An Implementation of CypherDB Processor Architecture to Protect Database Privacy in DBaaS

Bony H. K. Chen*[1], Paul Y. S. Cheung†[1], Peter Y. K. CheunC‡[2], and Yu-Kwong Kwok§[1]

[1]Department of Electrical and Electronic Engineering, The University of Hong Kong. Pokfulam Road. Hong Kong
[2]Department of Electrical and Electronic Engineering, Imperial College London. SW7 2AZ London. U.K.

Technical Report

Abstract

Security, particularly data privacy, is one of the biggest barriers toward 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 the data confidentiality from any honest-but-curious administrator. The prior effort on addressing this security problem is either prohibitively slow or highly restrictive in operations. We introduce CypherDB, a novel secure processor approach that uses cost-effective HW/SW co-design protection mechanisms to protect the confidentiality of an outsourced database processing in Cloud Computing environment with high performance. The protection mechanisms ensure that all sensitive data residing in off-chip memory are encrypted, and allow efficient en-/decryption and computation in the processor chip.

To evaluate its practicability and performance in real hardware platform, we implement the fully functional CypherDB secure processor on Field-Programmable Gate Array (FPGA). SQLite, a real database program, is modified with CypherDB secure instructions to be executed on our implemented FPGA platform. In this paper, the FPGA implementation detail of CypherDB secure processor and the modification on the SQLite are discussed.

*chk3e@hku.hk
†paul.cheung@hku.hk
‡p.cheung@imperial.ac.uk
§ricky.kwok@hku.hk

1 Introduction

In DBaaS, the DataBase Management System (DBMS) is installed in the host computer in the cloud. The users only need to upload their database, request and pay for the service while the Cloud Service Provider (CSP) is responsible for all the database management and administration work. The major challenge to protect data privacy is such service model is that the outsourced database is being executed in a remote machine, where the users lose control of their outsourced data. The key solution to protect data privacy is to encrypt the data without revealing the decryption key to the CSP. Homomorphic encryption [1, 2], which permits computation on encrypted data without receiving the decryption key, can successfully secure the data in the Cloud. However, these approaches are either prohibitively slow [1] or highly restrictive in arithmetic operations [2]. Alternatively, the decryption key and the decrypted data can still be stored in the cloud server but within a self-contained and tamper-resistance co-processor [3, 4]. These approaches requires an extra processing system which is usually with low computational power and expensive. Lastly, the computer architecture community has proposed a secure processor approach [5–9] which stores the decryption key in main processor and provides mechanism to protect the data in off-chip memory. Nevertheless, these approaches initially targets on Digital Right Management (DRM) and portable devices which protect the application program and data as a whole. This highly prohibits the dynamic data movement and parallel processing in cloud computing environment. How to make use of secure processor to protect data confidentially in cloud computing environment, especially on securing cloud database service, is still largely unknown and remains a significant challenge.

To be applicable in practical cloud computing environment, we propose a novel cloud system architecture called CypherDB [10, 11], which makes a practical use of secure processor to protect cloud database service. We develop a cost-effective protection mechanism that can secure an outsourced database being executed in cloud computing environment with high performance. While the CypherDB architecture is extensively studied in [10, 11], the major contribution of this work is a fully functional FPGA implementation of the CypherDB secure processor, where several optimization techniques and challenges of the implementation are presented. To the best of our knowledge, CypherDB is the first fully functional implementation prototype to use processor architectural design to successfully protect remote operation on encrypted database against any honest-but-curious administrator.

We implemented the look-ahead encryption scheme and the CypherDB secure processor on a Field Programmable Gate Array (FPGA) to validate and evaluate our designs. We also took a concrete example of a database application (SQLite [12]) to investigate the practicability and the performance impact of our proposed solution. The hardware design is specified in Verilog and is synthesized using 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 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.

2 Overview

Our implementation is based on the OR1200 core from 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 is 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 application and Linux application using newlib and uClibc libraries, respectively. The toolchain support can compile the database application program to be ran on Linux, which abstracts the file management layer to store the database.

Figure 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.

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

Since 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. Due to the fact that the OpenRISC project does not provide SD card controller implementation support, the implementation in [13] is employed. A self-developed performance counter is

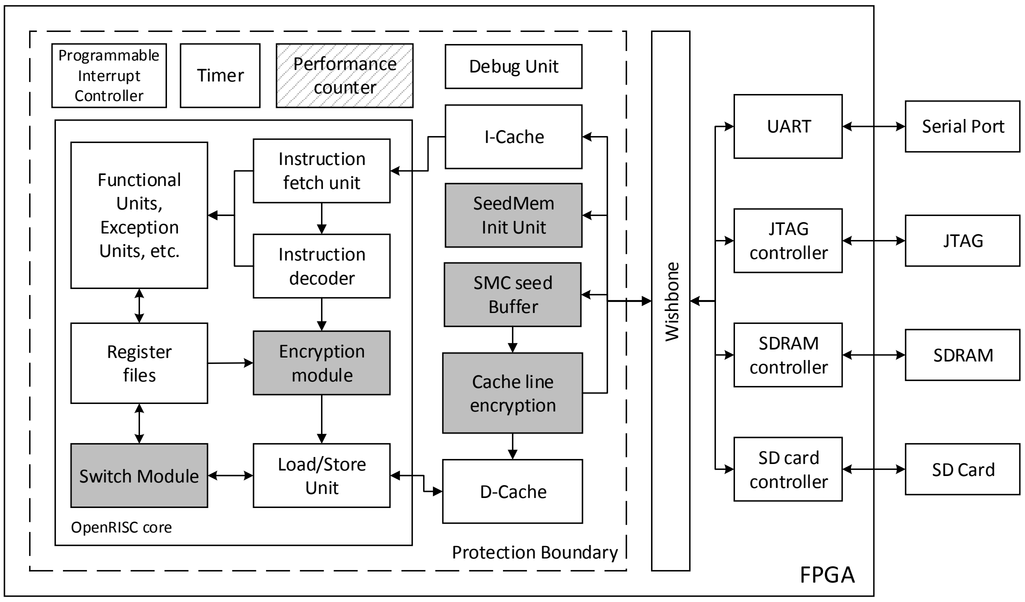


Figure 1: An overview of CypherDB secure processor implementation. The gray compoents are the additional security modules implemented in our design. A performance counter (shaded) is also implemented to measure the evaluation metrics in hardware.

also implemented to measure the evaluation metrics in hardware.

2.1 Impact to Processor Pipeline

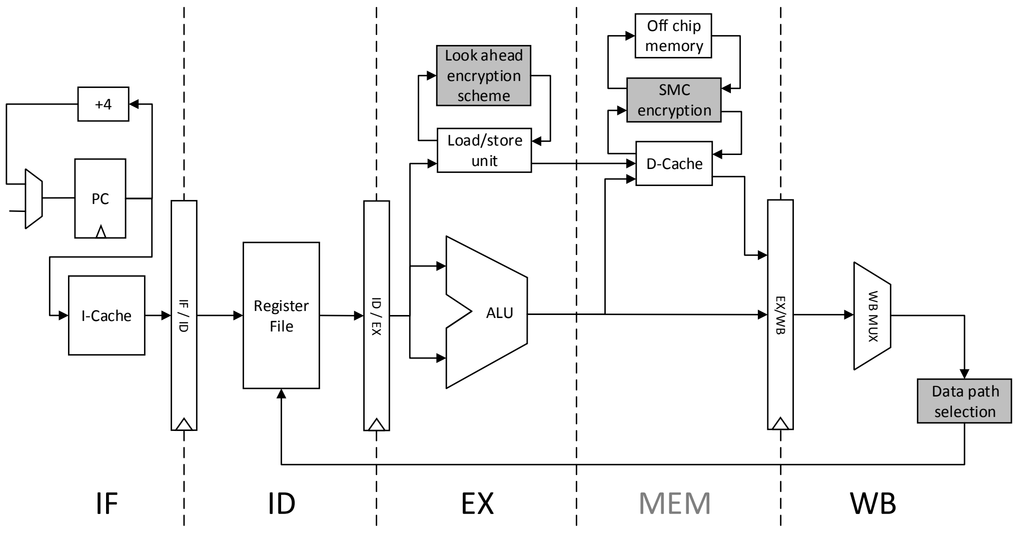


Figure 2: A diagram showing the impact of our implemented modules towards the processor pipeline. The gray components represent our additional security modules for the CypherDB secure processor.

Figure 2 shows how the additional security modules of our design influence the processor 4-stage pipeline. The security modules that used to implement the look ahead encryption scheme are at EX stages and the data path selection module is at WB stage. The SMC encryption modules are at EX stage (or MEM in traditional 5-stage pipeline), which are operating between data cache and off-chip memory. Therefore, the operation latency of these security modules could have a direct impact to the processor pipeline.

3 Processor Core

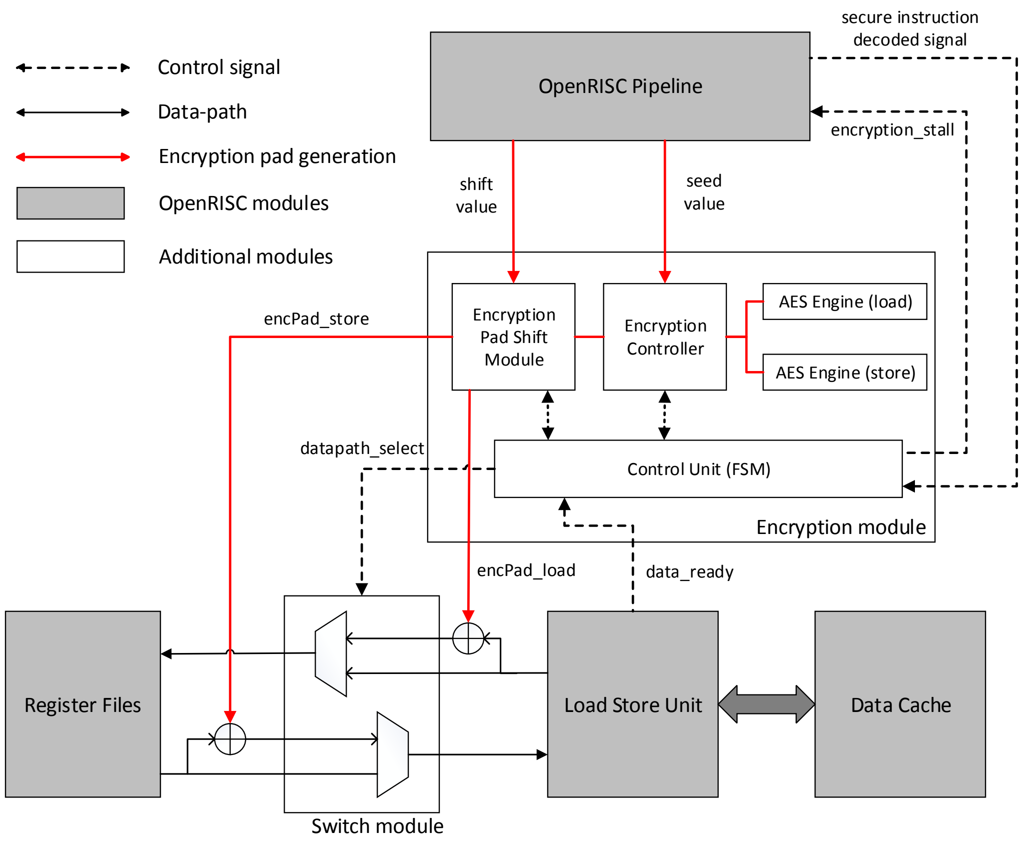


Figure 3: A Block diagram of the CypherDB secure processor core implementation.

The goal of our processor core design is to support the look-ahead encryption scheme. Figure 3 depicts a high-level overview of our secure processor core implementation. The blocks in gray 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, of which 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 to 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.

3.1 Encryption Engines

Advanced Encryption Standard (AES) is one of the most widely used symmetric encryption which has outstanding performance in hardware [14, 15]. We employ the open source AES core [16] released in the OpenCore Community [17]. The implemented 128-bit AES engine can complete the encryption in 12 clock cycles at maximal 160MHz clock frequency on our FPGA platform.

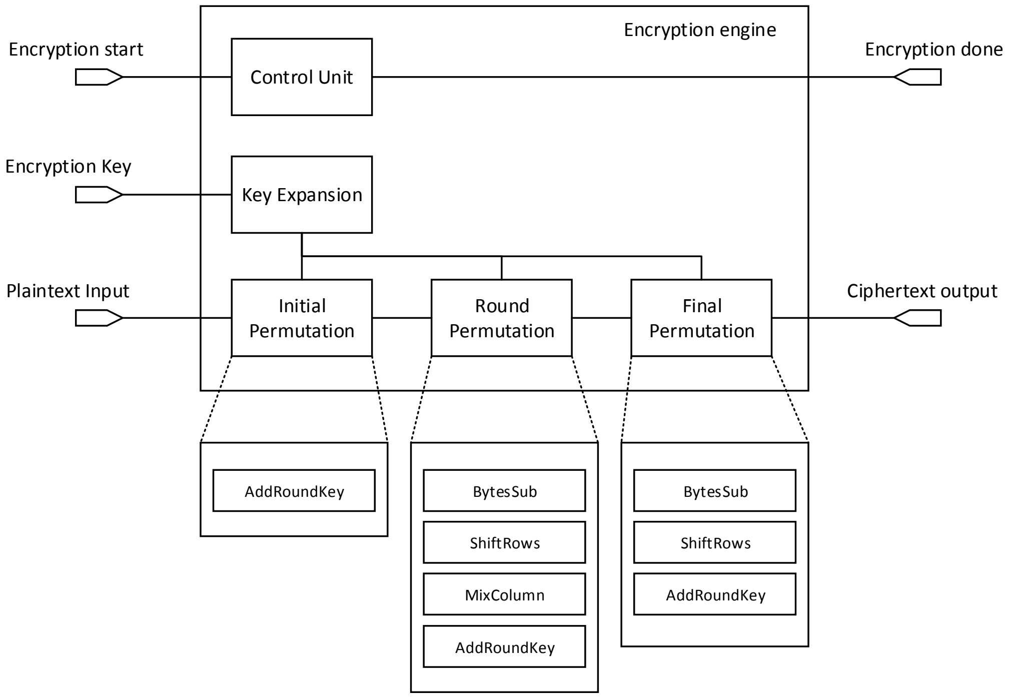
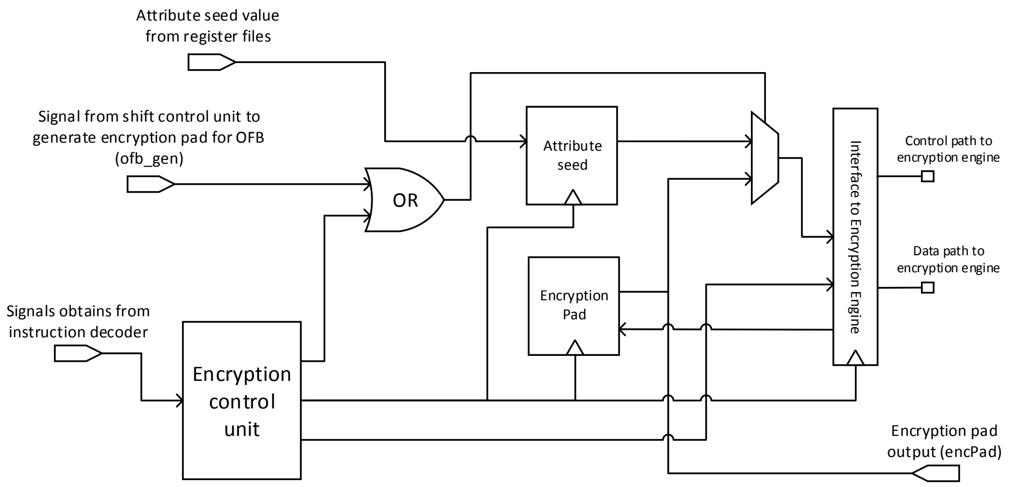


Figure 4: The AES encryption engine employed in our implementation [16].

Figure 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 4 transformation founctions: AddRoundKey, BytesSub, ShiftRows, and MixColumn. Two separate encryption engines for load and store data path are implemented to avoid any resource contention.

3.2 Encryption Controller



Attribute seed value from register files
Signal from shift control unit to generate encryption pad for OFB (ofb_gen)
Signals obtains from instruction decoder
OR
Attribute seed
Encryption Pad
Encryption control unit
Interface to Encryption Engine
Control path to encryption engine
Data path to encryption engine
Encryption pad output (encPad)

Figure 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 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 control the data input to the encryption engine. The control unit decodes the *l.seed* instruction to store the attribute seed value to 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.

3.3 Encryption Pad Shift Module

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

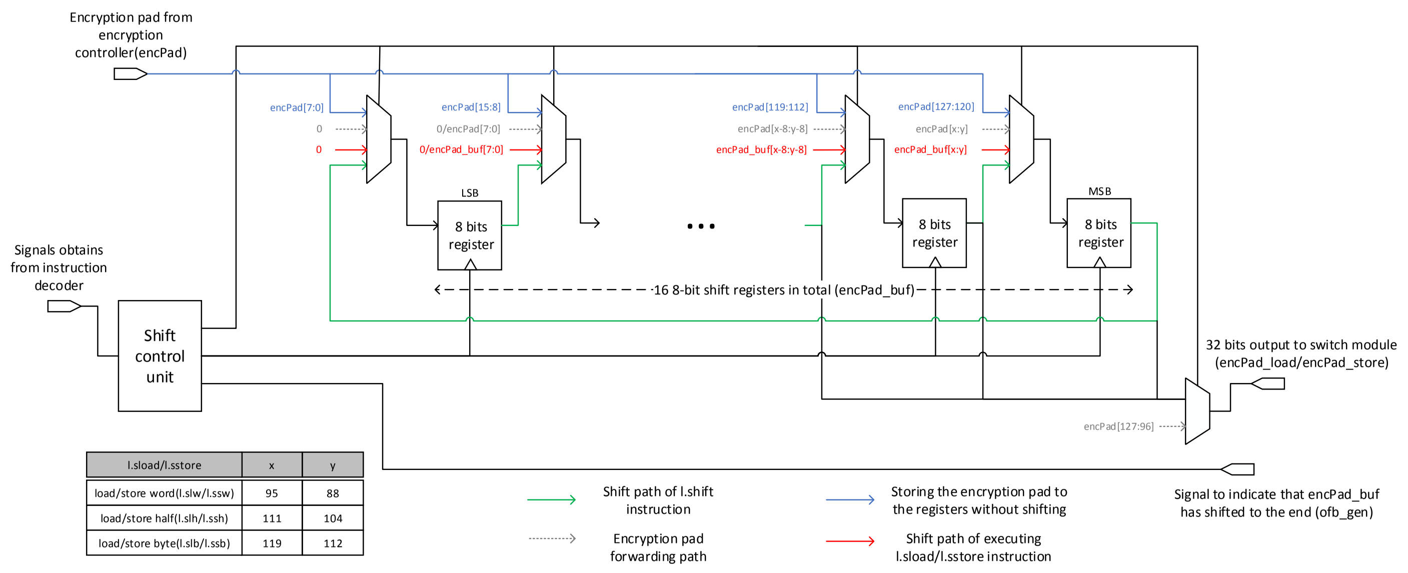
Since the shift module is within the processor pipeline, its operation latency is critical to the overall performance. The most performance critical scenario is that two $l.sload$ (or two $l.sstore$) instructions or $l.sload/l.sstore$ and $l.shift$ instructions are executed consecutively. This requires the shifting operation to be completed in two processor cycles (or one processor cycle in 5-stage pipeline) (see Section 2.1). A näive implementation is to shift the 128-bit encryption pad with arbitrary bits in one single cycle. However, such implementation could consume a lot of resource (almost half 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 $l.sload/l.sstore$ instruction is much more often that $l.shift$ instruction. In fact, our implementation on SQLite shows that a careful design can eliminate the use of $l.shift$ instruction to shift the encryption pad. Therefore, our design only supports quick shifting operation for the $l.sload/l.sstore$ 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 $4\times$ as compared to the näive approach. Figure 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 3.6. 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 3)

3.4 Data Path Switching Module

Since the switch module is located at the memory access critical path, it has to be simple and fast. It consists of only two 2-to-1 multiplexers at 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 Exclusive-OR (XOR) operation, the added latency is only the accumulated combinational logic delay of the multiplexers and XOR gates.

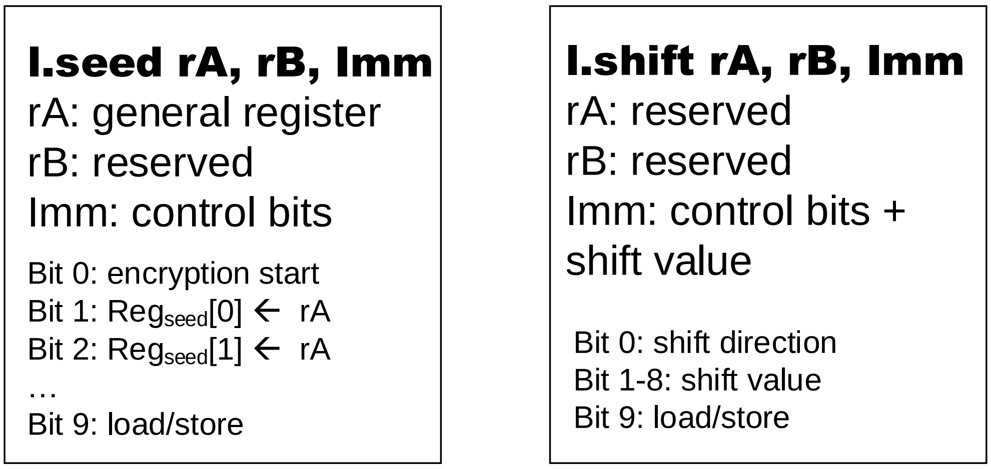
Upon detecting any secure load/store instruction, the processor pipeline sends a signal to the control unit. The control unit schedules these signals with the data arrival signal ($data\_ready$) from the load store unit. It then controls the multiplexers in the data-path switch module accordingly.


Encryption pad from encryption controller(encPad)
Signals obtains from instruction decoder
Shift control unit
encPad[7:0]
0
0
encPad[15:8]
0/encPad[7:0]
0/encPad_buf[7:0]
LSB
8 bits register
encPad[119:112]
encPad[x-8:y-8]
encPad_buf[x-8:y-8]
encPad[127:120]
encPad[x:y]
encPad_buf[x:y]
MSB
8 bits register
8 bits register
16 8-bit shift registers in total (encPad_buf)
32 bits output to switch module (encPad_load/encPad_store)
encPad[127:96]
Signal to indicate that encPad_buf has shifted to the end (ofb_gen)
Shift path of l.shift instruction
Encryption pad forwarding path
Storing the encryption pad to the registers without shifting
Shift path of executing l.sload/l.sstore instruction

l.sload/l.sstore	x	y
load/store word(l.slw/l.ssw)	95	88
load/store half(l.slh/l.ssh)	111	104
load/store byte(l.slb/l.ssb)	119	112

Figure 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.

3.5 Control Unit



l.seed rA, rB, Imm
rA: general register
rB: reserved
Imm: control bits

Bit 0: encryption start
Bit 1: $Reg_{seed}[0] \leftarrow rA$
Bit 2: $Reg_{seed}[1] \leftarrow rA$
…
Bit 9: load/store

l.shift rA, rB, Imm
rA: reserved
rB: reserved
Imm: control bits + shift value

Bit 0: shift direction
Bit 1-8: shift value
Bit 9: load/store

Figure 7: The structure and format of the custom *l.seed* and *l.shift* instructions.

The purpose of the control unit is to setup 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 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.

Figure 7 illustrates the format of the *l.seed* and *l.shift* instructions. Both instructions contain a 10-bit control vector as the immediate value in the operand. The *l.seed* instruction additionally specifies the general purpose register in order to store the attribute seed value from register file into the attribute seed buffer Reg_{seed}. The control unit decodes the 10-bit control vector for various operations (e.g. select load/store encryption path, trigger encryption engine etc).

3.6 Pipeline Stall

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

3.7 OpenRISC Core Modification

A minor modification of OpenRISC is necessary to support the look-ahead encryptoin 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:* 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 $l.sload/l.sstore$ instruction are handled as normal $l.load/l.store$ instruction excepts 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 has to be fetched from the off-chip Dynamic Random Access Memory (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 8 shows the timing diagram of this subtle fault 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. Therefore, 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 setup two data paths for *DATA* and *DATA_BUF*.

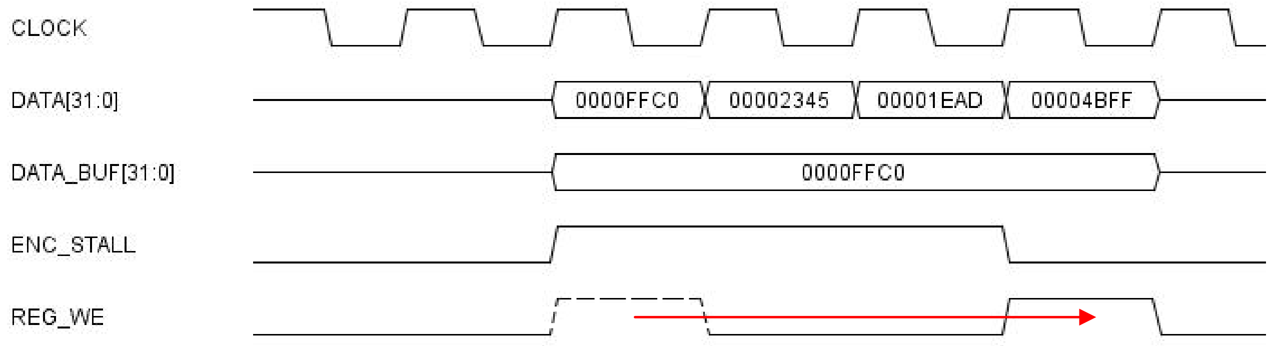


Figure 8: A diagram showing the timing error without the data buffer in the load store unit. *DATA* is the incoming data from the data cache. *DAT_BUF* is the data buffer implemented to tackle this problem. *ENC_STALL* asserts when the data is ready but the encryption has not yet finished. *REG_WE* is the control signal to write *DATA* into the register.

4 SMC Encryption Modules

The goal of our SMC encryption modules design is to encrypt the cache line in SMC for setting up data-path ③ as described in [10]. Since the data cache access is heavily relied on the behaviour of the SMC seed buffer access, the challenge of the actual implementation is thus on 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 resource 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.

4.1 SMC Seed Buffer

The SMC seed buffer is with the same structure as the data cache – 1-way direct-mapped cache architecture, each with 16-byte line size. Both seed buffer and data cache are operating at write-back mode.

4.2 Cache Controller

The implementation of the cache controller can be best illustrated using a state diagram. The overall implementation consists of 12 states, of which 4 additional states are introduced to perform SMC seed buffer access. For the 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 10 presents the state diagram for the execution along data path 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 8 states, where the function of each of these state is described in Table 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 as 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. Our implementation to realize such execution flow involves 4 additional states in addition to the original cache controller, which is described in Table 2. The modification of the state diagram involves redirecting some execution to access the seed buffer before the actual designated execution to be performed. Figure 11 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 10 and Figure 11 are from the same FSM, where the original 8 states are reused in both execution. The execution is multiplexed at *LOADSTORE*, *LOOP3* and *FLUSH* states. The control signal is generated in the circuit as shown in Figure 9, which checks whether the secure execution has been setup and whether the cache line is within the SMC.

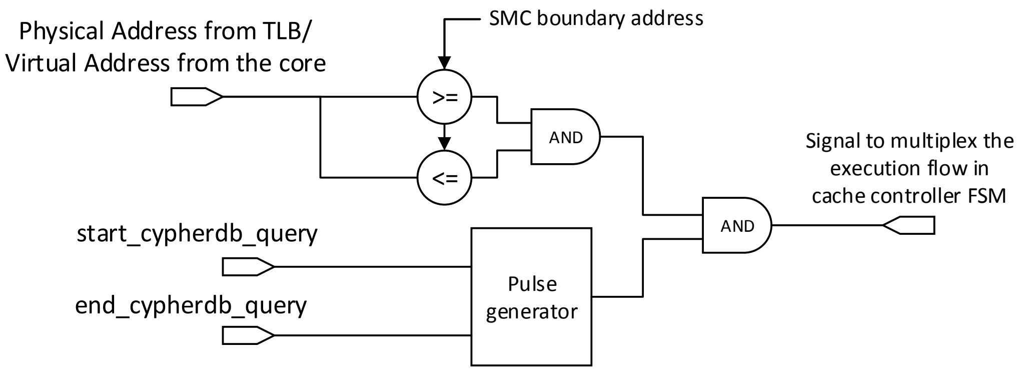

SMC boundary address
Physical Address from TLB/ Virtual Address from the core
>=
<=
AND
AND
Signal to multiplex the execution flow in cache controller FSM
start_cypherdb_query
end_cypherdb_query
Pulse generator

Figure 9: 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 setup via *start_cypherdb_query* instruction and 2) the cache line is within SMC.

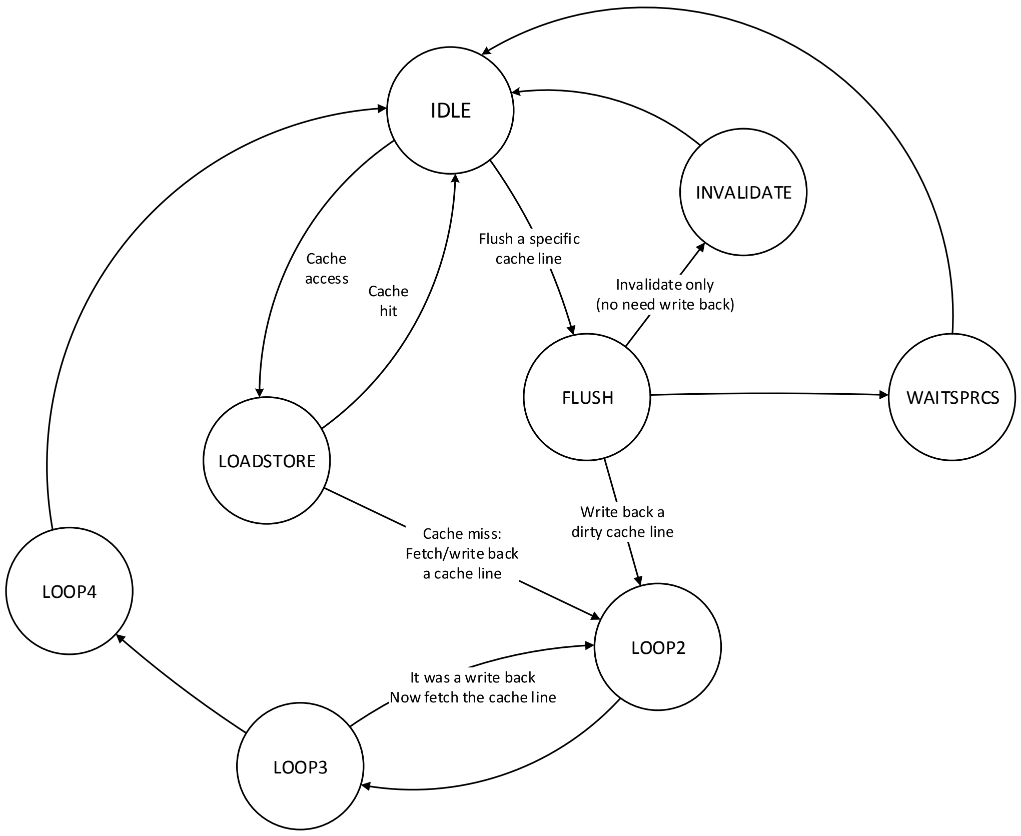

IDLE
INVALIDATE
FLUSH
WAITSPRCS
LOADSTORE
LOOP4
LOOP2
LOOP3
Cache access
Cache hit
Flush a specific cache line
Invalidate only (no need write back)
Write back a dirty cache line
Cache miss: Fetch/write back a cache line
It was a write back Now fetch the cache line

Figure 10: A FSM diagram of the original cache controller from the OpenRISC project. This is used for the execution along data path 1 and 2.

Table 1: FSM states involved in the execution along data path 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 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.

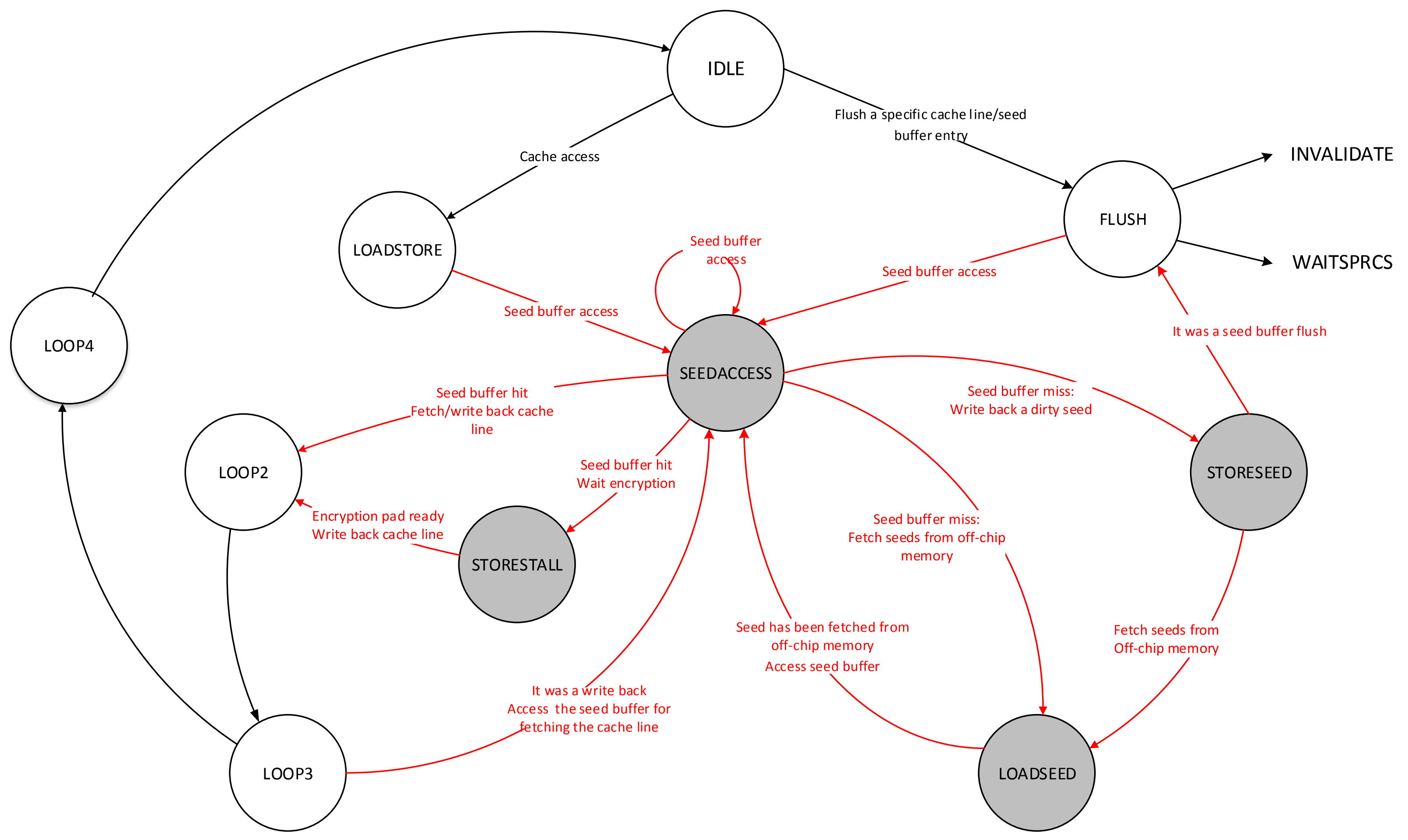


Figure 11: A FSM diagram of the cache controller for execution along data path 3. Four additional states (in gray) are added in the FSM and the execution is redirected to these additional states. The red arrow represents the re-directed or additional execution flow.

4.3 Cache Line Encryption

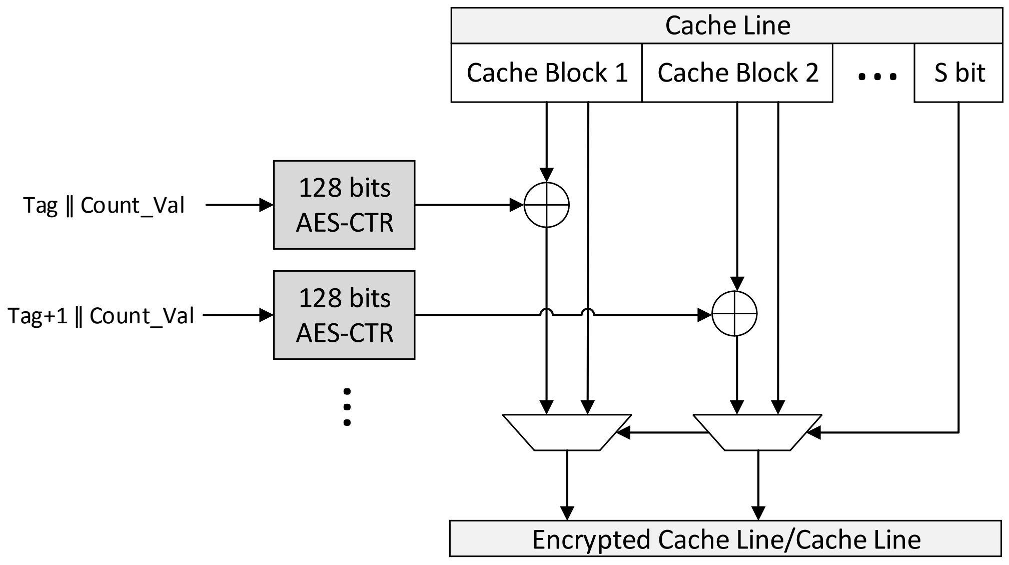


Figure 12: A diagram showing a cache line is encrypted with multiple encryption engines in parallel. The S-bit associated with each cache line is used to determine whether the cache line needs to be encrypted.

Figure 12 depicts the cache line encryption mechanism in hardware. Upon the availability of the *count_val* in 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 9 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.

Since the SMC should be allocated to a block of physically contiguous memory block. The tool used for allocating contiguous memory region is however not available in our evaluation setup. We takes an alternative approach that uses virtual memory address from the core to setup the SMC. The SMC boundary address is thus the starting address of stack and heap memory. The process id is further used to ensure that the virtual address used is from the same process.

4.4 SeedMem Initialization Unit

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

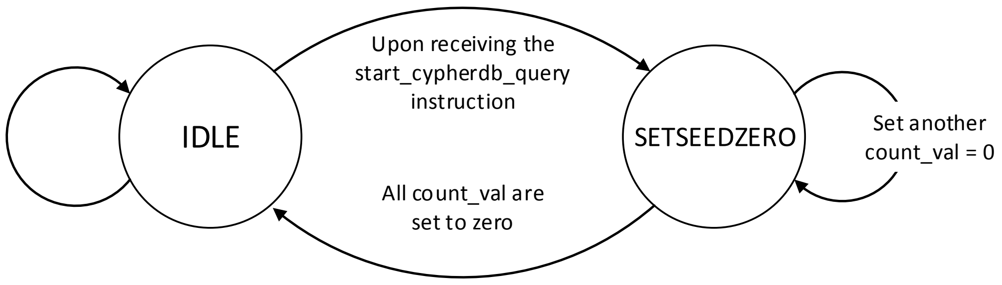

Upon receiving the start_cypherdb_query instruction
IDLE
SETSEEDZERO
Set another count_val = 0
All count_val are set to zero

Figure 13: A stage diagram describing the operation of the SeedMem Initialization Unit.

5 SQLite Modification

SQLite is an embedded Structured Query Language (SQL) database engine which does not have a separate sever process. It is the most widely deployed database 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 the 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 on the source code is made possible. Last, it consists of a SQL database engine which can parse and execute standard SQL query over a relational database. The study on SQLite application can thus be extended to other SQL database engine easily.

To apply the look ahead encryption scheme into a practical database application, the modification of SQLite mainly involves two components: 1) issuing $l.seed$ instruction to store attribute seed into Reg_{seed} in the processor, 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" statements is used as a SQL query example to illustrate how the attribute seeds are identified practically. Finally, the modification on the data access layer is discussed.

5.1 Background: Virtual Database Engine

Figure 14 depicts the architecture of SQLite. The VDBE implements an abstract computing engine to perform SQL specified execution 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 labeled as P1, P2 and P3. Operand P1 and P2 is 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 [18] for the detail description of each opcode used in SQLite.

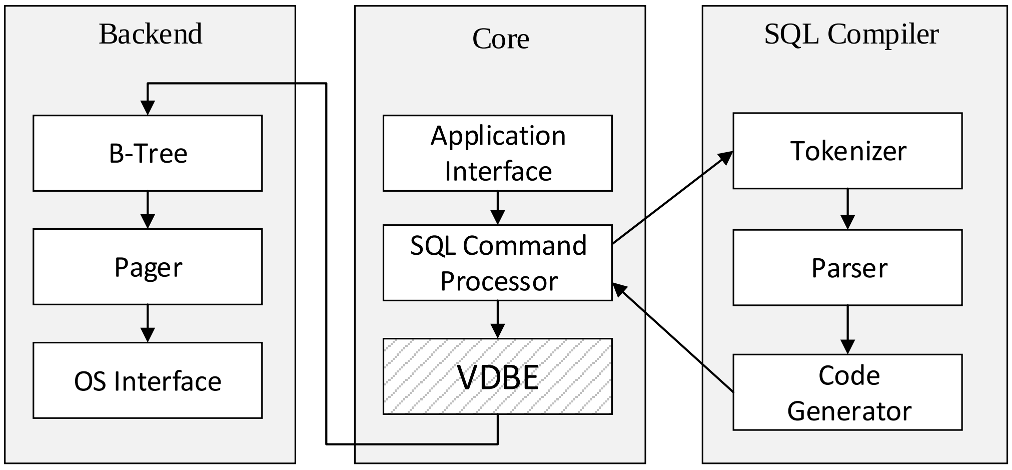


Figure 14: The architecture of SQLite [19].

5.2 Attribute Seed

Since the attribute seed of each attribute data is ($databaseID \parallel tableID \parallel rowID \parallel columnID \parallel cntr$). 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 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 record 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 “EXPLAIN" command, is shown in Table 3. The instructions is 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. 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 3: The VDBE program of the query statement "SELECT * FROM region;". The bolded number is used to be 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. Jumpt 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

5.3 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 en-/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 Database Management System (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 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 performance penalty on copying the attribute data into the redirected buffer, it should be noted that this is solely an implementation issue. A 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.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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 Architectural support for programming languages and operating systems*, 2000, pp. 168 – 177.

[6] 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.

[7] 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.

[8] 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.

[9] D. Evtyushkin, 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.

[10] 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.

[11] 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 *2015 International Conference on Field Programmable Technology*, 2015, pp. 248–251.

[12] "Sqlite homepage," SQLite, 2014. [Online]. Available: http://www.sqlite.org/

[13] A. Edvardsson, "Sdc/mmccontroller design document," 2009. [Online]. Available: http://opencores.org/project,sdcard_mass_storage_controlle

[14] "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

[15] 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.

[16] R. Usselmann, "Advanced encryption standard/rijndael ip core," 2015.

[17] "Opencores community," 2014. [Online]. Available: http://opencores.org

[18] "The sqlite virtual machine," SQLite, 2014. [Online]. Available: http://www.sqlite.org/opcode.html

[19] "The architecture of sqlite," SQLite, 2014. [Online]. Available: http://www.sqlite.org/arch.html