

SCA-Resistant and High-Performance Embedded Cryptography Using Instruction Set Extensions and Multi-Core Processors

Zhimin Chen

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirement for the degree of

Doctor of Philosophy
in
Computer Engineering

Patrick R. Schaumont, Chair
Claudio da Silva
Michael S. Hsiao
Leyla Nazhandali
Kazuo Sakiyama
Danfeng Yao

June 17, 2011

Blacksburg, Virginia

Keywords: Embedded Security, Side-Channel Attack,
Virtual Secure Circuit, Secure Processor

SCA-Resistant and High-Performance Embedded Cryptography Using Instruction Set Extensions and Multi-Core Processors

Zhimin Chen

Abstract

Nowadays, we use embedded electronic devices in almost every aspect of our daily lives. They represent our electronic identity; they store private information; they monitor health status; they do confidential communications, and so on. All these applications rely on cryptography and, therefore, present us a research objective: how to implement cryptography on embedded systems in a trustworthy and efficient manner.

Implementing embedded cryptography faces two challenges - constrained resources and physical attacks. Due to low cost constraints and power budget constraints, embedded devices are not able to use high-end processors. They cannot run at extremely high frequencies either. Since most embedded devices are portable and deployed in the field, attackers are able to get physical access and to mount attacks as they want. For example, the power dissipation, electromagnetic radiation, and execution time of embedded cryptography enable Side-Channel Attacks (SCAs), which can break cryptographic implementations in a very short time with a quite low cost.

In this dissertation, we propose solutions to efficient implementation of SCA-resistant and high-performance cryptographic software on embedded systems. These solutions make use of two state-of-the-art architectures of embedded processors: instruction set extensions and multi-core architectures. We show that, with proper processor micro-architecture design and suitable software programming, we are able to deliver SCA-resistant software which performs well in security, performance, and cost. In comparison, related solutions have either high hardware cost or poor performance or low attack resistance. Therefore, our solutions are more practical and see a promising future in commercial products. Another contribution of our research is the proper partitioning

of the Montgomery multiplication over multi-core processors. Our solution is scalable over multiple cores, achieving almost linear speedup with a high tolerance to inter-core communication delays. We expect our contributions to serve as solid building blocks that support secure and high-performance embedded systems.

This dissertation is dedicated to
my wife *Qian Liu* for her loving support and encouragement,
and to my parents and parents-in-law for all their care and support.

Acknowledgements

First and foremost I would like to sincerely thank my advisor, Dr. Patrick Schaumont, for his guidance, support, and advice. His example of dedication, enthusiasm, and integrity is the most valuable lesson I have learnt through the years. I truly believe that working with him for four years has helped me to achieve my original goal when I came to the U.S., that is to do meaningful and enjoyable research. I also believe that Dr. Schaumont's personal characteristics have significant influence on me which will definitely benefit me in my future life and career.

My gratitude also goes to other members in my Ph.D. advisory committee, Dr. Leyla Nazhanlali, Dr. Michael Hsiao, Dr. Kazuo Sakiyama, Dr. Danfeng Yao, and Dr. Claudio da Silva. I am lucky to have Dr. Claudio da Silva as my teacher in the Stochastic Signals and Systems course, which helped me a lot with my research. I am also grateful to Dr. Danfeng Yao who provided me with valuable advices and help regarding my future career. Thanks also go to Dr. Kazuo Sakiyama, Dr. Michael Hsiao, and Dr. Leyla Nazhanlali for offering helpful feedbacks to my research work.

I would also like to thank Dr. Alessandro Forin and Neil Pittman for the opportunity they gave me to do research on processor microarchitecture during my internship at Microsoft Research. Their mentorship advanced my professional knowledge a lot. The summer I spent in Seattle is one of the most joyful summers I have ever had.

I also deeply appreciate the help from my coworkers and labmates, Ambuj Sinha, Xu Guo, Abhranil Maiti, Michael Gora, Sergey Morozov, Sachin Hirve, Suvarna Mane, and Srikrishna Iyer. Without help from Ambuj Sinha, my research would not have been

so smooth. Suggestions and support from other labmates are also valuable to refine my work. Most importantly, all the labmates have made our laboratory an enjoyable place to work at.

My thanks also go to my beloved friends, Xu Guo, Tong Liu, Daocheng Huang and Ruxi Wang. Through the years, especially right after landing at the U.S. for the first time, their being-around prevented me from feeling lonely. They have been making my life more colorful.

Finally, special words go to my beloved friends, Wenchao Li, Ruirui Gu, Shaoshan Liu, Zhanpeng Jin, and Weiqin Ma. From time to time, I cannot help recalling the exciting moments we spent together.

Contents

1	Introduction	1
1.1	Security on Embedded Systems	2
1.2	Secure Embedded Cryptography Design Challenges	4
1.3	Related Work - A Brief Review	5
1.3.1	Solutions to Higher Performance	5
1.3.2	Solutions to Stronger Security	7
1.4	Our Contributions	8
1.5	Organization	11
2	Preliminaries	13
2.1	Side-Channel Attacks and Countermeasures	13
2.1.1	SCA Concept	13
2.1.2	An SCA Example	15
2.1.3	SCA Countermeasures	20
2.2	State-of-the-Art Embedded Microarchitectures	24
2.2.1	Instruction Set Extensions	24
2.2.2	Multi-core Processors	26
2.3	Conclusion	27
3	Slicing Attack on A ‘Perfectly’ Masked AES	30
3.1	A ‘Perfect’ Algorithmic Masking on AES	30

3.2	Slicing Attack	31
3.3	Experimental results	34
3.4	Conclusion	39
4	Higher-Order Circuit Effects Break Secure Circuits	41
4.1	Analysis on Masked Circuits	42
4.1.1	Two Conditions for Perfect Masking	42
4.1.2	Higher-Order Circuit Effects Causing Side-Channel Leakage	44
4.2	Experimental Results	48
4.3	Conclusion	54
5	SCA-Resistant Software on Dual-Core Processor	56
5.1	Virtual Secure Circuit (VSC)	57
5.1.1	Micro-Processor Assumptions and Side-Channel Leakage	57
5.1.2	Dual-Core Architecture for VSC	60
5.1.3	Creating <i>Program</i> and $\overline{Program}$	61
5.1.4	VSC Is Equivalent to DRP Circuit	63
5.2	Synchronization of two cores	63
5.3	Implementation Details on VSC-AES	68
5.4	VSC AES Prototype Resists SCA	71
5.4.1	VSC AES Prototype	71
5.4.2	Results	72
5.5	Discussion	75
5.6	Previous Works	79
5.7	Conclusion	80
6	SCA-Resistant Software Using Instruction Set Extensions: Initial So-	
	lution	81
6.1	Virtual Secure Circuit on A Balanced Processor	82
6.1.1	Implementation	84

6.1.2	Security Analysis	88
6.2	VSC Programming	88
6.2.1	Bitslice Programming	90
6.2.2	VSC Programming Based on Bitslice	92
6.3	Experimental Results	93
6.3.1	Experimental Setup	94
6.3.2	Results	94
6.3.3	Analysis Based on Results	98
6.4	Conclusion	100
7	SCA-Resistant Software Using Instruction Set Extensions: Improved Solution	101
7.1	An Improved Protection Solution	103
7.1.1	Secure Processor	103
7.1.2	Programming Method	107
7.2	Experimental Results	108
7.2.1	Implementation of Secure Processor on FPGA	108
7.2.2	Experimental Results	110
7.3	Analysis	112
7.3.1	Security Analysis	113
7.3.2	Cost and Performance Analysis	114
7.3.3	Analysis Summary	116
7.4	Conclusions	116
8	High-Performance Montgomery Multiplication for Public-Key Cryptography on Multi-Core Processors	118
8.1	Introduction	118
8.2	Sequential Montgomery Multiplication and Sequential Schemes	122
8.3	Parallel Schemes	125

8.3.1	Multi-core Model	125
8.3.2	Comparing row-based and column-based partitioning	126
8.3.3	parallel Separated Hybrid Scanning (pSHS)	129
8.4	Analysis	133
8.4.1	Analysis on an example	133
8.4.2	Quantitative analysis on general cases	136
8.4.3	Analysis conclusion	141
8.5	Implementing RSA with pSHS	143
8.6	Experimental Results	144
8.6.1	Experimental Setup	144
8.6.2	Experimental Results	145
8.6.3	Discussion	151
8.7	Conclusion	152
9	Conclusions	153
	Publications	156
	Bibliography	159

List of Figures

1.1	Different parallel programming schemes	7
2.1	Concept of SCA	14
2.2	Setup for SCA	16
2.3	An example of SCA results	20
2.4	Comparison between CMOS standard NAND gate and DRP NAND gate	23
2.5	A typical example of a processor with instruction set extensions	26
2.6	Diagram of ARM11 MPCore	28
2.7	Diagram of IBM CELL processor	28
3.1	Algorithmic Masking on AES SBox	31
3.2	Design under Test	35
3.3	Joint probability of power and mask	37
3.4	Total and partial probability mass function of the mask	38
3.5	Resulting mask bias for the selected range of power values	39
3.6	DPA correlation graph on signal with mask bias	40
4.1	Model for inter-wire capacitance	46
4.2	Model for IR Drop	47
4.3	Masked Galois Field multiplier	49
4.4	Simulation results for glitches	50
4.5	Inter-wire capacitance between outputs and inside $GF(2^2)$ multipliers . .	51
4.6	Simulation results for inter-wire capacitance	52

4.7	Power grid resistance	53
4.8	Simulation results for IR Drop	53
5.1	Processor architecture	59
5.2	A dual-core architecture to implement a Virtual Secure Circuit	60
5.3	An example of Virtual Secure Circuit	62
5.4	Mapping a DRP path to processor activity	64
5.5	Mapping from software dataflow to secure circuit	65
5.6	The synchronization scheme with an example	67
5.7	Representative conversion examples	69
5.8	CPA result on unprotected AES on the dual-core prototype	73
5.9	CPA result on protected VSC-AES on the dual-core prototype	74
5.10	CEA results on unprotected AES and dual-core VSC AES	77
6.1	Concept of balanced processor and VSC programming	83
6.2	Examples of balanced instructions	86
6.3	VSC program conversion and corresponding equivalent DRP circuit	89
6.4	Comparing regular, bitsliced, and VSC programming	91
6.5	Experimental setup for single-core VSC	95
6.6	CPA results on unprotected AES and single-core VSC AES	96
6.7	CEA results on unprotected AES and single-core VSC AES	97
7.1	Secure processor micro-architecture	104
7.2	A basic slice-based secure pipeline	105
7.3	Secure processor implementation on FPGA	109
7.4	Cost and performance (cycle cost) of different countermeasures	114
8.1	Analysis model and data flow of Montgomery multiplication	123
8.2	Row-based and column-based task partitioning	127
8.3	An example of pSHS ($s = 6, p = 3, q = 2$)	130

8.4	An example of pSHS in time domain with different TTUs	135
8.5	The smallest allowance time of m and c	137
8.6	Available time and request time of data communications	139
8.7	A 4-core architecture	145
8.8	pSHS's execution time against q	146
8.9	pSHS's execution time against different word lengths	148
8.10	pSHS's execution time based on analysis	151

List of Tables

4.1	State changes and corresponding currents	47
5.1	CPA attack results summary	76
5.2	CEA attack results summary	78
6.1	Attack results summary for single-core VSC	99
7.1	Attack results summary	111
8.1	Meanings of Symbols	131
8.2	Meanings of Terms	136
8.3	Execution Time of SOS	147
8.4	pSHS's Execution Time (et, cycles) & Speedup (sp)	147
8.5	Execution Time of sequential modular exponentiation based on SOS . . .	150
8.6	Execution Time (et, 10000*cycles) & Speedup (sp) of parallel modular exponentiation based on pSHS	150

Chapter 1

Introduction

Many embedded electronic systems, including RFIDs, wireless sensors, smart cards, wireless car keys, smart phones, tablets, and so on, are used to represent personal identification, to store private information, to monitor realtime status, and to do confidential communications. They need to access, store, and manipulate sensitive and private information. As a result, information security on embedded systems has become not just an option but a necessity. When sending emails via mobile devices, we want them to be confidential. When unlocking our cars with wireless keys, we have to guarantee no one else can receive the wireless signal and replay it to open our cars illegally. When downloading applications, we have to make sure there is no virus or Trojans. All of the above show that embedded security has become an important aspect of embedded systems.

To guarantee security, cryptography is an indispensable component. Cryptographic implementation has been under discussion for many years, especially for general-purpose computing systems. However, implementing cryptography on embedded systems still needs a lot of research effort due to its new features, such as limited resources, low budgets, and openness to physical attacks. As cryptography is needed in more and more embedded systems, designers face an urgent challenge: how to implement embedded cryptography while taking into account security, performance, and cost.

In this dissertation, we present our contributions on analyzing and implementing embedded cryptography. We propose solutions to efficient implementation of secure and high-performance embedded cryptographic software. The most important feature of these solutions is the proper usage of two state-of-the-art techniques of embedded processors: instruction set extensions and multi-core architectures. We show that, with proper processor microarchitecture design and suitable software programming, we are able to overcome different design challenges and to implement secure, high-performance, and efficient cryptography on embedded systems.

1.1 Security on Embedded Systems

Security resides in different aspects of embedded systems. Take a cell phone for example [1]. The product developers want to prevent their designs from being copied and reused by other competitors. Communication service providers want to guarantee that only authorized users can access the network. Content providers concern about illegal usage and copy of their digital contents. An end-point user cares about his/her private data stored in the cell phone. When he/she uses cell phone for bank transactions, fraudulent transactions must be forbidden. Virus and Trojans against smart phones are also worries for end users. All of the above cases show us that security almost covers every aspect of such an embedded system.

Most of the security issues are not unique for smart phones. In general, we can summarize security problems on embedded systems in five elements: confidentiality, authenticity, integrity, availability, and non-repudiation [2]. Confidentiality assures that information is only readable by authorized persons or organizations. Authenticity means that data or information are genuine and that different parties involved are who they claim they are. Integrity means that data is not modified unnoticeably. Availability ensures that services offered by embedded systems are available to legitimate users, without being disrupted by attacks. Non-repudiation implies that one party cannot

deny his/her transaction behavior.

Cryptography is a cornerstone to protect the above security elements. There are a number of cryptographic algorithms designed to encrypt and decrypt data, to do digital signature and verification, and to check integrity of data. These algorithms can be grouped into three different categories, including symmetric-key algorithms, asymmetric-key algorithms, and hashing algorithms [3, 4]. Symmetric-key algorithms are a class of algorithms that use identical cryptographic keys for both encryption and decryption. Symmetric-key cryptography is usually used to provide confidentiality, that is to make information unreadable to persons without the cryptographic key. Typical symmetric-key algorithms include Digital Encryption Standard (DES) [5], Advanced Encryption Standard (AES) [6], and RC4. Asymmetric-key algorithms use a pair of keys to do encryption and decryption, respectively. Knowing one key of a pair does not reveal any information of the the other key. Asymmetric-key cryptography is used to support authenticity. RSA [7], DSA [8], and ECC [9] are three popular asymmetric-key algorithms. From the computational point of view, most asymmetric-key algorithms are very expensive when compared with symmetric-key algorithms. Hashing algorithms are one-way mathematical functions that convert a large, possibly variable-sized message into a small digest with fixed length. It is very difficult to find two different messages with the same digest. It is even more difficult to generate a message for a specific digest. Hashing algorithms are often used to protect integrity. Some representative examples are MD5 [10], SHA-1 [11], and the upcoming SHA-3 [12].

An important point to note is that cryptographic algorithms are only a set of mathematical functions. To guarantee security, embedded system designers first need to properly utilize them in a secure protocol. Second, designers also have to implement them securely. Most cryptographic algorithms are secure assuming that attackers have no access to their internal states. When it comes to implementation, designers must make sure that this assumption is valid. This refers to secure implementation of cryptography.

1.2 Secure Embedded Cryptography Design Challenges

Cryptography on embedded systems is different from cryptography on general-purpose computers. One design challenge is how to implement cryptography with constrained resources. Generally speaking, embedded systems are smaller than general-purpose computers. They are designed for only one or a few dedicated functions. Correspondingly, embedded systems have lower resource budget. For example, embedded processors usually have narrower data path, fewer pipeline stages and fewer advanced features when compared with general-purpose processors. This means that there are less data processed by each instruction, fewer instructions that run in parallel and more control costs and memory access costs. Moreover, the clock frequency of embedded processors is often much lower. These inevitably decrease the performance of embedded systems. As the requirements for ubiquitous computing become higher, we find that sometimes conventional embedded processors are not enough to achieve the design target.

Another design challenge to embedded cryptography is the openness to physical attacks. A number of embedded devices are portable or are deployed in the field. They can be lost or stolen. Attackers could easily get physical access to these devices and launch attacks as they want. This makes embedded cryptography more vulnerable. Physical attacks can be roughly grouped into two categories: invasive attacks and non-invasive attacks [13]. Invasive attacks involve direct electrical access to the internal of a embedded device. For example, attackers can open the package of embedded devices or IC chips and place probes to monitor the internal signals. Non-invasive attacks do not break into the devices. Instead, they break cryptography by using the physical characteristics of the device which can be observed without breaking into the device. For example, power attacks [14] uncover the embedded secret keys by only observing the inputs/outputs and the power consumption of the device. Beside power, other physical characteristics include electromagnetic radiation [15], timing [16, 17], and so on. A

non-invasive attacks is also called Side-Channel Attack (SCA). Compared with invasive attacks, SCA usually has lower attack costs and lower possibility of being detected. Therefore, it has presented a great concern to embedded cryptography designers.

Although embedded systems also face logical security problems, e.g. trusted computing [18], we focus on SCA when we talk about security in this dissertation. This does not mean that logical security is not important. We choose physical security as our research topic because of our expertise in circuit design and computer architecture. We expect logical security of embedded systems to be handled by others, e.g. people with experiences in security on general-purpose computers. Between invasive physical attacks and SCA, we select the latter, since SCA usually has lower attack costs, which makes it a threat to a large amount of embedded systems.

1.3 Related Work - A Brief Review

1.3.1 Solutions to Higher Performance

Since embedded systems have very specific applications, one solution to the low-performance problem is to design hardware coprocessors to accelerate desired calculations. For cryptographic applications, we see hardware coprocessors almost for every cryptographic algorithm [19, 20, 21, 22, 23, 24]. Integrating coprocessors is always accompanied by additional hardware cost. Since coprocessors are algorithm-specific, it is not easy to reuse them for other purposes. Therefore, using cryptographic coprocessors is feasible for embedded systems that are specifically designed for security, such as RFIDs, Car keys, and cryptography accelerators [25].

For other embedded systems, where cryptography is needed but not as the main application, coprocessors are considered to be too expensive. In such a case, instruction set extension (ISE) is a possible way of solving the problem. ISE implements smaller operations, usually the most time-consuming part of an algorithm, as special instructions and uses dedicated hardware for acceleration. Unlike coprocessors, ISE does not implement

the whole algorithm. Its hardware cost is lower. Moreover, ISE has higher reusability over coprocessors. This is because it is possible to find similar low-level operations in different algorithms. For example, ISE that accelerates binary polynomial operations for ECC can also be used to accelerate AES's MixColumns and InvMixColumns operations [26]. Due to the above good features, extensible embedded processors [27, 28] are becoming popular. ISEs for cryptography are also under hot discussion [29, 30, 31, 32, 33].

Recently, as transistors continue to shrink in size, designers obtain more hardware resources than ever before. Correspondingly, designing multi-core embedded processors becomes feasible. Since multi-core solutions are still pure software, they are not as efficient as the hardware-based ISEs and coprocessors. However, their advantage is high flexibility. Multi-core processors are suitable for general-purpose embedded systems such as PDAs, tablets, and smart phones. Recent tablets, such as Apple iPad 2, Motorola Xoom, and RIM PlayBook, are already using dual-core embedded processors. For cryptographic engineers, the remaining problem is how to make use of the multi-core processors to accelerate cryptography. In other words, how to do parallel programming for cryptographic algorithms. The easiest way is to run multiple cryptographic tasks on multiple cores so that the throughput increases linearly with the number of cores, as shown in Figure 1.1a. This approach is widely adopted due to its design simplicity. One possible problem is that embedded systems are low-end devices. In most cases, there is no need to perform multiple cryptographic tasks in parallel. In such cases, multi-core processors can not improve the performance. This requires a second solution that partitions one cryptographic task for multiple cores, shown in Figure 1.1b. Considering cryptographic algorithms are usually very compact, simply dividing one algorithm into several parts triggers a bunch of communications due to data dependency. Many researchers prefer to modify the algorithms so that data dependency among different parts is minimized. For example, to accelerate RSA, one of the most time-consuming cryptographic algorithms, we see bipartite [34, 35] and tripartite [36] partitioning of RSA's cornerstone: modular multiplication. These solutions cannot be easily scaled to

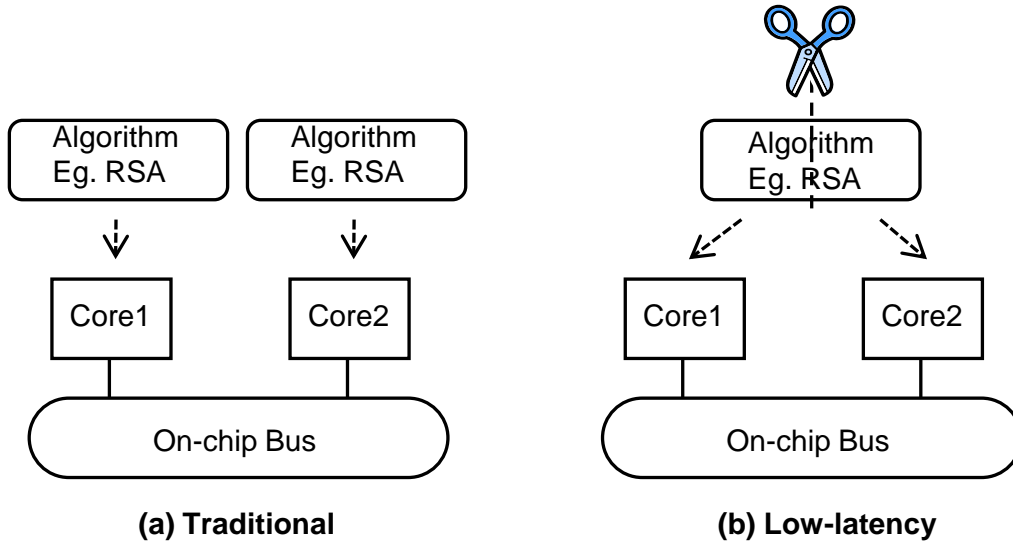


Figure 1.1: Different parallel programming schemes. (a) n tasks mapped to n cores; (b) 1 task mapped to n cores.

multiple cores, since each part still runs on a single processor core. We also see solution that converts the calculation to the residual number system [37] so that long-wordlength operations of RSA can be converted to operations on smaller words, which then leads to a set of parallel operations. However, the resulting solution is not fully balanced over the complete set of processor cores.

1.3.2 Solutions to Stronger Security

For different acceleration solutions, we need corresponding protection schemes. Hardware coprocessors often use secure logic styles, including WDDL [38], MDPL [39], RSL [40], DRSL [41], DWDDL [42] and so on, which convert the standard circuits to special ones. These special circuits have reduced side-channel leakage. The cost of secure logic styles is pretty high. For example, WDDL increases the circuit area by 3.4 times; MDPL's area overhead is 4.54 times. Although some other logic styles like SABL [43], SecLib [44], DyCML [45] and DDSLL [46] have lower area overhead (1 to 2 times), they are based on customized cell libraries. The design cost is very high. Until now, copro-

processors based secure logic styles have not been broadly accepted by industry due to their obvious disadvantages.

Instruction set extensions also use secure logic styles for protection purpose. The difference between protecting coprocessor and protecting ISEs is that ISEs in secure logic styles are usually not enough from a systematic point of view. This is because sensitive data also exists in other parts of the processor, such as register file and memory. Therefore, besides protecting ISEs with secure logic styles, some researchers propose to employ other approaches to protect the other parts in the system [47, 48]. A problem with these solutions is that protecting the entire processor, no matter which approach is used, is a very complicated procedure. In the end, these solutions still turn out to be expensive, although a little more efficient than coprocessors in secure logic styles.

Protecting pure software cryptography on regular processors often turns to modifications on the algorithm so that data manipulation generates data-independent physical information. A popular way of implementing this concept is to randomize the internal variables while keeping the overall functions of cryptographic algorithms unchanged [49, 50, 51, 52]. This approach is called masking. To achieve satisfying security, the randomization could be very complex [53, 54, 55]. A drawback of these protection schemes is their low performance. A recent work shows that a third-order masking scheme decreases the performance by more than 200 times [55]. In our understanding, this is the major barrier that prevents these solutions from practical applications.

1.4 Our Contributions

According to Section 1.3, we see that most current SCA-resistant implementations are either too expensive or too slow. We also notice that current cryptographic algorithms are not well mapped to multi-core processors in scalable and balanced ways. The objective of our study is to find solutions to the above two problems. Specifically, our goal is to deliver SCA-resistant and high-performance cryptographic software

on embedded systems. As software is the most popular implementation of embedded cryptography, we hope that our solutions can benefit a large range of embedded systems.

To achieve the above objective, a suitable processor micro-architecture is very important. SCA on embedded software is a problem that involves both software and hardware. On the one hand, software is the approach to realizing the cryptographic functions; on the other hand, hardware, the processor, is the source that leaks side-channel information. Similarly, performance issue on embedded systems is also closely related to both software and processor. We believe that focusing on microarchitecture is a good starting point to solve the security and performance problems, since microarchitecture is the bridge that connects software and hardware circuits. It is easy to make the best use of both software and hardware and finally hit the sweet spot in the design space. In detail, we base our security and performance solutions on two microarchitectures: instruction set extensions and multi-core processors. For security, we adjust the microarchitectures to support SCA-resistant software designs. For performance, we focus on multi-core microarchitecture and choose RSA, one of the most time-consuming cryptographic algorithms, as our research target. We investigate RSA's cornerstone, Montgomery multiplication, and parallelize it in a scalable and balanced way. As a result, our security solution performs well across the board in terms of security, performance, and cost, without obvious disadvantages in any aspect. Our performance solution is scalable and offers almost linear speedup to the number of cores with high tolerance to the latency of inter-core communication.

We summarize our contributions as follows.

1. As a preliminary research, we investigate the existing SCA countermeasures from both algorithm and circuit levels. We show that masking, one of the popular countermeasures, is breakable at both levels. For a masked design, dependency between mask and power consumption makes it possible to introduce bias to the mask which finally leads to broken secure embedded systems. At the circuit level, we show that higher-order circuit effects, such as glitches, inter-wire capacitance,

and IR drop, are possible leakage sources for basic SCA. The preliminary research shows that masking is not perfect. It is expensive to achieve strong security.

2. We propose the concept of Virtual Secure Circuit (VSC) as a guideline to protect embedded software. VSC is based on the idea of differential logic, also called Dual-Rail Pre-charge (DRP) logic [38]. It makes low-cost modifications on the microarchitecture of processors so that DRP behavior which can be controlled by customized instructions or customized combination of instructions is supported. In such a way, software gets protection when every sensitive data is processed by those customized instructions.
3. Based on the VSC concept, we propose different implementation schemes using dual-core microarchitecture and instruction set extensions. We show our research steps and the results obtained at each step. Finally, we obtained secure solutions to embedded software on both multi-core platforms and instruction set extensions. Compared with other security solutions, our security solutions perform well across the board in terms of security, performance, and cost, without obvious disadvantages in any aspect.
4. We also propose a parallelization scheme, called pSHS, for Montgomery multiplication on multi-core processors. Without modifying the algorithmic algorithm, pSHS is able to accelerate Montgomery multiplication linearly to the number of cores with a high tolerance to the inter-core communication latency. Moreover, we give a comprehensive analysis on pSHS which is able to guide the programmers to choose the optimal number of cores that should be used by pSHS according to the inter-core communication delay. pSHS accelerates one single RSA program very efficiently. It not only increases the throughput but also decreases the latency. This makes it suitable for embedded public-key cryptography on embedded multi-core systems.

1.5 Organization

Most of our work has been published in conference proceedings and journals. In this dissertation, we try our best to present it in a systematic way. In addition, some details and results that are not presented in the previous publications are elaborated in this dissertation. The rest of this dissertation uses seven chapters to present our contributions.

In detail, this dissertation is organized in the following way: Chapter 2 introduces preliminary knowledge that is needed by this dissertation, such as Side-Channel Attacks, masking countermeasure, hiding countermeasure, multi-core processor microarchitecture, and instruction set extensions.

Chapter 3 presents our first preliminary research that analyzes the masking countermeasure at the algorithmic level [56]. We present how we manually introduce bias to algorithmic masking and how we successfully break a masking scheme for AES.

Chapter 4 shows our second preliminary research that investigates how higher-order circuit effects, such as glitches, inter-wire capacitance, and IR drop, influence the security of a masked circuit [57]. Both analysis and experimental results show that all those three higher-order circuit effects are possible side-channel leakage sources. To reduce the leakage, extraordinary efforts are required, which tells us that strong security achieved by masked circuit requires extremely high costs.

Chapter 5 proposes the concept for VSC. Following that, VSC on dual-core processors is presented as the first implementation scheme [58]. We show that, without modification to the processor cores, we are able to implement the VSC concept, achieving strong security with low hardware cost. This scheme works well in front of power attacks but requires additional countermeasures to disable electromagnetic attacks.

Chapter 6 gives another implementation scheme of VSC based on instruction set extensions [59]. This chapter shows how to implement VSC on a single-core processor. Compared with the dual-core implementation scheme, VSC on single-core processor has consistent security performance in front of both power attacks and electromagnetic

attacks.

Chapter 7 improves the implementation scheme in Chapter 6. Instead of implementing the secure instructions in the regular processor pipeline, we introduce to the processor a secure pipeline specifically for all the secure instructions. This secure pipeline allows us to do identical placement-and-route for direct and complementary parts, which is an essential requirement to obtain strong security with DRP countermeasure. In addition, we compare this solution with other representative solutions and show that only our solution hits the sweet spot in the design space.

Chapter 8 shifts our focus from SCA-resistance to high-performance cryptography. We present a scalable and balanced way, called pSHS, to partition the Montgomery multiplication which achieves a linear speedup on a message-passing multi-core processor [60, 61]. Comprehensive analysis shows that pSHS has a high tolerance to the inter-core communication latency. pSHS accelerates each single Montgomery multiplication and finally accelerates each single public-key cryptography that makes use of Montgomery multiplications. This makes it suitable for an embedded system where usually at most only one public-key requirement exists at one time.

Chapter 9 summarizes our research and analyzes some limitations which are useful for future research.

Chapter 2

Preliminaries

In this Chapter, we introduce preliminary knowledge of SCA and state-of-the-art embedded architectures. The rest of the dissertation will continuously refer back to this chapter for basic concepts.

2.1 Side-Channel Attacks and Countermeasures

In this section, we briefly introduce the basic concept of SCA, how to mount SCA in practice, and how to thwart SCA.

2.1.1 SCA Concept

Cryptographic algorithms are designed to resist at least thousands of years of cryptanalysis. That is, given that the attackers know the algorithm, the input data and the output data, any known method to extract the crypto-algorithm's secret key has an enormous computational complexity. Take brute-force attack as an extreme case, if an attacker is able to check 1 billion AES keys per second, the 128-bit AES algorithm [6] can resist a brute force attack for 10^{13} years.

Unfortunately, the security features of an algorithm alone are not sufficient to guarantee that their implementations are also secure. SCA is able to break a cryptographic

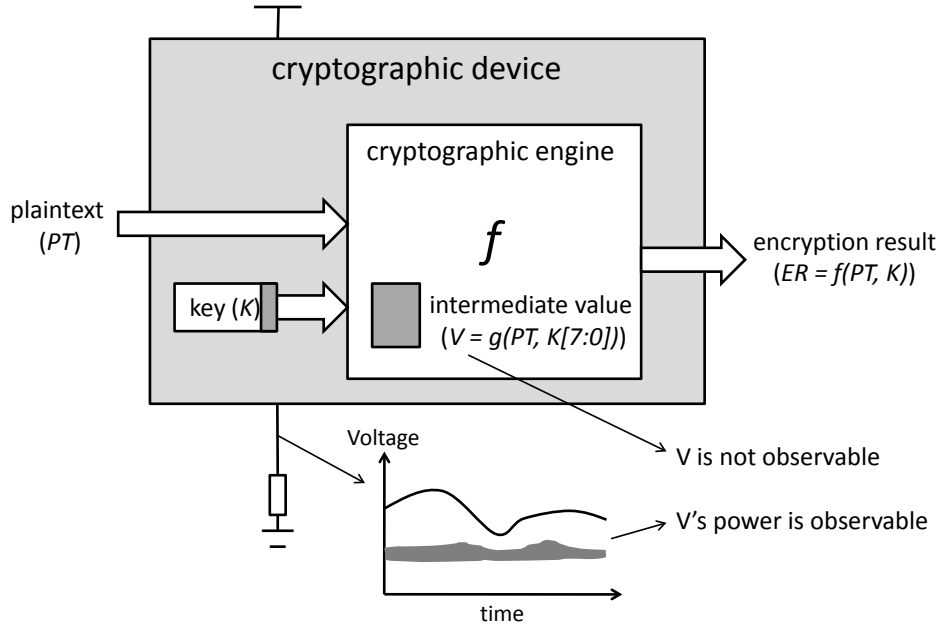


Figure 2.1: Concept of SCA.

device much faster than cryptanalysis. The reason is that, by exploiting the side-channel leakage of a device, SCA can reveal the secret key bit by bit or piece by piece. This breaks the exponential complexity of a brute force key search. In this section, we describe the concept of SCA.

Figure 2.1 shows a cryptographic device and how power attacks break it. The device implements a cryptographic algorithm, represented by f . f takes the plaintext (PT) and the key (K) as inputs, and generates the encryption result (ER) ($ER = f(PT, K)$). The internal secret key K is not directly observable through the ports of the device. The objective of SCA is to reveal the value of K .

Each bit of ER is related to every bit of PT and every bit of K . Suppose K has 128 bits, then a traditional brute-force attack needs to consider 2^{128} possible key values. It is this huge search space that ensures the security of the cryptographic algorithm. However, SCA does not try to break the entire K at once. Instead, SCA divides K into pieces, which are broken one by one. For example, a typical AES algorithm uses an 128-bit key. Using an SCA that reveals one key byte at a time, the search space for the

entire key is reduced from 2^{128} to $16 * 2^8 = 2^{12}$.

Obviously, this is a drastic reduction, and SCA achieves this as follows. Although every bit of the output ER is guaranteed to be related to every bit of K , this is not true for the intermediate values. We can always find intermediate values that are only related to a small part of K , e.g. one byte of K . For example, assume that we can find an intermediate value V which depends on a single key byte $K[7 : 0]$ and the plaintext PT . We can write $V = g(PT, K[7 : 0])$. Then $K[7 : 0]$ can be discovered with only 2^8 guesses.

To test which guess is correct, we therefore need to observe V . This variable is inside of the implementation, but it is indirectly observable through its power dissipation. Indeed, the power dissipated by V is a part of the power dissipated by the entire device, which can be measured by the attackers. With proper distinguishing techniques, the chip's overall power dissipation can be used in place of the power dissipation from V ; power dissipated by unrelated components can be treated as noise. In such a way, SCA can identify the entire secret key piece by piece and finally breaks the cryptographic implementation within a very short time.

There are several different distinguishing techniques, including difference of means, Pearson correlation, and mutual information. Correspondingly, we have different kinds of attacks: differential attack [14], correlation attack [62], and mutual information analysis [63]. In the rest of this dissertation, we mainly use correlation attacks.

2.1.2 An SCA Example

To make the above SCA concept more concrete, we present an SCA example in this section to show how to do measurement and analysis. We run an AES program on top of an embedded processor. The AES key is embedded in the on-chip memory. Such a cryptographic device receives plaintext and sends out ciphertext through its UART peripheral. To uncover the key with SCA, we build a setup whose block diagram and real picture are shown in Figure 2.2.

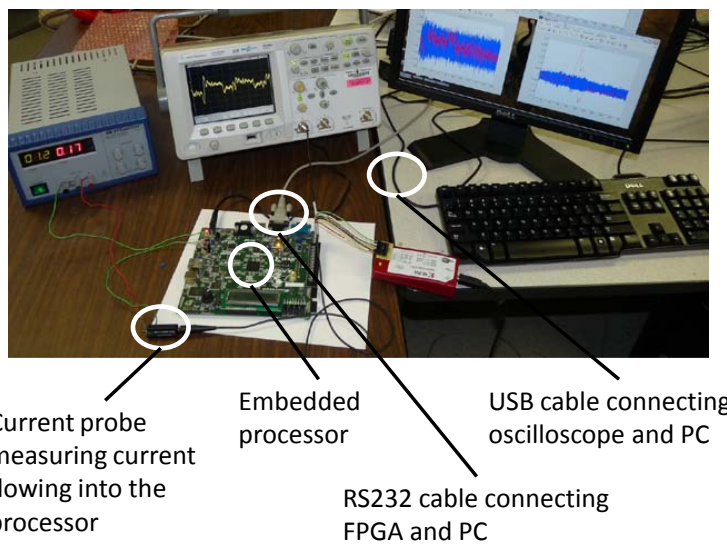
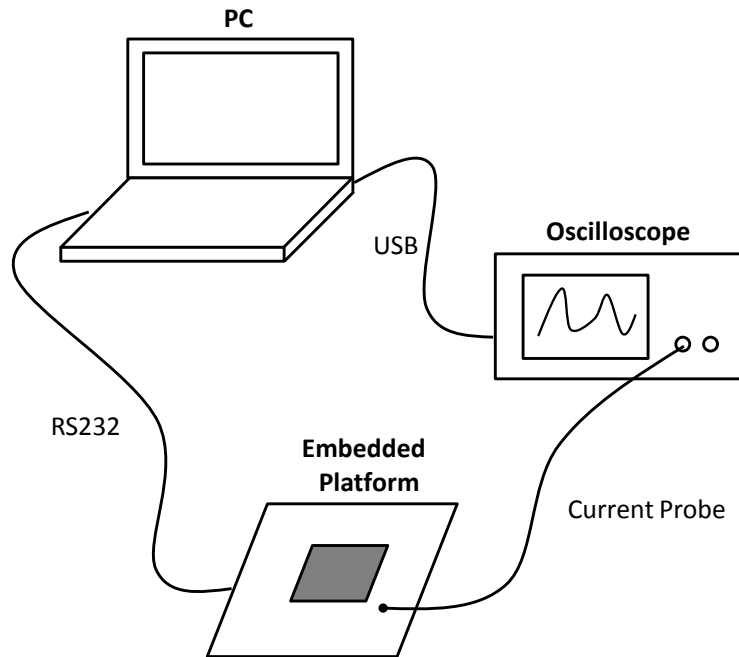


Figure 2.2: Setup for SCA. Photo was taken by the author in 2010.

The setup contains the cryptographic device, an oscilloscope (Agilent DSO5032A) and a PC. The three parts of the setup are connected in a circular fashion. A RS232 cable connects the cryptographic device and the PC. Between the oscilloscope and the

Algorithm 1 Program running on the embedded processor

```
exp_key  $\leftarrow$  KeyExpansion(key)  
loop  
  if UART is not empty then  
    wait until 16 bytes received  
    pt[15 : 0]  $\leftarrow$  UART_FIFO  
    ct[15 : 0]  $\leftarrow$  Cipher(pt, exp_key)
```

PC is a USB cable, through which the PC is able to send commands to and get sampling waveform from the oscilloscope. The oscilloscope uses a current probe (Tektronix CT-2) to monitor the current flowing into the cryptographic device. We use the current to represent the power consumption of the embedded system. Side-channel analysis requires a number of measurements with different inputs (plaintexts for AES). In this example, the result of one measurement is the current trace of the cryptographic device and the corresponding random plaintext block for encryption. Each measurement consists of the following 4 steps. A side-channel analysis that requires n measurements needs to repeat these 4 steps for n times.

- Step 1: The PC sends a random plaintext block (16 bytes) to the embedded platform through the RS232 cable.
- Step 2: The embedded processor in the platform receives the plaintext and encrypts it with the AES software. The program running on the embedded processor is shown in Algorithm 1. *Cipher* is an AES encryption function.
- Step 3: After sending out one block of plaintext, the PC sends command to the oscilloscope to sample the current trace when PowerPC is running the encryption.
- Step 4: After sampling is done, one current trace is sent back to PC for side-channel analysis.

After obtaining measurements, we move on to the analysis phase. First of all, we select a set of intermediate values (V in Figure 2.1) from the AES algorithm, each of

which is related to one byte of the cryptographic key of AES. Here the outputs of the **Substitution** step in the first round are the selected intermediate values, as shown in Equation 2.1, where $in[i]$ is the i -th byte of the 128-bit plaintext, $key[i]$ is the i -th byte of the 128-bit key, and $v[i]$ is the i -th byte of the 128-bit AES intermediate state.

$$v[i] = \text{subbytes}(in[i] \oplus key[i]); \quad (2.1)$$

In this formula, the plaintext in is known to attackers, while key and v are unknown. However, v is indirectly observable through the power consumption of the algorithm. Hence, we can create a hypothesis test for a key byte, using a power model for v . By making a guess for the i -th key byte, we can infer the value of $v[i]$. The power model used in the attack is an estimate for the power consumption of the hypothesized $v[i]$. The actual hypothesis test, explained later in this section, will compare the estimated power consumption with the measured power consumption to identify the most likely key byte hypothesis.

To map $v[i]$ to power hypothesis, two common power models are Hamming weight and Hamming distance. Usually, Hamming weight represents static power and Hamming distance represents dynamic (switching) power. Hamming distance requires two sequential values in one register, which is hard for attackers since they do not know which other variables share the same register with v . Therefore we use Hamming weight.

The last step is to distinguish the correct key hypothesis from other incorrect ones based on the measurements and the power model. Suppose we have n random plaintext blocks ($pt[1 \dots n]$). Each of them is used for one measurement. Correspondingly, we obtain n power traces, each of which contain m sampling points ($tr[1 \dots n][1 \dots m]$). A correlation attack takes pt and tr as inputs, and discovers AES's key byte by byte by focusing on the first round of AES. The details are presented in Algorithm 2. The basic process is to take $pt[1 \dots n]$ and a guess value of a key byte (key_guess) to calculate an intermediate value $v[1 \dots n]$ ($v[i] = f(key_guess, pt[i])$). Sampling a complete power trace guarantees that the operations on the intermediate value occur during the sampling

Algorithm 2 Correlation power attack on one key byte.

Require: $pt[1 \dots n]$ contains the random plaintext for encryption; $tr[1 \dots n][1 \dots m]$ contains the sampled power traces; f maps the inputs to the intermediate value v ; g calculates the correlation coefficient.

Ensure: $key_gap[j]$ is the CPA-attacked key byte.

/ Obtain correlation coefficient traces $corr$ */*

for $key_guess = 0$ to 255 **do**

for $i = 1$ to n **do**

$v[i] = f(key_guess, pt[i])$

$v_hw[i] = HammingWeight(v[i])$

for $i = 1$ to m **do**

$corr[key_guess][i] = g(v_hw[1 \dots n], tr[i][1 \dots n])$

/ Find the correct key byte */*

for $i = 1$ to m **do**

 find $|corr[key1][i]| = \max(|corr[0 \dots 255][i]|)$

 find $|corr[key2][i]| = \text{second_max}(|corr[0 \dots 255][i]|)$

$gap[i] = corr[key1][i] - corr[key2][i]$

$key_gap[i] = key1$

find $gap[j] = \max(gap[1 \dots m])$

return $key_gap[j]$ as the correct key byte

process. We use the Hamming weight of v ($v_hw[1 \dots n]$) to approximate the power dissipated by v . Then we calculate the correlation coefficient of v_hw and the actual measured power traces at each sampling point. As a result, we obtain a correlation coefficient trace $corr[1 \dots m]$ ($corr[i]$ is the correlation coefficient of $v_hw[1 \dots n]$ and $tr[1 \dots n][i]$). Details of correlation calculation are shown in Equation 2.2. $\overline{v_hw}$ and \overline{tr}_i are the mean values of vectors $v_hw[1 \dots n]$ and $tr[1 \dots n][i]$, respectively.

$$corr[i] = \frac{\sum_{d=1}^n (v_hw[d] - \overline{v_hw}) \cdot (tr[d][i] - \overline{tr}_i)}{\sqrt{\sum_{d=1}^n (v_hw[d] - \overline{v_hw})^2 \cdot \sum_{d=1}^n (tr[d][i] - \overline{tr}_i)^2}} \quad (2.2)$$

By now, we have obtained one coefficient trace $corr[1 \dots m]$ which is corresponding to one guess value of the key byte. Since there are 256 possible values for one byte of key, we can, therefore, obtain 256 coefficient traces. By grouping these coefficient traces together, we are able to identify one coefficient trace from all the other 255 traces. In particular, at some points (when the operations on the intermediate value occur),

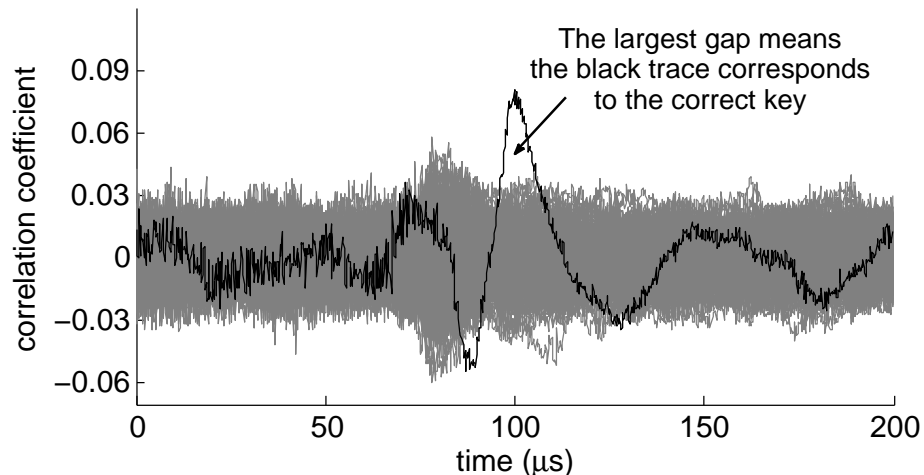


Figure 2.3: An example of SCA results. 256 correlation coefficient traces are used. Around time 100 μ s, the black trace which corresponds to the correct key byte emerges from all the other 255 traces. It turns out that this black trace is corresponding to the correct key hypothesis. So the attack is successful.

the abstract value of the coefficient trace corresponding to the correct key guess is much larger than all the other traces. Figure 2.3 gives an example of the correlation coefficient traces. We can see that around time 100 μ s, one trace emerged from all the other traces. And it turned out that the key guess corresponding to this trace is the correct key, which means correlation power attack on this key byte is successful.

2.1.3 SCA Countermeasures

Power attacks and electromagnetic radiation attacks are based on the dependency between power or electromagnetic radiation and immediate values in a cryptographic algorithm. To prevent SCA, we need to eliminate the dependency. In other words, the goal is to obtain data-independent power and electromagnetic radiation.

The earliest ways to thwart SCA were called “Ad-hoc Approaches” [64], such as adding noise, randomizing execution sequence [65, 66], randomly injecting redundant instructions [67] and limiting the dynamic variance of the current. Although they are effective to some extent, the drawback of this kind of countermeasures is that theoret-

ically they do not prevent attacks completely: attacks can still be successful by taking more samples or signal processing [68, 69].

For the purpose of completely preventing SCA, at least in theory, researchers presented two kinds of countermeasures: masking and hiding. In this dissertation, we mainly investigate these countermeasures and base our protection solutions on hiding.

Masking

Masking achieves the above goal by randomizing the intermediate values of the cryptographic algorithms. The basic concept can be described by Equation 2.3.

$$o = f(a) = f_1(a \oplus m) \oplus f_2(m) \quad (2.3)$$

In Equation 2.3, \oplus represents the masking operator. For example, it represents exclusive or for Boolean masking. $f(a)$ represents a cryptographic module with unmasked input a . We create a masked version of this module by introducing a random mask m and by partitioning the original function into two sub-functions f_1 and f_2 which process the masked signal $a \oplus m$ (a_m) and the random mask m respectively. The decomposition of f into f_1 and f_2 depends on the original cryptographic module. This is a design problem in itself, in particular when f contains non-linear terms. Nevertheless the decomposition has been demonstrated to be feasible [50].

One good feature of masking is that it can be implemented at the algorithm level without changing the low-level hardware to make their power characteristics independent of the processed data. There are also some assumptions for masking to achieve expected SCA-resistance. First, m needs to remain secret. A secret m implies that we cannot extract any information about a from observing a_m . For example, choosing a pseudo-random sequence for m is sufficient to obtain a statistically uniform distribution for a_m . This is necessary to guarantee the power consumption of f_1 and f_2 to be data-independent. Otherwise, if it is possible to predict m , then we would still be able to obtain the trace of the unmasked data a [70, 71, 72]. In Chapter 3, we will discuss more

details.

A well recognized threat to masking is called higher-order attacks [73, 74, 75, 76, 77, 69]. For example, a second-order attack exploits two intermediate values that are related to the same mask. For example, by processing these two masked values with a \oplus operation, we can remove the effect of masking. There are also other approaches to exploit second-order attacks [71, 72]. To prevent second-order attacks, designers need to integrate at least two independent masks for each unmasked value, called second-order masking. It is demonstrated that a $n + 1$ -order attack is able to break any n -order masking. However, as the order number increases, the difficulty of attacks increase exponentially. So it is believe that to a certain order, the attack becomes infeasible. Of course, higher-order masking is also more complicated and hence has higher costs and lower performance.

Another constraint for masking is that Equation 2.3 is defined at the logic level. Hardware (circuit) implementation is not always perfect. Circuit level issues may introduce unexpected side-channel information. For example, glitches in circuit have been demonstrated to a side-channel leakage source that is able to lead to successful attacks on masked designs [78, 79]. In addition to that, we will show that some other circuit level issues have similar negative influence on masking countermeasure in Chapter 4.

Hiding

The goal of hiding is to design cryptographic devices such that they have the same power consumption and electromagnetic radiation when manipulating different data. Hiding allocates the task of eliminating data dependency to the device hardware. Unlike masking which makes some statistical characteristics, typically the expectation, of the intermediate values statistically independent of the unmasked value, hiding eliminates any possible data-dependency. Therefore, hiding does not suffer from higher-order attacks.

There are several different approaches to implement hiding. One of them is the

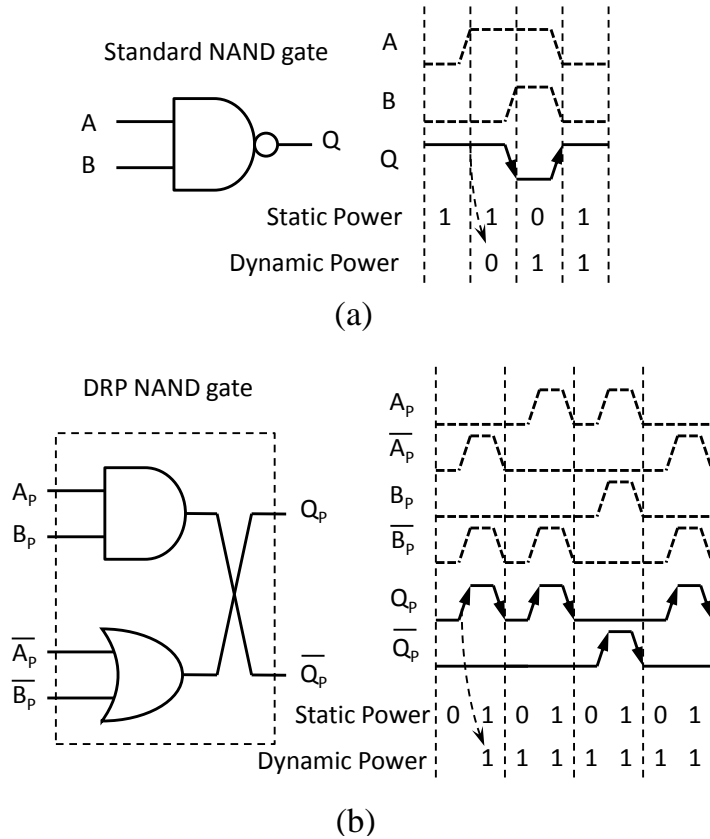


Figure 2.4: Comparison between CMOS standard NAND gate and DRP NAND gate. (a) A CMOS standard NAND has data-dependent power dissipation; (b) A DRP NAND gate has a data-independent power dissipation.

differential logic, also called Dual-Rail Precharge (DRP) Logic. We use an example to explain the details.

Figure 2.4 explains the operation of the DRP technique using a DRP NAND gate. In this example, we approximate static and dynamic power dissipation of a logic gate through the Hamming Weight and Hamming Distance of its output respectively. In the case of a single NAND gate (Figure 2.4a), the static and dynamic power dissipation depend on the input values of the gate. For example, if the static power is 0, both inputs must be 1. This side-channel leakage is the basis for SCA.

Figure 2.4b shows the same test case on a DRP NAND gate. In this case, the circuit encodes each logic value with a complementary pair $(A_p, \overline{A_p})$. Furthermore, each pair is

pre-charged to (0,0) in each clock cycle before evaluation. As a result, each clock cycle, every DRP signal pair shows exactly one transition from 0 to 1 and another one from 1 to 0. The resulting static and dynamic power dissipation are now independent of the input values of the DRP gate.

Despite the elegance of this concept, DRP circuits in hardware do have some disadvantages. First, DRP circuits are at least two times larger than equivalent standard CMOS circuits, and they have a much larger power dissipation. Second, the constant-power argument, based on Hamming Weight or Hamming Distance, does not always hold when low-level electrical effects are taken into account. Small asymmetries between the direct and complementary paths of a signal pair still may lead to residual side-channel leakage. Nevertheless, careful design is able to reduce the imbalance to a very low level. This requires much more power measurements for a successful SCA [80, 81].

So far, the DRP technique has been broadly used in hardware as secure circuits, for example in WDDL [38], and in MDPL [39]. However, software DRP technique has not been developed. The major reason for this is that DRP technique requires the executions of the direct and complementary datapaths in parallel. In regular processors, this cannot be realized.

2.2 State-of-the-Art Embedded Microarchitectures

In this section, we introduce the concept of two state-of-the-art embedded microarchitectures, including instruction set extensions and multi-core processors.

2.2.1 Instruction Set Extensions

Instruction set of a processor maps a set of basic operations to machine codes. The processor activates dedicated hardware to process each instruction. Putting instructions in sequence, we get a software routine which finishes a more complicated operation. For the purpose of generality, processor designers usually pick those most frequently used

basic operations and include them into instruction set.

Instruction set extensions are additional special instructions to the existing instruction set. These special instructions are implemented with dedicated hardware that is able to process some specific operations efficiently. Once extensions are added, the processor is largely targeted for specific applications and named as Application Specific Instruction-set Processor (ASIP). Since most embedded systems are application-specific, instruction set extensions are very useful. There are several commercial products of statically extensible or custom embedded microprocessors such as Tensilica Xtensa [27], MIPS Pro Series [82], and ARC 700 Series [83].

An on-going hot discussion on instruction set extensions is to implement them on reconfigurable devices, for example FPGAs [28, 84]. The idea is to implement the extensions with reconfigurable resources so that they can be updated according to different applications. This, to some extent, makes the architecture more general.

A typical example of a processor with instruction set extension is shown in Figure 2.5. In a five-stage processor, the original data path consists of 5 stages, including instruction fetch, instruction decode, execution, memory access, and write back. To add extensions, the decode stage should be extended so that it recognizes new instructions. After that, the processor is able to transfer operands to the execution stage in which dedicated hardware is added to process the extended operation. Although the concept of instruction set extensions is simple, integrating them into a processor still needs a lot of efforts especially when some extensions need multiple clock cycles to finish their job. In such a case, the processor pipeline needs to be stalled or other mechanisms should be used, for example out-of-order processing [85].

A challenge for instruction set extensions is how to enable compilers to make good use of them. Compilers that are originally designed for the existing instruction set need modifications to cover the new instructions. However, this modification is infeasible. On the one hand, compiler developers cannot foresee what instruction set extensions will be added by users. On the other hand, it is unrealistic to expect users to understand

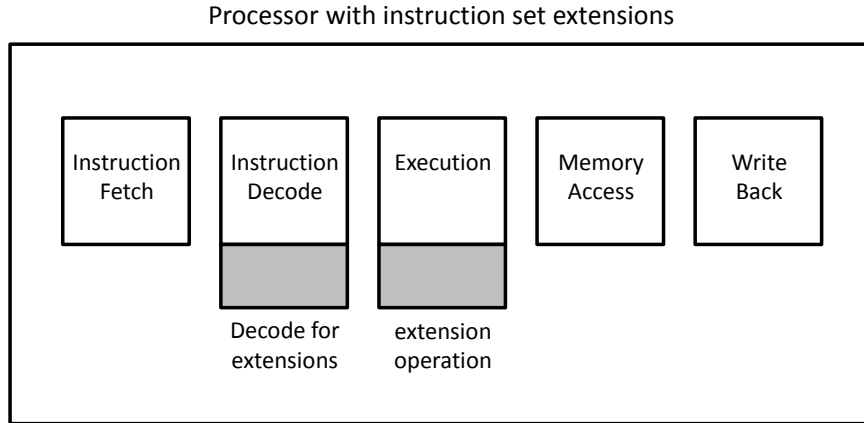


Figure 2.5: A typical example of a processor with instruction set extensions.

compilers and do modifications by themselves. One way to go around this problem is as follows. Users first do regular software programming. Second, they profile the program to determine the best instruction extensions for the performance. For example, they can select the most frequently executed code block and use one instruction to cover the operation. Third, with the information from the second step, processor generators are able to implement the new instruction; compilers know where the new instructions are used. Their job is just to replace the selected blocks of code with new instructions. The TIE compiler [86] for Xtensa processors is an example of this approach.

2.2.2 Multi-core Processors

As processors' power approaching the power wall, we have lost the option of improving performance by increasing the clock frequency. Alternatively, researchers found that multi-core processors were able to deliver the same computational capability with higher power and energy efficiency [87]. As a result, multi-core processors have become popular. And this is supported by the scaling trend of the transistors, which offers chip designers enough hardware resources in one chip.

Designing multi-core processor is not just copy-and-paste. An important task is to handle on-chip communication among different cores. According to the solutions to

this problem, multi-core processors are grouped into two categories: shared-memory processors [88, 89] and message-passing processors [90, 91]. Shared-memory processors implement inter-core communication by sharing memories that can be accessed by different cores. This requires either multi-port memories or shared buses and arbiters. Message-passing processors accomplish inter-core communication by transferring messages from one to another. The inter-core communication is done with a network that connects different cores. Different cores have their own memory space that is not directly accessible by other cores.

Currently, we can find a number of embedded multi-core processors in the market, including Nvidia Tegra 2, TI OMAP, Marvell Armada 628, Apple A5, Aeroflex Gaisler Leon3 processor, IBM CELL processor, and so on. Most of these processors use shared memory architecture. But as the number of cores increases, the shortage on scalability of shared-memory architectures will become significant. We can expect that message-passing architecture will also become popular. Figure 2.6 and Figure 2.7 show the diagrams of the shared-memory-based ARM11 MPcore processor and the message-passing-based CELL processor, respectively. In Figure 2.6, we see that four cores with their own L1 Cache are all connected to the Snoop Control Unit for synchronization of the L1 Cache contents and for sharing L2 Cache and main memory. In comparison, the CELL processor in Figure 2.7 has 8 Synergistic Processing Elements (SPEs). Each SPE has its own local storage. Eight SPEs are connected in a ring fashion by the Element Interconnect Bus (EIB). Communication among cores are done with messages transferred by EIB.

2.3 Conclusion

In this chapter, we explained the concept of SCA, how to mount SCA, and the techniques to prevent SCA. In addition, we also briefly introduced the basic concepts of instruction set extensions and multi-core architectures. In the rest of this dissertation,

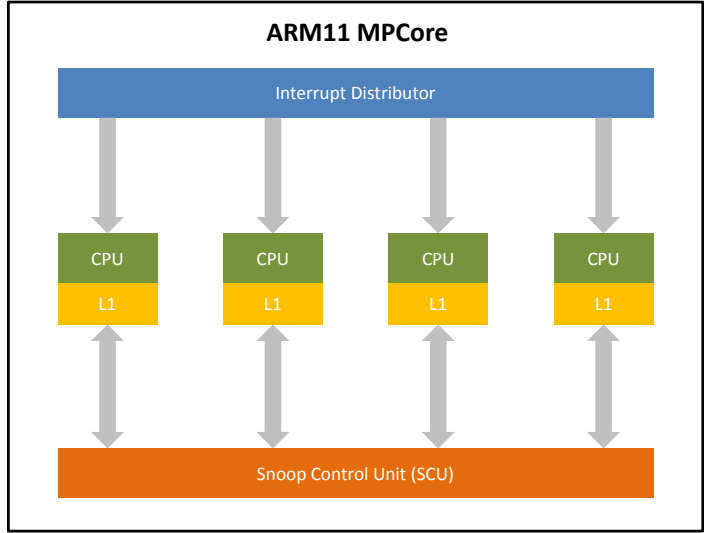


Figure 2.6: Diagram of ARM11 MPCore [89].

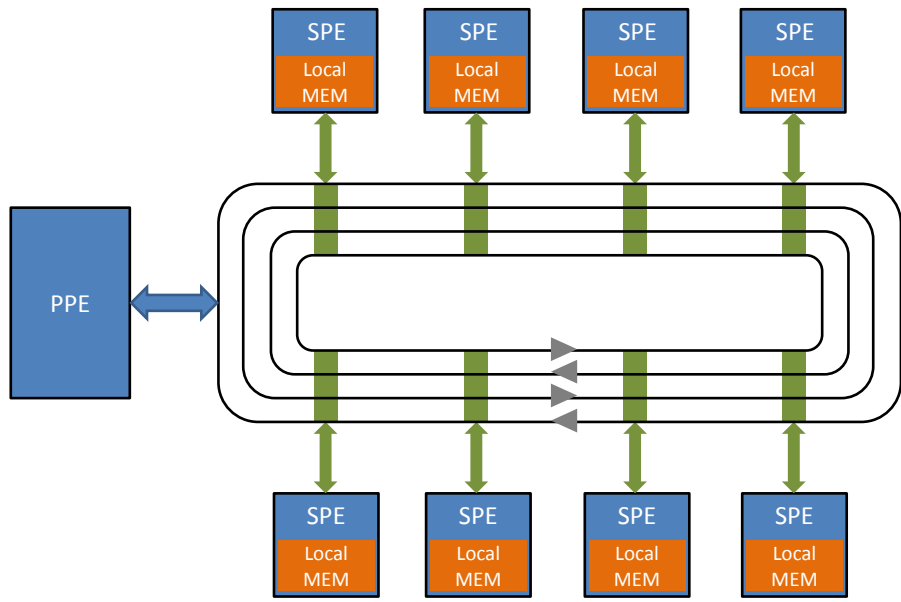


Figure 2.7: Diagram of IBM CELL processor [92].

we will present how we make use of both embedded architectures to thwart SCA and to improve performance.

Chapter 3

Slicing Attack on A ‘Perfectly’ Masked AES

In this chapter, we propose a new SCA method, called slicing attack, that breaks the masking countermeasure. The success of this attack helps us to polish our understanding on masking, especially algorithmic masking. It also serves as a guidance to the proposal of our own side-channel protection solutions.

In general, slicing attack breaks masking by introducing bias to the random mask in the analysis process, so that attackers can make better-than-random guesses on the mask and finally mitigate the protection of masking. Since the bias introduction is done by selecting power measurements that fall into a slice of their probability distribution, we call this attack slicing attack [56].

3.1 A ‘Perfect’ Algorithmic Masking on AES

We have introduced the concept of masking in Section 2.1.1. In this section, we introduce an example of masking on AES SBox [50]. The main component of the S-Box, as illustrated in Figure 3.1, is a masked $GF(2^8)$ inversion. The relationship between inputs and outputs is shown in Equation 3.1.

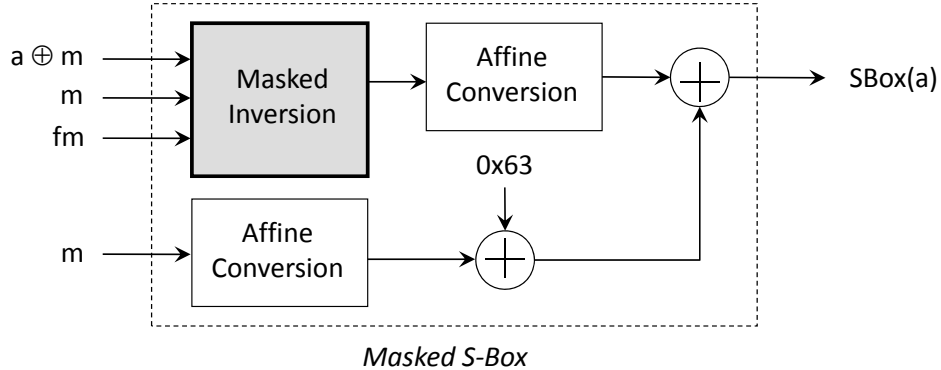


Figure 3.1: Algorithmic Masking on AES SBox.

$$o = f(a) = \text{affine}(\text{minv}(a \oplus m, m, fm)) \oplus \text{affine}(m) \oplus 0x63 \quad (3.1)$$

It takes $a \oplus m$, m and fm as inputs. Input $a \oplus m$ is the masked value of a , m is the mask, and fm is called a fresh-mask, and its purpose is to mask intermediate results within the inversion block. Every input to the masked AES SBox is independent of the unmasked input a , so that the internal states do not leakage side-channel information. The masked inversion is implemented using a combination of modular addition and modular multiplication. Implementation details can be found in [50].

3.2 Slicing Attack

Recently, it was shown that a Boolean masking scheme can be broken by first estimating the mask bit and then mounting a DPA based on these estimates [72, 71]. These attacks were demonstrated on single-rail as well as dual-rail masked logic, and they demonstrated a practical implementation of the so-called zero-offset second-order DPA [75].

In this section, we apply similar idea to algorithmic masking. Let's call p the power dissipation of a masked circuit, which is the power from $f_1(a \oplus m) \oplus f_2(m)$ in Equation 3.1. For a perfect masked circuit, the correlation between p and the unmasked input a must

be zero. Indeed, if there would be any correlation left, then it would mean the masked circuit is still susceptible to DPA. So we can write

$$\text{corr}(p, a) = 0 \tag{3.2}$$

However, masked circuits are designed without considering the effect of the mask m on the power. In general, the power dissipation will still be correlated to the mask m . This is considered harmless because m is a random number, without useful information to the attacker. But if p and m are correlated to some extent, this means that

$$\text{corr}(p, m) \neq 0 \tag{3.3}$$

As we discussed before, the mask m should remain a secret to maintain full side-channel resistance. Observing Equation 3.3, a question which comes to mind is: can we make use of the power p to estimate the mask m ? Indeed, this question has been answered in a positive manner for the specific case of single-bit Boolean masking schemes [71, 72]. However, there is no reason why this should be limited to single-bit masking. We will therefore derive a method that works on multi-bit algorithmic masking.

In the following, we treat the mask value m as well as the power dissipation p as discrete random variables. This assumption is valid if we approximate p for example with the hamming weight of the circuit under consideration. We will further see how this can be generalized to measured power values with a continuous distribution.

We can write the joint probability of the circuit consuming power p with mask value m as

$$\Pr(\text{Power} = p, \text{Mask} = m) = \Pr(p, m) \tag{3.4}$$

Since the mask has a uniform distribution, the marginal probability of m should be a constant. The marginal probability of m is found by summing out the joint probability over p .

$$Pr(m) = \sum_p Pr(p, m) = \text{Const} \quad (3.5)$$

The conditional probability of the power can now be found as follows.

$$Pr(p|m) = \frac{Pr(p, m)}{Pr(m)} \quad (3.6)$$

The conditional probability in Equation 3.6 is the probability that the power dissipation of the circuit is p , given that the mask equals m . Note that we indicated earlier that the power is correlated to the mask. This means that the conditional probability in Equation 3.6 cannot be independent of m . Thus, if we consider a given power measurement p_1 , then the conditional probability of this power measurement will change as the mask changes. We can express this as follows.

$$Pr(p_1|m) = \frac{Pr(p_1, m)}{Pr(m)} \neq \text{Const} \quad (3.7)$$

The inequality of Equation 3.7 becomes an attack method when we express the conditional probability of m . Through Bayes' theorem we can write

$$\frac{Pr(p|m)}{Pr(p)} = \frac{Pr(m|p)}{Pr(m)} \quad (3.8)$$

We can now write the inequality of Equation 3.7 as follows. Assume that we have a given power measurement p_a and two possible (and different) mask values m_1 and m_2 , then according to Equation 3.7 and Equation 3.8:

$$Pr(m_1|p_a) = \frac{Pr(p_a, m_1)}{Pr(p_a)} \neq Pr(m_2|p_a) = \frac{Pr(p_a, m_2)}{Pr(p_a)} \Big|_{m_1 \neq m_2} \quad (3.9)$$

In other words, Equation 3.9 shows that one mask value, say m_1 , is more likely to correspond to a given power measurement p_a than another mask value. Thus if we measure a given power level, then we can estimate with a better-than-random guess on what the mask value would be. In practice, power values are measured as continuous

quantities, and it is impossible to choose just one discrete power value. Therefore, to implement the test of Equation 3.9, we will choose p over a range of possible values, and all those p samples fall into this range build up the slice of samples we want. Accordingly, the inequality becomes a sum:

$$\sum_{p \in range} Pr(m_1|p) \neq \sum_{p \in range} Pr(m_2|p) \Big|_{m_1 \neq m_2} \quad (3.10)$$

In the case of continuous power values (such as obtained through physical measurements), the summations in Equation 3.10 become integrals. Clearly, there is a limit to this inequality. When we would integrate the power over the full range of possible values, Equation 3.10 becomes Equation 3.5 and turns from an inequality into an equality. So the key is to perform a partial selection of p over the possible range of values.

We can now summarize our attack method as follows.

Step 1: Collect power traces from the masked cryptographic circuit, and establish the possible range of power values.

Step 2: Select a slice of the possible range of power values, and discard all measurements which fall outside this range.

Step 3: Perform a DPA on the set of measurements obtained through step 2.

In the next section, we will demonstrate this surprisingly simple technique by means of a simulated attack on a masked SBOX.

3.3 Experimental results

In this section, we present experimental results based on logic-level simulation. The main idea of the experiment is to count the number of logic-1s in a circuit to estimate the power dissipation of a circuit. We call this a hamming-weight simulation. Our purpose is to show that algorithmically masked hardware circuits that successfully hide

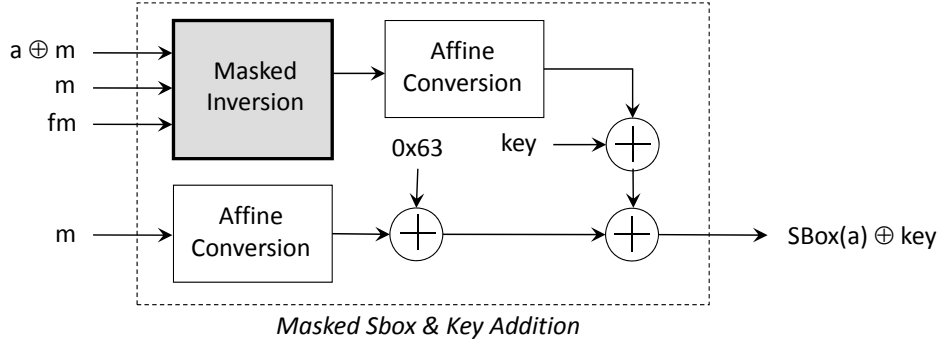


Figure 3.2: Design under Test.

the unmasked data still show dependence between the power and the mask. Furthermore, we make use of this dependence to introduce mask bias and successfully mount a power attack.

First, we describe the circuit module used for test in this experiment. We use a masked AES SBox (already mentioned in Section 3.2) with a key addition as the design under test. The masking methodology is given by [50]. The structure can be found in Figure 3.2.

We include an addition of a secret key value at the SBox output. The objective of our side channel analysis will be to find the secret key. In the simulation presented in this chapter, the key is 35.

The simulation is based on the gate-level netlist of the above design. To get this netlist, we first implement the masking scheme with GEZEL [93]. Next, we convert the GEZEL code to VHDL code and do synthesis with Design compiler. During synthesis, we must ensure that the XOR operations remain atomic elements. Therefore, the XOR operations are implemented in a separate hierarchy, and every XOR gate is a separate module. During synthesis, we then set a *don't-touch* attribute to the XOR modules, which prevents them from further logic expansion and/or optimization. The resulting synthesized VHDL netlist from Design Compiler is then converted back to GEZEL for hamming-weight simulation. As testbench, we exhaustively enumerate all three inputs from the masked SBOX: the masked data input $a \oplus m$, the mask signal m , and the fresh-

mask fm . For each triplet at the input, we obtain the hamming weight of the netlist, and we record the value of the SBOX output. These data sets are next subjected to our slicing attack.

Figure 3.3a shows the probability density function as the joint probability of the mask m and the power level p . Figure 3.3b is a detailed view of the elliptical area labeled in Figure 3.3a. Figure 3.3c illustrates the joint probability distribution of power p and mask m for two different values (128 and 135) of the mask.

Figure 3.3a and Figure 3.3b are 3-d graphs with power, mask, and joint probability labeled on ‘X-axes’, ‘Y-axes’, and ‘Z-axes’ respectively. The reason why only power and mask are shown here is that we draw the graphs from the perspective along with the Z-axes (XY view). The magnitude of probability is represented by the brightness. Figure 3.3c (XZ view) shows the joint probability distribution when $m = 128$ and $m = 135$. As we can see, the probability changes as the mask changes, which demonstrates the dependence between the power and the mask.

The next step is to find a way to exploit biased mask. As was demonstrated with Equation 3.10, we can get a biased mask signal by selecting samples from a restricted range of power levels. This reduced set of samples can then be used for side-channel analysis. Of course, the range of power levels must be sufficiently large so that a DPA can still succeed. In our first attempt, we only included samples with a power level in the slice from [0:216] (See Figure 3.3a). This range includes 5% of the total number of samples.

We can calculate the conditional probability mass function (PMF) of the mask as a parameter of the power level range selected. (We use PMF here instead of PDF, since, in the simulation, the power dissipation is a discrete random variable.) The marginal probabilities for the mask values are illustrated in Figure 3.4. We included two different curves. The dotted line is the marginal probability when we choose to include all samples. As expected, the dotted line is constant, which means that we are using an unbiased mask signal. The solid line in Figure 3.4 represents the marginal probability of the mask

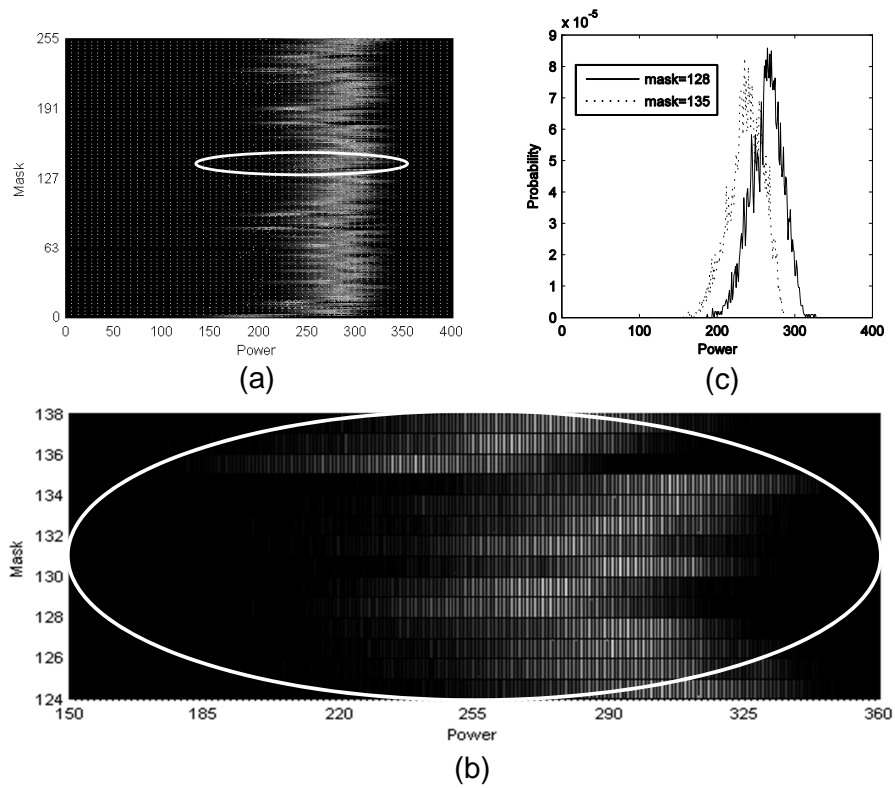


Figure 3.3: Joint probability of power and mask. (a) Joint Probability $Pr(p, m)$ for a masked AES SBox; (b) Detailed view of the elliptical area labeled in (a); (c) Joint probability $Pr(p, m)$ when $m = 128$ and $m = 135$ illustrate dependence of the power on the mask.

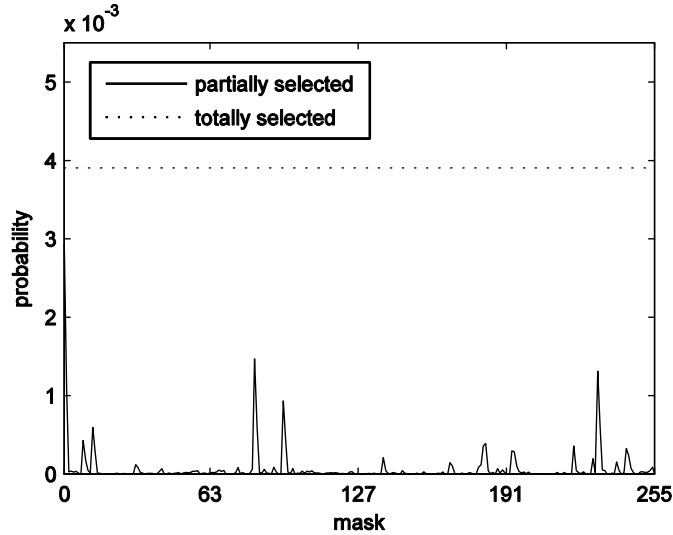


Figure 3.4: Total and partial probability mass function of the mask. The partial probability mass function is over selected range of power values ([0:200]).

signals for partially selected samples with a power level in the slice [0:216]. This line is not a constant, which means that the mask is biased.

We can further analyze the probability for each bit in the mask. This probability is represented in Figure 3.5. This figure illustrates that, for power samples in the slice [0:200], the probability for mask bits to be 1 is almost always less than 0.5. In other words, a reasonable estimate for the mask for all power samples in the slice [0:200] is an all-zero value. What should be mentioned is that the slice [0:200] is a simple choice but maybe not the optimal one. However, this example is enough to demonstrate our analysis in Section 3.2.

Finally, we can mount a DPA attack with this slice of power samples by assuming that the mask is always 0. While our guess for the mask is not deterministically correct, it is a correct guess from a statistical point of view. We have implemented the DPA based on the correlation of the hamming weight of unmasked input a . Figure 6 shows the resulting correlation graph. The highest peak (0.1658) appears at 'guess_key = 35' which is correct. Three other peaks can also be distinguished at 79 (-0.1476), 112 (0.1433), and 159 (-0.1459). In our simulated attack, guess_key 35 is the correct one.

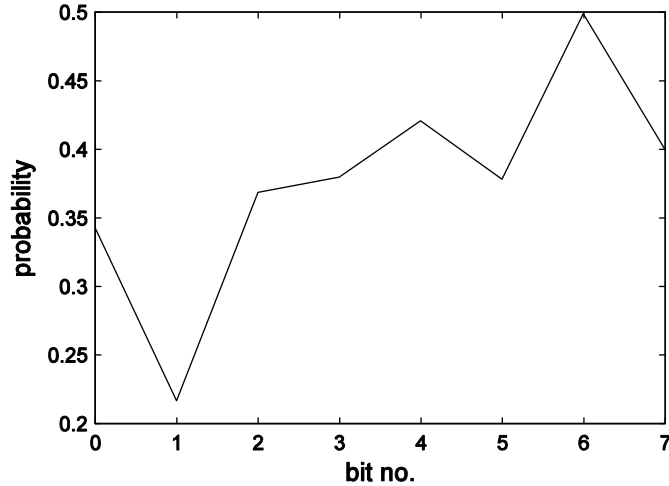


Figure 3.5: Resulting mask bias for the selected range of power values.

Figure 3.6 demonstrates how we successfully identified the correct key, even though the figure indicates that there are several other candidates with a correlation value close to the optimal one. The cause of this effect is that our mask estimation is obtained through a stochastic process. What’s more, the fresh-mask fm is still unknown, this makes it harder to get a obvious peak. Still, we see that due to partial observation of the power samples, the key guess space due to masking is reduced from 256 possibilities to only 4. Right now the power model used for attack is very simple. As the model improves, the attack result should be better.

3.4 Conclusion

This chapter illustrated how mask bias can be obtained on algorithmic masking. The power dissipation of masked hardware circuits is uncorrelated to the unmasked data values, and therefore cannot be used for DPA. However, we showed that the power dissipation of a masked hardware circuit may still be correlated to the mask. Because of this correlation, it is possible to bias the mask by selecting only a small slice over the entire power probability density function. We applied this technique using an AES SBox

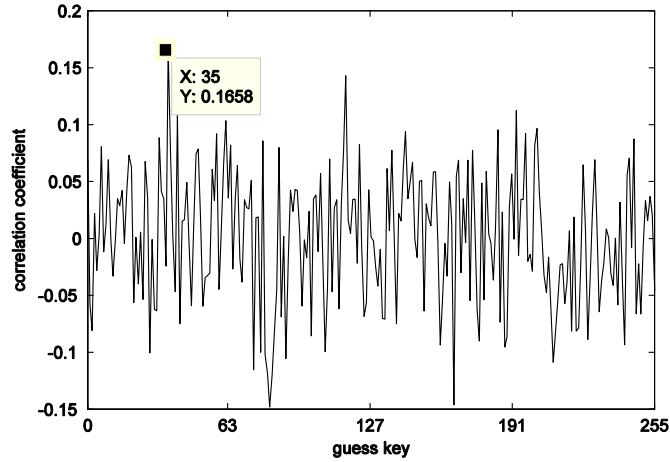


Figure 3.6: DPA correlation graph on signal with mask bias.

with perfect masking. Using logic-level simulation, we demonstrated the dependence between the power dissipation and the mask value. By slicing the power PDF before mounting a DPA, we can bias each bit from the mask. In our case, we introduced a bias towards logic-0. Our conclusion is that algorithmic masking remains susceptible to direct DPA by making clever use of the power probability density function.

Chapter 4

Higher-Order Circuit Effects Break Secure Circuits

In this chapter, we continue discussing security issues of cryptographic implementations. This chapter, together with Chapter 3, help us to understand the existing SCA countermeasures. Accordingly, we propose new protection solutions in the following chapters.

Instead of looking into signal processing methods, here we focus on circuit-level issues. We take a close look at the circuit properties that depend on more than one value in the circuit. We call them *higher-order circuit effects*. We first derive two conditions for secure masking of circuits: 1) the random masks need to be unbiased, and 2) the power consumption needs to be independent of the unmasked data. In logic simulations, for example in simulations based on toggle counts, these two conditions can be satisfied by masking. In a real circuit, however, we find the second condition cannot easily be met because of higher-order circuit effects. While higher-order effects have been mentioned as a possible source of side-channel leakage [94], no detailed analysis has been presented.

Our contribution in this chapter includes detailed analysis of several common higher-order circuit effects, including glitches, inter-wire capacitance, and IR drop. Our analysis explains, in a consistent manner, why all higher-order circuit effects can cause side-

channel leakage in masked circuits. As far as we are aware, no such analysis has been performed before. Our second contribution is to demonstrate and to quantify the impact of circuit effects on side-channel leakage. We present HSPICE simulations on an algorithmic masked $\text{GF}(2^2)$ multiplier. In our experiment, we conclude that the leakage caused by each of the above-mentioned higher-order circuit effects are comparable.

4.1 Analysis on Masked Circuits

In this section we present an analysis of masked circuits. We first derive two conditions to implement perfect masking. Next, we show that higher-order circuit effects (such as glitches, inter-wire capacitance, and IR Drop) may break these conditions.

4.1.1 Two Conditions for Perfect Masking

The perfect masking condition requires that a logic-0 and a logic-1 appears with the same probability on all intermediate circuit nodes. Let's consider such an intermediate plaintext node a , which is masked using a mask m by means of Boolean masking (XOR). The value of masked node a_m is given by the following truth table.

a_m	m	a
0	0	0
0	1	1
1	0	1
1	1	0

Let F_0 be the probability that $m = 0$, and F_1 the probability that $m = 1$. Obviously $F_0 + F_1 = 1$. We can express the expectation of power consumption P with respect to the unmasked signal a as follows.

$$\begin{cases} P(a = 0) = F_0 \cdot P(a_m = 0, m = 0) + F_1 \cdot P(a_m = 1, m = 1) \\ P(a = 1) = F_0 \cdot P(a_m = 1, m = 0) + F_1 \cdot P(a_m = 0, m = 1) \end{cases} \quad (4.1)$$

In order to implement the perfect masking condition, the power consumption of the circuit needs to be independent from a in a statistical sense. Therefore, the first and second formula in Equation 4.1 should have the same expectation. There are many ways to fulfill this condition. The most common approach is to require the following.

$$F_0 = F_1 \quad (4.2)$$

$$P(a_m = 0, m = 0) + P(a_m = 1, m = 1) = P(a_m = 1, m = 0) + P(a_m = 0, m = 1) \quad (4.3)$$

We consider Equation 4.2 and Equation 4.3 as two general conditions for secure masked circuits. Equation 4.2 shows that the mask signal needs to be unbiased, while Equation 4.3 shows that the circuit must consume the same power for each value of the unmasked input in a statistical sense.

The above conditions can be easily expanded to more general masking arrangements. Given a circuit block with masked data input a_m , and mask m , where each of these can be words, then we can write the power consumption in terms of the unmasked data a as follows.

$$P(a) = \sum_m F_m \cdot P(a_m, m) \quad (4.4)$$

F_m is the probability distribution of the mask, while $P(a_m, m)$ is the power consumption of the masked circuit for each possible combination of mask and masked value. Accordingly, we can define two general conditions for secure masked circuits as follows:

$$F_i = F_j \Big|_{i \neq j} \quad (4.5)$$

$$\sum_m P(a_m = i \oplus m, m) = \sum_m P(a_m = j \oplus m, m) \Big|_{i \neq j} \quad (4.6)$$

The first condition Equation 4.5 requires the mask to have a uniform distribution. The second condition Equation 4.6 requires the circuit to consume the same power for

each possible value of the unmasked input in a statistical sense.

Thus, conditions Equation 4.2 and Equation 4.3, as well as their generalizations Equation 4.5 and Equation 4.6, express when perfect masking is achieved by a masked circuit implementation. First-order attacks on masked circuits are enabled by violation of either of these conditions. For example, it is known that a bias in the mask causes first-order side-channel leakage [70]. Indeed, bias represents a violation of Equation 4.2 or Equation 4.5. In order to understand the implications of the condition Equation 4.3 or Equation 4.6, we need a better understanding of the power consumption P . In a digital circuit, the dynamic power consumption is given by the following equations [95].

$$P_{avg} = a \cdot f_c \cdot V^2 \cdot C \quad (4.7)$$

$$P = I \cdot V \quad (4.8)$$

In Equation 4.7, P_{avg} is the average power consumption. α is the switching factor, f_c is the clock frequency, V is the voltage of power supply and C is the effective capacitance. In Equation 4.8, P is the instant power consumption. I is the instant current and V is the instant voltage. For a real attack, the average power consumption is the average value of all the power samples in one cycle. The instant power consumption corresponds to only one power sample. Both of them can be used for the side-channel analysis. We need to examine every term in Equation 4.7 and Equation 4.8 for a possible dependency to the value of the unmasked signal a . If such a dependency is found, then the condition of Equation 4.3 or Equation 4.6 no longer holds and first-order side-channel leakage appears.

4.1.2 Higher-Order Circuit Effects Causing Side-Channel Leakage

As it turns out, real circuits have a large number of higher-order effects that can cause the terms of Equation 4.3 and Equation 4.6 to become dependent on the circuit

state. We will describe three of them: glitches, inter-wire capacitance, and IR Drop. There may be other higher-order effects in circuits, but it is not our intention to be exhaustive. Instead, our objective is to show *how* and *why* higher-order circuit effects can be used to relate different intermediate circuit values. When these values contain both the information of the mask and the corresponding masked value (not necessary to be exactly m and a_m), the previous ‘perfect’ masking is not perfect anymore.

Glitch

First, consider again the effect of glitches. A glitch results in additional net transitions in a circuit, so that the switching factor α in Equation 4.7 appears to be increasing. Glitches are also state-dependent. For example, a glitch may appear when $a_m = 1$ and $m = 1$ ($a = 0$) but not in any other combination. Hence, glitches may cause an imbalance in Equation 4.3, which in turn results in a violation of the perfect masking condition.

Inter-Wire Capacitance

Second, consider the effect of the capacitance C on the average power consumption. The total capacitance of a circuit has two components: gate capacitance and wire capacitance. In deep submicron technology, the inter-wire capacitance accounts for more than 50% of the total capacitance for narrow wires [96]. Modeling of the wire capacitance in an integrated circuit is a non-trivial task. For simplicity, the following discussion is on the basis of the simplified model as shown in Figure 4.1. This circuit shows two wires $w1$ and $w2$, each with a different driver. There are two parts for the capacitance seen by each wire, one part between the wire and the semiconductor substrate ($C1$ and $C2$), and the other part between adjacent wires ($C3$). Wire $w1$ sees a single, fixed capacitor $C1$ and a second floating capacitor $C3$. The effective value of $C3$ as seen from wire $w1$ changes in function of $w2$ ’s voltage level. For example, when $w1$ carries a logic-1 while $w2$ carries a logic-0, then the effective capacitance on $w1$ is $C1 + C3$. However, when

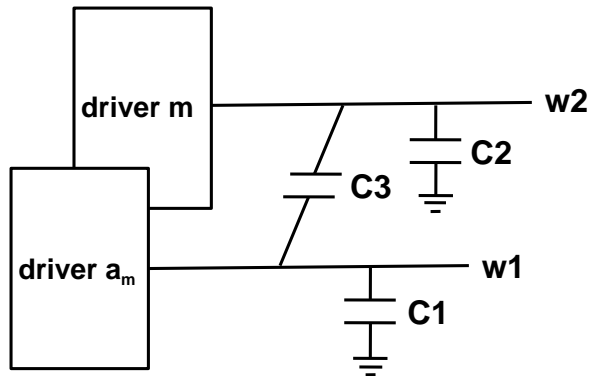


Figure 4.1: Model for inter-wire capacitance.

both $w1$ and $w2$ carry a logic-1, then the effective capacitance on $w1$ is only $C1$, since there is no current required to charge $C3$.

Now assume that wire $w1$ is driven by the masked signal a_m and wire $w2$ is driven by the mask m . Evaluating Equation 4.3, and assuming that logic-0 does not consume power, we find for the left side of Equation 4.3:

$$P(a_m = 0, m = 0) + P(a_m = 1, m = 1) = 0 + f_c(V^2 \cdot (C1 + C2))$$

While the right side evaluates to:

$$P(a_m = 1, m = 0) + P(a_m = 0, m = 1) = f_c \cdot V^2 \cdot (C2 + C3) + f_c \cdot V^2 \cdot (C1 + C3)$$

Clearly, the right hand side is a factor $2 \cdot f_c \cdot V^2 \cdot C3$ bigger than the left hand side. This factor is caused by taking inter-wire capacitance $C3$ into account. As chip feature sizes continue to shrink, inter-wire capacitance becomes more significant. This implies that the possible asymmetry of Equation 4.3 is likely to deteriorate further with shrinking feature size.

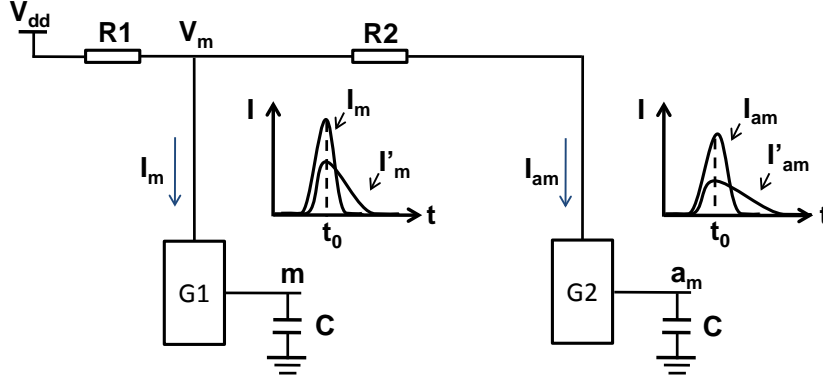


Figure 4.2: Model for IR Drop.

Table 4.1: State changes and corresponding currents.

State Change	Current into G1	Current into G2
m (0 to 0), a_m (0 to 0)	0	0
m (0 to 1), a_m (0 to 0)	I_m	0
m (0 to 0), a_m (0 to 1)	0	I_{am}
m (0 to 1), a_m (0 to 1)	I'_m	I'_{am}

IR Drop

Third, consider the influence of ‘IR drop’ [95] on the instant power consumption. In an integrated circuit, the power supply is connected to every gate through a network called ‘power grid’. When current flows from the power pins to each gate, the resistance of the power grid causes a local decrease of the voltage which is called IR drop.

In Figure 4.2, there are two gates $G1$ and $G2$, driving a mask m and a masked value a_m respectively. The power grid includes two resistors. V_{dd} is the voltage on the external chip input, and V_m the local voltage at gate $G1$. Suppose m and a_m are both 0. Then the circuit has 4 possible state changes that are described in Table 4.1. Table 4.1 also defines the currents for each gate during each state change.

When we ignore IR drop, I_m and I'_m would be identical. However, due to the resistances in the power grid, the instant switching current depends on the number of switching gates. The current profiles for I_m and I'_m are as shown in Figure 4.2. If we

consider the instant current near the inputs' switching activity (for example $t = t_0$) for each case, then we find:

$$I_m|_{t=t_0} > I'_m|_{t=t_0} \quad (4.9)$$

and similarly

$$I_{am}|_{t=t_0} > I'_{am}|_{t=t_0} \quad (4.10)$$

Evaluating this observation in Equation 4.3, the left hand side is equal to

$$P(a_m = 0, m = 0) + P(a_m = 1, m = 1) = V_{dd} \cdot 0 + V_{dd} \cdot (I'_m + I'_{am})$$

The right hand side, on the other hand, is equal to

$$P(a_m = 1, m = 0) + P(a_m = 0, m = 1) = V_{dd} \cdot I_{am} + V_{dd} \cdot I_m$$

Because of Equation 4.9 and Equation 4.10, we see that the instant power consumption at t_0 for right hand side is larger than the instant power consumption at the same time for left hand side. Hence, Equation 4.3 may not be satisfied because of IR drop and a side-channel leak appears.

According to the above analysis, higher-order circuit effects make the correlation between different intermediate circuit values a common phenomenon in a real circuit. This presents a risk for the perfect masking scheme.

4.2 Experimental Results

In this section, we describe a series of experiments that we did to test the above analysis. We will not only demonstrate that the higher-order circuit effects lead to side-channel leakage, but also show that the amount of leakage of each of them are comparable.

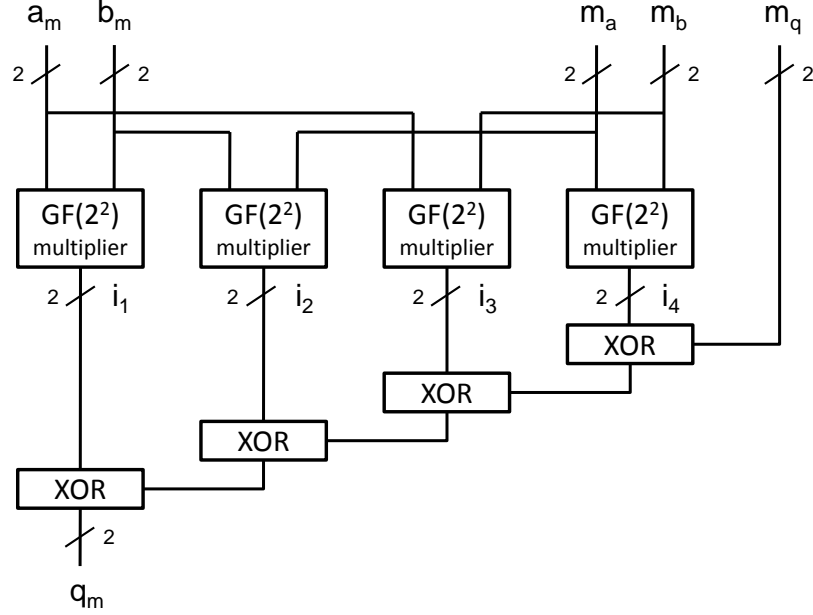


Figure 4.3: Masked Galois Field multiplier.

Ideally, we would use actual masked circuits and physical measurements in order to characterize these side-channel leaks. This has the disadvantage that any measurement of side-channel leakage cannot be attributed to individual circuit effects, but only to the circuit as a whole. For this reason, we initially investigated a masked circuit using HSPICE simulation, and we constructed the simulation setup such that effect of glitches, inter-wire capacitance and IR-drop could be investigated separately. Our approach is to explore the difference between the circuits with and without higher-order circuit effects.

The circuit under simulation is shown in Figure 4.3. It is a masked Galois Field multiplier: a critical part of a masked AES S-Box. The circuit has 5 inputs: a_m , b_m , m_a , m_b , and m_q and 1 output: q_m (2 bit for each). a_m and b_m are masked values of a and b ($a_m = a \text{ XOR } m_a$, $b_m = b \text{ XOR } m_b$); q_m is the masked value of q ($q_m = q \text{ XOR } m_q$). The circuit consists of 4 GF(2²) multipliers and a set of XOR gates (36 standard gates in total).

Glitches. We perform two simulations to test glitches: one with the entire circuit in Figure 4.3; the other only with the GF(2²) multipliers. According to [79], only the XOR

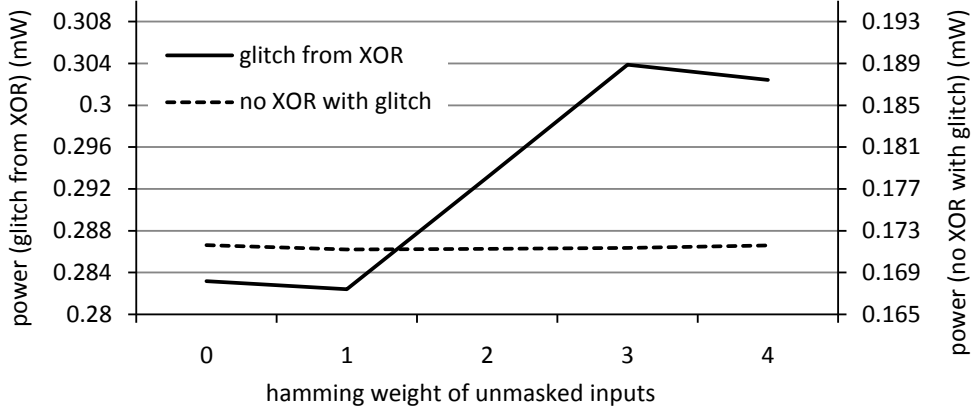


Figure 4.4: Simulation results for glitches. Delation between hamming weight of unmasked inputs and power is depicted.

gates leak side-channel information through glitches. Therefore, we expect unmasked-data dependent power in the first experiment while no leakage in the second. In each simulation, we take the following steps:

- 1): Switch the input from 0 to every possible value n . Accordingly, we obtain 1024 average current values for the entire circuit, proportional to the average switching power.
- 2): Every average current value is mapped to a set of unmasked inputs a and b . We group the 1024 average current values in terms of the hamming weight (from 0 to 4) of the unmasked input. By averaging each group, we obtain the mean power for each hamming weight as the experimental result.

The result is shown in Figure 4.4. As we can see, the majority momentum of mean power in the first simulation is increasing as the hamming weight of the unmasked inputs increases. In contrast, mean power in the second simulation almost remains the same. Hence, we can attribute the leakage to the glitches in the XOR gates.

What should be mentioned is that glitches in the XOR gates are related to the arrival time of the inputs which is decided by the layout of the circuit and other factors.

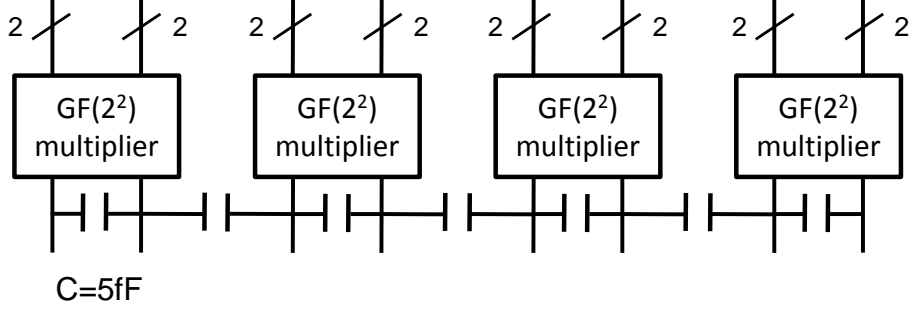


Figure 4.5: Inter-wire capacitance between outputs and inside GF(2²) multipliers.

In our experiment, we performed the first simulation several times with different arrival sequence of the inputs. The results turn out to be similar as the one shown in Figure 4.4.

Inter-Wire Capacitance. Inter-wire capacitance can exist between the outputs of the GF(2²) multipliers. We can also find inter-wire capacitance between nets in different GF(2²) multipliers, if they are placed close to each other. Furthermore, inter-wire capacitance is influenced by many factors, for example the layout of the circuit. In our simulations, we made a reasonable assumption for inter-wire capacitance: comparable to the gate capacitance [96]. Because it is really hard to eliminate glitches from the XOR gates, to see the individual influence of the inter-wire capacitance, we rule out the XOR gates and perform simulation just on the GF(2²) multipliers.

The first simulation is performed with inter-wire capacitances added between the outputs of the GF(2²) multipliers, shown in Figure 4.5. The second one is done without inter-wire capacitance, which is exactly the same as the second simulation for glitches. In each simulation, we take the same steps mentioned in the previous subsection.

The results are presented in Figure 4.6, where we can see the average power increases as the hamming weight goes from 0 to 4 in the first simulation but changes little in the second one. Besides that, if we increase the value of the inter-wire capacitances, the average power for the first simulation increases more quickly.

IR Drop. Like the experiment for inter-wire capacitance, we only use the GF(2²) multipliers (without XOR gates) in the simulation to test IR drop. The method of the

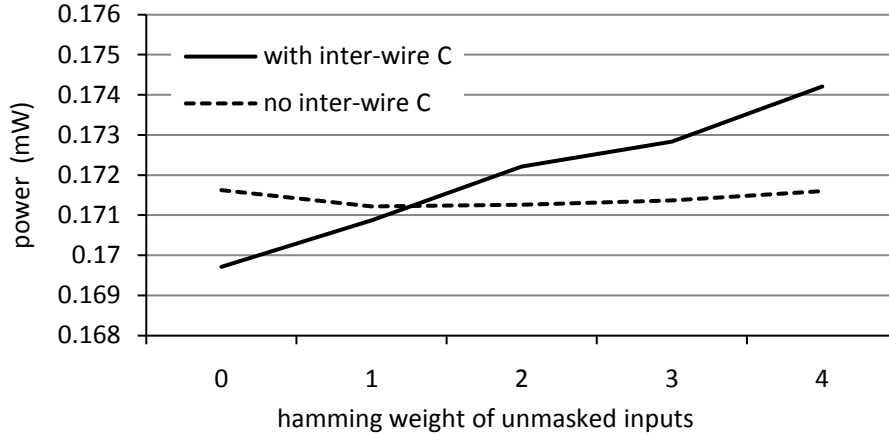


Figure 4.6: Simulation results for inter-wire capacitance. Dependency exists between hamming weight of unmasked inputs and power with inter-wire capacitance.

simulations for IR drop is similar to the previous one. We first simulate the circuit without resistance in the power grid. After that, resistances with a reasonable value [96] are added as shown in Figure 4.7. Vdd is the power input from outside. Every line in the Figure 4.7 is a wire of the power grid. We connect the crossing points of these wires to the power input of each logic gate in the circuit under test. We use the similar way as the previous sub-sections to add stimuli to the inputs and to process the current traces. The only difference is that we do not use average values, but a single value sampled not long after the inputs' switching activity. The results in Figure 4.8 indicate that the addition of resistances into the power grid causes more instant power for higher hamming weight. We also did simulations with different resistance values. Results show that the larger the resistance is, the more the power increases as the hamming weight goes from 0 to 4.

According to the perfect masking conditions, the power should be independent on the unmasked inputs. With no doubt, it should also be independent on the hamming weight of them. From the above experimental results, we can see the higher-order circuit effects discussed in Section 4.1.2 introduce dependence between the power and the unmasked input. Therefore, they are indeed possible sources of side-channel leakage. Furthermore, as the hamming weight increases, the power in the Figure 4.4, Figure 4.6, and Figure 4.8

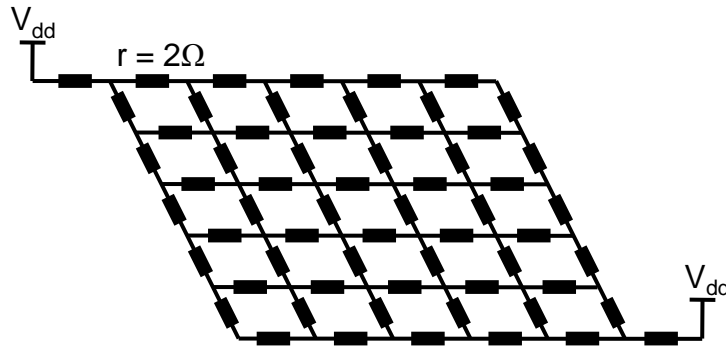


Figure 4.7: Power grid resistance.

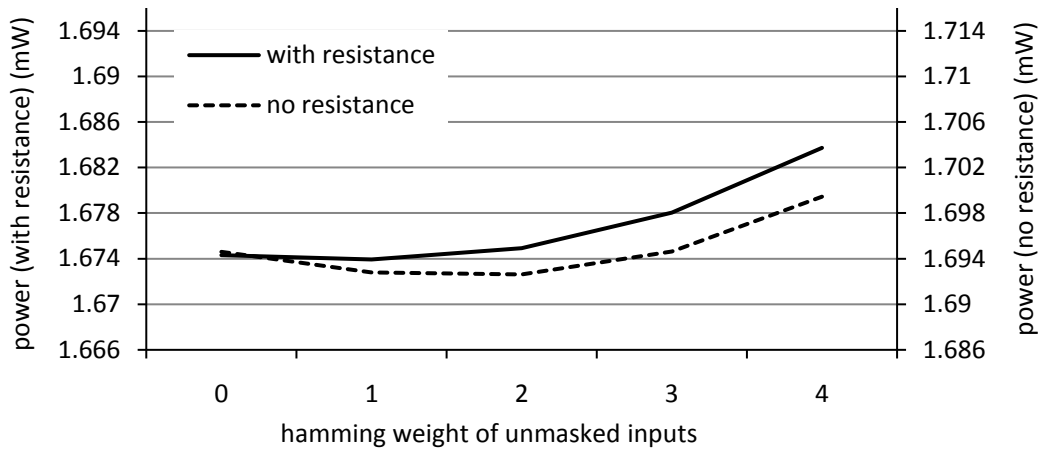


Figure 4.8: Simulation results for IR Drop. Dependency exists between hamming weight of unmasked inputs and power with IR Drop.

all goes up. This means the effects of these higher-order circuit effects are additive. In a real circuit, where all these circuit effects exist, we can find the overall leakage will be larger than the one caused by each one of them.

We also notice that the power changes slightly even when no higher-order effects are added. We consider this as influence from some existing circuit effects, such as static leakage and early propagation.

We also quantify the relative side-channel leakage for each circuit effect. It is not easy to define the magnitude of a leakage. Usually, this is partially dependent on the attack method. Here, for the convenience, we base our analysis on the attack with hamming weight. According to the relationship between power consumption and Hamming weight of the unmasked inputs, we can find the maximum power variations for glitches, inter-wire capacitance and IR drop are 0.025 mW, 0,0045 mW, and 0.0094 mW respectively. The ratio is 5.6 : 1 : 2.1. Clearly, their leakage are comparable.

4.3 Conclusion

This chapter introduced an analysis of masked circuits from the circuit-level perspective. We defined two general conditions for secure masking. Both of these conditions can be easily achieved at the logic-level, which abstracts voltage into discrete logic-levels and which abstracts time into clock cycles. However, the conditions for secure masking are much harder to achieve in real circuits, in which we have to allow for various electrical and analog effects. We showed that glitches, inter-wire capacitance, and IR Drop can all cause side-channel leakage. We evaluated our analysis using HSPICE simulations, and we found that leakage caused by these 3 circuit effects is comparable.

We summarize Chapter 3 and Chapter 4 as follows. Even though the secure circuits have already traded several times [39] or even more than 10 times [97] more area and power than regular circuits for better SCA-resistance, low-level physical characteristics of circuits still cause vulnerabilities. And it can be anticipated that solutions to these

low-level circuit issues may require even more area and power. Considering that the existing secure circuits already found it hard to be accepted by the industry due to their high area and power costs, it is reasonable to expect a dead end even if we can find solutions to those low-level circuit issues.

On the other hand, our advanced SCA analysis method demonstrated that simple randomization technique is not enough to protect secure algorithms. More complicated randomization schemes are required [53, 98]. As expected, this increases the costs and makes the solutions even more algorithm-specific.

Therefore, we consider that simply using either the secure circuits or the masking algorithms will be infeasible to provide efficient or practical countermeasures.

Chapter 5

SCA-Resistant Software on Dual-Core Processor

Starting from this chapter, we present our SCA-resistant solutions. According to Chapter 3 and Chapter 4, we notice that neither secure algorithms nor secure circuits can deliver practical SCA-resistant solutions individually. Our understanding is that a feasible SCA-resistant solution requires the designers to search for a design platform that can easily find a balance between different aspects, including security, performance, and cost. In our opinion, a microprocessor that possesses SCA-resistant features could be such a platform. On the one hand, a microprocessor by itself is a hardware circuit, to which secure circuit techniques can be applied. In the meantime, since microprocessor is a reusable hardware, this means that the additional hardware cost spent on the secure circuit is also reusable. This alleviates the pressure of high hardware cost on secure circuit solutions. On the other hand, as microprocessor connect both software and hardware, it is easy to make design trade-offs between performance and hardware cost. Therefore, microprocessor offers the designers a lot of flexibility to find balanced solutions. Moreover, since software is the most popular cryptographic implementation on embedded systems, such solutions can find its position in a large amount of applications.

In this chapter, we introduce our first SCA-resistant solution based on a dual-core

processor. We propose a protection solution to cryptographic software with the dual-rail pre-charge (DRP) technique. We will show that our method creates the software equivalent of a DRP secure circuit. We name this concept Virtual Secure Circuit (VSC).

The key contributions of this chapter are as follows. *First*, we perform security analysis on VSC and explain why a software-based DRP technique can withstand side-channel power attacks. *Second*, we construct a VSC protected AES prototype on a dual-PowerPC computing platform. *Third*, we successfully demonstrate the effectiveness VSC’s protection by means of a set of side-channel attacks on the prototype based on real-world measurements. Attack results show that dual-core VSC offers a considerable improvement on the security by increasing the attack difficulty at least 80 times. Even 32 million measurements are not sufficient for a successful full attack.

5.1 Virtual Secure Circuit (VSC)

In this section, we introduce the Virtual Secure Circuit (VSC), a software concept equivalent of DRP circuit in hardware, which has already been detailed as a popular SCA countermeasure in Section 2.1.3. We first clarify a few initial assumptions on the hardware. Next, we describe a dual-core architecture that serves as the target for VSC, and we present the design of a VSC by means of an example. Finally, we show that a VSC is functionally equivalent to a DRP circuit in hardware, with similar security properties.

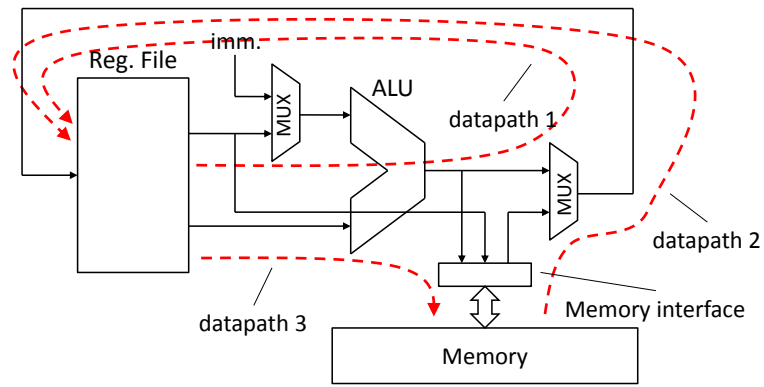
5.1.1 Micro-Processor Assumptions and Side-Channel Leakage

While VSC is a software-based concept, its ultimate objective is to reduce the side-channel leakage originating from hardware, and more specifically from the microprocessors that execute VSC software. Because microprocessors exist in all shapes and sizes, we make the following assumptions regarding their implementation. First, we assume that we can build the multi-core platform from small microprocessors or micro-controllers,

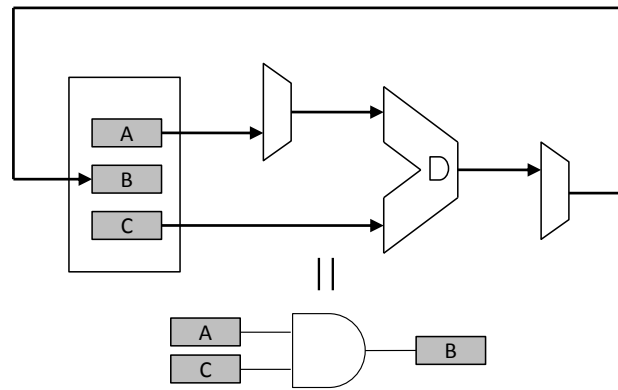
which have a well-defined instruction-execution pattern. For example, we assume deterministic memory-access time (no cache), and we assume there are no state-dependent processor features (no branch predictors). Second, we assume that VSC can run as an atomic thread of control on the multi-core architecture. Thus, we assume that interrupts and exceptions can be disabled for the duration of the complementary program execution. These two assumptions ensure that it is feasible to maintain synchronization between the cores. These assumptions do not mean that VSC will never be able to support more complex architectures. In this chapter, as the first step, we only focus on the simple case. Even in its simplest form, we can still point out practical implementation scenarios for VSC. For example, tiled processors [99], an important category of multi-core processors, are usually built with small processing cores and local memories. The local memories for 8 Synergistic Processing Elements (SPE) in the Cell processor [90] are not cache either.

Given the above assumptions, we now analyze what parts of the micro-processor are potential sources of side-channel leakage. For this purpose, we analyze the flow of information within a microprocessor, as shown in Figure. 5.1a. We can distinguish three different datapaths. The first datapath is the computational datapath. It starts from the register file, goes through the Arithmetic Logic Unit (ALU), and returns to the register file. There are two additional datapaths for memory-operations. The memory-load datapath is used to transfer information from memory to the register file. The memory-store datapath is used to transfer information from the register file to the memory.

Each of the above datapaths is a potential source of side-channel leakage. For example, Figure. 5.1b shows the execution of an `and` operation, which configures the computational datapath as an array of `AND` gates. Clearly, this operation leads to data-dependent power dissipation which must be avoided. If we can protect each of these three datapaths, the microprocessor instructions that only make use of these datapaths will also be secure. In the following section, we explain how this is done in VSC.



(a)



(b)

Figure 5.1: Processor architecture. (a) Three datapaths in a processor [100]; (b) Datapath for AND operation.

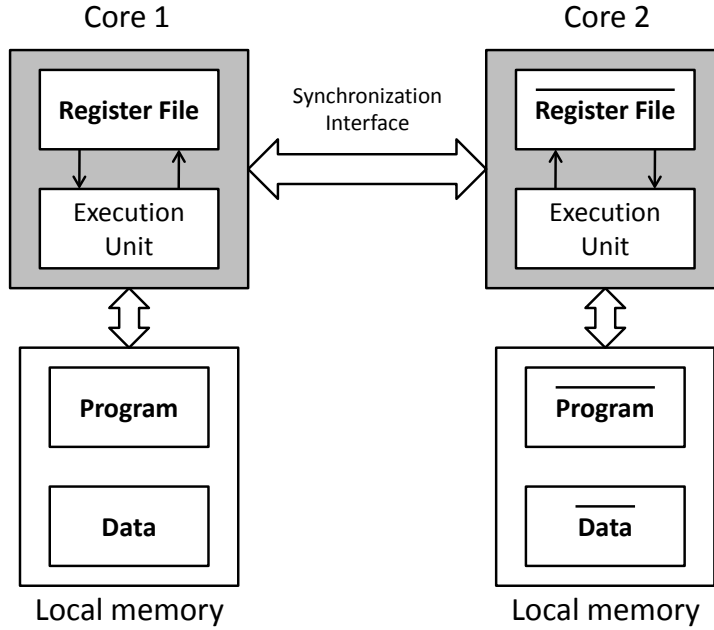


Figure 5.2: A dual-core architecture to implement a Virtual Secure Circuit.

5.1.2 Dual-Core Architecture for VSC

Figure. 5.2 illustrates a dual-core architecture used for a VSC. In a multi-core architecture, two structurally identical cores are selected. Both cores execute complementary versions of the same program: core 1 executes *Program*, while core 2 executes $\overline{Program}$. The rules for creating the instructions of *Program* and $\overline{Program}$ follow the principles of the DRP technique. For each instruction in *Program*, there is a complementary instruction in $\overline{Program}$. A *complementary instruction* pair maps complementary input data into complementary output data. Section 5.3 will describe such instruction pairs in more details.

During the execution of these complementary programs, both cores maintain cycle-accurate synchronization, which is achieved by means of a synchronization interface. This ensures that both cores execute complementary instruction pairs in the same clock cycle. As a result of executing complementary instructions, both cores also handle complementary data stored in their register file as well as in their local memory. To extend the DRP technique into storage, the register file and the local memory are both pre-

charged whenever new data is stored. In this way, any data-dependent power consumption in core 1 is matched by a complementary data-dependent power consumption in core 2, so that the overall power consumption of the multi-core system is data-independent.

To implement the above concept, we need 1) to convert the ordinary software into the complementary *Program* and $\overline{Program}$, and 2) to design a dual-core system where two cores can be synchronized with clock-cycle accuracy.

5.1.3 Creating *Program* and $\overline{Program}$

A case study is used to explain how the program conversion looks like. Figure. 5.3 shows the conversion of part of the AES encryption round. In Figure. 5.3a, the AES `AddRoundKey` and `SubByte` operations are implemented with two lines of C code (shown in bold typeface). The assembly code consists of a `xor` operation, a memory load operation, and a memory store operation. After VSC conversion, two complementary programs are shown in Figure. 5.3c. The conversion includes two steps. The first step is to create complementary instruction pairs. The second step is to add the pre-charge operations.

Step 1: Complementary instruction pairs are shown in bold typeface in Figure. 5.3c. We first consider the conversion of the `xor` operation. Its complementary instruction, `xnor`, is unavailable on the PowerPC instruction set used here. Therefore the original logic function $r8 = r7 \oplus r6$ is expanded into $r8 = r7 \cdot \overline{r6} + r6 \cdot \overline{r7}$ in *Program* and into $r8 = (r7 + \overline{r6}) \cdot (r6 + \overline{r7})$ in $\overline{Program}$. In Figure. 5.3c, the converted code uses 5 steps to complete the new equations with the help of 3 temporary registers $r9$, $r10$, and $r11$. The remaining `xor` operations, on line 2 and line 4, are used to invert the lowest byte and do not violate the complementary rule. Finally, since the memory-load (`lbzx`) and memory-store (`stb`) instructions do not change their operands' value, their complementary instructions are themselves.

Step 2: The instructions in regular typeface in Figure 5.3c are used for pre-charge. They do not affect the computation results. Instead, they reset the execution circuits

```

unsigned char in, key, out;
unsigned temp;
unsigned char sbox[256] = {0x63, ...};
temp = in ^ key;
out = sbox[temp];

```

(a) Original C program

```

; r6 = in, r7 = key, r3 = sbox = 0xXX00, r4 = &out
xor r8, r7, r6 ; add key
lbzx r3, r5, r8 ; sbox lookup
stb r3, 0(r4) ; store out

```

(b) Original Assembly

Program	Program
<i>; r6 = in, r7 = key,</i>	<i>; r6 = $\overline{\text{in}}$, r7 = $\overline{\text{key}}$,</i>
<i>; r0 = 0, r11 = 0xff,</i>	<i>; r0 = 0, r11 = 0xff,</i>
<i>; r12 = 0xXX00,</i>	<i>; r12 = 0xXX00,</i>
<i>; (r12) = 0</i>	<i>; (r12) = 0.</i>
<i>; xor conversion</i>	<i>; xor conversion</i>
1. xor r9,r0,r0	1. xor r9,r0,r0
2. xor r9,r7,r11	2. xor r9,r7,r11
3. xor r10,r0,r0	3. xor r10,r0,r0
4. xor r10,r6,r11	4. xor r10,r6,r11
5. and r11,r0,r0	5. or r11,r0,r0
6. and r11,r9,r6	6. or r11,r9,r6
7. and r9,r0,r0	7. or r9,r0,r0
8. and r9,r10,r7	8. or r9,r10,r7
9. or r8,r0,r0	9. and r8,r0,r0
10. or r8,r9,r11	10. and r8,r9,r11
<i>; lbzx conversion</i>	<i>; lbzx conversion</i>
11. lwzx r3,r12,r0	11. lwzx r3,r12,r0
12. lbzx r3,r5,r8	12. lbzx r3,r5,r8
13. lwzx r9,r12,r0	13. lwzx r9,r12,r0
<i>; stb conversion</i>	<i>; stb conversion</i>
14. stwx r9,r4,r0	14. stwx r9,r4,r0
15. stb r3,0(r4)	15. stb r3,0(r4)
16. stwx r9,r12,r0	16. stwx r9,r12,r0
17. lwzx r9,r12,r0	17. lwzx r9,r12,r0

(c) Converted VSC assembly code

Figure 5.3: An example of Virtual Secure Circuit. (a) KeyAddition and SubByte operations in C code; (b) Compiled assembly code; (c) Converted VSC assembly code.

and storage, between the computational instructions. Most instructions in Figure 5.3c (namely, those in the computational datapath) use a single pre-charge instruction. Memory-load and Memory-store instructions may need more than a single pre-charge instruction, depending on the presence of sensitive data in the address or data of the memory-access operation. Section 5.3 will discuss these in more details.

VSC is not a free lunch. Converting ordinary programs to VSC increases the software footprint and the execution time. The penalty of VSC will be shown in Section 6.3.

5.1.4 VSC Is Equivalent to DRP Circuit

Finally, we demonstrate that VSC is functionally equal to DRP circuit. Figure 5.4 illustrates the process to map a path of a DRP circuit to processor execution. We first convert a DRP circuit to a pipelined version according to the instruction set architecture of the processor. After that, we process the pipelined circuit sequentially. Based on this, we can map the VSC assembly code in Figure 5.3c to a DRP circuit, as shown in Figure 5.5. Each active (bold) instruction of *Program* or $\overline{Program}$ corresponds to a single logic gate (or function) in the circuit. The numbers annotated within the logic gates correspond to line numbers in the programs. In between each logic gate, a register is inserted. The pre-charge operation will reset that register before loading it with sensitive data. *Program* and $\overline{Program}$ execute in lockstep, and for each tuple of instructions, exactly two complementary gates of the circuit will evaluate. Hence, we conclude that VSC is a sequentialized version of DRP. Also, VSC may inherit the properties of DRP circuits.

5.2 Synchronization of two cores

The purpose of processor synchronization is to ensure that the complementary instructions are executed at the same clock cycle. This requires that not only the instructions but also the processor pipelines are synchronized. For parallel programming, we

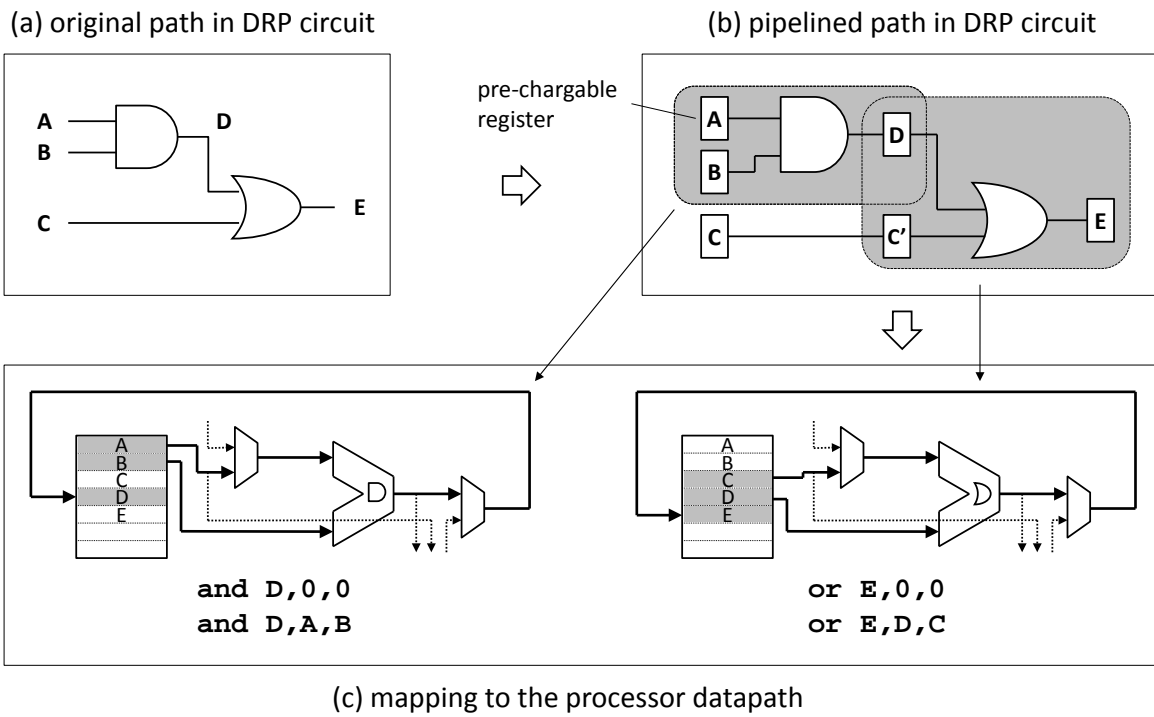


Figure 5.4: Mapping a DRP path to processor activity. (a) original path in DRP circuit; (b) pipelined path in DRP circuit; (c) mapping to the processor datapath.

<pre> Program ; r0 = 0, r11 = 0xff, ; r12 = 0xXX00, ; (r12) = 0 ; xor conversion 1. or r9,r0,r0 2. xor r9,r7,r11 3. or r10,r0,r0 4. xor r10,r6,r11 5. and r11,r0,r0 6. and r11,r9,r6 7. and r9,r0,r0 8. and r9,r10,r7 9. or r8,r0,r0 10. or r8,r9,r11 ; lbzx conversion 11. lwzx r3,r12,r0 12. lbzx r3,r5,r8 13. lwzx r9,r12,r0 ; stb conversion 14. stwx r9,r4,r0 15. stb r3,0(r4) 16. stwx r9,r12,r0 17. lwzx r9,r12,r0 </pre>	<pre> Program ; r0 = 0, r11 = 0xff, ; r12 = 0xXX00, ; (r12) = 0. ; xor conversion 1. or r9,r0,r0 2. xor r9,r7,r11 3. or r10,r0,r0 4. xor r10,r6,r11 5. or r11,r0,r0 6. or r11,r9,r6 7. or r9,r0,r0 8. or r9,r10,r7 9. and r8,r0,r0 10. and r8,r9,r11 ; lbzx conversion 11. lwzx r3,r12,r0 12. lbzx r3,r5,r8 13. lwzx r9,r12,r0 ; stb conversion 14. stwx r9,r4,r0 15. stb r3,0(r4) 16. stwx r9,r12,r0 17. lwzx r9,r12,r0 </pre>
---	--

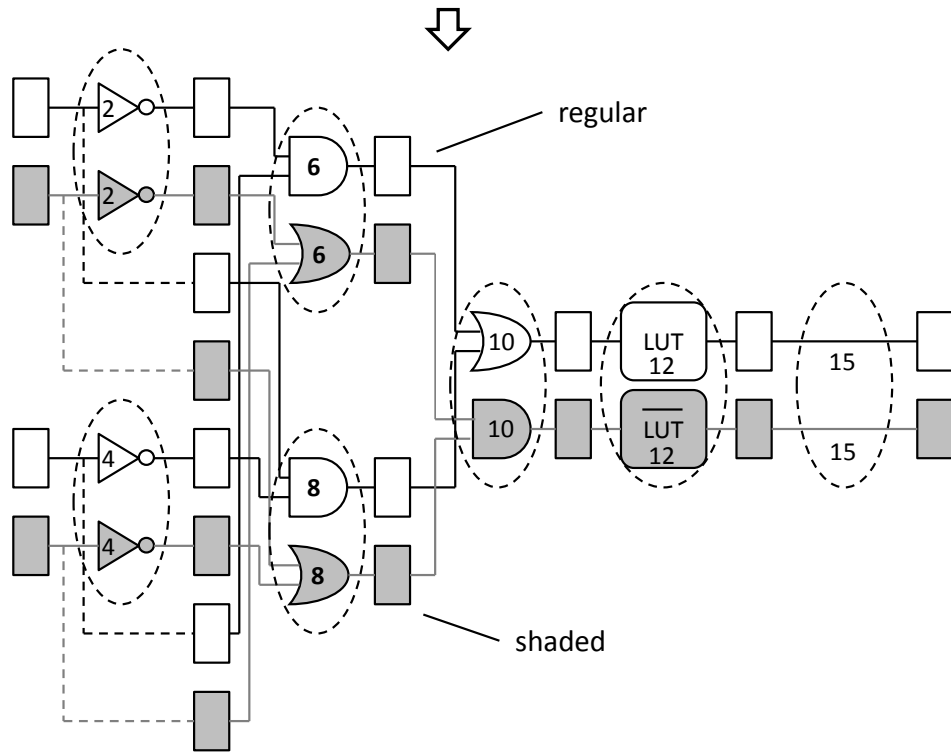


Figure 5.5: Mapping from software dataflow to secure circuit. By instantiating the each processor's active datapath at different time, we obtain a DRP secure circuit.

already have a synchronization method `barrier` that guarantees that any thread/process stops at this point and cannot proceed until all other threads/processes reach this barrier. However, `barrier` does not ensure the instructions right after it on different cores start at the same clock cycle. Another synchronization scheme at a lower level is needed. We call it `pipeline_sync`.

Before presenting our solution, we first introduce a bus protocol used by many processors. The peripheral bus usually has an `ack` signal which starts from a memory peripheral to the processor. During every `load` operation, the processor first uses the address bus and some control signals to send out a memory access request. After that, the selected peripheral uses the `ack` signal to notify the processor that the required data is ready on the data bus. If the selected peripheral is not able to offer the requested data right away, the `ack` signal will be kept invalid for a while until the data is ready. When waiting for the `ack` signal to be valid, the processor is in a fixed state (`wait_ack`).

Our solution makes use of the above memory access protocol, shown in Figure 5.6. A Synchronization Unit (SU) is attached to both cores' peripheral buses. Whenever one core initiates a `load` operation on the SU, the `ack` signal on its peripheral bus will be kept invalid. This means that every time a core tries to read data from SU, it enters the `wait_ack` state. When both cores are in the `wait_ack` state (several clock cycles after both of them initiate `load` operations on SU), `ack` signals become valid at the same time. So both cores jump out the `wait_ack` state at the same clock cycle and go on to process the following instructions. Moreover, if the `ack` signal is kept invalid for too long the processing cores consider this as an error and an exception will be launched. To avoid this, we first use a parallel programming's `barrier` to reduce the timing difference of the two cores before doing the `pipeline_sync`. This guarantees no exception occurs. By now, two cores have been totally synchronized. We repeat the above process every time before running the protected cryptographic software. In this way, the required synchronization is achieved.

The above synchronization scheme does not require any modification on the proces-

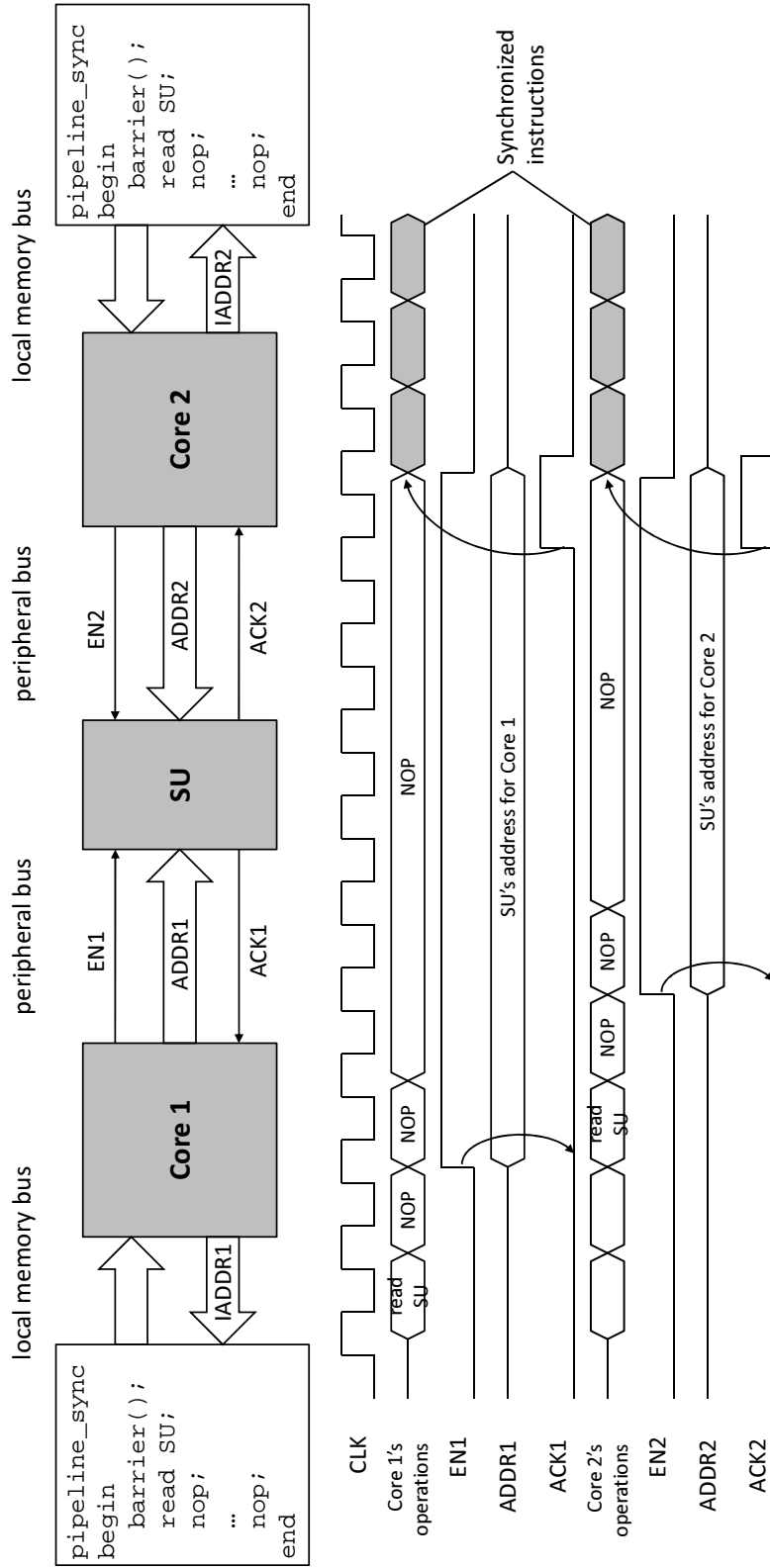


Figure 5.6: The synchronization scheme with an example.

sors. The SU is also very simple, so the cost is low. We tested it on PowerPC and MicroBlaze processors. Both worked correctly.

5.3 Implementation Details on VSC-AES

This section lists some detailed design issues of VSC-AES to share our experiences on the VSC programming techniques.

We first define two terms: ‘*sensitive data*’ and ‘*related data*’. *Sensitive data* is the intermediate data of the software that is related to the cryptographic key as well as the inputs that can be controlled by the user. *Related data* is the intermediate data that is not sensitive but will affect some *sensitive data*. Obviously, *sensitive data* is what we have to protect. Since *related data* will affect the *sensitive data*, during the VSC conversion, we also have to convert them into complementary format.

AES uses three kinds of instructions to process *sensitive data*: 1) the logic instructions, such as `xor`, `and`, `or`, and `not`; 2) the shift instructions; 3) the data instructions, such as `move`, `load` and `store`; Different categories operate differently in the processor. Accordingly, different conversion rules are needed.

Logic Instructions.

Logic instructions are easy to complement. For example, `and` and `or` are complementary. `not` complements itself. If a logic instruction has no complement in the instruction set (such as `xor`), we can decompose it into simpler logic operations and complement those. The rules for pre-charge are easy as well, and simply evaluate all-0 inputs for the pre-charged instructions. Figure 5.7 shows an example.

Shift Instructions.

Shift operations are usually implemented with several shift levels of combinational multiplex logic. They are basically data movement operations. So the complementary

logic instruction		shift instruction	
and r1,r2,r3		slwi r1,r2,imm (imm<=16)	
<i>Program</i>	$\overline{\text{Program}}$	<i>Program</i>	$\overline{\text{Program}}$
li r0,0 and r1,r0,r0 and r1,r2,r3	li r0,0 or r1,r0,r0 or r1,r2,r3	li r9,0 slwi r11,r9,imm slwi r11,r2,imm li r10,0 xor r1,r9,r9 xor r1,r11,r10	li r9,0 slwi r1,r9,imm slwi r1,r2,imm li r10,2 ^{imm} -1 xor r1,r9,r9 xor r1,r11,r10

data instructions			
stb r1,imm(r2)		lbzx r1,r2,r3 (access SBox)	
<i>Program</i>	$\overline{\text{Program}}$	<i>Program</i>	$\overline{\text{Program}}$
; (r12) = 0 ; r0 = 0 li r9,0 stw r9,imm(r2) stb r1,imm(r2) stwx r9,r12,r0 lwzx r9,r12,r0	; (r12) = 0 ; r0 = 0 li r9,0 stw r9,imm(r2) stb r1,imm(r2) stwx r9,r12,r0 lwzx r9,r12,r0	; (r12) = 0 ; r12 = 0xXX00 ; r2 = 0xXX00 ; r0 = 0 li r9,0 lwzx r1,r12,r0 lbzx r1,r2,r3 lwzx r9,r12,r0	; (r12) = 0 ; r12 = 0xXX00 ; r2 = 0xXX00 ; r0 = 0 li r9,0 lwzx r1,r12,r0 lbzx r1,r2,r3 lwzx r9,r12,r0

Figure 5.7: Representative conversion examples.

instruction are themselves. The same as logic instructions, shift instructions are also implemented in ALU. So the pre-charge method is the same as logic instructions. Besides that, special attention should be paid to the *related data* generated by shift operations. A shift or rotate instruction may shift the non-sensitive bits into dataflow, for example inserting 0 to the vacant bits. If these vacant bits are used to calculate the *sensitive data*, they become *related data*. Therefore, these bits need to be 1 in $\overline{\text{Program}}$. After the shift instruction, we need to invert the shifted-in *related data* in $\overline{\text{Program}}$ while keep them as they were in *Program*. Figure. 5.7 shows a representative example.

Data Instructions.

Data instructions do not change the value of the data, so *Program* and $\overline{\text{Program}}$ share the same data instruction. The pre-charge operations are more complex since they deal with the memory. This brings up the problem of how to apply the DRP rules to

the memory buses.

There are two scenarios. First, only the data bus carries *sensitive data*. Second, a special case happens when the processor tries to read an element from the SBox in the **SubByte** step. During this process, not only the data buses carry the sensitive **SubByte** result, the address buses are also sensitive since it is related to the **AddRoundKey**'s result. While the first case can easily be handled, special efforts are needed for the second case. To make sure the data buses carry complementary data, in $\overline{Program}$, complementary SBox elements should be stored in the complementary addresses. We define the SBox in the ordinary program, $Program$ and $\overline{Program}$ as $SBox$, $SBox_c$ and $\overline{SBox_c}$ respectively. Their relationship should be : $SBox(i) = SBox_c(i) = bitnot(\overline{SBox_c}(bitnot(i)))$. Moreover, the address buses should also be complementary. A possible problem is as follows. Suppose the $SBox_c$ and $\overline{SBox_c}$'s base addresses are both 0x0001. When $Program$ loads $SBox_c(0)$, $\overline{Program}$ loads $\overline{SBox_c}(255)$. The actual value on the address bus is the sum of the base address and the element's index, namely the offset address. In the above case, the values on the address buses are 0x0001 and 0x0100, which is obviously not complementary. Our solution is to align both $SBox_c$ and $\overline{SBox_c}$ to the 2^8 -byte boundary (SBox's base address is 0xXX00. XX means 'do not care' and can be different for $SBox_c$ and $\overline{SBox_c}$). In this case, the values on the address buses of two cores are 0xXX00 and 0xFF00. The sensitive part of the address buses are complementary.

We reserve a word which stores 0 in the memory. The 0 value is used to pre-charge the data buses. When the address buses contain sensitive data, the address of the reserved word should also be aligned to the 2^8 -byte boundary (e.g. r12=0xXX00 in Figure. 5.3). When accessing this reserved address, the lowest byte of the address bus is pre-charged to all 0. With the above preparation, the pre-charge operations reset the memory bus and the storage in the way shown in Figure. 5.7. Finally, if the value in the source register or memory is not used later, we reset it to 0.

With the above techniques, we were able to convert the full AES software into VSC protected AES.

5.4 VSC AES Prototype Resists SCA

In this section we construct a VSC AES prototype and demonstrate that VSC works in reality.

5.4.1 VSC AES Prototype

We use SASEBO-G board as our dual-core platform. The XC2VP30-5FG676C FPGA on board contains two identical embedded hard PowerPC cores. Based on that, we built a shared-memory dual-PowerPC processor. Other modules, such as the buses, memories, synchronization unit, and so on, are built with the FPGA resources. The design ran at 20MHz.

AES under test uses a 128-bit key (16 key bytes from `key[0]` to `key[15]`). We first implemented two version of AES: regular AES, and VSC AES, represented by `AES` and `VSC-AES` respectively. The implementation of `AES` was based on the standard AES algorithm description [6]. Based on that, we converted `AES` to `VSC-AES` by using the DRP technique to protect the three datapaths in the PowerPC processors. Some conversion technical details will be discussed in Section 5.3.

We use the analysis setup shown in Section 2.1.2 to test the SCA resistance of our dual-core based VSC implementation. We mount both correlation power attack (CPA) [62] and correlation electromagnetic attack (CEA). During each attack, the AES key remained unchanged. The `KeyGeneration` process only ran once. So our attack focused on the regular encryption operations.

All attacks focus on `SubByte`'s output in AES's first round. Since the attackers could not figure out the exact time when the sensitive data would appear before the analysis, the oscilloscope sampled the current trace of the entire first round. Moreover, to save the space of waveform and shorten the analysis time, the oscilloscope worked in the 'average' mode. Every current trace the computer obtained was the average of 32 normal traces (with the same plaintext). Hamming weight of the sensitive data is used

as the power model. Because of the pre-charge process, this power model is the best for VSC-AES. Hamming distance may work better for AES, but this requires the attackers to get access of the software, which cannot always be fulfilled. Therefore, Hamming weight is also chosen for AES. The improvement of VSC-AES over AES obtained based on this power model is a conservative one.

5.4.2 Results

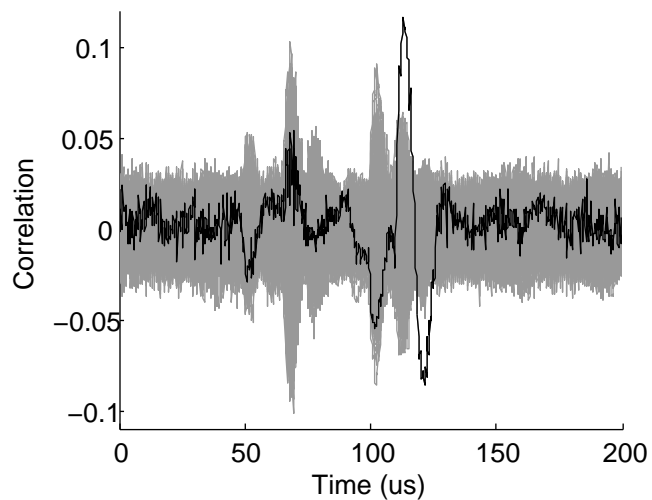
We discuss CPA and CEA separately here as we will see that their results are different.

Correlation Power Attack

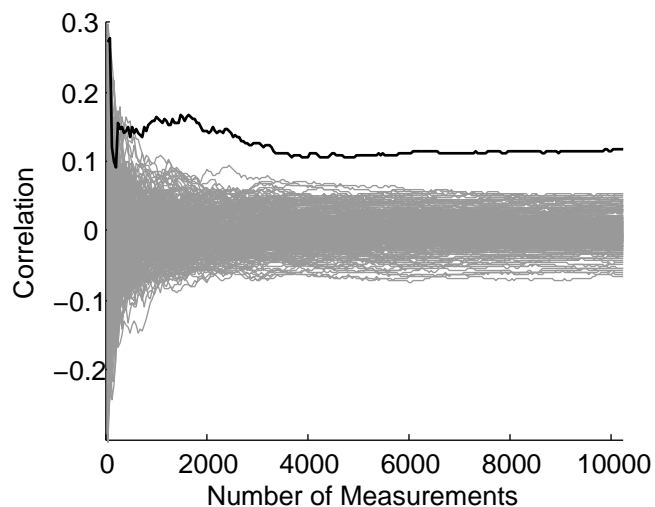
Figure 5.8 shows an example of the CPA results on one of AES's key bytes (key[3]). With only 256 averaged measurements, we were able to find the first correctly attacked key byte of AES. When the number of averaged measurements increased to 40960, all the 16 key bytes were revealed.

Figure 5.9 shows an example of the attack results on VSC-AES's key[3]. In contrast to AES, even with 40960 averaged measurements, none of the 16 key bytes were revealed. Only after the measurements number increased to 81920, the first uncovered key byte appeared. After increasing the averaged measurements number to 1024000, only 6 key bytes were attackable. In addition, we also used the Hamming weight of each bit of SubByte's output as the power model. It turned out that this power model worked better in our experiments. With the bit-based power model, attacks were able to find the first revealed key byte with around 20480 averaged measurements. Further, even with 1024000 averaged measurements (32 million regular measurements), 3 key bytes were still unattackable.

It is clear that, compared with AES, VSC-AES has obvious reduction of side-channel power leakage. We use the number of measurements to disclosure (MTD) to quantify the resistance against power attacks. Disclosure here means that at least one key byte

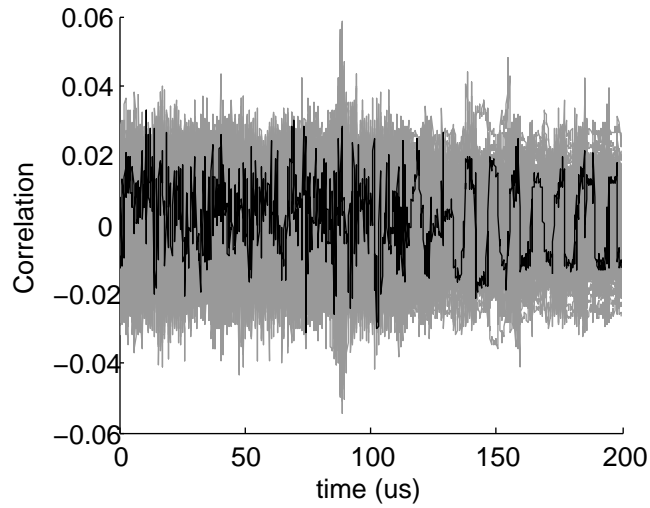


(a)

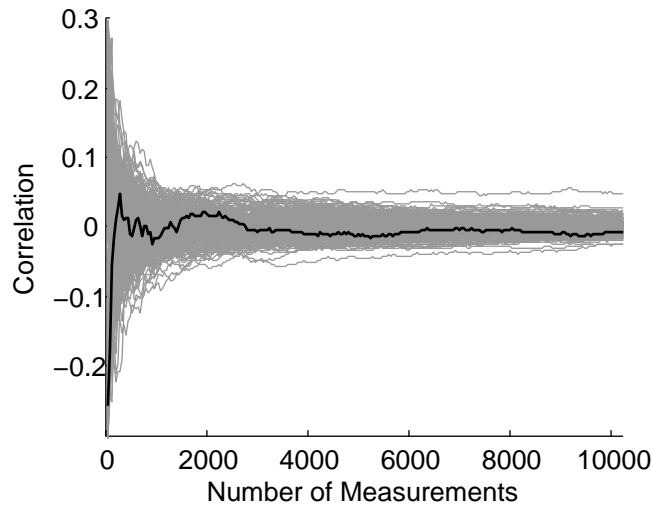


(b)

Figure 5.8: CPA result on unprotected AES on the dual-core prototype. (a) Correlation between the sampled current and the power estimations with 10240 averaged measurements; (b) Correlation between the sample current and the power estimations at the point where the attacked key is identified. *Correct key's trace is plotted in black, while all other key's traces are in gray. The emerged black trace means successful attack.*



(a)



(b)

Figure 5.9: CPA result on protected VSC-AES on the dual-core prototype. (a) Correlation between the sampled current and the power estimations with 10240 averaged measurements; (b) Correlation between the sample current and the power estimations at the point where the attacked key is identified. *Correct key's trace is plotted in black, while all other key's traces are in gray. The buried black trace means unsuccessful attack.*

is broken. Based on the *byte-based* Hamming weight model, VSC-AES's improvement on MTD over AES is 320 times. Based on the *bit-based* Hamming weight model, VSC-AES's improvement is 80 times. If we consider a successful power attack as discovering all key bytes, then attacks with over 32 million measurements were not able to succeed.

To further verify the correctness of VSC, we mounted attacks on variants of VSC-AES: 1) only the true path of VSC-AES without the complementary path (VSC-AES nocomp), 2) VSC-AES with the true path 1 clock cycle ahead of the complementary path (VSC-AES nosync), and 3) VSC-AES without pre-charge operations (VSC-AES noprch). All these designs could be broken much more easily than VSC-AES as shown in Table 5.1.

Table 5.1 summarizes the results of the above power attacks. VSC-AES pays 4.57 times of execution time and 6.21 times of footprint for a much higher capability of resisting power attacks. Also, complementary operations, pre-charge, and synchronization are demonstrated to be three indispensable conditions for protection.

In summary, through the experiments, VSC showed its effectiveness on protecting AES software. The side-channel resistance of VSC-AES is comparable to the resistance offered by the WDDL prototype IC chip [80]. Compared with the unprotected designs, their improvements in terms of MTD are both around 100.

Correlation Electromagnetic Attack

Besides CPA, we also mount CEA on the dual-core based VSC. Figure 5.10 shows an example of the CEA results on both AES and VSC-AES with 5120 averaged measurements. It turns out that CEA attacks are both successful. Table 5.2 gives more details on CEA results. It shows that VSC-AES does not offer improved resistance against CEA.

5.5 Discussion

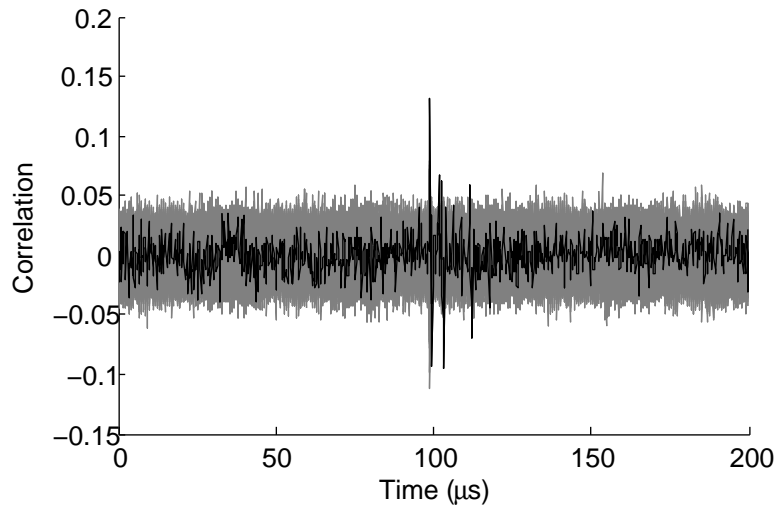
Based on the experimental results, we see that dual-core VSC supplies a good protection against CPA. The security improvement is around 80 times. When CEA is mounted,

Table 5.1: CPA attack results summary.

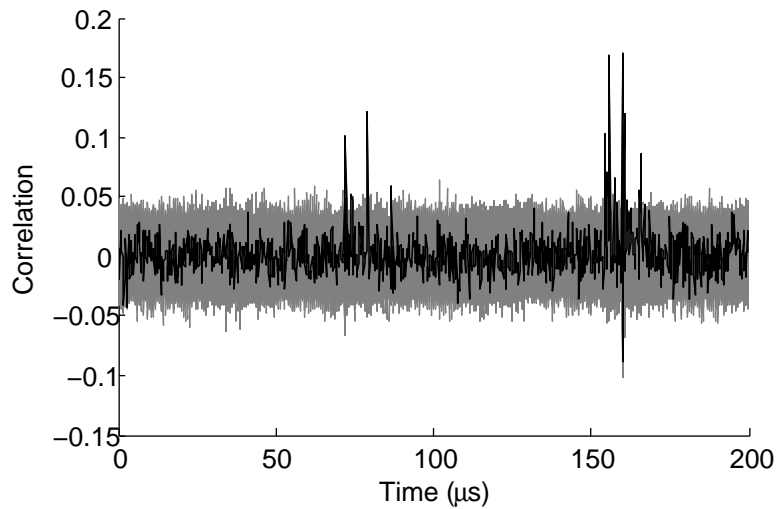
Parameter	AES	VSC-AES		VSC-AES nocomp	VSC-AES noprch	VSC-AES nosync
		byte-based	bit-based			
exec. time (μ s)	357	1630	—	—	—	—
footprint (kB)	1.9	5.9*2=11.8	—	—	—	—
MTD ¹	256	81920	20480	256	1204	1280
key bytes NOT found (CPA)						
@ 10240 ²	6	16	16	0	7	2
@ 40960 ²	0	16	14	0	6	1
@ 1024000 ²	0	10	3	0	0	0

¹ To find the first uncovered key byte.

² Number of averaged measurements: every averaged measurement is an average of 32 instant measurements with the same plaintext.



(a)



(b)

Figure 5.10: CEA results on unprotected AES and dual-core VSC AES. (a) unprotected AES (AES), (b) VSC-protected AES (VSC-AES) on the dual-core VSC prototype: Correlation between the sampled current and the power expectations with 5120 averaged measurements. The correct key's trace is plotted in black, while all other key's traces are in gray. The emerged black trace in both (a) and (b) means both attacks are successful.

Table 5.2: CEA attack results summary.

Parameter	AES	VSC-AES		VSC-AES	VSC-AES	VSC-AES
		byte-based	bit-based	nocomp	noprch	nosync
exec. time (μ s)	357	1630		—	—	—
footprint (kB)	1.9	5.9*2=11.8		—	—	—
# of key bytes NOT found (CEA)						
		0	8	0	0	0
@ 5120 ¹	3					
@ 10240 ¹	0	0	2	0	0	0
@ 40960 ¹	0	0	0	0	0	0
@ 1024000 ¹	0	0	0	0	0	0

¹ To find the first uncovered key byte.

² Number of averaged measurements: every averaged measurement is an average of 32 instant measurements with the same plaintext.

there is no improvement from dual-core VSC. Our explanation is that dual-core VSC has two cores too far away from each other. While a current probe always measures the current flowing into both cores, an H-field probe can easily pickup the signal mainly from only one core when located closer to it. Therefore, CEA can still break VSC on dual-core processors easily. Although dual-core VSC does not thwart CEA, in those cases where CEA is not a threat or is already prevented by other solutions, dual-core VSC is still an effective countermeasure.

5.6 Previous Works

A previous similar work, called MUTE-AES, was presented in 2008 [101]. Both VSC and MUTE-AES follow the idea of running the direct and complementary copies of a cryptographic algorithm on two identical cores to generate data-independent power. Despite their similarity, there are significant differences between our work and the previous work.

- VSC and MUTE-AES are at different level of abstraction. VSC deals with the processor instructions, while MUTE-AES is based on modifying AES algorithm. As a result, VSC is a WDDL-like general protection solution, while MUTE-AES only applies to the AES algorithm.
- Unlike VSC, MUTE-AES does not strictly follow the principles of the DRP technique. There is no pre-charge operations in MUTE-AES. Based on our experimental results in Table 7.1, without pre-charge (VSC-AES noprch), the crypto-system can still be easily attacked.
- We not only show VSC's improvement qualitatively but also quantify the improvement with real Side Channel Attacks. For MUTE-AES, the experiment was based on simulation and the improvement was not quantified. Hence, there is no way to see how much better security that MUTE-AES can supply.

- The method to perfectly synchronize two cores for MUTE-AES is to recognize the program pattern of the AES software's assembly codes. This method is potentially false positive, not to mention it is specific to a single program. In contrast, as shown in the Chapter 5.2, VSC's synchronization method is simpler and more generic and is not false-positive nor true-negative.

5.7 Conclusion

In this chapter, we have proposed VSC as a solution to protect software with the DRP technique. Following that, we use dual-core processors to implement that VSC concept. Analysis showed that a dual-core VSC was equivalent to a DRP circuit. It inherited the security features from the DRP circuits. To demonstrate this, a VSC protected full AES was implemented on a dual-PowerPC processor. Experiments showed that, when power attacks are mounted, the VSC protected AES software had a comparable security performance as the WDDL based AES IC prototype. However, when electromagnetic attacks are mounted, dual-core VSC does not supply any security improvement. In conclusion, we see the effectiveness of VSC. In circumstances where electromagnetic attacks are not a problem, dual-core VSC can be a good software protection solution. Otherwise, we still need to overcome the electromagnetic attack problem, which will be covered in the next two chapters.

Chapter 6

SCA-Resistant Software Using Instruction Set Extensions: Initial Solution

Chapter 5 tells us that VSC implemented on synchronized dual-core processors offers a good resistance to power attacks. However, due to the separation of two cores, electromagnetic radiation from each core can be picked up by EM probes. This requires designers to search for other approaches to preventing electromagnetic attacks. In this chapter, we introduce a solution that has the same security gain in front of both power attacks and electromagnetic attacks.

We still follow the VSC concept to realize SCA-resistance, but with only one single processor core. Considering regular single-core processors do not support execution of direct and complementary operations at the same time, we turn to instruction set extensions. By adding a set of special instructions, we enable the processors to behave the same as DRP circuits when executing these added instructions. We call these special instructions *balanced instructions* and the resulting processor *balanced processor*. In addition, since the direct datapath and its complementary counterpart are within the same processor, the distance between them is very close. It is expected that this solution

has similar security gain against both power attacks and electromagnetic attacks.

Similar to the dual-core VSC, this solution has an advantage of flexibility. Once the balanced processor is built up, the side-channel resistance is achieved completely in software. The second advantage is that the proposed solution is not limited to a specific cryptographic algorithm. The third advantage is the much lower hardware cost when compared with dedicated hardware DRP circuits. Our results show that hiding-based countermeasures can be applied using the existing ASIP technology. Hence, the proposed technique can be directly applied by ASIP users with a need for side-channel resistance in their designs.

The contributions of this chapter include 1) the concept of using a single balanced processor to support VSC, 2) a low-cost solution to implement a balanced processor and the demonstration of its security, 3) a secure programming method to implement a VSC, and 4) the demonstration and quantification of the effectiveness of our solution with real-world side channel attacks.

6.1 Virtual Secure Circuit on A Balanced Processor

The concept of the single-core VSC is illustrated in Figure 6.1. Similar to DRP circuits, the processor has two parts: one performs direct operations and the other performs complementary operations. Every direct operation in the first part has a complementary counterpart in the second part. A direct operation takes input in and generates output out . The complementary operation takes input \overline{in} and generates output \overline{out} . $\overline{in} = NOT(in)$ and $\overline{out} = NOT(out)$. NOT means the bit-wise inversion operation.

Each balanced instruction in the balanced processor chooses a pair of complementary operations from two parts of the processor and executes them at the same time. The cryptographic algorithm is programmed with this set of balanced instructions. Therefore, the cryptographic algorithm has both a direct execution path and a complementary path. To run the cryptographic algorithm, both direct and complementary plaintext (balanced

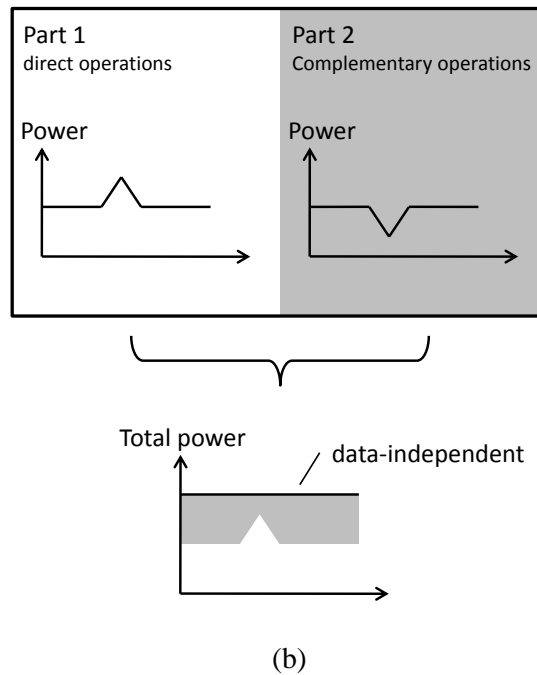
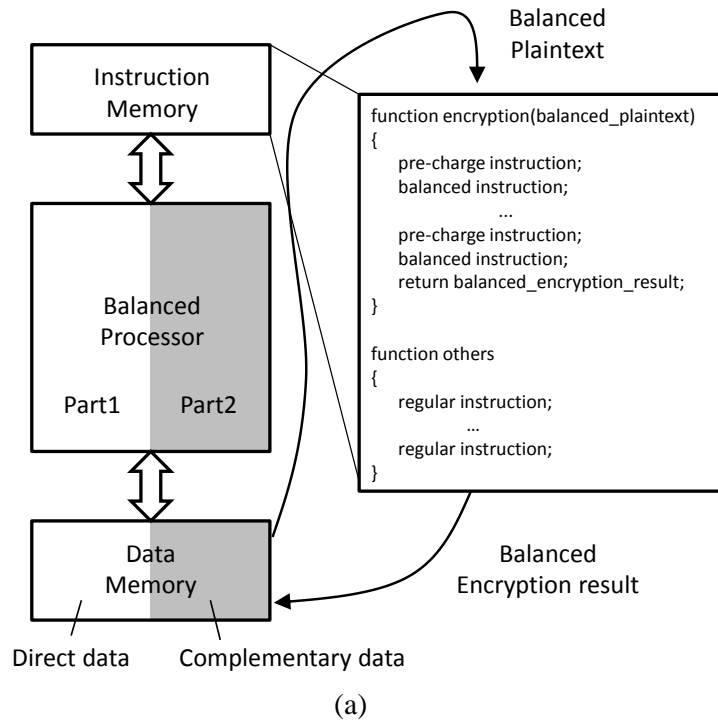


Figure 6.1: Concept of balanced processor and VSC programming. (a) Concept of the proposed solution: balanced processor and VSC programming; (b) Power dissipation from the new balanced processor system does not reveal the processed data.

input) are supplied to these two paths. Finally, the algorithm generates both the direct and the complementary encryption results (balanced outputs), shown in Figure 6.1a.

Besides the balanced instructions, the processor also has another set of instructions that perform the pre-charge operations in the same way as the DRP circuits. Before executing balanced instructions, a set of pre-charge instructions first clear the execution datapath. Following that, the balanced instruction finishes the calculation. Both the balanced instructions and pre-charge instructions can be added to a general-purpose processor as instruction set extensions.

With the above concept, we obtain another software counterpart of DRP circuit. For a balanced instruction, similar to the DRP gate shown in Figure 2.4, the power dissipation from the direct operation always has a complementary counterpart from the complementary operation. The sum of these two is a constant, shown in Figure 6.1b. Suppose the output of the balanced instructions contains the direct and complementary forms of the intermediate value V mentioned in Section 2.1.1, attackers are not able to go from the power of the balanced instructions to the value of V . Therefore, V is protected. If every intermediate value in the cryptographic algorithm is processed by the balanced instructions, then the entire algorithm is secured.

6.1.1 Implementation

The next step is to map the above concept to the processor implementation. This includes the implementation of the balanced instructions, the pre-charge instructions and the secure programming style that leads to DRP behavior.

Balanced Instructions

The instruction set architecture of a processor includes different instructions. The logic function of the software is realized with logic instructions, such as AND, OR, and NOT. In theory, any algorithm can be realized with these three instructions. For higher efficiency, processors also implement the arithmetic instructions that are frequently

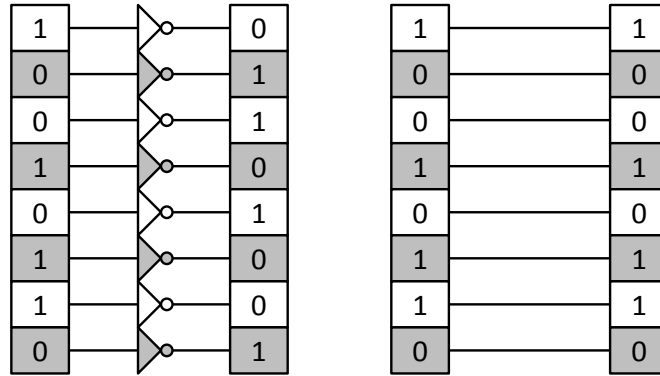
used, such as ADD, SUBTRACT, and MULTIPLY. In addition to that, due to the processor architecture, shift instructions and data instructions, such as MOVE, LOAD, and STORE, also appear in the instruction set architecture. Finally, the processor also provides the control instructions, such as JUMP and conditional JUMP.

As mentioned above, each balanced instruction executes a pair of complementary operations. To make the balanced datapath compatible with the regular datapath, the width of either the direct or the complementary datapaths is designed to be half of the regular datapath's width. Thus, the entire datapath of a balanced instruction is as wide as the regular instructions.

The integration of balanced instructions follows the rule that balanced instructions take balanced inputs and generate balanced outputs. The proposed solution chooses balanced instructions according to the existing regular instruction set architecture. This enables the design flow of the VSC software to reuse the existing compiler for the regular instructions. The details are as follows.

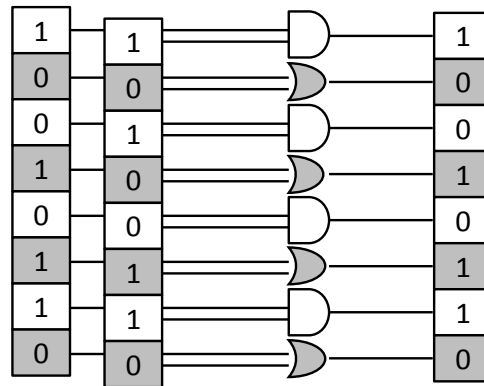
Among the logic operations, NOT is the complementary instruction for itself. This means that if half of the input is direct value and the other half is complementary value, the output of NOT is still a pair of complementary values. Therefore, the regular NOT instruction can also be used as the balanced NOT instruction. The AND operation has OR operation as complementary counterpart and vice versa. Therefore, the balanced AND instruction (b_and) should be half AND for the direct input and half OR for the complementary input. Similarly, the balanced OR instruction (b_or) should be half OR and half AND, shown in Figure 6.2. With balanced AND, OR, and NOT, any complex logic function can be realized.

Shift and data instructions are similar. They both move the programs' intermediate values from one storage location to another. Since 'move' operations do not change the values of the operands, their complementary counterparts are themselves. Similar to NOT instruction, shift and data instructions are shared by both balanced and regular instructions, shown in Figure 6.2.



(a) `inv r1, r2`

(b) `mov r1, r2`



(c) `b_and r1, r2, r3`

Figure 6.2: Examples of balanced instructions. (a) Regular INV can be used as a balanced INV; (b) Regular MOV can be used as a balanced MOV; (c) Balanced AND instruction uses half AND operators and half OR operators.

Designing balanced arithmetic instructions is doable but may have a higher increase of hardware resources. This is because we need to first convert the regular arithmetic modules to basic AND, OR, and NOT gates and then convert them to DRP gates. Such a conversion brings us an inefficient synthesis. However, if we compare such an area increase with the area of a processor, this increase is still quite small. If the algorithm uses arithmetic instructions, corresponding balanced arithmetic instructions may be needed.

Control-flow related instructions, e.g. BRANCH, do not leak useful side-channel in-

formation to DPA and EMA, as long as the control flow is not related to both plaintext and key. For example, symmetric-key cryptography usually has fixed control flow. The branch condition is irrelevant to either the plaintext or the secret key. Public-key cryptography's control flow is only related to the key but not the plaintext. While this may enable SPA, this problem can easily be solved by other methods, for example by using the Montgomery Ladder [102].

In summary, a minimum solution only needs to add two balanced instructions (balanced AND and balanced OR). NOT, shift, and data instructions can be shared between regular and balanced instructions. This makes the modification to the processor very tiny.

Pre-charge Instructions

As mentioned in Section 2.1.3, besides complementary datapath, DRP technique also requires pre-charge operations. The proposed solution also faces the problem on how to implement pre-charge instructions for the same purpose. A pre-charge operation needs to pre-charge not only the result storage but also the computational datapath. For a balanced processor, a pre-charge instruction needs to pre-charge the processor's datapath and the destination registers or memory locations.

In the DRP circuits, pre-charge is usually done by pre-charging the inputs. The proposed solution uses a similar way to realize the pre-charge operations. It turns out that by setting the input operands of different balanced instructions to 0, the corresponding outputs are either all 0 or all 1 (for NOT). Since all 0 and all 1 are both acceptable pre-charge results, the pre-charge operation of a balanced instruction can be done by just executing the same balanced instruction with all 0 inputs.

In summary, there is no need to add additional dedicated pre-charge instructions. Every balanced instruction can finish the pre-charge operation for itself by taking all 0 as the inputs.

6.1.2 Security Analysis


As discussed in Section 5.1.4, a processor has three different datapaths that may dissipate data-dependent power. If we can protect each of these three datapaths, the microprocessor instructions that only make use of these datapaths will also be secure.

Similarly, we demonstrate that a single-core VSC is functionally equal to a DRP circuit. Figure 6.3 shows an example of a regular program (Figure 6.3a), a VSC program (Figure 6.3b) and a DRP circuit (Figure 6.3c). All of them accomplish the same function. In the VSC program, bold instructions act as balanced instructions. Other instructions before each of them are used for the pre-charge purpose. 'b_and' and 'b_or' are balanced AND and OR instructions.

Two successive pre-charge and balanced instructions correspond to the pre-charge and evaluation activities of a logic gate (vector) in the circuit in Figure 6.3c. The numbers annotated within the logic gates correspond to line numbers of the balanced instructions in the VSC program. In between each logic gate, a register is inserted. They are equivalent to the registers in the processor's register file. The pre-charge instruction will reset that register before loading it with sensitive intermediate values. In addition, the direct and complementary operations are activated by the same balanced instruction at the same time. Therefore, when executing a pre-charge instruction followed by a balanced instruction, the circuits inside the processor behave exactly the same as a part of DRP circuit in Figure 6.3c. We conclude that such a single-core VSC is a sequentialized version of a DRP circuit. Moreover, single-core VSC also inherits the properties of DRP circuits.

6.2 VSC Programming

This section presents a method for VSC programming on the balanced processors so that SCA can be thwarted. Compared with the regular programming, VSC programming has two basic constraints. First, only the balanced instructions can be used to process

<pre> ; regular program 1. not %r1,%r6 2. not %r2,%r7 3. and %r6,%r2,%r3 4. and %r7,%r1,%r4 5. or %r3,%r4,%r1 </pre>		<pre> ; VSC program ; %r0 = 0 1. not %r0,%r6 2. not %r1,%r6 3. not %r0,%r7 4. not %r2,%r7 5. b_and %r0,%r0,%r3 6. b_and %r6,%r2,%r3 7. b_and %r0,%r0,%r4 8. b_and %r7,%r1,%r4 9. b_or %r0,%r0,%r1 10. b_or %r3,%r4,%r1 </pre>
<p>for pre-charge →</p> <p>for computation →</p>		
(a)		(b)

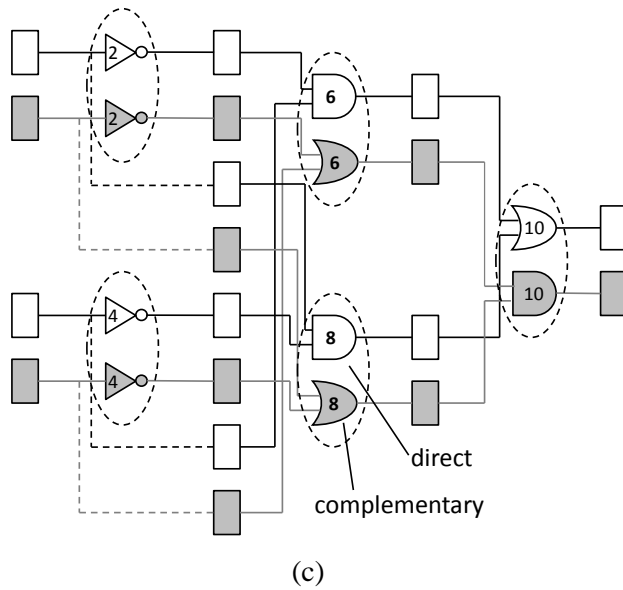


Figure 6.3: VSC program conversion and corresponding equivalent DRP circuit. (a) Regular program of Leon3 [103] assembly code; (b) VSC program of Leon3 assembly code; (c) Equivalent DRP circuit.

sensitive data. Second, the direct or the complementary datapath can only take half of the datapath of the processor. There could be different methods to deal with these two constraints. Here, we show a special solution based on bitslice programming.

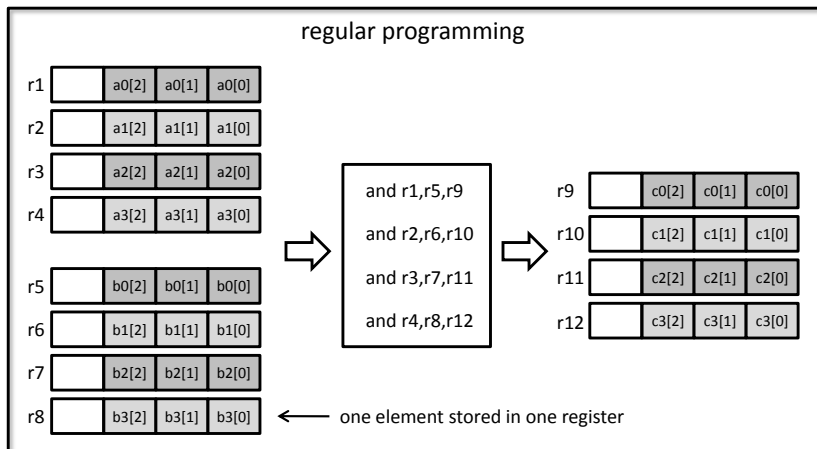
6.2.1 Bitslice Programming

Bitslice programming is a special programming method. The idea is to break the software applications into only logic operations at the bit level, such as AND, OR, NOT, and XOR. The software's logic function is realized bit by bit. This is similar to the hardware implementation, with every operation equivalent to one logic gate. Porting this idea to a n -bit processor results in n application instances running in parallel. Figure 6.4 shows a simple example on bitslice programming.

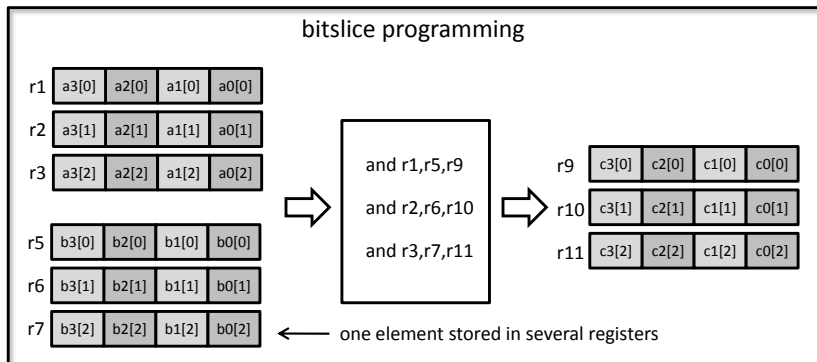
Suppose the application is AND operation and runs on a 4-bit processor. We use this application to process two arrays A: (a_0, a_1, a_2, a_3) and B: (b_0, b_1, b_2, b_3) and generate the result C: (c_0, c_1, c_2, c_3) . $c_i = a_i \text{ AND } b_i$. Every array element is 3-bit long, for example a_0 can be represented as $(a_0[2], a_0[1], a_0[0])$. Figure 6.4a shows the process based on regular programming. Every element is stored into a processor register. AND operations are used to process the elements, with each operation generating one element of C. In contrast, bitslice programming views the problem in a different way. Each array element is distributed over different registers. The operation on each array element is broken down to the bit level. Therefore, instead of 1 AND instruction, 3 AND instructions are needed to generate one element of C array. However, since the processor has a 4-bit wide datapath, another 3 exactly the same application instances can be done in parallel. Hence, the operations in Figure 6.4a can also be finished in Figure 6.4b.

Bitslice programming guarantees that the processor's datapath is under full usage, which may make it more efficient. However, it also has limitations. First, bitslice programming only uses logic instructions for computation. So for software applications that use a lot of arithmetic operations, bitslice programming loses efficiency. Second, running n identical application instances in parallel generates n times of intermediate results. These results need to be moved out to the memory in case the processor's registers are not enough. This will also reduce the efficiency.

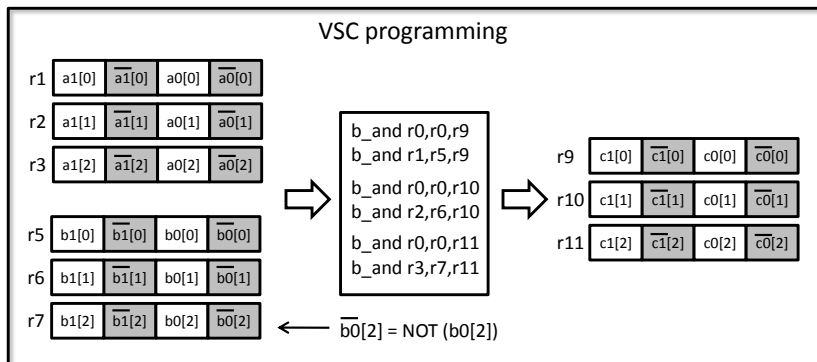
Even though it has the above limitations, bitslice programming for symmetric-key cryptography is still popular. By now, we have seen bitslice programming under inten-



(a)



(b)



(c)

Figure 6.4: Comparing regular, bitsliced, and VSC programming. Processing arrays A: (a_0, a_1, a_3, a_4) and B: (b_0, b_1, b_2, b_3) with AND operations to generate C: (c_0, c_1, c_2, c_3). Each array element is 3-bit long while the processor's datapath is 4-bit long. (a) regular programming; (b) bitslice programming; (c) VSC programming.

sive research on two most important symmetric-key algorithms, including DES [104] and AES [105, 106, 107]. The reason is that symmetric-key cryptography seldom uses arithmetic operations, which makes the first constraint not a problem. Similarly, public-key cryptography that mainly uses logic operations can also be implemented in a bitsliced way. For example, Batch Binary Edwards [108] bitslices Edwards-Curve-based ECC and is able to achieve high performance.

To deal with the second constraint, improved bitslice schemes have been proposed. The main idea is to reduce the number of encryption instances running in parallel, while maintaining a full usage of the processor's datapath. In this case, instead of taking only 1 bit of the register, one encryption instance takes multiple bits. Similar intermediate bits in one encryption instance are organized in one register. The operations on them are usually the same, even though not always identical as in the basic idea of bitslice programming. Once the operations for different bits in the same register are different, shift operations are employed to separate them for different operations and join them together after that. This sacrifices some computational efficiency but reduces data movements between the registers and the memory. The bitslice programming for AES that we will use for demonstration follows this method.

6.2.2 VSC Programming Based on Bitslice

Bit-slicing programming is a very suitable starting point for VSC programming. First, bitslice programming only uses bit-level logic instructions for computation. These instructions can easily find complementary counterparts. Second, bitslice programming can easily meet the requirement that direct operation only takes half of the processor's datapath. This can be done by simply changing the number of instances. Therefore, it is very easy to convert the bitslice programming to VSC programming.

Figure 6.4c shows how to convert the example depicted in Figure 6.4b to a VSC program. Instead of running 4 direct application instances together, we maintain 2 instances as direct and change another 2 to the complementary instances. This easily

fits into the balanced processor. In detail, doing VSC programming based on bitslice programming can simply go with the following two steps.

- Do bitslice programming for the cryptographic application. Implement even number of application instances in parallel. Make sure that each application instance either falls into the direct part or the complementary part of the balanced processor. This guarantees that, after converted a VSC, the application instance either remains as a direct instance or is completely converted to a complementary instance.
- Convert the regular instructions to balanced instructions. Add the correlated pre-charge operation before each balanced instruction.

To demonstrate the effectiveness of single-core VSC, we chose the AES algorithm as the attack target. The bitslice AES was designed according to the method presented in [106]. Following the above two steps, we obtained a VSC AES on a 32-bit processor. Every AES instance takes 16 bits of the processor's datapath. So in VSC AES, one direct AES and one complementary AES run in parallel. One detail about VSC AES is with the XOR instruction. Since the complementary counterpart of XOR, XNOR, cannot be pre-charged by simply applying 0 on the inputs, we expand XOR to AND, OR, and NOT instructions. Actually, the logic operation shown in Figure 6.3 is a XOR operation that has already been expanded.

From the existing efficient designs of bitslice symmetric-key cryptography, we can see that the mainstream symmetric-key cryptography algorithms can all be easily converted to VSC. So although the current version of VSC may not support the software with arithmetic operations well, it can still be applied to a great amount of security systems.

6.3 Experimental Results

To demonstrate that VSC on the complementary processor improves the resistance against SCA, this section presents the experimental results based on real attacks.

6.3.1 Experimental Setup

We used a Leon3 embedded processor [103] to build a balanced processor prototype. Both the regular bitsliced AES and the VSC bitsliced AES discussed in Section 6.2 were chosen as the attack targets. To convert Leon3 to a balanced processor, we replaced the operations of two existing instructions (ANDN and ORN) with complementary AND and OR operations. Since ANDN and ORN instructions are not used by the bitsliced AES, this does not influence the correctness of the software and helps to quickly build up the prototype.

The prototype was realized on a FPGA board with a Xilinx Spartan 3E-1600 FPGA. The balanced Leon3 processor was implemented with FPGA resources, clocked at 50 MHz, while the software applications under test were stored in the on-board DDR2 SRAM memory. An AES encryption key was stored in the DDR2 SRAM and could not be read out of the board.

The measurement setup has already been described in Section 2.1.2. The only difference is that we use both a current probe (Tektronix CT-2) and a H-field probe (ETS-LINDREN 903) to measure the current and electromagnetic radiation respectively, shown in Figure 6.5. To reduce noise sampled by probes, every sampled trace transferred to the PC is an average trace of 32 normal traces (with the same plaintext).

6.3.2 Results

We mount the same CPA and CEA to single-core VSC as to dual-core VSC in the previous chapter. Attacks focus on the same place: the outputs of the `SubBytes` step (16 bytes) in the first round of the AES algorithm.

Figure 6.6 and Figure 6.7 show some examples of the CPA and CEA results with 5120 averaged measurements. The x-axis represents time that each measurement covers. The first round execution of AES occurs within this time range. The y-axis represents correlation coefficient, the attack result. The black traces are the correlation coefficient traces corresponding to the correct key byte value. The gray traces correspond to other

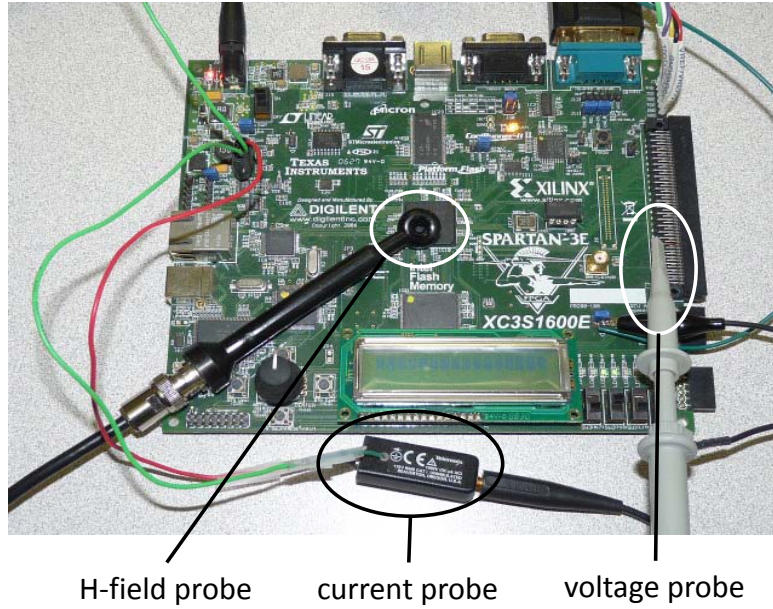


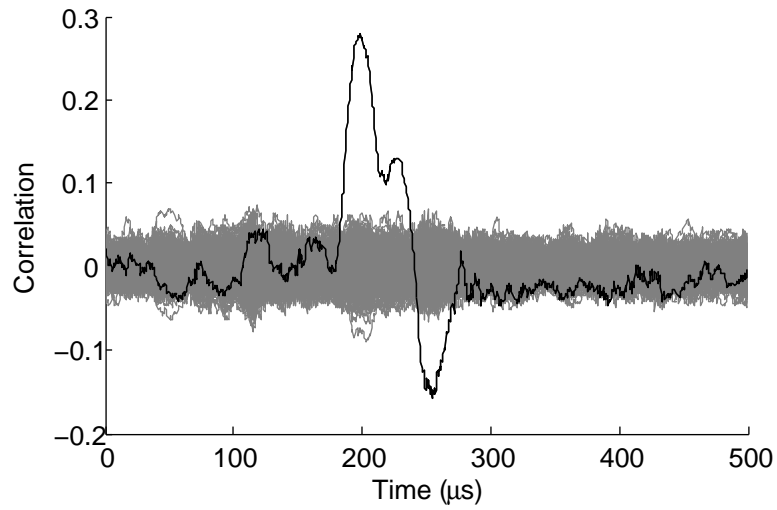
Figure 6.5: Experimental setup for single-core VSC. Photo was taken by the author in 2011.

incorrect key byte values.

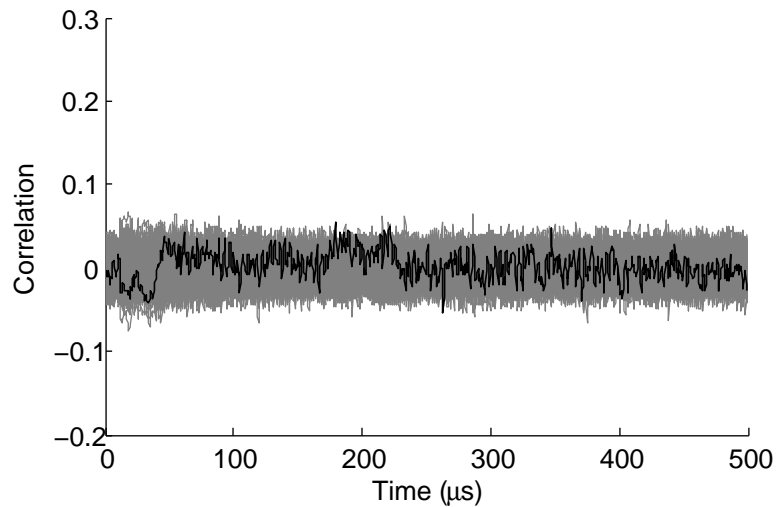
The black traces can be easily differentiated from others in Figure 6.6a and Figure 6.7a. This means that both CPA and CEA can attack the the unprotected AES (AES) successfully. In contrast, when attacking the VSC-AES implementation, the black traces are both buried into other gray traces in Figure 6.6b and Figure 6.7b. There is no way to identify them from other traces. Therefore, both CPA and CEA are unsuccessful.

Table 6.1 presents more detailed attack results. 1280 averaged measurements are sufficient for both CPA and CEA to uncover all 16 key bytes of the unprotected AES. To fully attack VSC-AES with the byte-based model, the number of averaged measurements increases to 25600 and 51200 for CPA and CES respectively. When the bit-based model is used, CPA finds it harder to get a full attack while CEA decreases the MTD of full attack to 25600 averaged measurements. Considering the most power attack cases, VSC-AES gains security improvements as high as 20 times in front of both CPA and CEA.

This work also tested CPA attacks on variants of VSC-AES, including VSC-AES `nocomp`, and VSC-AES `noprch`. Both designs could be broken with a similar effort to break AES,

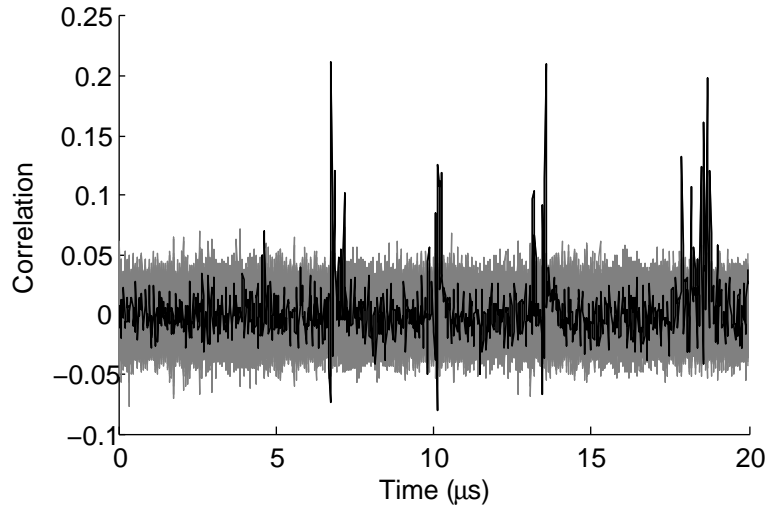


(a)

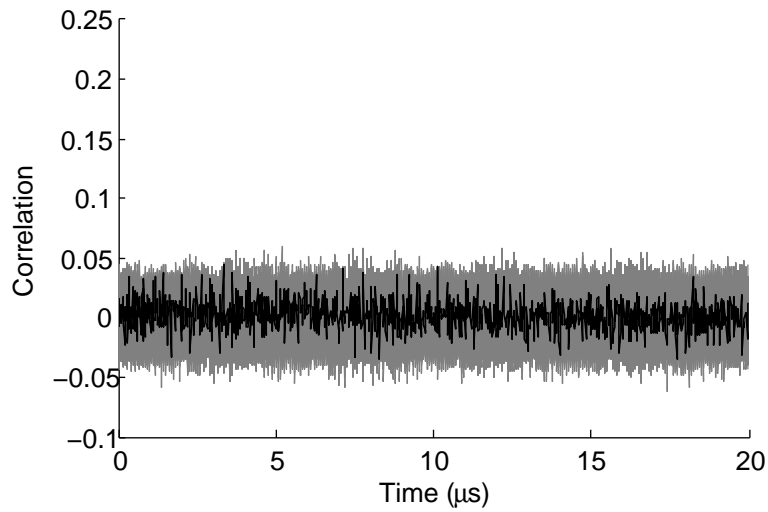


(b)

Figure 6.6: CPA results on unprotected AES and single-core VSC AES. (a) unprotected AES (AES), (b) VSC-protected AES (VSC-AES). Correlation between the sampled current and the power expectations with 5120 averaged measurements. The correct key's trace is plotted in black, while all other key's traces are in gray. The emerged black trace in (a) means a successful attack, while the buried black trace in (b) represents a unsuccessful attack.



(a)



(b)

Figure 6.7: CEA results on unprotected AES and single-core VSC AES. (a) unprotected AES (AES), (b) VSC-protected AES (VSC-AES). Correlation between the sampled current and the power expectations with 5120 averaged measurements. The correct key's trace is plotted in black, while all other key's traces are in gray. The emerged black trace in (a) means a successful attack, while the buried black trace in (b) represents a unsuccessful attack.

as shown in Table 6.1. This means that the design techniques that are integrated into single-core VSC are essential.

Table 6.1 also shows that VSC-AES pays 6.5 times decrease of throughput and 3.3 times increase of footprint to achieve SCA-resistance. The energy cost is estimated to be 6.5 times since the power remains more or less the same. The authors unrolled 10 rounds of the AES implementation. So the footprints for both AES and VSC-AES are relatively large. However, it is reasonable to expect both footprints to decrease by 10 times if the unrolling were not done.

To quantify the security improvement, we use *Measurement to Disclosure* (MTD), which means the number of measurements required for a successful attack. Larger MTD indicates better security. Here, we define the successful attack to be uncovering all 16 bytes of the AES key. Table 6.1 presents more detailed attack results. For AES, 1280 averaged measurements are sufficient to uncover all 16 key bytes. For VSC-AES, the number of averaged measurements for full attack increases to 25600. The improvement is 20 times. These results were obtained from the power expectation which is the Hamming weight of the entire byte of the *SubBytes*'s output. We name this as *byte-based* Hamming weight model. We also tested the power expectation which is the Hamming weight of each bit of the *SubBytes*'s output, named *bit-based* Hamming weight model. In our experiments, byte-based model is more efficient.

6.3.3 Analysis Based on Results

The experimental results shown above demonstrate that single-core VSC on the complementary processors presents an effective way to improve cryptographic software's resistance against SCA. Compared with the unprotected design, VSC-AES provides 20 times improvement.

Single-core VSC did not spend much effort to the structural symmetry between direct and complementary datapaths, although they are located in adjacent bits (Figure 6.2). Due to this reason, it is reasonable to see a smaller improvement than the WDDL

Table 6.1: Attack results summary for single-core VSC.

Parameter	AES	VSC-AES		VSC-AES nocomp	VSC-AES noprch
		byte-based	bit-based		
throughput (kb/s)	207*2=414	64	—	—	—
footprint (kB)	45.7	150.2	—	—	—
# of key bytes NOT found (CPA/CEA)					
	@ 512 ¹	16/16	16/15	4/9	7/9
	@ 1280 ¹	15/16	15/13	0/1	0/3
	@ 2560 ¹	12/13	13/9	0/0	0/0
	@ 5120 ¹	9/13	10/7	0/0	0/0
	@ 12800 ¹	4/7	5/1	0/0	0/0
	@ 25600 ¹	0/6	2/0	0/0	0/0
	@ 51200 ¹	0/0	2/0	0/0	0/0

¹ Number of averaged measurements: every averaged measurement is an average of 32 instant measurements with the same plaintext.

prototype IC [80].

Another advantage of single-core VSC is the low hardware cost. The only modification made to the hardware is two simple balanced instructions. In contrast, DRP circuits is usually from 3.4 times [38] to 12 times [42] larger than the regular circuits. This is a critical issue that prevents their broad acceptance by the industry. What single-core VSC pays is not the hardware cost, but the performance and the memory storage. Considering the memory storage is becoming cheaper and cheaper and is reusable, if the reduced performance is still acceptable to the applications, single-core VSC sees a promising future.

6.4 Conclusion

This chapter presents a new software countermeasure against Side Channel Attacks. The new countermeasure is based on the concept of VSC and implement VSC with instruction set extensions on one single core. Due to this, the resulting balanced processor system has similar power dissipation as the secure circuits and therefore can resist Side Channel Attacks as the secure circuits do. To support the balanced processor, this chapter also presents a method to do VSC programming. Real attacks showed that single-core VSC improves the security by 20 times. Although this security gain is not that high, single-core VSC has an advantage of high flexibility and low requirement for additional hardware cost. Unlike dual-core VSC, single-core VSC provides similar resistance against both power and electromagnetic attacks.

Chapter 7

SCA-Resistant Software Using Instruction Set Extensions: Improved Solution

The initial solution of implementing single-core VSC with instruction set extensions obtains 20-time security improvement. Compared with other countermeasures, this improvement is not high, and probably not satisfying in practice. The main reason for this is that the initial solution does not achieve identical placement and routing for the direct and complementary paths. In this chapter, we introduce another method to implement the same VSC concept also with instruction set extensions. This method allows us to integrate identical placement and routing to the processor implementation. As we will show later, the security gain is improved from 20 times to more than 1000 times.

Compared with the initial solution in Chapter 6, the improved solution here makes two changes with the processor microarchitecture. First, instead of integrating the instruction set extensions to the existing processor pipeline, we introduce a dedicated pipeline to implement all the secure (balanced) instruction set extensions. Second, the secure instruction set extensions have their own register file and local memory. This means that sensitive data does not share any storage with non-sensitive data. Separating

secure instructions and sensitive data from regular instructions and non-sensitive data makes it easy to apply special design rules to secure instructions, e.g. identical placement and routing between direct and complementary paths. It also enhances the logical security. For example, the improved solution enables to run multiple threads at the same time - only authorized threads have the access to the secure pipeline and its storage. As will be shown in Section 7.2, the improved solution has a much higher SCA-resistance. Although the improved solution has a little higher hardware costs (30% increase), we consider the area increase is still acceptable.

The improved solution presented in this chapter is the final solution in this dissertation. So besides the solution itself, we also present a comparison between this solution and other software protection solutions. We show that our improved solution is a new HW/SW-based solution. Among the HW/SW-based solutions [101], our improved solution is the only one that can deliver high SCA-resistance. Purely hardware-based countermeasures usually suffer from their high hardware cost. The SCA-resistant processor presented by Tillich et al. is at least 3 times as large as a regular processor [48]. On the other hand, software-based solutions usually have high design cost and poor performance. From [51] to [55], it has taken researchers one decade to find proper software-based solutions just for one algorithm: AES. The solution based on 3rd-order masking presented by Rivain et al. decreases the performance by 234 times over the unprotected solution [55]. In contrast, our solution is independent on any specific algorithm. We implemented our countermeasure based on an FPGA-based OpenSparc Leon-3 processor [103]. The processor area is enlarged by only 30%. The performance is decreased by 6.5 times. With 1000 times more measurements, real-world power and electromagnetic attacks are still far away from achieving the same level of success as on an unprotected solution. Based on these comparisons, we conclude that our improved solution not only delivers high SCA-resistance but also performs well in terms of cost and performance, without obvious disadvantages in any aspect.

7.1 An Improved Protection Solution

Similar to the initial solution, the improved solution consists of a secure processor and a corresponding programming method. The secure processor includes a secure pipeline with all the secure instruction set extensions based on VSC implemented in it. In such a way, execution within such a pipeline has data-independent power consumption and electromagnetic radiation. The same as the initial solution in Chapter 6, these instructions implement the dual-rail part (DR) of the DRP technique. An associated programming method helps and guides the software designers to program crypto-algorithms with secure instructions. The programming method implements the pre-charge part (P) of the DRP technique, and makes sure that the use of the secure instructions results in the desired DRP behavior.

The secure processor pipeline and associated programming method are systematically integrated together. Thanks to the secure pipeline hardware, software programmers do not have to worry about the low-level hardware behavior. Instead, they are only responsible for programming cryptographic algorithms with secure instructions. In other words, low-level hardware issues can be isolated from software by means of the secure instructions. Furthermore, the definition of a secure instruction-set results in a cost-effective hardware design, and it avoids complex application-specific hardware solutions.

7.1.1 Secure Processor

Processor Micro-Architecture

We propose a micro-architecture, shown in Figure 7.1. The processor has two pipelines. One for regular processing, called regular pipeline; the other for cryptographic processing, called secure pipeline. The secure pipeline is based on the Virtual Secure Circuit (VSC) concept which is also used in Chapter 5 and Chapter 6. VSC divides the secure pipeline into two parts. When running secure instructions, side-channel leakage from one part is always complementary to the leakage from the other. This way, the

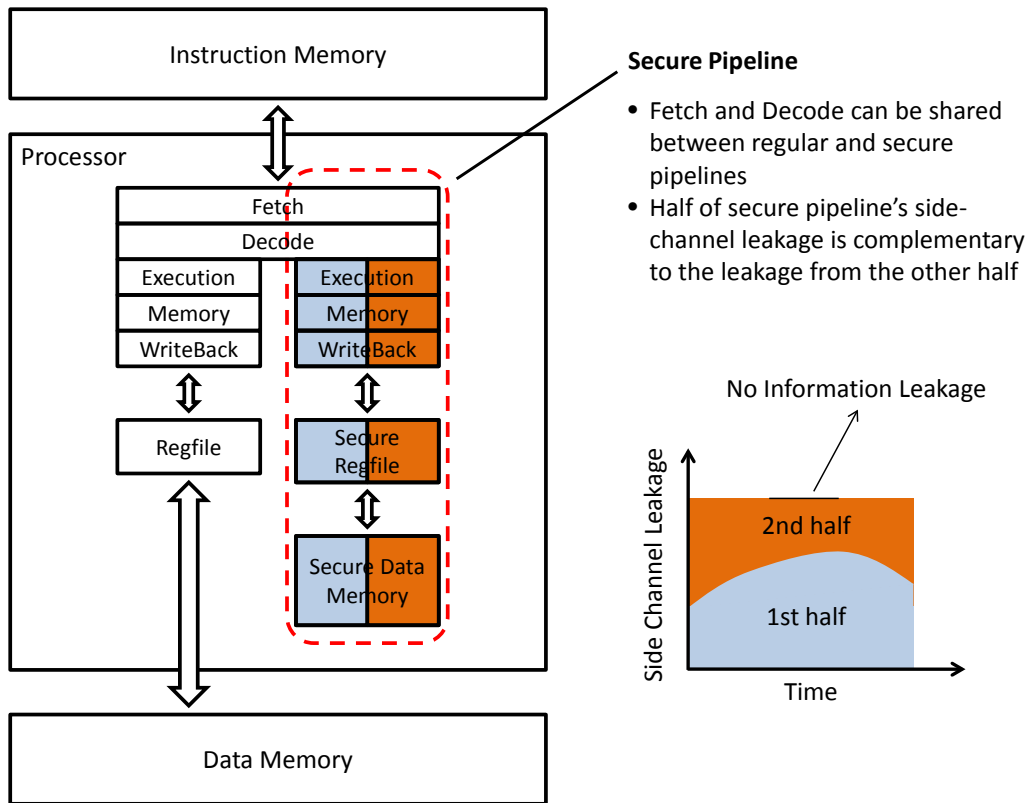


Figure 7.1: Secure processor micro-architecture.

secure pipeline has a data-independent power consumption and electromagnetic radiation. Each pipeline has its own register file and data memory. Intermediate sensitive values of cryptographic software are always kept within the secure pipeline. As a result, no useful side-channel leakage is leaked for power and electromagnetic attacks. Since the instruction opcodes are not dependent on operand values, power consumption of instruction fetching and decoding is of no use to physical attacks. So these two stages can be shared between regular pipeline and secure pipeline.

Our processor efficiently integrates a secure logic style into its micro-architecture. The arithmetic logic unit (ALU), the register file, and the memory, are all protected with a single and consistent countermeasure. Unlike the regular pipeline, the secure pipeline only implements instructions that are commonly used by crypto-algorithms. Its secure memory is also designed to be just big enough for crypto-algorithms. This is an

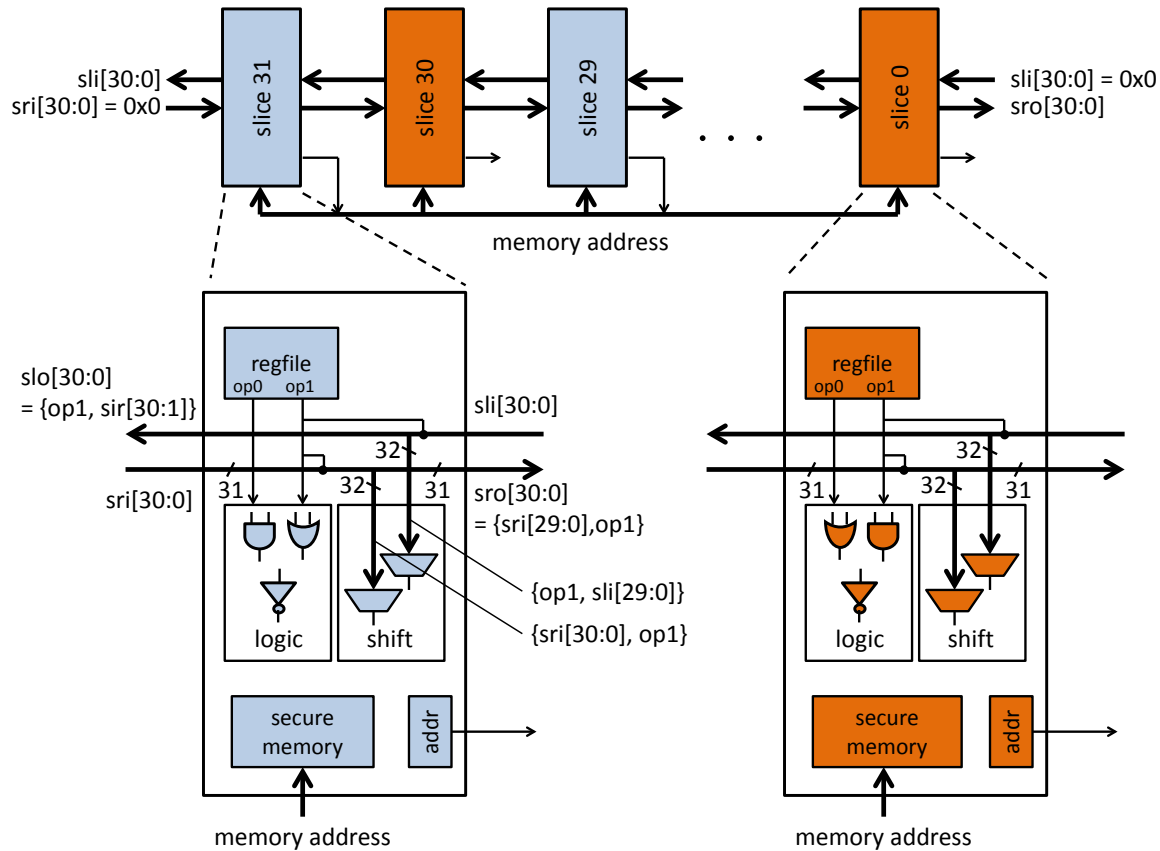


Figure 7.2: A basic slice-based secure pipeline.

important difference with previous secure processors [48, 109], which tried to support any software and any application. We will show that this incurs a significant extra hardware cost.

Processor Implementation

The DRP technique requires identical routing between direct and complementary paths. This requirement also holds true for the proposed secure pipeline. For this purpose, we introduce a slice-based pipeline, shown in Figure 7.2. The idea is to divide an n -bit pipeline into n slices with each slice processing one bit. By putting n such slices together in parallel, we obtain an n -bit pipeline. In an n -bit secure pipeline, $n/2$ slices implement the direct logic while the other $n/2$ implement the complementary

logic. For each direct-logic slice, there is a corresponding complementary-logic slice. The difference between these two slices is the complementary logic gates in ALU. Their storage parts are identical. To obtain identical routing between these two slices, we first implement the direct-logic slice. We then generate the complementary-logic slice by replacing the functional gates with the corresponding complementary gates while keeping the placement and routing the same.

Figure 7.2 illustrates such a pipeline with a basic instruction set that contains AND, OR, NOT, MOV, SHIFT, and memory access operations. AND, OR, and NOT operations' inputs and outputs always stay in the same slice. They can easily be implemented. The SHIFT operation takes inputs from other slices and produces data for the local slice. So we use two buses in opposite directions across the pipeline to transfer data among different slices. For each slice, the input could come from n different slices in an n -bit pipeline, including itself. Therefore, the SHIFT operation is a multiplexing process. Memory access operations of the secure pipeline need special treatment, since the address coming out of the pipeline contains both direct and complementary representations. We only use the direct part as the address. As a result, the memory address cannot carry sensitive values. This means, for example, that we cannot use a look-up table to implement the AES SBox. We can go around this problem by calculating the SubByte result on the fly. If performance is an issue, a special instruction can be added specifically for SBox calculation. Most other countermeasures have a similar restriction with respect to a lookup-table based SBox [48, 49, 55].

Use Model

The processor runs in two modes: regular mode and secure mode. The secure pipeline is active only in the secure mode. Customized instructions are included to do the mode switching and data transfer between two pipelines' register files. To support security at the system level, security-mode-switching instruction only runs when the software process has a high authority; data-transfer instructions only run in secure mode. Details

about security at this level are out of the scope of our research. A similar mode-switching can also be found for example in the ARM TrustZone processor [110].

7.1.2 Programming Method

This section discusses how to program the secure processor presented in Section 7.1.1. In Section 6.2, we showed that starting from bitsliced programming is a special way of doing VSC programming for single-core VSC. Here, we give a more generalized way of doing VSC programming as follows.

Two requirements for VSC programming on single core include:

- Only secure instructions can be used to process sensitive data. By sensitive data we mean data values that are dependent on both the observable data (inputs and outputs) and the secret key.
- Every secure instruction needs to pre-charge the destination before storing sensitive value into it.

Correspondingly, we propose the generalized programming flow in 5 steps.

Step 1, for an n -bit secure processor, implement cryptographic algorithms assuming that the processor is only $(n/2)$ -bit wide. In such a way, running the software on a n -bit processor results in two identical instances of the application running simultaneously.

Step 2, compile the software application to assembly source code.

Step 3, identify sensitive instructions in the assembly source code and convert them to secure instructions. If one sensitive instruction has no corresponding secure instruction, we decompose it into smaller instructions and find the secure counterparts. For example, if XOR is used to process sensitive data, we can expand the XOR into AND, OR and NOT operations. The result of Step 3 is that sensitive instructions will now be flowing through the secure processor pipeline instead of the regular pipeline, thus achieving side-channel resistance.

Step 4, add a corresponding pre-charge instruction before each secure instruction following the method discussed above.

Step 5, assemble and link the assembly coming out of Step 4 and obtain the executables.

As we can see, the programming method is only concerned with instruction mapping, and not with low-level hardware behavior. In this way, it is different from previous software-based countermeasures, which aim at removing side-channel leakage above the instruction-level. A further analysis will be discussed in Section 7.3.

7.2 Experimental Results

In this section, we show an implementation of our secure processor on FPGA. With real-world attacks, we demonstrate that its resistance to both power attacks and electromagnetic attacks is much better than an unprotected design and the solution in Chapter 6.

7.2.1 Implementation of Secure Processor on FPGA

We design our secure processor based on an OpenSparc Leon3 processor [103], same as Chapter 6 does. A secure pipeline with a separate register file is added to the processor, which contains secure instructions such as AND, OR, NOT, MOV, and SHIFT. We also introduce a secure mode to the processor and enable data transfer between the regular register file and the secure register file controlled by customized instructions. The secure 32-bit pipeline consists of 32 slices as described in Section 7.1.1. We use PlanAhead [111] to constrain each slice to a rectangle with identical placement, shown in Figure 7.3. The only difference between a direct and a complementary slice is that the AND/OR gates (implemented with stand alone LUTs) for AND/OR instructions in direct slice are replaced with OR/AND gates in complementary slice. Although we are not able to control the FPGA routing directly, the identical placement on every slice yields much

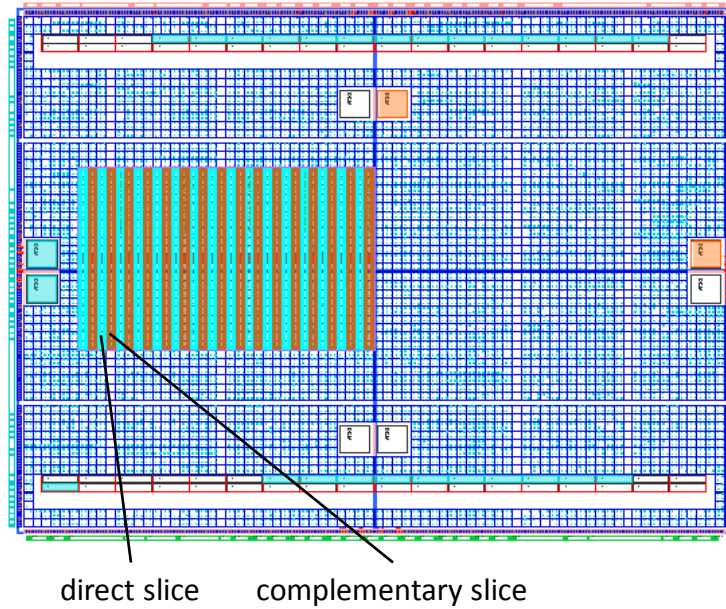


Figure 7.3: Secure processor implementation on FPGA. Secure pipeline is divided into 32 slices.

more similar routing than placement without constraints. The secure pipeline has a large 136-word general-purpose register file. The secure memory is supposed to be implemented with Block RAMs. Since Block RAMs in FPGA are located in dedicated positions, we were not able to integrate them in the secure pipeline. But we are still able program the software (AES-128) under test purely with the large register file. In ASIC, integrating RAMs to the secure pipeline is feasible. Finally, the new secure processor is implemented on a Spartan-3E 1600 FPGA running at 50 MHz.

The basic Leon3 processor, including pipeline, cache controllers, AMBA AHB interface, UART, and GPIO, takes 14103 LUTs, 4867 flip-flops, and 18 16k-bit BRAMs. Each slice of the secure pipeline takes 149 LUTs and 32 flip-flops. The register file is implemented as distributed RAM in LUTs. So in total, 32 slices take 4768 LUTs and 1024 flip-flops. A simple area cost comparison shows that the secure pipeline is around 30% ($\frac{4768+1024}{14103+4867} \approx 0.3$) as large as the basic Leon3. If we take the BRAM into account and keep 30% as the target, we can integrate 80k-bit RAM ($\frac{80}{18 \times 16} \approx 0.3$) to the secure pipeline, which is enough for most crypto-algorithms.

7.2.2 Experimental Results

We use the same measurement setup as Chapter 6 does, shown in Figure 6.5. We also port the same full AES-128 software, including protected and unprotected, to the new secure processor. After that, we mount real-world power and electromagnetic attacks. Both CPA and CEA focus on the output of the *SubByte* step in the first round. We use Pearson correlation between the measurement and the side-channel information hypothesis (Hamming weight of *SubByte*'s output) to differentiate correct key bytes from incorrect ones [62].

We gradually increase the number of averaged measurements from 512 to 512,000 and obtain the results shown in Table 7.1. With only 1280 averaged measurements, both CPA and CEA can fully break the unprotected AES software: all 16 key bytes are uncovered. To fully break the solution in Chapter 6, DPA needs 25600 averaged measurements while CEA needs 51200 averaged measurements. But with 51200 averaged measurements, CPA only breaks 8 key bytes and CEA only breaks 1 byte of our new solution. With half a million averaged measurements, which require 16 million instant measurements, CPA is still incapable of uncovering 7 key bytes and CEA only breaks 4 key bytes. This shows that our secure pipeline offers much stronger security than the initial solution in Chapter 6. Compared with the unprotected AES software, our solution obtains at least 1000 ($512000/512$) times security improvement. Note that the experimental setup is optimized for attacks with clean power source and with triggers inserted to help aligning different measurements. So even 16 million is an optimum number of measurements for a practical attack to get around half of the key bytes.

Our new solution is 6.5 times slower than the unprotected AES. Since we use the same software as Chapter 6, the footprints are also the same, around 3.3 times larger than the unprotected software. We notice that even the footprint of unprotected AES is quite large here. This is because the unprotected AES is not optimized. For example, the 10-round loop of AES is unrolled. As the unprotected AES's footprint decreases due to optimization, the software for our solution will also become smaller proportionally.

Table 7.1: Attack results summary.

Parameter	unprotected	balanced processor	secure processor
throughput (kb/s)	$207 \times 2 = 414$	64	64
footprint (kB)	45.7	150.2	150.2
key bytes NOT found (CPA/CEA)			
@ 512 ¹	3/3	16/16	16/16
@ 1,280 ¹	0/0	15/16	15/16
@ 12,800 ¹	0/0	4/7	14/16
@ 25,600 ¹	0/0	0/6	12/15
@ 51,200 ¹	0/0	0/0	8/15
@ 512,000 ¹	0/0	0/0	7/12

¹ Number of averaged measurements, every averaged measurement is an average of 32 instant measurements with the same plaintext.

7.3 Analysis

After one decade of research, we see different types of software protection solutions. Some researchers rewrite the cryptographic algorithms to introduce randomness to the processing while keeping the crypto functions unchanged [49, 50, 51, 52, 53, 55]. Some also shuffle the software instructions or insert random delays to increase the difficulty of locating sensitive side-channel information [67, 112, 47, 54]. We call these software-based countermeasures because they solve the problem using only software programming. We also see researchers proposing secure microprocessors so that the software running on top is automatically protected. Some researchers use secure logic styles [109, 48], while others introduce randomness to the processor execution [66, 113, 114]. We call these solutions hardware-based countermeasures since they are completely based on hardware design techniques. Besides the above two categories, other researchers propose solutions that involve both hardware and software [101]. We call these solutions hw/sw-based solutions.

To analyze our solution in terms of security, cost, and performance, we compare it with one other HW/SW-based solution in [101], one recent software-based solution [55], and one recent hardware-based solution [48]. According to [55] and [48], we consider the solutions presented by these two works are the best representatives in their own categories.

In [101], Ambrose et al. map a direct AES and a complementary AES to a synchronized dual-core processor, so that the total hamming weight of two related intermediate values in two cores is data-independent. Although our dual-core VSC solution in Chapter 5 is much better than their solution, we use their solution for the comparison purpose. In [55], Rivian et al. rewrite the AES algorithm so that the intermediate values are all randomized. In [48], Tillich et al. combines a secure logic style, to protect the instruction set extension, with architectural masking, to protect the storage. Since they did not mention a preferred secure logic style, we assume that they use DRP-based technique (WDDL [43]). We use the terms MUTE-AES, SW-MASKING, and HW-PROCESSOR

to represent the above solutions.

7.3.1 Security Analysis

MUTE-AES follows the DRP technique but without pre-charge operations. Although the total hamming weight of two related values in two cores are data-independent, it is still vulnerable in front of attacks that use hamming distance as the power model. Our dual-core VSC is better than MUTE-AES, shown in Section 5.6, but it still suffers from electromagnetic attacks.

Security of SW-MASKING is based on masking. A threat to masking is higher-order attacks. An n -order masking can be broken by an $(n + 1)$ -order attack. However, as the order of attacks increases, the difficulty of attacks also increases exponentially [64]. It is believed that to a certain order, the attacks will become infeasible. Oswald et al. and Tillich et al. showed that first-order masking can be broken by practical second-order attacks [77, 69]. Moreover, Coron et al. [98] even shows that 3rd-order attacks can be successful with an acceptable number of measurements. Therefore, we use the 3rd-order masking scheme for SW-MASKING in our cost comparison.

HW-PROCESSOR relies on secure logic circuit to protect instruction extensions. Since the processor with secure logic style has not been implemented, there are no detailed attack results published. Here, we assume that they use DRP technique. A major problem of DRP technique is that direct circuit and complementary circuit are usually of different lengths or widths, which results in residual side-channel leakage. Fortunately circuit designers are able to design almost identical direct and complementary paths so that the attacks are infeasible [81]. This means that, with careful circuit design, security of HW-PROCESSOR could be strong enough.

Our new solution follows the VSC concept and integrates the DRP technique into processor micro-architecture. It has similar security properties as DRP circuits. In addition, the slice-based pipeline design makes it very easy to realize identical direct and complementary routing. The experiments show a security improvement of at least

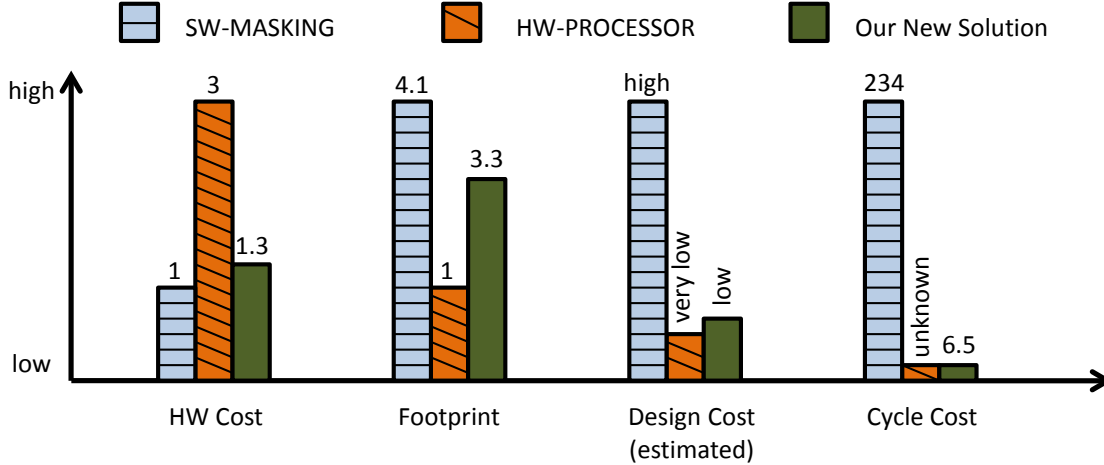


Figure 7.4: Cost and performance (cycle cost) of different countermeasures. The numbers represent the comparison between a solution and the related unprotected design. The height of the highest bar in each category is normalized.

1000 times over the related unprotected design. In total, 16 million measurements only uncovered around half of the key bytes. In ASICs, where routing is controllable, attacking it will be even harder.

Therefore, we consider that SW-MASKING, HW-PROCESSOR, and our solution are able to achieve satisfying security. The following analysis only focuses on these 3 solutions.

7.3.2 Cost and Performance Analysis

Figure 7.4 shows our analysis on SW-MASKING, HW-PROCESSOR, and our solution for different aspects of performance and cost, including hardware cost (circuit area), software cost (memory footprint), design cost, and cycle cost.

SW-MASKING causes no or very little hardware cost overhead. But its footprint increase 4.1 times over non-protected AES. HW-PROCESSOR requires no significant change on software, so the footprint should remain more or less the same. However, it increases the hardware cost. Although the authors did not use secure circuit to implement the entire processor, the hardware overhead of the instruction extension and

the architectural masking support is already large. According to our calculation, their secure processor with secure logic style implemented is at least 3 times as large as a basic Leon3 processor [115].

The hardware cost overhead of our solution is only 30% of a basic Leon3 processor. Compared with HW-PROCESSOR, ours reduces the overhead significantly. HW-PROCESSOR uses secure logic style to protect instruction set extension and masking to protect memory, which increases the hardware cost of the processor micro-architecture. In contrast, we integrate DRP technique at the architecture level, which enables us to protect every part of the processor with a simple and consistent countermeasure. Moreover, we do not expect the secure pipeline to execute all software: the processor still has a regular pipeline for that. The secure pipeline is only used for cryptographic software that needs side-channel resistance and it has a reduced instruction-set. The software footprint is increased for our solution. For secure AES, the footprint is increased by 3.3 times. But this is still smaller than SW-MASKING (4.1 times) in [55].

Another cost is design cost. In general, software-based solutions require the software designers to be fully responsible for the security problem. As a result, software designers need to understand cryptography and side-channel security very well. From [51] to [55], it has taken one decade for researchers to search for masking solutions for one single algorithm: AES. Given another algorithm, we can expect the design cost to be very high. Besides this, software designers have to guarantee that security features are implemented correctly. So far, we have not seen a systematic way to verify side-channel security at software-level. This means that software designers need to test their software with low-level methods, either hardware simulation or real attacks. This results in a high debug cost. In contrast, HW-PROCESSOR solves the problem at the processor level, with a one-time design effort. The design cost for software is the same as regular software design. For our solution, we define an instruction-set architecture for SCA-resistant programming. When using this instruction set, all that software designers have to do is follow our programming method, regardless of the low-level hardware behavior. So, we

see that although our solution increases the design cost (for software), it is much less than SW-MASKING. Since it is hard to quantify design cost exactly, Figure 7.4 reflects our estimates.

Performance is a big issue for SW-MASKING which decreases performance 234 times over an unprotected design. No result has been shown on the performance of HW-PROCESSOR. But since HW-PROCESSOR shifts the protection from software to hardware, it is expected that its performance is much better than SW-MASKING. Our solution decreases the performance by 6.5 times. Moreover, our solution can increase the performance easily by expanding the secure pipeline or adding customized instructions for specific algorithms.

7.3.3 Analysis Summary

According to the above analysis, we can see that our solution is the only hw/sw-based solution that delivers high SCA-resistance. The SW-MASKING's prominent advantage is low hardware cost. As Figure 7.4 shows, HW-PROCESSOR's advantage is few software modifications. But they also have obvious disadvantages. SW-MASKING has very high design cost and poor performance. HW-PROCESSOR has a high hardware cost. Our solution inherits advantages from both software-based and hardware-based solutions. In general, it performs well in all aspect, including security, cost, and performance. In most cases, it is close to the best solutions, without obvious disadvantages.

7.4 Conclusions

Based on the experimental and analysis results, we conclude that the new solution in this Chapter is effective and efficient. By comparing our solution and previous related solutions, we see the importance of making hardware and software working together. We also see that good processor micro-architecture and implementation are critical to realize a good concept. This is why our new solution is better than the work in Chapter 6 and

makes the VSC concept become a practical and promising SCA countermeasure.

Chapter 8

High-Performance Montgomery Multiplication for Public-Key Cryptography on Multi-Core Processors

In the previous chapters, we discussed how to use dual-core architecture and instruction set extensions to implement SCA-resistant solutions. In this chapter we focus on improving performance on multi-core architecture. The target application is one of the most time-consuming cryptography algorithms: RSA.

8.1 Introduction

As discussed in Section 2.2.2, multi-core architectures are replacing single-core architectures at a rapid pace. Indeed, multi-core architectures offer better performance and energy-efficiency for the same silicon footprint [116]. As a result, multi-core architectures can now be found in a wide range of system architectures, including servers, desktops, embedded and portable architectures [90, 89, 88, 117].

Parallel software development is a major challenge in the transition of single-core to multi-core architectures. Programming environments such as OpenMP or MPI provide a parallel programming model to abstract the underlying parallel architecture. A programmer then has to express a sequential algorithm as parallel, communicating tasks. Under ideal circumstances, a parallel version of a sequential program will run N times faster on N processors, yielding a speedup of N . In practice, the actual speedup is less, because of unbalanced partitioning of the algorithm, and because of inter-processor communication dependencies. Parallel-software programmers therefore try to balance the computational load as much as possible, and to minimize the effect of inter-core communication.

In this Chapter, we investigate how to parallelize the Montgomery Multiplication (MM) [118]. This is an algorithm for modular multiplication that is extensively used in public-key cryptography. In practice, the wordlength of such MM is many times larger than the wordlength of the underlying processor. This leads to multi-precision arithmetic. For example, a direct implementation of a 2048-bit multiplication on a 32-bit processor will require 4096 32-bit multiplications just to obtain all the partial products. This is because a multiplication has quadratic complexity. For modular multiplications, the complexity is even higher because the reduction (modular-part) needs to be implemented as well. Koç summarized several sequential solutions for multi-precision MM [119].

Several authors have explored straightforward parallelization of the MM by executing multiple independent MM on multiple cores in parallel [120, 121, 122, 123, 124]. This scenario is a *fixed-latency* scheme: it optimizes the throughput (MMs completed per second), but it does not enhance the latency of a single MM (time to finish one MM). Public-key operations such as RSA [7], DSA [8], and ECC [9] contain thousands of MMs. However, these MMs contain strong data dependencies, and the result of one MM may be needed as an operand for the next MM. A high-throughput, fixed latency scheme for MM therefore cannot accelerate a single instance of RSA, DSA, or ECC. In

an embedded-computing context, where a single user is waiting for the result of a single public-key operation, a *low-latency* scheme is needed instead. In the development of a low-latency MM, we are looking for the following properties.

- *Balanced task partitioning.* Partitioning divides one MM into several different parts and assigns them to different cores for computation. A balanced task partitioning means that each core has the same task load. Better balancing will improve the potential speedup that the parallel program can achieve.
- *High tolerance to the communication delay.* Running one MM with multiple cores inevitably leads to inter-core communication. Inter-core communication is usually much slower than within-core communication. Therefore, inter-core communication is a likely bottleneck in parallel solutions. A high tolerance to the communication delay means that the performance of the parallel solution is not severely influenced by inter-core communications. This feature makes it easier to port the parallel MM to different multi-core architectures and stabilizes the performance as the communication delay varies.
- *High scalability.* A good parallel design should be able to handle variations of the algorithm as well as of the target architecture. At the algorithm level, the design parameters of the public-key algorithms may vary (RSA-1024 and RSA-2048, for example). Also, future multi-core systems can be expected to have a larger number of cores than the systems of today, as a result of increased integration. Therefore, we are looking for a scalable, parallel implementation of the MM.

Several implementations of a parallel MM have been presented before, though we believe that none of these meets all the requirements we enumerated above. Bajard et al. proposed to formulate the MM using a residue number system (RNS) [37]. An RNS allows to break the long-wordlength operation of the MM into a set of smaller numbers, which then leads to a set of parallel operations. However, the resulting design is not fully

balanced over the complete set of processors. We will show that we can achieve a multi-precision implementation of MM that is, from a computational perspective, perfectly balanced over all processors.

Another parallelization of MM was proposed by Kaihara et al. with the Bipartite Montgomery formulation [34, 35]. A bipartite design breaks a MM in two parts, so that two processors can compute a MM in parallel. This solution cannot be easily scaled to multiple cores, since each part still runs on a single processor. Enhancements for further parallelization, such as the tripartite MM by Sakiyama et al., show that this parallelization is quite complex and the scalability is still limited [36].

Sakiyama et al. and Fan et al. describe parallel, scalable implementations of the MM using custom-designed processors [125, 126, 127]. Such implementations are not portable, and cannot be expressed on top of OpenMP or MPI because the underlying programming language does not support the specialized operators used in their designs. The same problem can also be found with an implementation of arithmetic modulo minimal redundancy Cyclotomic Primes for ECC by Baldwin et al. [128]. In contrast, we seek a portable, software-only formulation of the MM.

Our Contribution:

We propose a parallelization of the MM called parallel Separated Hybrid Scanning (pSHS). Instead of turning to the algorithm conversions like RNS and bipartite MMs, pSHS directly partitions a regular MM in such a way that each processing core has the same computational load. This partitioning method can scale over different number of cores. In addition, we present a detail analysis of the inter-core communication overhead of pSHS, and we conclude that this algorithm has a high tolerance to inter-core communication delays. We will show that pSHS has the 3 properties listed above: *balanced task partitioning*, *high tolerance to communication delay*, and *high scalability*.

We have built multi-core prototypes with 2, 4 and 8 soft-cores on an FPGA. Through our experiments, we see that pSHS offers a considerable speedup over the sequential solution and it is easy to scale over different numbers of cores. For example, a parallel

2048-bit MM implemented on 32-bit embedded cores has speedups of 1.97, 3.68, and 6.13 based on 2-, 4-, and 8-core architectures respectively when the inter-core communication latency is as high as 100 clock cycles. The efficiencies per core are as high as 0.99, 0.94, and 0.82 respectively. We also show that integrating pSHS to RSA does not lose any of pSHS’s advantages. The parallelized 2048-bit RSA encryption gains almost the same speedups as pSHS. The multi-core prototypes on the FPGA represent a category of multi-core architectures, where different cores run independently with the communication handled by a message-passing network. We expect the results obtained from our experiments to be portable to the similar multi-core architectures, for example the tiled processors [99, 129, 130, 91, 131].

8.2 Sequential Montgomery Multiplication and Sequential Schemes

Intuitively, modular multiplication requires division operations with arbitrary dividers which are hard to implement. MM can replace the modular- n operation with a division of the power of 2 (2^n). Thus, the division can be achieved by simple shifting, which is easy to implement in processors. This makes MM one of the most prevalent ways to implement modular multiplications and exponentiations. One of its sequential implementation schemes is shown in Algorithm 3 [119]. The most time-consuming part of Algorithm 3 is from line 2 to line 14. Our research will focus on this part and ignore the final subtraction in line 15 and 16, since it does not affect the overall performance too much. In addition, Walter presented Montgomery Exponentiation without final subtractions in MM by changing the bound of MM’s operands [132]. Such case, the influence of the final subtraction can be completely ignored. Considering the final subtraction has a very small influence to the overall performance, although the following discussion depends on Algorithm 3, the analysis also hold valid for the MM without final subtractions.

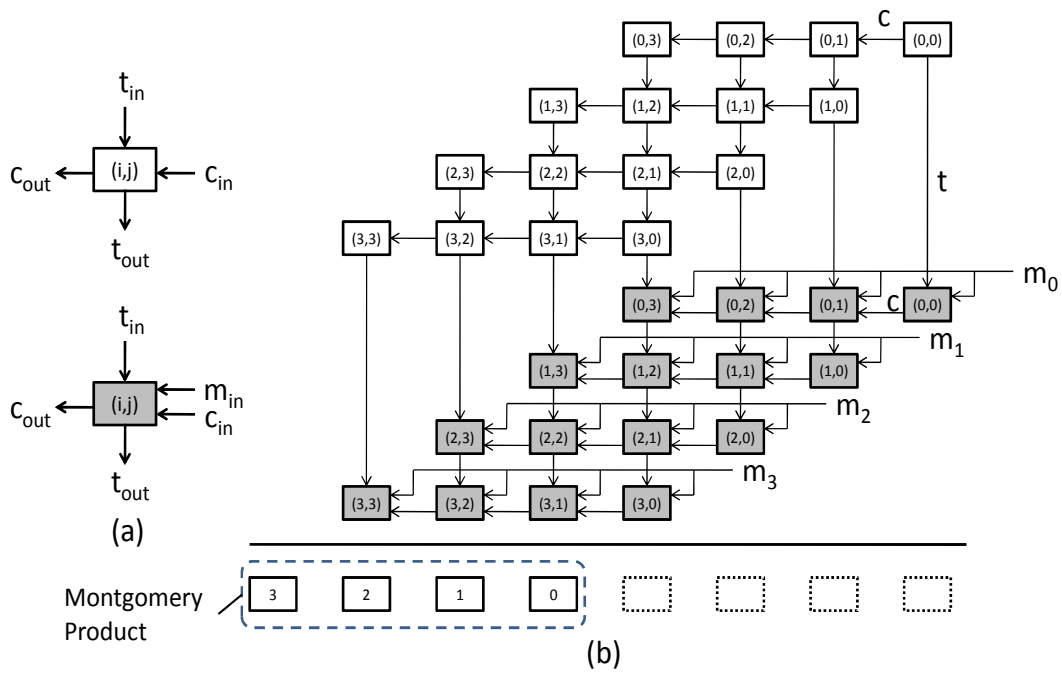


Figure 8.1: Analysis model and data flow of Montgomery multiplication. (a) Analysis model: boxes; (b) Data flow of Montgomery multiplication

Algorithm 3 Montgomery multiplication (SOS scheme) [119]

Require: An s -word modulus $N = (n_{s-1}, n_{s-2}, \dots, n_1, n_0)$, two operands $A = (a_{s-1}, a_{s-2}, \dots, a_1, a_0)$, and $B = (b_{s-1}, b_{s-2}, \dots, b_1, b_0)$ with $A, B < N$, $A, B, N > 0$, N being odd, and the constant $n' = -n_0^{-1} \bmod 2^w$. w is the word length of a system (usually 8, 16, 32, or 64). $T = (t_{2s-1}, t_{2s-2}, \dots, t_1, t_0)$ is a temporary array. $*$ means integer multiplication.

Ensure: $R = (r_s, r_{s-1}, \dots, r_1, r_0) = A * B * 2^{-n} \bmod N$. $n = w * s$.

```
1:  $T = 0$ 
2: for  $i = 0$  to  $s - 1$  do
3:    $C = 0$ 
4:   for  $j = 0$  to  $s - 1$  do
5:      $(C, S) = t[i + j] + a[j] * b[i] + C$ 
6:      $t[i + j] = S$ 
7:    $t[i + s] = C$ 
8: for  $i = 0$  to  $s - 1$  do
9:    $C = 0$ 
10:   $m = t[i] * n' \bmod 2^w$ 
11:  for  $j = 0$  to  $s - 1$  do
12:     $(C, d) = t[i + j] + m * n[j] + C$ 
13:     $t[i + j] = S$ 
14:    ADD( $t[i + s]$ ,  $C$ )           {addition and propagate carry to higher part of  $T$ }
15:  $R = (t_s, t_{s-1}, \dots, t_1, t_0)$ 
16: if  $R > N$  then
17:    $R = R - N$ 
```

After profiling Algorithm 3, we find two hot blocks: the two inner loops between line 5 and line 6 containing $a_j * b_i$ and between line 12 and line 13 containing $n_j * m_i$. To visualize the analysis, we use a white box to represent one iteration of the first loop and a shaded box for the second loop, shown in Figure 8.1(a). The inputs of a box include t_{in} and c_{in} . m_{in} is a third input to a shaded box. Every box generates two words of output with the higher part in c_{out} and the lower one in t_{out} . Figure 8.1(b) shows the data flow and data dependency of a 4-word MM. Boxes are located according to their index (i, j) . In total, there are 8 columns and 8 rows of boxes. Adding the partial results of all the boxes, we obtain the Montgomery product by keeping the upper half of the final result. In Figure 8.1(b), all the vertical connections represent dependencies based on t , while the horizontal connections represent dependencies based on c and m . All of the shaded boxes in a row share the same m , which is generated right before the first box in that row. Since the operations inside both the white and the shaded boxes are similar, we assume that they take the same amount of time, and define it as *calculation*

time unit (CTU). Besides the boxes, other operations, for example the generation of m in line 10, are simpler than the operations in one box. We approximate the time cost of them to be 0.

In [119], the authors analyzed several different sequential implementation schemes for MM. The difference among those methods comes from the sequence to calculate boxes. In general, two criteria are used to group the methods. 1) The way to handle those two kinds of boxes. If we operate them in sequence (first calculate all the white boxes and then all the shaded boxes), it is called 'separated'; if we interleave the rows or columns (for example one row of white ones and then one row of shaded ones), it is called 'coarsely integrated'; if we interleave the boxes (one white box (i, j) and then one shaded box (i, j)), it is called 'finely integrated'. 2) The order to calculate the boxes. The operation can be based on row in such a way that starting from the upmost row and ending at the bottom row with a right to left sequence inside each row. This method is called 'operand scanning'; another method is based on column in such a way that starting from the rightmost column and ending at the leftmost one with a top to bottom sequence inside each column. This method is called 'product scanning'. The above two scanning methods can be integrated together, which is called 'hybrid scanning'. Algorithm 3 presents a separated operand scanning (SOS) scheme.

8.3 Parallel Schemes

8.3.1 Multi-core Model

To make our programming scheme portable, we base our parallel scheme on a general message-passing multicore model as follows. In the multicore architecture, processing cores work independently with their own local memory. An on-chip network connects the cores together. Different cores communicate with each other by message passing through the network. When CORE0 needs to transfer data to CORE1, it packages the data (one or several words) as a message, sends it to the network and moves on. The

message will go through the network and be stored in CORE1's local memory until CORE1 reads it. If the message arrives later than CORE1's read operation, CORE1 will wait or be stalled until the message arrives.

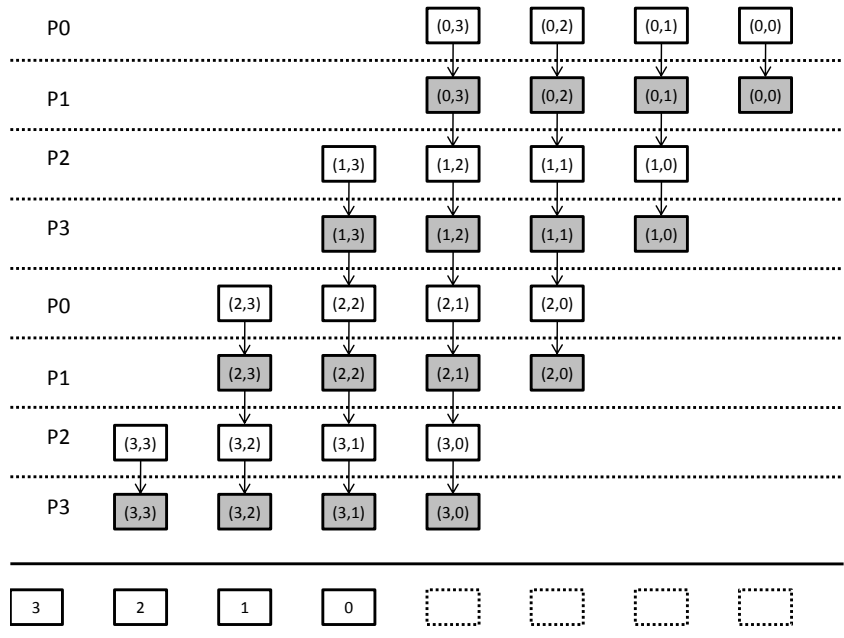
The communication latency is critical for the overall performance of a parallel system. In the analysis, we consider the worst case and use the longest communication latency for every message transfer and define it as *transfer time unit* (TTU). A parallel programming scheme that obtains satisfying analysis results from the worst case will yield good performance on many real multicore systems as well.

8.3.2 Comparing row-based and column-based partitioning

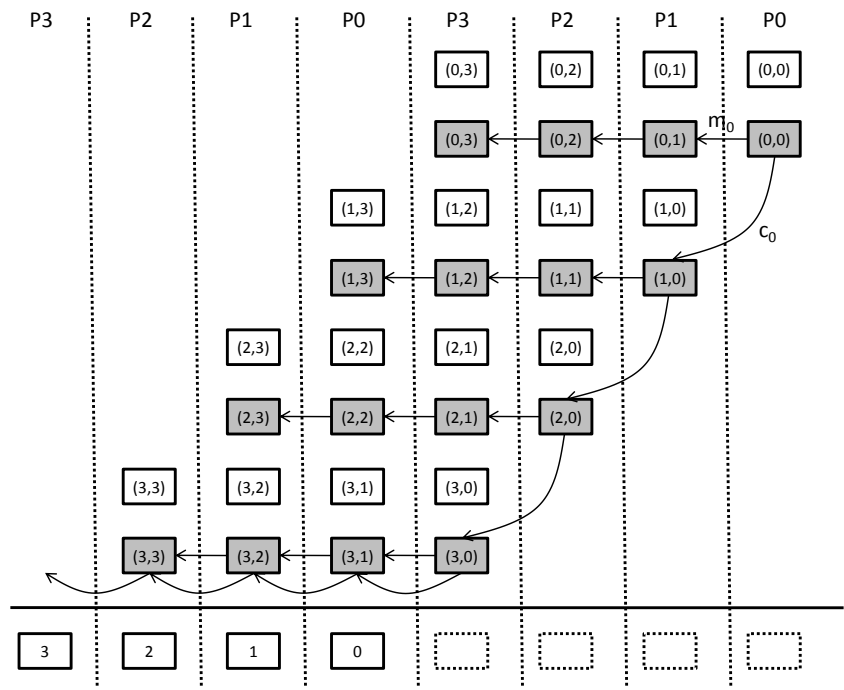
In order to obtain an efficient parallel program, we have to achieve two goals: balanced task partitioning and reduction of communication and synchronization cost. Montgomery Multiplication consists of both the regular operations, including the boxes, and the irregular operations, including generation of 'm' and the final subtraction. While it is easy to partition the process of the boxes, partitioning the irregular operations is not that easy. Fortunately, the irregular operations occur much less frequently and cost much less time than the regular operations. Therefore, we approximate a balanced task division to be assigning the same amount of boxes to different processors. Reduction of communication means less data dependency among different processors. Synchronization cost is the stalled time of a processor, which can be caused by the data dependency and the communication delay.

Row-based partitioning

One obvious method for task division is to assign different rows of boxes from the top to the bottom to different processors in a circular way. This method is illustrated in Figure 8.2(a), in which eight interleaved rows of boxes are handled by 4 cores (P_0 to P_3). To make the processors more synchronized, an 'integrated' method is used. This kind of task division follows the idea of operand scanning: each processor calculates boxes in



(a)



(b)

Figure 8.2: Row-based and column-based task partitioning. (a) row-based task partitioning; (b) column-based task partitioning.

each row from the right to the left.

Task partitioning. If the number of rows ($2 * s$) can be divided by the number of processors (p), then every processor is assigned with the same amount of boxes, which achieves a balanced task partitioning.

Communication. Based on the row-based partitioning, c and m stay at the same row while t goes from one row to another, which means two neighboring cores need to transfer t . Since t is the result of a box, it can only be transferred after the box's calculation is finished. We draw arrows to represent transferring t . For each processor, there are $s - 1$ or s words of t and the final carry word for 'send' and 'receive'. In total, the number of communications counts for approximately $2 * s^2/p$ per processor.

Column-based partitioning

Another method to partition the task is according to columns. This kind of task partitioning comes from the 'product scanning' method. The approach is to assign columns of boxes, from the right to the left, to processors in a circular way, shown in Figure 8.2(b).

Task partitioning. The entire task can be divided into two parts: the first half from column 0 to column $s-1$ and the second from column s to column $2 * s - 1$. We found that these two parts are complementary - shifting the second half to the right by s columns gives us a perfect rectangular. If the number of columns in the first half (s) can be divided by the number of processors (p), then every processor is assigned with the same amount of boxes, which leads to a balanced task partitioning.

Communication. In this method, m for a row of shaded boxes is generated in one processor P_i and then transferred to the nearest box in the same row that is assigned to the next processor P_{i+1} . After that, m is transferred in the same way until it reaches a box that is assigned to P_i again. Besides m , c also needs to be transferred. The first impression is that column-based division requires more data communication. However, improvements can be done to accumulate all the carries in one column to form a final one

(two words) and then transfer it to the next column. We represent the transfer of m and c by arrows from a box to another in Figure 8.2(b). Every time when a processor handles a column, it needs to transfer the words of m generated by the previous $p - 2$ processors as well as the words of m generated by itself. Plus, 2 words of c are transferred. Thus, the number of data communication operations is $(p - 1)s/p + 2 * 2s/p = s(p + 3)/p$ per processor.

Both of the above two schemes only require communication between neighboring processors. By comparison, we find that the column-based scheme needs less communication when $p < 2s - 3$, which usually holds true. Therefore, we base our proposed parallel programming scheme on column-based partitioning.

We also note that the Karatsuba optimization [133] is a well-known partitioning of long-wordlength multiplication. Although it leads to a sub-quadratic increase of the amount of multiplications, the Karatsuba algorithm also has a super-quadratic increase in the number of accumulations. In parallel software, where the cost of an addition and a multiplication is similar, Karatsuba optimization therefore does not lead to an obvious advantage. We do not take Karatsuba optimization into account.

8.3.3 parallel Separated Hybrid Scanning (pSHS)

We first use a small example to explain the overall idea of pSHS and then generalize it to a formal algorithm. In Figure 8.3, we show the example of a 6-word MM with 12 rows and 12 columns of boxes processed by 3 cores. We do the task partitioning based on columns. Two adjacent columns of boxes are grouped into a Task Block (TB). Task Blocks are assigned to cores in a circular fashion. From the right to the left, Task Blocks are indexed from TB[0] to TB[5]. Core P0 starts with TB[0] and then continues with TB[3]. Similarly, P1 first handles TB[1] and then TB[4]. Inside each TB, one core first processes all the white boxes and then the shaded boxes. We put a number in the upper right corner of each box to indicate its execution order. The operation inside one TB is similar to the Separated Operand Scanning (SOS) scheme in [119]. On the other hand,

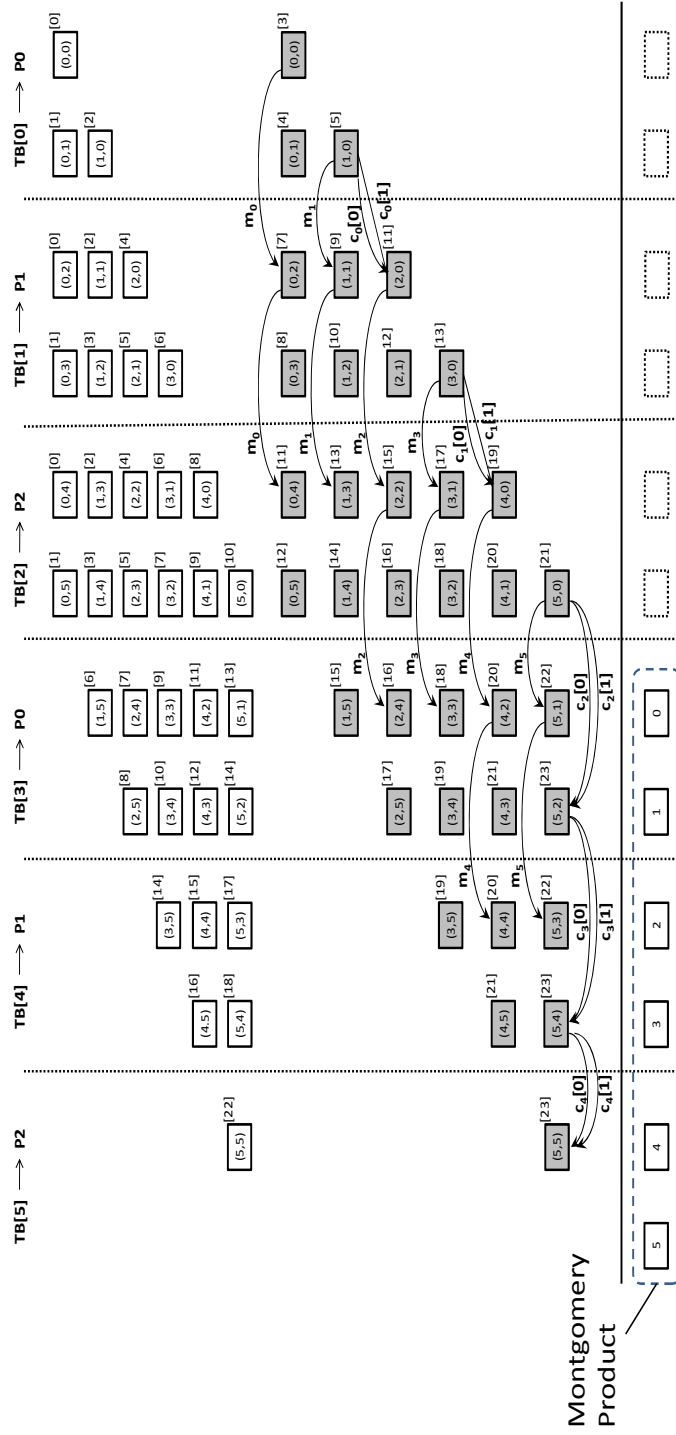


Figure 8.3: An example of pSHS ($s = 6, p = 3, q = 2$).

Table 8.1: Meanings of Symbols.

w	Word length of a processing core (usually 8, 16, 32, or 64);
s	Number of words in the multiplication operands or the modulus; for a 512-bit Montgomery multiplication on a 32-bit processor, $s = 16$; s can be divided by $p * q$;
p	Number of cores used for pSHS in the multicore system;
P_i	Order number of a core (from 0 to $p - 1$);
q	Number of columns of boxes in one TB.

the overall task partitioning is similar to Product Scanning. Therefore, we refer to our scheme as parallel Separated Hybrid Scanning (pSHS).

To integrate timing into the analysis model, *we define the top of a box as the beginning of the calculation and the bottom as the end*. All data communications between different TBs are shown as arrows in Figure 8.3. Each arrow represents one transfer operation. m_i has only one word and is sent out or required right before the rightmost shaded box of a row in each TB. As a result, one arrow for m_i starts from the top of a shaded box(i, j) and points to the top of the shaded box($i, j + q$). The carry c_i has two words. We use two transfer operations. Since c_i is generated after the last shaded box(k, l) of each TB and required by the next TB after shaded box($k, l + q$), we draw two arrows for c_i from the bottom of shaded box(k, l) to the bottom of the shaded box($k, l + q$) or the top of shaded box($k + 1, l + 1$).

The last boxes for P0, P1 and P2 share the same label: 23. This shows that *the task loads for different cores are the same (24 boxes)*. pSHS chooses column-based rather than row-based partitioning because this results in fewer messages between different TBs. This is because carries (c) of different rows in a TB can be accumulated and sent to the next TB at the end of the current TB.

To generalize pSHS scheme, we define the meanings of symbols that we will use in Table 8.1. In an s -word Montgomery multiplication, we group every q columns of boxes to form different TBs. In total, there are $2 * s/q$ TBs, which are assigned to p

Algorithm 4 parallel Separated Hybrid Scanning (pSHS)

Require: An s -word modulus $N = (n_{s-1}, n_{s-2}, \dots, n_1, n_0)$, two operands $A = (a_{s-1}, a_{s-2}, \dots, a_1, a_0)$, and $B = (b_{s-1}, b_{s-2}, \dots, b_1, b_0)$ with $A, B < N$, $A, B, N > 0$, N being odd, and the constant $n' = -n_0^{-1} \bmod 2^w$. w is the word length of a system (usually 8, 16, 32, or 64). s can be divided by $p \cdot q$. $N, A, \text{ and } B$ are stored locally for each core. $RT = (rt_{q+1}, rt_q, \dots, rt_1, rt_0)$ is a temporary array. $*$ means integer multiplication.

Ensure: $T = (t_{s-1}, t_{s-2}, \dots, t_1, t_0) = A * B * 2^{-n} \bmod N$. $n = w * s$. T is stored locally for each core. The complete Montgomery Multiplication consists of p copies of the following program. On processor P_i , execute:

```
1: for  $l = 0$  to  $(2 * s) / (p * q) - 1$  do {every iteration handles one TB}
2:    $k = P_i * q + l * p * q$ ;
3:    $pre\_pid = (P_i - 1) \bmod p$ ;  $next\_pid = (P_i + 1) \bmod p$ ;
4:   initialize  $RT$  to 0;
5:   for  $i = \max(0, k - s + 1)$  to  $\min(k + q, s) - 1$  do {white box}
6:      $ca = 0$ ;
7:     for  $j = \max(0, k - i)$  to  $\min(k - i + q, s) - 1$  do
8:        $(ca, rt[j - k + i]) = a[j] * b[i] + rt[j - k + i] + ca$ ;
9:        $h = \min(q, s - k + i)$ ;
10:       $(rt[h + 1], rt[h]) = ca + rt[h]$ ;
11:    for  $i = \max(0, k - s + 1)$  to  $\min(k + q, s) - 1$  do {shaded box}
12:      if  $i \geq k$  then {12-18 generate and communicate  $m$ }
13:         $m[i] = n' * t[i - k]$ ;  $\text{send}(next\_pid, m[i])$ ;
14:      else
15:        if  $i \geq k - (p - 1) * q$  then
16:           $m[i] = \text{receive}(pre\_pid)$ ;
17:        if  $i > k - (p - 2) * q$  then
18:           $\text{send}(next\_pid, m[i])$ ;
19:         $ca = 0$ ;
20:        for  $j = \max(0, k - i)$  to  $\min(k - i + q, s) - 1$  do
21:           $(ca, rt[j - k + i]) = n[j] * m[i] + rt[j - k + i] + ca$ ;
22:           $h = \min(q, s - k + i)$ ;
23:           $(rt[h + 1], rt[h]) = ca + rt[h]$ ;
24:        if  $i = \min(k - 1, s - 1)$  and  $k \neq 0$  then {24-27 communicate  $c$ }
25:           $\text{receive}(pre\_pid, c[0])$ ;  $\text{receive}(pre\_pid, c[1])$ ;
26:           $\text{ADD}(rt[q + 1 \text{ to } 0], c[1 \text{ to } 0])$ ;
27:           $\text{send}(next\_pid, rt[q])$ ;  $\text{send}(next\_pid, rt[q + 1])$ ;
28:        if  $k \geq s$  then
29:           $T[k - s + q - 1 \text{ to } k - s] = rt[q - 1 \text{ to } 0]$ ;
30:    if  $P_i = 0$  then
31:       $\text{receive}(pre\_pid, c[0])$ ;  $\text{receive}(pre\_pid, c[1])$ ;  $T[s] = c[0]$ ;
32:     $\text{send}(next\_pid, \text{all the results in } T \text{ stored locally})$ ; {share results}
33:     $rt[s/p - 1 \text{ to } 0] = \text{receive}(pre\_pid)$ ;
34:    store  $rt[s/p - 1 \text{ to } 0]$  to  $T$  in the correlated location;
35:    for  $k = 0$  to  $p - 3$  do
36:       $\text{send}(next\_pid, rt[s/p - 1 \text{ to } 0])$ ;
37:       $rt[s/p - 1 \text{ to } 0] = \text{receive}(pre\_pid)$ ;
38:      store  $rt[s/p - 1 \text{ to } 0]$  to  $T$  in the correlated location;
```

cores in a circular fashion. To get balanced task partitioning, s should be divisible by $p * q$. The algorithm is shown in Algorithm 4. After all TBs are processed (by line 31 in Algorithm 4), the final result has been generated but not shared: each core has one piece of it. There are many ways to distribute the result to every core. Line 32 to line 38 in Algorithm 4 implements a basic one: every piece of the result goes through every core one by one ($p - 1$ steps in total).

8.4 Analysis

This section analyzes the influence of inter-core communication latency on the overall performance of pSHS. Given a system and the application, unlike other parameters such as p , s , and CTU which have only a few possibilities, the inter-core communication latency TTU is usually more dynamic. This is because it is closely related to the status of the traffic in the communication network. Therefore, while the relationships between the performance and other parameters can be easily profiled with only a few trials by running the program on the system, the TTU 's influence needs to be formulated in a mathematical way. Actually, analyzing the influence of TTU is pretty complicated. To make it easier to understand, we first use an example to explain our analysis method. After that, a general quantitative analysis is presented. In the end, we are able to formulate the relationship between TTU and pSHS's performance when TTU is at a typical range, which shows that TTU 's influence on pSHS's performance is usually acceptable.

8.4.1 Analysis on an example

We use the same example mentioned in last section, in which a 6-word MM is processed by 3 cores. The analysis is performed in 3 steps starting from an ideal case and then approaching the answer to the above questions by considering practical and stricter conditions.

First step

Assume that $TTU = 0$. We first consider an ideal case where message communication latency is 0. According to time, we draw the parallel execution based on pSHS in Figure 8.4(a). The x-axis represents time. Six TBs with their boxes are located in six rows. The length of every box represents its execution time: CTU . The x-position of each box is decided by its starting time. Arrows show the message transfers. The length of the projection of one arrow on the x-axis indicates the longest time allowed for that message transfer, named *allowance time* (AT). We find that all AT s are no smaller than 0. Under the assumption that $TTU = 0$, all messages arrive at their destinations before the cores try to 'receive' them. This leads to a full parallelization of calculation. The execution time of Algorithm 4 (T_{pSHS}) is $24 * CTU$.

Second step

Assume that $0 < TTU < CTU$ and only one transfer exists between two cores. In this case, all cores receiving message with $AT = 0$ have to wait for the messages. In Figure 8.4(b), the black areas in TB[4] and TB[5] represent the time when P1 and P2 wait for the messages. After that, P2 needs to notify P0 the end of process. Finally, another 2 steps of transfers are used to distribute the final result. Therefore, T_{pSHS} is $24 * CTU + 8 * TTU$, shown in Figure 8.4(b).

Third step

Assume that $TTU > CTU$ and only one transfer exists between two cores. In this case, besides P1 and P2, also P0 has to wait for messages when processing TB[3], shown in Figure 8.4(c). Thus, T_{pSHS} will increase even more and even faster than the second step. As TTU increases further, black areas will eventually appear in TB[1] and TB[2], which results in T_{pSHS} 's fastest increasing speed against TTU .

According to the above analysis, we find that T_{pSHS} is a piecewise function of TTU . As TTU increases, T_{pSHS} 's slope also increases from piece to piece.

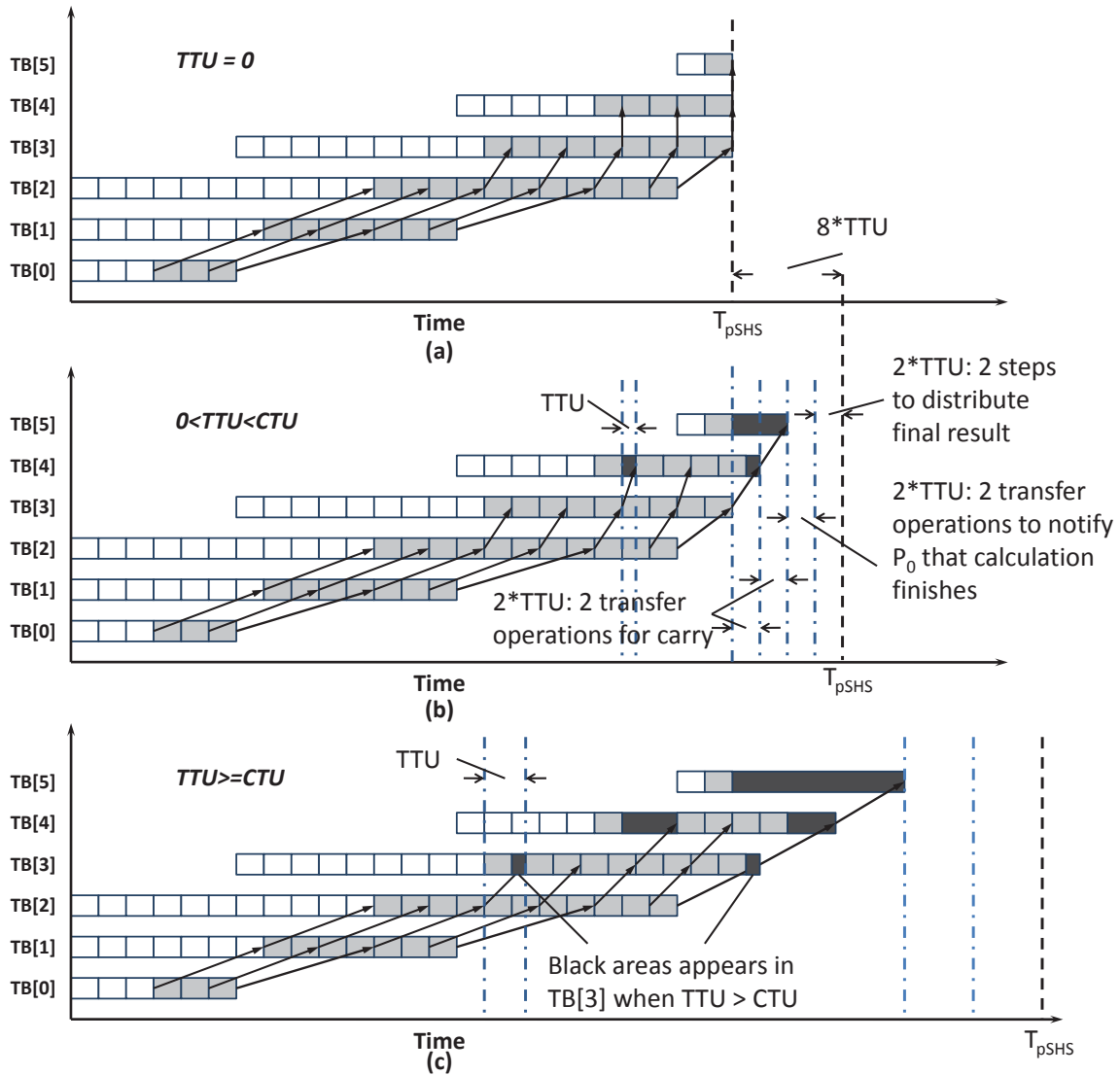


Figure 8.4: An example of pSHS in time domain with different TTU s. (a) $TTU = 0$; (b) $0 < TTU \leq CTU$; (c) $TTU > CTU$.

Table 8.2: Meanings of Terms.

<i>available time</i>	The time a message is sent from a core to the inter-core network;
<i>request time</i>	The time a message is requested by a core from the inter-core network;
<i>allowance time</i>	The longest average transfer time for each communication of a message that guarantees the receiving core does not have to wait for the message; one message may be transferred by several communications; different messages sent from the same Task Block could have different allowance time;
$t_m[i]$	The shortest <i>allowance time</i> of m messages sent from TB[i] to TB[i+1];
$t_c[i]$	The shortest <i>allowance time</i> of c messages sent from TB[i] to TB[i+1];
<i>CLT</i>	<i>Communication latency tolerance</i> ; the largest <i>TTU</i> that guarantees that the communication latency has the smallest influence on pSHS's execution time..

8.4.2 Quantitative analysis on general cases

With a similar analysis method, in this section, we perform analysis on general cases. We define 3 terms: *available time*, *request time* and *allowance time*, all of which are related to the inter-core communications. Suppose core P_i needs to transfer a data D to core P_{i+1} . *available time* of D is the time when P_i sends D to the inter-core network; *request time* of D is the time when P_{i+1} tries to receive D from the network; *allowance time* = (*request time* - *available time*)/(*number of communications*). *Allowance time* indicates the longest average transfer time for one communication that guarantees that P_{i+1} does not have to wait for the message. Table 8.2 summarizes the terms that will be used in the following analysis. With the similar method as in Section 8.4.1, we perform three steps of analysis as follows.

First step: $TTU = 0$

When $TTU = 0$, a core has to be stalled if the message received by it has a negative *allowance time*. During processing a TB, one core needs to receive several words of m

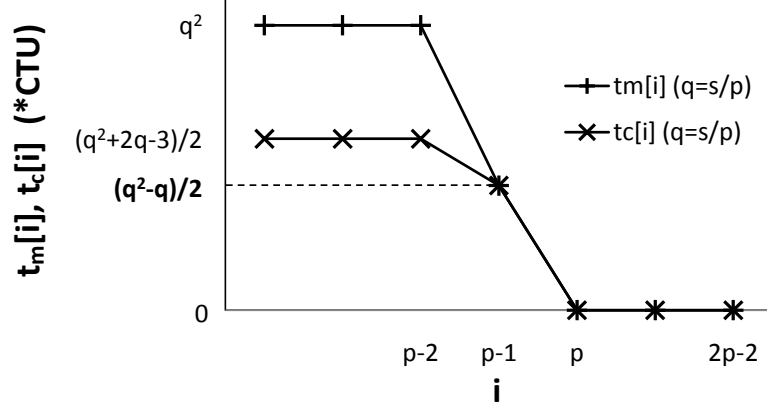


Figure 8.5: The smallest allowance time of m and c .

and two words of c . For example, P_1 needs to receive m_0 and m_1 when processing $TB[1]$ in Figure 8.3. For different m s transferred from $TB[i]$ to $TB[i+1]$, their *allowance time* could be different. We define $t_m[i]$ as the smallest allowance time of messages transferred from $TB[i]$ to $TB[i+1]$. If TTU is smaller than $t_m[i]$, the processing on $TB[i+1]$ does not need to wait for the messages sent from the process that handles $TB[i]$. Similarly, we use $t_c[i]$ to represent the smallest *allowance time* of c transferred from $TB[i]$ to $TB[i+1]$. Regardless of the data dependencies, the general expression of $t_m[i]$ is as follows, where $j = 1, 2, 3, \dots$

$$\frac{t_m[i]}{CTU} = \begin{cases} 2\lfloor \frac{i}{p} \rfloor q^2 + q^2, & i < \frac{s}{q} - 1, i \neq jp - 1 \\ 2(\lfloor \frac{i}{p} \rfloor + 1)q^2 + q, & i < \frac{s}{q} - 1, i = jp - 1 \\ 2(\frac{s}{pq} - \frac{3}{4})q^2 - \frac{q}{2}, & i = \frac{s}{q} - 1 \\ 2(\frac{s}{pq} - 1)q^2, & \frac{s}{q} \leq i < \frac{s}{q} + p - 2 \end{cases}$$

Once $i \geq s/q + p - 2$, there is no need to transfer m . With the same method, we can also obtain $t_c[i]$ as follows, where $j = 1, 2, 3, \dots$

$$\frac{t_c[i]}{CTU} = \begin{cases} \frac{1}{2}[(2\lfloor \frac{i}{p} \rfloor + 1)q^2 + 2q - 3], & i < \frac{s}{q} - 1, i \neq jp - 1 \\ \frac{1}{2}[2(\lfloor \frac{i}{p} \rfloor + 1)q^2 + 3q - 3], & i < \frac{s}{q} - 1, i = jp - 1 \\ \frac{1}{2}[2(\frac{s}{pq} - \frac{1}{2})q^2 - q], & i = \frac{s}{q} - 1 \\ (\frac{s}{pq} - 1 - \lfloor \frac{iq - s}{pq} \rfloor)q^2, & i > \frac{s}{q} - 1, i \neq jp - 1 \\ \frac{1}{2}[(\frac{2s}{pq} - 3)q^2 - q - 2q^2 \lfloor \frac{iq - s}{pq} \rfloor], & i > \frac{s}{q} - 1, i = jp - 1 \end{cases}$$

We draw $t_m[i]$ and $t_c[i]$ in Figure 8.5 as an example with $q = s/p$. According to the general expressions of $t_m[i]$ and $t_c[i]$, when $i < p$, $t_m[i]$ and $t_c[i]$ are both larger than 0. As i increases, both $t_m[i]$ and $t_c[i]$ decrease. For the last $p - 1$ TBs, their $t_m[i]$ and $t_c[i]$ become 0. Even if we consider the data dependencies, when $TTU = 0$, pSHS still achieves a full parallelization during the calculation.

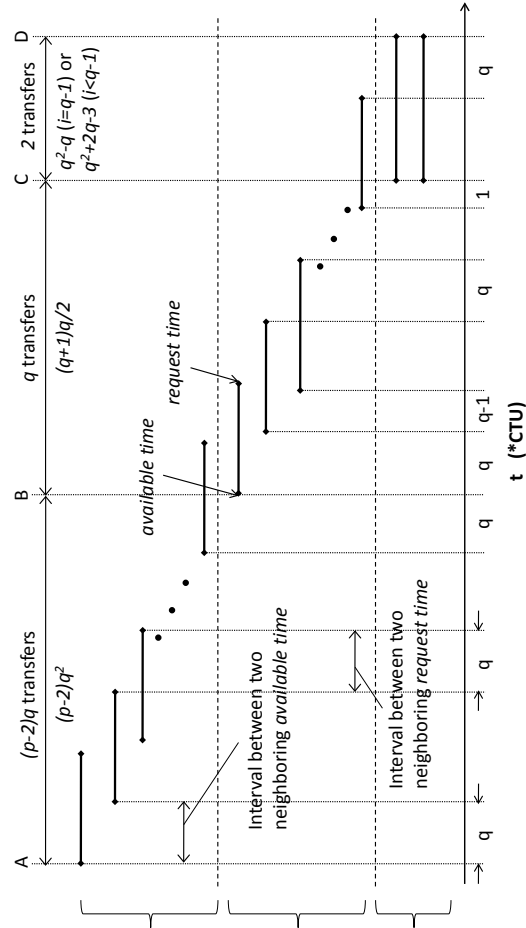
Second step: $TTU > 0$ and only one communication exist between two cores

In this case, the full parallelization is not realizable since the last $p - 1$ TBs will definitely have to wait for the messages ($t_c[i] = t_m[i] = 0, p \leq i \leq 2p - 2$). All the other TBs ($t_m[i], t_c[i] > 0, 0 \leq i \leq p - 1$) could be uninfluenced by the communication delay, when TTU is smaller than a certain limit (similar to the second step in Section 8.4.1). We define this communication delay limit as *communication latency tolerance (CLT)*. *CLT* guarantees that the communication latency has the smallest influence to pSHS's execution time. As discussed in Section 8.4.1, pSHS's execution time is a piecewise function of TTU . Intuitively, *CLT* indicates the largest TTU of the first piece. The following discussion discovers *CLT* and the execution time of pSHS.

Discovering *CLT*

CLT means the largest TTU that guarantees TBs with non-zero $t_m[i]$ and $t_c[i]$ not delayed by the communication delay. We first consider the case when $q = s/p$ (the largest q we can choose).

For TBs with non-zero $t_m[i]$ and $t_c[i]$, the *available time* and *request time* of each



GROUP1: send $(p-2) \cdot q$ words of m to the next core, which are received from the previous $p-2$ cores. The interval between two neighboring *available time* is always $q \cdot CTU$. The interval between two neighboring *request time* is also always $q \cdot CTU$.

GROUP2: generate q words of m and send them to the next processor. The interval between two neighboring *available time* reduces from $q \cdot CTU$ to $2 \cdot CTU$. The interval between two neighboring *request time* is also always $q \cdot CTU$.

GROUP3: generate c and send it to the next processor with two transfers. The time interval between two *available time* is 0. The time interval between two *request time* is also 0.

Figure 8.6: Available time and request time of data communications. Every communication in this example starts from $TB[i]$ and ends on $TB[i+1]$ ($i < p, q = s/p$).

data transfer from $TB[i]$ to $TB[i+1]$ can be illustrated in Figure 8.6. The horizontal axis represents time. A bunch of horizontal lines are used to represent data transfers from $TB[i]$ to $TB[i+1]$ with the starting point of a line as the *available time* and the end point as the *request time*. We categorize the lines into three groups. In GROUP1, $TB[i]$ sends $(p - 2) * q$ words of m to $TB[i + 1]$, which are received from the previous $p - 2$ cores. In GROUP2, $TB[i]$ generates q words of m , and sends them to $TB[i+1]$. In GROUP3, $TB[i]$ generates 2 words of c and sends them to $TB[i+1]$.

For simplicity, we divide the time domain into 3 parts corresponding to the three groups of lines: from A to B (AB), from B to C (BC), and from C to D (CD) in Figure 8.6. Accordingly, we get $AB = ((p - 2)q^2) * CTU$, $BC = ((q^2 + q)/2) * CTU$, $CD = (q^2 - q) * CTU$ when $i = q - 1$ or $CD = (q^2 + 2q - 3) * CTU$ when $i < q - 1$.

For AB , if $TTU \leq q * CTU$, the next TB does not have to wait for the messages in GROUP1. For messages in GROUP2 and GROUP3, if the communication network can finish $q + 2$ communications within time $(3q^2 - q) * CTU/2$, the next TB can be processed without waiting for messages in GROUP2 and GROUP3. Therefore, if the TTU is smaller than $\min(q * CTU, \frac{(3q^2 - q)}{2(q+2)} * CTU)$, the next TB does not have to wait for any message sent from the current TB, and this is the value of CLT . We approximate CLT with Equation 8.1.

$$\begin{aligned}
 CLT &= \min(q * CTU, \frac{(3q^2 - q)}{2(q + 2)} * CTU) \\
 &= \begin{cases} \frac{(3q^2 - q)}{2(q+2)} * CTU, & q = 1, 2, 3, 4 \\ q * CTU, & q > 4 \end{cases} \quad (8.1)
 \end{aligned}$$

Discovering execution time

When $0 < TTU < CLT$, the execution time changes as TTU changes because $TB[i]$ with $t_m[i - 1]$ or $t_c[i - 1]$ being 0 are delayed, which corresponds to $TB[i]$ ($i > p$) in Figure 8.4(b). In this case, core P_i is stalled by $2 * i * TTU$. The overall calculation is delayed by $2 * p * TTU$ (P_{p-1} sends two words of c to P_0 as the finishing signal). In addition, to share the final result, $p - 1$ steps of data transfer are needed. Therefore,

the total delay is $(3p - 1) * TTU$.

When $TTU > CLT$, the processing cores also have to wait for messages when processing other $TB[i]$ ($i \leq p$). Therefore, T_{pSHS} increases more quickly than the previous case. In detail, since the interval between two consecutive m messages are $q * CTU$ which approximately equals to CLT , once $TTU > CLT$, every TB except $TB[0]$ will be delayed by the communication latency. The additional delay on $TB[i]$ ($i \leq p$) depends on q m messages and 2 c messages. So the delay of the first half MM increases $p * (q + 2)$ times faster than TTU . When $i > p$, the increase of the execution time is exactly the same as the previous case when $TTU < CLT$, which is $(3p - 1)$ times faster than TTU . After adding these two parts together, we can see that, when $TTU > CLT$, the execution time increases $pq + 5p - 1$ times faster than TTU .

Considering SOS scheme is used in every TB , the time cost of pSHS can be represented by Equation 8.2, where PO means pSHS's parallel overhead due to the increased complexity when compared with SOS.

$$T_{pSHS} \doteq \begin{cases} \frac{T_{SOS}}{p} + (3p - 1) * TTU + PO, & TTU \leq CLT \\ (pq + 5p - 1) * (TTU - CLT) + T_{pSHS}(CLT), & TTU > CLT \end{cases} \quad (8.2)$$

Following the same method, we can also analyze the cases when $q < s/p$. The analysis results are similar.

8.4.3 Analysis conclusion

This section evaluates pSHS's typical performance according to the formulas obtained above. In addition, we also discuss how to choose optimal values for certain parameters to achieve the best performance.

Optimal q and p

Usually, given a platform and the application, most of the parameters are fixed, such as s and CTU . Parameters under programmers' control are q and p . Thus, we need to figure out the optimal number of columns of boxes that are grouped into one TB and the optimal number of cores that should be used to map pSHS.

From Equation 8.1, we can see that the largest q leads to the largest CLT . In addition, a smaller q requires more iterations of the i-loop in Algorithm 4, and hence *parallel_overhead* is larger. Therefore, we should always choose the largest possible number for q : (s/p) .

The relationship between T_{pSHS} and TTU is shown by Equation 8.2, when TTU is smaller than the *communication latency tolerance*. When TTU is larger than the tolerance, time cost of pSHS increases more quickly. Therefore, when TTU is small, we expect that a large number of cores will provide a better overall performance for pSHS. However, as TTU increase, larger p indicates higher increasing speed $(3p - 1)$ of the execution time. Therefore, it is possible that larger p does not always give better performance.

To find the optimal value of p purely based on Equation 8.2, we need to further formulate the term PO (the parallel overhead), which is related to the increased complexity of pSHS when compared with the sequential implementation. However, the term is highly system-dependent. Fortunately, we find another simpler way which can fulfill the same purpose. Since PO is not related to TTU , it can be measured on given a specific system. Due to the limited possibilities of other parameters, PO also have only a few possible values. After fixing PO , we can use Equation 8.2 to compare different choices of p under different values of TTU and finally obtain the optimal p for a given TTU . Both of the above conclusions will be demonstrated in Section 8.6.

Evaluation of pSHS’s typical performance

We can see that T_{pSHS} is a piecewise function of TTU : when $TTU < CLT$, T_{pSHS} stays on the first piece with the lowest increasing slope: $3p - 1$; when $TTU > CLT$, T_{pSHS} moves to another piece with higher increasing slope.

To process a s -word MM, $2s^2$ boxes should be processed in total. The execution time for a sequential program should be approximately $2s^2 * CTU$. The optimal execution time of pSHS should be $2s^2 * CTU/p$. As TTU increases from 0, T_{pSHS} ’s increasing rate, when compared with the optimal situation, is $(3p - 1)p/2s^2 * CTU$. In each box, 4 additions, 1 multiplication, 2 memory loads, 1 memory store, and 1 conditional jump are processed. Usually CTU ’s typical value is around 20 clock cycles. Suppose the application is 64-word MM (e.g. 2048-bit MM processed by a 32-bit 8-core processor), then the increasing rate equals to 0.1%. Even if the on-chip communication delay reaches CLT ($q * CTU = 160$ clock cycles), T_{pSHS} increases by 16%, when compared with the optimal case. The performance is degraded to $1/1.16 = 86\%$. Moreover, considering the influence of *parallel_overhead*, the actual degradation is even less. Therefore, when $TTU < CLT$, the influence of TTU on T_{pSHS} is acceptable. Plus, 160 clock cycles of delay also does not present a very high requirement to on-chip communications. Moreover, larger CTU , smaller word length of the processor, and longer operands of MM will all increase CLT and decrease T_{pSHS} ’s increasing speed.

8.5 Implementing RSA with pSHS

In order to verify pSHS’s effects on accelerating public-key cryptography, we implement a parallel RSA with pSHS. A small modification is made to pSHS in Algorithm 4 by adding the final subtraction. RSA’s encryption and decryption are modular exponentiation which is shown in Algorithm 5.

As we can see in Algorithm 5, modular exponentiation is mainly a set of MMs. Line 1 to line 3 do the preparation and do not cost much time. Therefore, the parallel modular

exponentiation's speedup should be very close to pSHS's.

8.6 Experimental Results

8.6.1 Experimental Setup

Experiments were built on a Xilinx Virtex 5 FPGA, where we implemented three multi-core prototypes with 2, 4, and 8 MicroBlaze cores ($w=32$, running at 100MHz) respectively. All cores are connected in a ring network, as shown with a 4-core example in Figure 8.7. Each core works independently with its own local memory. Communications between neighboring cores are implemented with Fast Simplex Link (FSL) buses. We modify FSL and make its delay programmable to emulate different communication latencies. Since the on-chip communication is not likely to have errors, the communication does not include a feedback from the receiver to the sender to verify the message is correct. A timer is attached to MicroBlaze 0 to measure the execution time. All cores and the timer are synchronized at the starting time. After every core finishes its job, MicroBlaze 0 reads the clock cycle counts from the timer, which is used as the execution time.

The FPGA with soft-cores is just for prototyping. It represents a category of multi-

Algorithm 5 parallel modular exponentiation based on pSHS

Require: An s -word modulus $N = (n_{s-1}, n_{s-2}, \dots, n_1, n_0)$, base $A = (a_{s-1}, a_{s-2}, \dots, a_1, a_0)$, with $0 < A < N$, and exponent $E = (e_{s-1}, e_{s-2}, \dots, e_1, e_0) (E > 0)$. $R = 2^n \pmod{N}$, $n = w * s$. pSHS(A, B, N, i) comes from Algorithm 5, which calculates $A * B * R^{-1} \pmod{N}$ in core P_i .

Ensure: $T = (t_{s-1}, t_{s-2}, \dots, t_1, t_0) = A^E \pmod{N}$. T is stored locally for each core.

The complete modular exponentiation consists of p copies of the following program. On core P_i , execute:

- 1: $pid = P_i$;
 - 2: $A = A * R \pmod{N}$;
 - 3: $T = R \pmod{N}$;
 - 4: **for** $i = 0$ to $n - 1$ **do**
 - 5: **if** $e_i == 1$ **then**
 - 6: $T = \text{pSHS}(A, T, N, pid)$;
 - 7: $A = \text{pSHS}(A, A, N, pid)$;
 - 8: $T = \text{pSHS}(T, 1, N, pid)$;
-

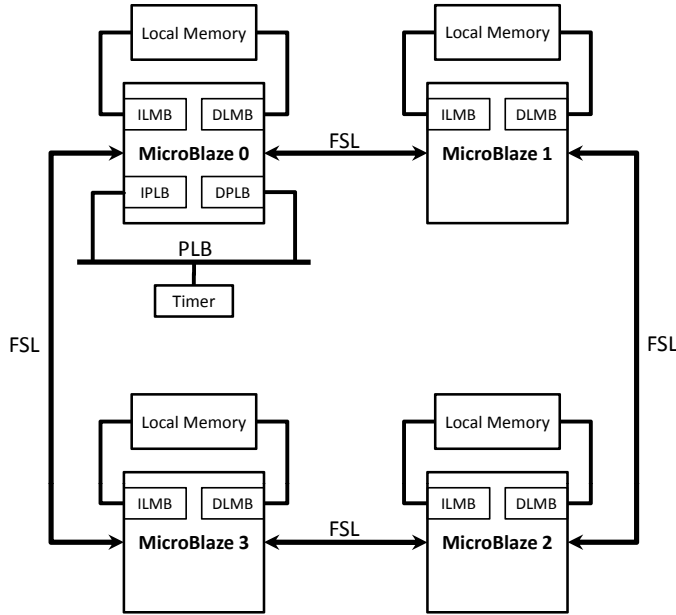


Figure 8.7: A 4-core architecture. Distributed local memory and FSL based message passing connection are employed.

core architectures where different cores runs independently and inter-core communication is done with a message-passing network. The programmable communication delay enables our prototype to represent different topologies of the network. The delay in the prototype indicates the worst case of the communication in all topologies. Therefore, we believe the results we obtained from the FPGA prototype also holds valid when applied to some existing and emerging multi-core architectures, for example the tiled processors [134, 99, 129, 130, 91, 131].

8.6.2 Experimental Results

In each platform, we implemented 512 ($s = 16$), 1024 ($s = 32$), and 2048 ($s = 64$) bit-long MMs and modular exponentiations. We programmed each of these in C according to Algorithm 4 and Algorithm 5. The compiler’s optimization level is 2.

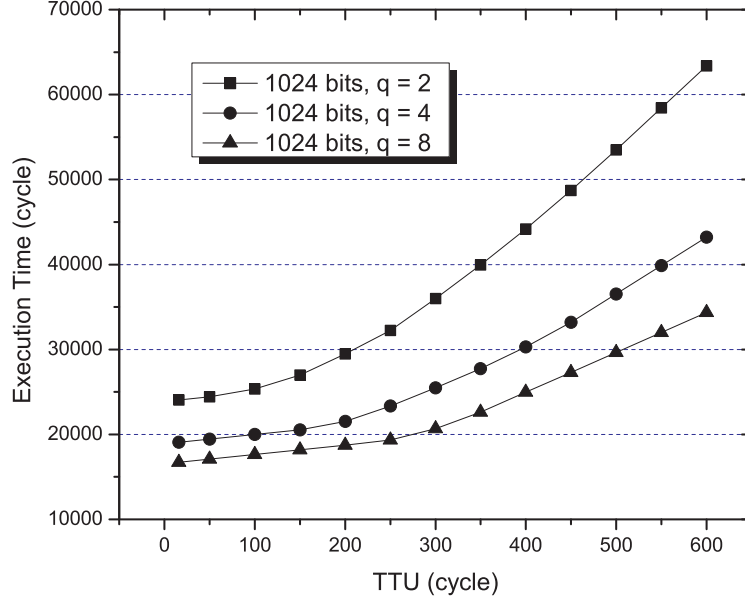


Figure 8.8: pSHS’s execution time against q . pSHS’s execution time of processing 1024-bit Montgomery multiplication with 4 cores ($s = 32, p = 4$): larger q has better performance and larger communication delay tolerance. Execution time increases faster for smaller q as TTU increases.

Finding the optimal number of columns in one TB

In every MM we tested, $q = s/p$ always leads to the best performance. We show the result with $s = 32, p = 4$ in Figure 8.8.

When TTU is small, the influence of communication delay is limited. However, since a smaller q requires more iterations of the i-loop in Algorithm 4, the performance is worse. Moreover, smaller q also leads to smaller delay tolerance, which means that its performance deteriorates more quickly as TTU increases. These two phenomena can both be observed in Figure 8.8. Therefore, *our conclusion in Section 8.4.3 that q should be s/p has been demonstrated.* In all the experiments mentioned below, we choose $q = s/p$.

Table 8.3: Execution Time of SOS.

operand	512 bits	1024 bits	2048 bits
execution time (cycles)	13953	53905	212145

Table 8.4: pSHS's Execution Time (et, cycles) & Speedup (sp).

p	TTU		32	100	300	500	700	900	1100
	p=2	512	et	8804	9144	10144	14563	19563	24563
sp			1.58	1.52	1.38	0.96	0.71	0.57	0.47
1024		et	29382	29856	30356	31856	34174	42248	50448
		sp	1.83	1.81	1.78	1.75	1.72	1.69	1.67
2048		et	107228	107568	108568	109568	110568	111568	112568
		sp	1.98	1.97	1.96	1.94	1.92	1.90	1.88
p=4	TTU		16	50	150	250	350	450	550
	512	et	5680	6054	7154	10472	13972	17472	20972
		sp	2.46	2.30	2.11	1.95	1.60	1.33	1.14
	1024	et	16716	17090	18190	19334	22609	27309	32009
		sp	3.22	3.15	3.06	2.96	2.88	2.79	2.61
	2048	et	56777	57151	58251	59351	60451	62109	66409
sp		3.74	3.71	3.68	3.64	3.61	3.57	3.54	
p=8	TTU		8	25	75	125	175	225	275
	512	et	4364	4752	6333	9003	11753	14503	17253
		sp	3.20	2.94	2.62	2.20	1.83	1.55	1.34
	1024	et	10762	11150	12300	13450	15486	19036	22586
		sp	5.01	4.83	4.60	4.38	4.19	4.01	3.82
	2048	et	32470	32858	34008	35158	36308	37458	40428
sp		6.53	6.46	6.24	6.03	5.84	5.66	5.25	

Overall performance of pSHS

We compare pSHS with the sequential version of MM to see the speedup. We implemented a sequential reference of SOS on MicroBlaze 0. The execution time of that design is listed in Table 8.3.

Since one MM has $2s^2$ boxes, according to Table 8.3, CTU is approximately 26 cycles. We calculate the speedup according to T_{SOS}/T_{pSHS} . The performance of pSHS is shown in Table 8.4.

Execution time of pSHS is listed in Table 8.4, which is visualized in Figure 8.9 for the case of $p = 4$. Each curve begins with a smaller slope followed by a larger slope

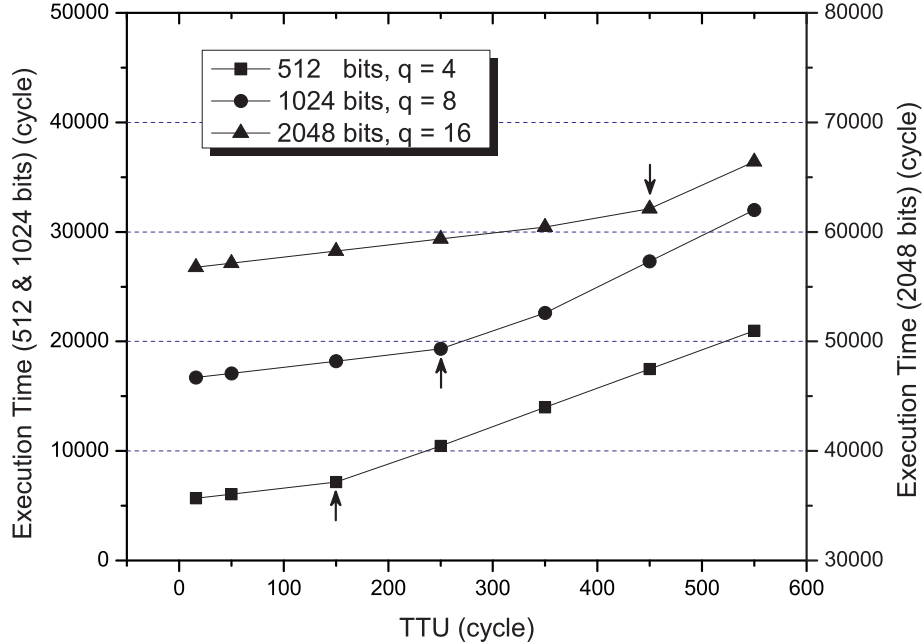


Figure 8.9: pSHS’s execution time against different word lengths. pSHS’s execution time of processing 512-, 1024-, and 2048-bit Montgomery multiplications with 4 cores ($p = 4, q = s/p$).

after TTU becomes larger than the delay tolerances. Based on $q = s/p$, we obtain q equals to 4, 8, and 16 for 512-, 1024-, and 2048-bit pSHS respectively. According to Equation 8.1, we get the following *communication latency tolerances* according to Equation 8.1: 96, 208, and 416 cycles. In Figure 8.9, the actual *communication latency tolerances* are 150, 250, and 450 cycles (indicated by arrows). *The results from analysis and experiments are very close, especially for longer operands.* The difference mainly comes from approximations in the analysis process and the discrete sample points of TTU . With TTU smaller than the *communication delay tolerance*, we calculate the slope of each of the curves. The results are all 11, which equals to the analysis result: for the case of $p = 4$, $3p - 1$ becomes 11. Therefore, *the relationship between T_{pSHS} and TTU shown in Equation 8.2 has been demonstrated to be true for the 4-core architecture.* With the same method, we find that it is also true for 2- and 8-core architectures.

When TTU is small, 4-core based pSHS’s speedup can be as high as 3.74, 3.22, and

2.46 for three operand lengths. Even when $TTU = 550$ cycles, the speedup of 2048-bit pSHS is still above 3.5. Based on 2 cores, the speedup of 2048-, 1024-, and 512-bit pSHS can be as good as 1.98, 1.83, and 1.58 respectively. Based on 8 processors, they can be 6.53, 5.01, and 3.20. Generally speaking, pSHS provides a good speedup.

Modular exponentiation's execution time highly depends on the number of '1's in the exponent. In our experiment, the exponents were randomly generated. So the probability of each bit to be '1' is close to 0.5. For comparison, we use the same inputs (A , B , and N) for both sequential modular exponentiation (with SOS) and parallel modular modular exponentiation (with pSHS). The results of the sequential and parallel versions are listed in Table 8.5 and Table 8.6 respectively.

After integrating pSHS to the modular exponentiation, the modular exponentiation's speedup is very close to pSHS's speedup, with a small drop. This lower speedup comes from the final subtraction in the modified pSHS and the preparation process, which are not parallelized. Therefore, *our Conclusion 3 in Section 8.5 has also be demonstrated.*

Finding the optimal number of cores

From Table 8.4, we realize that if the operand width (s) and the number of cores (p) are fixed, T_{pSHS} is only determined by TTU with the same slope shown in Equation 8.2. This demonstrates that *parallel_overhead* is irrelevant to TTU . As a programmer, to obtain the optimal value of p , the first step is to fix 'parallel_overhead'. For example, if the application is 1024-bit MM, three measurements of T_{pSHS} on the dual-, quad-, and octo-core platforms under a known TTU ($TTU < CLT$) would reveal *parallel_overhead* = 2404, 3064, and 3449 respectively in three platforms, according to Equation 8.2. Based on that, we are able to calculate the values of TTU that result in cross points among T_{pSHS} 's different curves corresponding to different p values. Suppose T_{pSHS} of dual-core and quad-core platforms are equal when $TTU = TTU_{cross}$, then if $TTU < TTU_{cross}$, quad-core platform performs better, when $TTU > TTU_{cross}$, dual-core platform performs better. We draw the curves in Figure 8.10 based Equation 8.2.

Table 8.5: Execution Time of sequential modular exponentiation based on SOS.

operand	512 bits	1024 bits	2048 bits
execution time (cycles)	10942151	85568236	646468715

Table 8.6: Execution Time (et, 10000*cycles) & Speedup (sp) of parallel modular exponentiation based on pSHS

		TTU	32	100	300	500	700	900	1100
		p=2	512	et	7006	7268	8040	11447	15307
sp	1.56			1.51	1.36	0.96	0.71	0.57	0.48
1024	et		46737	47270	48840	50410	54645	67239	80113
	sp		1.83	1.81	1.75	1.70	1.57	1.27	1.07
2048	et		32815	32919	33226	33532	33838	34145	34451
	sp		1.97	1.96	1.95	1.93	1.91	1.89	1.88
p=4	TTU		16	50	150	250	350	450	550
	512	et	4573	4861	5710	8242	10944	13646	16348
		sp	2.39	2.25	1.92	1.33	1.00	0.80	0.67
	1024	et	27028	27616	29343	31142	36246	43625	51004
		sp	3.17	3.10	2.92	2.75	2.36	1.96	1.68
	2048	et	17738	17852	18189	18526	18863	19369	20686
sp		3.64	3.62	3.55	3.49	3.43	3.34	3.13	
p=8	TTU		8	25	75	125	175	225	275
	512	et	3505	3807	5005	7043	9166	11289	13412
		sp	3.12	2.87	2.19	1.55	1.19	0.97	0.82
	1024	et	17681	18295	20101	21906	24991	30552	36126
		sp	4.84	4.68	4.26	3.91	3.42	2.80	2.37
	2048	et	10277	10397	10749	11101	11454	11806	12706
sp		6.29	6.21	6.01	5.82	5.64	5.48	5.09	

The cross point between dual- and quad-core platforms occurs when $TTU = 600$; the cross point between quad- and octo-core platforms occurs when $TTU = 200$. From the experiment results in Table 8.4, we can see that these two crossing points occurs around $TTU = 550$ and $TTU = 225$. In comparison, we can see that the calculated results and the experimental results are very close. Therefore, our method to find the optimal number of cores is correct.

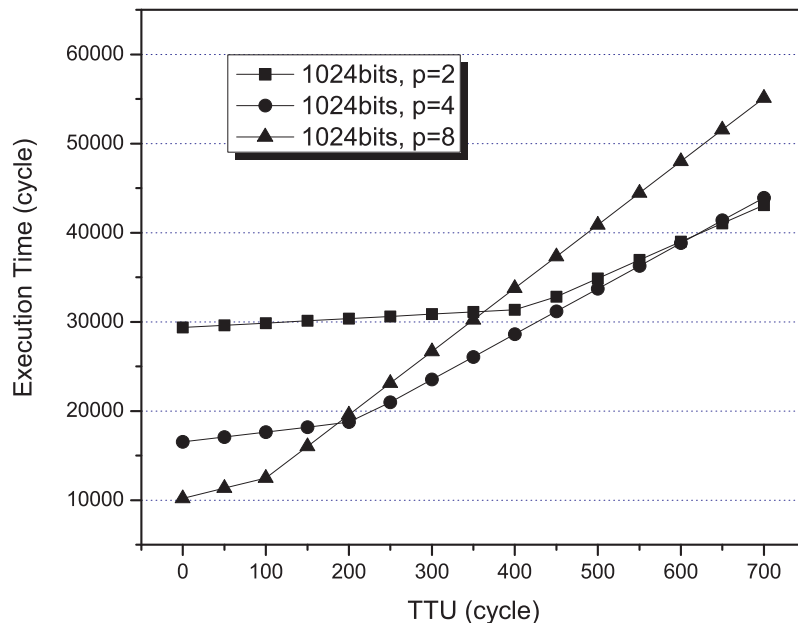


Figure 8.10: pSHS’s execution time based on analysis. 1024-bit pSHS’s execution time on 2-, 4-, and 8-core platforms based on calculation according to Equation 8.2.

8.6.3 Discussion

The multicore platforms we built are very close to the worst-case analysis model. Given better conditions, such as better topology, larger capacity of communication channels, the performance and communication latency tolerance can be improved further. However, even so, we already see a good speedup. Furthermore, using several hundreds of cycles to transfer one message between neighboring cores is not a difficult requirement for current on-chip multicore systems. Therefore, pSHS provides a good performance, a good portability and a good stability.

We find that the efficiency of pSHS is closely related to the number of words in the multiplication operand (s). Larger s leads to higher efficiency and vice versa. Based on the results, we find that pSHS has a very high efficiency when used to accelerate RSA and DSA on 32-bit systems. In addition, moving from Montgomery multiplication to modular exponentiation does not require too much additional operations. Therefore, it is reasonable to expect a good performance after integrating pSHS to RSA and DSA.

Because of shorter operands, based on 32-bit systems, ECC may not benefit as much of pSHS as RSA and DSA. However, in many low-end implementations where the word length of the cores is 8 bits [135] [136], pSHS can still be a good candidate for acceleration.

Ideally, pSHS requires s to be dividable by $p*q$. In some cases, this requirement cannot be satisfied. An easy solution is to extend the operands with 0 until the requirement is satisfied. Also, as the number of cores (p) increases to be equal to s , we can hardly gain additional speedup by adding more cores for computation. Even so, pSHS already shows a good scalability. For 2048-bit RSA, this limit number of p is 64 for 32-bit cores and 256 for 8-bit cores.

Compared with the parallelization obtained by concurrently performing multiple encryptions, whose ideal speedup could be linear, pSHS trades some throughput for much lower latency. Future work will investigate the combination of these two methods to find suitable tradeoffs between throughput and latency for different applications.

8.7 Conclusion

In this chapter, we propose pSHS as a parallel programming scheme for Montgomery multiplication based on multicore systems. Both analysis and experiments on real multicore prototypes show that pSHS provides a good speedup, large communication latency tolerance, good portability and good scalability. These features make pSHS a good parallel software solution for RSA, DSA, and ECC on multicore systems.

Chapter 9

Conclusions

In this dissertation, we have presented our work on designing SCA-resistant and high-performance cryptography on embedded systems. As embedded systems become ubiquitous in our daily lives, we have been dependent on these devices more than ever before. Meanwhile, the requirements for low cost and high performance have never been alleviated. This makes it a necessity to design dependable, trustworthy, and fast embedded systems. Our research fits well into this purpose since SCA-resistant and high-performance cryptography is a cornerstone to build such embedded devices. SCA has become a great concern to embedded security due to its low cost and high effectiveness. Faster processing of cryptography not only brings us higher performance, but also allows designers to choose more secure cryptographic algorithms for embedded applications. Therefore, we see a strong motivation and future application of our research.

We focus on improving embedded cryptographic software, achieving our goal by working on microarchitectures of embedded processors as well as on programming methods. The reason why we follow this approach is that our investigation on a SCA countermeasure, called masking, tells us that simply using secure transformation of cryptographic algorithms or secure logic circuits is not going to give us satisfying results. Working on microarchitectures, which connect software and hardware, is easy to make trade-offs between different aspects. In such a way, we hit the sweet spot in the design space.

We make use of two state-of-the-art microarchitectures for embedded processors, including instruction set extension and multi-core architecture. For SCA-resistance, we propose a solution concept called Virtual Secure Circuit, which emulates the behavior of a secure DRP circuit with microprocessors. Based on that, we implement this concept with both dual-core processors and special instruction set extensions attached to single-core processors. After three steps of research, we finally find a solution based on instruction set extensions that performs well across the board, including security, cost, and performance. We also note that the dual-core based Virtual Secure Circuit also delivers good SCA-resistance if electromagnetic attack is not a concern. For performance, we propose a low-latency programming method for Montgomery multiplication. This solution helps us to reduce the execution latency of one single RSA task in a scalable and efficient way, which makes it perfectly suitable for embedded systems where only one cryptographic task exists at the same time.

Compared with related works, our SCA-resistant solution is the only one that does not have obvious disadvantage. Others are either not secure enough, or suffer from high hardware costs, or have very poor performance. All these disadvantages are barriers that prevent them from practical applications. On the contrary, our SCA-resistant solution is flexible and performs well in security, performance, and cost. Therefore, it is more practical than others. In addition, DRP technique has been demonstrated to be effective to detect fault attacks [137, 138]. Our solution, which is a software version of DRP, also inherits this feature. With all the above good features, we are confident that our solution is able to offer good protection to cryptography in future secure embedded products.

To make our SCA-resistant solution a product, we expect the following refinements to be done. First, programming of the secure processor with a secure pipeline is currently done manually. This is inconvenient for programmers. Thus, automation is needed to identify sensitive intermediate variables and to convert a program for different processor with different widths of pipelines. Second, our SCA-resistant solution already achieved a high SCA-resistance. Further improvement may come from solving the early evaluation

problem [41, 139]. Some existing solutions at circuit level [140, 141, 41, 97] may give us clues. Third, additional mechanisms at the protocol level may enhance the security further. For example, assume an attack takes one day to break an unprotected system. To break our solution, it takes more than one year ($> 1000x$ improvement). This already prevents a lot of attackers. To further improve security, we can update the secret key every year. This protocol brings the solution to a even higher security level. Finally, an optional improvement can be done is to integrate pre-charge behaviors to hardware. This will save the pre-charge instructions. The software footprints will be at least halved.

Our parallel solution to Montgomery multiplication, for the first time, shows that a good programming is able to partition a cryptographic algorithm in an efficient and scalable manner, with a high tolerance to the inter-core communication latency. This tells us that reducing the execution latency is viable through multi-core parallel programming. We expect this approach to work for other algorithms. Considering most other cryptographic algorithms have much lighter computational load, the performance after partitioning may be more sensitive to the delay of inter-core communications.

In order to shorten the execution latency with parallel programming for other algorithms, we consider the following two points would help us. First, shorten the communication delay from the hardware perspective. This is already a design target for multi-core processor design. Second, integrate the cryptographic software as primitives to the operating systems. User-level threads usually do not have direct access to inter-core communication. They have to call system APIs for this purpose, which consumes a lot of time. Integrating these parallel programs to the OS can make the communication more efficient. The cost of doing this is low because embedded systems only use a limited number of cryptographic algorithms.

In the end, we want to mention that cryptography is only one component of embedded security. Designing secure systems requires proper integration of cryptography, protocol, and application. We hope that our contributions will offer us solid components for such a process.

Publications

Most of the above discussions have been published in journals, conference proceedings, technical reports. In addition to the story told in this dissertation, we also did some related research, such as automation of SCA-resistant design flow, system integration of cryptographic primitives, and so on. Details can be found from the following list of publications.

1. Z. Chen, A. Sinha, and P. Schaumont, "Using Virtual Secure Circuit to Protect Embedded Software from Side Channel Attacks," IEEE Transactions on Computers, (under review).
2. Z. Chen and P. Schaumont, "A Parallel Implementation of Montgomery Multiplication on Multi-core Systems: Algorithm, Analysis, and Prototype," IEEE Transactions on Computers, (to appear).
3. Z. Chen, X. Guo, A. Sinha, and P. Schaumont, "Data-Oriented Performance Analysis of SHA-3 Candidates on FPGA Accelerated Computers," Design, Automation and Test in Europe (DATE2011).
4. A. Sinha, Z. Chen, and P. Schaumont, "A Comprehensive Analysis of Performance and Side-Channel-Leakage of AES SBOX Implementations in Embedded Software," Workshop on Embedded Systems Security (WESS2010), 2010.
5. Z. Chen, A. Sinha and P. Schaumont, "Implementing Virtual Secure Circuit Using

- A Custom-Instruction Approach,” International Conference on Compilers Architecture and Synthesis for Embedded Systems (CASES 2010).
6. Z. Chen, and P. Schaumont, ”Virtual Secure Circuit: Porting Dual-Rail Pre-charge Technique into Software on Multicore,” IACR ePrint Archive 2010/272, April 2010.
 7. Z. Chen, and P. Schaumont, ”pSHS: A Scalable Parallel Software Implementation of Montgomery Multiplication for Multicore Systems,” Design, Automation and Test in Europe (DATE2010), pp. 843-848, 2010.
 8. Z. Chen, R. N. Pittman, A. Forin, ”Combining Multi-core and Reconfigurable instruction Set Extensions,” ACM SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA2010), pp. 33-36, 2010.
 9. Z. Chen, R. N. Pittman, A. Forin, ”MultiCore eMIPS,” Microsoft Research Technical Report MSR-TR-2009-113, 2009.
 10. Z. Chen, and P. Schaumont, ”Early Feedback on Side-Channel Risks and Accelerated Toggle Counting,” IEEE International Workshop on Hardware-Oriented Security and Trust (HOST2009), pp 90-95, 2009.
 11. Z. Chen, S. Haider, and P. Schaumont, ”Side-Channel Leakage in Masked Circuits Caused by Higher-Order Circuit Effects,” International Conference on Information Security and Assurance (ISA2009) , pp 327-336, 2009.
 12. Z. Chen, R. Nagesh, A. Reddy, and P. Schaumont, ”Increasing the Sensitivity of On-Chip Digital Thermal Sensors with Pre-Filtering,” IEEE Computer Society Annual Symposium on VLSI (ISVLSI2009), 2009.
 13. Z. Chen, X. Guo, R. Nagesh, A. Reddy, M. Gora, and A. Maiti, ”Hardware Trojan Designs on BASYS FPGA Board,” Embedded System Challenge Contest in Cyber Security Awareness Week (CSAW2008), 2008.

14. Z. Chen, S. Morozov, and P. Schaumont, "A Hardware Interface for Hashing Algorithm," ePrint IACR Archive, 2008/529, 2008.
15. Z. Chen, and P. Schaumont, "Improving Secure Hardware Masking Using an Equalization Technique," Workshop on Embedded Systems Security (WESS2008), 2008.
16. Z. Chen, and P. Schaumont, "Slicing Up a Perfect Hardware Masking Scheme," Workshop on Hardware-Oriented Security and Trust(HOST2008), 2008.
17. X. Guo, Z. Chen, and P. Schaumont, "Energy and Performance Evaluation of an FPGA-based SoC Platform with AES and PRESENT Coprocessors," Workshop on Systems, Architectures, Modeling, and Simulation 2008.
18. Z. Chen, and Y. Zhou, "Dual-Rail Random Switching Logic: A Countermeasure to Reduce Side-Channel Leakage," Workshop on Cryptographic Hardware and Embedded System (CHES2006), 2006.

Bibliography

- [1] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, “Security in Embedded Systems: Design Challenges,” *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 3, pp. 461 – 491, 2004.
- [2] W. Stallings, *Network and Internetwork Security: Principles and Practice*. Prentice Hall, 1995. ISBN:0-02-415483-0.
- [3] B. Schneier, *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley and Sons, New York, 1996.
- [4] W. Stallings, *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 1998.
- [5] National Bureau of Standards, “Data Encryption Standard,” *U.S. Department of Commerce, FIPS pub. 180*, 1993.
- [6] NIST, “AES: Advanced Encryption Standard,” available at <http://csrc.nist.gov/CryptoToolkit/aes/>.
- [7] R. L. Rivest, A. Shamir, and L. Adleman, “A Method for obtaining Digital Signatures and Public Key Cryptosystems,” *Communications of the ACM*, vol. 21, pp. 120–126, 1978.
- [8] National Institute of Standards and Technology (NIST), “Digital Signature Standard (FIPS 186-2),” 2000.

- [9] N. Koblitz, “Elliptic Curve Cryptosystems,” *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987.
- [10] R. L. Rivest, “The MD5 Message-Digest Algorithm,” *Request for Comments (RFC)*, vol. 1320, 1992.
- [11] NIST, “Secure Hash Standard,” *Federal Information Processing Standard, FIPS-181*, 1995.
- [12] NIST, “First Round Candidates of the SHA-3 Hash Function Competition,” available at http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_md1.html.
- [13] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov, “Cryptographic Processors - A Survey,” *Proceedings of the IEEE*, vol. 94, no. 2, pp. 357 – 369, 2006.
- [14] P. C. Kocher, J. Jaffe, and B. Jun, “Differential Power Analysis,” *In proceedings of Advances in Cryptography - CRYPTO, LNCS*, vol. 1666, pp. 388 – 397, 1999.
- [15] J.-J. Quisquater and D. Samyde, “ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards,” *In proceedings of Smart Card Programming and Security, LNCS*, vol. 2140, pp. 200 – 210, 2001.
- [16] P. Kocher, “Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems,” *In proceedings of Advances in Cryptography - CRYPTO*, vol. 1109, pp. 104 – 113, 1996.
- [17] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems, “A Practical Implementation of the Timing Attack,” *In proceedings of Smart Card Research and Applications, LNCS*, vol. 1820, pp. 167 – 182, 2000.
- [18] T. C. Group, “TCG Mobile Reference Architecture,” 2010. available at <http://www.trustedcomputinggroup>.

org/files/temp/644597BE-1D09-3519-AD5ADDAFA0B539D2/MPWG%
20tcg-mobile-reference-architecture-1.pdf.

- [19] A. Royo, J. Morán, and J. C. López, “Design and Implementation of a Coprocessor for Cryptography Applications,” *In proceedings of European Conference on Design and Test (EDTC)*, pp. 213 – 217, 1997.
- [20] M. K. Hani, T. S. Lin, and N. Shaikh-Husin, “FPGA Implementation of RSA Public-Key Cryptographic Coprocessor,” *In proceedings of TENCON*, vol. 3, pp. 6 – 11, 2000.
- [21] S. Okada, N. Torii, K. Itoh, and M. Takenaka, “Implementation of Elliptic Curve Cryptographic Coprocessor over $GF(2^m)$ on an FPGA,” *In Proceedings of Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 1965, pp. 215 – 242, 2000.
- [22] M. Lindemann and S. W. Smith, “Improving DES Coprocessor Throughput for Short Operations,” *In Proceedings of 10th USENIX Security Symposium*, pp. 67 – 81, 2001.
- [23] E. Trichina, T. Korkishko, and K. H. Lee, “Small Size, Low Power, Side Channel-Immune AES Coprocessor Design and Synthesis Results,” *In Proceedings of Advanced Encryption Standard - AES, LNCS*, vol. 3373, pp. 113 – 127, 2005.
- [24] J. Nick L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh, “Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor,” *In Proceedings of USENIX Security Symposium*, pp. 179 – 194, 2004.
- [25] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, and L. van Doorn, “Building the IBM 4758 Secure Coprocessor,” *IEEE Computer*, vol. 34, no. 10, pp. 57 – 66, 2001.

- [26] S. Tillich and J. Großschädl, “Accelerating AES Using Instruction Set Extensions for Elliptic Curve Cryptography,” *In Proceedings of Computational Science and Its Applications - ICCSA, LNCS*, pp. 665 – 675, 2005.
- [27] Tensilica Inc., “Xtensa 7 Product Brief,” available at http://www.tensilica.com/uploads/pdf/xtensa_7.pdf.
- [28] R. N. Pittman, N. L. Lynch, and A. Forin, “eMIPS, A Dynamically Extensible Processor,” *Microsoft Research Technical Report MSR-TR-2006-143*, 2006. available at <http://research.microsoft.com/enus/projects/emips/emipsreport1.pdf>.
- [29] J. Großschädl, S. Tillich, and A. Szekely, “Performance Evaluation of Instruction Set Extensions for Long Integer Modular Arithmetic on a SPARC V8 Processor,” *In Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD)*, pp. 680 – 689, 2007.
- [30] S. Tillich and J. Großschädl, “Instruction Set Extensions for Efficient AES Implementation on 32-bit Processors,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 4249, pp. 270 – 284, 2006.
- [31] J. Großschädl, S. Tillich, P. Inenne, L. Pozzi, and A. K. Verma, “When Instruction Set Extensions Change Algorithm Design: A Study in Elliptic Curve Cryptography,” *In Proceedings of 4th Workshop on Application Specific Processors (WASP)*, pp. 2 – 9, 2005.
- [32] S. Kumar and C. Paar, “Reconfigurable Instruction Set Extension for Enabling ECC on an 8-Bit Processor,” *In Proceedings of Field Programmable Logic and Application, LNCS*, vol. 3203, pp. 586 – 595, 2004.
- [33] K. Nadehara, M. Ikekawa, and I. Kuroda, “Extended Instructions for The AES Cryptography and Their Efficient Implementation,” *In Proceedings of IEEE Workshop on Signal Processing Systems (SIPS)*, pp. 152 – 157, 2004.

- [34] M. E. Kaihara and N. Takagi, “Bipartite Modular Multiplication,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 3659, pp. 201 – 210, 2005.
- [35] M. E. Kaihara and N. Takagi, “Bipartite Modular Multiplication Method,” *IEEE Transactions on Computers*, vol. 57, no. 2, pp. 157 – 164, 2008.
- [36] K. Sakiyama, M. Knežević, J. Fan, B. Preneel, and I. Verbauwhede, “Tripartite modular multiplication,” *COSIC internal report*, 2009.
- [37] J.-C. Bajard, L.-S. Didier, and P. Kornerup, “An RNS Montgomery Modular Multiplication Algorithm,” *IEEE Transactions on Computers*, vol. 47, no. 7, pp. 766 – 776, 1998.
- [38] K. Tiri and I. Verbauwhede, “A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation,” *In Proceeding of Design, Automation, and Test in Europe (DATE)*, pp. 246 – 251, 2004.
- [39] T. Popp and S. Mangard, “Masked Dual-Rail Pre-charge Logic: DPA Resistance without the Routing Constraints,” *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 3659, pp. 172 – 186, 2005.
- [40] D. Suzuki, M. Saeki, and T. Ichikawa, “Random Switching Logic: A countermeasure against DPA based on Transition Probability,” *Cryptology ePrint Archive Report 2004/346*, 2004.
- [41] Z. Chen and Y. Zhou, “Dual-Rail Random Switching Logic: A Countermeasure to Reduce Side-Channel Leakage,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 4249, pp. 242 – 254, 2006.
- [42] P. Yu and P. Schaumont, “Secure FPGA Circuits Using Controlled Placement and Routing,” *In Proceedings of 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, pp. 45 – 50, 2007.

- [43] K. Tiri and I. Verbauwhede, “Securing Encryption Algorithms against DPA at the Logic Level: Next Generation Smart Card,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 2779, pp. 125–136, 2003.
- [44] S. Guilley, P. Hoogvorst, Y. Mathieu, R. Pacalet, and J. Provost, “CMOS Structures Suitable for Secured Hardware,” *In Proceedings of Design, Automation, and Test in Europe (DATE)*, pp. 1414 – 1415, 2004.
- [45] M. W. Allam and M. I. Elmasry, “Dynamic Current Mode Logic (DyCML): A New Low-Power High-Performance Logic Style,” *IEEE Journal of Solid-State Circuits*, vol. 36, no. 3, pp. 550 – 558, 2001.
- [46] I. Hassoune, F. Mace, D. Flandre, and J.-D. Legat, “Dynamic Differential Self-Timed Logic Families for Robust and Low-Power Security ICs,” *Integration, the VSLI Journal*, vol. 40, no. 3, pp. 355 – 364, 2006.
- [47] S. Tillich and J. Großschädl, “Power-Analysis Resistant AES Implementation with Instruction Set Extensions,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 4727, pp. 303 – 319, 2007.
- [48] S. Tillich, M. Kirschbaum, and A. Szekely, “SCA-Resistant Embedded Processors - the Next Generation,” *In Proceedings of 26th Annual Computer Security Applications Conference (ACSAC)*, pp. 211 – 220, 2010.
- [49] J. Blömer and J. Guajardo and V. Krummel, “Provably Secure Masking of AES,” *In Proceedings of Workshop on Selected Areas in Cryptography (SAC), LNCS*, vol. 3357, pp. 69 – 83, 2005.
- [50] E. Oswald, S. Mangard, N. Pramstaller, and V. Rijmen, “A Side-Channel Analysis Resistant Description of the AES S-Box,” *In Proceedings of Workshop on Fast Software Encryption, LNCS*, vol. 3557, pp. 413 – 423, 2005.

- [51] M.-L. Akkar and C. Giraud, “An Implementation of DES and AES, Secure against Some Attacks,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 2162, pp. 309 – 318, 2001.
- [52] E. Oswald and K. Schramm, “An Efficient Masking Scheme for AES Software Implementations,” *In Proceeding of Information Security Applications (ISA), LNCS*, vol. 3786, pp. 292 – 305, 2006.
- [53] K. Schramm and C. Paar, “Higher Order Masking of the AES,” *Topics in Cryptography - CT-RSA, LNCS*, vol. 3860, pp. 208 – 225, 2006.
- [54] M. Rivain, E. Prouff, and J. Doget, “Higher-Order Masking and Shuffling for Software Implementation of Block Ciphers,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems, LNCS*, vol. 5747, pp. 171 – 188, 2009.
- [55] M. Rivain and E. Prouff, “Provably Secure Higher-Order Masking of AES,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems, LNCS*, vol. 6225, pp. 413 – 427, 2010.
- [56] Z. Chen and P. Schaumont, “Slicing Up a Perfect hardware Masking Scheme,” *In Proceedings of Workshop on Hardware-Oriented Security and Trust (HOST)*, pp. 21 – 25, 2008.
- [57] Z. Chen, S. Haider, and P. Schaumont, “Side-Channel Leakage in Masked Circuits Caused by Higher-Order Circuit Effects,” *In Proceedings of International Conference on Information Security and Assurance (ISA)*, pp. 304 – 309, 2009.
- [58] Z. Chen and P. Schaumont, “Virtual Secure Circuit: Porting Dual-Rail Pre-charge Technique into Software on Multicore,” *Cryptology ePrint Archive Report 2010/272*, 2010.
- [59] Z. Chen, A. Sinha, and P. Schaumont, “Implementing Virtual Secure Circuit Using a Custom-Instruction Approach,” *In Proceedings of International Conference on*

- Compilers, Architecture and Synthesis for Embedded Systems, (CASES)*, pp. 57 – 66, 2010.
- [60] Z. Chen and P. Schaumont, “pSHS: A Scalable Parallel Software Implementation of Montgomery Multiplication for Multicore Systems,” *In Proceedings of Design, Automation, and Test in Europe (DATE)*, pp. 843 – 848, 2010.
- [61] Z. Chen and P. Schaumont, “A Parallel Implementation of Montgomery Multiplication on Multi-core Systems: Algorithm, Analysis, and Prototype,” *IEEE Transactions on Computers*, (to appear).
- [62] E. Brier, C. Clavier, and F. Olivier, “Correlation Power Analysis with a Leakage Model,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 3156, pp. 16 – 29, 2004.
- [63] B. Gierlichs, L. Batina, P. Tuyls, and B. Preneel, “Mutual Information Analysis A Generic Side-Channel Distinguisher,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 5154, pp. 426 – 442, 2008.
- [64] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, “Towards Sound Approaches to Counteract Power-Analysis Attacks,” *In Proceedings of Advances in Cryptography - CRYPTO, LNCS*, vol. 1666, pp. 398 – 412, 1999.
- [65] L. Goubin and J. Patarin, “DES and Differential Power Analysis The “Duplication” Method,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 1717, pp. 158 – 172, 1999.
- [66] D. May, H. L. Muller, and N. P. Smart, “Non-deterministic Processors,” *In Proceeding of Information Security and Privacy, LNCS*, vol. 2119, pp. 115 – 129, 2001.

- [67] J. A. Ambrose, R. G. Ragel, and S. Parameswaran, “A Smart Random Code Injection to Mask Power Analysis Based Side Channel Attacks,” *Proceedings of the 5th IEEE/ACM international conference on hardware/software codesign and system synthesis*, pp. 51 – 56, 2007.
- [68] C. Clavier, J.-S. Coron, and N. Dabbous, “Differential Power Analysis in the Presence of Hardware Countermeasures,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 1965, pp. 252 – 263, 2000.
- [69] S. Tillich and C. Herbst, “Attacking State-of-Art Software Countermeasures - A Case Study for AES,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 5154, pp. 228 – 243, 2008.
- [70] B. Gierlichs, “DPA-Resistance Without Routing Constraints?,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 4727, pp. 107 – 120, 2007.
- [71] K. Tiri and P. Schaumont, “Changing the Odds Against Masked Logic,” *In Proceedings of Workshop on Selected Areas in Cryptography (SAC), LNCS*, vol. 4356, pp. 137 – 146, 2007.
- [72] P. Schaumont and K. Tiri, “Masking and Dual-Rial Logic Don’t Add Up,” *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 4727, pp. 95 – 106, 2007.
- [73] T. S. Messerges, “Using Second-Order Power Analysis to Attack DPA Resistant Software,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 1965, pp. 27 – 78, 2000.
- [74] K. Okeya and K. Sakurai, “A Second-Order DPA Attack Breaks a Window-Method Based Countermeasure against Side Channel Attacks,” *In Proceedings of the 5th International Conference on Information Security (ISC)*, pp. 389 – 401, 2002.

- [75] J. Waddle and D. Wagner, “Towards Efficient Second-Order Power Analysis,” *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, LNCS, vol. 3156, pp. 1 – 15, 2004.
- [76] E. Peeters, F.-X. Standaert, N. Donckers, and J.-J. Quisquater, “Improved Higher-Order Side-Channel Attacks with FPGA Experiments,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, LNCS, vol. 3659, pp. 309 – 323, 2005.
- [77] E. Oswald, S. Mangard, C. Herbst, and S. Tillich, “Practical Second-order DPA Attacks for Masked Smart Card Implementations of Block Ciphers.,” *In Proceeding of CT-RSA*, LNCS, vol. 3860, pp. 192 – 207, 2006.
- [78] S. Mangard, N. Pramstaller, and E. Oswald, “Successfully Attacking Masked AES hardware Implementations,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, LNCS, vol. 3659, pp. 157 – 171, 2005.
- [79] S. Mangard and K. Schramm, “Pinpointing the Side-Channel Leakage of Masked AES Hardware Implementation,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, LNCS, vol. 4249, pp. 76 – 90, 2006.
- [80] K. Tiri, D. Hwang, A. Hodjat, B. Lai, S. Yang, P. Schaumont, and I. Verbauwhede, “Prototype IC with WDDL and Differential Routing - DPA Resistant Assessment,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, LNCS, vol. 3659, pp. 354 – 365, 2005.
- [81] S. Guilley, L. Sauvage, P. Hoogvorst, R. Pacalet, G. M. Bertoni, and S. Chaudheri, “Security Evaluation of WDDL and SecLib Countermeasures against Power Attacks,” *IEEE Transactions on Computers*, vol. 57, no. 11, pp. 1482 – 1497, 2008.
- [82] MIPS Technologies Inc., “Pro Series Processor Cores,” available at http://www.mips.com/media/files/Pro_Series.pdf.

- [83] ARC Inc., “DesignWare ARC 700 Family Brochure,” available at http://www.synopsys.com/dw/doc.php/ds/cc/arc_700.pdf.
- [84] andrea Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegollo, and roberto Guerrieri, “A VLIW Processor With Reconfigurable Instruction Set for Embedded Applications,” *IEEE Journal of Solid-State circuits*, vol. 38, no. 11, pp. 1876 – 1886, 2003.
- [85] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. 3rd ed., 2003. ISBN: 1558607242.
- [86] R. E. Gonzalez, “Xtensa: A Configurable and Extensible Processor,” *IEEE Micro*, vol. 20, no. 2, pp. 60 – 70, 2000.
- [87] H. P. Hofstee, “Power Efficient Processor Architecture and The Cell Processor,” *In Proceedings of 11th International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 258 – 262, 2005.
- [88] Intel, “Dual-Core Intel Xeon Processor 5000 Series,” 2006. available at <http://www.intel.com/assets/PDF/datasheet/313079.pdf>.
- [89] ARM, “ARM11 MPCore Processor Technical Reference Manual,” 2008. available at http://infocenter.arm.com/help/topic/com.arm.doc.ddi0360f/DDI0360F_arm11_mpcore_r2p0_trm.pdf.
- [90] IBM DeveloperWorks, “Cell broadband engine programming handbook (version 1.1),” 2005. available at <http://www.ibm.com/developerworks/power/library/pa-cellperf/>.
- [91] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. M. C.-C. Miao, J. F. Brown, and A. Agarwal, “On-Chip Interconnection Architecture of the Tile Processor,” *IEEE Micro*, vol. 27, no. 5, pp. 15–31, 2007.

- [92] J. Stokes, “Introducing the IBM/Sony/Toshiba Cell Processor – Part II: The Cell Architecture,” available at <http://arstechnica.com/old/content/2005/02/cell-2.ars>.
- [93] P. Schaumont and I. Verbauwhede, “A Component-based Design Environment for Electronic System-Level Design,” *IEEE Design and Test of Computers, special issue on Electronic System-Level Design*, vol. 23, pp. 338 – 347, 2006.
- [94] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.
- [95] J. Rabaey, A. Chanadrakasan, and B. Nikolic, *Digital Integrated Circuits: A Design Perspective*. Prentice Hall, 2nd ed., 2003. ISBN: 0-13-090996-3.
- [96] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley, 3rd ed., 2005. ISBN-10 0-13-607694-7.
- [97] T. Popp, M. Kirschbaum, T. Zefferer, and S. Mangard, “Evaluation of the Masked Logic Style MDPL on a Prototype Chip,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 4727, pp. 81 – 94, 2007.
- [98] J.-S. Coron, E. Prouff, and M. Rivain, “Side Channel Cryptanalysis of a Higher Order Masking Scheme,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 4727, pp. 28 – 44, 2007.
- [99] S. Keckler, K. Olukotun, and P. H. Hofstee, *Multicore Processors and Systems*. Integrated Circuits and Systems, Springer, 2009.
- [100] J. Nurmi, *Processor Design*. 10.1007/978-1-4020-5530-0, Springer Netherlands, 2007.
- [101] J. A. Ambrose, S. Parameswaran, and A. Ignjatovic, “MUTE-AES: A Multiprocessor Architecture to Prevent Power Analysis Based Side Channel Attack of the AES

- Algorithm,” *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pp. 678 – 684, 2008.
- [102] P. L. Montgomery, “Speeding the Pollard and Elliptic Curve Methods of Factorization,” *Mathematics of Computation*, vol. 48 (177), pp. 243 – 264, 1987.
- [103] Aeroflex Gaisler, “LEON3 Multiprocessing CPU Core,” available at http://www.gaisler.com/doc/leon3_product_sheet.pdf.
- [104] E. Biham, “A Fast New DES Implementation in Software,” *In Proceeding of Fast Software Encryption (FSE)*, LNCS, vol. 1267, pp. 260 – 272, 1997.
- [105] E. Käsper and P. Schwabe, “Faster and Timing-Attack Resistant AES-GCM,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, LNCS, vol. 5747, pp. 1 – 17, 2009.
- [106] R. Könighofer, “A Fast and Cache-timing Resistant Implementation of The AES,” *In Proceeding of CT-RSA*, LNCS, vol. 4964, pp. 187 – 202, 2008.
- [107] M. Matsui and J. Nakajima, “On the Power of Bitslice Implementation on Intel Core2 Processor,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, LNCS, vol. 4727, pp. 121 – 134, 2007.
- [108] D. J. Bernstein, “Batch binary edwards,” *In Proceedings of Advances in Cryptography - CRYPTO*, LNCS, vol. 5677, pp. 317 – 336, 2009.
- [109] F. Regazzoni, A. Cevrero, F.-X. Standaert, S. Badel, T. Kluter, P. Brisk, Y. Leblebici, and P. Ienne, “A Design Flow and Evaluation Framework for DPA-Resistant Instruction Set Extensions,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, LNCS, vol. 5747, pp. 205 – 219, 2009.
- [110] ARM Ltd., “Trustzone technology overview,” available at <http://www.arm.com/products/security/>.

- [111] Xilinx, “PlanAhead user guide,” available at http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/PlanAhead_UserGuide.pdf.
- [112] C. Herbst, E. Oswald, and S. Mangard, “An AES Smart Card Implementation Resistant to Power Analysis Attacks,” *In Proceedings of Applied Cryptography and Network Security, LNCS*, vol. 3989, pp. 239 – 252, 2006.
- [113] J. Irwin, D. Page, and N. P. Smart, “Instruction Stream Mutation for Non-Deterministic Processors,” *In Proceeding of 13th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 286 – 295, 2002.
- [114] P. Grabher, J. Großschädl, and D. Page, “Non-deterministic Processors: FPGA-Based Analysis of Area, Performance and Security,” *In Proceedings of the 4th Workshop on Embedded Systems Security (WESS)*, pp. 1 – 10, 2009.
- [115] Aeroflex Gaisler, “LEON3/GRLIB SOC IP Library,” available at <http://www.gaisler.com/doc/Leon3%20Grlib%20folder.pdf>.
- [116] D. Geer, “Chip Makers Turn to Multicore Processors,” *Computer*, vol. 38, no. 5, pp. 11–13, 2005.
- [117] AMD, “Six-Core AMD Opteron Processor Product Brief.” available at <http://www.amd.com/us/products/server/processors/six-core-opteron/Pages/six-core-opteron-product-brief.aspx>.
- [118] P. L. Montgomery, “Modular Multiplication without Trial Division,” *Mathematics of Computation*, vol. 44, no. 170, pp. 519 – 521, 1985.
- [119] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr., “Analyzing and Comparing Montgomery Multiplication Algorithms,” *IEEE Micro*, vol. 16, no. 3, pp. 26 – 33, 1996.
- [120] N. Costigan and P. Schwabe, “Fast Elliptic-Curve cryptography on the Cell Broadband Engine,” *In Proceeding of Africa Crypt, LNCS*, vol. 5580, pp. 368 – 385, 2009.

- [121] R. Szerwinski and T. Güneysu, “Exploiting the Power of GPUs for Asymmetric Cryptography,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 5154, pp. 79 – 99, 2008.
- [122] A. Moss, D. Page, and N. P. Smart, “Toward Acceleration of RSA Using 3D Graphics Hardware,” *In Proceeding of Cryptography and Coding, LNCS*, vol. 4487, pp. 213 – 220, 2007.
- [123] S. Fleissner, “GPU-Accelerated Montgomery Exponentiation,” *In Proceedings of International Conference on Computational Science (ICCS), LNCS*, vol. 4487, p-p. 213 – 220, 2007.
- [124] N. Costigan and M. Scott, “Accelerating SSL using the Vector processors in IBM’s Cell Broadband Engine for Sonys Playstation 3,” in *In Proceedings of the SPEED Workshop*, 2009. <http://www.hyperelliptic.org/SPEED>, accessed 11/2009.
- [125] K. Sakiyama, L. Batina, B. Preneel, , and I. Verbauwhede, “Multicore Curve-Based Cryptoprocessor with Reconfigurable Modular Arithmetic Logic Units over $GF(2^n)$,” *IEEE Transactions on Computers*, vol. 56, no. 9, pp. 1269 – 1282, 2007.
- [126] J. Fan, K. Sakiyama, and I. Verbauwhede, “Elliptic curve cryptography on embedded multicore systems,” *Design Automation for Embedded Systems*, vol. 12, no. 3, pp. 231 – 242, 2008.
- [127] J. Fan, K. Sakiyama, and I. Verbauwhede, “Montgomery Modular Multiplication Algorithm for Multi-Core Systems,” *In Proceedings of IEEE Workshop on Signal Processing Systems*, pp. 261 – 266, 2007.
- [128] B. Baldwin, W. P. Maranane, and R. Granger, “Reconfigurable Hardware Implementation of Arithmetic Modulo Minimal Redundancy Cyclotomic Primes for ECC,” *In Proceedings of International Conference on Reconfigurable Computing and FPGAs*, pp. 255 – 260, 2009.

- [129] K. Sankaralingam, R. Nagarajan, R. Desikan, S. Drolia, M. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger, “Distributed Microarchitectural Protocols in the TRIPS Prototype Processor,” *Proceedings of the 39th Annual International Symposium on Microarchitecture*, pp. 480–491, 2006.
- [130] M. Baron, “Low-key Intel 80-core Intro: The Tip of the Iceberg,” *Microprocessor Report*, 2007.
- [131] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpfen, M. Frank, S. Amarasinghe, and A. Agarwal, “The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs,” *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.
- [132] C. D. Walter, “Montgomery Exponentiation Needs No Final Subtractions,” *Electronics Letters*, vol. 35, no. 21, pp. 1831 – 1832, 1999.
- [133] A. Karatsuba and Y. Ofman, “Multiplication of Many-Digital Numbers by Automatic Computers,” *In Proceedings of the USSR Academy of Sciences*, vol. 145, pp. 293 – 294, 1962.
- [134] Intel, “Intel Single-Chip Cloud Computers,” *available at <http://techresearch.intel.com/articles/Tera-Scale/1826.htm>*.
- [135] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, “Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 3156, pp. 119 – 132, 2004.
- [136] M. Koschuch, J. Lechner, A. Weitzer, J. Großschädl, A. Szekely, S. Tillich, and J. Wolkerstorfer, “Hardware/Software Co-design of Elliptic Curve Cryptography

- on an 8051 Microcontroller,” *In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 4249, pp. 430 – 444, 2006.
- [137] S. Bhasin, J.-L. Danger, F. Flament, T. Graba, S. Guilley, Y. Mathieu, M. Nassar, L. Sauvage, and N. Selmane, “Combined SCA and DFA Countermeasures Integrable in a FPGA Design Flow,” *In Proceedings of International Conference on ReConfigurable Computing and FPGAs (ReConfig)*, pp. 213 – 218, 2009.
- [138] N. Selmane, S. Bhasin, S. Guilley, T. Graba, and J.-L. Danger, “WDDL is Protected Against Setup Time Violation Attacks,” *In Proceedings of Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 73 – 83, 2009.
- [139] D. Suzuki and M. Saeki, “Security Evaluation of DPA Countermeasures Using Dual-Rail Pre-charge Logic Style,” *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS*, vol. 4249, pp. 255 – 269, 2006.
- [140] S. Bhasin, S. Guilley, F. Flament, N. Selmane, and J.-L. Danger, “Countering Early Evaluation: An Approach Towards Robust Dual-Rail Precharge Logic,” *In Proceedings of the 5th Workshop on Embedded Systems Security (WESS)*, pp. 1 – 8, 2010.
- [141] M. Nassar, S. Bhasin, J.-L. Danger, G. Duc, and S. Guilley, “BCDL: A High Speed Balanced DPL for FPGA with Global Precharge and No Earl Evaluation,” *In Proceedings of Design Automation for Embedded Systems (DATE)*, pp. 849 – 855, 2010.