Abstract of thesis entitled

“A Novel Architecture for Secure Database Processing in Cloud Computing”

submitted by

Hung-Kwan Chen

for the degree of Joint Doctor of Philosophy

at The University of Hong Kong and Imperial College London

in October 2016

Security, particularly data privacy, is one of the biggest barriers to the adoption of Database-as-a-Service (DBaaS) in Cloud Computing. Recent security breaches demonstrate that a more powerful protection mechanism is needed to protect data confidentiality from any honest-but-curious administrator. Typical prior effort on addressing this security problem is either prohibitively slow or highly restrictive in operation.

In this thesis, a novel cloud system architecture CypherDB, which makes use of a secure processor, is proposed to protect the confidentiality of outsourced database processing. To achieve this, a framework is developed to use these secure processors in the cloud for secure database processing. This framework allows distributed and parallel processing of the encrypted data and exhibits virtualization features in Cloud Computing. The CypherDB architecture also relies on two major components to protect the privacy of an outsourced database against any honest-but-curious administrator of high performance.

Firstly, a novel database encryption scheme is developed to protect the outsourced database which can be executed under a CypherDB secure processor with high performance. Our proposed scheme makes use of custom instructions

to hide the encryption latency from the program execution. This scheme is extensively validated through an integration with SQLite, a practical database application program.

Secondly, a novel secure processor architecture is also developed to provide architectural support to our proposed database encryption scheme and efficient protection mechanism to secure all intermediate data generated on-the-fly during query execution. The efficiency, robustness and the cost of our novel processor architecture are validated and evaluated through extensive simulations and implementation on a FPGA platform.

A fully-functional Field-Programmable Gate Array (FPGA) implementation of our CypherDB secure processor and simulation studies demonstrate that our proposed architecture is cost-effective and of high performance. Our experiment of running the TPC-H database benchmark on SQLite demonstrates 10 to 14 percent performance overhead on average. The security components in CypherDB consume about 21K Logic Elements and 54 Block RAMs on the FPGA. The modification of SQLite only consists of 208 lines of code (LOC).

An abstract of exactly 331 words

A Novel Architecture for Secure Database Processing in Cloud Computing

by

Hung-Kwan Chen

B.Eng., The University of Hong Kong, 2011

A thesis submitted in partial fulfilment of the requirements for
the Degree of Joint Doctor of Philosophy
at The University of Hong Kong
and Imperial College London.

October 2016

Declarations

Declaration of Originality

I hereby declare that this thesis represents my own work, except where due acknowledgement is made, and that it has not been previously included in a thesis, dissertation or report submitted to these universities or to any other institution for a degree diploma or other qualifications.

Declaration of Copyright

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work

Dedication

To my dearest parents, brothers and my love.

Acknowledgements

I would like to express my sincerest gratitude to my supervisors, Prof. Paul Cheung, Prof. Peter Cheung and Prof. Ricky Kwok, without whom the work described in this thesis would not have been possible. I still remember the first day I met Prof. Paul Cheung for my final year project during my undergraduate study and to now the completion of my PhD. He has given me substantial support and guidance throughout my university study. I also have to thank him for the introduction of this joint PhD degree with Imperial College London that provided me a chance to meet Prof. Peter Cheung, his twin brother, who is also my primary supervisor at ICL. Prof. Peter Cheung is as supportive and encouraging as Prof. Paul Cheung. I will never forget how Prof. Peter Cheung helped me to write the academic paper by reading it word-by-word and taught me how to write it in a better way. It is my greatest honour to be a student (probably the only student) co-supervised by Prof. Paul Cheung and Prof. Peter Cheung. I would also like to thank Prof. Ricky Kwok who is my secondary supervisor at HKU. I deeply appreciate his attitude, insightful thought and guidance, his sharing and also caring in relation to my study. This thesis could not be completed without his continuous stimulation of my research.

I would like to thank my examiners Prof. Simon Moore, Dr. David Thomas and Prof. Cho-Li Wang for their useful comments and suggestions that helped to improve my thesis.

I also have to thank CAS group at ICL and my student colleagues at HKU. It was definitely a great pleasure to have met a group of talents who worked intelligently, discussed critically and played hard with me throughout my PhD life. Special thanks to Justin Wong, Jungyi Liu, Hilda Xue, Shuanglong Liu, Guangwei Chen, Aryan Tavakkoli, James Davis, Edward Stott, Joshua Levine and James Mardell at ICL for their time spent with me in the laboratory

and at Holland Club/H-bar for pints of beer; and Ho-Cheung Ng, Junqi Deng and Sam Ho for the discussion of various academic or non-academic aspects at HKU. I would like to thank the landlady Gillie Gratham who provided enormous support to my living in London and also Raul, Hassan, Aizar, Ben for chatting, partying and cooking during my life in the UK.

To my friends Gary Chan, Tony Cheng, Leslie Pang, June-Limb Wong, Henry Yip and Jeffrey Wong, I am grateful to have had their support, encouragement, and blessing over all these years.

Last but not least, I would like to thank my beloved parents and brothers for their unconditional love and support during these years which let me pursue my own career path and chase my dream. Most importantly, I am most grateful to my girlfriend, Yoki, for her love and support for my studies and her encouragement to keep me moving forward whenever I was tired and frustrated.

Contents

Declarations	I
Dedication	II
Acknowledgements	III
List of Figures	XI
List of Tables	XVIII
List of Acronyms	XX
1 Introduction	1
1.1 Database-as-a-Service	4
1.2 Threat Model	5
1.3 Research Objectives	6
1.4 Contributions	7
1.5 List of Publications	9
1.6 Organization of the Thesis	9
2 Background	11
2.1 Database-as-a-Service	11
2.1.1 Deployment Model	12
2.1.2 Relational Data Model	12
2.1.3 Database Three Schema Concept	13
2.1.4 Database Query	14

2.1.5	Distributed Database	15
2.2	Processor Architecture	19
2.2.1	Memory Addressing Model	19
2.2.2	ISA architecture	21
2.2.3	Architectural Model	22
2.3	Cryptography: Encryption	24
2.3.1	Symmetric-key Cryptography	24
2.3.2	Public-key Cryptography	25
2.3.3	Digital Signature	26
3	Literature Review	29
3.1	Homomorphic Encryption	30
3.1.1	Fully homomorphic encryption	30
3.1.2	Partially homomorphic encryption	31
3.2	Commercially Available Architecture	32
3.2.1	Secure Co-Processor	33
3.2.2	Trusted Platforms	35
3.3	Academically Proposed Architecture	36
3.3.1	Secure Processor	37
3.3.2	Secure Hypervisor	40
3.3.3	Data-Centric Security	41
3.4	Chapter Summary	42
4	CypherDB System Overview	45
4.1	General Overview	45
4.1.1	Application Scenario	46
4.2	Architectural Model of a Secure Server/Node	49
4.2.1	On-Chip Encryption Key Exchange	49
4.2.2	On-Chip Computations on the Encrypted Data	50
4.3	Attestation	51

4.3.1	Secure Processor Attestation	51
4.3.2	Query Key Attestation	52
4.4	Execution Flow	53
4.5	Database Virtualization	55
5	Proposed Encryption Mechanism	57
5.1	Block-Cipher Encryption	57
5.2	Look-Ahead Encryption Scheme	60
5.2.1	Performance Impact	62
5.2.2	Discussion	63
5.3	Database Encryption	64
5.3.1	Attribute Encryption	64
5.3.2	Attribute Encryption Seed	64
5.3.3	Query Output Protection	66
5.3.4	Index Protection	66
5.3.5	An Illustrative Example	67
5.3.6	Remote Database Modification	68
5.4	Security Analysis	70
6	Processor Architecture	72
6.1	Database Profiling	72
6.2	Architectural Overview	75
6.2.1	Private Key Registers	77
6.2.2	New Security Instructions	78
6.3	Virtual to Physical Memory Mapping	80
6.4	Data Protection	82
6.4.1	DBPage Seed Buffer	82
6.5	Execution Protection	86
6.5.1	Cache Line Encryption	86
6.5.2	SMC Execution Setup	89

6.5.3	SMC Seed Buffer	91
6.6	Query Execution Example	94
6.7	Discussion	96
6.7.1	SMC Allocation	96
6.7.2	Comparison with Alternative Approaches	97
6.7.3	Extension on Speculative OoO multicore processor	99
6.8	Security Analysis	101
7	Implementation	103
7.1	Overview	103
7.1.1	Impact on the Processor Pipeline	105
7.2	Processor Core	107
7.2.1	Encryption Engines	107
7.2.2	Encryption Controller	108
7.2.3	Encryption Pad Shift Module	109
7.2.4	Data Path Switching Module	110
7.2.5	Control Unit	112
7.2.6	Pipeline Stall	112
7.2.7	OpenRISC Core Modification	112
7.3	SMC Encryption Modules	113
7.3.1	SMC Seed Buffer	114
7.3.2	Cache Controller	114
7.3.3	Cache Line Encryption	119
7.3.4	SeedMem Initialization Unit	120
7.4	SQLite Modification	120
7.4.1	Background: Virtual Database Engine	121
7.4.2	Attribute Seed	121
7.4.3	Data Access Layer	124

8	Evaluation	125
8.1	Performance Evaluation: Simulation	125
8.1.1	Simulation Framework	127
8.1.2	Workloads	127
8.1.3	DBPage Buffer Characteristic Study	131
8.1.4	SMC Buffer Characteristic Study	132
8.1.5	Query Output Encryption Sensitivity Study	134
8.1.6	Performance on Data Access Kernel	135
8.1.7	Performance on DBmBench	137
8.1.8	Performance Comparison with a Homomorphically Encrypted Database	139
8.1.9	Performance on TPC-H	140
8.2	Performance Evaluation: FPGA Platform	142
8.2.1	Experimental Setup	142
8.2.2	Look-ahead Encryption Scheme Study	144
8.2.3	SMC Encryption Study	145
8.2.4	Performance Study on SQLite	147
8.2.5	Resource Utilization	157
8.3	Storage Overhead	157
9	Conclusions	160
9.1	Securing DBaaS	160
9.2	Limitations	162
9.3	Future Research	164
9.3.1	Multi-core Architecture	164
9.3.2	Target Application	165
9.3.3	Compiler Support	165
9.3.4	Context Switching	166
9.3.5	Code Injection Attack Protection	168

A CypherDB ISA Extension on OpenRISC Architecture	171
Bibliography	179

List of Figures

2.1	The organization of a two-dimensional table in the relational data model	13
2.2	The database three schema model	14
2.3	The architecture of a distributed database system	17
2.4	An example showing the concept of query decomposition: (a) Query statement for illustration; (b) Relational algebra of the query statement; (c) One of the query decomposition plans in a distributed database system	18
2.5	The DBMS program and run-time allocated memory in virtual memory addressing space. This will be translated into physical memory addressing space during actual execution.	20
2.6	A simplified processor architecture showing the major components which include core, cache memory system and memory management unit	23
2.7	The mechanism of (a) creating a digitally signed message and (b) verifying the received digitally signed message	27
4.1	CypherDB application scenario	46
4.2	CypherDB execution model	47
4.3	Architectural model of each secure server/node. Components in grey represent untrusted entities while those in white represent trusted entities.	49

4.4	A conceptual architecture showing the operation of a secure processor	51
4.5	CypherDB execution flow	53
5.1	The modification of a software program that uses the custom secure instructions to pre-compute and shift the encryption pad in C and Assembly language	61
5.2	A diagram illustrating the performance impacts of decryption of the data <i>residing in the on-chip cache</i> where (a) the look-ahead encryption is not used, (b) the program execution is delayed by the look-ahead encryption mechanism and (c) the look-ahead encryption mechanism can completely hide the encryption latency	62
5.3	A diagram illustrating the performance impacts of decryption of the data <i>residing in off-chip memory</i> where (a) the look-ahead encryption is not used, (b) the delay caused by the look-ahead encryption mechanism is alleviated by the off-chip memory access and (c) the look-ahead encryption mechanism can completely hide the encryption latency	63
5.4	An example showing how an outsourced database record and index are encrypted	67
5.5	Algorithm of remote database modification and attribute seed management	69
6.1	The memory layout of a typical database application process. The database records are formatted in a structure of database pages where the database pages are stored in buffers allocated in heap memory.	73

6.2	A quantitative analysis of total stall time caused by the last-level data cache miss by executing the 22 queries in TPC-H using SQLite in SimpleScalar. The stall time contributed by the three different types of data: attribute data, metadata and execution data, are measured.	75
6.3	Proposed architecture of the CypherDB secure processor. Off-chip memory is partitioned into three compartments. Three data paths are set up to execute the three different types of data. The components in grey indicate additional hardware features.	76
6.4	The virtual to physical memory mapping of the three memory compartments. The DBPage buffer is freely mapped to any available physical memory space while the remaining stack and heap memory is allocated to a contiguous memory block as SMC. A contiguous memory block is also allocated to store the encryption seed as SeedMem.	81
6.5	A flow chart describing the hardware shifter that implicitly shifts the encryption pad for different granularity of <i>l.sload/l.sstore</i> instructions. The <i>l.shift</i> and <i>l.seed</i> can override the operation of hardware shifter to explicitly set the value of <i>Reg_{shift}</i>	83
6.6	A flow chart describing the execution flow of data path 1 and data path 2 with the assistance of a DBPage seed buffer. The DBPage buffer can reduce the frequency of off-chip memory access to fetch the attribute seed.	85
6.7	A flow chart describing the cache line encryption/decryption execution flow of data path 3 with the assistance of a SMC seed buffer	93

6.8	Diagrams demonstrating how the data are executed during a query processing example. The <i>Encrypted Data</i> are loaded along data path 2 and copied to <i>dataBuf</i> . The <i>dataBuf</i> is protected by cache line encryption along data path 3. The data paths used are highlighted in BLUE, the encryption keys used are highlighted in RED and the encryption seeds used are highlighted in GREEN. (a) Execution along data path 1 for non-sensitive data. (b) Execution along data path 2 to load the encrypted attribute data from DBPage memory. (c) Execution along data path 3 where the cache line is encrypted before evicting it from the processor. (d) Execution along data path 2 for query output protection where a query key is used to encrypt the query result	95
7.1	An overview of CypherDB secure processor implementation. The grey components are the additional security modules implemented in our design. A performance counter (shaded) is also implemented to measure the evaluation metrics in hardware.	104
7.2	A diagram showing the impact of our implemented modules on the processor pipeline. The grey components represent our additional security modules for the CypherDB secure processor.	105
7.3	A block diagram of the CypherDB secure processor core implementation	106
7.4	The AES encryption engine employed in our implementation [1]	108
7.5	Our implementation of the encryption controller. It controls the encryption in CTR or OFB mode by switching the data input to the encryption engines between the attribute seed buffer and the encryption pad buffer.	108

7.6	Our implementation of the encryption pad shift module. There are four data paths to write the encryption pad value to the shift registers. These data paths are multiplexed and controlled by the control unit.	111
7.7	A diagram showing the timing error without the data buffer in the load store unit. <i>DATA</i> is the incoming data from the data cache. <i>DAT_BUF</i> is the data buffer implemented to tackle this problem. <i>ENC_STALL</i> asserts when the data are ready but the encryption has not yet finished. <i>REG_WE</i> is the control signal to write <i>DATA</i> into the register.	113
7.8	A circuit to generate the control signal for multiplexing the execution flows in cache controller FSM. Cache line encryption is only executed when these two conditions are both satisfied: 1) the secure execution has been set up via the <i>start_cypherd_b_query</i> instruction and 2) the cache line is within the SMC.	116
7.9	A FSM diagram of the original cache controller from the OpenRISC project. This is used for the execution along data paths 1 and 2.	116
7.10	A FSM diagram of the cache controller for execution along data path 3. Four additional states (in grey) are added in the FSM and the execution is redirected to these additional states. The red arrow represents the re-directed or additional execution flow.	118
7.11	A diagram showing a cache line encrypted with multiple encryption engines in parallel. The S-bit associated with each cache line is used to determine whether the cache line needs to be encrypted.	119
7.12	A stage diagram describing the operation of the SeedMem Initialization Unit	120
7.13	The architecture of SQLite [2]	122

8.1	The DBPage buffer miss rate on increasing buffer block size in our kernel benchmarking with 20% selectivity on (a) sequential scan and (b) non-clustered index scan	131
8.2	The performance evaluation of the sequential scan and non-clustered index scan operators with increasing buffer block size .	132
8.3	The SMC buffer miss rate on executing the three queries in DBmBench with increasing buffer size	133
8.4	The SMC buffer miss rate of (a) uSS, (b) uIDX and (c) uNJ with increasing buffer block size	134
8.5	Performance impact of the query output protection with (a) increasing query output data size and (b) increasing query output entries	136
8.6	Performance evaluation of CypherDB with different SMC buffer size on DBmBench	138
8.7	Performance evaluation of CypherDB with different SMC buffer sizes on the 22 queries in TPC-H	141
8.8	Experimental setup on the FPGA	143
8.9	Total encryption stall cycles with increasing operating frequency of the AES encryption engine	144
8.10	The (a) SMC buffer mis rate and (b) SMC buffer write back rate on executing four bare-metal applications with SMC encryption on the FPGA	146
8.11	Performance evaluation of QUICKSORT on the FPGA	148
8.12	Performance evaluation of MEDIAN on the FPGA	148
8.13	Performance evaluation of KMEANS on the FPGA	149
8.14	Performance evaluation of MAX on the FPGA	149
8.15	Total CPU time spent on processing the DBmBench query benchmark on (a) 100MB, (b) 500MB and (c) 1GB databases	151

8.16	The SMC buffer miss rate of executing the queries in DBmBench over a (a) 100MB and (b) 1GB database on the FPGA	153
8.17	Performance evaluation of DBmBench on the FPGA	154
8.18	Performance evaluation of the nine queries in TPC-H on the FPGA	156

List of Tables

2.1	Description of the format of SQL queries. The basic syntax of the queries is bolded.	15
3.1	Comparison of CypherDB with related effort to secure out-sourced data processing	44
5.1	Custom secure instructions	60
6.1	A summary of the functionality of the four encryption keys used in secure query processing	78
6.2	New Instructions in CypherDB	79
7.1	FSM states involved in the execution along data paths 1 and 2 .	117
7.2	Additional FSM states involved in the execution along data path 3117	
7.3	The VDBE program of the query statement “SELECT * FROM region;”. The bolded number is used as one of the attribute seed components.	123
8.1	Parameters used in simulations	127
8.2	Queries used in the evaluations	128
8.3	The Slowdown Percentage of CypherDB and direct AES with the data access kernel on a sequential scan and non-clustered index scan	137
8.4	Total executed instruction overhead in DBmBench	139

8.5	Slowdown Percentage of CypherDB with a 128kB SMC Buffer and execution over a homomorphically encrypted database on DBmBench	140
8.6	Total executed instruction overhead in TPC-H	141
8.7	The default OpenRISC processor parameters	143
8.8	SMC seed buffer miss rate of the nine queries evaluated in TPC-H	155
8.9	Total resource utilization of the overall system. The displayed number is in the format of (CypherDB/Baseline). # denotes that the value is the same in CypherDB and baseline imple- mentation. * denotes that the value is the total resource con- sumption on the load and store execution path.	158

List of Acronyms

AES Advanced Encryption Standard.

ASIC Application Specific Integrated Circuit.

CIP Confidentiality and Integrity Protection.

CMA Contiguous Memory Allocator.

CPU Central Processing Unit.

CSP Cloud Service Provider.

CTR Counter-Mode.

DBaaS Database-as-a-Service.

DBMS Database Management System.

DBPage Database Page Compartment.

DIFT Dynamic Information Flow Track.

DRAM Dynamic Random Access Memory.

DRM Digital Right Management.

DRNG Digital Random Number Generator.

FIPS Federal Information Processing Standards.

FPGA Field Programmable Gate Array.

FSM Finite State Machine.

IFT Information Flow Tracking.

JTAG Joint Test Action Group.

KEM Key Encapsulation Mechanism.

LOC Line Of Code.

LRU Least Recently Used.

LSU Load Store Unit.

NIST National Institute of Standards and Technology.

NoSQL Non SQL.

OFB output feedback mode.

OLAP Online Analytical Processing.

OLTP Online Transaction Processing.

OPE Order Preserving Encryption.

OS Operating System.

OTA Online Trust Alliance.

OTP One-Time Pad.

PCI Peripheral Component Interconnect.

PKI Public Key Infrastructure.

PUF Physical Unclonable Function.

RDS Relational Database Service.

SDRAM Synchronous Dynamic Random-Access Memory.

SeedMem Seed Memory Compartment.

SMC Secure Memory Compartment.

SP Secret-Protected.

SQL Structured Query Language.

TCPA Trusted Computing Platform Alliance.

TDE Transparent Data Encryption.

TLB Translation Lookaside Buffer.

TPM Trusted Platform Module.

TSM Trusted Software Module.

TXT Intel Trusted Execution Technology.

VDBE Virtual Database Engine.

VM Virtual Machine.

XOM Execute-Only Memory.

XOMOS XOM Specific Operating System.

XOR Exclusive-OR.

Chapter 1

Introduction

Cloud computing has become a popular computing model for offering large-scale data storage and intensive computations of large amounts of data in a highly distributed manner. The effect of the economies of scale and the extensive use of parallel computing in the cloud results in a significant speed-up of the outsourced computation at a very low cost, as compared to the investment of deploying the same in-house computing infrastructure.

With the aforementioned advantages of Cloud Computing, it is attractive to run a database on a cloud computing platform. In fact, there are already numerous notable database vendors offering cloud database services. The adoption rate is high and is expected to rise. A survey from Tesora [3] shows that 68 percent of enterprises are using database technology in the cloud. Another survey from 451 Research [4] also indicates an expected increase in the overall adoption of cloud database services.

However, as indicated in these two surveys [3, 4], one of the biggest barriers to the adoption of cloud database services is the security concern, which ranked highest with 60 percent of the respondents in [3]. A global security study [5] also indicates that more than half of the participants rated data security as their major consideration of storing their data in a cloud. In [6], over 70 percent of the respondents rated security as their most important concern. Most importantly, these security concerns are realistic. The Online Trust

Alliance (OTA) [7] reported that over 740 million records were exposed in 2013. A security breach in Google [8] shows that any server administrator with access to the confidential information can abuse it at ease. An even worse issue is that any honest-but-curious server administrator is capable of performing massive breaches of privacy without being detected.

Assuring data security in Cloud Computing is thus of utmost importance for rapid adoption of cloud database services. In a broad sense, data security includes confidentiality, integrity and availability of the data itself. Confidentiality is the ability to hide the content of the data from unauthorized parties. Integrity is to protect the data from being modified by unauthorized parties. Availability is to ensure the accessibility of the data to authorized parties when needed. Our research focus is on protecting the data confidentiality.

One obvious solution to protect data confidentiality is by encryption. For example, Amazon Relational Database Service (RDS) [9] provides encryption services to encrypt the storage data (data-at-rest encryption). Similarly, Microsoft Azure SQL Database uses Transparent Data Encryption (TDE) [10] to encrypt database files at rest. However, current security solutions employed in Cloud Service Provider (CSP) are still insufficient due to the following reasons:

- The decryption keys are stored in the cloud server. Although a multitude of approaches are used to protect the decryption keys at cloud servers through software protection or a compact off-chip hardware security module [11]), it turns out that attackers can eventually break in using sophisticated software attacks [12, 13] or hardware attacks [14, 15].
- The data are only encrypted at rest where only the data in storage are protected. The encrypted data have to be decrypted and stored in the server memory before any query processing. In some cases, the hackers who manage to get administrative access to the servers [16] or the curious administrators [17] can gain access to all the unprotected data or even the decryption keys.

To tackle these security issues, a protection mechanism to secure the decryption key and protect the confidentiality of decrypted data in memory is needed.

There have been many proposed solutions. One interesting solution is to use homomorphic encryption [18–21], which permits computation on encrypted data without receiving the decryption key. As the decryption key is not disclosed and the encrypted data is never decrypted, it can successfully secure the data in the Cloud. However, these approaches are either prohibitively slow [18, 19] or highly restrictive in arithmetic operations [20, 21]. Alternatively, the decryption key and the decrypted data can still be stored in the cloud server but within a self-contained and tamper-resistant co-processor [22, 23]. Such an approach requires an extra processing system which usually has low computational power and is expensive. Lastly, the computer architecture community has proposed a secure processor approach [24–28] which stores the decryption key in the main processor and provides a mechanism to protect the data in off-chip memory. Nevertheless, these approaches initially target Digital Right Management (DRM) and portable devices which protect the application program and data as a whole. This highly prohibits dynamic data movement and parallel processing in the cloud computing environment. How to make use of a secure processor to protect data confidentially in the cloud computing environment, especially with regard to securing cloud database services, is still largely unknown and remains a significant challenge.

To be applicable to the practical cloud computing environment, the protection mechanism needs to address a number of challenging issues. In the first place, this protection mechanism should be able to secure the decryption key and protect the confidentiality of the decrypted data. It is equally important that this mechanism has to be economical and efficient to be of practical use. Trading security for performance [29] would be of less practical use because it leaves the users a burden to manage their own risks. Last but not least, the

protection mechanism has to preserve the characteristics of cloud computing so as to make cloud database services beneficial, as compared to traditional local database server approaches.

This thesis proposes a novel cloud system architecture called CypherDB [30, 31], which takes one step further to investigate the possibility and, most importantly, practicability of using a secure processor to protect cloud database services. We develop a cost-effective protection mechanism that can secure an outsourced database being executed in a cloud computing environment with high performance. We also extend the role of secure processor to secure cloud database services through a novel encryption scheme and processor architectural re-design.

1.1 Database-as-a-Service

To provide Database-as-a-Service, a cloud service provider not only provides hardware resources to store the database and host the application software but also the Database Management System (DBMS) to provide query services and data management to the database owner. The database owner is able to remotely create a database in the cloud and query, update or delete it later. These operations can be done via simple data exchange or query statements through some standard web consoles so that the database owner can access her database anytime and anywhere with an Internet connection. In the cloud, the database can be partitioned and stored in geographically separate data centres. There are also multiple computing nodes that are able to access the distributed database and are designed for providing query services. Although a complex architecture and operations are required to query the distributed database, this is completely hidden and abstracted from the database owner. In other words, the database owner has no knowledge of how her database is stored and what operation is performed on the database. The database owner can query the database as if it is stored on a single server.

Most cloud database workloads can be categorized into online transaction processing (OLTP) or online analytical processing (OLAP). OLTP is characterized by a large number of update-intensive online transactions (insert, delete or update records in databases) that are typically used to facilitate and manage usual business applications. The queries involved are usually short and simple. The effectiveness of an OLTP is measured by the number of transactions per second. OLAP is, on the contrary, characterized by a relatively low volume of transactions but a large number of read-intensive operations that are usually used to retrieve data for data mining and decision support applications. The queries involved are often very complex and involve aggregation. OLAP's effectiveness is measured by response time. In this thesis, we are particularly interested to investigate our proposed secure processor approach towards OLAP workload that is currently poorly supported by some existing solutions using homomorphic encryption.

1.2 Threat Model

We assume the CSP in our model is honest-but-curious [32]. In general, it obeys any communication protocol agreed with the database owner and deploys database operations correctly. Any detectable tampering can be protected by a contractual agreement between them. However, this cannot protect any passive attack leaving no trace on the system (i.e. read/copy data from storage disk, main memory or probe processor-to-memory data bus). The goal of an adversary is to read the content of the database without being detected. In summary, we consider the following threats and assumptions:

- An adversary can launch a malicious process or Virtual Machine (VM) or even get access to the Operating System (OS) layer to copy or read the data in off-chip memory. Simple memory bus probing to read the data in processor-to-memory traffic is also possible.

- The communication channels between the database owner and secure server or among secure servers and secure nodes are all open and subject to eavesdropping.
- The DBMS is assumed to be securely booted by employing secure boot technology (e.g. Bastion [33], TPM [11]). Run-time Execution Validator [34] can also be employed to ensure that the DBMS is running as expected. Code injection attack is therefore not considered in this thesis and is assumed that this attack can be tackled by the aforementioned techniques.
- Side-channel attacks such as timing-attack or power-analysis attack are not considered in this paper because these attacks are prohibitively costly to implement in a typical data centre environment. Indeed, to launch these attacks, the attacker needs to breach the physical security of the data centre and this is highly unlikely to happen. Any hardware tampering on the processor is also assumed to be infeasible in the cloud environment.

1.3 Research Objectives

In the design of our protection mechanism for securing cloud database service, we address four fundamental aspects: security, performance, cost and deployment in the cloud. These four aspects are elaborated below.

Security. We have designed a robust protection mechanism to secure the decryption key and decrypted data during query processing in cloud database services. In our system, the decryption key is never disclosed in any circumstance and used data are always protected by encryption in system memory so that the confidentiality of the data is always protected from any adversary with administrative access. Our proposed system not only protects the database data, but also any intermediate data generated on-the-fly during

query processing.

Performance. The protection mechanism is of high performance. By high performance we mean the overhead on the query execution time is minimal. The overhead mainly comes from encrypting and decrypting the data for security purposes. Our design aims to reduce this overhead using architectural techniques. Our design objective is to have a performance close to optimal. By optimal performance we mean to have the query execution time close to that without any data protection.

Cost. The protection mechanism has to be cost-effective to be applicable in practice. By cost-effective we mean the extra hardware/software resources and memory storage are minimal and reasonable while maintaining high-performance query execution in cloud database services. Our proposed solution aims to optimize the architectural design for effective query execution over encrypted data at minimal cost.

Deployment in the cloud. We have designed a protection mechanism that can suit the cloud computing environment. In particular, the encrypted data can move dynamically in the cloud for parallel processing. Our design objective is to maintain the five essential characteristics [35] of Cloud Computing according to the National Institute of Standards and Technology (NIST) definition. The five characteristics are on-demand self-service, broad network access, resource pooling, rapid elasticity and measured service.

1.4 Contributions

To the best of our knowledge, CypherDB is the first to use a processor architectural design to successfully protect remote operation on encrypted databases against any honest-but-curious administrator. Our main contributions are summarized as follows:

- *System Architecture:* We have developed a framework that can use a

secure processor in cloud database services. A protocol is designed for the cloud database users to request a secure query execution in our proposed system and architecture. The impact of this architecture toward virtualization is also qualitatively discussed.

- *Database Encryption Scheme*: We have developed a novel encryption scheme to protect databases which can be executed under our secure processor with high performance. The proposed encryption scheme requires custom instruction support and minor modification of the database application program. The encryption scheme is validated through an integration to SQLite¹ and is proven to be robust through extensive simulation and execution on a Field Programmable Gate Array (FPGA) platform.
- *Secure processor architecture*: We have developed a secure processor architecture to 1) provide architectural support for our proposed database encryption scheme, and 2) provide an efficient protection mechanism, based on cache line encryption [36], to protect all intermediate data generated on-the-fly during query execution. The efficiency, robustness and the cost of our novel processor architecture are validated and evaluated through extensive simulation and implementation on the FPGA.
- *FPGA implementation*²: We have implemented our proposed secure processor on a FPGA platform. There are three key advantages of our FPGA implementation. Firstly, the implementation forces the architecture design to be complete. It further enables us to discuss some practical issues of the secure processor implementation. Secondly, it can evaluate the cost of the design by measuring the resource consumption in the FPGA. Lastly, it enables us to evaluate our proposed solution over a large database on a FPGA platform, which is prohibitively slow to run in sim-

¹The modified SQLite program, written in C, is available at <https://github.com/hkbonychen/CypherDB-SQLite.git>

²The FPGA implementation, written in Verilog, is available at <https://github.com/hkbonychen/CypherDB-FPGA.git>

ulations. This thesis provides an accurate and comprehensive evaluation of our proposed encryption scheme and secure processor architecture.

1.5 List of Publications

- B. H. K. Chen, P. Y. S. Cheung, P. Y. K. Cheung, and Y. K. Kwok, “CypherdB: A novel architecture for outsourcing secure database processing,” *IEEE Transactions on Cloud Computing*, 2016, accepted for publication and to appear.
- B. H. K. Chen, P. Y. S. Cheung, P. Y. K. Cheung, and Y. Kwok, “An efficient architecture for zero overhead data en-/decryption using reconfigurable cryptographic engine,” in *Proceeding of 2015 International Conference on Field Programmable Technology*, 2015, pp. 248–251.

1.6 Organization of the Thesis

The organization of this thesis is as follows. In Chapter 2, we discuss some background information that is related to our work and assists readers to better understand it. In Chapter 3, we provide a literature review of earlier research work and existing solutions that are relevant to our work. We also provide a detailed comparison of our proposed solution with these prior work. In Chapter 4, we present an overview of CypherDB cloud system architecture and discuss the framework for deploying secure processors in cloud database services. In Chapter 5, we present our novel look-ahead encryption scheme and qualitatively discuss its security and performance impact. In Chapter 6, we present our secure processor architecture. We first present a scrutiny of the processor execution profile of a database application. This scrutiny motivates our processor architectural design. We then present the architectural design to support our look-ahead encryption scheme and the protection of intermediate data generated during query processing. In Chapter 7, we first present

our implementation of our secure processor on the FPGA and then describe our modification on SQLite to support our look-ahead encryption scheme. In Chapter 8, we present our evaluation of our proposed scheme and secure processor design in simulations and execution on a FPGA platform. In Chapter 9, we conclude the thesis and discuss some future research directions.

Chapter 2

Background

This chapter presents some basic background knowledge on three aspects: Database as-a-Service, Processor architecture and Cryptography. These three aspects are individually a huge topic to be described in detail such that we may not be able to cover them all in this thesis. Instead, this chapter intends to discuss some of this background information that can facilitate the understanding of readers in regard to the work reported in this thesis. We first discuss the model and architecture of cloud database services. We then introduce the processor architecture on which the database management system is running. Finally, we describe the cryptography that is employed in this thesis.

2.1 Database-as-a-Service

To provide DBaaS, a DBMS has to be installed into the cloud server to manage the database. We first discuss the deployment model of the DBMS in this section. We will then go through the database system from data model, schema design and database query to the architecture of distributed databases, database fragmentation and query decomposition in cloud database models.

2.1.1 Deployment Model

There are numerous notable database vendors offering cloud database services, which is mainly in one of these two deployment models: VM image or Database-as-a-Service (DBaaS). In the first deployment approach, the DBMS is installed and run on a VM. Users can upload the VM image (with a DBMS installed) to the cloud. The CSP is responsible for the infrastructure support while the users have to perform database management and administration. In the DBaaS approach, the DBMS is installed on the host computer in the cloud. The users only need to request and pay for the service while the CSP is responsible for all the management and administration work. While each deployment model has its own benefits and drawbacks, it is shown in [37] that consolidation in logical databases is much more cost-effective and performance efficient than the “DB-in-a-VM” approach. Recently, the Oracle pluggable database [38] also adopted logical database consolidation to provide DBaaS. Our research focus is thus on the DBaaS deployment model.

2.1.2 Relational Data Model

Cloud databases can be stored in one of these two data models: Structured Query Language (SQL) databases or Non SQL (NoSQL) databases. SQL databases organize the data in its relational model, where the data are stored in columns and rows. On the contrary, NoSQL databases organize the data in non-relational model. Although NoSQL can scale better in a cloud environment, it is still disputable as to whether SQL or NoSQL databases are more suitable and efficient for cloud computing [39]. However, SQL databases are still the most commonly used data model nowadays [40]. Our research therefore targets on SQL database rather than NoSQL databases.

The relational model is the most common data organization in SQL databases. It organizes data into collections of two-dimensional tables as shown in Figure 2.1. Attribute refers to a column while tuple/record refers to a row in a

table. Attribute data are a cell/entry in a specific column and row. A database usually contains a collection of tables.

Figure 2.1: The organization of a two-dimensional table in the relational data model

A relational database is designed to have a profound effect on how easy the data can be accessed and managed. To connect multiple tables together, each record has a unique identifier that serves two major purposes: 1) it facilitates the process of locating a correct record upon request and 2) links records in a table to records in other tables which can obtain and integrate information from multiple tables. Relationships are a logical connection between different tables. Depending on the design strategy, connection of tables can be in the form of either a one-to-one, or a one-to-many relationship.

2.1.3 Database Three Schema Concept

The aim of three schema concepts is to separate each user's view of a database from the way it is physically represented by abstracting the underlying layer with a new schema on top. Starting from the bottom layer as depicted in Figure 2.2, physical schema pertains to the actual storage of data and its form of storage like files, indices, etc. It presents the storage location of an attribute in a memory page. Logical schema defines the structure of a table and relationships in the database. External schema is simply the user's view of

the database. In relational databases, it is common that the external schema is the same as the logical schema.

Figure 2.2: The database three schema model

2.1.4 Database Query

To extract data from the outsourced database, the database owner can send a query to the cloud for data lookup or modification. A query is simply a statement requesting either retrieval or modification of information. In database languages, data definition language (DDL) is used to specify a database schema and data manipulation language (DML) enables the database owner to access or manipulate data. Among the various query languages, Structured Query Language (SQL) is the dominating well-known standard for expressing language. We will therefore describe six different kinds of queries (CREATE TABLE, DROP, SELECT, INSERT, UPDATE, DELETE) in SQL format.

CREATE TABLE and DROP statement is DDL language that is used to establish a new table or abandon an existing table. To create a table, the statement has to describe the scheme of the table – the set of column name

and data type. Dropping a table is easier by specifying the table name only. SELECT, INSERT, UPDATE and DELETE statements are DML language that requires data manipulation. INSERT and DELETE statements are used to add or remove records from a table. UPDATE statement is to modify attribute data from a record. SELECT statement is able to retrieve selected data that match the criteria specified in the statement. A brief description of the queries and the corresponding formats are shown in Table 2.1. Readers are also referred to [41] for the detailed syntax and usage of the SQL statements.

Table 2.1: Description of the format of SQL queries. The basic syntax of the queries is bolded.

Query	Query format and description
CREATE	CREATE TABLE [table name] ([column definitions])
DROP	DROP TABLE [table name]
SELECT	SELECT [column names] FROM [table name] WHERE [conditions] GROUP BY [column name] ORDER BY [column name]
INSERT	INSERT INTO [table name] VALUE ([values])
UPDATE	UPDATE [table name] SET [VALUES] WHERE [conditions]
DELETE	DELETE FROM [table name] WHERE [conditions]

2.1.5 Distributed Database

As the database grows huge, it is impossible to store the entire database in one single storage device due to hardware constraints and it is also inefficient to use one single server to query a huge database. A distributed database system allows a database to be partitioned/fragmented into multiple databases of smaller size. Each of these small databases is connected to an individual computing node so that the database is managed in a distributed manner. By storing the partitioned database in multiple nodes, it not only relieves the hardware burden of having an extremely large storage device and powerful server but also makes it more efficient because the partitioned database is

now much smaller to be processed and such processing can be massively performed in parallel. These advantages however come at the expense of a more complicated system architecture and database synchronization.

Architecture

In a distributed database system, all application servers and storage nodes are connected to each other as shown in Figure 2.3. The application server possesses a global view of all the partitioned data belonging to the database owner. When the database owner talks to the application server using SQL language, the application server can draft an optimal solution to decompose the query into sub-queries and distribute the sub-queries workload to local storage nodes to which the related databases are connected. Each storage node has a local DBMS installed to perform the sub-query execution. The result of each sub-query execution in each storage node is then sent to the corresponding application server to combine and join the data together in order to produce the final query output to the database owner.

Database Fragmentation

A table can be fragmented into several tables so that the original table can be reconstructed from the fragments. The fragments, as stated above, are scattered and stored in different geographically separated storage nodes. There are basically two schemes of fragmentation: 1) horizontal fragmentation that splits the table by assigning records to different fragments and 2) vertical fragmentation that divides the table by splitting the columns. Because vertical fragmentation requires a more complicated strategy to split and combine the fragments afterwards, we consider horizontal fragmentation as our case study in this thesis.

Figure 2.3: The architecture of a distributed database system

Query Decomposition

Query decomposition and rewriting are necessary when the application server needs to distribute the query workload to local storage nodes. Because it remains a huge topic in database systems, the detailed theory and technique will not be discussed in this thesis. Instead, we use a simple example to describe query decomposition and rewriting in distributed databases as shown in Figure 2.4.

Figure 2.4: An example showing the concept of query decomposition: (a) Query statement for illustration; (b) Relational algebra of the query statement; (c) One of the query decomposition plans in a distributed database system

The aim of query decomposition is to convert a high-level query into a relational algebraic query so as to rewrite the query and find an optimized query plan. Consider a SELECT query as shown in Figure 2.4(a). The query can be decomposed into relational algebra as shown in Figure 2.4(b). This includes a join operator to collect the set of records in S and E that are equal on the common attribute name “number”, a select operator to filter the records with the attribute “hours” larger than 10, and a project operator to unify all attribute

“name” from the filtered records. Figure 2.4(c) shows the corresponding query plan in a distributed database. Assume database S and E are partitioned into S1, S2 and E1, E2 respectively. The select operation is first pushed to the storage nodes having S1 and S2 to filter the record that has attribute “hour” larger than 10. Then the filtered records are sent to other storage nodes containing E1 and E2 for further join operation. After the join operation, the resulting filtered records are then unified for project operation.

2.2 Processor Architecture

The DBMS of each computing node in a cloud database system has to be executed on the underlying hardware. The first step to understand how this works is to realize the memory addressing model which outlines the method of converting a piece of software into a memory model that the processor can execute. We will then describe the Instruction Set Architecture (ISA) that is an interface of the processor and the memory model so that the software program converted in this memory model can be executed in the processor. Finally, we will describe the architectural model of the processor which serves as the fundamental hardware system that performs the actual execution.

2.2.1 Memory Addressing Model

Although a DBMS consists of a number of complicated software layers, these software layers are eventually compiled into a fragment of machine codes and some static data to be executed on a computing platform. These machine codes are stored in memory, which can be presented in its virtual addressing space as shown in Figure 2.5. Apart from these codes and data, the remaining memory space will be statically allocated as a stack or dynamically allocated as a heap to be used during program execution. During execution, database data are most likely to be stored in heap memory and will be accessed in

accordance with the program execution.

Figure 2.5: The DBMS program and run-time allocated memory in virtual memory addressing space. This will be translated into physical memory addressing space during actual execution.

This virtual addressing space is provided by the operating system so that the actual physical memory space can be abstracted from the programmer. Eventually, the program and data have to be stored in the physical memory space that is the actual physical memory location in the off-chip memory. A page table, managed by the operating system, is thus used to store the virtual-to-physical memory address translation. During the execution, the program executes the code according to the virtual memory addressing space while this will be translated to the physical memory addressing space by the operating system in run-time.

2.2.2 ISA architecture

The compiled program (ready-to-be-executed code) stored in the main memory is in the form of machine codes that consist of a sequence of commands. These commands are known as instructions, which are the basic elements to communicate with a processor and act as an interface between a computer's software and hardware. An instruction consists of an op-code and operands. The structure and format of the op-code and operands varies among different processor architectures. The format and organization of an instruction is referred to ISA. There are mainly two types of ISA: 1) complex instruction set architecture (CISC) and 2) reduced instruction set architecture (RISC). CISC involves many specialized instructions, some of which may only be rarely used in practical programs. Its structure is usually more complex than RISC. On the other hand, instruction of RISC has a much simpler structure. The idea of RISC is to optimize the processor architecture for processing some frequently used instructions, while the less common operations are implemented as sub-routines. Because RISC is simpler and is used throughout our implementation, we discuss the detail of RISC instruction structure and format. In particular, ISA of MIPS architecture is used as an illustrative example.

RISC instruction consists of two components, namely op-code and operands. An op-code specifies the operation to be performed while operand specifies the data that the op-code will process. MIPS instructions fall into three categories: R-type, I-type, and J-type.

R-type instructions refer to register-type instructions where the format and semantic is shown as Equation 2.1. The processor performs operations as specified in op-code on the value stored in \$rs and \$rt and stores the operation result in \$rd.

$$opcode \quad \$rd, \$rs, \$rt \quad (2.1)$$

I-type instructions refer to immediate-type instructions where the format and semantic is shown as Equation 2.2. The processor performs operations as

specified in op-code according to the value stored in \$rs and the value indicated in immed and stores the operation result in \$rt.

$$opcode \quad \$rd, \$rs, \$rt \quad (2.2)$$

J-type instructions refer to jump-type instructions where the format and semantic is shown as Equation 2.3. PC is the program counter that stores the current address of the instruction being executed. Jump-type instructions update the PC with the value specified in the target.

$$opcode \quad target \quad (2.3)$$

2.2.3 Architectural Model

A processor consists of three major sub-systems, including processor core, cache memory system and memory management unit. A simplified processor architecture is shown in Figure 2.6.

The processor core is responsible for executing the instructions. It involves a 5-stage pipeline to fetch the instruction from memory (fetch stage), decode the instruction to figure out the operation and data involved (decode stage), perform arithmetic operations on the data (execution stage), load/store the data from/to the memory (memory stage) and write back the result of the arithmetic operation to the destination register (write-back stage).

The cache memory is used to reduce the average cost of loading the data from off-chip memory to the processor core for processing. It is a small and fast on-chip memory which stores frequently used data from off-chip memory. Most processors have multiple levels of cache that form a hierarchy of cache levels. The first-level cache is usually split into separate data and instruction cache while other higher levels consist of a larger and unified cache. There are two different cache architectures, namely write-through and write-back

Figure 2.6: A simplified processor architecture showing the major components which include core, cache memory system and memory management unit

cache. In write-through cache, every write (store) to the cache causes a write to the off-chip memory. In write-back cache, a write (store) to the cache is not immediately written to the off-chip memory. Instead, the cache containing the data being written over is marked as dirty. The dirty cache line is written back to the off-chip memory only when it has to be evicted from the processor.

The memory management unit is responsible for the translation of virtual memory address to physical address. It consists of a small and fast on-chip memory but it is, unlike cache memory which is used to store data, used to store the virtual-to-physical memory translation table and is called translation look-aside buffer (TLB). Similar to the idea of cache memory, it is used to store the virtual-to-physical memory page table on-chip so that the translation can be performed instantly without spending time on accessing the page table entries in off-chip memory.

2.3 Cryptography: Encryption

Cryptography is used to construct a protocol that can prevent third parties or the public from tampering with private messages. An adversary can tamper with a private message through various methods such as eavesdropping the communication channel, modifying the messages, etc. In this thesis, we will focus on using encryption to protect the confidentiality of data as discussed in Chapter 1. There are two major encryption methods: symmetric-key cryptography and asymmetric-key cryptography, each of which has its own advantages and disadvantages. We will show a practical use of combining these two encryption methods for Key Encapsulation Mechanism (KEM) and describe a mechanism of using digital signatures to authenticate and protect the integrity of a message.

2.3.1 Symmetric-key Cryptography

Symmetric-key cryptography uses the same cryptographic keys (K) for both encryption of plaintext¹ (D) and decryption of ciphertext² (C). The symmetric-key encryption function (Enc) takes the cryptographic keys and plaintext (or ciphertext) as inputs to compute the ciphertext (or plaintext) as the output. This can be seen from the following equations:

$$Encryption : C = Enc(D, K) \quad (2.4)$$

$$Decryption : D = Enc(C, K) \quad (2.5)$$

There are two categories of encryption algorithm, namely stream ciphers and block ciphers. Stream ciphers (e.g. one-time-pad) are based on generating an “infinite” cryptographic keystream (K) to encrypt the data (D) one bit at a time. There is a strict requirement of the keystream that it has to be

¹Plaintext refers to original data without encryption.

²Ciphertext refers to the data after encryption.

random and never be reused. This usually requires the length of keystream to be almost identical to the length of the data to be encrypted. Block ciphers (e.g. AES, DES, Blowfish) work on larger chunks of data (i.e. blocks) at a time. The encryption function usually takes a block of data and applies the same cryptographic key (unlike stream cipher that requires an “infinite” long keystream) to encrypt the data. The encryption function itself involves key-dependent permutation of values and is composed of different chaining modes of operation that makes it secure even though the same cryptographic key is used on different blocks of data.

The cryptographic keys (K) represent a shared secret between two or more parties. They must be identical to encrypt and decrypt the plaintext and ciphertext so that the same value can be retrieved from the encryption algorithm. This secret key also represents a critical secret in any symmetric-key encryption which means a private and secure channel has to be set up to exchange this shared secret key for symmetric-key cryptography. Although there are few block cipher approaches that are proven to be fast and efficient in encryption and decryption, the requirement that both parties have to share the same secret key is one of the main disadvantages of symmetric-key cryptography.

2.3.2 Public-key Cryptography

Public-key cryptography (a.k.a. asymmetric cryptography) is designed to overcome the weakness of symmetric cryptography because the secret key does not need to be shared in this cryptographic system. It uses a pair of keys: public key (PK) that is not a critical secret and can be disseminated widely and a private key (SK) which is a critical secret known only to the owner. To encrypt a message (D), any person can encrypt it using the public key of the receiver using an appropriate public encryption function (E_{PK}). RSA and ElGamal are some typical algorithms of Public-key cryptography. This encrypted message (or ciphertext) (C) can only be decrypted with the receiver’s private key

(SK) using the same encryption function but with SK as the cryptographic key (E_{SK}). This can be seen from the following equations:

$$\text{Encryption} : C = E_{PK}(D) \quad (2.6)$$

$$\text{Decryption} : D = E_{SK}(C) \quad (2.7)$$

The security of the public-key cryptographic system relies on a mathematical problem that there is currently no efficient solution to solve without knowing the private key. This mathematical problem usually involves integer factorization, discrete logarithm and elliptic curve relationships which requires complex computation. It is therefore computationally inefficient and is usually used only for small blocks of data. One practical use of public key cryptography is key encapsulation. Key encapsulation is an encryption technique designed to make hybrid use of symmetric and asymmetric encryption so as to take advantage of the fast encryption of symmetric cryptography and without the need to set up a private and secure channel for secret key exchange in asymmetric cryptography. In general, the data are encrypted using symmetric cryptography while its secret key is further encrypted using asymmetric encryption. In other words, the private and secure channel is setup via asymmetric cryptography.

2.3.3 Digital Signature

Public key cryptography is not only used to encrypt the data to protect confidentiality, but also used to sign digital signatures ($Sig(M)$) to authenticate a digital message (M). Authentication can prevent man-in-the-middle attack where an adversary uses a forged message to replace the original one. To sign a digital signature, the user uses the private key (SK) to encrypt the message (M) so as to generate a digital certificate. Any person can use the corresponding public key (PK) to decrypt the signature and verify that the signature ($Sig(M)$) is indeed from the one with the proper private key.

Digital signature is often used in conjunction with a cryptographic hash function to protect the integrity of the message. A cryptographic hash function is a one-way function that maps data of arbitrary size to a fixed size bit string. It is infeasible to find two different messages with the same hash value and also computationally infeasible to generate the input message from its hash value. These properties favour the protection of the integrity of the message. As a result, a typical way to authenticate a message is first hashing the message with a cryptographic hash function, then generating a digital signature of the hashed value using public key cryptography. It is as illustrated in Figure 2.7(a). To create a digitally signed message, the message (M) is first hashed using a cryptographic hash function to generate a bit string. The hashed value ($Hash(M)$) will then be encrypted using the private key ($E_{SK}(hash(M))$) to create a signature. By combining the original message (M) and the digital signature, a digitally signed message ($Sig(M)$) can be generated.

Figure 2.7: The mechanism of (a) creating a digitally signed message and (b) verifying the received digitally signed message

To verify the digitally signed message, the verifier reverses the procedure of creating $Sig(M)$. On one hand, the message (M') contained in $Sig(M)$ is hashed

and waits for further verification. On the other hand, the signature contained in $Sig(M)$ is decrypted using the corresponding public key ($E_{PK}(signature)$) and the hashed value ($Hash(M)$) obtained is computed during the creation of $Sig(M)$. By comparing these two hashed values, the verifier can ensure the integrity of the message.

Under this protection mechanism, an adversary is unable to forge the message (M) because of the cryptographically hashed value. An adversary can also forge the message and also hashed value but it is also forbidden with the use of a digital signature because the signature cannot be forged without knowing the private key SK .

Chapter 3

Literature Review

As discussed in Chapter 1, the goal of our research is to protect the confidentiality of outsourced data against any honest-but-curious adversary. In this chapter we survey some earlier research work and existing solutions that are relevant to our work. This chapter covers three separate approaches taken in the relevant prior works.

The first approach is to use homomorphic encryption [18, 20, 21, 42–44] throughout the system. As homomorphic encryption allows computation to be carried out directly on the ciphertext without prior decryption, this approach provides a pure software solution to secure the outsourced data processing against the curious-but-honest adversary model. This solution can be processed on existing available hardware without any modification or addition.

The use of commodity hardware security features [11, 22, 23, 45, 46] to provide enhanced data security is then considered. Such an approach potentially offers stronger security and higher performance when compared to the software-based approach. At the same time, it employs existing hardware security features and minimizes or even avoids hardware modifications to existing commodity systems. Potentially it offers an economical solution to the data security problem. Nevertheless, whether such commodity systems are able to provide sufficient security support to guard against attacks on outsourced data is investigated and examined.

Lastly the use of dedicated novel hardware architectures [24–27, 33, 47–51] to improve the security of a general-purpose computing platform is considered. These solutions move the root of trust to the processor chip and modify the processor architecture in order to better secure the data and the system as a whole. Our solution mainly focuses on architectural support towards secure data outsourcing. Although some of these techniques do not directly address the problem of securing outsourced data processing, our proposed solution can leverage and build upon these techniques.

3.1 Homomorphic Encryption

The advantage of homomorphic encryption is that it allows computations to be performed directly on the ciphertext. As a result, the encrypted data can be processed without any information leakage. The result from such computation is a new ciphertext, which can be later decrypted (with a proper decryption key) by the user to recover the result in plaintext. In general, such encryption technique can be categorized into either fully or partially homomorphic encryption.

3.1.1 Fully homomorphic encryption

Fully homomorphic encryption [18] can evaluate an arbitrary number of additions and multiplications (and thus any operation) on encrypted data. It is an excellent way to protect the confidentiality of outsourced data in Cloud Computing. The data owner can encrypt their sensitive data and send them out to the cloud. The encrypted data can be manipulated on the remote machine without any information leakage to the cloud administrator or a third-party. The result can then be sent back to the data owner, who recovers the results through the use of the secret decryption key. Because the data are never decrypted in the cloud, its privacy is fully protected. However, fully

homomorphic encryption suffers from a serious problem: the computation is prohibitively slow [52], which makes it impractical to use. Subsequent research attempts to use fully homomorphic encryption for secure data processing [19] also incurs unacceptable computational overheads.

3.1.2 Partially homomorphic encryption

Due to the impracticality of fully homomorphic encryption, partially homomorphic encryption is used as an alternative to secure outsourced data without compromising much on performance. Although partially homomorphic encryption is much more efficient in terms of computation, it can only support limited operations (either addition or multiplication but not both). For example, Paillier encryption [42] can only support addition of actual value by multiplying the encrypted data. ElGamal encryption [43] is multiplicatively homomorphic so that multiplication on ciphertext is equivalent to that on plaintext. In some cases, security is traded for efficiency. For instance, Order Preserving Encryption (OPE) [44] preserves the order of plaintext after encryption. It can strongly facilitate order comparison operation but it leaks the order in the first place.

Because there are limited operations supported by partially homomorphic encryption, a natural solution is to leverage its best computational capability to extract a subset of encrypted data and then offload it to the data owner for the remaining computation. Prior work [53] proposed to partition the encrypted database into buckets and augment them with indices. Querying the outsourced database results in extracting buckets of encrypted data to the data owner, of which indices match with the query. Ge *et al.* [54] demonstrated the use of Paillier encryption to support summation and average operation over an encrypted database. Agrawal *et al.* [55] proposed to use OPE to build database indices and to perform range queries. These solutions all rely heavily on the data owner to complete the tasks that cannot be done in the cloud.

There are also hybrid approaches which make use of more than one partially homomorphic encryption to support additional, but still limited, operations. CryptDB [20, 37] integrates various partially homomorphic encryption schemes to serve the major functionalities required in database applications. Each datum is encrypted with various encryption schemes, one on top of another, forming multiple encryption layers. It can only support a few operations including equality check, range check, summation, word search, equality join and range join operations, and is barely sufficient for a typical Online Transaction Processing (OLTP) workload. To support Online Analytical Processing (OLAP) workload, MONOMI [21] proposed to use a split server/client approach to extend the functionality of CryptDB. In brief, it optimizes the system by executing as much of the query as possible that can be efficiently accomplished over encrypted data in the cloud, and offloading the remaining execution to the data owner. In terms of performance, CryptDB reduces throughput by 27% on the OLTP benchmark while MONOMI introduces an overhead ranging from 3% to 133% on OLAP. In terms of security, these solutions are still not perfect. As it uses some weaker encryption schemes (e.g. OPE) that trade performance over security, it requires careful design from the administrators to balance security and performance.

Homomorphic encryption, though powerful, suffers from a lot of shortcomings. It is either unacceptably inefficient or provides only limited support for operations, resulting in it not being useful in solving practical cloud-based computational problems.

3.2 Commercially Available Architecture

In this section, we review some existing security solutions in commercial products. In general, they fall into one of two categories: 1) a secure co-processor approach that executes sensitive information on a standalone and secure computing platform, or 2) a generic approach that builds roots of trust in hard-

ware, either as a co-processor or as a main processor, that ensures that the computing platform is trustworthy.

3.2.1 Secure Co-Processor

The IBM 4758 secure co-processor [56, 57] is a self-contained, tamper-resistant and responsive computing device attached to the host computer via a Peripheral Component Interconnect (PCI) interface. It also contains a factory-built private key of a public and private key pair and uses the public key cryptographic algorithm to enable attestation and symmetric key exchange with remote parties. Internally, there are cryptographic accelerators to perform data en-/decryption and digital signatures. It also contains limited memory to store the encrypted or decrypted data inside the platform. Because it is designed as a general purpose secure co-processor, it remains a high cost solution.

TrustedDB [22] employs the IBM 4764 secure co-processor [58], a successor of IBM 4758, to protect the data confidentiality of an outsourced database in the cloud. Due to the limited computational power and storage capacity of the secure co-processor, it still relies on the host computer to execute non-sensitive data and store the outsourced database. The secure co-processor is responsible for most query executions, including parsing the encrypted queries sent from the client. The queries, after parsing, are further re-written into public sub-queries or private sub-queries. Public sub-queries are handled by the host computer to process unencrypted data while private sub-queries are executed by the secure co-processor to compute encrypted data. The final query result is assembled and encrypted by the secure co-processor. Under such architecture, sensitive data can only be decrypted and stored inside the secure co-processor. However, its limiting computational power and high latency data communication cause $1.03\times$ to $8\times$ slowdown on OLAP workloads.

A FPGA provides an alternative to a custom secure co-processor. It exhibits the feature of hardware security and reconfigurability, making it a good

alternative to build a trusted computing platform. Eisenbarth *et al.* [59] proposed to build the root of trust on FPGA with the same Trusted Platform Module (TPM) functionalities, which allows flexible updates to TPM functionality. Eguro *et al.* [60] proposed to use a FPGA to build a custom execution engine for secure data processing, where the encrypted data can be decrypted, computed and re-encrypted inside the reconfigurable logic. The bitstream is protected by encryption and securely loaded into the FPGA via the on-board secure boot logic. These proposals provide a general discussion on the possibility of using a FPGA to build a trusted computing platform but lacks an in-depth discussion of its performance impact.

Inherited from Eguro’s proposal, Cipherbase [23] proposed to use a FPGA-based query co-processor to process an encrypted database in the cloud. Similar to TrustedDB, it also relies on the host computer to execute non-sensitive data and store the outsourced database. But it leverages the framework in CryptDB, so the majority of the query execution is done at the host computer. The FPGA is only responsible for those operations that CryptDB cannot support or can barely support with low efficiency (e.g. addition, multiplications). This has two major benefits over TrustedDB. First, the major query execution is offloaded to the host computer, which has higher computational power. Second, it reduces data communication over the PCI bus. However, it also poses a potential hazard for performance because the FPGA generally runs at a slower clock frequency than Application Specific Integrated Circuit (ASIC) and data communication over the PCI bus is still quite costly. Its evaluation over OLTP workloads shows 17% to 20% throughput degradation but the performance over OLAP workloads is still largely unknown. The authors also suggested that it may not perform well over OLAP workloads because data mining queries typically touch a large number of records. It greatly amplifies the performance overhead caused by the FPGA and data communication.

A secure co-processor often has limited computational resources. Relying

on a secure co-processor to execute a large amount of sensitive information can cause significant performance overhead.

3.2.2 Trusted Platforms

Trusted Computing Platform Alliance (TCPA) has specified TPM [11], a small chip mounted on the motherboard, to provide a root of trust for a computing platform. Its main objective is to protect the platform from software attack by providing a remote attestation and protected storage mechanism. The attestation is based on a white-list approach that compares the hashing measurement of all the software layers with the approved value stored inside the chip. It means that any software attack on the system would alter the hashing measurement and therefore TPM can detect and respond to the tampering. TPM provides specific access control on the encrypted data called sealing. A sealed storage can only be deciphered in a specific system configuration. A number of works propose using TPM to provide a trusted platform in Cloud Computing [61, 62]. However, a major disadvantage of TPM is that it offers protection of encryption keys but does not protect the encryption key and the decrypted data once unsealed. Thus, the decrypted data and encryption key can still be obtained by an attacker through cold-boot attack [14].

Intel Trusted Execution Technology (TXT) [45] also builds upon TPM to provide protected execution. The processor support extends the security of TPM in that it erases the memory content in off-chip memory if any tampering is detected by TPM. This targets against memory snooping software attacks but not against physical attacks. Intel AES-NI [63] provides instruction support to use the on-chip Advanced Encryption Standard (AES) cryptographic engine. It also has an on-chip Digital Random Number Generator (DRNG) [64] to generate the encryption key. However, they are originally designed for acceleration purposes where the encryption key can be stored and the decrypted data are stored outside the processor chip. Sensitive information thus

remains unprotected in off-chip memory.

ARM TrustZone [46] introduces a set of security extensions to the ARM processor to provide a trusted execution environment by virtually partitioning all hardware resources into two isolated subsystems - “secure world” and “normal world”. The root of trust is provided in hardware by implementing an additional bit in the system bus, cache and Translation Lookaside Buffer (TLB) for strong isolation between these two subsystems. Security sensitive applications can run in a secure world and is protected from any software attack in a normal world. ARM TrustZone architecture also enforces secure boot mechanisms and secure I/O paths via trusted peripherals. Unlike TPM and secure co-processor approaches, there is no cryptographic engine or factory-installed private key burnt into the processor. In fact, ARM’s model provides a generic but isolated system for executing security sensitive applications. It does not consider the privacy protection of off-chip memory.

Maxim Integrated (formerly Dallas Semiconductor) manufactures the DS5250 secure microprocessor [65]. As its architecture is very similar to the secure processor approach, we include it in Section 3.3.1 below for ease of discussion.

Commercial architecture usually aims at providing a trusted platform for software to run (the software stack is securely booted). It relies on software to perform security critical tasks but does not consider physical attacks.

3.3 Academically Proposed Architecture

Architectural support for building a secure computing platform has been widely studied in academia. Its advantage regarding security is two-fold. First, it builds the root of trust in the processor core. This is a more secure approach than building it on software, based on the assumption that probing the internal components of the processor chip is sufficiently difficult. Second, it greatly reduces the attack surfaces in software because software can leverage the hardware support to reduce its code size. Hardware architecture can

provide security support in various ways. It includes, but is not limited to, off-chip memory protection in secure processors, enhanced hypervisor’s isolation between virtual machines, and data-centric security.

3.3.1 Secure Processor

The concept of a secure processor is fundamentally based on bus encryption proposed by Best [66, 67] in 1979. In his proposal, all data and instructions are encrypted and only decrypted inside the microprocessor chip. Based on this architecture, VLSI technology [68] proposed having an on-chip memory management unit for encrypted instruction and data transfer from an off-chip memory on a page-by-page basis. General Instrument Corporation [69] further optimized the system using a triple Data Encryption Standard in block chaining mode for encryption and a keyed hash algorithm for authentication. The Maxim DS5250 secure microprocessor [65] is designed to meet the physical security requirements of Federal Information Processing Standards (FIPS) 140 and Common Criteria certifications. In addition to the triple-DES bus encryption, it has a microprobe shield and environmental sensors to trigger a rapid “zeroization” of secure information as a tamper response. All these industrial efforts target standalone embedded processor applications where software is pre-loaded into the system during manufacturing.

Until recently, there are a number of works targeting DRM, in which software is dynamically and remotely installed into the system requiring similar but more sophisticated architecture. In general, each secure processor is associated with a factory-built private key. To protect the application software, software vendors encrypt their application’s binary using a symmetric-key encryption, where the key is distributed to each secure processor using its public-private key pair. Their security goal is to protect application code and data as a whole.

The Execute-Only Memory (XOM) [24] architecture assumes that operat-

ing systems are untrusted. It provides an isolated memory compartment in hardware for each application. Each compartment has its own session key and this key is used to en-/decrypt the associated instruction and data. At any time there is only one memory compartment active and the corresponding session key is loaded for on-chip en-/decryption. To manage the session keys and the hardware resources, a XOM Specific Operating System (XOMOS) [70] is also developed. The XOMOS virtualizes the session key table to support as many applications to be run in its own memory compartment as possible.

AEGIS [25], having a similar security model, provides more flexibility of the protection mechanism. Apart from the untrusted operating system solution, AEGIS also develops a scheme which uses a trusted security kernel to handle multi-tasking and provide security features for applications. The security kernel can be securely booted using the root of trust in the processor chip. It further replaces the root of trust from a factory-installed private key to a Physical Unclonable Function (PUF) [71]. It also uses a One-Time Pad (OTP) encryption scheme proposed by Yang *et al.* [36] to address the performance overhead in XOM architecture. In XOM, the encryption latency is added directly to the memory access critical path which results in high performance penalty. The proposed OTP encryption yields significant improvement in performance because it allows the en-/decryption to be performed in parallel to memory access from off-chip memory.

Secure architecture is also designed to protect sensitive data in portable devices. Relying on a permanent private key inside the processor simply restricts the portability of trust from one device to another. Secret-Protected (SP) architecture [26] and Bastion [33] focus on the management of keys. They propose new architectures to bind the critical secret to a user-defined master key instead of a factory-installed private key.

In SP [26] architecture, a user master key is generated by hashing a passphrase from the user's input via a secure I/O. After the master key is generated, a

hierarchical key chain can be further developed for various cryptographic operations. A Trusted Software Module (TSM) is used to manage the key chain, including the user's master key. The TSM's functionalities can vary for various applications depending on the security requirement. The SP architecture supports a concealed execution mode to protect the execution of TSM. In this concealed execution mode, all instructions and data going to the off-chip memory are protected by encryption and hashing. In a new version of SP [72], it extends local trust of a user's secret on his own device to a remote trust model on multiple devices owned by a single authority.

Bastion [33] architecture, based on SP architecture, provides scalable TSM establishment and individual attestation through a trusted hypervisor. The Bastion architecture first secures the execution of hypervisor, which in turn provides execution protection to the multiple TSMs invoked in the system. Each TSM has its own isolated memory compartment enforced by access rules implemented in TLB. On-chip cryptographic engines are used to protect runtime memory used by the trusted hypervisor and the TSMs.

Iso-X [27], similar to Bastion, provides hardware enforced isolation on TLB and off-chip memory encryption to multiple trusted processes. The major difference between Iso-X and Bastion is that it eliminates the trusted hypervisor and pushes the management of multiple trusted processes into hardware. This is achieved by reserving a memory region for storing management information, where it is only accessible by the Iso-X hardware. The management information involves a compartment vector and a compartment table. Compartment vector is used to map the physical memory page to each compartment, while compartment table describes each compartment created in the system.

A secure processor provides off-chip memory encryption and an integrity check mechanism for application code and data as a whole. It does not consider an application scenario that the data provider and software provider are from different parties. Also, when the application and data need to be migrated from

one to another secure processor, the data have to be re-encrypted because of the different encryption keys used in different secure processors.

3.3.2 Secure Hypervisor

With the emergence of Cloud Computing, hypervisors are becoming more complicated and thus more vulnerable to attacks. Besides, the lack of physical control and transparency of computing platforms also raise security concerns from users. In view of the existing vulnerabilities of hypervisors, there are a number of works proposed to use new hardware architecture to reduce the attack surface of hypervisors.

NoHype [47] proposed to allocate resources to each guest VM without any hypervisor. The guest VM has full control and direct access to the underlying dedicated hardware. Each VM is allocated a single processor and hardware-enforced partitioned memory. As each VM owns a core and a block of partitioned memory, the hardware resource is not shared with other tenancies. This eliminates the vulnerability caused by a multi-tenant and shared resource environment, but also lessens the advantage of resource pooling in Cloud Computing.

HyperWall [48], on the other hand, does not remove the entire hypervisor. Instead, it uses new hardware architecture to enhance the isolation between virtual machines by extending the root of trust in the processor to hypervisor. This is achieved by implementing a confidentiality and integrity check logic in TLB. Akin to Iso-X, a portion of off-chip memory is reserved to store a Confidentiality and Integrity Protection (CIP) table when the computer boots up. The CIP table storing the mapping of hypervisor and access rights to the physical memory pages is only accessible by the hardware. Thus, the isolation of memory partition among guest VMs is strictly enforced in hardware.

While HyperWall focuses on VM isolation, HyperCoffer [49] protects VMs through cryptographic methods. It adopts AISE [28] encryption and BMT

[28] integrity check techniques. AISE and BMT were originally designed for a secure processor to remedy the disadvantage of OTP encryption [36] so that it can extend the protection to permanent storage. HyperCoffer leverages the concept of “shim” in Overshadow [73], which is a piece of software to mediate all data communications between the protected VM and the untrusted hypervisor. Each VM is encrypted with its own encryption key, which is stored inside the processor during execution. Similar to secure processor architecture, it has a factory-installed private key in the processor for attestation and key exchange of the VM encryption key. Because each VM is encrypted with its own encryption key, its data are strongly protected from other VMs.

Hardware-enhanced virtualization aims at providing strong isolation among virtual machines. Again, it does not consider the SaaS execution model in Cloud Computing.

3.3.3 Data-Centric Security

Protecting an application code and data as a whole is sometimes undesirable in certain execution models (e.g. SaaS model in Cloud Computing). Sometimes users only want to protect a piece of data/document, regardless of the applications running on it. There are two main scopes of data-centric security: access controls and privacy protection, which are usually implemented at OS or application level. Recently, Chen *et al.* [50, 51] proposed two architectures to secure information sharing on multiple devices. They both leverage hardware architecture to enhance the data access controls and policies.

The first proposed architecture [50] uses SP architecture [26] to build a TSM at the application domain. The TSM is responsible for implementing various policies for access controls. Since TSM is directly protected by SP’s hardware mechanism, a trust chain is formed from the underlying hardware to the application domain, bypassing the operating system which may be compromised. Upon receiving encrypted data/documents, the TSM first decrypts

the data and evaluates the policies for access control. The decrypted data are stored in the TSM buffer, a memory region protected by cache line encryption in SP architecture. Then, the TSM authenticates the recipient and checks the policies before granting access to the protected data/documents.

The second proposed architecture, DataSafe [51], further eliminates TSM from the application. It allows data to be self-protected from any unvetted third-party programs. To achieve this, the DataSafe architecture provides two key protection mechanisms: 1) a run-time policy handler that translates high-level policies expressed in software into hardware tags and, 2) fine-grained run-time data tracking to propagate the hardware tags. The access control policies are first associated with the protected data and then further translated into hardware tags. Since application programs and the operating system are not involved in this process, the protection is made independent of any third-party programs. However, the DataSafe architecture relaxes the adversary model from physical attacks and does not focus on protecting data confidentiality during execution.

These solutions consider an execution model where protected data are dissociated from the application program. They focus on providing architectural support toward access control policies. Our approach takes direct encryption/decryption support in hardware to protect the confidentiality of encrypted data.

3.4 Chapter Summary

In this chapter, we reviewed three main categories of past work that aim at providing a secure computing platform for data processing: 1) homomorphic encryption, 2) commercially available architecture, and 3) academically proposed architecture. All of these solutions suffer from one of the following shortcomings: 1) inefficiency, 2) sharing a weaker adversary model, and 3) targeting different application domains and a different execution model from ours. Our

approach explores opportunities to leverage hardware architecture techniques to protect the confidentiality of outsourced data securely and efficiently. We build upon existing techniques to build a hardware root of trust and create secure execution environments, but propose new techniques to enable efficient computations on encrypted data. Our techniques enable distributed and parallel data processing in the Cloud Computing environment and are complementary to existing techniques that create secure execution environments. A comparison of past work with our approach is shown in Table 3.1.

	Goal	Adversary Model	Root of Trust	Protection Domain	Is data always encrypted in off-chip memory?	Is re-encryption required for data migration? ^b	Efficiency ^a
Homomorphic Encryption	Gentry [19]	Honest-but-curious administrator attack	Encryption algorithm	Data (database)	Yes	No	Very poor
	CryptDB [20, 37]	Honest-but-curious administrator attack	Encryption algorithm	Data (database)	Yes	No	Moderate
	MINOMI [21]	Honest-but-curious administrator attack	Encryption algorithm	Data (database)	Yes	No	Moderate
Commercially Available Architecture	TrustedDB [22]	Insider software and physical attack	IBM 4764	Data (database)	Yes (data only decrypted in co-processor)	No	Poor
	Cipherbase [23]	Honest-but-curious administrator attack	FPGA	Data (database)		No	Moderate
	TPM [11]	Third parties software attack	TPM	Computing platform	No (data is not protected after decryption)	N/A	N/A
	ARM TrustZone[46]	Third parties software attack	Main processor	Application software	No	N/A	N/A
	MAXIM [65]	Insider software and physical attack	Main processor	Application software	Yes	Yes	N/A
Academically Proposed Architecture	XOM [24], AEGIS [25]	Insider software and physical attack	Main processor	Application software	Yes	Yes	N/A
	SP [26], Bastion [33]	Insider software and physical attack	Main processor	Trusted software module	Yes	Yes	N/A
	Iso-X [27]	Third parties software attack	Main processor	Application software	No	N/A	N/A
	NoHype [47]	Third parties software attack	Main processor	Virtual machine	No	N/A	N/A
	HyperWall [48]	Third parties software attack	Main processor	Virtual machine	No	N/A	N/A
	HyperCoffer [49]	Insider software and physical attack	Main processor	Virtual machine	Yes	Yes	N/A
	Chen [50]	Insider software and physical attack	Main processor	Data	Yes	Yes	N/A
	DataSafe [51]	Third parties software attack	Main processor	Data	No	N/A	N/A
Our Approach	CypherDB	Honest-but-curious administrator and physical attack	Main processor	Data (database)	Yes	No	Good

Table 3.1: Comparison of CypherDB with related effort to secure outsourced data processing

^aOnly work on database application is compared.

^bOnly work with off-chip memory encryption requires re-encryption.

Chapter 4

CypherDB System Overview

In addition to protecting data confidentiality, our CypherDB computing platform supports highly efficient computations on encrypted data in a distributed Cloud Computing environment. To achieve this goal, each computing node in the cloud must be able to execute the encrypted database with high performance and security. This chapter first describes the general overview of the system and the high-level architectural model of each secure node. We then present the attestation protocol and the execution flow for the database owner to start a secure query session. Finally, we discuss some virtualization issues in the cloud.

4.1 General Overview

Figure 4.1 depicts our CypherDB cloud model as described in Chapter 1, which involves two parties: a CSP and a database owner. To protect data confidentiality, the database owner encrypts the outsourced database with symmetric key encryption [32] before exporting it to the CSP for future querying. The CSP hosts the database server and provides storage and database administration services to the database owner.

To process the encrypted database, a server/node in CSP has to be “empowered” with two features: 1) equipping a CypherDB secure processor and, 2)

Figure 4.1: CypherDB application scenario

having the database encryption key K_{db} stored inside the processor chip. In addition, a query encryption key K_Q has to be stored inside the server's processor chip to protect the query result. We refer to these “empowered” servers/nodes as secure servers/nodes. A normal server/node (without CypherDB secure processor or database encryption key) is not capable of processing the encrypted database.

We envision that CSP could set up a secure cluster which consists of a collection of servers/nodes equipped with CypherDB secure processors. To query the outsourced database, the database owner communicates with a single secure server as if the entire database were stored in it. In CSP, the outsourced encrypted database is partitioned and stored in a distributed manner within the secure cluster. The secure server manages the query processing in such distributed database and sends back the encrypted query result to the database owner.

4.1.1 Application Scenario

For simplicity and without the loss of generality, we consider a scenario, as shown in Figure 4.2, where the outsourced database is partitioned and stored in two storage nodes, S_1 and S_2 , each of which can perform a scan operation

Figure 4.2: CypherDB execution model

locally and filter out the useful data to a computational server C , for the corresponding query execution. For example, the database owner can issue a query to, but not limited to, the computational server in the form of “SELECT A FROM B_1 AND B_2 WHERE $B_1.K = B_2.K$ AND C_1 AND C_2 ” where A is the set of data to be returned to the client, K is the attribute for join operation, and C_1 and C_2 are the conditions over the attributes in B_1 and B_2 respectively. This query may not need to be encrypted because it leaks almost no sensitive information from the issued query. The computational server first pushes the selection workload to the storage nodes. After that, the storage nodes send back a set of selected records $t[i]$ satisfying the specified conditions. The computational server will then compute the final join or aggregation workload, producing the final result to the client.

Before outsourcing the database to the cloud, the database owner first encrypts each record of the database $t[i]$ with an encryption key K_{db} and then appends it with a metatext $t[i]*$, transforming each record from $t[i]$ to $[Enc(t[i], K_{db}), t[i]*]$. The metatext $t[i]*$, which is useful for the secure execution in the server, can be sent in clear form since it reveals no information about the encryption. The details of this metatext will be explained in Chapter 5. As a result, every storage node and the computational server now need to handle each record in the form of $[Enc(t[i], K_{db}), t[i]*]$ instead of $t[i]$ in storage, communication, or execution.

In each secure server/node, the metatext of each record $t[i]*$ is used in conjunction with the stored on-chip database encryption key K_{db} to perform computations on the encrypted records. The data confidentiality is strongly protected by the CypherDB secure processor during execution. The query result is further encrypted by query encryption key K_Q in the secure server before sending it back to the database owner.

4.2 Architectural Model of a Secure Server/Node

Figure 4.3: Architectural model of each secure server/node. Components in grey represent untrusted entities while those in white represent trusted entities.

The security goal of each secure server/node is to perform computation on encrypted data without information leakage to any honest-but-curious adversary. In other words, with the assumption that the processor is unbreakable, the sensitive data have to be encrypted all the time outside the processor boundary. To enable this feature, each CypherDB secure server/node provides architectural support for two important mechanisms: 1) secure on-chip encryption key exchange and 2) efficient on-chip encrypted data computation. Figure 4.3 depicts a high-level architectural model of a secure server/node. CypherDB secure processor and software support work together to enable computations on encrypted data inside the processor. The other hardware components and operating system are all untrusted entities, which are unable to obtain the actual value of the sensitive data.

4.2.1 On-Chip Encryption Key Exchange

To exchange the encryption key securely to the secure processor, we rely on a unique private key burned into the processor chip, which is not accessible by any software (no instruction to read or extract the key). Database encryp-

tion key K_{db} and query encryption key K_Q can thus be sent to the processor chip through KEM [74] - encrypting K_{db} and K_Q with the public key of the processor and decrypting it using the on-chip private key. As a result, the encapsulated database key and query key are $E_{PK}(K_{db})$ ¹ and $E_{PK}(K_Q)$ respectively. Only a processor with a corresponding private key is able to extract the key out from $E_{PK}(K_{db})$ and $E_{PK}(K_Q)$. This allows different database owners to perform on-chip key exchange of their K_{db} and K_Q to a single secure server/node. Thus, secure servers/nodes can process the encrypted database from any database owners. Such key exchange protocol only needs to be done once for the database owner to use the service. The database owner's credentials including the encrypted database encryption key can be stored in a secure server and brought up only when the database owner logs in to use the service.

4.2.2 On-Chip Computations on the Encrypted Data

With the database encryption key stored on-chip, the encrypted data can be deciphered inside the processor chip so that computations can be performed on the plaintext. If the data need to be modified, they are further enciphered before writing them back to off-chip memory. A conceptual architecture for such execution model is shown in Figure 4.4. However, a naive implementation of such architecture results in both performance and security issues.

In terms of performance, these en-/decryption layers introduce additional execution latencies to the overall execution. Moreover, such on-chip decrypt-process-encrypt operation has to be performed on every off-chip processor-to-memory data transaction. A simple database operation such as Query 1 in TPC-H [75] can result in millions of off-chip memory transactions. To remedy this, a novel scheme, that can offload the cryptographic latencies with the assistance of the metatext of a record $t[i]*$, is developed. Comprehensive instructions support is also proposed. The details of such scheme will be

¹ $E_{PK}()$ represents a public key encryption using the public key as described in Section 2.3.2

Figure 4.4: A conceptual architecture showing the operation of a secure processor

described in Chapter 5.

In terms of security, the intermediate data generated during execution and the query result returning back to the database owner could leak sensitive information of the encrypted data. To protect these data, a run-time off-chip memory protection mechanism is employed. The query result is also protected using the aforementioned look-ahead encryption scheme. The detail of these protection mechanisms will be discussed in Chapter 6.

4.3 Attestation

Attestation has to be performed for the secure processor from CSP and also the query key from the database owner. The purpose of attestation is to ensure the trustworthiness of the corresponding parties. This is to avoid any impersonation attacks [76] where an adversary can successfully pretend to be a legitimate party in the system.

4.3.1 Secure Processor Attestation

Although the database and query encryption keys are secured by the on-chip key exchange protocol, an adversary can send a forged public key to the database owner and pretend the public key belongs to one of the secure nodes in a secure cluster. If the database owner is unable to verify the trust-

worthiness of the public key, the adversary can obtain the database encryption key with ease.

Therefore, attestation of the secure processor is crucial to ensure that the critical secret - the database encryption key is trustworthy. Attestation of the secure processor can be achieved by certifying itself using Public Key Infrastructure (PKI) with the use of its unique on-chip private key. It works by the following two-step protocol:

1. During manufacturing, the chip manufacturer generates a pair of public and private keys for each secure processor. The private key is burnt into the chip and is used to sign its corresponding public key. This generates a digital certificate for the public key.
2. For attestation, the CSP sends the public keys and the certificates for each secure node to the database owner. The database owner can thus obtain the public keys and verify their certificates accordingly.

As the private key of the secure processor is always kept secret, an adversary is unable to forge the digital certificate. This certificate can thus be used to attest the secure processor.

4.3.2 Query Key Attestation

At the user's side, an attacker can get the encrypted database key $E_{PK}(K_{db})$ and encrypted query key $E_{PK}(K_Q)$ from the communication channel. Having obtained $E_{PK}(K_{db})$, the attacker can send a forged query key K_F in an attempt to obtain the query result. This can be done by encrypting K_F with the public key of the secure processor to obtain $E_{PK}(K_F)$. These $E_{PK}(K_{db})$ and $E_{PK}(K_F)$ can then be sent to the secure processor. Due to the fact that the secure processor is unable to identify the ownership of the encryption keys, an attacker can easily obtain the query result without knowing the database encryption key.

To ensure that the query key indeed originated from the database owner, she has to, following the discussion in Section 2.3.3, create a digitally signed query key to authenticate the query key and ensure that it has not been tampered with. In short, the database owner first hashes the encapsulated query key $Hash(E_{PK_C}(K_Q))$ and encrypts it using her own private key to create a digital signature. Similarly, she can also authenticate the database encryption using the same digital signature by embedding the hash value of the encapsulated database key $Hash(E_{PK_C}(K_{db}))$. She can then send the digitally signed query key $Sig(E_{PK_C}(K_{db}), E_{PK_C}(K_Q))$ to the server.

At the server side, the query key can be extracted and verified using the digital signature. The hashed value is extracted by decrypting the digital signature Sig and is compared with the immediate hashing of the received encapsulated query key $(E_{PK_C}(K_Q))$.

4.4 Execution Flow

Figure 4.5: CypherDB execution flow

Figure 4.5 depicts the execution flow of outsourcing and querying the encrypted database outsourcing. This can be seen as an execution setup of secure query processing from the application perspective. The database owner first encrypts the database with K_{db} and specifies the columns to be built as indices before outsourcing it to the CSP. The distributed encrypted database is managed by multiple secure nodes whereas the indices are built accordingly.

To query the outsourced database, the database owner can set up a secure query session by sending K_{db} and a query encryption key K_Q to the allocated CypherDB secure processors. In summary, it involves the following procedures:

1. The database owner requests a secure query service from CSP and specifies the resources needed (e.g. number of cores, bandwidth).
2. The CSP allocates the resources as requested and sends all the public keys $\{PK_i\}$ and digital certificates $\{Sig(PK_i)\}$ of the secure processors involved to the database owner.
3. The database owner checks the trustworthiness of the secure processor by verifying $\{Sig(PK_i)\}$ and then encrypts K_{db} with all verified public keys, resulting in a set of encapsulated database keys $\{E_{PK_i}(K_{db})\}$. K_Q is only encrypted with the public key of the secure server PK_C he talks to, resulting in encapsulated query key $E_{PK_C}(K_Q)$.
4. The database owner sends all encrypted database encryption keys $\{E_{PK_i}(K_{db})\}$, query encryption key $E_{PK_C}(K_Q)$, the verification metric $Sig(E_{PK_C}(K_{db}), E_{PK_C}(K_Q))^2$ to the CSP.
5. The CSP distributes $\{E_{PK_i}(K_{db})\}$ to the appropriate secure processors in order to store K_{db} securely on-chip. It further stores K_Q into the secure processor chip of the secure server and checks the verification metric $Sig(E_{PK_C}(K_{db}), E_{PK_C}(K_Q))$.

²The verification metric $Sig(E_{PK_C}(K_{db}), E_{PK_C}(K_Q))$ is used to authenticate the database and query key at the same time.

Upon receiving the query from the database owner, the secure server first pushes the selection workload to the pre-allocated secure nodes. After that, the secure nodes send back a set of relevant encrypted records. The secure server then computes the final join or aggregation workload. Finally, the query output is encrypted with K_Q before returning to the database owner.

4.5 Database Virtualization

As we envision that CSP could establish a secure cluster formed by a collection of secure servers and nodes, resource pooling can be performed in this secure cluster for secure query processing. In this section, we discuss five important virtualization features that may be potentially impacted by our proposed solution as follows:

- **Resource Allocation:** Because each CypherDB secure processor can accommodate the database encryption keys from different database owners, the CSP can allocate any available secure server/nodes, following the aforementioned key exchange protocol, to the database owners upon a request for a secure query session.
- **Resource Re-allocation:** Migration of processes may be necessary due to load balancing and fault tolerance in the Cloud (i.e. move the query processing process from one secure server/node to another). In this case, the CSP can re-allocate some secure server/nodes and send the corresponding secure processors' public keys to the database owner for exchanging the database encryption key with those "new" secure server/nodes. On the other hand, database owners may also request more resources by repeating the secure query session request.
- **Distributed Storage and Processing:** In CypherDB, the database is encrypted at a logical level that eases the partitioning process. Horizontal

partitioning (i.e. on a record basis) can be performed directly on the encrypted database without involving any re-encryption. The distributed database can also be processed in parallel using different secure nodes.

- Multi-tenancy: With different database and query encryption keys, the CypherDB secure processor is capable of processing encrypted databases from various database owners. Through switching the use of the database/query encryption key, each secure server/node is able to support multi-tenant query processing.
- Abstraction to End-User: The protection mechanism is abstracted to the end user. From the user's perspective, data are protected by strong encryption while the encryption keys are protected by the secure processor. Our proposed solution guarantees that data are only decrypted inside the processor chip without information leakage to off-chip hardware components. Also, the user only needs to talk to the secure server whereas the back-end resource allocation and query processing are all hidden from users.

Chapter 5

Proposed Encryption

Mechanism

The architecture of the CypherDB secure processor requires additional encryption (or decryption) operations at the secure processor boundary that encrypts (or decrypts) the data to (or from) memory. One major disadvantage of this architecture is that the en-/decryption operation is on the memory access critical path. This chapter presents our novel look-ahead encryption scheme to solve this problem. We first describe the encryption mechanism and its impact on our scheme. Then, our proposed encryption scheme and its performance impact are discussed. To illustrate the usage of our scheme, we present our technique to encrypt an outsourced database with the look-ahead encryption scheme. Finally, a security analysis of our proposed data encryption scheme is discussed.

5.1 Block-Cipher Encryption

Our proposed look-ahead encryption scheme relies on block cipher encryption in two different modes: 1) Counter-Mode (CTR) and 2) output feedback mode (OFB). These two encryption modes [77] make use of a data-independent and unique seed value s to decouple the en-/decryption computation from the ac-

Algorithm 1 Pseudo-code of AES-CTR and AES-OFB encryption

```
1: /* AES-CTR encrypt data with less than or equal to 128 bits */
2: function  $Enc_{ctr}(s, d, K_{db})$ 
3: Input:  $s$  = seed value,  $d$  =  $l$ -bit data,  $K_{db}$  = database key
4: Output:  $y$  =  $l$ -bit ciphertext
5:    $p = AES(s, K_{db})$ 
6:   for  $i = 1, \dots, l$  do
7:      $y_i = d_i \oplus p_i$ 
8:     /* Decryption:  $a_i = y_i \oplus p_i$  */
9:   end for
10:  return  $y$ 
11: end function
12:
13: /* AES-OFB encrypt data with larger than 128 bits */
14: function  $Enc_{ofb}(s, d, K_{db})$ 
15: Input:  $s$  = seed value,  $d$  =  $l$ -bit data,  $K_{db}$  = database key
16: Output:  $y$  =  $l$ -bit ciphertext
17:    $p^0 \leftarrow s$  /* seed value  $s$  is used to get the first encryption pad  $p^1$  */
18:    $m \leftarrow \lceil \frac{l}{128} \rceil$  /* to calculate the number of encryption pads needed */
19:   for  $h = 1, \dots, m$  do
20:      $p^h = AES(p^{h-1}, K_{db})$ 
21:   end for
22:    $p = p^0 p^1 \dots p^m$ 
23:   for  $i = 1, \dots, l$  do
24:      $y_i = d_i \oplus p_i$ 
25:     /* Decryption:  $a_i = y_i \oplus p_i$  */
26:   end for
27:  return  $y$ 
28: end function
```

tual data value, which are formalized in Algorithm 1. AES [78] is used as the block cipher encryption because of its efficiency in hardware [79] and its well-proven security.

CTR encryption encrypts any data less than or equal to 128 bits. The function Enc_{ctr} takes 128-bit seed value s , l -bit long data d and the database key K_{db} as inputs and generates a l -bit long ciphertext as the output. In this function, an encryption pad p is generated using AES encryption $AES(s, K_{db})$ which takes the 128-bit seed value s and database key K_{db} (line 5). The encryption is done by performing an Exclusive-OR (XOR) operation on the most significant l bits of the encryption pad with the data (lines 6-9). To encrypt the attribute longer than 128 bits, OFB encryption function Enc_{ofb} takes the same input and generates the same output as Enc_{ctr} but with a longer input data d and ciphertext y . It first generates a series of 128-bit encryption pads to be encrypted with the data (lines 17-21). These pads are then concatenated together (line 22). The most significant l bits of the concatenated encryption pad $p^0 p^1 \dots p^m$ are then used to encrypt the attribute data (lines 23-26). Decryption is also done by performing an XOR operation on ciphertext with the same encryption pad (lines 8 and 25).

The use of these two modes of encryption achieves two important objectives: 1) pre-computing the en-/decryption operations and, 2) transforming block cipher into stream cipher. Their impacts on our look-ahead encryption scheme are described as follows:

1. *Latency offloading*: Prior to actual data processing, the encryption seed can be pre-fetched to compute the encryption pad in advance (lines 4 and 15). Thus, the actual data en-/decryption latency is reduced to simple XOR operations (lines 5 and 19). This offloads the en-/decryption latency of block cipher from the actual data processing path.

2. *Execution-friendly encryption interface*: Encrypting the data in streaming fashion simplifies the code generation task for accessing the encrypted at-

tribute data because the data manipulations on attribute data are often byte or word, instead of block, oriented. Padding of byte-wide attribute data is thus unnecessary, unlike the use of any block-cipher encryption.

5.2 Look-Ahead Encryption Scheme

Our novel encryption scheme takes advantage of the aforementioned properties of CTR and OFB encryption (latency offloading and execution-friendly encryption interface) to alleviate the shortcoming of our proposed architecture through the use of custom instruction support. Table 5.1 describes the function of each of the four custom instructions. *l.seed* and *l.shift* are used to manipulate the seed encryption. *l.sload* and *l.sstore* are similar to the normal load/store instructions *l.load* and *l.store*, except that data are now decrypted and encrypted accordingly.

Table 5.1: Custom secure instructions

Instructions	Functional Description
<i>l.seed</i>	to pre-compute the encryption pad using the seed value taken from the general purpose register or immediate value in operand
<i>l.shift</i>	to shift the encryption pad in accordance with the value indicated in the operand
<i>l.sload</i> / <i>l.sstore</i>	to load/store the data in byte (<i>l.slbz</i> / <i>l.ssbz</i>), half-word (<i>l.slh</i> / <i>l.ssh</i>) or word (<i>l.slw</i> / <i>l.ssw</i>) length with decryption from/into the general-purpose register

The mechanism of our proposed scheme can be best illustrated using an example as shown in Figure 5.1, which demonstrates the necessary software modification (in red) of a segment of code both in C and assembly language. The function of this simple program is to find a particular character in an array of *name*. Consider the case where *name* = “*alex*”, which is stored as a 32-bit string *0x616c6578*, is now encrypted as *0xa358f2bc*. To process this encrypted data, the program first searches the seed value using a mapping function *seed()*. The returned value, which is the seed value, is then loaded

into an on-chip register Reg_{seed} through $l.seed$ instruction. The encryption pad $0xc23497c4.....$ can then be pre-computed before the actual data are loaded. To decrypt the data, the load instruction $l.lbz$ is replaced by a secure load instruction $l.slbz$. This takes the first byte of the encryption pad to perform an XOR operation with the data being loaded. Finally, $l.shift$ is issued to shift the encryption pad by a byte. Through shifting the encryption pad, the encrypted data can be loaded and decrypted iteratively. The encryption pad only needs to be generated once for each data item. The same mechanism also applies to storing the encrypted data to the memory (replace $l.slbz$ with $l.ssbz$).

<pre>// return the seed value of the input item int seed(void *item); asm(l.seed %0, 0x1:"r" (seed(name))); for (i=0; i<strlen(name); i++) { if (name[i] == 'e') { ... do something; break; } asm(l.shift 0x1); }</pre>	<pre>l.seed r5, 0x1 .loop: l.slbz r2, 0(r3) l.sfnei r2, 0x65 l.bf .L2 l.shift 0x1L2: ... do something</pre>				
Data	<table> <tr> <td>(ASCII) name = alex</td> <td>(Cipher) name = 0xa358f2bc</td> </tr> <tr> <td>(Hex) name = 0x616c6578</td> <td>(Enc Pad) pad = 0xc23497c4...</td> </tr> </table>	(ASCII) name = alex	(Cipher) name = 0xa358f2bc	(Hex) name = 0x616c6578	(Enc Pad) pad = 0xc23497c4...
(ASCII) name = alex	(Cipher) name = 0xa358f2bc				
(Hex) name = 0x616c6578	(Enc Pad) pad = 0xc23497c4...				

Figure 5.1: The modification of a software program that uses the custom secure instructions to pre-compute and shift the encryption pad in C and Assembly language

It can be seen that if the $l.seed$ instruction is issued before the $l.sload/l.sstore$, the encryption pad can be pre-computed and the encryption latency can be hidden from the program execution. Moreover, the shifting of the encryption pad can be made in accordance with the encrypted data being loaded. It makes the en-/decryption be performed seamlessly with the actual program execution.

5.2.1 Performance Impact

Ideally, the en-/decryption latency can be completely hidden from the overall execution time with our proposed scheme. In reality, the performance impact of our scheme, however, relies on how early the pre-computation of the encryption pad can be performed. In this section, we discuss qualitatively the performance implication of our scheme for the total execution time.

We first consider the case where the encrypted data are already fetched into the on-chip data cache. Without our scheme, the decryption latency is directly added to the total execution time as shown in Figure 5.2 (a). Our scheme can result in two possible scenarios, depending on the overlapping time of encryption pad computation and program execution. If *l.seed* is issued early enough so that the encryption pad is generated before *l.sload* is executed, it can hide the entire encryption latency and result in almost zero overhead execution (see Figure 5.2 (c)). Otherwise, the execution has to be delayed until the computation on the encryption pad is finished (see Figure 5.2 (b)).

Figure 5.2: A diagram illustrating the performance impacts of decryption of the data residing in the on-chip cache where (a) the look-ahead encryption is not used, (b) the program execution is delayed by the look-ahead encryption mechanism and (c) the look-ahead encryption mechanism can completely hide the encryption latency

On the other hand, if data are residing in off-chip memory, the encrypted

data have to be loaded into the processor before actual data decryption occurs. The memory access latency is thus added directly to the total execution time as shown in Figure 5.3 (a). Although this memory access further delays the execution, it provides more time margin for the encryption pad computation in our scheme (see Figure 5.2 (b)). If the encryption latency is less than memory access latency, the total execution time will not be delayed by the data en-/decryption in our scheme (see Figure 5.2 (c)).

Figure 5.3: A diagram illustrating the performance impacts of decryption of the data *residing in off-chip memory* where (a) the look-ahead encryption is not used, (b) the delay caused by the look-ahead encryption mechanism is alleviated by the off-chip memory access and (c) the look-ahead encryption mechanism can completely hide the encryption latency

5.2.2 Discussion

Despite the advantages of CTR and OFB encryption, these encryption modes require the encryption seed s to be different from each other under the same encryption key K_{db} [80]. Thus, the encryption seed used in our scheme not only has to be loaded prior to the encrypted data for latency hiding, but also needs to be unique to each encrypted datum for security purposes.

As a result, the encryption seed has to be carefully chosen depending on

the application. If we naively assign a random unique seed to each datum, it can introduce both performance and storage overhead. This is because the additional seed value requires off-chip memory storage and extra execution cycles to fetch it into the processor for encryption pad computation. In the next section, we describe how to encrypt an outsourced database using our proposed scheme.

5.3 Database Encryption

From the perspective of the database owner, the data to be outsourced are organized in a two dimensional logical table. We refer to a cell and a row of this logical table as attribute and record respectively. The outsourced database can therefore be protected by encrypting each attribute in this table. Our design also reduces the storage of encryption seed significantly. In this section, we first describe how to encrypt each attribute using AES-CTR and AES-OFB encryption. Then, the detail of the corresponding encryption seed is presented. Finally, some auxiliary database protection mechanisms are discussed.

5.3.1 Attribute Encryption

Each attribute is encrypted in either AES-CTR or AES-OFB. AES-CTR is used to encrypt attributes less than 128 bits like INTEGER or REAL data type while AES-OFB is used; otherwise, on data type like VARCHAR or BLOB. An encrypted record, $Enc(t[i], K_{db})$ (as described in Chapter 4), is therefore a collection of the corresponding encrypted attributes.

5.3.2 Attribute Encryption Seed

To maintain the uniqueness of the encryption seed, each attribute across a database must own a distinct seed to each other (*spatial uniqueness*), whereas the seed for the same attribute must not repeat upon every update operation

on that attribute (*temporal uniqueness*). The design of seed components has to satisfy spatial and temporal uniqueness, while the logical data structure is leveraged to formulate the seed during program execution.

Seed Components

Due to the security concerns on seed uniqueness, our design is to leverage the *logical schema* of the database to formulate the seed. In the structure of a logical schema, if each element of the schema has its own identifier (*ID*), each attribute can be identified by $(databaseID, tableID, rowID, columnID)$ which is spatially unique across various databases and tables. Temporal uniqueness can be achieved by appending a global incremental counter *cntr* to each record, which is shared by each attribute within that record. This counter is the record metatext $t[i]*$ as described in Chapter 3. Although having a record counter introduces a penalty on re-encrypting a non-updated attribute within that record, this penalty is insignificant because most workload is read-intensive rather than write-intensive. The update-intensive workload also requires extensive read operations to search for appropriate records to write [81].

Seed Formulation

In a typical database application, a logical schema is used in most operating layers and is eventually translated into its physical schema in order to locate the record in a database file. The formation of the logical schema identifier can thus be embedded into the logical-to-physical schema translation software process. In other words, the actual program execution is able to “generate” the encryption seeds by re-using some software execution parameters, at runtime. Take SQLite [82] as an example. The address of an attribute in a record is stored in an array $aOffset[ColumnID]$ where *ColumnID* is the index. This array, including the index, is used for every record access during execution. If the *l.seed* instruction is issued immediately after this translation process, the

index is possibly still stored inside registers or the on-chip cache. Thus, using this index as one of our attribute seed components can reduce additional load operations on fetching the encryption seed from the off-chip memory.

In summary, using the logical schema and record counter to formulate the attribute seed exhibits three main advantages: 1) ensuring the spatial and temporal uniqueness, 2) reducing the storage overhead and, 3) minimizing performance overhead by reducing off-chip memory access. Most seed components can be “generated” during program execution, except that the record counter requires external storage.

5.3.3 Query Output Protection

After processing the query on the encrypted database, the query output has to be sent back to the user. Protection of the query output is necessary because it contains aggregation results or the actual data of the encrypted database.

The query output is protected by encrypting it in the same way as attribute encryption. This can be achieved because the result of the *SELECT* query can always be organized in a database table where the *select-list* right after the *SELECT* clause defines the columns of the table. For instance, the *callback* function in SQLite returns a row of the query output at a time, where the items in the *select-list* are stored in an array. Each item of the query output can therefore be encrypted by AES-CTR and AES-OFB, except that the attribute seed is in the form of $(queryID, rowID, columnID)$.

5.3.4 Index Protection

If the index contains no sensitive information, the corresponding attribute can be outsourced in plaintext without encryption. The index can thus be built on this attribute. In some cases, the index may be part of the sensitive information to be protected (e.g. personal identifier). Index protection is thus necessary.

However, encrypting the attribute with either AES-CTR or AES-OFB prohibits B⁺-tree indexing, which is one of the most commonly used indexing strategies in a database system. To allow remote indexing and protect the indices at the same time, we employ OPE to encrypt the indices [83].

OPE is an encryption scheme that can perform order operations on ciphertexts in the same way as plaintexts (i.e. $Enc(x) > Enc(y)$ iff $x > y$) and is well-proven to reveal no additional information about the plaintext values besides their order [44, 84]. The use of OPE to encrypt the indices can achieve two important objectives: 1) to allow remote indexing in the cloud for scalability and elasticity and, 2) to leverage the order-preserving property of OPE for efficient B-tree index search because decryption is not required for range check operation.

5.3.5 An Illustrative Example

Figure 5.4: An example showing how an outsourced database record and index are encrypted

Figure 5.4 shows an example of a record of a database table containing three attributes in a database table. Attributes longer than 128 bits (*Remark*) is encrypted with AES-OFB, and otherwise (*ID*, *Name*), encrypted with AES-CTR using K_{db} . Each corresponding attribute seed is shown in the format of

(*databaseID*, *tableID*, *rowID*, *columnID*). Assume the attribute *ID* is used as an index; it is further encrypted by OPE and the output *ID_{OPE}* is appended to the encrypted tuple. While the encrypted tuple is sent to the Cloud, the column *ID_{OPE}* is used directly (without decryption) to create the indices in the Cloud. The encrypted record is stored according to the storage strategy of the DBMS.

5.3.6 Remote Database Modification

As a database can dynamically change over time, a mechanism to support remote modification of the encrypted database is necessary. The major challenge in CypherDB is that users can easily lose track of the attribute seeds being used in encrypting the data. To overcome this challenge, a dynamic database attribute seed management is developed to ensure the uniqueness of the attribute seed used. The key idea is to let users do bookkeeping of the attribute seed table.

For any database management system, CREATE TABLE, INSERT, DELETE and UPDATE are the most commonly used operations in modifying any database. We therefore discuss the working principles of the attribute seed management under these operations in Figure 5.5.

It can be seen that the attribute seeds are used and updated by these operations. As a result, synchronization of the attribute seed table between the user's side and the cloud's side is needed so that both sides have the most updated attributed seed. This attribute seed management can be automated by a software layer, which consists of 234 Line Of Code (LOC)s in our implementation prototype.

- CREATE TABLE
 1. Issue the CREATE TABLE query statement.
 2. Receive *databaseID* and *tableID* allocated by DBMS.
- INSERT
 1. Chooses an unused *rowID* and attribute counter from the attribute seed table.
 2. Encrypts the record with the corresponding attribute seed.
 3. Insert the encrypted record to the Cloud.
- DELETE
 1. Retrieve the *rowID* of the records being deleted from the outsourced database.
 2. Update the attribute seed tables by incrementing the counter value of the corresponding *rowID*.
 3. Delete the records from the outsourced database.
- UPDATE
 1. Retrieve the records that needs to update.
 2. Delete the records from the outsourced database.
 3. Update the attributes in local server.
 4. Re-encrypt the records with an unused *rowID* and attribute counter.
 5. Insert the updated and encrypted records.

Figure 5.5: Algorithm of remote database modification and attribute seed management

5.4 Security Analysis

The security of AES-CTR and AES-OFB is well-proven [85], except that they pose a strong requirement on the encryption seed which must be unique for each datum under a single encryption key; otherwise, the confidentiality of the data may be compromised due to the “two-time” pad attack caused by re-using the same encryption pad. In our proposed encryption scheme, each attribute seed is spatially and temporally unique across the databases for the same database owner, as described in Section 5.3.2. Various database owners have their own unique database encryption keys K_{db} such that the seed uniqueness concern is confined to a single party. It therefore greatly simplifies the attribute seed management and relies on the DBMS to handle the seed uniqueness. Re-encrypting the database with a new encryption key may be necessary when any of the attribute seeds, either the logical schema ID or tuple counters, overflow. These parameters are set to a sufficiently large value to avoid frequent re-encryption. Although these two encryption modes introduce additional parameters (attribute seed) that require special management to maintain its uniqueness, they are more secure than the conventional AES encryption because the encrypted data are now non-deterministic due to the unique encryption seed being used. It means that even if two attributes are of the same value, the encrypted data look completely different.

Our design does not rely on encrypting the query statements. One potential drawback is the information leakage from the query statements. We refer to it as indirect information leakage. Take a query statement like “SELECT NAME, AGE FROM TABLE WHERE ID=‘1234’;” as an example. An adversary can learn from this unencrypted query statement that the user is searching for NAME and AGE with a certain ID number. However, he is unable to learn the exact information because all the query outputs are encrypted. The corresponding information leakage is at most the number of entries satisfying the WHERE clause. Such indirect information leakage can be solved by sending

dummy output results but this is not considered in our study.

The use of OPE to encrypt indices can leak the order of the sensitive information due to the nature of the encryption algorithm. However, we note that this information leakage is inevitable with the use of B^+ tree indexing, even if the indices are encrypted with strong encryption. B^+ tree uses a binary search tree which stores and accesses the indices in ascending/descending order. An adversary can passively observe the storage of these indices or the access pattern in order to learn the order of the indices. Since the use of B^+ tree naturally discloses the order of the indices, the use of OPE can thus achieve better performance (without any decryption) and does not sacrifice any security at the same time.

Chapter 6

Processor Architecture

This chapter describes CypherDB secure processor architecture. The main goal of this processor architecture is to provide architectural support to our proposed look-ahead encryption scheme (see Section 5.2) and protect the privacy of any intermediate data stored in off-chip memory in high performance. This chapter is organized as follows. We first investigate a typical database application to identify the sensitive data that need protection. Based on this investigation, we then present an overview of the architecture, which provides three separate data paths for secure execution. After that, the design of each of these three data paths is discussed. Finally, a query execution example is presented, and the security of this architecture is discussed.

6.1 Database Profiling

Figure 6.1 presents the memory layout of a typical database application which outlines the necessary data in a typical database application. The database records are packed and stored in a format called a payload. This payload contains a record header to describe the features of the record and attribute offset to locate each attribute within a record. Multiple payloads are organized on a database page. Each database page has its own page header and record pointer array. During the execution, the DBMS allocates a segment of heap

Figure 6.1: The memory layout of a typical database application process. The database records are formatted in a structure of database pages where the database pages are stored in buffers allocated in heap memory.

memory, forming a Database Page buffer, to accommodate multiple database pages. It can be seen that all of the aforementioned database data can be classified into three types:

- Attribute data: the database record outsourced by the database owner.
- Metadata: the non-sensitive information such as page header, record offset, payload header and attribute offset that is useful for the DBMS to manage the storage or access of the database records.
- Execution data: the intermediate value generated on-the-fly during program execution stored in heap or stack memory.

In order to evaluate the performance impact of these three types of data accesses during a database query operation, we investigated the off-chip memory access profile, which is reported as having the most impact on execution time when performing database queries [86]. Our investigation is based on executing the 22 queries in TPC-H [75] on SQLite using a cycle accurate simulator, SimpleScalar [87]. Figure 6.2 depicts the breakdown of each type of data contributing to the last-level cache miss, which shows three important observations and insights:

- Almost half of last-level data cache misses are caused by loading the metadata. These data contain non-sensitive information, and thus encryption is not required.
- Execution data are used frequently which results in a high cache hit rate (95%). However, over 14% to 63% of last-level data cache misses are caused by loading these execution data from off-chip memory. These data need to be encrypted at the processor boundary due to the high data reuse profile in the last-level cache.
- Last-level data cache misses on loading the attribute data vary from 2% to 34%. These data are protected by our proposed look-ahead encryption

Figure 6.2: A quantitative analysis of total stall time caused by the last-level data cache miss by executing the 22 queries in TPC-H using SQLite in SimpleScalar. The stall time contributed by the three different types of data: attribute data, metadata and execution data, are measured.

scheme, where encryption latency can occasionally be hidden from the off-chip memory data access (see Section 5.2.1).

6.2 Architectural Overview

Figure 6.3 depicts the architectural model of the CypherDB processor. The key idea behind the CypherDB architecture is to deploy dedicated hardware data paths to separately handle the aforementioned three different types of data commonly found in any database application. These three different types of data are stored in different locations in off-chip memory, which are designated as “compartments”: *Seed Memory Compartment (SeedMem)* for storing encryption seeds, *Secure Memory Compartment (SMC)* for storing execution data, and *Database Page Compartment (DBPage)* for storing attribute data and metadata. Three data paths are designed to load the data into the processor in DBPage and SMC memory compartments securely and efficiently (see Figure 6.3):

Figure 6.3: Proposed architecture of the CypherDB secure processor. Off-chip memory is partitioned into three compartments. Three data paths are set up to execute the three different types of data. The components in grey indicate additional hardware features.

- Data path ①: It is an unmodified data path akin to that used in a conventional processor to process non-sensitive metadata, which causes no processing penalty.
- Data path ②: The attribute data are executed on this data path with the use of our proposed look-ahead encryption scheme. Since registers are the only visible storage for software, the AES encryption engine is therefore placed between the level 1 data cache and the register files.
- Data path ③: All execution data have to follow this data path where the AES encryption engine is placed closest to the chip boundary. The encryption or decryption is done on the entire cache line which is completely transparent to the application software.

Four new special registers are used to store the encryption keys. There is no instruction to read the contents of these registers, which protect these encryption keys from any malicious software. A random number generator is used to create random bit vectors for security purposes. An addition bit (*S_bit*) is appended to each data cache line entry to indicate whether protection is required. DBPage and SMC seed buffers are on-chip memory used to store the attribute seeds and the SMC encryption seeds, respectively. The purpose of these on-chip buffers is to reduce the off-chip memory access of fetching the seed for encryption.

6.2.1 Private Key Registers

There are four encryption keys employed in our secure query processing: on-chip private key, database encryption key, query encryption key and SMC encryption key. Their functionalities are summarized in Table 6.1.

An on-chip private key is written into the on-chip non-volatile register during chip manufacturing. There is no instruction support to modify or extract this key from the register. For the other three encryption keys, there is only

Table 6.1: A summary of the functionality of the four encryption keys used in secure query processing

Encryption Keys	Functionality
On-chip Private Key	Encrypt or decrypt the database encryption and query encryption key in key encapsulation mechanism, and sign the digital certification for secure processor attestation
Database Encryption Key	Encrypt or decrypt the attribute data during secure query processing in our proposed encryption scheme
Query Encryption Key	Encrypt the query result using our proposed encryption scheme
SMC Encryption Key	Encrypt or decrypt the cache line in SMC to protect execution data during query processing

instruction support to store the keys in these on-chip registers but not to extract them. Database and query encryption keys are obtained from the key encapsulation mechanism described in Section 4.2.1. They can be dynamically stored in the on-chip registers with special instructions. A SMC encryption key is generated by the on-chip random number generator at the beginning of a secure query execution.

6.2.2 New Security Instructions

The new instructions and their functionalities to support CypherDB are summarized in Table 6.2. The architecture of these new instructions on OpenRISC (our implementation prototype) is presented in Appendix A.

During the key exchange process between the database owner and the cloud servers, the database encryption key K_{db} and query encryption key K_Q are stored in the on-chip registers via the *load_encKey* instruction. The *load_encKey* instruction takes $E_{PK}(Key)$ as the input which contains either an encrypted database or query encryption keys and decrypts it using the on-chip private key. The *switch_encKey* instruction is used to switch the use of encryption key, between database encryption key and query encryption key, along data path ②. The *verify_encKey* instruction verifies the certifi-

Table 6.2: New Instructions in CypherDB

New Instructions	Description
start_cypherdb_query	Allocate SMC and generate SMC encryption key, and initiate the encryption seed in SeedMem
end_cypherdb_query	Dis-allocate SMC and destroy SMC encryption key
l.seed	Load/Store attribute seed for attribute decryption/encryption
l.shift	Shift the encryption pad according to the value indicated in the operand of the instruction
l.sload/l.sstore	Decrypt/encrypt attribute data with the encryption pad
set_sr	Set a security register value which stores the parameter to be used in the proposed architecture
load_encKey	Load the encrypted database key and encrypted query key; and decrypt the keys with private key and store in on-chip registers
switch_encKey	Switch the encryption key used in the AES encryption engine along data-path ② between database key and query key
verify_encKey	Verify the database key and query key by checking the digital signature $Sig(E_{PK_C}(K_{db}), E_{PK_C}(K_Q))$ for on-chip verification of the query encryption key
sign_cert	Sign a certificate with on-chip private key for processor attestation

cate $Sig(E_{PK}(K_{db}, K_Q))$ by checking the signature, decrypting $E_{PK}(K_{db}, K_Q)$ and comparing the K_{db} and K_Q with the on-chip key value. The *sign_cert* instruction uses the on-chip private key to generate a digital certificate for processor attestation.

To start the query execution, the *start_cypherd_query* instruction is executed to perform three tasks: 1) allocate the SMC memory, 2) generate a per-query SMC encryption key and 3) initiate the encryption seed in SeedMem. The *l.sload*, *l.sstore*, *l.shift* and *l.seed* instructions are used to realize our proposed encryption scheme. The *set_sr* instruction is used to write or read some parameters in our proposed architecture to security registers (e.g. SMC and SeedMem memory boundary address). To end the query execution, the *end_cypherd_query* instruction is executed to free the SMC memory and destroy the SMC encryption key.

6.3 Virtual to Physical Memory Mapping

The concept of memory compartment is realized in virtual and physical memory addressing space as shown in Figure 6.4. There are two contiguous memory blocks separately allocated to be the SMC and SeedMem while the other memory space can be freely allocated to the application process. The starting and ending address of these two contiguous memory blocks (*SMC_start*, *SMC_end*, *SMC_seed_start*, *SMC_seed_end*) can be stored in the security registers via the *set_sr* instruction.

To make use of these memory compartments in physical memory, the database application process first sets up its own virtual memory space. The DBMS can still use the stack and heap memory as normal except that these memory regions in virtual memory are mapped to the SMC in physical memory. Meanwhile, the DBPage buffer and the array of attribute seed can be freely allocated to other physical memory space. Two custom memory allocators are used to allocate the heap memory separately towards execution data and

Figure 6.4: The virtual to physical memory mapping of the three memory compartments. The DBPage buffer is freely mapped to any available physical memory space while the remaining stack and heap memory is allocated to a contiguous memory block as SMC. A contiguous memory block is also allocated to store the encryption seed as SeedMem.

database pages, which we define as *smalloc* and *dpmalloc*, respectively. The *smalloc* allocates the memory in SMC for execution data while the *dpmalloc* freely assigns the non-SMC memory region to database pages. The memory allocation mechanism of *smalloc* and *dpmalloc* is the same as the *malloc* in GNU C [88] library except that *smalloc* and *dpmalloc* monitor the memory usage of different memory regions.

The SMC encryption seed is stored in SeedMem. The cache controller fetches the encryption seed from SeedMem to the SMC seed buffer to encrypt/decrypt the cache line in SMC. On the other hand, the attribute seeds are stored in the heap memory in an array so that they are stored consecutively in distinct (non-consecutive) memory pages. The attribute seeds have to be explicitly loaded into the DBpage seed buffer or written back to the off-chip memory via the *l.seed* instruction.

6.4 Data Protection

The attribute data are protected by our proposed look-ahead encryption scheme. It uses the *l.seed* instruction to load the encryption seed into an on-chip register *Reg_{seed}* and compute the encryption pad, *l.sload/l.sstore* to load (or store) and decrypt (or encrypt) the data, and uses *l.shift* to shift the encryption pad accordingly (see Section 5.2). As this *l.shift* instruction is used every time to shift the encryption pad, it introduces additional instruction and execution overhead. To alleviate these drawbacks, a hardware shifter is used to shift the encryption pad automatically after each *l.sload/l.sstore* instruction based on the observation that the attribute data are usually accessed in sequence from the first byte to the last byte.

Figure 6.5 depicts the flow chart of the hardware shifter. The key idea of this hardware shifter is to perform encryption pad shifting implicitly after each *l.sload/l.sstore* instruction but allows explicit shifting using *l.shift* and *l.seed* instructions. The hardware shifter is able to implicitly shift the encryption pad in accordance with the granularity of the *l.sload/l.sstore* instruction. *l.shift* can override *l.sload/l.sstore* to shift the encryption pad to the value indicated in the operand. The total number of bits being shifted is recorded and stored in a special register *Reg_{shift}*. When the encryption pad is shifted to the end ($Reg_{shift} \geq 16$), a new encryption pad is generated for the use of AES-OFB encryption. To decrypt/encrypt another attribute datum, the *l.seed* instruction can generate a new encryption pad and reset *Reg_{shift}* to zero.

6.4.1 DBPage Seed Buffer

The purpose of the DBPage seed buffer is to store a subset of attribute seeds to on-chip memory, so that the attribute seeds can be served faster when needed. As described in Section 5.3, each attribute datum requires an attribute seed in our proposed look-ahead encryption scheme. This attribute seed is formulated with the use of logical schema, which can be “generated” during program

Figure 6.5: A flow chart describing the hardware shifter that implicitly shifts the encryption pad for different granularity of *l.sload/l.sstore* instructions. The *l.shift* and *l.seed* can override the operation of hardware shifter to explicitly set the value of *Reg_{shift}*.

execution. The only seed component requiring extra storage is the record counter, which is shared by all attributes within a record. A mapping table is used to take *rowID* as the input to retrieve the corresponding record counter.

Without the DBPage seed buffer, off-chip memory access of the record counter is necessary to encrypt/decrypt each attribute being stored/loaded. One of the key ideas of our proposed encryption scheme is to pre-compute the encryption pad so as to hide the encryption latency from the program execution. An off-chip memory access of the record counter virtually lengthens the latency of the encryption pad computation by an off-chip memory access latency. This requires *l.seed* to be issued even earlier which makes the task of encryption latency hiding much more difficult (see Section 5.2.1).

The DBPage seed buffer can reduce the frequency of off-chip memory access by keeping a number of record counters in the buffer. As fetching the record counters in an on-chip buffer can be achieved in a few processor cycles, it is much faster than off-chip memory access which takes hundreds of cycles to complete. Figure 6.6 depicts a flow chart of our proposed encryption scheme with the assistance of a DBPage seed buffer. We can see that *l.seed* is used to fetch the record counter in SeedMem. This operation is redirected to the DBPage seed buffer. The DBPage seed buffer calls a function *seed(rowID)* to search for the record counter locally. If the record counter is stored in the buffer, it can be used directly to compute the encryption pad; otherwise, the record counter is fetched from the off-chip memory. The execution flow along data path ② is very similar to data path ①, except that the data being fetched/stored need to wait for the encryption pad until it is ready and undergoes decryption/encryption (XOR operation) before being written into cache/off-chip memory.

Figure 6.6: A flow chart describing the execution flow of data path 1 and data path 2 with the assistance of a DBPage seed buffer. The DBPage buffer can reduce the frequency of off-chip memory access to fetch the attribute seed.

6.5 Execution Protection

We employ the cache line encryption technique proposed in [36] and use a random SMC encryption key for each query session requested by the database owner. This design is based on the following observations:

- From the perspective of the database owner, execution data are only useful during query processing but not after the query result is generated. The execution data can thus be discarded after each query execution. Generating a random per-query session key is thus viable in our architectural design.
- As shown in our architectural profiling of a database application (see Section 6.1), the execution data in the last-level data cache are frequently reused. Encrypting these data at the processor boundary can reduce encryption frequency and thus alleviate the encryption penalty.
- As the caching system is transparent to the application software, encrypting the cache line can thus reduce the necessary software modification to protect execution data.

6.5.1 Cache Line Encryption

Algorithm 2 presents the pseudo-code of the cache line encryption/decryption in write-back mode. Write back cache is employed in our architecture so that each cache line in the SMC is decrypted/encrypted using AES-CTR when it is loaded into (or written back from) the processor. AES-CTR is used to offload the encryption penalty from the critical path by parallelizing the seed encryption and off-chip memory access of the cache line. As the last-level cache line usually consists of multiple 128-bit cache blocks, multiple hardware encryption engines can be operated in parallel on those cache blocks. Thus, the entire cache line can be decrypted/encrypted at most in a single cycle.

Algorithm 2 Pseudo-code of cache line encryption and decryption

```
1:  $l$  = cache line size (bytes)/16 /* to calculate the number of cache block
   within a cache line */
2:
3: /* Decryption of an incoming encrypted cache line */
4: function DECRYPTCACHELINE( $ECL_{in}$ ,  $K_{SMC}$ ,  $SMC\_IV$ ,  $TAG$ ,  $count\_val$ )
5: Input:  $ECL_{in}$  = encrypted cache line to be fetched,  $K_{SMC}$  = SMC en-
   cryptation key,  $SMC\_IV$  = initialization vector for SMC encryption,  $TAG$  =
   Tag portion of physical memory address,  $count\_val$  = the counter value of
   SMC encryption seed
6: Output:  $CL_{out}$  = decrypted cache line
7:   for  $1 \leq i \leq l$  do /* decrypting each cache block in parallel */
8:      $CL_{out}[i] = ECL_{in}[i] \oplus AES[SMC\_IV \parallel TAG + i \parallel count\_val, K_{SMC}]$ 
9:   end for
10:   $CL_{out} = CL_{out}[1] \parallel CL_{out}[2] \parallel \dots \parallel CL_{out}[l]$ 
11:  return  $CL_{out}$ 
12: end function
13:
14: /* Encryption of outgoing cache line */
15: function ENCRYPTCACHELINE( $CL_{in}$ ,  $K_{SMC}$ ,  $SMC\_IV$ ,  $TAG$ ,  $count\_val$ )
16: Input:  $ECL_{in}$  = cache line to be evicted,  $K_{SMC}$  = SMC encryption key,
    $SMC\_IV$  = initialization vector for SMC encryption,  $TAG$  = Tag portion
   of physical memory address,  $count\_val$  = the counter value of SMC en-
   cryptation seed
17: Output:  $CL_{out}$  = encrypted cache line
18:   for  $1 \leq i \leq l$  do /* encrypting each cache block in parallel */
19:      $ECL_{out}[i] = CL_{in}[i] \oplus AES[SMC\_IV \parallel TAG + i \parallel count\_val, K_{SMC}]$ 
20:   end for
21:   $ECL_{out} = CL_{in}[1] \parallel CL_{in}[2] \parallel \dots \parallel CL_{in}[l]$ 
22:  return  $ECL_{out}$ 
23: end function
```

To ensure security, a unique seed is generated for each cache line for the entire SMC memory space. This unique cache line seed, $CLSeed$, is produced by concatenating two fields: 1) the tag portion of the physical memory address TAG , and 2) a unique counter value $count_val$, while the remaining bit is padded with a random bit vector SMC_IV :

$$CLSeed = SMC_IV \parallel TAG \parallel count_val \quad (6.1)$$

The $count_val$ associated with each cache line is stored in $SeedMem$, and is incremented every time the cache line is written back to off-chip memory. The TAG and SMC_IV are obtained from on-chip address bus and security register, respectively.

To obtain the $count_val$ from $SeedMem$, the cache controller uses the physical address of the cache line ($ADDR_{CL}$) and the $SeedMem_start$ stored in the security register to calculate the physical address of the $count_val$ ($ADDR_{SMCseed}$) to be fetched as shown in the following equation:

$$\begin{aligned} bit_shift &= \log_2(cache\ line\ size(bytes)) - \log_2(count_val_size(bytes)) \\ ADDR_{SMCseed} &= ((ADDR_{CL} - SMC_start) \gg bit_shift) + SeedMem_start \end{aligned} \quad (6.2)$$

We can see that it first takes the offset of the cache line address in the SMC and replaces the base address from SMC_start to $SeedMem_start$. Then, the offset value is divided by the cache line size so that the offset value is now incremental towards consecutive cache lines. As the $count_val$ can be larger than a byte, it is further multiplied by the size of $count_val$. The size of the $SeedMem$ is therefore proportional to the SMC size, cache line size and $count_val$ size as shown in the following equation:

$$SeedMem_{size} = (SMC_end - SMC_start) \gg bit_shift \quad (6.3)$$

For a computing system which uses a 64-byte cache line size and 2-byte *count_val* as encryption seed, the memory overhead caused by SeedMem is around 3%. This shows that the memory overhead caused by SeedMem is acceptable.

6.5.2 SMC Execution Setup

To allow protection of execution data along data path ③, an execution environment has to be setup which mainly involves four procedures: *STARTQUERYEXEC*, *ENDQUERYEXEC*, *FETCHCACHELINE* and *EVICTCACHELINE* as described in Algorithm 3.

To start a secure query execution, *STARTQUERYEXEC* function is called to initialize the *SMC_encryption_key*, *SMC_IV*, and to set up the SMC by obtaining its starting address *SMC_start* and ending address *SMC_end*. All *count_val* stored in off-chip memory is also initialized to zero while the SMC seed buffer entries are all invalidated. The initialization of *count_val* is to indicate that the cache lines in the SMC are all unencrypted. Our algorithm ensures that the encrypted cache lines always have non-zero *count_val*.

To fetch a cache line from off-chip memory, the cache controller calls *FETCHCACHELINE* to load the corresponding *count_val* and decrypt the cache line in the SMC. All the fetched cache lines in the SMC assert the *S_bit* in the data cache entries, but only those encrypted cache lines undergo decryption. The unencrypted cache line can be directly stored in the on-chip data cache. *EVICTCACHELINE* is called in two scenarios: to write back or to flush the cache lines. Both operations evict the cache lines to off-chip memory. To encrypt the cache line, the *count_val* is incremented to avoid the *CLseed* from being reused. The cache lines with *S_bit* asserted are encrypted before writing them to the off-chip data buses.

The secure query execution is ended with function *ENDQUERYEXEC*. This erases the *SMC_encryption_key*, *SMC_IV* and abandons the SMC by

setting the corresponding registers to zero. However, the memory content in the SMC does not need to be erased because it is protected by encryption. As the *SMC_encryption_key* is destroyed, the content cannot be recovered. This can save the time-consuming off-chip memory clean-up operation.

Algorithm 3 Pseudo-code of execution setup in the SMC

```

1: randomBit(k) = obtain a k-bit vector from random number generator
2: fetchCountVal(addr) = obtain the count_val with address equals to addr
3: writeCountVal(addr, count_value) = store the count_val to SeedMem with
   address equals to addr
4: S_bit(TAG) = the secure bit of an on-chip cache line with the tag address
   equal to TAG
5:
6: /* Start of the secure query execution */
7: function STARTQUERYEXEC
8:   SMC_encryption_key = randomBit(128)
9:   SMC_IV = randomBit(128 – lengthOf(TAG) – lengthOf(count_val))
10:  Obtain SMC_start & SMC_end
11:  Set all count_val = 0
12:  If SMC encryption seed buffer exists, invalidate all entries
13: end function
14:
15: /* Fetching an encrypted cache line from off-chip memory */
16: function FETCHCACHELINE(CLRAM, CLCACHE)
17: Input: CLRAM = cache line fetched from off-chip memory
18: Output: CLCACHE = cache line to be stored in on-chip cache
19:   if (SMC_start ≤ addr ≤ SMC_end) then
20:     S_bit(tag) = 1
21:     count_val = fetchCountVal(addr)
22:     if count_val ≠ 0 then
23:       CLCACHE = DECRYPTCACHELINE(CLRAM)
24:     else
25:       CLCACHE = CLRAM
26:     end if
27:   else
28:     CLCACHE = CLRAM
29:   end if
30: end function

```

Algorithm 3 Pseudo-code of execution setup in the SMC (continued)

```
31: /* Evicting a cache line to off-chip memory */
32: function EVICTCACHELINE( $CL_{CACHE}$ ,  $CL_{RAM}$ )
33: Input:  $CL_{CACHE}$  = cache line to be evicted from on-chip cache
34: Output:  $CL_{RAM}$  = cache line to be stored in off-chip memory
35:   if  $S\_bit(tag) = 1$  then
36:      $count\_val = fetchCountVal(addr) + 1$ 
37:      $writeCountVal(addr, count\_val + 1)$ 
38:      $CL_{RAM} = ENCRYPTCACHELINE(CL_{CACHE})$ 
39:   else
40:      $CL_{RAM} = CL_{CACHE}$ 
41:   end if
42: end function
43:
44: /* End of the secure query execution */
45: function ENDQUERYEXEC
46:   Set  $SMC\_encryption\_key = 0$ 
47:   Set  $SMC\_start = SMC\_end = 0$ 
48:   Set  $SMC\_IV = 0$ 
49: end function
```

6.5.3 SMC Seed Buffer

Because *count_value* is the only component that requires off-chip memory storage in *CLSeed*, the SMC seed buffer is used to store this value on-chip so as to minimize the off-chip memory access. Similar to the DBPage seed buffer, the purpose of the SMC seed buffer is to store a portion of *count_value* on-chip so that they can be served fast when needed. Figure 6.7 presents the flow chat of the cache line encryption with the use of a SMC seed buffer. The overall execution can be divided into three types: normal execution, cache line decryption and cache line encryption.

In normal execution, a cache line is loaded into the on-chip cache or written back to the off-chip memory without any encryption/decryption. This execution flow is to serve any data execution outside the SMC or unencrypted cache line in the SMC.

FETCHCACHELINE function is called to decrypt the cache line, if necessary, when a cache line is fetched from the SMC. The *fetchCountal* function

first tries to retrieve the corresponding *count_value* from the SMC seed buffer. The best case is that the *count_val* is found in the SMC seed buffer. This value can be used to pre-compute the encryption pad. The computation of encryption pad can thus be made parallel with the off-chip memory access of fetching the cache line. If the *count_val* is unfortunately not found in the buffer, it has to be fetched from the off-chip memory before fetching the cache line. After that, the encryption pad can still be pre-computed before the cache line is loaded. However, this causes additional off-chip memory access to fetch the *count_val*. The worst case scenario is that there is a conflict miss in the SMC seed buffer. Due to the conflict miss, the dirty seed (the seed being incremented) has to be first written back to the off-chip memory before fetching the correct *count_value* from it. This results in two additional off-chip memory accesses.

EVICTCACHELINE function is called to encrypt a cache line when it is to be evicted to the SMC. Again, the *fetchCountal* function is used to get the *count_value* from the seed buffer. As the cache line can be evicted immediately, the computation of the encryption pad has nowhere to hide. The cache controller has to wait for the encryption pad computation before evicting and encrypting the cache line. Similar to the cache line decryption, the best case is to have the *count_value* stored in the buffer which introduces encryption latency only. The worst case is to have a conflict miss, which causes two additional off-chip memory access latencies plus encryption latency. Nevertheless, the performance penalty caused by the cache line encryption can be greatly alleviated with the use of an on-chip victim buffer. A victim buffer can be used to hold the cache lines being written to the off-chip memory. In other words, it postpones the operation of writing back the cache line. This provides more timing margin to write back an incremented *count_value*, load the correct *count_value* and encrypt *CLseed* to generate the encryption pad.

Figure 6.7: A flow chart describing the cache line encryption/decryption execution flow of data path 3 with the assistance of a SMC seed buffer

6.6 Query Execution Example

The query processing over an encrypted database is protected by executing different kinds of data along the three data paths in our design. In this section, we use a query “SELECT Name, Remark FROM table WHERE $ID_{OPE} < 24675431$ ” (see Figure 5.4) as an illustrative example to demonstrate how these three data paths provide protection for query processing. In this query, the operation involves a B^+ tree search on the index (ID_{OPE}), copying the relevant attributes ($Name$, $Remark$) to some temporary variables ($dataBuf$ in Figure 6.8) and returning back the query result ($dataBuf$) to the user. Figure 6.8 shows the data paths and the involved hardware components of this query operation in CypherDB. The data flow is explained as follows.

Before the start of query execution, the database application follows two steps to set up the secure execution environment: 1) load and verify the database and query encryption keys via the *load_encKey* and *verify_encKey* instructions respectively, and 2) call *STARTQUERYEXEC* function to generate the SMC encryption key and set up the SMC via the *start_cypherDB_query* and *set_sr* instructions (see Section 6.5.2). The database encryption key is set as the default encryption key along data path ②.

The query execution starts by following data path ① to process some meta-data and search for the index as shown in Figure 6.8(a). Note that the index is encrypted with OPE such that no decryption is needed for the index search. For a matched condition ($ID_{OPE} < 24675431$), the corresponding encrypted attribute has to be processed along data path ② as illustrated in Figure 6.8(b). Before accessing the encrypted attribute, the seed value is first loaded into the seed register Reg_{seed} via the *l.seed* instruction. The encryption pad is generated using a database encryption key with the value stored in Reg_{seed} and the attribute counter in the DBPage seed buffer. To access the encrypted attribute, the encrypted data are loaded into the processor via the *l.sload* instruction.

The result of the operation has to be stored in the SMC. It is first stored

Figure 6.8: Diagrams demonstrating how the data are executed during a query processing example. The *Encrypted Data* are loaded along data path 2 and copied to *dataBuf*. The *dataBuf* is protected by cache line encryption along data path 3. The data paths used are highlighted in BLUE, the encryption keys used are highlighted in RED and the encryption seeds used are highlighted in GREEN. (a) Execution along data path 1 for non-sensitive data. (b) Execution along data path 2 to load the encrypted attribute data from DBPage memory. (c) Execution along data path 3 where the cache line is encrypted before evicting it from the processor. (d) Execution along data path 2 for query output protection where a query key is used to encrypt the query result

in on-chip cache until the cache line needs to be evicted from the processor. As seen in Figure 6.8(c), prior to storing the modified cache line (cache line 3) back to the off-chip memory, the whole cache line is encrypted with the SMC encryption key and an incremented counter value in the SMC seed buffer together with the physical address along data path ③. At the end of the query processing, the data stored in *dataBuf* has to be sent back to the user as a query output. This has to be done in three steps (see Figure 6.8(d)):

- Load the encrypted cache line along data path ③;
- Switch the encryption key on data path ② from the database encryption key K_{db} to query encryption key K_Q via the *switch_encKey* instruction.
- Encrypt the query output using K_Q and our proposed encryption scheme (see Section 5.3.3) along data path ②.

6.7 Discussion

6.7.1 SMC Allocation

In CypherDB architecture, it requires a contiguous physical memory space to allocate the SMC. This is to assist the hardware protection circuit to identify the SMC using the boundary addresses. There are various methods to allocate a secure memory compartment.

One approach to obtain this contiguous physical memory region is to use Contiguous Memory Allocator (CMA) [89]. CMA is developed to allow big physically contiguous memory allocations which have already been used to support x86 and ARM architecture. It was originally designed to support device drivers that use a large physical memory. This can however be leveraged to allocate the SMC in our system where the kernel can thus map the heap and stack memory in virtual memory space to this physical contiguous memory

region. The disadvantage of this memory allocator is that it has to be initialized early at boot time, which means that the SMC and SeedMem have to be defined at the system bootup. Regarding the large Dynamic Random Access Memory (DRAM) in modern computing systems, a sufficiently large SMC can be set up at boot time to support the secure query execution. As this approach relies on the kernel to set up the SMC, a secure launch or boot technology [11] has to be employed to launch the kernel module that is responsible for the memory allocation and address mapping.

Another approach is to employ some existing hardware-assisted isolated execution solutions such as Iso-X [27] or Bastion [33]. This approach has two advantages. First, it can dynamically set up isolated memory compartments at run-time. Second, the isolation among memory compartments is enforced at hardware level which provides better protection to the memory compartments. This approach uses a hardware-assisted mechanism to set up the memory compartment and perform the virtual to physical memory mapping, which does not rely on any security kernel. For example, Iso-X uses a bit vector to indicate whether each physical page belongs to any compartment. This bit vector is stored in a reserved memory region at system bootup and is only accessible by the processor. When a new physical memory page is allocated to a compartment, the corresponding bit in the bit vector is asserted. Similarly, this bit vector can be leveraged to specify whether the physical page belongs to the SMC and SeedMem in our design so that the SMC and SeedMem can be dynamically allocated.

6.7.2 Comparison with Alternative Approaches

One proposed solution to protect the intermediate value generated on-the-fly during execution is to use the technique of Information Flow Tracking (IFT) [90, 91]. This method taints all the intermediate data that have an implicit and explicit relationship with the sensitive information. For example, architectural

supported Dynamic Information Flow Track (DIFT) [90] is proposed to identify spurious information flows from the I/O and monitor their use so as to protect the program execution from this spurious information. Although IFT provides a solution to track the data, it does not provide any solution to protect spurious or sensitive information. How to protect the privacy of the execution data in our design remains an unknown and challenging question. Nevertheless, IFT can potentially exhibit false positives or false negatives which make it still challenging to put it into practice [92].

Cache line encryption using AES-CTR has been extensively studied in [25, 36]. However, these approaches use cache line encryption to protect the entire memory space where application and data are encrypted as a whole. In [50], it is suggested to encrypt/decrypt the user’s data using a TSM and store it in a TSM buffer, while the TSM buffer is protected by cache line encryption. This however introduces an additional encryption/decryption layer to re-encrypt the encrypted user’s data into application data, which can potentially result in tremendous performance overhead in a database application. For example, with a limited size of DBPage buffers, the database pages, which contain the user’s data, have to be frequently swapped in/out from/to the persistent storage. This requires frequently re-encrypting the user’s data into application data (or vice versa). Our proposed lookahead encryption scheme can effectively eliminate this re-encryption layer. Also, the separation of the memory compartments can result in a smaller SMC. A smaller SMC requires less encryption seed. It turns out that it not only reduces memory overhead of storing the encryption seed in the SeedMem but also minimizes the performance overhead. In an ideal case where the SMC seed buffer can accommodate all SMC encryption seeds, the performance overhead caused by cache line encryption can be extremely small.

6.7.3 Extension on Speculative OoO multicore processor

Speculative execution allows instructions to be executed before it is known whether they are needed. A common form of speculative execution is based on branch prediction so that the non-yet-determined instructions (i.e. instructions after a branch) can be executed ahead of the branch. Although the instructions are executed, they are only committed if the branch prediction is true.

Under our proposed architecture, the look-ahead encryption which involves *l.seed*, *l.shift*, *l.sload* and *l.sstore* can be speculatively executed and committed to only when the prediction is correct. More resources (e.g. encryption pad buffer) are needed to store the speculatively executed result. The description of each instruction is as follows:

- *l.seed/l.shift*: The encryption pad can be speculatively computed/shifted to store the pad in a temporary data buffer. The result is committed to by overwriting the encryption pad used in the pipeline.
- *l.sload/l.sstore*: The incoming data can be decrypted and stored in the load store unit (LSU). The outgoing data can be encrypted and stored in a store buffer. Similar to normal load/store instructions, the result is committed to by writing the value into register/off-chip memory until the prediction is proven to be true.

The other security instructions (e.g. *start_cypherdb_query*) do not support speculative execution. These instructions are mainly used to set up or quit the secure execution which occurs infrequently during program execution. Therefore, speculatively executing them is expected to have little performance improvement but is costly on reverting the execution if the branch prediction is wrong.

The attribute encryption can be extended to an out-of-order (OoO) processor. The problem associated with OoO execution is due to the fact that

the instruction executed in the processor pipeline is not of the same order as compiled in software. In our attribute encryption model, the additional instructions are assumed and supposed to be issued in the following order: 1) l.seed (to generate the encryption pad), 2) l.shift (optional, to shift the encryption pad) and 3) l.sload/l.sstore (to load and decrypt/encrypt the data). These instructions do not necessarily need to be issued consecutively but happen to follow the above sequence within a piece of code. Hazards occur when the execution is not in accordance with the above sequence:

- The l.sload/l.sstore instruction is executed before the l.seed/l.shift instructions
- The l.shift instruction is executed before the l.seed instruction

To tackle these hazards, a similar scoreboard approach can be used to figure out the above dependency in hardware. That is to say l.sload/l.sstore will not be executed when l.shift/l.seed has not completed its execution. Also, l.shift will not be executed when l.seed has not completed. The dependency among these instructions can be identified in hardware through setting priority for the decoded signal in the instruction decoder. This signal priority information will be stored in the scoreboard to avoid the hazard.

In multi-core systems, the process may run using different cores from time to time. If the OS migrates the process between cores, the on-chip private key has to be migrated to another core as well. Under our current architecture, the database and query encryption key can be migrated by first destroying the keys in the core and then loading the encrypted database and query encryption key to another core (i.e. use *load_encKey* instruction to load $E_{PK}(K_{db})$ and $E_{PK}(K_Q)$ into the core to be migrated). A better approach would be to perform an on-chip key migration. Since the cores all reside within a single chip (i.e. within the protection boundary), the encryption key can actually be transferred from one core to another using the on-chip bus. In this case, some ISA design may be needed to support such operation.

6.8 Security Analysis

CypherDB architecture employs three different encryption keys for various encryption purposes: 1) database encryption key to encrypt an outsourced database, 2) query encryption key to encrypt query results, and 3) SMC encryption key to encrypt cache lines which contain sensitive execution data. These encryption keys are stored securely in on-chip key registers. The advantage of using three separate encryption keys is two-fold. First, it isolates the encryption protection of attribute data and on-the-fly execution data by employing different encryption keys. Even if one can break the encryption in the SMC, the most critical secret - the database encryption key K_{db} is still safe from tampering. Secondly, the uniqueness of the seed used in database encryption can remain “unpolluted” regardless of the execution in the SMC. This makes the re-encryption of the attribute data with a new seed value become data-update dependent rather than program-execution dependent.

The protection of execution data is achieved by encrypting each cache line in the SMC using AES-CTR encryption. As discussed in Section 5.4, the security strength of AES-CTR relies on the uniqueness of the encryption seed being used. In our proposed architecture, each cache line uses $CLSeed$ as the encryption seed. This seed is spatially unique among the cache lines with the use of the TAG portion of the physical address, and is temporally unique with the use of an incremental counter value $count_val$. The last component of $CLSeed$ is SMC_IV . It is an initialization vector to make the $CLSeed$ even more unpredictable as recommended in [93]. The use of a random SMC encryption key per query session can also refresh the spatial uniqueness and temporal uniqueness of the physical address and incremental counter. By having a per query session SMC encryption key and a sufficiently larger incremental counter, re-encryption of the execution data in the SMC can be avoided.

To crack database encryption, an adversary can send a forged seed value to the seed register in order to perform a “two-time-pad” attack. This is however

outside the scope of this thesis because such attack has to alter the program execution which can be protected by [34]. Also, it will also result in an incorrect database operation, which violates our curious-but-honest administrator threat model.

Another possible attack can be performed by sending a forged SMC boundary address (altering the *SMC_start* and *SMC_end* value in *set_sr* instruction) to the processor so that the execution data are no longer protected by cache line encryption. This can be achieved by tampering with the setup process of the SMC. However, it is strongly prohibited due to the protection of the securely launched kernel module or the hardware assisted memory compartment setup (see Section 6.7.1). The detailed security analysis of these two approaches can be found in [25, 27]. Alternatively, one can also modify the address value in *set_sr* instruction even though the SMC is securely set up. Again, since the program execution of the DBMS can be protected by [34], such attack is also impossible.

Chapter 7

Implementation

We implemented the look-ahead encryption scheme and the CypherDB secure processor on a FPGA to validate and evaluate our designs. We also used a concrete example of a database application (SQLite [82]) to investigate the practicability and performance impact of our proposed solution. The hardware design is specified in Verilog and is synthesized using the ALTERA Quartus 14.0 design tool. The processor runs at 50MHz on the DE2i-150 board with a Cyclone IV SoC FPGA with 64MB off-chip Synchronous Dynamic Random-Access Memory (SDRAM). The SQLite is written in C language and is compiled using the OpenRISC-specific GCC compiler. The database software is able to run on top of Linux on our implemented FPGA platform. This chapter discusses the implementation detail of our design. We first give an overview of our hardware design. Then we describe our implementation of the CypherDB secure processor in more detail. Finally, the realization of the look-ahead encryption scheme on SQLite is further discussed.

7.1 Overview

Our implementation is based on the OR1200 core from the OpenRISC project. The OR1200 core is a simple 4-stage pipeline 32-bit RISC processor, where the EX and MEM stages of a typical MIPS processor are combined into one stage.

The OpenRISC project was chosen as our implementation prototype because it provides comprehensive simulation tools and debugging support for development. It is also supported by a 32-bit GNU toolchain to compile bare-metal applications and Linux applications using newlib and uClibc libraries, respectively. The toolchain support can compile the database application program to be run on Linux, which abstracts the file management layer to store the database.

Figure 7.1: An overview of CypherDB secure processor implementation. The grey components are the additional security modules implemented in our design. A performance counter (shaded) is also implemented to measure the evaluation metrics in hardware.

Figure 7.1 illustrates our CypherDB secure processor implementation. An encryption module and a switch module are implemented inside the processor core to realize the look-ahead encryption scheme. The switch module is used to switch from data path ① to data path ② and vice versa.

The SMC seed buffer and cache line encryption module are implemented to encrypt/decrypt the cache line in the SMC. This sets up the protection along data path ③. SeedMem Init Unit is used to initialize SMC encryption seeds (*count_val*) in SeedMem.

Because the SDRAM is not large enough to accommodate the database to

be executed in our experiments, the databases to be executed are stored in a SD card and a SD card controller is implemented accordingly. As the OpenRISC project does not provide SD card controller implementation support, the implementation in [94] is employed. A self-developed performance counter is also implemented to measure the evaluation metrics in hardware.

7.1.1 Impact on the Processor Pipeline

Figure 7.2: A diagram showing the impact of our implemented modules on the processor pipeline. The grey components represent our additional security modules for the CypherDB secure processor.

Figure 7.2 shows how the additional security modules of our design influence the processor 4-stage pipeline. The security modules that are used to implement the look-ahead encryption scheme are at the EX stage and the data path selection module is at the WB stage. The SMC encryption modules are at the EX stage (or MEM in a traditional 5-stage pipeline), which are operating between data cache and off-chip memory. As a result, the operation latency of these security modules could have a direct impact on the processor pipeline.

Figure 7.3: A block diagram of the CypherDB secure processor core implementation

7.2 Processor Core

The goal of our processor core design is to support the look-ahead encryption scheme. Figure 7.3 depicts a high-level overview of our secure processor core implementation. The blocks in grey are the original OpenRISC components whereas the white blocks, the encryption module and switch module, are the additional modules implemented for our look-ahead encryption scheme. Our modification to OpenRISC is minimal - mainly two components are modified in our system: (1) instruction decoder - to identify the custom secure instructions, and (2) Load Store Unit (LSU) - to buffer the data for encryption/decryption and notify the encryption module about the arrival of data.

Our design aims to minimize the interference with the processor pipeline, so that the processor can operate at its maximum frequency. We therefore avoid integrating the encryption Finite State Machine (FSM) into the processor control unit. Instead, a separate encryption control unit is built which takes the secure instructions and processor states as inputs. The output of this control unit is an encryption stall to the processor pipeline. The only combinational logic added to the OpenRISC pipeline is the switch module which is designed to have minimal logic delay. In this section, each additional component of the encryption module is separately described in detail.

7.2.1 Encryption Engines

AES is one of the most widely used type of symmetric encryption which has outstanding performance in hardware [63, 95]. We employ the open source AES core [1] released in the OpenCore Community [96]. The implemented 128-bit AES engine can complete the encryption in 12 clock cycles at maximal 160MHz clock frequency on our FPGA platform.

Figure 7.4 shows the block diagram of the implemented AES engine. The encryption engine takes a 128-bit key for key expansion, and 128-bit data input to perform 10 rounds of permutation, which consists of four transformation

Figure 7.4: The AES encryption engine employed in our implementation [1]

functions: AddRoundKey, BytesSub, ShiftRows, and MixColumn. Two separate encryption engines for load and store data paths are implemented to avoid any resource contention.

7.2.2 Encryption Controller

Figure 7.5: Our implementation of the encryption controller. It controls the encryption in CTR or OFB mode by switching the data input to the encryption engines between the attribute seed buffer and the encryption pad buffer.

Figure 7.5 depicts the implemented circuit of the encryption controller, which serves three purposes. First, it provides an asynchronous interface to the AES encryption engine since the encryption engine can be operating at a different frequency than the processor core. Second, it consists of an encryption pad buffer (*encPad*) to store the encryption output from the encryption engine. This buffered value is either forwarded to the encryption pad shift module or to the switch module. Last, it formulates the attribute seed and controls the data input to the encryption engine. The control unit decodes the *l.seed* instruction to store the attribute seed value in an attribute seed buffer from the register file. It selectively chooses the attribute seed buffer or encryption pad buffer to be encrypted to support AES-CTR and AES-OFB encryption, respectively.

7.2.3 Encryption Pad Shift Module

The purpose of the encryption pad shift module is to shift the encryption pad according to the flow chart described in Section 6.4, which provides three different operations: 1) shifting the encryption pad according to the value specified in the *l.shift* instruction, 2) shifting the encryption pad according to the *l.sload/l.sstore* granularity, and 3) storing the newly generated encryption pad to the shift registers.

As the shift module is within the processor pipeline, its operation latency is critical to the overall performance. The most performance-critical scenario is that two *l.sload* (or two *l.sstore*) instructions or *l.sload/l.sstore* and *l.shift* instructions are executed consecutively. This requires the shifting operation to be completed in two processor cycles (or one processor cycle in a 5-stage pipeline) (see Section 7.1.1). A naïve implementation is to shift the 128-bit encryption pad with arbitrary bits in one single cycle. However, such implementation could consume a lot of resources (almost half the size of the OpenRISC core). The challenge of the shift module implementation is therefore to meet the performance constraint with reasonable resource consumption.

Our shift module design is based on the observation that the shifting operation caused by the *l.load/l.store* instruction takes place much more often than the *l.shift* instruction. In fact, our implementation on SQLite shows that a careful design can eliminate the use of the *l.shift* instruction to shift the encryption pad. Thus, our design only supports quick shifting operation for the *l.load/l.store* instruction in one single cycle. The *l.shift* instruction can only shift the encryption 8 bits at a time in multiple cycles, whereas the processor is stalled until the shifting is completed. Our design reduces the resource consumption by 4× as compared to the naïve approach. Figure 7.6 shows the implementation of our shift module which supports the aforementioned operations. Our implementation also provides an encryption pad forwarding path for performance consideration. The detail of this encryption pad forwarding can be seen in Section 7.2.6. The output of the shift module is taken from the most significant 32 bits from the shift registers (see *encPad_load* and *encPad_store* signals in Figure 7.3)

7.2.4 Data Path Switching Module

Because the switch module is located along the memory access critical path, it has to be simple and fast. It consists of only two 2-to-1 multiplexers on both load and store data paths. They are used to switch the incoming and outgoing data with or without decryption and encryption, respectively. Together with the XOR operation, the added latency is only the accumulated combinational logic delay of the multiplexers and XOR gates.

Upon detecting any secure load/store instruction, the processor pipeline sends a signal to the control unit. The control unit schedules these signals with the data arrival signal (*data_ready*) from the load store unit. It then controls the multiplexers in the data path switch module accordingly.

Figure 7.6: Our implementation of the encryption pad shift module. There are four data paths to write the encryption pad value to the shift registers. These data paths are multiplexed and controlled by the control unit.

7.2.5 Control Unit

The purpose of the control unit is to set up the control path for different components in the encryption module upon receiving the custom secure instructions. It is composed of a FSM to generate the control signals and combinational circuit to decode the secure instruction. It receives the custom secure instructions from the instruction decoder and intervenes in the processor pipeline via *datapath_select* and *encryption_stall* control signals. These two signals are used to control the data path switch module and stall the processor pipeline.

7.2.6 Pipeline Stall

Encryption hazard occurs when the processor has to be stalled due to the encryption pad computation. This not only includes the latency of the encryption engine but also the encryption pad shift module, which takes one additional cycle to store the encryption pad in the shift registers. To minimize the encryption stall, the encryption pad buffer (*encPad*) in the encryption controller is forwarded to the switch module (*encPad_load/encPad_store*). Meanwhile, *encPad* is shifted prior to being stored in the shift registers. This further eliminates the additional cycle to shift the encryption pad.

7.2.7 OpenRISC Core Modification

A minor modification of OpenRISC is necessary to support the look-ahead encryption scheme. It includes the instruction decoder to decode the custom secure instructions and the load store unit to buffer the data to be decrypted.

1) *Instruction Decoder*: The instruction decoder is modified to decode the secure instructions. A forwarding path is implemented from the register file to the encryption controller so that the *l.seed* instruction can store the attribute seed value to *Reg_{seed}*. The *l.sload/l.sstore* instructions are handled as normal *l.load/l.store* instruction except that a data path switching signal is passed to the control unit of the encryption module.

2) *Load Store Unit*: Pipeline stalling due to encryption can lead to timing error when the data have to be fetched from the off-chip DRAM via the cache. Since cache line filling accesses DRAM in blocks, any pipeline stall due to seed encryption could lead to wrong data being written to the register.

Figure 7.7 shows the timing diagram of this subtle fault which occurs in the load store unit. If there was no encryption stall, the data value (0x0000FC00) would be written into the register. The subsequent data value would then be used to fill the cache line. However, the encryption stall, if it occurs, delays the register write signal until the encryption finishes. Consequently, a wrong value (0x00004BFF) will be written into the register. To tackle this subtle data fault, we use a data buffer, *DATA_BUF*, to store the right value to be written and set up two data paths for *DATA* and *DATA_BUF*.

Figure 7.7: A diagram showing the timing error without the data buffer in the load store unit. *DATA* is the incoming data from the data cache. *DATA_BUF* is the data buffer implemented to tackle this problem. *ENC_STALL* asserts when the data are ready but the encryption has not yet finished. *REG_WE* is the control signal to write *DATA* into the register.

7.3 SMC Encryption Modules

The goal of our SMC encryption modules design is to encrypt the cache line in the SMC for setting up data path ③ as described in Section 6.5. Since the data cache access heavily relies on the behaviour of the SMC seed buffer access (see Figure 6.7), the challenge of the actual implementation is thus how to schedule and coordinate the off-chip memory accesses for the data cache

and the SMC seed buffer.

Our implementation redesigns the original cache controller so that the scheduling and coordination of the off-chip memory accesses is tightly coupled into one single controller module. This approach is cost-effective because a lot of hardware resources can be reused. In addition to the cache controller, a SMC seed buffer, a cache line encryption engine, and a SeedMem initialization unit are also implemented.

7.3.1 SMC Seed Buffer

The SMC seed buffer has the same structure as the data cache – 1-way direct-mapped cache architecture, each with a 16-byte line size. Both seed buffer and data cache operate in write-back mode. The address to locate a buffer entry can be calculated using Equation 6.2 in Section 6.5.1.

7.3.2 Cache Controller

The implementation of the cache controller can be best illustrated using a state diagram. The overall implementation consists of 12 states, into which four additional states are introduced to perform SMC seed buffer access. For ease of illustration, separated state diagrams are used to describe the executions along different data paths. They are however from the same FSM. The execution flow can be multiplexed so that a single cache controller can serve multiple execution flows.

Figure 7.9 presents the state diagram for the execution along data paths ① and ② where cache line encryption is not involved. It is the original unmodified cache controller in the OpenRISC design. The cache controller consists of eight states, where the function of each of these states is described in Table 7.1.

To implement data path ③ with cache line encryption, the SMC seed buffer has to be accessed for every off-chip memory access in order to get the encryption seed for cache line encryption/decryption. If the seed is stored

inside the seed buffer, it is referred to a buffer hit where the corresponding value can be used to compute the encryption pad. A buffer miss occurs when the encryption seed is not inside the seed buffer, which requires additional off-chip memory access to fetch/write back the seed entry. The detail of the SMC seed buffer execution flow is shown in Figure 6.7 (see Section 6.5.3). Our implementation to realize such execution flow involves four additional states in addition to the original cache controller, which is described in Table 7.2. The modification of the state diagram involves redirecting some executions to access the seed buffer before the actual designated execution to be performed. Figure 7.10 depicts the state diagram of the execution along data path ③ with cache line encryption. Since there is no victim buffer in OpenRISC, the write back operation of data cache has to be stalled until the encryption pad is computed at *STORESTALL* state.

It should be noted that the state diagrams in Figure 7.9 and Figure 7.10 are from the same FSM, where the original eight states are reused in both executions. The execution is multiplexed at *LOADSTORE*, *LOOP3* and *FLUSH* states. The control signal is generated in the circuit as shown in Figure 7.8, which checks whether the secure execution has been set up and whether the cache line is within the SMC. The pulse generator is used to assert an internal signal upon detecting the *start_cypherdb_query* until the *end_cypherdb_query*. It is to indicate the secure query session in hardware.

Figure 7.8: A circuit to generate the control signal for multiplexing the execution flows in cache controller FSM. Cache line encryption is only executed when these two conditions are both satisfied: 1) the secure execution has been set up via the *start_cypherdb_query* instruction and 2) the cache line is within the SMC.

Figure 7.9: A FSM diagram of the original cache controller from the OpenRISC project. This is used for the execution along data paths 1 and 2.

Table 7.1: FSM states involved in the execution along data paths 1 and 2

FSM State	Description
IDLE	Wait for one of the following data cache access operations: load/store/invalidate/flush/write-back.
LOADSTORE	Perform off-chip memory access to fetch or write-back the cache line. It also support cache inhibit memory access operation.
LOOP2	Perform consecutive load/store operations to the off-chip memory for the entire cache line. Each operation read/write 4 bytes data.
LOOP3	Determine the next FSM state for operation. It either ends the data access operation at LOOP4 or WAITSPRCS or continues loading data from off-chip memory at LOOP2.
LOOP4	End the data access operation and return to IDLE state.
FLUSH	Perform flush operation in one of the following ways: 1) invalidate the cache line only at INVALIDATE, 2) write back the dirty cache line at LOOP2, or 3) do nothing at WAITSPRCS.
INVALIDATE	Invalidate the cache line and return to IDLE state.
WAITSPRCS	Wait until the operation has completed and return to IDLE state.

Table 7.2: Additional FSM states involved in the execution along data path 3

FSM State	Description
SEEDACCESS	Perform a seed buffer access and determine one of the following operations: 1) compute the encryption pad and fetch/write-back a data cache line for a seed buffer hit, or 2) perform off-chip memory access to fetch/write back a seed buffer entry for a seed buffer miss.
STORESEED	Write back a seed buffer entry to the off-chip memory.
LOADSEED	Fetch a seed buffer entry from the off-chip memory.
STORESTALL	Stall the data cache access operation and wait for the encryption pad computation. The corresponding seed value is incremented in the seed buffer.

Figure 7.10: A FSM diagram of the cache controller for execution along data path 3. Four additional states (in grey) are added in the FSM and the execution is redirected to these additional states. The red arrow represents the re-directed or additional execution flow.

7.3.3 Cache Line Encryption

Figure 7.11: A diagram showing a cache line encrypted with multiple encryption engines in parallel. The S-bit associated with each cache line is used to determine whether the cache line needs to be encrypted.

Figure 7.11 depicts the cache line encryption mechanism in hardware. Upon the availability of the *count_val* in the SMC seed buffer, the *CLseed* is encrypted to pre-compute the encryption pad which is later used to encrypt the cache blocks in parallel using multiple AES encryption engines. The circuit in Figure 7.8 is used to set the S-bit in each cache line and this bit is later used to multiplex the data path with or without encryption.

As discussed in Section 6.3, the SMC should be allocated to a block of physically contiguous memory block. The tool used for allocating the contiguous memory region (see Section 6.7.1) is however unavailable in our evaluation setup. We take an alternative approach that uses a virtual memory address from the core to set up the SMC. The SMC boundary address is thus the starting address of stack and heap memory. A process identifier (PID) is further used to ensure that the virtual address used is from the same process.

7.3.4 SeedMem Initialization Unit

The FSM of the SeedMem initialization unit is as shown in Figure 7.12. Its purpose is to set the *count_val* in off-chip memory to zero upon the start of the secure query execution (see Algorithm 6.5.2). It is made up of a 2-stage FSM which stalls the processor pipeline and issues a write request to the cache controller over the memory region in the SeedMem.

Figure 7.12: A stage diagram describing the operation of the SeedMem Initialization Unit

7.4 SQLite Modification

SQLite is an embedded SQL database engine which is widely deployed nowadays, including high-profile projects from Apple, Facebook, Dropbox, etc. Although SQLite is not a client/server database application, it is used as our application case study due to three reasons. First, SQLite is light-weight enough to be executed in our resource-constrained FPGA evaluation platform. Its library size can be less than 500KB. Second, it is open-sourced so that modification of the source code is made possible. Last, it consists of a SQL database engine which can parse and execute standard SQL queries over a relational database. The study of SQLite application can thus be extended to other SQL database engines easily.

To apply the look-ahead encryption scheme to a practical database application, the modification of SQLite mainly involves two components: 1) issuing the *l.seed* instruction to store the attribute seed in an on-chip seed register

Regseed, which involves identifying the logical schema identifier used in the application software, and 2) constructing a separate secure data access layer which uses *l.sload/l.sstore* to access the data instead of normal *l.load/l.store*. Our implementation shows that actual modification is minimal which consists of 208 additional LOCs in total.

In this section, we first describe the Virtual Database Engine (VDBE) in SQLite. The VDBE is the heart of SQLite which touches almost all library functions in a virtual machine language. Based on this VDBE, “SELECT” statement is used as a SQL query example to illustrate how the attribute seeds are identified practically. Finally, the modification of the data access layer is discussed.

7.4.1 Background: Virtual Database Engine

Figure 7.13 depicts the architecture of SQLite. The VDBE implements an abstract computing engine to perform SQL specified executions over the underlying database files. It is able to run a program written in its virtual machine language, where the goal of each program is to interrogate or change the database. The program is made up of VDBE-specific instructions. Each instruction contains an opcode and at least three operands labelled as P1, P2 and P3. Operands P1 and P2 are an integer value while P3 is a pointer to a data structure or a string. There are 158 opcodes defined by the VDBE. The reader is referred to [97] for a detailed description of each opcode used in SQLite.

7.4.2 Attribute Seed

As described in Section 5.3.2, the attribute seed of each attribute datum is (*databaseID* || *tableID* || *rowID* || *columnID* || *cntr*). While *cntr* has to be obtained from external storage, other components can be generated from the logical-to-physical schema translation. To illustrate this feature in SQLite, we

Figure 7.13: The architecture of SQLite [2]

use a SQL query example to show how to leverage the VDBE execution to obtain these attribute seed components.

Consider a query statement “SELECT * FROM region;” which is used to extract all records from a database table named “region”. The corresponding VDBE program (sequence of VDBE instructions), which is generated using SQLite built-in query plan analysis via the “EXPLAIN” command, is shown in Table 7.3. The instructions are executed in sequence starting at Addr 0 and the operation of the instruction is also described in the table. The attribute seed components can be obtained from this VDBE program execution in the following procedures:

- Obtain *databaseID* from P1 in the *Transaction* instruction (Addr 11).
- Obtain *tableID* from P2 in the *OpenRead* instruction (Addr 2).
- Obtain *columnID* from P2 in the *Column* instruction (Addr 4-6).

The missing *rowID* is obtained from the the execution of the Column instruction. As shown in Figure 6.1, each record is packaged into a format called payload. In SQLite, this payload header contains the corresponding *RowID*. In the execution of the Column instruction, the payload header is scanned via a function called *getCellInfo()*.

Table 7.3: The VDBE program of the query statement “SELECT * FROM region;”.
The bolded number is used as one of the attribute seed components.

Addr	Opcode	P1	P2	P3	Description
0	Trace	0	0	0	
1	Goto	0	11	0	Jump to address 11
2	OpenRead	0	11	0	Open the table in P2(region)
3	Rewind	0	9	0	The next use of Column instruction will refer to the first record in the database table. Jump to address 9 if the table is empty
4	Column	0	0	1	Extract column 0 and store it in register 1
5	Column	0	1	2	Extract column 1 and store it in register 2
6	Column	0	2	3	Extract column 2 and store it in register 3
7	ResultRow	1	3	0	Construct the query result from register 1 to 3
8	Next	0	4	0	Find the next record in the table and jump to address 4
9	Close	0	0	0	
10	Halt	0	0	0	
11	Transaction	0	0	0	Start a read-transaction on database in P1
12	VerifyCookie	0	22	0	
13	TableLock	0	11	0	Lock the table in P2 (region)
14	Goto	0	2	0	Jump to address 2

7.4.3 Data Access Layer

The purpose of modifying the data access layer is to replace the normal *l.load/l.store* with *l.sload/l.sstore* so that the encrypted attribute can be encrypted/decrypted within the processor chip using our CypherDB secure processor. SQLite provides two functions: *sqlite3VdbeSerialGet()* and *sqlite3VdbeSerialPut()* for reading and writing the attribute data, respectively. These two functions are leveraged to incorporate the look-ahead encryption scheme.

Since the encrypted database may contain unencrypted data (e.g. unencrypted index field), the DBMS must be able to access both types of data. We therefore implemented separate data access functions, *SerialGetScopy()* and *SerialPutScopy()*, for encrypted data whereas the SQLite can selectively call different functions to access encrypted or unencrypted data.

One implementation issue is that the data access function provided by SQLite passes the string type data via a data pointer. This implies that the actual data access process is done at each individual library function within SQLite. To ease the implementation, a buffer is used to store the attribute data where the data pointer is redirected to this buffer. This moves the actual data access process from each individual function to the *SerialGetScopy()* or *SerialPutScopy()* functions. Although this may incur a performance penalty on copying the attribute data into the redirected buffer, it should be noted that this is solely an implementation issue. Better implementation can make use of a compiler to track the flow of the data pointer so that the attribute data access function can be modified in each individual SQLite library function.

Chapter 8

Evaluation

This chapter evaluates various aspects of the CypherDB secure processor design and implementation. Experiments on simulation and the FPGA platform were conducted to provide a comprehensive evaluation of the look-ahead encryption scheme and processor design. Our evaluation employed two approaches. First, simulation allowed us to evaluate the performance under various processor parameters flexibly. We thus could study the performance implication of various architectures. Then we further implemented it on the FPGA to complete the architectural design in hardware. This further enables us to discuss some practical issues of the actual implementation, evaluate the cost of resource consumption (the additional hardware resource usage for the implemented new security modules and the storage overhead of the encrypted database were also studied) and evaluate the execution on a larger dataset which is prohibitively slow when run in simulations. We first present the performance evaluation in simulations before proceeding to the discussion of FPGA evaluation.

8.1 Performance Evaluation: Simulation

This section evaluates the CypherDB secure processor through simulations. We conducted detailed characteristic studies on the additional security com-

ponents or features, including DBPage Buffer, SMC Buffer and the query output protection mechanism. After that, performance studies using various representative workloads were performed.

The performance evaluation was based on comparing the total execution cycles between the execution on an encrypted database using CypherDB techniques and the execution on a normal (non encrypted) database. Here, we refer to the execution on a normal database as the baseline execution in the rest of this thesis. The result of the comparison is reported as slowdown percentage. CypherDB incurs performance overhead in two places: 1) extra instruction executions when copying the on-chip database seeds into Reg_{seed} to perform on-chip data decryption/encryption; 2) extra memory accesses when fetching the counter value of the record counters or SMC seeds from off-chip memory. Our simulation studies thus focus on these two aspects. The performance model is shown in the following equation:

$$\begin{aligned}
 Slowdown = 1/C_b * (C_{enc} + C_{insn} + DBBuf_miss_num * (MemLat + EncLat) \\
 + SMCBuf_miss_num * (MemLat + EncLat))
 \end{aligned}
 \tag{8.1}$$

where:

C_{enc} is the cycle used in the XOR operation in data encryption/decryption

C_{insn} is the cycle caused by executing additional security instruction for data encryption/decryption

$DBBuf_miss_num$ is the total number of buffer misses in the DBPage seed buffer

$SMCBuf_miss_num$ is the total number of buffer misses in the SMC seed buffer

$MemLat$ is the cycle used in off-chip memory access

$EncLat$ is the cycle used in computing the encryption pad

C_b is the cycle used in baseline execution (execution over a normal database)

8.1.1 Simulation Framework

We evaluated CypherDB architecture using the SimpleScalar tool set [87]. We modelled a speculative out-of-order processor with a two-level cache hierarchy with separate data and instruction caches. The parameters used in the simulations are shown in Table 8.1.

Table 8.1: Parameters used in simulations

Parameters	Specifications
Processor Architecture	MIPS ISA
L1 I/D cache	Split, 32KB, 2-way, 32B blocks size, 2 cycles latency
L2 I/D cache	Split, 2MB, 4-way, 64B blocks size, 6 cycles latency
Load/Store queue size	64
Register update unit size	128
TLB	4-way, 128 entries
Main Memory	4GB, 200 cycles latency
AES latency	80 cycles

For each measurement, we executed the query twice. The first query was to fill the pipeline and the data page buffers in main memory. All performance measurements, including the query output protection, were only taken on the second query execution. In the simulations, we ignored the database/query encryption key exchange and random per-query key initialization overhead because it is negligible when compared to the steady-state performance.

8.1.2 Workloads

In our study, we 1) used a data access kernel to evaluate the performance impact of the on-chip decryption/encryption on data path ②; and then 2) ran a

scaled down workload, DBmBench [98], on a real database system, SQLite [82], in order to study the effect of the cache line decryption/encryption engines ③; 3) Finally, a comprehensive performance evaluation of different representative query executions was conducted with the use of the TPC-H [75] benchmark. In all experiments, the data set was generated by the TPC-H data generator and encrypted with our database encryption method described in Section 5. Table 8.2 summarizes the queries used in our three kinds of workload.

Table 8.2: Queries used in the evaluations

Kernel Benchmarking	
Q1	SELECT sum(a3) FROM T1 WHERE Lo < a2 < Hi
Q2	SELECT sum(a3+a4) FROM T1 WHERE Lo < a2 < Hi
Q3	SELECT sum(a3+a4+a5/100+a6/100) FROM T1 WHERE Lo < a2 < Hi
DBmBench [98]	
uSS	SELECT distinct(a3) FROM T1 WHERE Lo < a2 < Hi ORDER BY a3
uIDX	SELECT avg(a3) FROM T1 WHERE Lo < a2 < Hi
uNJ	SELECT avg(T1.a3) FROM T1, T2 WHERE T1.a1 = T2.a1 AND Lo < a2 < Hi
TPC-H	
Q1-Q22	The queries are listed in [75] in detail

SQLite[82] was chosen due to its simplicity and open-source feature. Although it is different from a server-based DBMS, we believe that our implementation can also be made on these server-based DBMSs. Our implementation shows that modification of the DBMS happens in two places: 1) at the beginning and end of the query execution and 2) the data access kernel. These two places can be easily located in the DBMS. The modification of the data access kernel lies in the translation between logical to physical schema which exists in almost all DBMSs but just in different approaches.

Data Access Kernel: Our data access kernel was configured to perform a sequential scan and B⁺-tree non-clustered index scan on a 100MB data file

consisting of 600k records. These two scan operators were chosen because they are the most dominant data access operators in any DBMS. They represent two major data access patterns: predominantly sequential (sequential scan) and predominantly random (non-clustered index scan) record access. We further compared the performance impact of AES-CTR in CypherDB with an alternative approach that uses a conventional AES block cipher to perform the same on-chip decryption/encryption of the attribute data. The AES encryption/decryption was also executed on-chip where the data being loaded were first decrypted before undergoing further computation or immediately encrypted when the data were to be stored under the l.store instruction. The purpose of this experiment was to investigate the performance gain of our look-ahead encryption as compared to conventional data encryption using 128-bit AES encryption. In this approach, we assumed that each attribute is encrypted with 128-bit AES (e.g. encrypting a 32-bit integer to 128-bit ciphertext). This assumption was made due to the fact that some related work like CryptDB [20] or Cipherbase [23] also encrypts the attribute data in such granularity. During the execution, the ciphertext is first loaded. Since it is now in 128 bits, more load instructions and more off-chip memory access (if the ciphertext is stored across two cache lines) are needed. Extra execution cycles are needed for the AES encryption/decryption. The performance model of this simple AES implementation is thus as follows:

$$Slowdown = 1/C_b * (C_{load} + C_{mem} + C_{enc}) \quad (8.2)$$

where:

C_{load} is additional cycles caused by ciphertext loading

C_{mem} is additional cycles caused by memory access

C_{enc} is additional cycles caused by encryption/decryption

C_b is the cycles used in baseline execution (execution over a normal database)

As discussed in Section 5.2.1, the attribute seed is composed of *databaseID*, *tableID*, *ColumnID*, *rowID*, and *cntr*, among which only *cntr* needs to be stored in off-chip memory to maintain temporary uniqueness. Therefore, look-ahead encryption requires an extra off-chip memory access to fetch the *cntr* for the computation of the encryption pad. To remedy this performance overhead, the DBPage seed buffer is used to fetch or pre-fetch a portion of the *cntr* value on-chip to serve fast when they are in need. The record access pattern (i.e. the sequence where records are accessed upon query processing) hence causes a performance impact due to the DBPage seed buffer architecture. Each DBPage buffer entry consists of multiple *cntr* which is the minimal block size to be fetched each time from off-chip memory. Because the record *cntr* is stored in sequential order, sequential record access benefits most from this architecture but not random record access.

DBmBench: DBmBench contains three representative queries that accurately mimic the well-known TPC database workload [98] at the micro-architecture level. These queries include three dominating operators: 1) sequential scan (uSS); 2) index scan (uIDX) and; 3) join query (uNJ).

To compare the performance of CypherDB architecture with the same query execution over an encrypted database using homomorphic encryption, we created a homomorphically encrypted database using the same methodology in [20]: the database is encrypted with OPE for range check, Paillier Encryption for summation, and blowfish for equality check. To construct a similar query execution environment over the homomorphically encrypted database in line with [20], an application program written in C is used to perform the aforementioned three representative queries using the unmodified SQLite libraries. To perform summation operation, gmp library is used to multiply the encrypted data.

TPC-H: We evaluated the performance of CypherDB over the 22 queries in

TPC-H. In our experiments, the evaluation was performed on a 100MB dataset, which is the largest volume of data that can be processed in a reasonable time under our simulation environment.

For all the above workloads, all records were encrypted with our database encryption technique. Indices were encrypted with OPE and appended to the record. These workloads are repeatedly used in our FPGA evaluation.

8.1.3 DBPage Buffer Characteristic Study

We analyzed the DBPage buffer behaviour on running the data access kernel. We first evaluated the DBPage buffer miss rate against different buffer block size which is shown in Figure 8.1(a) and Figure 8.1(b) on sequential scan and index scan, respectively. The off-chip memory access fetches a block of data which contains multiple bytes. The conducted experiments evaluated the optimal block size to be used in the DBPage buffer.

Figure 8.1: The DBPage buffer miss rate on increasing buffer block size in our kernel benchmarking with 20% selectivity on (a) sequential scan and (b) non-clustered index scan

A miss of DBPage buffer refers to a failed attempt to locate the corresponding record counter. Not surprisingly, the buffer miss rate decreases with increased buffer block size on the sequential scan operator but remains constant for the index scan due to the corresponding sequential and random tuple access pattern. In the index scan, the buffer miss rate reduces proportion-

ally with the number of attributes being evaluated within a record due to the shared record counter.

Figure 8.2: The performance evaluation of the sequential scan and non-clustered index scan operators with increasing buffer block size

To investigate its performance impact, Figure 8.2 depicts the percentage slowdown of these two scan operators on various buffer block sizes. The performance slowdown is caused by the DBPage buffer misses and the execution of the additional instructions to copy the seed into *Reg_{seed}*. The result shows that only a small buffer block size with 8 bytes can reduce the slowdown percentage significantly from more than 300% to around 40%. However, a large block size can cause significant performance overhead on the index scan because of its high buffer miss rate and the memory access overhead on loading a large buffer block. As a result, a small DBPage buffer block size of 8 bytes can give fairly good performance on both sequential and index scan operators.

8.1.4 SMC Buffer Characteristic Study

We analyzed the SMC buffer behaviour on running three representative queries in DBmBench. We emulated the SMC buffer with the conventional cache

architecture and investigated its characteristics using three metrics: 1) buffer size; 2) associativity and; 3) buffer block size.

The analysis was based on the buffer miss rate. A buffer miss occurs when there is an attempt to load/store a cache line from/to the off-chip memory, but fails to fetch the corresponding cache line seed in the SMC buffer. Figure 8.3 shows the buffer miss rate with increasing buffer size with fully associative and Least Recently Used (LRU) replacement policies.

Figure 8.3: The SMC buffer miss rate on executing the three queries in DBmBench with increasing buffer size

The result shows that the miss rate starts to drop at 1kB buffer in uIDX while this happens in uSS and uNJ at 64kB buffer. This is because the frequent B⁺tree access reduces the buffer miss rate significantly with increased buffer size in uIDX. In contrast, uSS and uNJ are dominated by sequential operations, resulting in an insignificant impact of the increased buffer size.

Although uSS and uNJ maintain a relatively high buffer miss rate, the seed buffer access is only 16% and 19% of the total off-chip cache line transaction on uSS and uNJ respectively. In other words, most of the cache line transactions are done on the DBpage compartment but not on the secure memory compartment. In contrast, almost all the cache line transactions in uIDX query happen

on the secure memory compartment. This is because B^+ tree reduces the number of accesses to the database but, in return, generates more intermediate data for execution.

We also evaluated the buffer miss rate with different associativity, ranging from 2-way to 64-way. All three queries, however, maintained a steadily high buffer miss rate with increasing buffer size.

Figure 8.4 shows the buffer miss rate on various buffer block sizes. Interestingly, all three queries exhibited the same characteristic that the buffer miss rate dropped significantly at an 8-byte block size. It shows that the working data set of the DBMS is usually in the size of 256 bytes.

Figure 8.4: The SMC buffer miss rate of (a) uSS, (b) uIDX and (c) uNJ with increasing buffer block size

8.1.5 Query Output Encryption Sensitivity Study

In order to study the effect of the enlarged query output data size and entries towards the overall performance penalty, we further conducted two experi-

ments to evaluate it.

The first experiment was performed to evaluate the overall performance slowdown against the query output data size. The query output size was controlled by a function `substr()` to extract a sub-string of some binary data from a database containing very large images. We studied the slowdown percentage caused by the query output protection with the query output data size ranging from 1 byte to 1GB bytes. Figure 8.5(a) shows the corresponding evaluation. It shows that the performance penalty was less than 0.1% when the query output data size was less than 10k. More performance slowdown (up to 2.3%) can be observed with increasing output data size because more output data are being encrypted. When the data size starts to exceed 100MB, it starts to drop off. This alleviation is due to the increased query execution cycles on accessing more data pages, flushing and filling the data page buffers.

The second experiment was performed to evaluate the performance slowdown against the query output entries. In this experiment, we used the same dataset in TPC-H as in our previous experiments. To control the number of query output entries, we used a query - “SELECT attribute FROM table LIMIT n;” to select the number of query output entries from the database by varying the variable `n`. We studied the slowdown percentage caused by the query output protection with the query output data entries ranging from 1 entry to 100M entries. The evaluation is shown in Figure 8.5(b). The performance penalty increased with the number of query output entries (up to 0.9%) but dropped off when it exceeded 10k entries.

8.1.6 Performance on Data Access Kernel

In our experiments, a dedicated function `sum(n)` was used to add *n* attributes within a record together. Such summation operation was performed on 20% of the records. Based on the investigation in Section 8.1.3, DBPage seed buffer of an 8-byte block size, which can accommodate four record counter values, is

(a)

(b)

Figure 8.5: Performance impact of the query output protection with (a) increasing query output data size and (b) increasing query output entries

most efficient in both sequential and random record access.

Table 8.3 compares the performance of CypherDB approach with a direct AES encryption approach. CypherDB executes additional instructions of around 20% on a sequential scan and 1.2% on an index scan. Using direct AES encryption introduces on-chip decryption latency for loading all encrypted data. As each attribute is encrypted in 128-bit cipher blocks, the decrypted data used up more space in cache memory, resulting in extra off-chip memory accesses to fetch the data. It resulted in a 160% slowdown while CypherDB had a steady performance of around 45% for the sequential scan operator. This comparison shows that AES-CTR, in addition to its software-friendly characteristic, performs better than a direct AES approach in terms of performance.

Table 8.3: The Slowdown Percentage of CypherDB and direct AES with the data access kernel on a sequential scan and non-clustered index scan

		CypherDB	Direct AES
Sequential Scan	sum(1)	44%	103%
	sum(2)	44%	116%
	sum(4)	49%	160%
Non-Clustered Index Scan	sum(1)	25%	27%
	sum(2)	25%	28%
	sum(4)	25%	29%

8.1.7 Performance on DBmBench

As the SMC seed buffer stores the counter value for cache-line encryption, its implementation is akin to conventional cache architecture. According to the investigation in Section 8.1.4, a SMC seed buffer with an 8-byte block size, fully associative, and with least-recently-used replacement policy yields the best performance.

Based on this architecture, we evaluated the performance impact of CypherDB with SMC buffer sizes ranging from 32kB to 128kB as shown in Figure 8.6. This range of buffer size was evaluated because our investigation as shown in Figure 8.3 indicated that uSS and uNJ have observable decreases in buffer

miss rate when the buffer size increases from 32KB to 128KB. Without any on-chip DBPage or SMC buffer, the performance overhead on DBmBench is around 17% on average, ranging from 13% to 20%. The performance overhead is not significantly higher even without an on-chip buffer because our data path design introduces no processing penalty on the execution of non-sensitive metadata, which is proven to be the culprit of most data stalls in our benchmark profiling. To achieve better performance, on-chip buffers can significantly reduce the performance slowdown of uIDX query from 20% to 12%. A slight performance improvement can also be observed on uSS and uNJ queries by reducing 2% and 3% slowdown percentage respectively. Less than 0.3% slowdown is attributed to the query output encryption for all queries.

Figure 8.6: Performance evaluation of CypherDB with different SMC buffer size on DBmBench

Table 8.4 shows the executed instruction overhead on the three queries in DBmBench, as introduced by copying the record counter into seed register Reg_{seed} . The instruction overhead of uNJ is relatively high because the execution involves a nested loop join on two tables. Extra instructions are used to copy the corresponding *tableID* of the record counter on this query execution.

In contrast, uSS and uIDX execution involves one table only, and therefore has less instruction overhead than uNJ execution.

Table 8.4: Total executed instruction overhead in DBmBench

	uSS	uIDX	uNJ
Overhead percentage	12%	19%	37%

8.1.8 Performance Comparison with a Homomorphically Encrypted Database

We further compared the performance of CypherDB architecture with the execution on a homomorphically encrypted database. Our evaluation, as shown in Table 8.5, shows that the execution over a homomorphically encrypted database suffers from a significant slowdown ranging from 52% on uSS to 1333% on uIDX. The slowdown on uSS is minimal because only equality and range check are performed in this query, where blowfish and OPE encryption are very efficient in these operations respectively. In spite of their efficiency in these operations, blowfish and OPE encryption transform any data less than 64 bits into a 64-bit cipher. It thus induces extra execution overhead due to handling the enlarged data throughout the DBMS execution, resulting in a moderate performance overhead on uSS execution. On the other hand, uIDX and uNJ both involve summation, which requires the multiplication of two 2048-bit Paillier encrypted data. By transforming a 32-bit integer summation function into a multiplication operation of two 2048-bit data, the execution cycles of uIDX and uNJ increases by $14\times$ and $8\times$ respectively. CypherDB, on the contrary, maintains a steadily reasonable performance overhead of 11% to 17%, regardless of the operations involved.

Table 8.5: Slowdown Percentage of CypherDB with a 128kB SMC Buffer and execution over a homomorphically encrypted database on DBmBench

	CypherDB	Homomorphic Encryption
uSS	11%	52%
uIDX	12%	1333%
uNJ	17%	705%

8.1.9 Performance on TPC-H

Figure 8.7 illustrates the performance of CypherDB on TPC-H queries. The average performance overhead is only 10% ranging from 3% for query 11 to 23% for query 14. The query output encryption results in 0.35% slowdown on average. We also evaluated the instruction count overhead of the 22 queries in TPC-H as summarized in Table 8.6. On average, there are around 14% more instructions being executed in our proposed solution, ranging from 8% in query 12 to 28% in query 14. This instruction overhead is the main contributor to the performance degradation.

Figure 8.7: Performance evaluation of CypherDB with different SMC buffer sizes on the 22 queries in TPC-H

Table 8.6: Total executed instruction overhead in TPC-H

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Overhead percentage	10%	17%	12%	19%	14%	14%	11%	17%	11%	15%	11%
	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Overhead percentage	8%	14%	28%	13%	13%	13%	19%	9%	21%	10%	16%

8.2 Performance Evaluation: FPGA Platform

This section evaluates the performance of our look-ahead encryption scheme and the secure processor implementation on a FPGA. We first evaluated the proposed encryption scheme and the SMC encryption mechanism separately. After that, an overall evaluation of running representative database queries on SQLite is discussed. All experiments in this section were based on the FPGA implementation and executions on the FPGA board.

The performance evaluation on the FPGA was based on comparing the total CPU time (during query processing) between the execution on an encrypted database and the baseline execution. Similarly, the result of the comparison is reported as slowdown percentage. The same set of workloads used in simulation was also employed in FPGA evaluation.

8.2.1 Experimental Setup

Figure 8.8 shows the experimental setup on a FPGA board. The databases to be queried (either encrypted or unencrypted) and the record counters were stored in a SD card while the SQLite executable and Linux were loaded into the SDRAM during system boot-up. A terminal was connected to the serial port to send out queries to and receive query results from the FPGA. The Linux UART and SD/MMC device driver were used to provide an abstraction of file system and I/O to the SQLite. A host computer computer was connected to the FPGA via Joint Test Action Group (JTAG) for debugging. The hardware performance counter was observed with the use of the SignalTap Logic Analyzer [99]. The default parameters in Table 8.7 were used to implement the soft core in the FPGA.

Four evaluation metrics were used in all the experiments on the FPGA: total execution cycles, encryption stall cycles, SMC seed buffer miss rate and total Central Processing Unit (CPU) time spent on execution. Total execution cycles, encryption stall cycles and SMC seed buffer miss rate were measured

Figure 8.8: Experimental setup on the FPGA

Table 8.7: The default OpenRISC processor parameters

Parameters	Specifications
Processor	OR1200 core, 50MHz
SDRAM	64MB, 100MHz
I/D cache	64KB, direct-mapped, 32B line
SeedBuf	32kB, direct-mapped, 16B line

by the self-developed hardware performance counter. The CPU time spent on execution was obtained by the timing function in Linux. The accurate timing information was provided by a hardware tick timer implemented in the FPGA. The correctness of the implementation was verified by checking the correctness of the query output in response to the corresponding input query.

8.2.2 Look-ahead Encryption Scheme Study

The purpose of this study was to evaluate the encryption stalls caused by the look-ahead encryption module described in Section 7.2. The encryption stall is caused by two factors: 1) the time that the *l.seed* instruction is issued to compute the encryption pad and 2) the encryption latency of the AES encryption. In reality, the encryption engine can be operating at different frequencies which could result in various encryption latencies. To investigate this impact, the data access kernel described in Section 8.1.2 was executed on the FPGA whereas the data file was stored in the SD card.

Figure 8.9: Total encryption stall cycles with increasing operating frequency of the AES encryption engine

Figure 8.9 shows the total number of encryption stall cycles with varying operating frequency of the AES engine. An exponential relationship between

the encryption latency and the encryption stalls is observed because there are multiple encrypted data being accessed in the kernel which are all affected by the lengthened encryption latency. The latency hiding of the look-ahead encryption scheme relies on the clock ratio between the processor and the AES engine. Our study shows that the encryption engine should not be $4\times$ slower than the processor in order to hide most of the encryption latencies from the program execution.

8.2.3 SMC Encryption Study

The purpose of this study was to evaluate the efficiency of the modified cache controller and the cache line encryption described in Section 7.3. Since the tool for allocating contiguous physical memory blocks for the SMC in Linux was unavailable, we used four bare-metal applications instead in this study. This is because these bare-metal applications can run directly on hardware so that a custom page table can be created for SMC allocation, which does not rely on Linux for the virtual-to-physical memory mapping. These four bare-metal applications (QUICKSORT, MEDIAN, MAX, KMEANS) were chosen because they represent various commonly used data processing functions and are simple for development in a bare-metal environment.

The key modification of the cache controller is to manage the *count_val* in the SMC seed buffer and compute the encryption pad if the corresponding *count_val* is in the buffer. The major performance penalty is caused by a SMC buffer miss and dirty seed write-back. We therefore evaluated the SMC buffer miss rate and write-back rate for these four bare-metal applications as shown in Figure 8.10. It can be seen that the SMC characteristic varied with different applications. In some cases like MAX function which demonstrates a highly sequential memory access pattern, the buffer miss rate and write-back rate are constantly low. Other applications however show significant improvement with a larger SMC seed buffer size.

(a)

(b)

Figure 8.10: The (a) SMC buffer mis rate and (b) SMC buffer write back rate on executing four bare-metal applications with SMC encryption on the FPGA

Its performance impact is two-fold. First, a buffer miss or write-back causes additional off-chip memory access to fetch or store the *count_val*. Second, if the data cache line has to be written back, the cache line encryption has to stall the processor (see Section 7.3.2). However, the current implementation poses one limitation that the processor clock (50MHz) runs slower than the SDRAM (100MHz). This diminishes the performance penalty caused by the SMC seed buffer miss or write-back. To better reflect its performance in a realistic setup, a delay circuit was implemented on the off-chip memory access path to inject additional latencies in order to emulate a setup with a higher processor clock (up to 1600MHz). The evaluation of this emulated processor clock frequency can help us to study the characteristic of the SMC encryption mechanism.

Figure 8.11 to Figure 8.14 show the performance of the four bare-metal applications with various SMC seed buffer sizes at different (emulated) clock frequencies. These show that the performance penalty is high (up to 80% in QUICKSORT and MEDIAN) with a large processor's clock to SDRAM's clock ratio but the performance is significantly improved with a large SMC seed buffer size. Applications like MAX with a low SMC seed miss/write-back rate show reasonable performance penalty. On the other hand, applications like KMEANS which take few data inputs for intensive computation have very little negative impact on the overall performance.

This study demonstrates that the SMC seed buffer behaves differently toward different applications. However, a larger SMC seed buffer is beneficial to most applications in general.

8.2.4 Performance Study on SQLite

SQLite was used in our study of a practical database application. We first evaluated the impact of our look-ahead encryption scheme on this application before an overall evaluation with both the encryption scheme and SMC

Figure 8.11: Performance evaluation of QUICKSORT on the FPGA

Figure 8.12: Performance evaluation of MEDIAN on the FPGA

Figure 8.13: Performance evaluation of KMEANS on the FPGA

Figure 8.14: Performance evaluation of MAX on the FPGA

encryption mechanism.

In all experiments, CPU time was taken as the metric of performance evaluation. CPU time only measures the amount of time spent on executing instructions (either from user or kernel space) in the processor, which can neglect the time waiting for I/O operations. Since the performance impact is mainly on the processor core and SDRAM, CPU time can provide a more meaningful evaluation of the performance.

Besides, due to the fact that our study on the query output protection mechanism in simulations demonstrates a negligible performance impact, this was not evaluated in the following experiments so as to ease the correctness verification process.

Look-ahead Encryption Only

DBmBench was used to evaluate the performance impact of the look-ahead encryption scheme. In all experiments, each query was run five times and the average CPU runtime was recorded. We ran the queries on a 100MB, 500MB and 1GB database comprising around 0.6, 4 and 8 million records, respectively.

Figure 8.15 shows the total CPU time spent on processing the queries on the 100MB, 500MB and 1GB databases. It was found that the performance overhead caused by the look-ahead encryption scheme is negligible in uSS and uNJ due to our careful design of hardware and modification over SQLite. uIDX shows a modest performance penalty ranging from 14% to 26%. This performance impact reduces with increasing database size.

To investigate the reason for these performance overheads, we analyzed the CPU time spent on user codes and kernel codes separately. Our investigation found that the CPU time spent on user code ranges from 1% to 11%. These overheads are caused by the extra instructions since they were inserted in user code space. On the other hand, the CPU time spent on kernel code ranged from 20% to 43%. This is because the record counters occupied a small

(a) 100MB database

(b) 500MB database

(c) 1GB database

Figure 8.15: Total CPU time spent on processing the DBmBench query benchmark on (a) 100MB, (b) 500MB and (c) 1GB databases

portion of the physical memory. With the reduction of the available memory, more time is spent on the kernel to allocate and free memory for the query execution. This overhead is observable due to an experimental limitation that only a small memory (64MB) is available in our FPGA evaluation platform. We believe that it would be alleviated with the use of a larger memory.

Look-ahead Encryption + SMC Encryption

The challenge of evaluating SMC encryption performance impact on SQLite is that it requires contiguous physical memory space allocation which Linux does not support. SQLite however relies on Linux libraries for execution. Our solution to this challenge was to remove the cache line encryption data path and change it to a normal (unencrypted) data path, but still keep the encryption control path active. Since our implementation constructed a control path and data path separately and the actual data encryption was reduced to XOR operations (with negligible performance penalty), disabling the encryption data path caused no influence on the performance evaluation.

The evaluation was performed on 100MB and 1GB databases. We first evaluated the SMC buffer miss rate on DBmBench query execution as shown in Figure 8.16. The execution on both databases demonstrated a reduction in miss rate with increasing buffer size. It can also be seen that the reduction of miss rate was especially more significant when the buffer size increased from 8k to 32k but tended to flatten out after 32k. This matches the similar evaluation in simulations (see Figure 8.3).

The performance evaluation is depicted in Figure 8.17. The performance slowdown was measured by a comparison with CPU time in the baseline execution. It can be seen that an increased SMC buffer size had no effect on the query execution over a 100MB database but improved the performance on a 1GB database. The reason is that the improvement accumulated sufficiently to be observable in the evaluation using the 1GB database. As compared to the

(a) 100MB database

(b) 1GB database

Figure 8.16: The SMC buffer miss rate of executing the queries in DBmBench over a (a) 100MB and (b) 1GB database on the FPGA

similar evaluation in simulations (see Figure 8.6), both evaluations show that index scan can benefit from an increased SMC seed buffer size than sequential scan and nested loop join.

(a) 100MB database

(b) 1GB database

Figure 8.17: Performance evaluation of DBmBench on the FPGA

We further evaluated nine queries in TPC-H with a 32kB SMC seed buffer as shown in Figure 8.18. The reason only nine out of 22 queries were evaluated

is mainly due to the limited resources of the FPGA evaluation board used. The 64MB SDRAM used was too small to accommodate the memory usage (stack and heap memory) during SQLite execution. Meanwhile, for several queries which needed to scan through the records one by one, it took a sufficiently long processing time on the FPGA. In addition, each query was run three times to take an average result. This made the processing time even longer. Therefore, these nine queries are the queries that could be executed on our evaluation platform in a reasonable time.

The corresponding SMC seed buffer miss rate is depicted in Table 8.8. It can be seen that the performance over an encrypted database outperformed the unencrypted one in Q6 and Q22. We note that the database layout of an encrypted database is different from the unencrypted one. This may cause certain variation in the physical data access pattern, although the query execution procedure is the same. In Q3 and Q20, the performance significantly improved because the index scan is extensively executed in these queries. The time spent on allocating memory space is alleviated with a larger database size. The performance on Q13 was constantly high over 100MB and 1GB databases. Our investigation shows that it is due to the implementation issue described in Section 7.4.3 where the string-type data are additionally buffered for ease of our implementation. In Q13, the query involves an operation to scan through a large amount of string-type data which results in high performance penalty. This can be alleviated with a more careful modification of SQLite. The SMC seed buffer miss rate is kept constantly ranging from 11% to 30%. This shows that the 32kB seed buffer is applicable for a reasonable performance.

Table 8.8: SMC seed buffer miss rate of the nine queries evaluated in TPC-H

	Q2	Q3	Q4	Q6	Q8	Q13	Q16	Q20	Q22
100MB database	15%	24%	20%	21%	23%	28%	11%	21%	15%
1GB database	21%	27%	19%	20%	24%	20%	30%	25%	19%

(a) 100MB database

(b) 1GB database

Figure 8.18: Performance evaluation of the nine queries in TPC-H on the FPGA

8.2.5 Resource Utilization

We evaluated the resource usage of our implemented system. Table 8.9 compares the resource utilization of our implemented secure processor with the original OpenRISC. The resource usage of the modified components in OpenRISC and the additional modules are listed. The encryption engine, encryption controller and shift module were duplicated for the separate secure data load and store paths. The data cache controller was modified and cache line encryption engines were used. It can be seen that our implementation required $3\times$ more resources than the original OpenRISC implementation. However, 87% of these extra resources was incurred by the encryption engines. This overhead is mainly due to the extreme simplicity of OpenRISC where the AES encryption engine is itself larger than the processor core. We believe that further optimization (e.g. resource sharing) can reduce the resource consumption caused by the AES encryption engines. Overall, the entire system consumes less than 24% of the available FPGA resources on the DE2i-150 board.

8.3 Storage Overhead

Our database encryption technique does not consume much extra storage because the attribute encryption often preserves the size of the attribute data. The only exception is due to the compression technique in SQLite. In SQLite, integer values are not stored in a fixed-size data container but varies from 1 to 4 bytes to save space. For example, an integer value of “229” is stored in two bytes while the value of “2298336” is stored in three bytes. However, our observation is that this compression discloses the range of the integer value from its storage size. We therefore disabled this feature and stored every integer value in four bytes. Disabling this compression feature introduced 1% storage overhead.

Most of the storage overhead comes from the index protection because the

Table 8.9: Total resource utilization of the overall system. The displayed number is in the format of (CypherDB/Baseline). # denotes that the value is the same in CypherDB and baseline implementation. * denotes that the value is the total resource consumption on the load and store execution path.

	Logic Cells	Dedicated Registers	BRAM
Encryption Module*	11558 / 0	469 / 0	0 / 0
- AES Engine*	9912 / 0	261 / 0	0 / 0
- Encryption Controller*	546 / 0	60 / 0	0 / 0
- Shift Module*	1008 / 0	0 / 0	0 / 0
LSU	388 / 162	0 / 0	0 / 0
Instruction Decoder	326 / 243	0 / 0	0 / 0
CPU core (others)	3595 / 3477	103 / 155	2 / 2
D-Cache	10026 / 598	356 / 113	111 / 69
- AES Engine $\times$ 2	8991 / 0	160 / 0	0 / 0
- Cache Controller	380 / 219	15 / 12	0 / 0
- Cache RAM	240 / 219	76 / 84	64 / 64
- Cache TAG	60 / 53	13 / 17	5 / 5
- Seed Buffer RAM	150 / 0	77 / 0	32 / 0
- Seed Buffer TAG	47 / 0	14 / 0	10 / 0
SeedInitMem Core	77 / 0	0 / 0	0 / 0
I-Cache#	323	39	69
SD Card Controller#	3079	685	2
SDRAM Controller#	3244	706	6
UART#	676	74	2
JTAG#	780	145	0
Others	1369 / 1216	123 / 167	4 / 4
Total	35441 / 13798	2700 / 2084	196 / 154

indices are encrypted in OPE and appended to each record. A lengthened record also consumes extra database pages to store, which in turn introduces more database page headers which consumes additional storage. It consumes 28% storage overhead in total.

The remaining storage overhead comes from the record counter. The size of this record counter depends on the total number of records (not the size) in the database and the size of the record counter. Assume the record counter is of the size of 4 bytes, which is large enough to avoid wrapping around. The extra storage consumed is only 3.3% in a TPC-H dataset.

Therefore, the storage overhead caused by our database encryption technique can be at best 4% (indices are not encrypted) or at worst introduce 31% additional storage overhead (indices are encrypted with OPE).

Chapter 9

Conclusions

9.1 Securing DBaaS

Security remains one of the biggest obstacles toward the adoption of cloud database services. Data confidentiality can be protected by encryption. However, the recent security solutions store the decryption key in a breakable cloud server and disclose the content of the data after decryption. Recent security breaches have demonstrated that a strong security model is needed in Cloud Computing. Particularly, the decryption key and the decrypted data have to be properly protected against any honest-but-curious administrator.

To be applicable in a practical cloud environment, the protection mechanism should address two important and challenging aspects: security and performance. In terms of security, the protection of the decryption key and the decrypted data has to be strictly enforced. In other words, the decryption key should never be disclosed and the data should always be protected by encryption, even during query processing. In terms of performance, the protection mechanism should introduce minimal overhead in relation to the overall query execution time. Enforcing a high security protection mechanism with high performance is a challenging design problem.

We have developed a system architecture CypherDB to secure database processing in the cloud with high performance. The key idea is to keep the

decryption key stored inside the processor chip without disclosing it in any circumstance and develop a protection mechanism with the use of this undisclosed key. Although keeping the decryption key inside the processor chip is in line with other proposals, a distinctive feature of our research is that our protection mechanism can work tightly with the database application to reduce the encryption overhead. Besides, our protection mechanism is designed to be applicable in a cloud computing environment where the encrypted data can move dynamically among servers for parallel processing. In summary, CypherDB architecture addresses four essential issues in securing a cloud database service: security, performance, cost and deployment in the cloud.

The security of CypherDB is strictly enforced by not disclosing the decryption key and encrypting any sensitive data in off-chip memory. Based on the assumption that micro-probing on the processor chip is infeasible in a cloud computing environment, the decryption key of the database owner can be securely delivered to the secure processor chip in the cloud through a key encapsulation mechanism and processor attestation. The database data, query result and any intermediate data are always encrypted in off-chip memory through our novel look-ahead encryption scheme and cache line encryption. CypherDB also employs different encryption keys for different data and various protection purposes. This further enhances the security by not overusing the encryption key.

The design of the look-ahead encryption scheme and cache line encryption are also motivated by their performance impact. Both designs aim to hide the encryption latency from program execution or off-chip memory access as much as possible. In the implementation, these two protection mechanisms are also highly optimized by a careful design of an automatic shift module and a buffer to store the encryption seed. Their performance impact is extensively studied and verified through simulation and execution on hardware. The experiment of running TPC-H query workloads on SQLite demonstrates 10 percent and

14 percent performance overhead on average, in simulation and execution on a FPGA, respectively. This result proves that our CypherDB architecture can secure the database processing with high performance.

Our design is cost-effective. Our implementation shows that the resource consumption (excluding the AES encryption engines) is only 18 percent more than the baseline architecture.¹ Although four AES encryption engines are currently used in our implementation which consumes 80 percent extra resources, this can be optimized in our future work which is discussed in Section 9.3. The modification of SQLite to support our look-ahead encryption scheme is also minimal which only incurs 208 LOCs while the attribute seed management software consists of 234 LOCs. This shows that our design uses up minimal resources and is able to achieve high performance.

CypherDB is highly applicable in Cloud Computing. In the first place, our look-ahead encryption scheme encrypts the database at a logical level. The encrypted database can thus be partitioned into records and dynamically allocated in the cloud servers for parallel processing. Secondly, the key encapsulation mechanism allows the database encryption key to be stored in multiple secure processors, which enables resource pooling in a cluster of processing nodes with CypherDB secure processors. Rapid elasticity and multi-tenancy are also possible because the database encryption key is not bound to a particular secure processor. The number of secure processors allocated for secure query processing is also on-demand.

9.2 Limitations

CypherDB architecture is not without its limitations. Perhaps one major limitation of CypherDB is that it is not entirely self-contained. It relies on other security systems to enforce the database protection. For example, it relies on some secure boot technology and run-time execution validator to protect the

¹The baseline architecture here means the unmodified OpenRISC project.

integrity of the DBMS. It also takes advantage of some hardware-assisted isolated execution solutions for memory allocation. The security of the CypherDB system is therefore dependent on the security of other employed systems. This complication is due to the fact that the decryption key, though never undisclosed, is still used to decrypt the data at the server side. Therefore, multiple protection mechanisms are needed to protect the use of this encryption key and the decrypted data in the system. Though complicated, our study shows that it has a tremendous performance advantage over homomorphic encryption. CypherDB is thus a practical solution, yet it requires a collaboration of other techniques for security.

Another limitation is that it requires modification of the application code. Our experience shows that the actual modification can be minimal but has to be careful. The modification is just to issue the *l.seed* instruction at the right place with the correct seed value and replace normal load/store instructions with secure load/store instructions. Identifying an ideal place to issue or replace these instructions requires careful observation and investigation. Issuing the *l.seed* instruction too close to the actual data access instruction cannot hide much latencies using the look-ahead encryption scheme. Fortunately, the application program is not bound to a particular database owner. The database application program can serve multiple databases from different users so that the modification only needs to be performed once by the database software vendor.

The data in the allocated SMC cannot be swapped to/from the hard disk in our current prototype poses another limitation. The reason is that the cache line in the SMC is encrypted with its physical address. Swapping in and out another cache line reuses the same physical address, which may potentially cause “two-time-pad” attack. Fortunately, recent computer systems have a large memory for SMC allocation and the volatile memory is cheap. Allocating a large SMC in main memory without swapping in/from the hard disk can

be made possible. If swapping is unavoidable, a potential solution to this is to re-encrypt the cache line (or a memory page) with AES in normal mode (without the encryption seed). However, this requires re-encryption which may potentially cause certain performance overhead.

9.3 Future Research

9.3.1 Multi-core Architecture

The research described in this thesis can be extended in several directions. One of the most challenging directions is to extend the current work to multi-core systems. A simple approach is to replace the core in a multi-core system with our proposed secure processor core so that each core can support encrypted data processing. This however leads to inefficient use of the encryption engine and large resource consumption.

There are better strategies to implement CypherDB architecture on multi-core systems. For example, every core can have its own encryption module for look-ahead encryption. However, this requires synchronization among cores because the encrypted data may be loaded into a core without a proper encryption pad. It is also expensive to implement. On the other hand, a single encryption module can be shared by all the cores so that synchronization is easier to perform. However, resource contention may arise in this approach. To process the query, one can assign a single core to a single query execution so that each core serves one tenant at a time. This is simpler to implement because no synchronization of the encryption pad is needed across multiple cores. Nevertheless, this does not achieve high performance. On the contrary, multiple cores can be used to serve the secure query execution for a single tenant. To achieve this, future work on exploring methodology of encryption pad synchronization is needed. Extending the CypherDB architecture to a multi-core system is a challenging and interesting research area.

9.3.2 Target Application

Another interesting direction is to extend the CypherDB architecture to a column-oriented database. A column-oriented database serializes the column value together where the column can match multiple rows. This poses great demand on encryption pad generation. On the positive side, AES encryption is highly pipelined so that the encryption pad can be generated with high throughput. However, this may require sophisticated mapping between the encryption pad and the encrypted data being loaded. There is a great potential to extend the CypherDB architecture to a column-oriented database but still requires careful design and investigation.

We envision that the CypherDB architecture may not be limited to database applications. In fact, other data management applications also require a certain form of data model and data access function where our look-ahead encryption scheme can be applied. The potential challenge is the choice of attribute seed. SQL databases have a well-defined data model that can easily make use of the logical schema to formulate the attribute seed. This also potentially can be used in documents like XML to support NoSQL cloud applications. The trick is how to encrypt the data in different granularity using the AES-CTR and AES-OFB. Heavily using AES-CTR will introduce too many unique encryption seeds to be managed while using AES-OFB can potentially degrade the performance because shifting the encryption for AES-OFB is costly. This entails interesting future work of extending the current hardware architecture to other applications and there is the possibility of modification and optimization of the current architecture. Applying our proposed encryption scheme to other data processing application is thus highly possible in our future research.

9.3.3 Compiler Support

With the insight from the limitation of CypherDB, one future direction is to provide the CypherDB architecture with intelligent compiler support. This

is an interesting topic to investigate how an intelligent compiler can support the proposed look-ahead encryption. In our experience, the insertion of the *l.seed/l.shift/l.sload/l.sstore* instructions are closely related to the data model and how the software accesses the data in accordance with the data model. It is hard to rely on the compiler completely but the compiler may be able to assist the programmers to locate the right positions.

For example, the relational data model in our work, according to our investigation, can have a unique seed in the format of (*databaseID*, *tableID*, *ColumnID*, *rowID*, *cntr*) for each attribute datum. First, a formal language can be invented to describe the relation between the encryption seed and the data. This formal language is to be parsed and interpreted by the compiler. After that, the programmers are required to annotate the variables in software which are to be used as the encryption seed. The compiler can then interactively analyze the performance implication of the look-ahead encryption and suggest the best positions to insert those instructions in order to yield better overall performance. With proper annotation in the database application program, the compiler may be able to automatically provide an optimal solution to implement the look-ahead encryption.

9.3.4 Context Switching

Context switching happens frequently in modern computers which support multi-tasking and interrupts. It is however expensive because it requires saving the state of the processor (register values) into the stack so that the current state can be recovered later.

This is especially expensive in our proposed architecture because all the internal states of the processor have to be stored in the stack and stored in the SMC, including the states of the encryption module like encryption pad, shift register and the control states. In normal processors, only register values have to be stored. Since the process being switched in may consist of malicious code

that intends to read the data in the cache, the sensitive data residing in on-chip cache is also subject to attack upon context switching. Thus, the cache line with *S_bit* asserted needs to be evicted under our architecture. Writing back these cache lines requires further encryptions which causes large performance overhead upon every context switch.

There are several potential solutions to reduce the cost of context switching. A simple approach is to lock the cache lines with *S_bit* asserted during a context switch. By locking the cache lines containing sensitive data, a hardware circuit can forbid any data access (including load and store operation) to those locked cache lines. It can also prevent these cache lines from being evicted from the processor. By abandoning the cache lines with *S_bit* asserted, the system can ensure the sensitive data cannot be leaked but it virtually decreases the cache size to be used by the process being switched in. A better approach is to have dynamic cache line locking, which allows the cache lines with *S_bit* asserted to be evicted from the processor with encryption on-demand (i.e. the cache lines do not need to be evicted immediately upon a context switch). Upon a context switch, all sensitive cache lines are locked. If the incoming process consumes very little data cache, most of the sensitive cache lines do not need to be evicted, thus reducing the penalty. However, if the incoming process unfortunately requires a large data cache, the sensitive cache lines have to be evicted with encryption on demand. It makes the performance penalty dependence of the data cache usage of the process being switched in. To reduce the penalty, another interesting approach is to maximize the opportunity of encryption pad pre-computation so as to hide the cache line encryption latency as much as possible. This can potentially be achieved by bookkeeping the *TAG* and *count_val* of the sensitive cache line on-chip so that the encryption pad can be pre-computed once the effective address of the next load/store operation is calculated. More aggressively, all encryption pads of the sensitive cache lines can be pre-computed and stored on-chip but it consumes a lot of

on-chip storage. Alternatively, predictively pre-computing the encryption pad can serve as a less aggressive approach.

Context switching has an impact on encryption key management as well. There are mainly two scenarios: 1) The process/interrupt handler being switched in is not a secure execution. 2) the process/interrupt handler being switched in is a secure execution. If the incoming process does not involve an encryption key, the key can be stored in the on-chip register without any changes. Since the encryption key cannot be read, the incoming process cannot read or know the encryption key. However, if the incoming process involves another encryption key, the original encryption key has to be stored somewhere so that the on-chip register can be used to store the new encryption key. This can be solved in two approaches. First, a larger on-chip memory can be used to store a multiple but fixed number of encryption keys inside the processor. Multiple encryption keys can thus be multiplexed to be used while all of them are forbidden to be read or stored. This solution however is not scalable for multi-tenancy. Another approach is to allow the encryption key to be encrypted with key encapsulation (i.e. $Enc(K_{db})$) before storing it back to the main memory. Since the encryption key is protected by key encapsulation (i.e. encrypting the key with asymmetric key encryption using the processor public key), the encryption key is still protected even if it is now stored in main memory. But the drawback of this approach is the excessive overhead on encrypting the encryption key with asymmetric key encryption. Encryption key management and performance overhead caused by context switching is important to make our proposed architecture compatible with modern processor's standards.

9.3.5 Code Injection Attack Protection

The idea of code injection attack is to redirect the original program execution to run a piece of malicious code so as to perform various active attacks. Common code injection attacks such as buffer overflow attack and SQL injection attack

has been the most prevalent attacks in the past few decades. Although we do not consider code injection attacks in this thesis (only passive attack on the confidentiality of data is considered in our work), we discuss a few techniques that can be used in conjunction with our CypherDB architecture.

In our proposed architecture, we can prevent an adversary from getting the encryption keys (the critical secret) because these keys never leave the processor. This allows a strong security protection of the confidentiality of data. It however cannot forbid an adversary to manipulate the secure processor to indirectly learn the original value of the encrypted data without getting the encryption key. For example, an adversary can perform a code injection attack so that the malicious program can issue the *l.sload* instruction to load and decrypt the data and then issue the *l.store* instruction to store the data back to the memory without further encryption. This will then violate the protection mechanism of the proposed solution. By having such run-time attack, an adversary can decrypt the data one by one.

To tackle this problem, we believe that there are several proposed solutions that can be leveraged to our proposed architecture to thwart code injection attacks. First, a run-time validator [34] can be used to ensure that the original trusted program is running but not any untrusted program (i.e. code-injection attack). The idea of this protection mechanism is to associate each instruction with a signed hash value so that each instruction (or a batch of instructions) being executed has a signature identifying that it is an authorized instruction. Because malicious code being injected does not contain a valid signature, it provides a tamper-resistant mechanism to prevent code injection attack. Another tamper-resistant approach is to restrict the memory page access with the techniques proposed in Iso-X [27] or Bastion [33]. These approaches share the same technique that makes use of TLB to restrict the access to a particular running application process. Bastion [33] uses a trusted software module to handle the virtual-to-physical address page table. Iso-X [27] uses a hardware-

protected memory compartment table and a bit vector to store the hardware usage profile of the secure process. Interestingly, these techniques can potentially be leveraged into our proposed architecture. For example, the encryption key can be hashed to create a signature. This signature acts as an index to the corresponding encryption key and is stored in the page table (in Bastion architecture) or memory compartment table (in Iso-X architecture). Having the encryption keys bounded to some dedicated memory pages, code injection attack is thus made difficult since the malicious code has to be injected/stored in the right memory pages. It is interesting to incorporate these techniques into our proposed architecture for our future research.

Appendix A

CypherDB ISA Extension on OpenRISC Architecture

start_cypherdb_query

31	26	25	21	20	16	15	11	10	0
opcode		reserved		A		B		reserved	
6 bits		5 bits		5 bits		5 bits		11 bits	

Format:

start_cypherdb_query rA, rB

Description: Start the secure query execution by setting up the SMC, generating the SMC encryption key and initialize the encryption seeds in SeedMem. The content of general-purpose rA and rB are moved to security registers of SMC starting address and ending address respectively. A random number generated from on-chip random number generator is moved to the key register of SMC encryption key.

Implementation:

SR[SMC_start] $\leftarrow$ rA

SR[SMC_end] $\leftarrow$ rB

KeyReg[SMC] $\leftarrow$ random number from on-chip random number generator

Initialize the SeedMem

end_cypherdb_query

31	26	25		0
opcode		N		
6 bits		26 bits		

Format:

end_cypherdb_query

Description: End the secure query execution by zeroizing the contents of start and end address of SMC in security registers, and the SMC encryption key stored in key register.

Implementation:

SR[SMC_start] $\leftarrow$ Zero

SR[SMC_end] $\leftarrow$ Zero

KeyReg[SMC] $\leftarrow$ Zero

l.seed

31	26	25	21	20	16	15	11	10	0
opcode		K		A		B		K	
6 bits		5 bits		5 bits		5 bits		11 bits	

Format:

l.seed rA, rB, K

Description: Sign-extending eight most significant bits of the immediate value and then logically ORed with the contents of general-purpose register rA. The result defined the seed register into where the contents of general-purpose rB are moved.

Implementation:

$$\text{SeedReg}[\text{exts}(\text{K}[10:2]) \text{ OR } \text{rA}] \leftarrow \text{rB}$$
l.shift

31	26	25	21	20	16	15	11	10	0
opcode		K		A		B		K	
6 bits		5 bits		5 bits		5 bits		11 bits	

Format:

$$\text{l.shift} \quad \text{rA}, \text{rB}, \text{K}$$

Description: The eight most significant bits of the immediate value is sign-extended before it is logically ORed with the contents of general-purpose rA and rB. The result is moved to the shift register.

Implementation:

$$\text{ShiftReg} \leftarrow \text{rA OR rB OR exts}(\text{K}[10:2])$$
l.sload

31	26	25	21	20	16	15	0
opcode		D		A		K	
6 bits		5 bits		5 bits		16 bits	

Format:

$$\text{Load Byte:} \quad \text{l.slbz} \quad \text{rD}, \text{K(rA)} \quad \text{l.slbs} \quad \text{rD}, \text{K(rA)}$$

$$\text{Load Half Word:} \quad \text{l.slhz} \quad \text{rD}, \text{K(rA)} \quad \text{l.slhs} \quad \text{rD}, \text{K(rA)}$$

$$\text{Load Word:} \quad \text{l.slwz} \quad \text{rD}, \text{K(rA)} \quad \text{l.slws} \quad \text{rD}, \text{K(rA)}$$

Description: The format of all secure load instructions (load byte <l.slb>, load half word <l.slh> and load word <l.slw>) is the same as the normal load

instructions (l.lb, l.lh and l.lw) except the opcodes are different for differentiation. The implementation of l.sload instruction is also similar to normal load instruction in various granularity except an additional XORed with the encryption pad. In brief, the offset (immediate value) is sign-extended and added to the contents of general-purpose register rA. The sum represents an effective address. The bytes (in different granularities) in memory addressed by EA are XORed with the corresponding bytes of the encryption pad for decryption, of which the result is loaded into the low-order bits (in different granularities) of general-purpose register rD. High-order bits (if any) of general-purpose register rD are replaced with zero.

Implementation:

Load Byte: $EA \leftarrow \text{exts}(K) + rA[31:0]$
 $rD[7:0] \leftarrow \text{Enc_Pad}[Y:Y-7] \text{ OR } (EA)[7:0]$
 $rD[31:8] \leftarrow 0$

Load Half Word: $EA \leftarrow \text{exts}(K) + rA[31:0]$
 $rD[15:0] \leftarrow \text{Enc_Pad}[Y:Y-15] \text{ OR } (EA)[15:0]$
 $rD[31:16] \leftarrow 0$

Load Word: $EA \leftarrow \text{exts}(K) + rA[31:0]$
 $rD[31:0] \leftarrow \text{Enc_Pad}[Y:Y-31] \text{ OR } (EA)[31:0]$

l.sstore

31	26	25	21	20	16	15	11	10	0
opcode		K		A		B		K	
6 bits		5 bits		5 bits		5 bits		11 bits	

Format:

Store Byte: l.ssb K(rA), rB

Store Half Word: l.ssh rD, K(rA)

Store Word: l.ssw rD, K(rA)

Description: The format of all secure store instructions (store byte <l.ssb>,

store half word <l.ssh> and store word <l.sw>) is the same as the normal store instructions (l.sb, l.sh and l.sw) except the opcodes are different for differentiation. The implementation of l.sstore instruction is similar to normal store instruction in various granularity except an additional XORed with the encryption pad. In brief, the offset (immediate value) is sign-extended and added to the contents of general-purpose register rA. The sum represents an effective address. The lower-order bytes (in different granularities) of general-purpose rB are XORed with the corresponding bytes of the encryption pad for encryption before storing the result to memory location addressed by EA.

Implementation:

Store Byte: $EA \leftarrow \text{exts}(K) + rA[31:0]$

$(EA)[7:0] \leftarrow \text{Enc_Pad}[Y:Y-7] \text{ OR } rB[7:0]$

Store Half Word: $EA \leftarrow \text{exts}(K) + rA[31:0]$

$(EA)[15:0] \leftarrow \text{Enc_Pad}[Y:Y-15] \text{ OR } rB[15:0]$

Store Word: $EA \leftarrow \text{exts}(K) + rA[31:0]$

$(EA)[31:0] \leftarrow \text{Enc_Pad}[Y:Y-31] \text{ OR } rB[31:0]$

set__sr

31	26	25	21	20	16	15	0
opcode		D		A		K	
6 bits		5 bits		5 bits		16 bits	

Format:

set__sr rD, rA, K

Description: Sign-extending the immediate value and then logically ORed with the contents of general-purpose register rD. The result defined the security register into where the contents of general-purpose rA are moved.

Implementation:

$SR(rD \text{ OR } \text{exts}(K)) \leftarrow rA$

load_encKey

31	26	25	21	20	16	15	11	10	0
opcode		D		A		B		K	
6 bits		5 bits		5 bits		5 bits		11 bits	

Format:

load_encKey rD, rA, rB, K

Description: The offset (immediate value) is sign-extended and added to the contents of general-purpose register rA. The sum represents an effective address. The contents of general-purpose register rB represents the total number of bytes that are used in this instruction. Thus, EA and the content of rB specifies the memory space of the data to be used. The bytes in memory addressed by EA are fetched and decrypted using the processor private key before storing the result into the key register defined by the contents of general-purpose register rD.

Implementation:

$$EA \leftarrow \text{exts}(K) + rA[31:0]$$

$$\text{KeyReg}(rD) \leftarrow E_{SK}((EA)[rB:0])$$

switch_encKey

31	26	25	21	20	16	15	0
opcode		K		A		K	
6 bits		5 bits		5 bits		16 bits	

Format:

switch_encKey rA, K

Description: The immediate value is sign-extended before it is logically ORed with the contents of general-purpose rA. The result specifies the target key register, of which its contents is moved to the data path key register to switch

the encryption key used along the same data path.

Implementation:

$$\text{DataPathKey} \leftarrow \text{KeyReg}(\text{rA OR exts(K)})$$

verify_encKey

31	26	25	21	20	16	15	11	10	0
opcode		K		A		B		K	
6 bits		5 bits		5 bits		5 bits		11 bits	

Format:

verify_encKey rA, rB, K

Description: The offset (immediate value) is sign-extended and added to the contents of general-purpose register rA. The sum represents an effective address. The contents of general-purpose register rB represents the total number of bytes that are used in this instruction. Thus, EA and the content of rB specifies the memory space of the data to be used. The data within this memory space is fetched and verified using a verification function. The verification result is stored in a security register which indicates the status of the encryption key verification.

Implementation:

$$\text{EA} \leftarrow \text{exts(K)} + \text{rA}[31:0]$$

$$\text{SR}(\text{Verify_encKey}) \leftarrow \text{Verify}((\text{EA})[\text{rB}:0])$$

sign_cert

31	26	25	21	20	16	15	11	10	0
opcode		D		A		B		K	
6 bits		5 bits		5 bits		5 bits		11 bits	

Format:

sign_cert rD, rA, rB, K

Description: The offset (immediate value) is sign-extended and added to the contents of general-purpose register rA. The sum represents an effective address. The contents of general-purpose register rB represents the total number of bytes that are used in this instruction. Thus, EA and the content of rB specifies the memory space of the data to be used. The bytes in memory addressed by EA are fetched and encrypted using the processor private key. The result is a signed certificate and is stored into a security register.

Implementation:

$$EA \leftarrow \text{exts}(K) + rA[31:0]$$

$$SR(\text{CPU_Cert}) \leftarrow E_{SK}((EA)[rB:0])$$

Bibliography

- [1] R. Usselmann, “Advanced encryption standard/rijndael ip core,” 2015.
- [2] “The architecture of sqlite,” SQLite, 2014. [Online]. Available: <http://www.sqlite.org/arch.html>
- [3] “Survey finds database in the cloud taking over in enterprises,” Trust Catalyst, 2015. [Online]. Available: <http://www.tesora.com/press-releases/survey-finds-database-in-the-cloud-taking-over-in-enterprises/>
- [4] M. Aslett and G. Zwakma, “Next-generation operational databases: 2012-2016,” 451 Research, 2013. [Online]. Available: <https://451research.com/report-long?icid=2852>
- [5] K. Getgen, “2009 encryption and key management industry benchmark report: A risk management benchmark for data protection,” Trust Catalyst, 2009. [Online]. Available: <http://www.trustcatalyst.com/2009EncryptionSurvey.php>
- [6] J. McKendrick, “The rapidly accelerating cloud-enabled enterprise,” 2015. [Online]. Available: <http://www.oracle.com/us/products/database/2015-ioug-survey-db-manageability-2542988.pdf>
- [7] “Online trust alliances finds data breaches spiked to record level in 2013,” Online Trust Alliance, 2014.
- [8] “Google find employees for breaching user privacy,” CNET

- News, 2010. [Online]. Available: <http://www.cnet.com/uk/news/google-fired-engineer-for-privacy-breach/>
- [9] “Amazon relational database service user guide,” Amazon, pp. 143–146, 2016. [Online]. Available: <http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/rds-ug.pdf>
- [10] “Oracle advanced security transparent data encryption best practices,” Oracle, 2012. [Online]. Available: <http://www.oracle.com/technetwork/database/security/twp-transparent-data-encryption-bes-130696.pdf>
- [11] “Trusted platform module specification version 1.2 revision 103,” Trusted Computing Group, 2007. [Online]. Available: <http://www.trustedcomputinggroup.org>
- [12] D. Perez-Botero, J. Szefer, and R. B. Lee, “Characterizing hypervisor vulnerabilities in cloud computing servers,” in *Proceedings of the 2013 International Workshop on Security in Cloud Computing*, 2013, pp. 3–10.
- [13] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007, pp. 552–561.
- [14] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, “Lest we remember: Cold-boot attacks on encryption keys,” *Commun. ACM*, vol. 52, no. 5, pp. 91–98, May 2009.
- [15] M. Gorobets, O. Bazhaniuk, A. Matrosov, A. Furtak, and Y. Bulygin, “Attacking hypervisors via firmware and hardware,” in *blackhat USA 2015*.
- [16] Slashdot, “Wordpress hacked, attackers get root access,” 2011.

- [17] R. Tate, “Why you shouldn’t trust facebook with your data: An employee’s revelations,” 2010.
- [18] C. Gentry, “A fully homomorphic encryption scheme,” Ph.D. dissertation, Stanford, CA, USA, 2009.
- [19] Y. Gahi, M. Guennoun, and K. El-Khatib, “A secure database system using homomorphic encryption schemes,” in *Proceedings of the 3th International Conference on Advances in Databases, Knowledge, and Data Applications*, 2011, pp. 54–58.
- [20] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, “Cryptdb: Protecting confidentiality with encrypted query processing,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 85–100.
- [21] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, “Processing analytical queries over encrypted data,” in *Proceedings of the 39th international conference on Very Large Data Bases*, 2013, pp. 289–300.
- [22] S. Bajaj and R. Sion, “Trusteddb: A trusted hardware based database with privacy and data confidentiality,” in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, 2011, pp. 205–216.
- [23] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan, “Orthogonal security with cipherbase,” in *6th Biennial Conference on Innovative Data Systems Research*, January 2013.
- [24] D. Lie, C. Thekkath, P. Lincoln, M. Mitchell, D. Boneh, J. Mitchell, and M. Horowitz, “Architectural support for copy and tamper resistant software,” in *Proceedings of the ninth international conference on Archi-*

- tectural support for programming languages and operating systems*, 2000, pp. 168 – 177.
- [25] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, “Aegis: Architecture for tamper-evident and tamper-resistant processing,” in *Proceedings of the 17th Annual International Conference on Supercomputing*, 2003, pp. 160–171.
 - [26] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, “Architecture for protecting critical secrets in microprocessors,” in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005, pp. 2–13.
 - [27] D. Evtvyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. Abu Ghazaleh, and R. Riley, “Iso-x: A flexible architecture for hardware-managed isolated execution,” in *Microarchitecture, 2014 47th Annual IEEE/ACM International Symposium on*, Dec 2014, pp. 190–202.
 - [28] B. Rogers, S. Chhabra, M. Prvulovic, and D. Solihin, “Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly,” in *Microarchitecture, 2007 40th Annual IEEE/ACM International Symposium on*, Dec 2007, pp. 183–196.
 - [29] R. J. Santos, J. Bernardino, and M. Vieira, “Balancing security and performance for enhancing data privacy in data warehouses,” in *Trust, Security and Privacy in Computing and Communications, 2011 IEEE 10th International Conference on*, Nov 2011, pp. 242–249.
 - [30] B. H. K. Chen, P. Y. S. Cheung, P. Y. K. Cheung, and Y. K. Kwok, “Cypherdb: A novel architecture for outsourcing secure database processing,” *IEEE Transactions on Cloud Computing*, 2016, to be published.
 - [31] B. H. K. Chen, P. Y. S. Cheung, P. Y. K. Cheung, and Y. Kwok, “An efficient architecture for zero overhead data en-/decryption using reconfig-

- urable cryptographic engine,” in *2015 International Conference on Field Programmable Technology*, 2015, pp. 248–251.
- [32] O. Goldreich, *Foundations of Cryptography: Volume 2, Basic Applications*. New York, NY, USA: Cambridge University Press, 2004.
- [33] D. Champagne and R. Lee, “Scalable architectural support for trusted software,” in *High Performance Computer Architecture, 2010 IEEE 16th International Symposium on*, Jan 2010, pp. 1–12.
- [34] E. Aktas, F. Afram, and K. Ghose, “Continuous, low overhead, run-time validation of program executions,” in *Microarchitecture, 2014 47th Annual IEEE/ACM International Symposium on*, Dec 2014, pp. 229–241.
- [35] T. G. Peter Mell, “The nist definition of cloud computing,” National Institute of Standards and Technology, 2011. [Online]. Available: <http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf>
- [36] J. Yang, Y. Zhang, and L. Gao, “Fast secure processor for inhibiting software piracy and tampering,” in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.
- [37] C. Curino, E. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich, “Relational cloud: A database service for the cloud,” in *5th Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2011.
- [38] “Delivering database as a service (dbaas) using oracle enterprise manager 12c,” Oracle, 2013. [Online]. Available: <http://www.oracle.com/technetwork/oem/cloud-mgmt/dbaas-overview-wp-1915651.pdf>
- [39] J. Cogswell, “Sql vs. nosql: which is better,” 2012. [Online]. Available: <http://insights.dice.com/2012/07/16/sql-vs-nosql-which-is-better/>

- [40] S. Yegulalp, “Not so fast, nosql – sql still reigns,” 2014. [Online]. Available: <http://www.infoworld.com/article/2607910/database/not-so-fast--nosql---sql-still-reigns.html>
- [41] M. T. Ozsü, *Principles of Distributed Database Systems*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.
- [42] P. Paillier, “Public-key cryptosystems based on composite degree residuosity classes,” in *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, 1999, pp. 223–238.
- [43] T. Elgamal, “A public key cryptosystem and a signature scheme based on discrete logarithms,” *Information Theory, IEEE Transactions on*, vol. 31, no. 4, pp. 469–472, Jul 1985.
- [44] A. Boldyreva, N. Chenette, Y. Lee, and A. O’Neill, “Order-preserving symmetric encryption,” in *Advances in Cryptology - EUROCRYPT 2009*. Springer Berlin Heidelberg, 2009, vol. 5479, pp. 224–241.
- [45] “Intel® trusted execution technology,” Intel, 2010. [Online]. Available: <http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf>
- [46] “Arm security technology - building a secure system using trustzone technology,” ARM Limited, 2009. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf
- [47] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, “Nohype: virtualized cloud infrastructure without the virtualization,” in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010, pp. 350–361.
- [48] J. Szefer and R. B. Lee, “Architectural support for hypervisor-secure virtualization,” in *Proceedings of the Seventeenth International Conference*

- on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 437–450.
- [49] Y. Xia, Y. Liu, and H. Chen, “Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks,” in *High Performance Computer Architecture, 2013 IEEE 19th International Symposium on*, 2013, pp. 246–257.
 - [50] Y.-Y. Chen and R. B. Lee, “Hardware-assisted application-level access control,” in *Proceedings of the 12th International Conference on Information Security*, 2009, pp. 363–378.
 - [51] Y.-Y. Chen, P. A. Jamkhedkar, and R. B. Lee, “A software-hardware architecture for self-protecting data,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012, pp. 14–27.
 - [52] C. Gentry, S. Halevi, and N. P. Smart, “Homomorphic evaluation of the aes circuit,” in *In CRYPTO*, 2012.
 - [53] H. Hacigümüş, B. Iyer, C. Li, and S. 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*, 2002, pp. 216–227.
 - [54] T. Ge and S. Zdonik, “Answering aggregation queries in a secure system model,” in *Proceedings of the 33rd International Conference on Very Large Data Bases*, 2007, pp. 519–530.
 - [55] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, “Order preserving encryption for numeric data,” in *International Conference on Management of Data*, 2004, pp. 563–574.
 - [56] M. Lindemann, R. Perez, R. Sailer, L. van Doorn, and S. Smith, “Building the ibm 4758 secure coprocessor,” *Computer*, vol. 34, no. 10, pp. 57–66, 2001.

- [57] S. W. Smith, R. Perez, S. H. Weingart, and V. Austel, “Building a high-performance, programmable secure coprocessor,” in *Computer Networks (Special Issue on Computer Network Security)*, vol. 31, April 1999, pp. 831–860.
- [58] “Ibm 4764 pci-x cryptographic coprocessor,” IBM, 2007. [Online]. Available: <https://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml>
- [59] T. Eisenbarth, T. Güneysu, C. Paar, A.-R. Sadeghi, D. Schellekens, and M. Wolf, “Reconfigurable trusted computing in hardware,” in *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*, 2007, pp. 15–20.
- [60] K. Eguro and R. Venkatesan, “Fpgas for trusted cloud computing,” in *Field Programmable Logic and Applications, 2012 22nd International Conference on*, 2012, pp. 63–70.
- [61] F. J. Krautheim, D. S. Phatak, and A. T. Sherman, “Introducing the trusted virtual environment module: A new mechanism for rooting trust in cloud computing,” in *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, 2010, pp. 211–227.
- [62] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, “Flicker: An execution infrastructure for tcb minimization,” in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2008, pp. 315–328.
- [63] “Securing the enterprise with intel® aes-ni,” Intel, 2010. [Online]. Available: <http://www.intel.com/content/www/us/en/enterprise-security/enterprise-security-aes-ni-white-paper.html>
- [64] “Intel® digital random number generator (drng) software implementation guide,” Intel, 2014. [Online]. Available:

able: https://software.intel.com/sites/default/files/managed/4d/91/DRNG_Software_Implementation_Guide_2.0.pdf

- [65] “Secure microcontroller user’s guide,” Maxim Integrated, 2014. [Online]. Available: <http://pdfserv.maximintegrated.com/en/an/AN4828.pdf>
- [66] R. M. Best, “Microprocessor for executing enciphered programs,” Patent US 4 168 396, 09 18, 1979.
- [67] —, “Preventing software piracy with crypto-microprocessors,” in *Computer Society International Conference*, 1980.
- [68] R. Takahashi and D. Heer, “Secure memory management unit for microprocessor,” Oct. 20 1998, uS Patent 5,825,878.
- [69] B. Candelore and E. Sprunk, “Secure processor with external memory using block chaining and block re-ordering,” Oct. 1 2003, eP Patent App. EP19,980,118,843.
- [70] D. Lie, C. A. Thekkath, and M. Horowitz, “Implementing an untrusted operating system on trusted hardware,” in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003, pp. 178–192.
- [71] G. Suh, C. O’Donnell, and S. Devadas, “Aegis: A single-chip secure processor,” *Design Test of Computers, IEEE*, vol. 24, no. 6, pp. 570–580, Nov 2007.
- [72] J. S. Dwoskin and R. B. Lee, “Hardware-rooted trust for secure key management and transient trust,” in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007, pp. 389–400.
- [73] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, “Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems,” in *Proceedings of the 13th International Conference on*

Architectural Support for Programming Languages and Operating Systems, 2008, pp. 2–13.

- [74] E. Barker, L. Chen, and D. Moody, “Recommendation for pair-wise key-establishment schemes using integer factorization cryptography,” National Institute of Standards & Technology, 2014. [Online]. Available: <http://dx.doi.org/10.6028/NIST.SP.800-56Br1>
- [75] “The tpc-h benchmark,” Transaction Processing Performance Council, 2014. [Online]. Available: <http://www.tpc.org/tpch>
- [76] C. Adams, *Encyclopedia of Cryptography and Security*. Boston, MA: Springer US, 2011, pp. 596–596. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-5906-5_80
- [77] H. Lipmaa, P. Rogaway, and D. Wagner, “Comments to nist concerning aes-modes of operations : Ctr-mode encryption,” in *Symmetric Key Block Cipher Modes of Operation Workshop*, 2000.
- [78] “Specification for the advanced encryption standard (aes),” Federal Information Processing Standards Publication 197, 2001. [Online]. Available: <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>
- [79] A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar, “An fpga implementation and performance evaluation of the aes block cipher candidate algorithm finalists,” in *AES Candidate Conference*, 2000, pp. 13–27.
- [80] M. Dworkin, “Sp 800-38a . recommendation for block cipher modes of operation: Methods and techniques,” National Institute of Standards & Technology, Gaithersburg, MD, United States, 2001.
- [81] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier, “Fast updates on read-optimized databases using multi-core cpus,” *Proc. VLDB Endow.*, vol. 5, no. 1, pp. 61–72, Sep. 2011.

- [82] “Sqlite homepage,” SQLite, 2014. [Online]. Available: <http://www.sqlite.org/>
- [83] D. Liu and S. Wang, “Programmable order-preserving secure index for encrypted database query,” in *Cloud Computing, 2012 IEEE 5th International Conference on*, June 2012, pp. 502–509.
- [84] R. A. Popa, F. H. Li, and N. Zeldovich, “An ideal-security protocol for order-preserving encoding,” in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013, pp. 463–477.
- [85] T. G. Wayne Jansen, “Guidelines on security and privacy in public cloud computing,” National Institute of Standards & Technology, Dec 2011.
- [86] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, “Dbmss on a modern processor: Where does time go?” in *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999, pp. 266–277.
- [87] T. Austin, E. Larson, and D. Ernst, “SimpleScalar: An infrastructure for computer system modeling,” *Computer*, vol. 35, no. 2, pp. 59–67, Feb. 2002.
- [88] S. Loosemore, R. M. Stallman, R. McGrath, A. Oram, and U. Drepper, “The gnu c library reference manual,” Free Software Foundation, 2015.
- [89] “A deep dive into cma,” 2012. [Online]. Available: <http://lwn.net/Articles/486301>
- [90] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, “Secure program execution via dynamic information flow tracking,” in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004, pp. 85–96.
- [91] Y.-Y. Chen, P. A. Jamkhedkar, and R. B. Lee, “A software-hardware

- architecture for self-protecting data,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012, pp. 14–27.
- [92] S. Zdancewic, “Challenges for information-flow security,” in *In Proc. Programming Language Interference and Dependence*, 2004. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.128.6747>
- [93] D. A. McGrew, “Counter mode security: Analysis and recommendations,” 2002. [Online]. Available: <http://www.mindspring.com/~dmcgrew/ctr-security.pdf>
- [94] A. Edvardsson, “Sdc/mmcccontroller design document,” 2009. [Online]. Available: http://opencores.org/project,sdcard_mass_storage_controll
- [95] R. B. Lee and Y.-Y. Chen, “Processor accelerator for aes,” in *Proceedings of the 2010 IEEE 8th Symposium on Application Specific Processors*, 2010, pp. 16–21.
- [96] “Opencores community,” 2014. [Online]. Available: <http://opencores.org>
- [97] “The sqlite virtual machine,” SQLite, 2014. [Online]. Available: <http://www.sqlite.org/opcode.html>
- [98] M. Shao, A. Ailamaki, and B. Falsafi, “Dbmbench: Fast and accurate database workload representation on modern microarchitecture,” in *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, 2005, pp. 254–267.
- [99] “SignalTap ii with verilog designs,” Altera Coporation, 2012. [Online]. Available: ftp://ftp.altera.com/up/pub/Altera_Material/12.1/Tutorials/Verilog/SignalTap.pdf