frac{N}{D} - 1\right) \cdot C_{cycle\_server} \cdot \eta_{addition} \quad (1)$$

where N is the size of the entire database in bits, $D = 32$ (32 bit integers), C_{cycle_server} is the cost of one CPU cycle on server hardware, $\eta_{addition} = 1$ is the average number of CPU cycles required for an addition operation [26]. $C_{bit\_transmit}$ is the cost of transmitting 1 bit of data from the service provider to the client (discussed above).

(A) Transferring encrypted data to client. The first baseline solution that provides data confidentiality works by transferring the entire database to the client. The client then decrypts the database and performs the aggregation. The cost of this alternative becomes

$Cost_{transfer}$ = Cost of data transmission + Cost of decryption on client + Cost of addition operations on client.

$$Cost_{transfer} = N \cdot (C_{bit\_transmit} + C_{bit\_decryption}) + \left(\frac{N}{D} - 1\right) \cdot C_{cycle\_client} \cdot \eta_{addition} \quad (2)$$

Where $C_{bit\_decryption} = 8$ picocents is the (normalized) cost of decrypting one bit with AES-128 in a medium-sized (M) enterprise and $C_{cycle\_client} = 2$ picocents is the cost of a single client CPU cycle in medium sized enterprises. Naturally we observe that here the cost of transferring the database to the client dominates.

(B) Cryptography. Traditional additive homomorphisms [33, 34, 37] have been used in existing work [22, 42] to allow servers to run aggregation queries over encrypted data. Additive homomorphic encryption allows the computation of the encryption of the sum of a set of encrypted values without requiring their decryption in the process.

Existing homomorphisms require the equivalent work of at least a modular multiplication in performing their corresponding operation (e.g., addition). Moreover, for security, this modular multiplication needs to be performed in fields with a large modulus. For efficiency [42] goes one step further and proposes to perform aggregation in "parallel" by simultaneously adding multiple 32-bit integer values (in fact exactly 32 integer pairs – 32 integers in each 1024 bit chunk). They achieve this by adding two 1024-bit chunks of data (encrypted by the client before outsourcing) at a time. Due to the properties of the Paillier cryptosystem, each such addition involves one 2048-bit modular multiplication[5].

The server then computes the encrypted sum of all such large integers (achieved by a single modular multiplication of the 2048-bit encrypted values – this multiplication happens modulo 2048) and returns the result to the client. The client decrypts the (2048 bit) result into a 1024 bit plaintext, splits this into 32 integers of 32 bits each, and computes their sum. The cost of this scheme is given by

$Cost_{homomorphic}$ = Cost of Modular Multiplications on server + Cost of data transmission + Cost of decryption on client + Cost of addition operations on client

$$Cost_{homomorphic} = \frac{B_h}{D} \cdot \left(\left(\frac{N}{B_h} - 1\right) \cdot C_{modular\_mul} + 2 \cdot B_h \cdot C_{bit\_transmit} + C_{homomorphic\_dec} + \left(\frac{B_h}{D} - 1\right) \cdot C_{cycle\_client} \cdot \eta_{addition} \right) \quad (3)$$

where $B_h = 1024$ is the plaintext block size and $C_{modular\_mul}$ is the cost of performing a single modular multiplication modulo 2048 on the server. $C_{homomorphic\_dec}$ is the cost of performing the single decryption on client and involves modular multiplication and exponentiation.

(C) SCPUs. A possible use of a SCPU in the execution of the aggregation query is to perform the aggregation fully inside the SCPU. The result can then be re-encrypted and transmitted back to the client.

In addition to the core CPU processing costs (which can be computed directly from the cost of SCPU cycles), data transfer overheads are incurred i.e., to bring encrypted data into the SCPU and then transfer the encrypted results back to the host server. The total cost of the solution becomes

$Cost_{scpu}$ = Cost of data transmission between the host server and SCPU + Cost of decryption inside SCPU + Cost of addition operations inside SCPU + Cost of data transmission from server to client + Cost of decryption at client

$$Cost_{scpu} = \left\lceil \frac{N}{B_s} \right\rceil \cdot \left(\delta_{srv} \cdot C_{cycle\_srv} + \delta_{scpu} \cdot C_{cycle\_scpu}\right) + N \cdot C_{bit\_decryption\_scpu} + \left(\frac{N}{D} - 1\right) \cdot C_{cycle\_scpu} \cdot \eta_{addition\_scpu} + B_c \cdot C_{bit\_encryption\_scpu} + B_c \cdot C_{bit\_transmit} + B_c \cdot C_{bit\_decyption\_client} \quad (4)$$

[4]The cost of reading data from storage into main memory is a common factor in all solutions and thus not included here

[5]To process n-bit plaintexts, Paillier operates in $n^2 = 2048$ bit fields for 1024 bit plaintexts. Ciphertexts are 2048 bit.

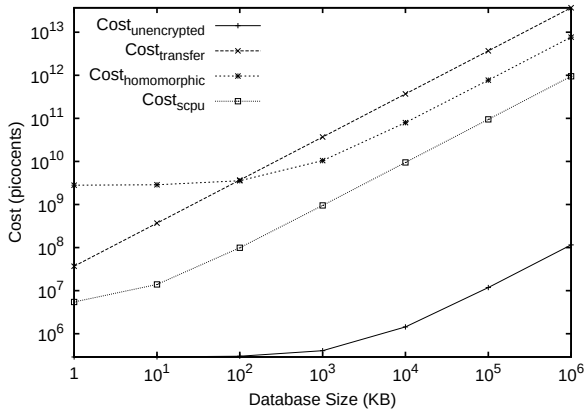

Figure 3: Comparison of outsourced aggregation query solutions (logarithmic)

Where δ_{srv} and δ_{scpu} are the server and SCPU cycles used to setup data transfer and include the cost of setting up and handling DMA interrupts. C_{cycle_scpu} is the cost of a SCPU cycle. $B_s = 64KB$ is the block size of data transmitted between the server and the SCPU in one round and B_c is the cipher block size (128 bits for AES). $\eta_{addition\_scpu} = 2$ is the number of cycles per addition operation in the SCPU for 64 bit addition (on a 32 bit architecture).

Figure 3 shows the cost relationship between the solutions. It can be seen that for any data set sizes $Cost_{scpu} < Cost_{homomorphic}$ and $Cost_{scpu} < Cost_{transfer}$. We also note that for data sets of size $< 100KB$, the cost of client-side homomorphic decryptions (which involves modular exponentiation) dominates and exceeds the data transmission cost in $Cost_{transfer}$. Overall, the use of SCPUs is the most efficient from a cost-centric point of view, by more than an order of magnitude when compared with cryptographic alternatives.

3.3 Generalized Argument

Yet, the use of a specific cryptographic mechanism in the analysis above seems to leave doubts regarding the generality of the conclusion. The fact that this holds today does not seem to imply its validity tomorrow. Thus, it would be important to provide a generalized argument as to why this conclusion should hold in general. We do so here.

Recall that current cryptographic constructs that guarantee confidentiality are based on trapdoor functions [21]. Currently viable trapdoors are based on modular exponentiation in large fields (e.g., 2048 bit modular operations) and viable homomorphisms involve a trapdoor for computing the ciphertexts (the encrypted data on which computations are performed). Additionally, the homomorphic operation itself involves processing these encrypted values at the server in large fields, while respecting the underlying encryption trapdoor. This involves at least the cost of a modular multiplication in large fields (e.g., 2048 or 4096 bits) [33, 34, 37]. This fundamental cryptography has not improved in efficiency in decades and would require the invention of new mathematical tools before such improvements are possible.

Thus, overall, for large scale, efficient deployments (e.g., clouds) where CPU cycles are extremely cheap (e.g., 0.45 picocents/cycle), performing (the cheapest, least secure) homomorphic operations (modular multiplication) comes at a pricetag of at least 30,000 picocents [14] even for values as small as 32-bit (e.g., salaries, zipcodes).

Thus, even if we assume that in future developments homomorphisms will be invented that can allow full Turing Machine languages to be "run" under the encryption envelope (today we can just add or multiply numbers), unless new trapdoor math is discovered (none was in the past 3+ decades), each operation will cost at least 30,000 picocents when run on efficient servers. By comparison, SCPUs process data at a cost of 56 picocents/cycle. This is a difference of several orders of magnitude in cost!. We also note that, while ECC signatures (e.g., even the weak ECC-192) may be faster, ECC-based trapdoors would be even more expensive, as they would require two point multiplications, coming at a price-tag of least 780,000 cycles ([17] page 402).

Yet, this is not entirely accurate, as we also need to account for the fact that SCPUs need to read data in before processing. The SCPUs considered here feature a decryption throughput of about 10-14 MBytes/second for AES decryption [5], confirmed also by our benchmarks. This limits the ability to process data and the real costs of e.g., comparing (in a JOIN operation) two 32-bit integers becomes dominated not by the single-cycle conditional JUMP CPU operation but by the cost of decryption. At 166-200 megacycles/second (recall the 4764 was benchmarked as 166-200MHz x86 equivalent), this results in the SCPU having to idly wait anywhere between 47 and 80 cycles for decryption to happen (in the crypto engine module) before it can process the data. This in effect results in an amortized SCPU cost of between 2632 and 4480 picocents (3556 picocents on average) for each operation which reduces the above 3 orders of magnitude difference to only one order of magnitude, yet still in favor of SCPUs [6].

The above holds even for the case when the SCPU has only enough memory for the two compared values. Further, in the presence of significantly higher, realistic amounts of SCPU memory (e.g., $M = 32$MBytes for 4764-001), optimizations can be achieved for certain types of queries such as relational JOINs. The SCPU can read in and decrypt data pages (instead of single data items) and run the (partial) JOIN query over as many of the decrypted data pages as would fit in memory at one time. This results in significant savings. To illustrate, consider a page size of P (32-bit words) and a simple JOIN algorithm for two tables of size N 32-bit integers each (we're just concerned with the join attribute). Then the SCPU will perform a number of $(N/P)^2 + (N/P)$ page fetches each involving also a page data decryption at a cost of $P \cdot 3556$ picocents[7]. Thus we get a total cost (decryption + operations) of $(\frac{N^2}{P} + N) \cdot 3556 + N^2 \cdot 56$. For reasonable sizes (e.g., $P = M/2/4 = 4$ million 32-bit words can be accommodated) this cost becomes 3+ orders of magnitude lower than the $N^2 \cdot 30000$ picocent cost incurred in the cryptography-based case!

Cost vs. Performance. Given these 3+ orders of magnitude cost advantages of the SCPU over cryptography-based mechanisms, we expect that for the above discussed aggregation query mechanism [42], the SCPU's overall *performance* will also be at least comparable if not better despite the CPU speed handicap. We experimentally evaluated this hy-

[6]The cost can be reduced further by up to 50% if instead of AES, a cipher is built using a faster cryptographic hash as a pseudo-random function [10].

[7]Recall that decrypting 32-bits incurs an amortized cost of around 3556 picocents.

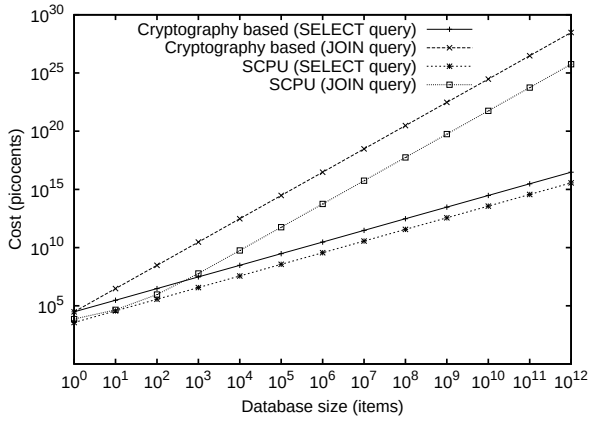

Figure 4: The SCPU is 1-5 orders of magnitude cheaper than deploying cryptography. (logarithmic)

pothesis and achieved a throughput of about 1.07 million tuples/second for the SCPU. By contrast, in [42] best-case scenario throughputs range between 0.58 and 0.92 million tuples/second (and at much higher overall cost).

Why Cost and Not Performance? Overall, we believe – despite being tightly related through the energy factor, *cost per transaction* constitutes an often more appropriate and practical metric of performance than overall *asymptotic runtime complexity* – which is neither meant to capture constants nor any other cost-driven performance factors. Moreover, transaction cost provides increased versatility in system design, allowing fine-tuning of a desired overall cost/performance ratio. This is especially true in scenarios involving highly-parallelizable processing, e.g., relational queries that can often be rewritten to target different data segments in parallel. In such cases, a straightforward strategy for maximizing performance while minimizing cost can be applied: pick the lowest per-transaction cost solution and allocate enough hardware resources to it to achieve a desired transaction throughput. We believe such cost-centric computing paradigms have the potential to fundamentally change the way systems and algorithms are designed.

Conclusion Summary. As Figure 4 illustrates, for linear processing queries (e.g., SELECT) the SCPU is roughly one+ order of magnitude cheaper than any cryptography based mechanisms. For JOIN queries, the SCPU costs drop even further even when assuming no available memory. Finally, in the presence of realistic amounts of memory, due to increased overhead amortization, the SCPU is *multiple* orders of magnitude cheaper than software-only cryptographic solutions on legacy hardware.

Cheaper Niches. We note that the above conclusion *may not* apply to targeted niche scenarios. E.g., it is entirely possible that by maintaining client pre-computed metadata server-side, processing of a pre-defined expected set of queries can be aided. To protect data confidentiality, this metadata requires the use of cryptography (by definition) – but due to the pre-computation step it *may* result in fewer such operations and a lower cost; all this while balancing a storage – expressivity – query arrival rate trade-off. This is a bit counter-intuitive – after all it seems that the access to the encrypted metadata brings us back to square one – but we should not pessimistically discount it yet. Depending on

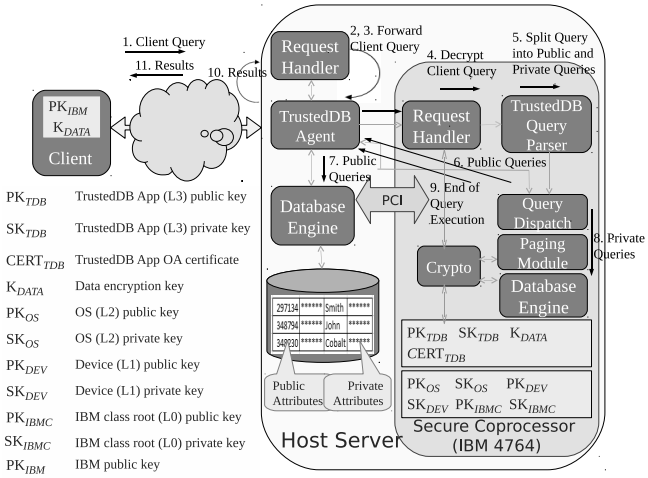

Figure 5: TrustedDB architecture

the cost of the additional required data storage at the time[8] and incoming query patterns, such query-targeted, niche solutions may turn out to be cheaper than general-purpose full-fledged SCPU-backed databases like TrustedDB.

4. ARCHITECTURE

TrustedDB is built around a set of core components (Figure 5) including a *request handler*, a *processing agent and communication conduit*, a *query parser*, a *paging module*, a *query dispatch module*, a *cryptography library*, and two *database engines*. While presenting a detailed architectural blueprint is not possible in this space, in the following we discuss some of the key elements and challenges faced in designing and building TrustedDB.

4.1 Outline

Challenges. The IBM 4764 SCPU presents significant challenges in designing and deploying custom code to be run within its enclosure. For strong security, the underlying hardware code as well as the OS are embedded and no hooks are possible e.g., to augment virtual memory and paging mechanisms. We were faced with the choice of having to provide virtual memory and paging in user land, specifically inside the query processor as well as all the support software. The embedded Linux OS is a Motorola PowerPC 405 port with stripped down libraries required to support the IBM cryptography codebase and nothing else. This constituted a significant hurdle, as cross-compilation became a complex task of mixing native logic with custom-ported functionality. The SCPU communicates with the outside world synchronously through fixed sized messages exchanged over the PCI-X bus in exact sequences. Interfacing such a synchronous channel with the communication model of the query processors and associated paging components required the development of the TrustedDB Paging Module. The SCPU's cryptographic hardware engine features a set of latencies that effectively crippled the ability to run for highly interactive mechanisms manipulating small amounts of data (e.g., 32 bit integers). To handle this we ported several cryptographic primitives to be run on the SCPU's

[8]Which will be a function of both the number of supported queries and the data size. Storing one bit for one year currently costs 100+ picocents [12–14].

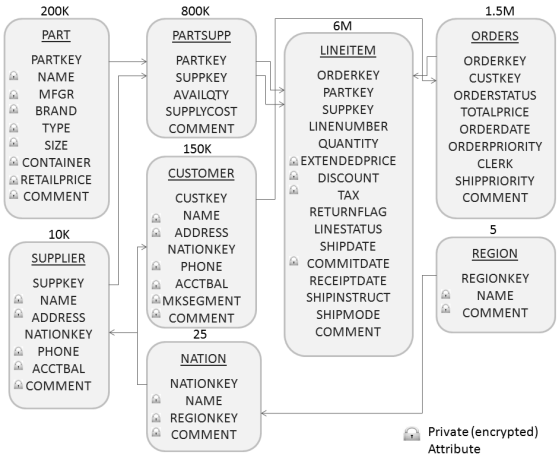

Figure 6: Database Schema

main processor instead, and thus eliminate the hardware latencies for small data items. Space constraints prevent the discussion of the numerous other encountered challenges.

Overview. To remove SCPU-related storage limitations, the outsourced data is stored at the host provider's site. Query processing engines are run on both the server and in the SCPU. Attributes in the database are classified as being either public or private. Private attributes are encrypted and can only be decrypted by the client or by the SCPU.

Since the entire database resides outside the SCPU, its size is not bound by SCPU memory limitations. Pages that need to be accessed by the SCPU-side query processing engine are pulled in on demand by the Paging Module.

Query execution entails a set of stages. (0) In the first stage a client defines a database schema (and partially populates it). Sensitive attributes are marked, i.e., by deploying the "SENSITIVE" keyword that the client layer transparently processes by encrypting the corresponding attributes:

```
CREATE TABLE customer(ID integer primary key,
Name char(72) SENSITIVE, Address char(120) SENSITIVE);
```

(1) Later, a client sends a query request to the host server through a standard SQL interface. The query is transparently encrypted at the client site using the public key of the SCPU. The host server thus cannot decrypt the query. (2) The host server forwards the encrypted query to the Request Handler inside the SCPU. (3) The Request Handler decrypts the query and forwards it to the Query Parser. The query is parsed and rewritten as a set of sub-queries, and, according to their target data set classification, each query is identified as being either public or private. (4) The Query Dispatcher forwards the public queries to the host server and the private queries to the SCPU database engine while handling dependencies. The net result is that the maximum possible work is run on the host server's cheap cycles. (5) The final query result is assembled, encrypted (and digitally signed if correctness assurances are desired) by the SCPU database and the Query Dispatcher and sent back to the client.

4.2 Query Parsing

Outline. Sensitive attributes can occur anywhere within a query (e.g., in SELECT, WHERE or GROUP-BY clauses, in aggregation operators, or within sub-queries). The Query Parser's job is then:

- To ensure that any processing involving private attributes is done within the SCPU. All private attributes are encrypted using a shared data encryption keys between the client and the SCPU (Section 4.4), hence the host server cannot decipher these attributes.
- To optimize the rewrite of the client query such that most of the work is performed on the host server. This significantly increases performance.

To exemplify how public and private queries are generated from the original client query we use examples from the TPC-H benchmark [3]. TPC-H does not specify any classification of attributes based on security. Therefore, we define a attribute set classification into private (encrypted) and public (non-encrypted). The resultant schema is listed in Figure 6. In brief, all attributes that convey identifying information about Customers, Suppliers and Parts are considered private. The resulting query plans (including rewrites into main CPU and SCPU components) for TPC-H queries Q3, Q4, and Q6 are illustrated in Figure 7.

Aggregation Example. For queries that have WHERE clause conditions on public attributes, the server can first SELECT all the tuples that meet the criteria. The private attributes' queries are then performed inside the SCPU on these intermediate results, to yield the final result. For e.g., query $Q6$ of the TPC-H benchmark is processed as shown in Figure 7 (a) [9]. The host server first executes a public query that filters all tuples which fall within the desired *ship date* and *quantity* range, both of these being public attributes. The result from this public query is then used by the SCPU to perform the aggregation on the private attributes *extended price* and *discount*. While performing the aggregation the private attributes are decrypted inside the SCPU. Since the aggregation operation results in a new attribute composing of private attributes it is re-encrypted before sending to the client. This encryption is also done within the SCPU.

Note that the execution of private queries depends on the results from the execution of public queries and vice-a-versa even though they execute in separate database engines. This is made possible by the TrustedDB Query Dispatcher in conjunction with the Paging Module.

Grouping Example. If the client query specifies a GROUP or ORDER BY on public attributes but the selection includes an aggregation of the private attributes, the grouping or sort operation is performed inside the SCPU. Figure 7 (b) illustrates this for the TPC-H query $Q3$. If the aggregation did not involve any private attributes then the host server performs all the GROUP BY and sorting operations.

Nested Queries. The case of nested queries is similar, yet additional care should be taken when computing execution plans to limit the amount of data transfer between the host server and the SCPU which may result in sub-optimal performance. One such example is query $Q4$ of the TPC-H benchmark which includes a sub-query on a private attribute. The query plan illustrated in Figure 7 (c) runs the removal of duplicates on attribute *order key* within the SCPU. An alternative would be to perform this operation on the host server. The choice to do this in the SCPU is made to reduce the traffic over the PCI interface. Efficient handling of nested queries is subject to ongoing work.

[9]More efficient query plans for these queries exist and constitute the subject of ongoing work.

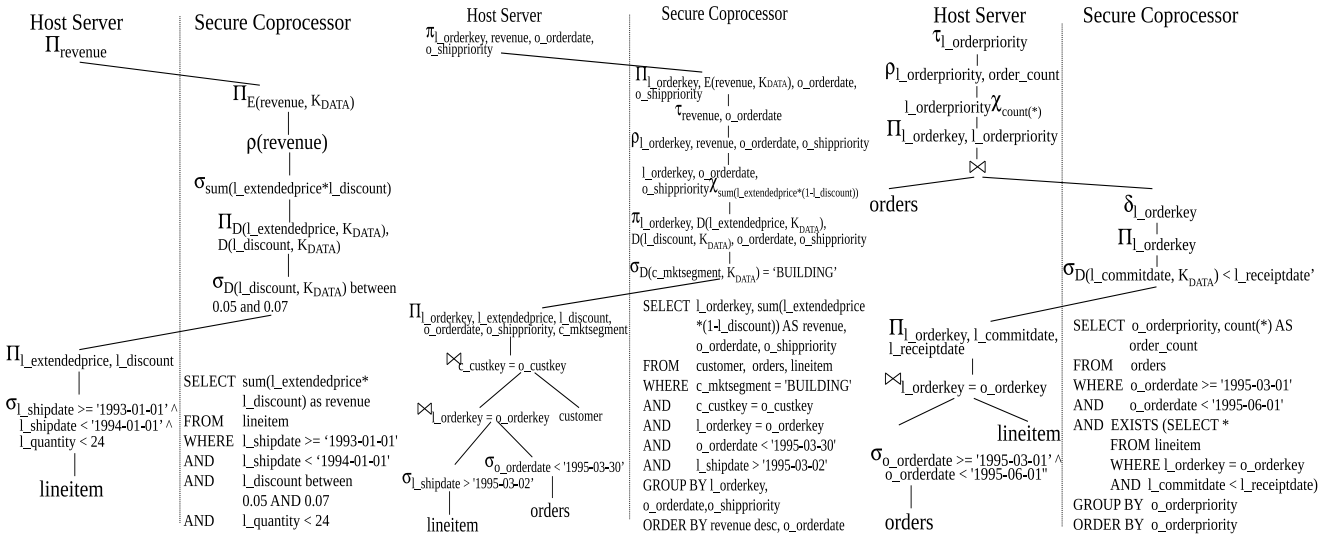

Figure 7: TrustedDB query plans for TPC-H queries (a) Q6, (b) Q3, and (c) Q4.

4.3 Dispatching. Paging. DBMS Engines.

The TrustedDB stack inside the SCPU includes a Query Dispatcher, a Paging Module and a DBMS Engine as well as a general communication conduit library running on top of the PCI-X bus. These components connect to the world through the server/host-side TrustedDB Agent running on the main server CPU.

As discussed above, the Query Dispatcher is in charge of forwarding public sub-queries to the host server and private queries to the SCPU DBMS Engine while handling all dependencies between the sub-queries. The main goal is to maximize the utilization of cheap cycles on the host server.

The SCPU DBMS Engine (a heavily modified PowerPC ported SQLite core) interfaces with the Paging Module to extend its database pages buffer pool with storage outsourced transparently on demand to the server. To achieve this, the TrustedDB Paging Module traps I/O requests made by the SCPU DBMS engine. It then interacts with the external TrustedDB Agent and (en)decrypts (e)ingres pages on demand. This paging mechanism is key in enabling support for large databases.

The main CPU DBMS is an unmodified MySQL 14.12 Distrib 5.0.45 engine. As the TrustedDB stack makes no assumptions about this engine, it can be substituted with any other SQL engine with minimal effort (SQL syntax related).

4.4 Security

Data Encryption is only one of the links in a chain of trust that ensures the security of TrustedDB. Several other assurances are required. Clients need to be confident that (i) the remote SCPU was not tampered with and (ii) runs the correct TrustedDB code stack (including the correct user-land TrustedDB modules as well as the underlying OS and SCPU hardware logic). Finally, clients need to have the means to (iii) communicate secretly with the TrustedDB modules running inside the SCPU.

(i) is assured by the tamper-resistant construction of the SCPU which meets the FIPS 140-2 level 4 [1] physical security requirements. In the event of SCPU tamper detection, sensitive memory areas containing critical secrets are automatically erased. (ii) can be ensured by deploying the SCPU *Outbound Authentication* [39] mechanisms discussed below. (iii) is achieved by deploying public-private key cryptography in key messaging stages. Both, client and the SCPU possess a public-private key pair (Figure 5). Not unlike HTTPS/SSL communication between a web-browser and server, messages sent between the client and the SCPU are encrypted [10]. The client's public key certificate and key PK_{CLIENT} is provided to the SCPU on demand. The SCPU public key is available via Outbound Authentication (OA). While full details are significantly more complex and out of scope [39], we now discuss the main OA insights.

Outbound Authentication. The main idea behind outbound authentication is to provide ways for clients to ascertain the security state of a remote secure processing unit that allows running of arbitrary code. This is achieved by bootstrapping a chain of trust rooted in the clients' trust in known basic pieces of logic running on the SCPU.

The SCPU is divided into four layers running software of differing levels of trust: Layer 0 (Miniboot 0 software), Layer 1 (Miniboot 1 software), Layer 2 (OS), and Layer 3 (application software). The layers can be loaded in increasing order of their number. Once loaded, each layer is issued a public key pair (Figure 5) certified by the layer below. The Layer 0 public key (PK_{IBMC}) is certified by the manufacturer using its private key. The Layer 1 public key (PK_{DEVICE}) is certified by the layer 0 private key (SK_{IBMC}). The Layer 2 public key (PK_{OS}) is certified by the layer 1 private key (SK_{DEVICE}) and finally the Layer 3 public key (PK_{TDB}) is certified by the layer 2 private key (SK_{OS}). This effectively builds a chain of trust rooted in the manufacturer's public key (PK_{IBM}). The chain of all these public key certificates constitutes the *Outbound Authentication certificate*. It can be used to provably assert to remote parties *what software* currently runs inside the SCPU as shown in Figure 8(a). The OA certificate guarantees that the public key PK_{TDB} indeed belongs to an application running inside a SCPU.

Moreover, the OA certificate chain also contains the signed crypto hash of the software installed in each Layer. By veri-

[10] And thus, despite acting as a communication conduit between the client and the SCPU, the server cannot perform man-in-the-middle attacks and gain access to sensitive data.

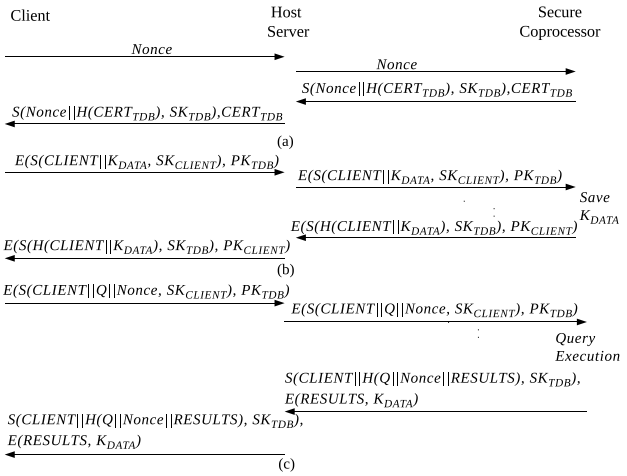

Figure 8: (a) Acquiring Outbound Authentication certificate. (b) Setup of Data Encryption Key. (c) Secure Transmission of query results

fying this software hash against an expected published value (e.g., for the TrustedDB stack) the client can trust the software executing in all the layers. If the software in any layer is updated then the key pairs for that layer and all layers above it will be re-generated and re-attested. The updated software hashes will be available the next time client requests an OA certificate. This satisfies requirement (ii).

The TrustedDB application running within the SCPU generates its own public-private key pair (PK_{TDB}, SK_{TDB}). Any request made by the client to the SCPU is encrypted using the public key PK_{TDB} (or a symmetric key generated out of a Diffie-Hellman key exchange). The client trusts the SCPU public key as it is attested to using the OA certificate.

Data Encryption. While communication security has been discussed above, data confidentiality still remains to be guarded. The client database consists of public (non-encrypted) and private (encrypted) attributes. The private attributes are transparently encrypted by the client layer before the database is uploaded to the host server or as part of any insert/update queries. The encryption is performed using the data encryption key K_{DATA} generated by the client (of at least 160 bits [10]). The TrustedDB stack within the SCPU gains access to this key as shown in Figure 8(b).

Conversely, when query execution generates new *private* attributes they are encrypted inside the SCPU before sending them back to the client. E.g., consider the query `SELECT avg(SALARY) FROM EMPLOYEES`. The result of `avg(SALARY)` is an aggregation over an existing private attribute. In this case the result is encrypted inside the SCPU using the key K_{DATA} (Figure 8(c) - and signed if needed for authenticity) before transmitting back to the client[11].

For increased efficiency, different avenues for data encryption are available. (I) In the simplest case, the entire database is encrypted at page level and TrustedDB is fetching and decrypting *pages* on demand. This features the highest throughput, but limits the ability to use the main CPU's cycles in the process (as the data set is first decrypted inside the SCPU before anything). This case is suitable for queries on data sets that feature ONLY sensitive attributes. (II) For more complex queries, a straightforward application of a symmetric encryption mechanism such as AES can be envisioned. Each tuple is encrypted separately with K_{DATA} and some randomness added to ensure indistinguishability. This mechanism is viable for BLOBS and large-sized attribute values (e.g., hundreds of KBytes+) but less so for small data items such as integers due to encryption latencies and ciphertext blow-up factors. (III) Finally, for fine-grained encryption of data, each individual attribute value within each tuple is encrypted separately with random keys generated by a cryptographic hash function based cipher initialized with K_{DATA} and per-tuple additional data that guarantees its uniqueness across the entire database. Extensive details and cryptographic proofs are out of scope. The result is based on a NMAC construction [10, 21]:

$$E(tbl.attr.val) = ctr_{attr} \,||\, tbl.pri\_key \,||\, idx_K \,||\, (tbl.attr.val \oplus k)$$
$$k = F(K_{DATA}[idx_K] \,||\, ctr_{attr} \,||\, tbl.pri\_key || F(K_{DATA}[idx_K])) \quad (5)$$

where *tbl* is the table name, *tbl.attr* is the attribute to be considered, *tbl.attr.val* is the plaintext value of the current tuple, *tbl.pri_key* is the primary key of the current tuple in table *tbl*, ctr_{attr} is a unique identifying number associated with *tbl.attr*[12], idx_K is an index in a table of K_{DATA} keys which allows multiple such keys to exist simultaneously (and be refreshed periodically) for increased security, and $F(\cdot)$ is a cryptographic hash function (SHA,MD5) [10][13].

5. DISCUSSION

Handling Database Updates. Data manipulation queries (INSERT, UPDATE) also undergo a rewrite. Moreover, any new generated attribute values are re-encrypted within the SCPU before updating the database. For illustration, consider the query `UPDATE EMPLOYEES SET SALARY = SALARY + 2000 WHERE ZIP = 98239`. If `SALARY` is a private attribute then the query works by first decrypting the `SALARY` attribute, performing the addition and then re-encrypting the updated values and is executed within the SCPU. On the other hand if `SALARY` is a public attribute then the query will be executed entirely by the host server.

Limitations of the current query parser. The current query parser does not provide efficient support for query nesting, specifically for queries with large intermediate results. In ongoing work we try to understand whether something can be done to alleviate the costs associated with such large intermediate results that need to be transfered synchronously through the SCPU. A key element in future work will involve designing query parsers to generate efficient query plans while optimizing trade-offs between multiple metrics such as overall dollar cost of execution or performance, in addition to data confidentiality constraints.

Access Patterns. The current implementation reveals access patterns to the externally stored data pages. While this may not be an issue in many applications, it would be interesting to explore the deployment of efficient private information retrieval mechanisms to prevent the server from learning these patterns.

Caching. TrustedDB query execution can result in intermediary results which can often be cached and re-used for

[11]We note that in this case, also correctness guarantees – against an actively malicious server – can be achieved for free, but only for queries involving solely private attributes.

[12]For storage efficiency we don't want to use the entire *tbl.attr* value in the result.

[13]Note that the known MD5 *collision - resistance* related vulnerabilities are not a problem here where it is used as a source of randomness only.

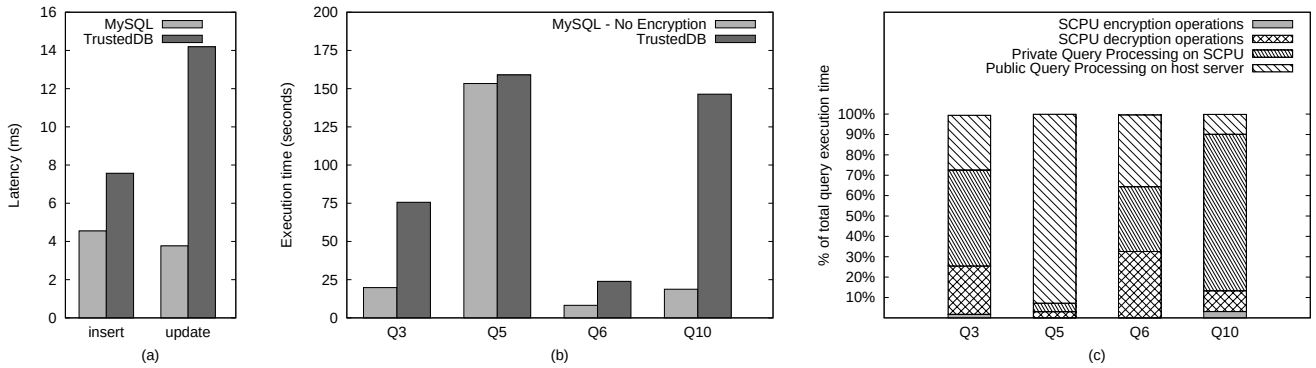

Figure 9: (a) Query latencies. (b) Execution times. (c) Query time profile.

efficiency. Similarly, the query plan decisions need not be repeated, but rather previously devised plans can be cached, e.g., for hot queries. The plans can be encrypted and stored on external storage with the aid of the Paging Module.

Parallelism. The current prototype runs on a single SCPU. An extension to multiple SCPUs is straightforward and would allow better scaling to target desired throughputs.

Compliance. TrustedDB can also be easily augmented with a Compliance Module to support regulatory compliance. Policies that regulate data can be communicated to and enforced by the SCPU securely at runtime. Since sensitive attributes are only accessed within the SCPU the enforcement of policies is guaranteed at very little additional cost. An example of such a compliance policy would be to enforce tuple-level mandatory retention times as well as additional lifecycle management enablers.

6. EXPERIMENTS

Setup. The SCPU of choice is the IBM 4764-001 PCI-X with the 3.30.05 release toolkit featuring 32MBytes of RAM and a PowerPC 405GPr at 233 MHz. The SCPU sits on the PCI-X bus of an Intel Xeon 3.4 GHz, 4GB RAM Linux box (kernel 2.6.18). The server DBMS is a standard MySQL 14.12 Distrib 5.0.45 engine. The SCPU DBMS is a heavily modified SQLite custom port to the PowerPC. The TrustedDB stack (including Communication Conduit, Query Parser, Query Dispatcher, Paging Module, Crypto Engine, TrustedDB Agent etc.) is written in C.

Throughput and Latency. In Section 3 we discussed a benchmarked throughput of 10-14 MB/second for page-level AES encryption. We have also discussed how to achieve a 1+ million tuples/second throughput in an aggregation query on relational data that does not involve public attributes.

While the above numbers are impressive, they only hold for niche cases such as aggregation. For *general* queries, the data needs to be provided encrypted per-tuple with additional fields that allow the SCPU DBMS to reconstruct the individual tuple decryption key (equation (5), Section 4.4).

In such cases, fewer optimizations are possible, and the resulting queries have higher execution times. For example for $F(\cdot)$=MD5 in equation (5), TrustedDB can achieve sustained I/O decryption throughputs of over 80,000 sensitive attribute values per second, and does not limit the non-sensitive data related throughput (processed by the server).

A comparison of MySQL and TrustedDB latencies of INSERT and UPDATE queries are illustrated in Figure 9 (a) and are likely dominated by I/O elements in both cases.

TPC-H Query Load. To evaluate the runtime of generalized queries, instead of concocting our own, we chose several non-nested[14] queries ($Q3$, $Q5$, $Q6$ and $Q10$) from the TPC-H set [3] of varying degrees of difficulty and privacy. These queries all target the standard TPC-H schema in Figure 6 augmented with SENSITIVE attributes. The TPC-H scale factor was 1 (database size of $1GB$).

These queries were selected as they require operations on private attributes and are sufficiently complex. Figure 9 (b) shows the execution times compared to a simple unencrypted MySQL setup.

Figure 9 (c) also depicts the breakdown of times spent in execution of the public (executed on host server) and private (executed on the SCPU) sub-queries. The execution times of private queries include the time required for encryption and decryption operations inside the SCPU. The public queries executed on the host server also include the processing times to interface the TrustedDB stack with the server database engine and output the final results.

As can be seen, when compared with the completely unsecured baseline, security does not come cheap – the execution times are higher by factors between 1.03 and 7.8. These factors benefit from the TrustedDB leveraging of the untrusted server's CPU for non-sensitive query portions. However, recall from Section 3 that the actual costs are orders of magnitude lower than any solution based on software-only cryptography on legacy server hardware.

7. RELATED WORK

Trusted Hardware in Data Management. In [9] SCPUs are used to retrieve X509 certificates from a database.

[38] uses multiple SCPUs to provide key based search. The entire database is scanned by the SCPUs to return matching records. [36, 44] implement arbitrary joins by reading the entire database through the SCPU.

Chip-Secured Data Access [29] uses a smart card for query processing and for enforcing access rights. The client query is split into three parts - server, client and a terminal query. The idea is that the server query performs majority of the computation. The solution is limited by the fact that the client query executing within the smart cannot generate any

[14]Recall that the TrustedDB query parser does not yet efficiently support query nesting, especially when it results in large intermediary results.

intermediate results since there is no storage available on the card. In follow-up work, GhostDB [32] proposes to embed a database inside an USB key equipped with a CPU. It "allows linking [of] private data carried on the USB Key and public data available on a public server. GhostDB ensures that the only information revealed to a potential spy is the query issued and the public data accessed."

[28, 31] explore the use and limitations of SCPUs for outsourced databases and "sketch how a practical system meeting the definition could be built and proven secure".

[8] discusses the use of a SCPU-based service in a secure multi-party setting in which " participating data providers send encrypted relations to the service that sends the encrypted results to the recipients".

In [11] a database engine is proposed inside a SCPU for data sharing and mining. The SCPU fetches data from external sources using secure jdbc connections. The entire data is treated as private which means that queries must be completely executed inside the coprocessor and the host server resources are not utilized. The overhead of jdbc connections is between 20% and 70% of the total query processing time.

Queries on Encrypted Data. Hacigumus et al. [23] propose a method to execute SQL queries over partly obfuscated outsourced data. The data is divided into secret partitions and queries over the original data can be rewritten in terms of the resulting partition identifiers; the server then performs queries over the partitions. The information leaked to the server is claimed to be 1-out-of-s where s is the partition size. This balances a trade-off between client-side and server-side processing, as a function of the data segment size. At one extreme, privacy is completely compromised (small segment sizes) but client processing is minimal. At the other extreme, a high level of privacy can be attained at the expense of the client processing the queries in their entirety after retrieving the entire dataset.

In [24] the authors explore optimal bucket sizes for certain range queries. Similarly, data partitioning is deployed in building "almost"-private indexes on attributes considered sensitive. An untrusted server is then able to execute "obfuscated range queries with minimal information leakage". An associated privacy-utility trade-off for the index is discussed. The main drawbacks of these solutions lies in the fact that the cost-security trade-off is linear. The increase in security is linear in the increase in the cost of the solution, not unlike case (A) in Section 3.

Aggregation queries over relational databases is provided in [22] by making use of homomorphic encryption based on Privacy Homomorphism [37]. The authors in [18] have suggested that this scheme is vulnerable to a cipher text only attack. Instead [18] proposes an alternative scheme to perform aggregation queries based on bucketization [23]. Here the data owner precomputes aggregate values such as SUM and COUNT for partitions and stores them encrypted at the server. Although this makes processing of certain queries faster it does not significantly reduce client side processing.

Query Correctness. Numerous query correctness solutions for outsourced frameworks have been proposed. Due to space constraints we direct the reader to [20].

8. CONCLUSIONS

This paper's contributions are two-fold: (i) the introduction of new cost models and insights that explain and quantify the advantages of deploying trusted hardware for data processing, and (ii) the design and development of TrustedDB, a trusted hardware based relational database with full data confidentiality and no limitations on query expressiveness.

This work's inherent thesis is that, at scale, in outsourced contexts, computation inside secure hardware processors is orders of magnitude cheaper than any equivalent cryptography performed on a provider's unsecured common server hardware, despite the overall greater acquisition cost of secure hardware. We thus propose to make trusted hardware a first-class citizen in the secure data management arena. Moreover, we hope that cost-centric insights and architectural paradigms will fundamentally change the way systems and algorithms are designed.

9. ACKNOWLEDGMENTS

We would like to thank Sumeet Dash and Nilesh Mahajan for their help with implementing TrustedDB, as well as our sponsors, NSF (0937833, 0845192, 0803197), CA Technologies, NGC, Xerox, IBM, and Microsoft.

10. REFERENCES

[1] FIPS PUB 140-2, Security Requirements for Cryptographic Modules. Online at `http://csrc.nist.gov/groups/STM/cmvp/standards.html#02`.

[2] The UBENCH Toolkit. Online at `http://www.phystech.com/download/ubench.html`.

[3] TPC-H Benchmark. Online at `http://www.tpc.org/tpch/`.

[4] IBM 4758 PCI Cryptographic Coprocessor. Online at `http://www-03.ibm.com/security/cryptocards/pcicc/overview.shtml`, 2006.

[5] IBM 4764 PCI-X Cryptographic Coprocessor. Online at `http://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml`, 2007.

[6] IBM 4765 PCIe Cryptographic Coprocessor. Online at `http://www-03.ibm.com/security/cryptocards/pciecc/overview.shtml`, 2010.

[7] Daniel Abadi, Michael J. Carey, Surajit Chaudhuri, Hector Garcia-Molina, Jignesh M. Patel, and Raghu Ramakrishnan. Cloud databases: What's new? *PVLDB*, 3(2):1657, 2010.

[8] Rakesh Agrawal, Dmitri Asonov, Murat Kantarcioglu, and Yaping Li. Sovereign joins. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *ICDE*, page 26. IEEE Computer Society, 2006.

[9] Alexander Iliev and Sean W Smith. Protecting Client Privacy with Trusted Computing at the Server. *IEEE, Security and Privacy*, 3(2), Apr 2005.

[10] Mihir Bellare. New proofs for nmac and hmac: Security without collision-resistance. pages 602–619. Springer-Verlag, 2006.

[11] Bishwaranjan Bhattacharjee, Naoki Abe, Kenneth Goldman, Bianca Zadrozny, Chid Apte, Vamsavardhana R. Chillakuru and Marysabel del Carpio. Using secure coprocessors for privacy preserving collaborative data mining and analysis. In *Proceedings of the 2nd international workshop on Data management on new hardware*, 2006.

[12] Yao Chen and Radu Sion. On the (Im)Practicality of Securing Untrusted Computing Clouds with

Cryptography. Online at http://www.cs.sunysb.edu/~sion/research/.

[13] Yao Chen and Radu Sion. To Cloud or Not To. Online at http://www.cs.sunysb.edu/~sion/research/.

[14] Yao Chen and Radu Sion. On securing untrusted clouds with cryptography. In *WPES '10: Proceedings of the 9th annual ACM workshop on Privacy in the electronic society*, pages 109–114, New York, NY, USA, 2010. ACM.

[15] CNN. Feds seek Google records in porn probe. Online at http://www.cnn.com, January 2006.

[16] CNN. YouTube ordered to reveal its viewers. Online at http://www.cnn.com, July 2008.

[17] Tom Denis. *Cryptography for Developers*. Syngress.

[18] Einar Mykletun and Gene Tsudik. Aggregation Queries in the Database-As-a-Service Model. *Data and Applications Security*, 4127, 2006.

[19] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2010.

[20] Michael Gertz and Sushil Jajodia. *Handbook of Database Security: Applications and Trends*. Springer.

[21] O. Goldreich. *Foundations of Cryptography I*. Cambridge University Press, 2001.

[22] Bala Iyer Hakan Hacigumus and Sharad Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In *Database Systems for Advanced Applications*, volume 2973, pages 633–650, 2004.

[23] Hakan Hacigumus, Bala Iyer, Chen Li and Sharad Mehrotra. Executing SQL over Encrypted Data in the Database-Service-Provider Model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 216–227, 2002.

[24] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *Proceedings of ACM SIGMOD*, 2004.

[25] HweeHwa Pang and Arpit Jain and Krithi Ramamritham and Kian-Lee Tan. Verifying Completeness of Relational Query Results in Data Publishing. In *Proceedings of ACM SIGMOD*, 2005.

[26] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2008.

[27] Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer and Leendert van Doorn. Building the IBM 4758 Secure Coprocessor. *IEEE*, 34(10), 2001.

[28] Murat Kantarcioglu and Chris Clifton. Security issues in querying encrypted data. In Sushil Jajodia and Duminda Wijesekera, editors, *DBSec*, volume 3654 of *Lecture Notes in Computer Science*, pages 325–337. Springer, 2005.

[29] Luc Bouganim and Philippe Pucheral. Chip-secured data access: confidential data on untrusted server. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 131–141. VLDB Endowment, 2002.

[30] Maithili Narasimha and Gene Tsudik. DSAC: integrity for outsourced databases with signature aggregation and chaining. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 235–236, 2005.

[31] Einar Mykletun and Gene Tsudik. Incorporating a secure coprocessor in the database-as-a-service model. In *IWIA '05: Proceedings of the Innovative Architecture on Future Generation High-Performance Processors and Systems*, pages 38–44, Washington, DC, USA, 2005. IEEE Computer Society.

[32] Nicolas Anciaux, Mehdi Benzine, Luc Bouganim, Philippe Pucheral and Dennis Shasha. GhostDB: Querying Visible and Hidden Data Without Leaks. In *Proceedings of the ACM SIGMOD international conference on Management of data*, 2007.

[33] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of EuroCrypt*, 1999.

[34] Pascal Paillier. A trapdoor permutation equivalent to factoring. In *PKC '99: Proceedings of the Second International Workshop on Practice and Theory in Public Key Cryptography*, pages 219–222, London, UK, 1999. Springer-Verlag.

[35] M. O. Rabin. Digitalized signatures and public-key functions as intractable as factorization. Technical report, Cambridge, MA, USA, 1979.

[36] Rakesh Agrawal, Dmitri Asonov, Murat Kantarcioglu, Yaping Li. Sovereign Joins. In *Proceedings of the 22nd International Conference on Data Engineering*, page 26. IEEE Computer Society, 2006.

[37] Ronald Rivest, Len Adleman and Michael Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, 1978.

[38] S. W. Smith and D. Safford. Practical server privacy with secure coprocessors. *IBM SYSTEMS JOURNAL*, 40(3), 2001.

[39] Sean W. Smith. Outbound authentication for programmable secure coprocessors. Online at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.58.4066.

[40] Michael Stonebraker, Daniel J. Abadi, David J. DeWitt, Samuel Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[41] Alexander Thomson and Daniel J. Abadi. The case for determinism in database systems. *PVLDB*, 3(1):70–80, 2010.

[42] Tingjian Ge and Stan Zdonik. Answering Aggregation Queries in a Secure System Model. In *Proceedings of the 33rd international conference on Very large data bases*, pages 519–530. VLDB Endowment, 2007.

[43] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, 2010.

[44] Yaping Li. Privacy Preserving Joins on Secure Coprocessors. Technical Report UCB/EECS-2008-158, EECS Department, University of California Berkeley, Dec 2008. http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-158.html.