Formal Design of Cryptographic Hardware

Tohoku University

Graduate School of Information Sciences

Formal Design of Cryptographic Hardware

Rei Ueno

Supervisor:
Prof. Takafumi Aoki
Prof. Naofumi Homma

Dissertation presented in
partial fulfillment of the
requirements for the degree
of ***Doctor of Philosophy***

Formal Design of Cryptographic Hardware

Rei Ueno

Acknowledgments

I wrote this thesis as the summarizing compilation of my study at Aoki Laboratory. Although the road was steep and complicated, many amazing people led me to a better direction with their support, which made me possible to reach such a honorable goal (and a starting point of new road as a researcher).

First of all, I would like to show the greatest appreciation to my supervisor Professor Takafumi Aoki. He has greatly encouraged me in my learning and study, and has always trusted me even when I had made mistakes and failures. He has also taught me the importance of challenging myself to everything, and let me find delight in study, research, and modern computer games.

I am awfully grateful to Professor Hiroki Shizuya and Professor Masanori Hariyama. I could surely improve my thesis thanks to their insightful and valuable comments.

I owe a very important debt of gratitude to my supervisor Professor Naofumi Homma. I could not have written this thesis as readily as I had done unless he had supervised me. I have always had the honor of asking him about both research and off-research problems, which was essential for pursuing my daily study and living. I believe that what he taught should offer the promise of a brilliant future for me.

I have really enjoyed collaborative research and discussions with many outstanding researchers, which brought expertised knowledge and a remarkable outcome to me. While it would be hard to count everyone, a part of the people who especially helped me write my thesis and journal/conference papers is listed here with great appreciation: Dr. Sumio Morioka, Professor Yasuyuki Nogami, Dr. Kazuhide Fukushima, Dr. Shinsaku Kiyomoto, Dr. Yuto Nakano, Professor Jean-Luc Danger, Professor Sylvain Guilley, Dr. Yves Mathieu, Professor Makoto Nagata, Dr. Noriyuki Miura, Professor Kazuo Sakiyama, Dr. Takeshi Sugawara, Professor Yu-ichi Hayashi, Dr. Daisuke Fujimoto, and Dr. Kazuhiko Minematsu.

I had fortune to be a member of Tohoku University's Aoki Laboratory and Homma Laboratory, which gave me the good opportunity to share offices with many teachers and friends whom I esteem. I would like to thank them who made me have the fulfilling time of my life. Further-

more, I would like to extend my respect and gratitude to my seniors and friends graduated from the laboratory. Please let me introduce my lovely teachers, friends, and seniors here with great admiration: Dr. Koichi Ito, Mr. Ville Yli-Mäyry, Ms. Manami Suzuki, Mr. Hirokazu Oshida, Mr. Shota Hishinuma, Mr. Ryuichi Fufimoto, Mr. Akira Ito, Mr. Sora Endo, Mr. Kohei Koiwa, Mr. Hayato Mori, Mr. Daisuke Miyata, Mr. Kouya Yodokawa, Mr. Takehisa Okano, Mr. Jun-ichi Hata, Mr. Shotaro Sawataishi, Mr. Kohei Kazumori, Mr. Shuto Funakoshi, Dr. Sho Endo, Mr. Takafumi Hibiki, Mr. Kotaro Okamoto, Mr. Hajime Uno, Mr. Yukihiro Sugawara, Mr. Shoei Nashimoto, Mr. Daisuke Ishihata, and Mr. Wataru Kawai.

I was also lucky to study in such a beautiful city, where it is easy to find refreshments, restaurants, and amusements to get away from my work. I want to thank such shops, in particular, coffee shop To-mon, Chinese restaurant Fukuraien, Japanese soba restaurants Myouan and Koushouan, tonkatsu restaurant Katsusei, washu bar RAMBLE, and wine bar Clos de la Barre. They provided me the mental energy to pursue my (sometimes hard) work with their fine dishes and beverages.

Finally, special thanks to my dearest family, who have kindly supported me with their understanding, encouragement, sacrifice, and endless love.

Rei Ueno

January 2018, Sendai

Abstract

Cryptography has been widely deployed for secure information systems with secret communication, authentication, and digital signature. As a result of rapidly increasing LSI systems, hardware implementation of cryptographic algorithms is being essential to realize cryptographic operations efficiently from only transaction servers to resource-constraint embedded devices. In addition, there is high demand of cryptographic hardware resistant to tampering attacks such as Side-Channel Attacks (SCAs), because cryptographic hardware is used as a security primitive and root-of-trust for information systems. Thus, rapid design and verification methods for various cryptographic hardware are strongly required as more and more cryptographic algorithms, hardware architectures, and countermeasures against tampering attacks are being developed.

On the other hand, the conventional Electronic Design Automation (EDA) tools with Hardware Description Languages (HDLs) have difficulty in designing cryptographic hardware. While most modern cryptography is based on Galois-Field (GF) arithmetic, the conventional EDA tools have not supported high-level description and automatic synthesis of GF arithmetic circuits. The lack of high-level design methodology for GF arithmetic circuits forces designers to describe the structural details of cryptographic hardware with massive low-level logical expression by hand, which makes it difficult to design, debug, and optimize cryptographic hardware. To make matters worse, the difficulties with the verification are more serious than those with the design itself. We cannot verify circuits with input bit length greater than 128 bits by the common logic simulation while cryptographic hardware frequently have more than 128-bit operands for the resistance to cryptanalysis attacks. Moreover, it is basically difficult to apply the conventional high-level synthesis and formal verification methods to practical GF arithmetic circuits because they have been basically developed for integer and floating-point arithmetic. Nevertheless, complete verification of cryptographic hardware is quite important because they are frequently used for mission-critical and high-security systems.

To address these problems, a formal design method for GF arithmetic circuits was presented. The method describes GF arithmetic circuits with a hierarchical mathematical graph called *GF*

Arithmetic Circuit Graph (*GF-ACG*). Since GF-ACG hierarchically represents functions of GF arithmetic circuits by GF equations, the circuit function can be formally verified by equivalence checking of the GF equations between hierarchies. It was shown that we could completely verify a 128-bit GF multiplier and a 128-bit Advanced Encryption Standard (AES) hardware using Gröbner basis and polynomial reduction techniques for the equivalence checking.

However, it is still challenging to apply the formal design method to practical cryptographic hardware. The above method was applied to only standard and straight-forward architectures without optimization techniques nor higher-degree functions. For example, while redundant GF representations and pipelining techniques are sometimes useful for high-performance and/or SCA-resistant cryptographic hardware design, the conventional GF-ACG cannot describe them. In addition, the computation time for a Gröbner basis heavily depends on the degree and number of variables derived from the target circuit function, and therefore the algebraic techniques have difficulty in verifying functions of higher-degree (e.g., AES decryption hardware and efficient AES encryption hardware with resister-retiming techniques) and circuits optimized at logic-level (e.g., tamper-resistant AES hardware). Thus, design methodology that can be applied to a wider variety of cryptographic hardware is quite necessary in the fields of EDA and information security.

This dissertation studies a formal design of cryptographic hardware. The contribution of this dissertation is threefold.

Firstly, we propose a new formal design methodology for cryptographic hardware. We propose a new GF-ACG which can represent a wider variety of GF arithmetic circuits for practical cryptographic hardware including those with redundant GF representations and with pipeline architectures. We also propose two new equivalence checking methods for GF-ACGs and then present a verification algorithm combining three equivalence checking methods with the conventional one. The proposed algorithm can verify circuits with higher-degree functions and logic-level optimization in a systematic manner. The effectiveness and efficiency of the proposed algorithm are demonstrated through its applications to various GF arithmetic circuits and cryptographic hardware. For example, we successfully verify AES decryption hardware and tamper-resistant AES hardware while the conventional methods fail.

Secondly, we present an automatic generation system of GF multipliers for cryptographic hardware design developed based on the proposed method. The system generates verified HDL codes from multiplier specification. The system supports automatic synthesis of $GF(p^m)$ multipliers where $p = 2, 3, 5, 7, 11$ and $2 \leq m \leq 256$. In addition, the system can also generate SCA-resistant GF multipliers based on Generalized Masking Scheme (GMS). The SCA-resistance of generated circuits is verified by new algorithms proposed in this dissertation. As a result of experimental generation, we confirm that the system can generate large GF multipliers (e.g., whose

input bit length is 256×77) within a practical time. The system generates compatible HDL codes which can be used in the conventional EDA tools, and therefore the proposed formal design methodology and the conventional EDA tools are connectable through the system.

Thirdly, in order to show the significance of the proposed methodology, we design highly efficient cryptograhic hardware architectures focusing on AES. We first present a highly area-time efficient $GF(2^8)$ inversion circuit andan AES S-box based on a combination of redundant and non-redundant GF arithmetic. We then present a high throughput/gate AES hardware compressing encryption and decryption datapaths. Finally, we present an efficient SCA-resistant AES hardware architecture based on a variation of GMS. The proposed AES hardware achieves approximately 20–50% higher area-time efficiency (i.e., lower energy) than the conventional best ones. While the proposed design utilize some practical techniques such as redundant GF arithmetic, pipelining, resister-retiming (resulting in higher-degree function), and/or logic-level optimization, we confirm that the proposed formal design methodology is applicable to such practical and state-of-the-art cryptographic hardware.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

Information and Communication Technology (ICT) have been providing various services through the Internet. As a result of the development of communication and semi-conductor technologies, Large-Scale Integrated circuits (LSIs) have been widely deployed in a lot of products such as servers, laptop PCs, smart cards, mobile phones, consumer electronics, and automotives in order to connect them to networks. The Internet of Things (IoT) is considered as a promising paradigm in the domain of ICT where every thing or object is connected to interact and cooperate with each other for reaching their application goals. Hence, digital information on the Internet is being more and more valuable because of rapidly increasing the number of IoT devices. To protect the digital information from cyber attacks, cryptography plays an essential role in secure information systems performing secret communication, authentication, and digital signature [129]. For example, secure communication protocols including Secure Socket Layer/Transport Layer Security (SSL/TLS) preserve privacy and data integrity during communication on the basis of cryptography. In addition, since more and more IoT devices will

be connected through not only secure Local Area Network (LAN) but also common and insecure networks, there is a higher risk of cyber attacks such as eavesdropping and falsification. Thus, the importance of cryptography significantly increases in the context of IoT applications.

Cryptography has been studied in academia since the 1970s, and many cryptographic algorithms have been proposed and developed. Currently, Galois-Field (GF) arithmetic is widely used in many cryptographic algorithms. GF is a finite set where addition, subtraction, multiplication, and division (excluding division by zero) are defined. Since a GF has finite elements, GF arithmetic is more suitable to modern LSI systems and computers than common infinite fields (e.g., the field of complex number). Cryptographic algorithms can be classified into symmetric key and public key, both of which frequently employ GF arithmetic. For example, an ISO/IEC standard cipher Advanced Encryption Standard (AES), which is one of the most commonly used symmetric key algorithms throughout the world, exploits a byte-wise GF arithmetic to obtain resistance to cryptanalyses [100]. Furthermore, Elliptic Curve Cryptography (ECC), which is a public key cryptography widely used since 2004, utilizes addition of points on elliptic curve over GF [70]. Pairing-Based Cryptography (PBC), which is a kind of next-generation cryptography, is also constructed using a bilinear mapping to a GF from elliptic curve(s). Thus, cryptography is closely related to GF arithmetic.

In general, cryptographic algorithms require high computational costs of GF arithmetic in order to guarantee their security (i.e., resistance to cryptanalyses). For example, as of 2017, we require at least 1,000 multiplications with more than 100-bit inputs for a secure ECC-based communication and authentication. Although symmetric key ciphers including AES can be performed more than 10,000 times faster than public key cryptography, they sometimes have to encrypt/decrypt over 10,000 times longer data than public key cryptography for their main use such as secret communication and storage encryption. Hence, when implementing cryptographic algorithms, it is important to realize cryptographic operations (i.e., GF arithmetic) efficiently so as to satisfy constraints of latency, area, and power/energy consumption. In particular, hardware implementation by Application Specific Integrated Circuit (ASIC) is sometimes required for transaction servers and resource-constraint devices such as smart cards and Radio Frequency IDentifier (RFID) tags.

Hardware (i.e., LSI systems) consists of four blocks: input/output (I/O), memory, control unit, and datapath. Here, the datapath design mainly determines overall performance such as latency, throughput, area, and power/energy consumption. Therefore, design of GF arithmetic circuits, which occupies a major part of datapath, is quite important when designing cryptographic hardware. By contrast to other parts of LSI systems, the performance of an arithmetic circuits is determined by its hardware algorithm for arithmetic in addition to device- and logic-level optimizations. Such hardware algorithms are called arithmetic algorithms. There is a variety of arith-

metic algorithms for one arithmetic function [72, 113, 116]. Cryptographic hardware designers should design GF arithmetic circuits properly in the front-end stage in order to achieve required performance.

In addition, given that cryptographic hardware is used as a security primitive and root-of-trust for information systems, it should be resistant to tampering attacks. Originally, cryptographic algorithms were designed on the basis of a classical communication model where there is attacker(s) in the insecure communication channel between a sender and a receiver. (Contemporary standard cryptographic algorithms are sufficiently secure under the model as evaluated by e-government agencies such as NIST [104], CRYPTREC [43], and NESSIE [102].) However, in the last decades, implementation (or physical) attacks on cryptographic modules are attracting much attention due to the wider spread of cryptographic hardware. Such attacks retrieve secret information by tampering on the sender's or receiver's cryptographic module in addition to eavesdropping/modifying data in the communication channel. Side-channel Attack (SCA) is a typical implementation attack which exploits side-channel information (e.g., power consumption, electromagnetic radiation, and computation time) to retrieve secret key in the cryptographic module non-invasively [32, 71]. It is feasible to perform SCA with commercial and off-the-shelf products such as laptop PC and oscilloscope and is basically difficult to detect SCA by LSI. In addition, SCA can be practical to many devices, such as smart cards, which can be accessed physically by attackers. Thus, SCA is considered as one of the most powerful implementation attacks and a practical threat in the context of IoT applications. Accordingly, cryptographic hardware designers should apply countermeasures against implementation attacks including SCAs if a designed device can be a target of such attacks. Since major information leakage through side-channel is caused by datapath [83, 137], the designers should design SCA-resistant GF arithmetic circuits. Consequently, there is a high demand of designing various GF arithmetic circuits for cryptographic hardware because more and more cryptographic algorithms, implementation attacks, and their countermeasures are being developed.

However, the conventional Electronic Design Automation (EDA) tools with Hardware Description Languages (HDLs) have been mainly developed for integer and floating-point arithmetic, and have not supported high-level description and automatic synthesis of GF arithmetic circuits, which results in two major problems on cryptographic hardware design. Firstly, the lack of high-level design methodology forces designers to describe the structural details of GF arithmetic circuits with massive low-level (i.e., logical) expressions by hand, which causes a difficulty in designing, debugging, and optimizing cryptographic hardware. It is also difficult to apply the conventional high-level synthesis methods for integer to GF arithmetic because the operation rule of GF arithmetic differs depending on bit length and modular (i.e., irreducible) polynomial.

Secondly, functional verification of cryptographic hardware is quite time-consuming. A bug in cryptographic hardware can be critical vulnerability of the system [14], and ideally the functionality of cryptographic hardware should be completely verified even in the front-end design. Nevertheless, it is usually impossible to verify cryptographic hardware completely due to the long input bit length and complexity of logical structure. The most common functional verification using logic simulation, which is performed by examining test vectors to check the corresponding output, cannot verify cryptographic hardware completely. This is because the verification time increases exponentially to the input bit length. Note that cryptographic algorithms usually have more than 64-bit operands for being resistant to exhaustive search and cryptanalysis attacks. Although some functional verification methods are employed in commercial EDA tools, it is difficult to apply them to practical cryptographic hardware such as AES hardware and 128-bit GF multipliers [29, 30, 36, 46]. To make matters worse, the above methods requires to prepare a reference (i.e., golden) model prior to the verification of target circuits, which is frequently impossible in case of cryptographic hardware design.

To address these problems, a formal design method of GF arithmetic circuits was proposed in [63, 64, 110]. In the formal design method, we describe GF arithmetic circuits using formal representation called *Galois-Field Arithmetic Circuit Graph* (*GF-ACG*), which is a hierarchical mathematical graph with functional assertion represented by GF arithmetic equations. We can then perform a formal functional verification of GF arithmetic circuits described using GF-ACG by checking the equivalence of GF equations between hierarchies, namely, checking whether the functional assertion of higher-level description can be derived from those of lower-level description. Since the functions are represented by GF arithmetic equations, the equivalence checking can be resolved into the mathematical problem of solving a system of simultaneous equations. In order to proceed the functional verification in a systematic manner, computer algebra techniques based on Gröbner basis and polynomial reduction [31, 40] were introduced in [63], which makes it feasible to verify an AES hardware and a 128-bit multiplier completely in a practical time.

However, there are still issues on the design of practical cryptographic hardware even by GF-ACG. One major issue is the limitation of design space. Although many GF arithmetic algorithm and hardware architectures have been proposed for high-performance and/or SCA-resistant cryptographic hardware so far, the existing GF-ACG was applied to only standard and straight-forward ones. In other words, the existing GF-ACG has difficulty in designing practical cryptographic hardware. For example, there are several methods for representing the elements of a GF, which can determine the performance of GF arithmetic algorithms. The existing GF-ACG can handle only standard non-redundant GF representations while several GF arithmetic algorithms based on redundant representations have been recently presented [47, 68, 144]. The effectiveness of

such redundant GF arithmetic in designing high-performance cryptographic hardware have been shown in the previous works [60, 101], and therefore the importance of handling redundant GF representation is increasing in order to design practical cryptographic hardware. In addition, the existing GF-ACG focuses on combinational arithmetic circuits and cannot handle sequential ones requiring several clock cycles for a computation (e.g., pipelined circuits). In the design of cryptographic hardware, pipelining techniques are useful fo enhancing throughput and implementation efficiency and for achieving the resistance to SCA. While glitches (a.k.a dynamic hazard) of arithmetic circuits causes SCA leakage [83, 92], a pipelining technique can mitigate SCA leakage by suppressing glitches [103,134]. In 2015, the resistance of countermeasures using pipeline was formally described and generalized in [122]. Thus, pipelined GF arithmetic circuits are also essential for practical cryptographic hardware design.

The formal verification based on GF-ACG has another issue on its application to practical cryptographic hardware, which also leads to the limitation of design space by GF-ACG. Computation time for a Gröbner basis (i.e., verification time) heavily depends on the degree and number of variables derived from the target GF equations (i.e., the target circuit function). For example, AES decryption is represented by GF equations of higher degrees than those of encryption. Accordingly, AES decryption hardware and unified AES encryption/decryption hardware cannot be verified in realistic time even by the formal verification based on computer algebra. Moreover, while the hierarchical description of GF-ACG is essential to reduce the number of variables per Gröbner basis computation (i.e., verification time), such description is unavailable for arithmetic circuits optimized in logic-level, which are commonly used for designing efficient arithmetic circuits. Thus, a formal verification method whose processing time does not heavily depend on the degree and number of variables is also strongly required for practical cryptographic hardware design.

In summary, the conventional EDA tools and formal methods have a difficulty in designing practical cryptographic hardware. High-level data structure and fast formal verification method for a wider variety of GF arithmetic circuits are highly demanded in the fields of EDA and information security.

1.2 Contributions

This dissertation studies a formal design of cryptographic hardware through a threefold-contribution of (a) theory, (b) implementation, and (c) applications, namely,

(a) To present a new formal design methodology of cryptographic hardware.
(b) To produce an automatic generation system of GF arithmetic circuits for cryptographic hardware.

(c) To design highly efficient AES hardware.

In (a), we present a newly extended GF-ACG and its verification which can handle various GF algorithms including practical ones. Compared to the conventional GF-ACG, the proposed GF-ACG can represent a wider variety of GF arithmetic circuits including those based on redundant representations and with pipelining. We then propose two formula evaluation methods for verifying GF-ACG. The first method is based on a Natural Deduction (ND) system for the first-order predicate logic, and can verify GF arithmetic circuits with the functions of higher degrees very fast. The second one, which exploits a canonical form of a logic function called Positive-Polarity Reed Muller (PPRM), can verify flattened circuits (i.e., circuits optimized in logic-level) promptly using a reference circuit which can be generated by the means of GF-ACG. The proposed formal verification combines the two methods with the conventional one based on computer algebra in an efficient manner. Thus, the proposed GF-ACG and formal verification can be efficiently applied to the design and verification of various GF arithmetic circuits including ones used in practical cryptographic hardware. The effectiveness and efficiency are demonstrated through experimental designs and verifications of some GF arithmetic circuits including AES hardware. In addition, the generality and applicability of the proposed formal design method are discussed.

In (b), we present an automatic generation system of GF multipliers for cryptographic hardware design on the basis of the proposed formal design methodology. The system outputs a verified HDL description of a GF multiplier, given a circuit specification including bit length, modular polynomial, GF representation, and multiplication algorithm. In addition, the system can also generate SCA-resistant GF multipliers based on Generalized Masking Scheme (GMS) [122]. The SCA-resistance property is formally verified using a newly proposed algorithms in this dissertation. As a result of the experimental generations, we confirm that the proposed system can generate large multipliers such as ones with 256×77 bit inputs in practical time. The generated HDL description can be used in the conventional EDA tools (e.g., logic synthesis tool). Thus, the system can be considered as an implementation and interface of the proposed formal design methodology, and used for connecting the proposed methodology to the conventional EDA tools.

In (c), to show the significance of the proposed methodology, we design efficient AES cryptographic hardware. In particular, two AES hardware are developed. The first one is a unified AES hardware that supports both encryption and decryption, which achieves the highest throughput/gate efficiency and the lowest power and energy consumption than the conventional ones by exploiting several datapath optimizations. The second one is a SCA-resistant AES hardware with a GMS-based countermeasure, which utilizes a pipelining technique to suppress glitch effects and also employs resister-retiming and operation-reordering to reduce latency without area overhead. The designed hardware is evaluated using a logic synthesis and a gate-level timing simulation.

We can confirm that the first hardware is approximately 53–72% better than the conventional best ones in terms of throughput/gate, and the second architecture achieved the smallest circuit area and 11–21% lower-latency than the conventional best ones with a GMS-based countermeasure. While the conventional EDA tools and formal methods cannot be applied to the circuits having high-degree functions and/or utilizing a pipelining. On the other hand, the proposed formal design methodology is applicable to such practical and state-of-the-art cryptographic hardware designs.

1.3 Thesis Overview

This dissertation contains six chapters as follows:

Chapter 1 introduces this study with background, problems, and contributions.

Chapter 2 provides the basics of cryptographic hardware design. First, we describe GF arithmetic briefly, and introduce GF representations handled in this dissertation. We then describe an overview of cryptography including its building blocks and implementation. We also describe the conventional design and verification methods for arithmetic circuits togethher with their problems on the application to cryptographic hardware. Finally, we introduce the existing formal design and verification methods of GF arithmetic circuits based on GF-ACG, and discuss the issues.

Chapter 3 presents a new formal design methodology of cryptographic hardware. We first describe the conventional formal design and verification methods of GF arithmetic circuits based on GF-ACG. We then describe the new GF-ACG which can handle various GF representations including redundant ones and can also handle the time modality and sequential arithmetic circuits, such as pipelined ones. Moreover, we introduce two equivalence checking methods based on a natural deduction method for the first order predicate logic and a PPRM expansion, respectively. We then present a new verification algorithm combining the two equivalence checking methods with the conventional algebraic method. We demonstrate the effectiveness and efficiency of the proposed methodology through its applications to parallel multipliers, AES hardware, and tamper-resistant cryptographic hardware. Finally, we discuss the generality and applicability of the proposed methodology.

Chapter 4 produces an automatic generation systems of GF arithmetic circuits for cryptographic hardware. The proposed system can generate verified multipliers over $GF(p^m)$, where $p = 2, 3, 5, 7, 11$ and $2 \leq m \leq 256$. Such multipliers are used for symmetric key ciphers, ECC, and PBC for example [56]. In addition, the system can also generate SCA-resistant GF multipliers based on GMS and verify the SCA resistance property formally by newly proposed algorithms proposed in this dissertation. The performance of the proposed system is evaluated through experimental multiplier generations. As a result, we confirm that the proposed system can synthesize large multipliers with 256-bit inputs within a practical time.

Chapter 5 designs highly efficient AES hardware as applications of the propose design methodology. We first design a highly efficient $GF(2^8)$ inversion circuit and an AES S-box based on a combination of redundant and non-redundant GF representations. We then design a high throughput/gate AES hardware compressing encryption and decryption detapaths. The designed hardware architecture can be applied and useful to other modern ciphers because it exploits datapath optimization techniques (e.g., resister-retiming, operation-reordering, unification of encryption and decryption paths, and merging linear mappings). We also design an SCA-resistant AES hardware, which achieves a higher efficiency than the conventional ones by the optimization techniques similar to the above. Note that the above hardware can be designed using the proposed formal design methodology. Thus, we confirm that the formal design methodology would be useful for such state-of-the-art and practical cryptographic hardware design.

Chapter 6 contains concluding remarks.

2 Preliminaries

2.1 Introduction

This chapter introduces basic and preliminary information on cryptographic hardware design. We first describe overview of GF arithmetic including the importance of GF representation in designing GF arithmetic circuits. We then introduce cryptography, especially form viewpoints of cryptographic algorithms based on GFs, its implementation, and attacks on cryptographic implementation. Moreover, we explain the conventional design methods of arithmetic circuits before describing their problems on cryptographic hardware design. We also explain the conventional verification methods and their problems. Finally, we explain formal design of GF arithmetic circuits based on GF-ACG, and describe its practical issues on cryptographic hardware design.

2.2 Galois-Field Arithmetic

2.2.1 Overview

Field is an algebraic structure with addition and multiplication. In other words, a field is defined as a set where addition, subtraction, multiplication, and division (excluding division zero) are defined, and its elements are closed under the operations. For example, the field of rational numbers, the field of real numbers, and the field of complex numbers are the most commonly known fields. The above fields are called infinite fields since they have infinite elements. On the other hand, fields with finite elements are called GFs (or finite fields). A GF with ω elements is denoted by $GF(\omega)$, where ω is an integer referred to as the order of the field. GFs are classified into two types: prime field and extension field. A prime field $GF(p)$ has the order of a prime number p while an extension field $GF(p^m)$ has the order of the mth power of p. Here, p is referred to as the characteristics of the fields, and m is an integer greater than one.

A prime field $GF(p)$ consists of finite natural numbers $0, 1, \ldots, p-1$. The addition $\oplus$ and multiplication $\otimes$ over $GF(p)$ are defined as

$$a \oplus b = (a + b) \bmod p, \tag{2.1}$$

$$a \otimes b = (a \times b) \bmod p, \tag{2.2}$$

where a and b are elements of $GF(p)$ (i.e., a and $b \in \{0, 1, \ldots, p-1\}$).

We then describe extension field $GF(p^m)$. An extension field $GF(p^m)$ consists of polynomials of degrees up to $m-1$ over $GF(p)$. Let x and $H(x)$ be a formal variable and an irreducible polynomial of degree m over $GF(p)$. The addition $\boxplus$ and multiplication $\boxtimes$ over $GF(p^m)$ are defined as

$$A \boxplus B = (A \oplus B) \bmod H(x), \tag{2.3}$$

$$A \boxtimes B = (A \otimes B) \bmod H(x), \tag{2.4}$$

where A and B are elements of $GF(p^m)$. Constructing an extension field (e.g., $GF(p^m)$ in this case) using an mth degree irreducible polynomial is called the mth degree extension, and the field before extension (e.g., $GF(p)$ in this case) is called subfield. As in Eqs. 2.3 and 2.4, the operations over an extension field is realized by a combination of operations over its subfield.

In an extension field with a formal variable x, x is not a normal variable representing an unknown nor arbitrary number, but an element constructs the extension field. Therefore, in order to distinguish x and other normal variables, we replace x with β which is a root of $H(x)$. Since $H(\beta) = 0$ which is followed by $\beta^m = \beta^m - H(\beta)$, the modulo operations over $GF(p^m)$ described in Eqs. 2.3 and 2.4 can be considered as substitution of $\beta^m - H(\beta)$ to β^m in order to

reduce the degree of the resulting polynomial to at most $m-1$. This construction of extension field is called the (algebraic) adjunction of β to $GF(p)$. This is similar to constructing the field of complex number from that of real number by adjunction of the imaginary number i which is a root of an irreducible polynomial x^2+1. While two constructions are the essentially equal, the construction by adjunction is useful for introducing GF representations.

In this subsection, while we described only an extension field of a prime field, we can further extend an extension field $GF(p^{m_0})$ using an irreducible polynomial of degree of m_1 over $GF(p^{m_0})$ (and more repeatedly). Such extension fields which are derived by repeating extension are called tower fields (or composite fields) and denoted by $GF((p^{m_0})^{m_1})$ in the above case. The operations over tower fields are defined by a combination of subfield arithmetic recursively. Tower-field arithmetic is frequently used for compact hardware implementation than the corresponding extension field.

In the following, we use $+$ and $\times$ even for the operations for GFs when they can be correctly grasped.

2.2.2 GF representations

This subsection introduces GF representations (i.e., how to represent elements of GF) in the order of non-redundant and redundant ones. In addition, we describe the differences between GF representations, especially from the viewpoint of hardware implementation.

Non-redundant GF representations

Let q be a prime or the power of prime. Giving elements of $GF(q^m)$ as polynomials with β whose degree is less than m is equivalent to representing $GF(q^m)$ by an m-dimensional linear space over $GF(q)$ with a basis $\{\beta^{m-1}, \beta^{m-2}, \ldots, \beta^0\}$. Here, the basis $\{\beta^{m-1}, \beta^{m-2}, \ldots, \beta^0\}$ is called a Polynomial Basis (PB), and thus, elements of GF is represented non-redundantly by a basis, that is, a combination of m linearly independent elements. PB is the most common representation for GFs. Actually, many modern cryptographic algorithms such as AES are defined using arithmetic operations over PB-based GFs.

Normal Basis (NB) is another non-redundant GF representation [54, 99]. An NB-based $GF(q^m)$ is given as $\{\alpha^{q^{m-1}}, \alpha^{q^{m-2}}, \ldots, \alpha^{q^0}\}$, where α is an element of the (PB-based) $GF(q^m)$ satisfying $\alpha^{q^{m-1}} + \alpha^{q^{m-2}} + \cdots + \alpha^{q^0} = 1$. For example, an NB-based $GF(2^3)$ with an irreducible polynomial $H(\beta) = \beta^3 + \beta + 1$ is given by $\{\alpha^4, \alpha^2, \alpha^1\}$, where $\alpha = \beta^3 = \beta + 1$.

The major advantage of NB in designing GF arithmetic circuits is that the qth power of any element represented by NB is calculated by solely cyclic shift (without any logic gate). Let a be elements of an NB-based $GF(q^m)$ and a is represented as $a = a_{m-1}\alpha^{q^{m-1}} + a_{m-2}\alpha^{q^{m-2}} + \cdots +$

$a_0\alpha^{q^0}$, where $a_{m-1}, a_{m-2}, \ldots, a_0$ are elements of $GF(q)$. Since $(y+\hat{y})^q = y^q + \hat{y}^q$ and $y^{q^m} = y$ for any values of y and $\hat{y} \in GF(q^m)$, $a^q = a_{m-2}\alpha^{q^{m-1}} + a_{m-3}\alpha^{q^{m-2}} + \cdots + a_0\alpha^{q^1} + a_{m-1}\alpha^{q^0}$. Hence, the usage of NB can lead to compact hardware implementation for many applications with a lot of qth powering operation [33, 34]. On the other hand, since two-input GF multipliers based on PB are superior to those based on NB in terms of latency and circuit area, it is important to use proper representation for required circuit function. Moreover, a combination of PB and NB is useful for designing efficient GF arithmetic circuits [106].

Redundant GF representations

We then describe redundant GF representations.

The performance of GF arithmetic circuits heavily depends on irreducible polynomial that defines rules of arithmetic operations. According to [143], typical GF arithmetic algorithms [58, 63] are more suitable to GFs where the modular polynomials have less terms. However, given $GF(q^m)$, such good irreducible polynomials is not always available for the modular polynomial of GF. For example, polynomial in the form of $x^m - 1$, which is called binomial, is one of optimal modular polynomial for GFs. Nevertheless, $x^m + 1$ is available for constructing $GF(2^m)$ since $x^m + 1$ is always reducible over $GF(2)$ for $m \geq 2$. Thus, in contrast to non-redundant GF representations, two redundant GF representations called Polynomial Ring Representation (PRR) [47, 60, 68, 133] and Redundantly Represented Basis (RRB) [101, 144] were proposed and developed. $GF(p^m)$ based on redundant representations has a set of polynomials whose degrees are up to $n-1$, where $n > m$, and uses an nth degree reducible polynomial as the modular polynomial instead of mth degree irreducible polynomial. Thus, redundant GF representations can be sometimes useful for designing efficient arithmetic circuits over $GF(p^m)$ where an nth degree reducible polynomial better than mth degree irreducible polynomial is available for the modular polynomial.

Informally, a PRR-based $GF(q^m)$ is defined as an m-dimensional subspace of an n-dimensional linear space over $GF(q)$. This indicates that a PRR-based GF is also equivalent to a Cyclic Redundancy Code (CRC), which is a kind of error-correction code.

We then describe the construction of PRR-based GF. Let $G(x)$ be a polynomial of degree $n-m$, which is relatively prime to the mth degree irreducible polynomial $H(x)$. Let $P(x)$ be an nth degree polynomial given by the product of $G(x)$ and $H(x)$. Here, PRR-based $GF(p^m)$ is defined as a set of up to $(n-1)$th degree polynomials divisible by $G(x)$ with addition and multiplication modulo $P(x)$. For example, a PRR-based $GF(2^2)$ with $P(x) = x^3 + 1$ (which follows $H(x) = x^2 + x + 1$ and $G(x) = x + 1$) is given by a set $\{0, x+1, x^2+1, x^2+x\}$, where $x^2 + 1$ is the multiplicative unit element. See [60] for a construction method of mapping between PRR- and PB-based GFs.

Table 2.1 Comparison between PRR- and PB-based $GF(2^m)$ parallel multipliers

m	$P(x)$	Critical delay	
		PRR	PB
4	x^5+1	T_A+3T_X	T_A+3T_X
10	$x^{11}+1$	T_A+4T_X	T_A+5T_X
12	$x^{13}+1$	T_A+4T_X	T_A+5T_X
13	$x^{16}+x^3+1$	T_A+5T_X	T_A+6T_X
18	$x^{19}+1$	T_A+5T_X	T_A+6T_X
19	$x^{22}+x^3+1$	T_A+6T_X	T_A+7T_X
20	$x^{25}+1$	T_A+5T_X	T_A+6T_X
21	$x^{24}+x+1$	T_A+5T_X	T_A+6T_X
28	$x^{29}+1$	T_A+5T_X	T_A+6T_X
36	$x^{37}+1$	T_A+6T_X	T_A+7T_X
42	$x^{42}+x+1$	T_A+7T_X	T_A+8T_X
52	$x^{53}+1$	T_A+6T_X	T_A+7T_X
58	$x^{59}+1$	T_A+6T_X	T_A+8T_X
59	$x^{61}+x^{26}+1$	T_A+8T_X	T_A+9T_X
133	$x^{136}+x^{43}+1$	T_A+10T_X	T_A+11T_X
514	$x^{943}+x+1$	T_A+11T_X	T_A+12T_X
819	$x^{821}+x^{313}+1$	T_A+12T_X	T_A+14T_X

Table 2.1 shows a comparison of the minimum critical delay of PRR- and PB-based $GF(2^m)$ parallel multipliers for several values of m, where the critical delays are derived from the number of gates on critical path, and T_A and T_X denote the delays of an AND gate and XOR gate, respectively. In Tab. 2.1, PRR-based multiplier has the smaller delay than PB-based ones. As mentioned before, the performance of PRR-based GF arithmetic circuits can be better than those of PB- and NB-based GF arithmetic circuits when the modular polynomial of PRR-based GF (i.e., $P(x)$) is given by a binomial. For example, if an mth polynomial $x^m+x^{m-1}+\cdots+1$ is irreducible over $GF(2)$, we can construct a PRR-based $GF(2^m)$ with $P(x)=x^{m+1}-1$ because $x^{m+1}-1=(x-1)(x^m+x^{m-1}+\cdots+1)$. Here, the polynomial $x^m+x^{m-1}+\cdots+1$ is called the mth degree All One Polynomial (AOP). Irreducible AOPs are sometimes important for constructing optimal NB and RRB in addition to PRR. Note that PRR-based GF should use a formal variable as its element in contrast to other representations because x no longer preserves the attributes of a root of irreducible polynomial.

The major advantages of using the binomial are as follows: (i) Parallel multiplication can be given as the discrete time Wiener-Hopf equation, and (ii) Squaring operation and a part of constant multiplication are performed only by bit-wise permutation (i.e., wiring). The advantage (i) indicates that reduction by $P(x)$ in multiplication can be performed only by bit-wise permutation while multiplication based on non-redundant representation requires some addition over $GF(2)$ for the reduction. In (ii), squaring and a part of constant multiplication can be performed with-

out any logic gate since they are the special case of multiplication. Accordingly, the PRR-based design can be more efficient than conventional designs.

We then introduce RRB. RRB is another redundant representation of GF [101, 144]. Each element of RRB-based GF is represented by an m-dimensional linear space using a root of an mth degree irreducible polynomial similarly to PB and NB. However, each element of RRB-based $GF(p^m)$ is represented with n linearly dependent elements $\beta^{n-1}, \beta^{n-2}, \ldots, \beta^0$ [*1]. The modular polynomial of RRB is given by an nth degree reducible polynomial $P(\beta)$ $(= G(\beta)H(\beta))$. Since $P(\beta) = 0$, we can perform the reduction using $\beta^n = \beta^n - P(\beta)$ instead of $\beta^m = \beta^m - P(\beta)$. Since the modular polynomial $P(\beta)$ can be selected as same as $P(x)$ for PRR-based GF, we can design efficient GF arithmetic circuits based on RRB-based GF, especially when the mth degree AOP is irreducible, that is, a binomial $\beta^{m+1} - 1$ can be used. Note that the elements of RRB-based GFs are represented in a non-unique manner because $\beta^n, \beta^{n-1}, \ldots, \beta^0$ are linearly dependent, which is typical difference between PRR and RRB.

RRB-based GF with a binomial also has the advantage of two-input multiplication, squaring operation, and constant multiplication as well as PRR. When the mth degree AOP is irreducible over $GF(p)$, multiplication over RRB-based $GF(p^m)$ can be performed using Cyclic Vector Multiplication Algorithm (CVMA) [107], which is one of the fastest multiplication algorithm for type-I Optimal NB (ONB). Type-I ONB can be used if and only if the mth degree AOP is irreducible over $GF(p)$. It is given by $\{\beta^{2^{m-1}}, \beta^{2^{m-2}}, \ldots, \beta^{2^0}\} = \{\beta^m, \beta^{m-1}, \ldots, \beta^1\}$, which indicates we can derive RRB by adding a base element β^0 to ONB. Thus, we can design more efficient multipliers by combining RRB and CVMA. As an example, let us consider a $GF(2^4)$ multiplier based on RRB. Let a and b $(\in GF(2^4))$ be the inputs and c $(\in GF(2^4))$ be the output. In RRB, a is given by $a_4\beta^4 + a_3\beta^3 + \ldots a_0$, where $a_0, a_1, \ldots a_4$ are elements of $GF(2)$. b and c are also given in the same manner. The multiplication is represented by

$$c = a \times b = c_4\beta^4 + c_3\beta^3 + c_2\beta^2 + c_1\beta + c_0, \tag{2.5}$$

[*1] Therefore, RRBs are not bases of m-dimensional linear space. Note that the name RRB that a PB $\{\beta^{n-1}, \beta^{n-2}, \ldots, \beta^0\}$ can be redundantly represented by $\{\beta^{n-1}, \beta^{n-2}, \ldots, \beta^0\}$ with additional elements $\beta^{n-1}, \beta^{n-2}, \ldots, \beta^m$.

where

$$c_0 = (a_1 + a_4)(b_1 + b_4) + (a_2 + a_3)(b_2 + b_3), \tag{2.6}$$

$$c_1 = (a_0 + a_1)(b_0 + b_1) + (a_2 + a_4)(b_2 + b_4), \tag{2.7}$$

$$c_2 = (a_0 + a_2)(b_0 + b_2) + (a_3 + a_4)(b_3 + b_4), \tag{2.8}$$

$$c_3 = (a_0 + a_3)(b_0 + b_3) + (a_1 + a_2)(b_1 + b_2), \tag{2.9}$$

$$c_4 = (a_0 + a_4)(b_0 + b_4) + (a_1 + a_3)(b_1 + b_3). \tag{2.10}$$

The critical delay of such an RRB-based multiplier is $T_A + 2T_X$, while those of multipliers based on non-redundant representations are T_A+3T_X [101]. The gate count of the RRB-based multiplier requires only 10 AND and 25 XOR gates [101], whereas that of a PRR-based multiplier requires 25 AND and 20 XOR gates [47]. Nekado et al. [101] designed a more efficient $GF((2^4)^2)$ inversion circuit based on RRB by utilizing the above advantage.

The above was brief description about typical GF representations. It was proven that all GFs with the identical order are isomorphic to each other regardless of representations and defining polynomials. Since most cryptographic algorithms are defined using PB-based GF arithmetic, elements of PB-based GF are frequently converted to the corresponding elements of other representations suitable to the operation [128].

Moreover, there exists a representation different from both redundant and non-redundant representations, called logarithmic representation. The logarithmic representation can perform multiplication and division with only an addition and a subtraction over $\mathbb{Z}/(q-1)\mathbb{Z}$, respectively. On the other hand, addition and subtraction based on logarithmic representation requires a large table with a size of at least $r^2 \log r$ (r is the order of GF), and there is no efficient conversion method between GFs based on logarithmic and other representation. Therefore, the logarithmic representation is scarcely used in cryptographic algorithms.

2.3 Cryptography and Its Implementation

2.3.1 Overview of cryptography

Information security is technology to counteract various threats to information systems. Table 2.2 examples of threats to information systems. The threats can be classified into two types: deliberate and environmental/accidental ones. Deliberate threats are caused by malicious actions of the attackers. Environmental/Accidental threats are caused unintentionally (but they might be exploited by the attackers). While information security covers both of the threats in a broad since, information security in a common sense counteracts deliberate threats. ISO/IEC 27002, which

Table 2.2 Threats to information systems

Type of threats	Examples
Deliberate	Eavesdropping, falsification, malware, ransomware, computer virus, unauthorized access, denial of service
Environmental/Accidental	Blackout, fire, earthquake, soft error, bugs in software or hardware, equipment malfunction, operational mistake, week password

is the standard for information security management systems, define information security as the preservation of the following three properties:

- Confidentiality
  Only authorized users can use services (e.g., read messages and data). Unauthorized ones cannot use services.
- Integrity (or authenticity)
  Messages and data cannot be modified due to unauthorized people nor accidents. In practice, integrity indicates that authorized users can detect the modification (i.e., authenticity).
- Availability
  Authorized users can use services whenever they want.

Cryptography mainly provides confidentiality and integrity. Cryptography basically consists of three building blocks: (i) cryptographic algorithm, (ii) one-way hash function, and (iii) cryptographic Pseudo Random Number Generator (PRNG), which are used in a combination to satisfy the above properties. In the following, we briefly describe each building block.

(i) Cryptographic algorithms

Cryptographic algorithms translate messages into ciphertext which cannot be read by unauthorized people (i.e., attackers), and basically consist of encryption and decryption functions. Encryption translates plaintext P into ciphertext C using encryption key K_E while decryption inversely obtains P from C using decryption key K_D. Let Enc and Dec be encryption and decryption functions, respectively. The relation between encryption and decryption is represented by

$$C = \mathrm{Enc}(P, K_E), \tag{2.11}$$

$$P = \mathrm{Dec}(C, K_D). \tag{2.12}$$

Cryptographic algorithms are classified into two types: symmetric key and public key cryptography. Symmetric key cryptography uses the identical value for K_E and K_D (i.e., $K_E = K_D$) while public key cryptography uses different values for them (i.e., $K_E \neq K_D$). In symmetric key

cryptography, K_E and K_D are also called secret key. On the other hand, in public key cryptography, encryption key K_E and decryption key K_D are called public key and secret key, respectively. Confidentiality of cryptographic algorithms is based on that of secret key. In other words, cryptographic algorithms are designed so that it is computationally difficult to obtain ciphertext from plaintext without secret key. Therefore, users are authorized by holding the secret key. In addition, since only authorized users with secret key can perform (symmetric key) encryption and decryption, cryptographic algorithms are sometimes used for generating authentication tags and digital signatures. Therefore, system designers should carefully design the systems so that secret key should not be leaked.

Most of modern cryptographic algorithms are defined using GF arithmetic because GFs have finite elements and GF arithmetic is suitable to modern LSI systems rather than infinite field arithmetic. In addition, logical structure of multiplicative inversion operation and Discrete Logarithm Problem (DLP) over GFs can be exploited to design cryptographic algorithms resistant to cryptanalyses. Cryptographic algorithms are further explained in Sections 2.3.2 and 2.3.3.

(ii) One-way hash functions

Hash functions are functions that compute values with a fixed length (i.e., hash values) from a messages with arbitrary length. One-way hash functions (or cryptographic hash functions) is hash functions used in cryptography. One-way hash functions are employed for detection of message modification (i.e., checking the integrity) because the equivalence of two messages can be easily checked by compering their hash values. One-way hash functions have two properties called one-wayness and collision-resistance in order to prevent the attackers from replacing a message to other messages which have the hash value equal to that of the original one. One-wayness indicates computational difficulty in computing messages inversely from a hash value. Collision-resistance indicates computational difficulties in finding two messages with the identical hash value and in finding messages with a given hash value. Since the above properties make it infeasible to replace massage without changing the hash value, one-way hash functions can be used for checking integrity of messages.

(iii) Cryptographic PRNG

PRNG deterministically generates random bit strings without any statistical biases from a seed value. PRNG is used for generating cryptographic keys in security applications. On the other hand, by contrast to common PRNGs (e.g., those for simulation), PRNGs for security applications require one additional property called unpredictability, which indicates computational difficulty in predicting the next bit from the previous bits. Cryptographic PRNGs generate unpredictable random bit string, and are frequently constructed with symmetric key ciphers and/or one-way hash

functions.

2.3.2 Cryptographic algorithms

In this subsection, we describe cryptographic algorithms, which is one of the most important building block. We first briefly explain symmetric and public key cryptography, and then introduce some examples of cryptographic algorithms.

Symmetric key cryptography

In symmetric key cryptography, an encryption key and the corresponding decryption key are given by the identical value, that is, $K_E = K_D$. The major advantage of symmetric key ciphers is that encryption and decryption can be performed 100 to 30,000 times faster than those of public key cryptography. The main use of symmetric key ciphers is secret communication, storage encryption, and so on. On the other hand, there is a problem of key delivery which implies that sender and receiver should share the secret key in advance. Key delivery should be realized by public key cryptography.

The basic idea of encryption in symmetric key cryptography is to translate plaintext to ciphertext using a huge random substitution table depending on secret key. Ideally, the ciphertexts is seems to be random bit string, and can be inversely transformed into plaintext (i.e., decrypted) only by people with the secret key. However, since it is difficult to implement such huge table in practice, encryption and decryption of symmetric key ciphers are realized by repeating small substitutions (e.g., byte-wise) and permutation. Symmetric key cryptography is classified into two ciphers: block cipher and stream cipher. A block cipher has a fixed input bit length, and its encryption and decryption are performed in a block-wise manner. On the other hand, a stream cipher generates key stream from an initial secret key and initial vector with a fixed bit length before its encryption (or decryption) which is performed by a bit-parallel XOR with plaintext (or ciphertext).

Public key encryption

In public key encryption, an encryption key and the corresponding decryption key are given by different values, that is, $K_E \neq K_D$. Since encryption keys are public data without contaminating nor compromising security, public key cryptography can achieve more functions than symmetric key cryptography, such as key delivery. In addition, public key cryptography is suitable to one-to-many communication because decryption can be performed only by people with decryption key. Conversely, digital signatures which can be verified using public key is realized by encrypting a hash value of messages with decryption key.

The security (e.g., confidentiality) of public key cryptography comes from computationally dif-

Table 2.3 Examples of ISO/IEC 18033 standard cryptographic algorithms and used GFs

Public key cryptography	Symmetric key cryptography		
Key Encapsulation Mechanisms (KEMs)	Block ciphers	Stream ciphers	
ECIES-KEM, PSEC-KEM	AES, Camellia	MUGI	SNOW2.0, KCipher-2
$GF(2^m)$ $(m \geq 160)$, $GF(p)$ $(p \geq 2^{163})$	$GF(2^8)$	$GF(2^8)$	$GF(2^8), GF((2^8)^4)$

ficult mathematical problems. For example, RSA, which is a pioneering public key cryptography, exploits difficulty in factorizing large numbers while it is much easier to compute the product of prime numbers. As other examples, DLP over GFs and DLP over elliptic curves (ECDLP) are also such mathematical problems for public key cryptography. Public key cryptography from ECDLP is called ECC. ECC can be used with shorter keys than the previous ones, which leads to less computational cost of its encryption and decryption. Therefore, ECC is widely used since 2004.

In recent years, as a result of the development of networks, there is high demand of public key encryption with more sophisticated functions than the conventional ones, such as ID-based Encryption (IDE) [21], Attribute-Based Encryption (ABE) [55,124], Public key Encryption with Keyword Search (PEKS) [22], and Functional Encryption (FE) [69, 112]. Such encryption schemes can be realized using bilinear mappings on elliptic curves. Encryption schemes using bilinear mappings is called PBC, which have been intentionally studied and developed, and is expected as one of next-generation cryptography.

Examples of cryptographic algorithms based on GF arithmetic

Table 2.3 shows examples of ISO/IEC 18033 standard cryptographic algorithms and used GFs. Many symmetric ciphers use $GF(2^m)$ (especially $GF(2^8)$) because elements of $GF(2^m)$ can be represented by m bits,and inversion and matrix operations over $GF(2^m)$ are useful to functions for resistance against cryptanalyses. Recently, Authenticated Encryption with Associated Data (AEAD), which is a scheme of symmetric encryption securely unifying confidentiality and integrity (authenticity), attracts much attention due to a lot of practical attacks exploiting lack of integrity. It is notable that TLS 1.3 (working draft as of November 2017) will employ only AEADs in its cipher suite and will no longer use ciphers without authenticity because of many practical attacks on block ciphers without authenticity (e.g., BEAST attack [49] and Lucky Thirteen attack [6]). For example of AEAD, AES-GCM [52, 89] is one of the most commonly used AEAD. In addition, ChaCha20-Poly1305 [74] is an emerging AEAD which have been already introduced to TLS, OpenSSH, and, Google Chrome, and Google web services. AEAD consists of two blocks: encryption and authentication tag generation. AES-GCM and ChaCha20-Poly1305 employ $GF(2^{128})$ and $GF(p)$ where $p = 2^{130} - 5$ for authentication tag generation, respectively.

Furthermore, as of 2017, an international competition for AEAD (CAESAR) [42] is being held, where many candidate AEADs of CAESAR also employ GF arithmetic with large orders.

On the other hand, public key cryptography employ larger GFs than symmetric ciphers because its security comes from difficulties of factorization in a large prime, DLP of large GFs, and so on. In particular, since public key cryptography based on GFs with odd characteristics can use shorter key than that based on $GF(2^m)$, ECC and PBC over $GF(3^m)$, $GF(7^m)$, and $GF(p)$ where $p \geq 2^{160}$ have been developed [9, 13, 23, 50, 51, 56, 75]. Moreover, as an emerging public key cryptography, some post-quantum cryptography such as lattice-based cryptography [90] and Multivariate Public Key Cryptography (MPKC) [44] also uses GF arithmetic.

2.3.3 Block ciphers and modes of operation

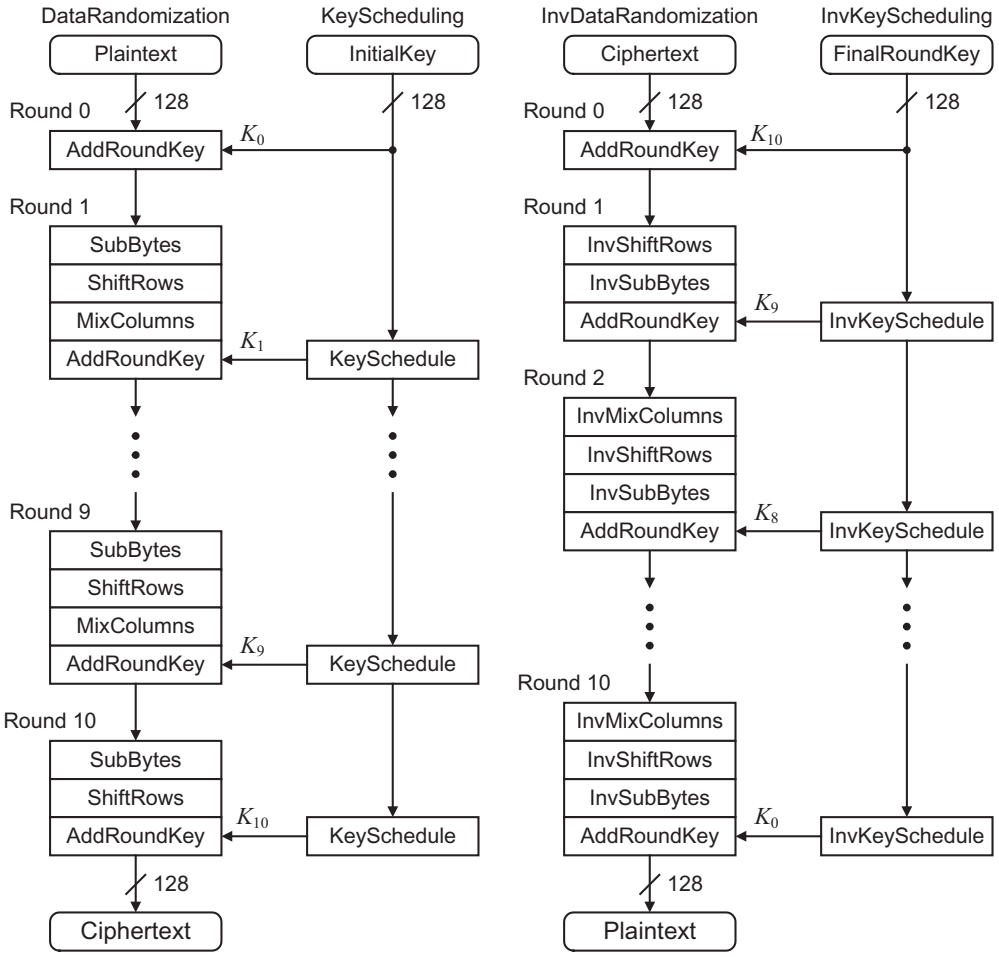
This subsection focuses on an block cipher AES since AES is an representative and the most commonly used symmetric cipher. In addition, we introduce modes of operation, which are rules for block ciphers to encrypt/decrypt messages longer than their block length with preserving confidentiality (and authenticity).

128-bit block cipher AES

AES was designed by Rijmen and Daemen in 2000 before its specification was published by NIST. Currently, AES have been specified by ISO/IEC 18033 which is the standard for cryptographic algorithms, and its use is also recommended by Japanese e-government agency CRYPTREC. AES was designed not only to be secure against the conventional cryptanalyses [15, 88], but also to be efficiently implemented on many platforms such as CPUs, microcontrollers, FPGAs, and ASIC.

As described Section 2.3.2, while symmetric cipher encryption transforms plaintext to ciphertext using a huge random table depending on secret key, such a huge table cannot be implemented in practice. Therefore, AES (and other block ciphers) encryption is performed by applying a function called round repeatedly. The round function of AES consists of an 8-bit substitution, a 32-bit permutation, and round key addition. Inputs of each round function are given by the output of previous round and round key, which is computed from the secret key using a key scheduling function. AES can be efficiently implemented on both software and hardware thanks to such round construction. In addition, round-wise design makes it easier to evaluate its security and performance.

Figure 2.1 shows the flows of (a) encryption and (b) decryption of AES. The block and key lengths of AES are given by 128 bits, and the number of rounds is ten. The 128-bit intermediate values and round keys are represented by 4×4 matrices over PB-based $GF(2^8)$ with an irreducible



(a) Encryption (b) Decryption

Fig. 2.1 (a) Encryption and (b) decryption flows of AES.

polynomial $\beta^8+\beta^4+\beta^3+\beta+1$; and therefore, the round and key scheduling functions is defined by arithmetic operations over the $GF(2^8)$.

The round function consists of four subfunctions: (a) SubBytes, (b) ShiftRows, (c) MixColumns, and (d) AddRoundKey. (a) SubBytes applies a byte-wise substitution (i.e., a nonlinear function over $GF(2^8)$) to each byte (i.e., element of $GF(2^8)$). The nonlinear function is called S-box. (b) ShiftRows is row-wise cyclic shift, where each row is shifted cyclically to left by the row index. (c) MixColumns performs a column-wise linear transformation. Namely, a fixed

matrix M is multiplied to each columns from left. The matrix M is given by

$$M = \begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_3 & v_0 & v_1 & v_2 \\ v_2 & v_3 & v_0 & v_1 \\ v_1 & v_2 & v_3 & v_0 \end{pmatrix}, \tag{2.13}$$

where $v_0 = \beta$, $v_1 = \beta + 1$, $v_2 = 1$, and $v_3 = 1$. Finally, (d) AddRoundKey adds round key to the intermediate value over $GF(2^8)$, and is realized by a bit-parallel XOR. Since the round function (i.e., subfunctions) is bijection, AES decryption is performed by its inverse functions, namely, InvSubBytes, InvShiftRows, InvMixColumns, and AddRoundKey.

On the other hand, the key scheduling function consists of an XOR chain (i.e., addition over $GF(2^8)$) following three subfunctions: (e) RotWord, (f) SubWord, and (g) AddConstant. These subfunctions including the XOR chain are row-wise (i.e., word-wise) operations for the matrix of round key. (e) RotWord performs a bit-wise rotation of the first row. (f) SubWord applies the S-box to elements of the output of RotWord. (g) AddConstant adds a round constant to the output of SubWord. Then, the next round key is calculated by XOR chain of the output of AddConstant following rows of current round keys. Since the key scheduling function is also bijective, the round keys can be calculated from the final round key in the reverse order during decryption.

The S-box of AES consists of inversion over $GF(2^8)$ and an affine transformation over $GF(2)$. Inversion operations over $GF(2^m)$ are known as a useful component for m-bit substitution functions to be against major cryptanalysis techniques [108]. On the other hand, matrix operations over GFs (e.g., MixColumns) can efficiently permutate intermediate values while their inverse functions (i.e., inverse matrix) are easily found. Thus, such GF operations are frequently used to construct symmetric ciphers [7, 57, 132, 136].

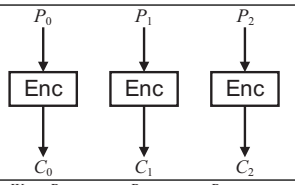
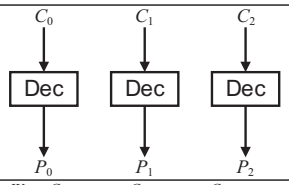
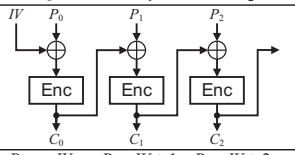
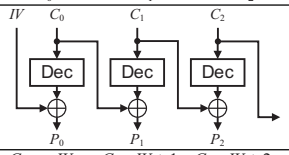
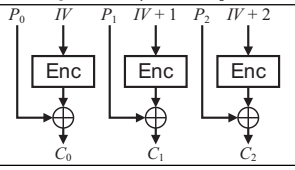
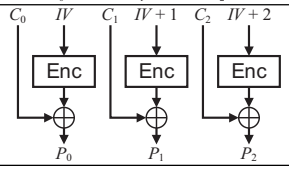
Modes of operation

Modes of operation are rules for block ciphers to encrypt/decrypt message longer than their block length because block ciphers can encrypt/decrypt messages with their block length at a time.

Table 2.4 show the typical modes of operations and their features. In the columns of Encryption and Decryption, P_ξ and C_ξ denote the ξth block of plaintext and ciphertext, respectively. The blocks "Enc" and "Dec" denote the encryption and decryption, respectively. The operator $\oplus$ denotes bit-parallel XOR. In addition, IV denotes the initial vector with the block length. Note that we omitted the input of secret key to Enc and Dec.

The Electrical CodeBook (ECB) mode is the most simplest modes of operation. In the ECB encryption, each plaintext block is independently encrypted and the corresponding ciphertext block

Table 2.4 Typical modes of operation

Modes	Encryption flow	Decryption flow	Block-wise parallelism		Recommended?
			Encryption	Decryption	
ECB			Available	Available	No (due to insecurity)
CBC			Not available	Available	Yes
CTR			Available	Available	Yes

is directly given by the output. Although ECB can be the most easily implemented because of no additional operations for the mode, the use of ECB is not recommended due to critical security flaws. (ECB encryption is also called encryption without modes of operation.) Since the ciphertext blocks which is obtained from the identical plaintext blocks should be equal, information on plaintext would be estimated from the ciphertext. In addition, if the attackers change the order of the ciphertext blocks, we cannot detect the changes of blocks.

The Cipher Block Chaining (CBC) mode solves the above flaws using feedbacking ciphertext blocks to the next block encryption. In the CBC mode encryption, the ciphertext blocks should be different even if they are obtained from the identical plaintext. In addition, the change of block order can be detected because the decrypted plaintext blocks corresponding to changed blocks are broken[*2].

Finally, the CounTeR (CTR) mode uses a block cipher like a stream cipher. In the CTR encryption (or decryption), key stream blocks are generated from IV and a counter before XORing plaintext (or ciphertext) to obtain ciphertext (or plaintext). Note that, in Tab. 2.4, the operator $+$ with IV denotes the integer addition. In contrast to ECB and CBC modes, in the CTR mode, a bit-flip in ciphertext causes a bit-flip at the identical location of the plaintext, which cannot be detected. While such a lack of authenticity can be a security risk, it can be an advantage in some applications such as streaming distribution because errors in ciphertext is not diffused.

The modes of operation have an influence on not only security but also the performance of im-

[*2] It is difficult to detect such changes if values of plaintext blocks are uniformly distributed. Message Authentication Code (MAC) and AEAD can provide the authenticity even in this case.

Table 2.5 Comparison of cryptographic software and hardware implementation

Implementation	Cost	Flexibility	Latency	Power/Energy consumption	Circuit area	Tamper-resistance
Software	**Low**	**High**	High	High	Large	Low
Hardware	High	Low	**Low**	**Low**	**Small**	**High**

plementation. For example, the CBC mode encryption cannot work with block-wise parallelism (e.g., multicore and pipelining) to enhance throughput and implementation efficiency due to the feedback of ciphertext blocks. On the other hand, in the CTR mode, both encryption and decryption are parallelizable. In addition, the CTR mode does not require to implement decryption algorithm because the CTR decryption is performed using not decryption algorithm but encryption algorithm. When implementing block ciphers, the designers should select architectures suitable to the modes of operation used in the application.

2.3.4 Cryptographic implementation

There is two types of cryptographic implementation: software and hardware implementation. In this dissertation, software implementation indicates implementation on CPUs and microcontrollers, among other things. Hardware implementation indicates implementation on FPGA (or Programmable Logic Device (PLD)) and ASIC. Such computing modules for cryptographic operations is called cryptographic modules in this dissertation. In the following, we describe characteristics of each implementation.

Table 2.5 illustrates comparison between hardware and software implementation, where Cost denotes cost for design and implementation, Flexibility indicates easiness/difficulty of changes and modification of functionality, Latency denotes latency of encryption and decryption (which is closely related to throughput), Power/Energy consumption denotes power and energy consumption for cryptographic operations, Circuit area denotes the minimal circuit area or footprint for implementation, and Tamper-resistance indicates achievable robustness to implementation attacks. In each column of Tab. 2.5, the bold characters emphasize that the implementation is superior to the other in the above aspect.

Software implementation requires lower cost than hardware implementation because we can use commercial and ready-made products such as CPUs and memories. In addition, we can easily change and modify the functionality of software implementation by solely unpdating programs. On the other hand, the main drawbacks of software implementation is latency and power/energy consumption since software implementation should perform cryptographic operations using the existing instruction set. Some instruction sets would have difficulty in performing bit-wise opera-

tions and GF arithmetic operations on software efficiently due to the fixed operand length and/or lack of instruction. Hardware implementation can achieve much lower latency, smaller area, and lower power/energy consumption using dedicated datapath for cryptographic operations while it is quite difficult to change and modify the functionality of hardware after its manufacture and market introduction.

Nowadays, there is high demand of cryptographic hardware because of criteria of performance and security. Cryptographic algorithms have been deployed in more and more embedded and resource-constraint devices such as mobile phones, smart cards, and RFID tags. Nevertheless, such applications sometimes require very high speed response in order to realize automatic ticket gates, access management systems, and so on. Moreover, some devices with battery and wireless power supply have strong constraints of power/energy consumption. Cryptographic hardware is essential for such devices and applications in order to achieve practical latency and power/energy consumption.

In addition, cryptographic modules sometimes require countermeasures against implementation attacks which retrieve secret key by tampering or physical methods. Countermeasures for software implementation are limited because software implementation cannot change datapath and should perform cryptographic operations by the existing instruction set. On the other hand, cryptographic hardware can employ many countermeasures at levels of hardware algorithm, gate, and transistor. Thus, cryptographic hardware can achieve higher resistance than software.

Furthermore, cryptographic hardware is widely used as a root-of-trust for information systems since it is basically difficult to modify the manufactured IC chips. Actually, Trusted Platform Module (TPM) chips, which are one of ubiquitous secure coprocessor and are exploited as a root-of-trust for many systems, are on the basis of cryptographic hardware.

While semi-conductor and computing technologies are being developed, many studies have been devoted to cryptographic algorithms that can be easily and efficiently implemented on specific devices. For example, ARX-based ciphers [11, 74] perform encryption and decryption only by arithmetic Addition (i.e., addition over $\mathbb{Z}/m\mathbb{Z}$), Rotation, and bit-parallel XOR in order to achieve lower latency than AES on CPUs and microcontrollers without AES-NI[*3]. In addition, lightweight ciphers, which have higher performance than AES and ISO/IEC 18033 standard ciphers in a specific aspect such as latency, area, and program memory size, have been developed since 2002, the standard of lightweight ciphers ISO/IEC 29192-2 was published in 2012.

[*3] AES-NI (AES New Instructions) is an extension of x86 instruction set (i.e., hardware support) for AES operations and is implemented on some CPUs made in Intel and AMD. The usage of AES-NI makes AES encryption and decryption very fast. In addition, AES-NI is also useful for eliminating a cache timing leak, which is a kind of side-channel leak exploited by the SCA attackers [141].

2.3.5 Implementation attacks and countermeasures

Implementation attacks are attack on cryptographic modules using tamper or physical methods. Originally, cryptographic algorithms are designed based on the model where the attackers can access only ciphertexts (and plaintexts). If we use cryptographic algorithms whose security is comprehensively evaluated by evaluation committees, the attackers cannot retrieve secret information by eavesdropping/modifying ciphertext on the communication channel. However, to use cryptographic algorithms, cryptographic algorithms should be implemented as cryptographic modules, and the attackers are likely to access the modules physically to retrieve secret information. While it is difficult to break cryptographic algorithms, the cryptographic implementation can be critical vulnerability; and therefore implementation attacks are attracting much attention. Implementation attacks requires physical access to target device. However, IoT applications motivate implementation attacks because some devices (e.g., smart card) can be bought and belong to the attackers, and may not be under users' observation. Thus, implementation attacks and their countermeasures are major topics in information security.

Implementation attacks are classified into invasive and non-invasive attacks. Invasive attacks inspect and/or modify internals of cryptographic modules. Invasive attacks are quite powerful attacks because the attacker directly observe ROMs and signals on wire/bus. However, invasive attacks sometimes require impractical costs such as domain knowledge and extremely expensive equipments. To defeat invasive attacks, for example, a countermeasure gathers components for cryptographic operations into a single chip to prevent ROM analysis and wire probing. Other countermeasure employs an active shield to detect the invasion.

On the other hand, non-invasive attacks are performed through irregular I/O without invading internals of chip. Non-invasive attacks are considered as more practical threat than invasive one because it is more difficult to detect non-invasive attacks and they can be performed with commercial and off-the-shelf equipments.

SCA [32, 71] and Fault Injection Attack (FIA) [16, 20] are typical non-invasive attacks[*4]. SCA retrieves secret key by observing and analyzing side-channel information such as power consumption, EM radiation, and computation time. For example, Differential Power Analysis (DPA) and Differential EM Analysis (DEMA) analyze a number of respective power and EM traces during encryption or decryption by a statistical means, namely, calculating the correlation between side-channel information and guessed secret keys. DPA and DEMA are considered as ones of the most powerful SCA which can be applied to most symmetric ciphers. On the other hand, the FIA

[*4] In this dissertation, we classify FIA into non-invasive attacks because it does not use permanent faults and some FIAs can be performed without unsealing chips. FIA is sometimes called semi-invasive attack.

attackers first inject faults into cryptographic modules by physical means (e.g., laser irradiation and overclock), obtain faulty ciphertexts, and then estimates the secret key from the faulty ciphertext. Here, the fault indicates temporary misoperation of cryptographic modules such as bit-flip of intermediate values.

Many countermeasures against the above attacks have been developed. Masking is a typical countermeasure against SCA, which apply random masks to intermediate values in order to decorrelate side-channel information and secret information [65, 103, 114, 122, 140]. Hiding is another typical countermeasure, which makes side-channel information constant regardless of operating intermediate values such as Dual-rail with Precharge Logic (DPL) [37,119,138]. Countermeasures against FIA basically exploit redundancy (e.g., duplication and error-correction codes) for fault-tolerance or fault detection. As mentioned before, cryptographic hardware can achieve higher resistance than software by using dedicated datapath, architectures, and cells. Major SCA-leakage (especially, DPA) comes from datapath, and countermeasures against FIA involve modification of datapath. Therefore, to design cryptographic hardware resistant to SCA and/or FIA, the designer should apply proper countermeasure(s) against expected attacks to datapath, which is mainly composed of GF arithmetic circuits.

2.4 Design of Arithmetic Circuits

2.4.1 Overview

The increasing size and complexity of LSI systems introduce EDA tools to LSI design. EDA tools have been developed with transition of key technologies of LSI design in the order of mask-pattern, transistor-level, and gate-level. Since the 1980s, high-level design methodology based on HDLs and logic synthesis have been the mainstream of LSI design. Figure 2.2 shows the LSI design flow using HDLs and logic synthesis. HDLs directly describe architectures, behaviors, and algorithms of hardware. Then, logic synthesis automatically synthesize the gate-level description corresponding to the HDL code.

In the high-level design, circuits are designed in a top-down manner from higher-level to lower-level of abstractions. Currently, LSIs are designed with two major stages: front-end and back-end. The front-end stage generates an HDL code for gate-level netlist which describes connection between each cell (i.e., gate) using logic synthesis tool and higher-level HDL description. In other words, the front-end stage designs a hardware algorithm to realize the functionality of the designing LSI system. On the other hand, the back-end stage generates physical layout data form the gate-level netlist using back-end tools such as Place and Route (P&R) and floor planning. While the back-end stage is closely related to the physical properties of transistors and semi-conductor

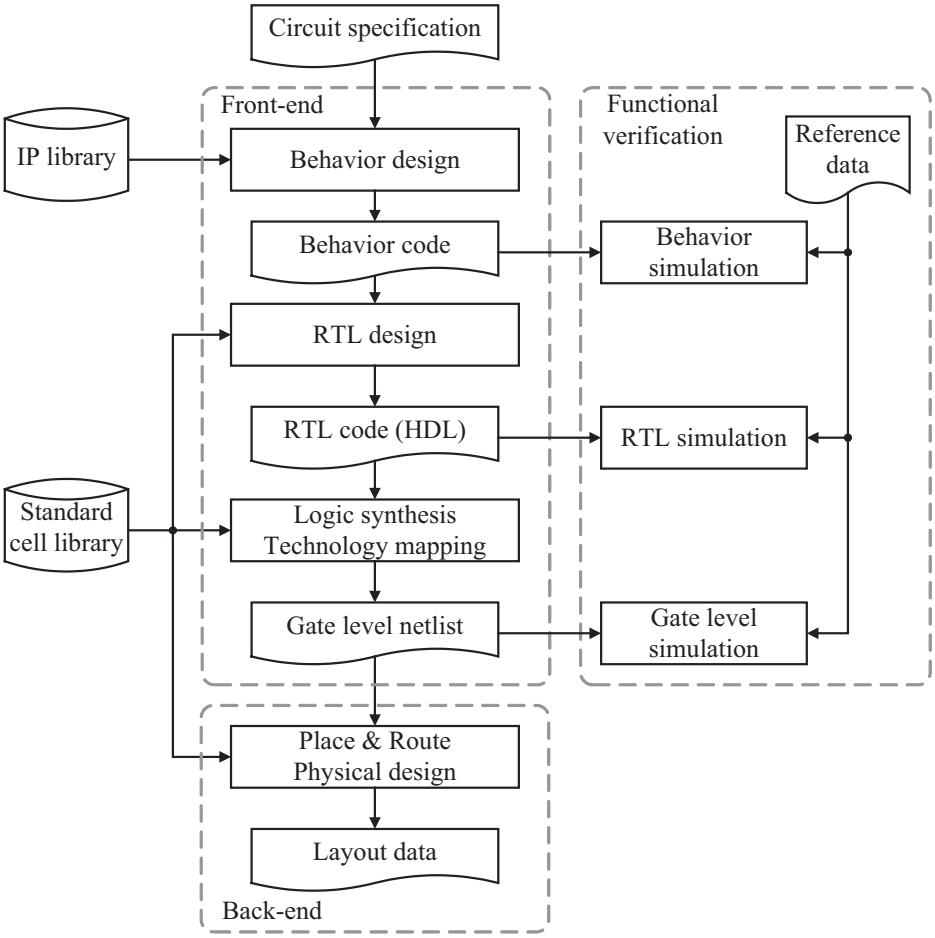


Fig. 2.2 LSI design flow.

manufacturing processes, the performance of arithmetic circuits is determined by hardware algorithm (i.e., arithmetic algorithm) in addition to the processes. Since the performance of LSI system heavily depends on that of arithmetic circuits, arithmetic algorithm design in the front-end stage is quite important. In the following, we focus on the front-end stage design.

In the front-end stage, circuits are described by HDLs with various levels of abstractions where gate-level description is the lowest-level. Typically, abstractions of HDL description are given by gate-level, Register Transfer Level (RTL), and behavior-level. We briefly explain each level in the ascending order of abstraction levels.

- **Gate-level**

  Gate-level description is the lowest-level description in HDL, and describes circuits as netlists of connected logic gates. A gate-level description targets a specific technology. In other words, a gate-level description use logic gates in the library of target technology to description a circuit. On the other hand, description based on logical expression instead of

logic gates in the library is also available. Such description is sometimes called logic-level. Given a technology, logic-level description is translated to the gate-level one by technology mapping.

- **RTL**
  RTL is a higher level than gate-level. RTL description explicitly describes registers and dataflow (i.e., transfer) between registers. The dataflow is described as a combinational circuit consisting of multiplexer, decoder, and arithmetic operators, among other things. Note that word-level arithmetic operators are available only if the used HDL and EDA tools support the high-level data structure for them. Logic synthesis tools can usually generate the gate-level description form an RTL description. The stages after generating RTL description are almost automated by EDA tools.

- **Behavior-level**
  Behavior-level is a higher level than RTL, and describes the behavior of LSI or the algorithm (sometimes like software). Behavior-level description is sometimes given by not HDL, but C++ and C. In this level, circuits are described by a processing flow based on arithmetic formulae without defining architecture given by registers, multiplexers, and Finite State Machines (FSMs). Then, high-level synthesis tools transform behavior-level description into the RTL where resources are assigned and scheduled as optimally as possible. However, high-level synthesis is not always available. In such case, the behavior code in Fig. 2.2 is not directly input to the EDA tool, and each module is designed in RTL description by hand.

The usage of high-level description makes it unnecessary to describe the structural detail of circuits by hand, which leads to simplification of circuit description, efficient debugging, and the reduction of design costs. On the other hand, the designer should describe the arithmetic circuits if the arithmetic circuits provided by EDA vendors do not satisfy required performance. In addition, high-level description and automatic synthesis are unavailable unless the EDA vendor has provided the high-level data structures.

2.4.2 Problems on designing cryptographic hardware

As mentioned in the previous subsection, the designer should describe the arithmetic circuits by hand if the arithmetic functions are not supported by EDA tools. In addition, although logic synthesis tools are useful for shortening the duration of design, the designer should design dedicated arithmetic algorithms if the logic synthesis tool cannot achieve required performance. Note

that, while logic synthesis tools can efficiently compress random logic in many cases, they are not suitable to circuit with regularity including arithmetic circuits.

Thus, the conventional EDA tools have the following two major problems on designing cryptographic hardware. (a) There are few EDA tools that support GF arithmetic, and (b) GF arithmetic circuits should be designed using abstract lower than logic-level. These problems forces designers to describe the structural detail of cryptographic hardware (i.e., GF arithmetic circuits) with massive low-level (i.e., logic-level) expressions by hand, which lead to the long duration and high cost of design of cryptographic hardware. In addition, it is difficult to apply the conventional high-level design methodology for integer ring arithmetic because arithmetic rules of GF can differ depending on the modular polynomial and GF representation given a bit length. More precisely, given a bit length and GF arithmetic operation, circuit structure in logic-level is not uniquely determined, and there is a number of variation of the circuit structures.

In summary, the conventional high-level design methodology (i.e., EDA tools) has difficulty in designing cryptographic hardware based on GF arithmetic; and therefore, new design method dedicated for GF arithmetic circuits is highly demanded.

2.5 Functional Verification of Arithmetic Circuits

2.5.1 Overview

In LSI design, functional verification should be required in order to check whether the circuit description satisfies the circuit specification. Due to the increasing size and complexity of LSI systems, the cost of functional verification is said to account for more than 80% of the total design cost [46]. In addition, it is also said that ASIC respins are mostly caused by bugs in RTL and gate-level description of HDL, that is, omission of verification in front-end stage. In the top-down approach of high-level design, it is quite important to remove bugs in earlier stages in order to reduce design costs. In the following, we describe the functional verification at RTL and gate-level.

2.5.2 Related works

There are three types of functional verification of LSI: equivalence checking with reference, property checking, and theorem proving. Among them, equivalence checking with reference is basically used for arithmetic circuits. Figure 2.3 shows the overview of the functional verification using reference, which are basically given by a table representing input-output relation (i.e., test data) or circuit whose functionality is guaranteed (i.e., golden model). The equivalence indicates the logical equivalence of target circuit and reference. In other words, the outputs are always

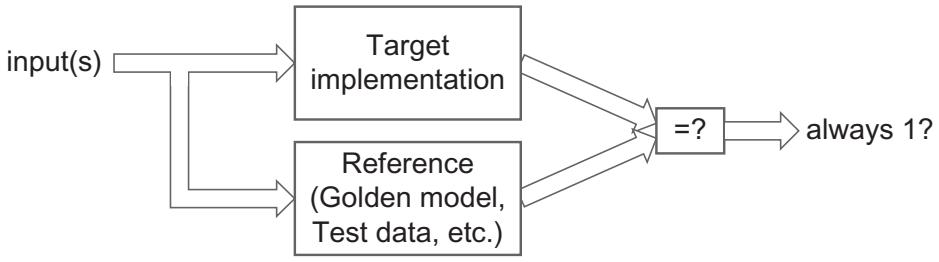


Fig. 2.3 Overvew of functional verification using reference.

equivalent for any expected input. In the following, we describe two conventional methods for the equivalence checking: (i) logic simulation and (ii) formal verification.

(i) Logic simulation

Logic simulation is a classical and the most widely used method. In logic simulation, the target circuit (and reference circuit) is simulated on software (e.g., verilog-XL), in order to check whether the responses and behaviors are valid and correct. Reference is sometimes given by a table representing input-output relation. If we perform an exhaustive verification using logic simulation, the verification time increases exponentially by input bit length. Therefore, logic simulation cannot verify circuits with large input completely in practice. As another method similar to logic simulation, emulation using a reconfigurable device (e.g., FPGA) can perform exhaustive verification several thousands times faster than logic simulation. Nevertheless, the achievable verifcation coverage of emulation would be insufficient for some circuits with large inputs.

Cryptographic algorithms basically have more than 64-bit operands, which indicates that logic simulation requires to check at least 2^{64} responses to verify cryptographic hardware completely. In particular, cryptographic algorithms are designed to be secure against an brute force attack (i.e., exhaustive search of secret key), which implies that exhaustive verification is radically impossible. For example, exhaustive verification of AES hardware requires to check 2^{256} responses (i.e., 128-bit plaintext and 128-bit secret key), which is obviously infeasible. Actually, NIST published AES Algorithm Validation Suite (AESAVS) which specifies the procedures (i.e., tests) involved in validating AES implementations, but the successful validation by AESAVS does not imply complete conformance of the AES implementation because AESAVS employs a Monte Carlo test using only a small number of possible cases, as stated in AESAVS [10]. In addition, although NIST also published the specifications of validating implementation of other ciphers, some issues on detecting bugs by them have been pointed out [145].

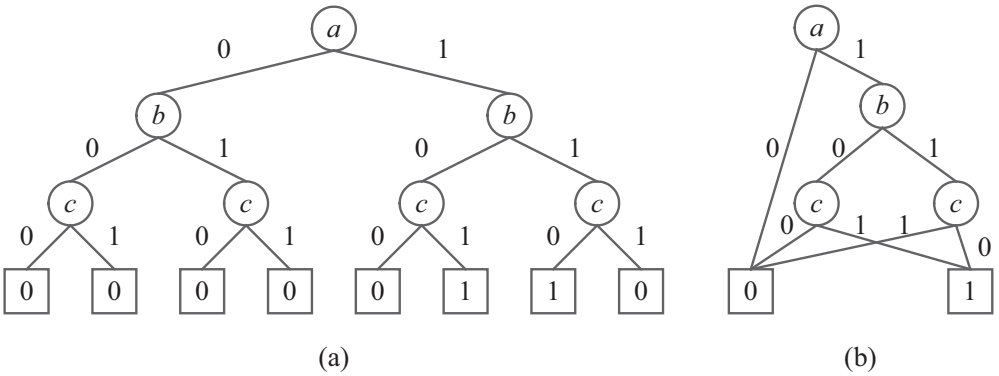


Fig. 2.4 (a) BDD and (b) ROBDD for $a \wedge (b \oplus c)$.

(ii) Formal verification

Formal verification methods verify arithmetic circuits by giving a proof that the target circuit satisfies the properties (e.g., correctess of functionality) which are described by mathematical formulae. Formal verification has a potential for solving the above problem on verifcation coverage, and has been studied mainly in academia.

Formal verification for the equivalence checking first transform the target circuit and reference into the identical data structure (e.g., Boolean expression), and then check whether they are mathematically equivalent. There are three conventional methods based on (1) Binary Decision Diagram (BDD), (2) SATisfiability (SAT) solver, and (3) PPRM expansion.

(1) BDD is a data structure for logic functions $f : \mathbb{B}^n \rightarrow \mathbb{B}$ (where $\mathbb{B} = \{0, 1\}$), and has been mostly used for formal verification of arithmetic circuits. For example, Fig. 2.4(a) shows a BDD for $a \wedge (b \oplus c)$, where $\wedge$ and $\oplus$ denote OR and XOR operators, respectively. In Fig. 2.4, each node corresponds to a logic variable, and weight of each edge under a node corresponds to assignment of the variable to 0 or 1. Therefore, leaves correspond to the output of the representing function, and we can know the output for an input by following edges downwardly from the root (i.e., node on the top).

It is difficult to use the BDD in Fig. 2.4 for formal verfication because BDDs for the identical function can differ depending on the expression of logic function. Therefore, Reduced Ordered BDD (ROBDD) is used for formal verification. Figure 2.4(b) shows the ROBDD for f. ROBDD is a canonical form of BDD where redundant variables are removed and the order of variables are fixed. In other words, the circuit function can be verified by isomorphic dicision of two ROBDDs for target and reference circuits because two ROBDDs have the identical form if and only if they are equivalent.

The size of ROBDD (i.e., the number of nodes in ROBDD) has a big impact on the verification

time based on ROBDD. The size of ROBDD depends on the variable order and the representing function. For example, there is an optimal veriable order for two-input Ripple Carry Adders (RCAs), which makes ROBDD represent RCAs with a size increasing polynomially by the input bit length. On the other hand, since it was proven that the size of ROBDD for multipliers always increases exponentially by the input bit length for any variable order, it is different to verify large multipliers using ROBDD [29].

(2) Since the equivalence checking can be easily resolved into an SAT problem, SAT solvers are sometimes used for formal verification. Many SAT solvers for formal verification transform the circuits into logic formulae in the Conjunctive Normal Form (CNF), and then examine satisfiability of the logic formulae. State-of-the-art SAT solvers can solve examplary SAT problems with several millions variables and handreds terms in several hours, which correspond to some circtuis with several ten thousands gates if the SAT problem is suitable to the SAT solver. On the other hand, the SAT solvers have difficulty in solving some SAT problems (e.g., a part of randomly generated ones) even with only several hardreds variables.

Finally, (3) PPRM expansion is a dedicated method for equivalence checking of arithmetic circuits over $GF(2^m)$ in contrast to other methods. In the method, we represent input-output relations of the circuits with logic equations, and then check whether the logic equations of reference circuit logically imply those of target circuit. The check is performed by expanding the logic equations to the PPRM form, which is a canonical form based on AND-XOR. Since most arithmetic circuits over $GF(2^m)$ mainly consists in AND and XOR gates, PPRM would be suitable to handle the circuits. In [95], it was shown that this method can be efficiently applied to $GF(2^m)$ arithmetic circuits described in logic-level, and successfully verifeid some components of AES hardware in a few seconds.

The above methods can reduce the verification time by exploiting similarities of targe and reference circuits. If an equivalent signal between two circuits is detected, it can be considered as the same signal. Thus, using the similarities (i.e., by checking equivalence of only different points), the above formal verification methods can verify larger circuits than the straight-forward approach.

2.5.3 Problems on verifying cryptographic hardware

BDDs and SAT solvers have difficulties in verifying practical cryptographic hardware. Most cryptographic algorithms use addition, multiplication, and inversion over GFs in a combination. As mentioned before, the size of BDD (i.e., verification time) increases exponentially by the input bit length. In addition, inversion circuits has more complicated structure than multipliers. Thus, huge GF arithmetic circuits (i.e., cryptographic hardware) cannot be verified by BDDs in a

practical time. Actually, in [80], it was shown that a BDD could not verify GF multipliers whose input bit lengh is more than 22. Recently, BDD is extended to Binary Moment Diagram (BMD), which is a word-level graph structure [30]. Multiplicative BMD (*BMD) is a canonical form of BMD which can be exploited by formal verification of arithmetic circuit. However, *BMD have been mainly developed for integer ring arithmetic, and there are a few reports on *BMD for GF arithmetic. The existing *BMDs for GF arithmetic are limited, are applied to only specific GFs and circuits, and have difficulty in handling practical ones [94, 98, 135].

In addition, while many SAT solvers focus on CNF (i.e., AND-OR form), AND-XOR form can efficiently compress logical expression of many cryptographic algorithms [96]. In other words, CNF is not suitable to represent cryptographic algorithms, and such SAT solvers are not suitable to formal verification of cryptographic hardware. Although a SAT-solver called CryptoMiniSat has been developed for cryptographic applications, no SAT-solvers including CryptoMiniSat can verify more than 16-bit GF multipliers [80].

Note that it is also difficult to reduce verification time using similarities because pairs of arithmetic algorithms frequently have few similarities, and it is sometimes difficult to detect equivalent signals.

Although PPRM expansion succeeded in verifying some AES components such as S-box and MixColumns circuits using the corresponding reference circuits, it would be applicable to word-level verification and huge circuits (e.g., whole of cryptographic datapath).

Finally, the most fatal problem of the above methods is that the methods should require reference circuits (i.e., golden model), which are usually not prepared for sophisticated functions and cryptographic datapths. Thus, formal verification which does not require golden model is quite important.

2.6 Formal Design of GF Arithmetic Circuits

2.6.1 Overview

Formal design method of GF arithmetic circuit was proposed in [63, 64, 110]. In this method, GF arithmetic circuits are represented with a hierarchical and mathematical graph called GF-ACG. In GF-ACG, the functions of GF arithmetic circuit can be represented by GF equations including word-level. The usage of GF-ACG enables us not only to describe GF arithmetic circuit with various levels of abstraction, but also verify its functions formally by equivalence checking between hierarchies (i.e., formula evaluation) without any reference circuit. For the formal verification, an algebraic verification procedure with a GB and polynomial reduction technique has been introduced in [63]. We can completely verify 256-bit GF multipliers and AES (encryption) hardware

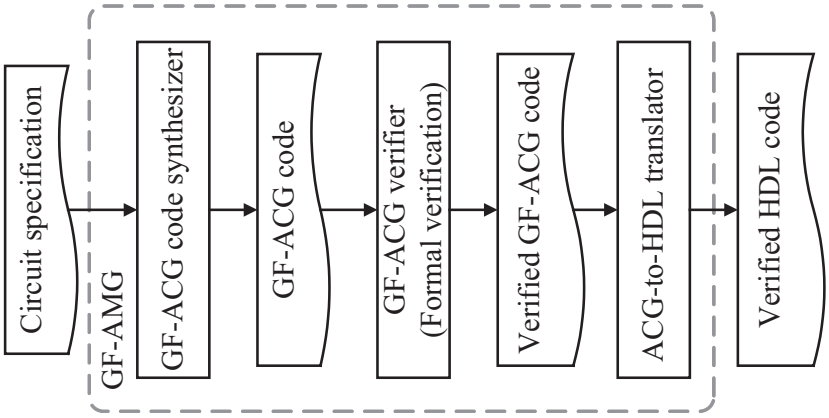


Fig. 2.5 Block diagram of GF-AMG.

using GF-ACG in a practical time.

Actually, GB has been studied in the field of cryptography. Other algebraic formal verification without reference circuit was reported in [79–81]. In [80], formal verification of $GF((2^{m_0})^{m_1})$ was shown. In [79, 81], methods for optimizing parameters in computing GB (i.e., term order and variable order) were proposed. However, these method was applied to only PB-based GF multipliers, and not applied to circuits based on other GF representations and cryptographic hardware, which have higher-degree functions with a large number of variables and terms. They have a significant impact on computation time of GB.

2.6.2 Automatic generation system for GF arithmetic circuit

GF Arithmetic Module Generator (GF-AMG) is an automatic generation system for $GF(2^m)$ multipliers, is a main application of GF-ACG [109], and is open to public via our website [1]. Given multiplier specifications, GF-AMG generates only HDL code whose function is verified based on GF-ACG. GF-AMG would be useful for the designers even unfamiliar with GF arithmetic in order to reduce design cost of GF arithmetic circuits.

Figure 2.5 shows the block diagram of GF-AMG, which consists of (i) GF-ACG code synthesizer, (ii) GF-ACG verifier, and (iii) ACG-to-HDL translator. The GF-ACG code synthesizer generates GF-ACG code according to the users' design specification. In [109], specifications are given by bit length, irreducible polynomial, and GF representation chosen from PB and NB. The GF-ACG verifier proceeds to formally verify the generated GF-ACG code. The ACG-to-HDL translator finally translates the verified GF-ACG code to the corresponding HDL code. The system in [109] supports generation of Mastrovito and Massey-Omura multipliers, which are typical multipliers over PB- and NB-based GFs, respectively.

2.6.3 Issues on formal design of cryptographic hardware

GF-ACG can be efficiently applied to basic arithmetic circuits such as two-input multipliers and straight-forward AES hardware (i.e., without optimization). On the other hand, in practice, more complex circuits and various optimization techniques are used in designing cryptographic hardware. For example, resister-retiming and operation-reordering/merging are quite useful for reducing critical delay and circuit area. However, the existing GF-ACG has restriction of handleable arithmetic algorithms and hardware architectures, and has difficulties in designing many practical cryptographic hardware.

In the following, we describe some issues on designing practical GF arithmetic circuit and cryptographic hardware design by GF-ACG. Firstly, the existing GF-ACG focuses on only non-redundant GF representations (i.e., PB and NB), and does not support redundant representations. Since some redundant GF arithmetic algorithms have higher performance than non-redundant ones [101], it is quite important to design GF arithmetic circuits based on redundant GF representations.

Secondly, although pipelining is used not only for enhancing throughput and implementation efficiency, but also as a countermeasure against DPAs, the existing GF-ACG cannot represent such sequential circuits. Nevertheless, the conventional GF-ACG does not represent time modality (i.e., clock cycle) and sequential arithmetic circuits including pipelined ones because they focuses on only combinational circuits that performs the computation in a single clock cycle. Needless to say, countermeasure against DPAs should be sometimes mandatory for cryptographic hardware.

Thirdly, some hardware architectures and optimization techniques are unavailable due to verification time based on GB. The computation time of GB heavily depends on the degree and number of variables of circuit function. Many cryptographic algorithms employ high-degree functions such as inversion over GF, which sometimes make computation of GB infeasible. For example, AES decryption hardware cannot be verified by the existing GF-ACG because AES decryption is represented by higher-degree functions than encryption. In addition, resister-retiming and operation-reordering sometimes lead to the increase of function degree, which indicates that some AES encryption hardware cannot be also verified by the existing GF-ACG. Moreover, while well-hierarchical representation of GF-ACG is useful for reducing verification time (i.e., computation time of GB), it is difficult to represent circuits optimized at logic-level in a hierarchical manner. In such cases, computation of GB would be infeasible due to the increase of the number of variables. For example, AES hardware with a masking-based countermeasure cannot be verified in a practical time.

In summary, the design space of the existing GF-ACG is very limited, and many practical cryptographic hardware is out of the design space. The formal design method which covers a

wider design space is quite desirable.

2.7 Conclusion

Modern cryptographic algorithms are closely related to GF arithmetic, and a lot of next-generation encryption schemes are also on the basis of GF arithmetic. In addition, Cryptographic hardware is essential for resource-constraint devices, tamper-resistance, and root-of-trust. Thus, it is quite important to design and verify GF arithmetic circuits for cryptographic hardware. Nevertheless, the conventional EDA tools have difficulties in designing and verifying GF arithmetic circuits. While the formal design based on GF-ACG is a promising method, it still has some issues on designing practical cryptographic hardware.

3

Formal Design Methodology of Cryptographic Hardware

3.1 Introduction

Chapter 2 described the increasing importance of GF arithmetic circuits for cryptographic hardware. This chapter presents a new formal design methodology which can handle and rapidly verify various GF arithmetic circuits including practical cryptographic hardware described before.

In this chapter, we first introduce the conventional GF-ACG and its formal verification by the algebraic procedures as previous works.

We then propose the formal design methodology of cryptographic hardware. We first proposed a new GF-ACG which can handle redundant GF representations in addition to non-redundant ones, and also can handle registers and clock cycles on the basis of a modal logic. On the other hand, since the proposed GF-ACG cannot be directly applied to the following verification due to redundantly represented values and the modal logic representation, we also describe methods for sound and complete verification.

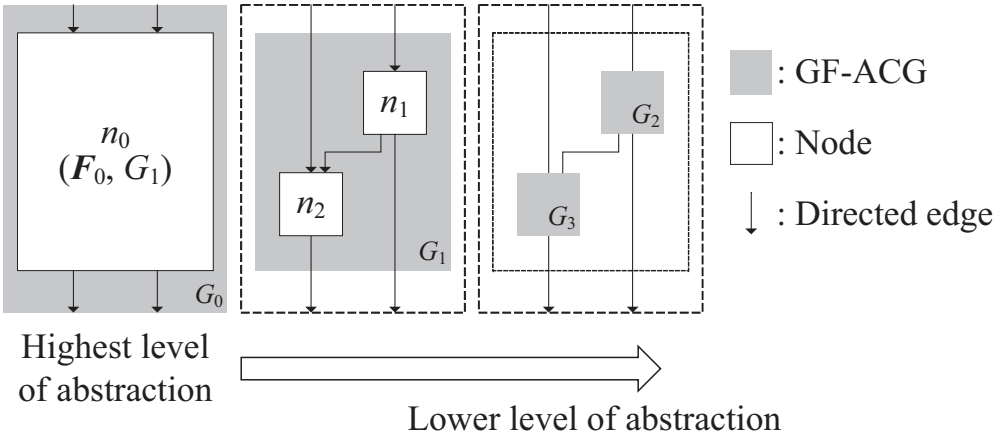


Fig. 3.1 Overview of GF-ACG

Next, we proposed new formal verification of GF arithmetic circuits for cryptographic hardware. We propose two formula evaluation methods for verifying GF-ACG. One is based on the natural deduction for the first order predicate logic, and can rapidly verify higher-degree functions frequently used in cryptographic hardware. The other is based on a combination of PPRM expansion and the conventional GB-based method, and can rapidly verify circuits with logic-level optimizations. We then propose an efficient and complete verification algorithm which combines the above formula evaluation methods including the conventional one.

To demonstrate the effectiveness and efficiency of the proposed methodology, we show its applications to various circuits, namely, PRR-based multipliers, AES encryption and decryption hardware, Masked AES hardware, and a DPA-resistant cryptographic hardware utilizing pipeline.

Finally, we discuss the generality and applicability of the proposed methodology.

3.2 Previous Work

3.2.1 Formal Description of GF Arithmetic Circuits

Figure 3.1 shows the overview of GF-ACG. A GF-ACG G is an directed graph and is defined by

$$G = (\boldsymbol{N}, \boldsymbol{E}), \tag{3.1}$$

where $\boldsymbol{N}$ and $\boldsymbol{E}$ denote sets of nodes and directed edges, respectively. A node represents an arithmetic circuits by its functional assertion and internal structure. A directed edge represents a dataflow between nodes associated with a GF variable. Since each node has its internal structure represented by GF-ACG, GF-ACG represents a circuit in a hierarchical manner.

A node $n\ (\in \boldsymbol{N})$ is defined by $(\boldsymbol{F}, G_{in})$, where $\boldsymbol{F}$ is the functional assertion given be a set of GF

equations and G_{in} is the internal structure given by a low-level GF-ACG. A functional assertion is represented using GF equations (formulae) $F_l = F_r$, where F_l and F_r are the respective output and input expressions and each expression is given by variables, constants, or combinations of them connected by the arithmetic operators $+$, $-$, and $\times$.

A node at the lowest-level of abstraction, which does not possess an internal structure, is defined by $(\boldsymbol{F}, nil)$. The lowest-level nodes have functions which are assumed to be correct; and therefore, nodes which do not have their internal structures represent cells (including standard and custom ones), logic gates, or IP cores.

A directed edge $e \in \boldsymbol{E}$ is defined as $(n_s, \boldsymbol{n}_e, v)$, where n_s is the start node, $\boldsymbol{n}_e$ is the set of end nodes, and v represents a variable of GF. The directed edge represents an external input or output for the given GF-ACG if either n_s or $\boldsymbol{n}_e$ is nil. A GF variable is associated with a GF. Let $GF_{(r)}$ be a formal representation of $GF_{(r)}$. A $GF_{(r)}$ is represented as $(\boldsymbol{B}, \boldsymbol{C}, IP)$, where $\boldsymbol{B}$ denotes the basis, $\boldsymbol{C}$ is the coefficient vector, and IP is the irreducible polynomial. If $GF_{(r)}$ represents an mth-degree extension field of $GF(q)$ (i.e., $GF_{(r)} = GF_{(q^m)}$), $\boldsymbol{B}$, $\boldsymbol{C}$, and IP are given by

$$\boldsymbol{B} = (B_{m-1}, B_{m-2}, \ldots, B_i, \ldots, B_0), \tag{3.2}$$

$$\boldsymbol{C} = (\boldsymbol{C}_{m-1}, \boldsymbol{C}_{m-2}, \ldots, \boldsymbol{C}_i, \ldots, \boldsymbol{C}_0), \tag{3.3}$$

$$IP = \beta^m + c_{m-1}\beta^{m-1} + \cdots + c_i\beta^i + \cdots + c_0\beta^0, \tag{3.4}$$

where B_i $(0 \leq i \leq m-1)$ denotes a basis element (i.e., $B_i = \beta^i$ for PB and $B_i = \alpha^{q^i}$), $\boldsymbol{C}_i$ is the coefficient set of degree i (i.e., set of elements of $GF(q)$), and $c_i \in GF(q)$ is an element of $\boldsymbol{C}_i$. On the other hand, if $GF_{(r)}$ represents a prime field $GF(p)$, we let $\boldsymbol{B} = 1$, $\boldsymbol{C} = (\{0, 1, \ldots, q-1\})$, and $IP = nil$. Thus, $GF_{(r)}$ can represent both prime and extension fields including tower fields. A GF variable v is defined as $(GF_{(q)}, (h, l))$, where h and l denote the most and least degrees of v, respectively. The ordered pair (h, l) is called the degree range. Using the above notation, we handle a specific variable v_d of degree d. Note that every GF variable satisfies the field equation $v^\omega = v$ (ω is the order of field), which is mandatory in order to make the following verification complete.

A GF variable can be decomposed/composed into an expression with sub-variables at a lower/higher level of abstraction. In other words, we can change the level of abstraction in the edge representation using decomposition/composition nodes. A decomposition node decompose v into $h - l + 1$ GF variables. There are two types of decomposition nodes. For example, Fig. 3.2 shows two decomposition nodes for a GF variable v with a degree range $(1, 0)$, where

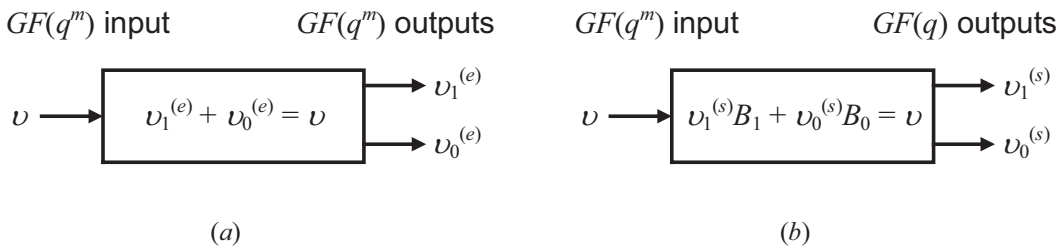


Fig. 3.2 Decomposition nodes with functional assertion given by (a) Eq. 3.5 and (b) Eq. 3.6.

the functional assertions of decomposition nodes of (a) and (b) are generally given by

$$\upsilon_h^{(e)} + \upsilon_{h-1}^{(e)} + \cdots + \upsilon_l^{(e)} = \upsilon, \tag{3.5}$$

$$\upsilon_h^{(s)} B_h + \upsilon_{h-1}^{(s)} B_{h-1} + \cdots + \upsilon_l^{(s)} B_l = \upsilon, \tag{3.6}$$

respectively. The decomposition node (a) decomposes $\upsilon \in GF(q^m)$ into $\upsilon_d^{(e)}$s ($l \leq d \leq h$). The GF variable $\upsilon_d^{(e)}$ is an element of $GF(q^m)$, and is formally represented by $((GF_{(q^m)}, (d, d)))$. On the other hand, the decomposition node (b) decomposes $\upsilon \in GF(q^m)$ into $\upsilon_d^{(s)}$s. The GF variable $\upsilon_d^{(s)}$ is an element of $GF(q)$, and is formally represented by $(GF_{(q)}, (h', l'))$, where (h', l') denotes the degree range of decomposed variable. Thus, while decomposition node (a) does not change the abstraction level of the input GF variable in terms of GF representation[*1], decomposition node (b) is used for changes the abstraction level of a GF variable into lower-level (i.e., subfield). In addition, we define composition nodes by exchanging the input and output of decomposition nodes. Note that decomposition and composition nodes are the lowest-level node (i.e., do not have internal structure) because change of abstraction should be performed by only wiring.

The above GF-ACG can represent GF arithmetic circuits based on PB and NB (i.e., non-redundant representations). In addition, GF-ACG can also represent any kind of logic circuits. Logic signals in logic circuits are represented by pseudo-logic variables, which is a GF variable that can take 0 or 1. Let $Logic$ be the field consisting of pseudo-logic variables, and u be a

[*1] On the other hand, decomposition node (a) changes the abstraction level of GF variable into lower-level in terms of degrees (or digits) of GF variables because GF variables after decomposition have constraints about their degree ranges compare to the input GF variable

pseudo-logic variable. $logic$ and u are given by

$$Logic = ((1), (\{0, 1\}), nil), \tag{3.7}$$

$$u = (Logic, (0, 0)). \tag{3.8}$$

In addition, every pseudo-logic variables should meet the idempotent constraint $u^2 = u$ instead of field equation. Using such a pseudo-logic variables, GF-ACG can represent logic circuits in the similar manner to GF arithmetic circuits. The functional assertions of nodes for typical logic operations $\mathrm{NOT}(u)$, $\mathrm{OR}(u, v)$, $\mathrm{AND}(u, v)$, and $\mathrm{XOR}(u, v)$ are given by

$$\mathrm{NOT}(u) = 1 - u, \tag{3.9}$$

$$\mathrm{OR}(u, v) = u + v - uv, \tag{3.10}$$

$$\mathrm{AND}(u, v) = uv, \tag{3.11}$$

$$\mathrm{XOR}(u, v) = u + v - 2uv, \tag{3.12}$$

respectively. Note that the above representations of logic operations does not depend on GF. Although we showed typical logic operations, we can also represent any kind of logic circuits, including negative logic gates and functional gates. Thus, we can describe arithmetic circuits hierarchically from logic-level to word-level of abstractions.

3.2.2 Formal Verification of GF-ACG

The usage of GF-ACG enables us to formally verify GF arithmetic circuits by equivalence checking between hierarchies. Since circuit functions are represented by GF equations, the equivalence checking is resolved into formula evaluation. So far, the formula evaluation has been performed the automatic algebraic procedures based on GB and polynomial reduction.

Algorithm 3.1 describes the GF-ACG verification in an algorithmic manner, which was presented in [63]. Algorithm 3.1 verifies that the functional assertion of the highest-level node is correctly derived form GF-ACG of the internal structure, assuming that the functions of lowest-level nodes are correct. Given a GF-ACG, Algorithm 3.1 recursively applies the verification to its internal structures. In Line 6, the formula evaluation "GB-BasedEvaluation" is applied to all nodes with an internal structure. GB-BasedEvaluation returns true if and only if the functional assertion and internal structure of the input node are equivalent. On the other hand, no procedures are applied to the lowest-level nodes because their functional assertion are assumed to be correct. Thus, Algorithm 3.1 returns true if and only if all the nodes excluding the lowest-level are equivalent to their internal structures, and the function of the highest-level node is correctly realized by the lowest-level nodes (e.g., logic circuit).

Algorithm 3.1 GF-ACG verification

Input: GF-ACG $G = (\boldsymbol{N}, \boldsymbol{E})$
Output: Verification result $res \in \{\text{true}, \text{false}\}$

```
1: function VERIFY(G)
2:     Bool res ← true;
3:     for all n = (F, G_in) ∈ N do
4:         if G_in ≠ nil then
5:             res ← res & Verify(G_in);
6:             res ← res & GB-BasedEvaluation(n);
7:         end if
8:     end for
9:     return res;
10: end function
```

We then describe formula evaluation based on polynomial reductions by GB (i.e., GB-BasedEvaluation). Since the functional assertions are given by GF-ACG, we can perform the equivalence checking by checking whether the functional assertions (i.e., GF equations) of a node are correctly derived from the simultaneous equations representing its internal structure. In [63], this formula evaluation was resolved into an ideal membership problem, which is solved by the algorithm based on polynomial reductions by GB in a systematic manner. In the following, we introduce the ideal, polynomial reduction, and GB before describing the formula evaluation algorithm.

An ideal is defined as a specific sub-ring of a ring[*2]. Let $\mathbb{R}$ and $\mathbb{I}$ be a ring and its sub-ring. $\mathbb{I}$ is an ideal of $\mathbb{R}$ if $c_0 f_0 + c_1 f_1 \in \mathbb{I}$ where f_0, f_1 and c_0, c_1 are arbitrary elements of $\mathbb{I}$ and $\mathbb{R}$, respectively. If $\mathbb{I}$ is represented by $\mathbb{I} = \{\sum c_\zeta f_\zeta \mid c_\zeta \in \mathbb{R}\}$ ($f_0, f_1, \ldots, f_\zeta, \ldots, f_{\ell-1}$ are elements of $\mathbb{I}$), $\mathbb{I}$ is called an ideal generated by $f_0, f_1, \ldots, f_{\ell-1}$, and is denoted by $\langle f_0, f_1, \ldots, f_{\ell-1}\rangle$. $\boldsymbol{I} = \{f_0, f_1, \ldots, f_{\ell-1}\}$ is called ideal basis.

Then, let $\mathbb{R}$ be a multivariate polynomial ring over $GF(\omega)$, and let us consider the following simultaneous equations:

$$\begin{cases} f_0 &= 0 \\ f_1 &= 0 \\ &\vdots \\ f_{\ell-1} &= 0 \end{cases}. \tag{3.13}$$

Here, every element $f \in \mathbb{I} = \langle f_0, f_1, \ldots, f_{\ell-1}\rangle$ satisfies $f = 0$ because f is given in a form of $c_0 f_0 + c_1 f_1 + \cdots + c_{\ell-1} f_{\ell-1}$. Ideal membership problem indicates whether or a polynomial f is a

[*2] Ring is an algebraic structure and a superset of field. A ring is defined as a set where addition, subtraction, and multiplication are defined such as the ring of integer.

Algorithm 3.2 Formula evaluation based on polynomial reduction by GB

Input: Node $n = (\boldsymbol{F}, G_{in})$
Output: Formula evaluation result $res \in \{\mathrm{true}, \mathrm{false}\}$
1: **function** GB-BASEDEVALUATION(n)
2: **set** $\boldsymbol{S} \leftarrow \emptyset$;
3: **for all** $n = (\boldsymbol{F}', G'_{in}) \in \boldsymbol{N}'(\in G_{in})$ **do**
4: $\boldsymbol{S} \leftarrow \boldsymbol{S} \cup \boldsymbol{F}'$;
5: **end for**
6: $\mathbf{GB} \leftarrow$ GröbnerBasisOf($\boldsymbol{S}$);
7: **Bool**$res \leftarrow$ IsIdealMember($\boldsymbol{F}, \boldsymbol{GB}$);
8: **return** res;
9: **end function**

decision problem which decides whether or not a polynomial f is a member of an ideal $\mathbb{I}$. Hence, we can verify a circuit function $f \in \boldsymbol{F}$ by solving the ideal membership problem determined by the simultaneous equations representing internal structure. Note that, in the formal verification, GF equations are considered as polynomials in the form of $\mathrm{lhs} - \mathrm{rhs}$.

Polynomial reduction is a technique for solving ideal membership problems. A polynomial reduction transforms a multiplication polynomial into a remainder divided by other multivariate polynomial. Therefore, we can solve an ideal membership problem by repeating polynomial reductions of f until f cannot be divided by any element generating $\mathbb{I}$, and check whether or not the result is equal to 0. Here, the result is called by the normal form of f by $\boldsymbol{I}$, and is denoted by $\mathrm{NF}_{\boldsymbol{I}}(f)$. If $\mathrm{NF}_{\boldsymbol{I}}(f) = 0$, $f \in \mathbb{I}$.

However, $\mathrm{NF}_{\boldsymbol{I}}(f)$ is not unique and depends on the order of polynomial reduction, and $\mathrm{NF}_{\boldsymbol{I}}(f)$ is sometimes not equal to 0 even if $f \in \mathbb{I}$. The non-uniqueness of normal forms causes false-positive of the verification although soundness is quite important for formal verification. On the other hand, GB is an ideal basis which can make normal forms always unique. Formally, the necessary and sufficient condition that an ideal basis is GB is that $f \in \mathbb{GB}$ if and only if $\mathrm{NF}_{\boldsymbol{GB}}(f)$, where $\mathbb{GB}$ denotes the ideal generated by $\boldsymbol{GB}$. In addition, there is an algorithm called Buchberger's algorithm, which transforms an ideal basis to the equivalent GB in finite steps [31]. In addition, it was proven that Buchberger's algorithm always terminates in finite step. Thus, we can perform sound and complete verification thanks to the uniqueness of $\mathrm{NF}_{\boldsymbol{GB}}(f)$ and total correctness of Buchberger's algorithm. See [40, 41] for details of GB and Buchberger's algorithm.

Algorithm 3.2 verifies functional assertion of a node based on polynomial reduction by GB. In Algorithm 3.2, Lines 3–5 extract simultaneous equations representing the internal structure. Note that equations are considered as polynomials in Algorithm 3.2 as stated before. Line 6 computes

Algorithm 3.3 Decision procedure for ideal membership problem

Input: Functional assertion $\boldsymbol{F}$, Gröbner basis $\boldsymbol{GB}$
Output: Decision $res \in \{\text{true}, \text{false}\}$

```
 1: function IsIdealMember(F, GB)
 2:     Bool res ← true;
 3:     for all f ∈ F do
 4:         if NF_GB(f) = 0 & GB ≠ {1} then
 5:             res ← res & true;
 6:         else
 7:             res ← res & false;
 8:         end if
 9:     end for
10:     return res;
11: end function
```

the GB equivalent to the simultaneous equations using Buchberger's algorithm (or improved one). Line 7 verifies GF equations in $\boldsymbol{F}$ are included in the ideal generated by $\boldsymbol{GB}$ using the algorithm "IsIdealMember," and Line 8 returns the result.

Algorithm 3.3 is a decision procedure for ideal membership problem (i.e., IsIdealMember). Line 4 computes $\mathrm{NF}_{GB}(f)$, where f is an equation in vrF. If and only if $\mathrm{NF}_{GB}(f) = 0$, f is correctly derived from the internal structure. If $\mathrm{NF}_{GB}(f) \neq 0$, the circuit is wrong. Note that, if $\boldsymbol{GB} = \{1\}$, the simultaneous equations have no solution, which indicates that the circuit is wrong[*3].

The computation time of Algorithms 3.2 and 3.3 is mainly determined by that of GB computation. Although a GB computation always terminates, the computation time heavily depends on the degree and number of variables of the simultaneous equations. For example, if the simultaneous equations are linear, the GB computation time increases the cubit of κ, where κ denotes the number of variables. The hierarchical description of GF-ACG reduces the degree and number of variables par GB computation, which leads to the reduction of total verification time. Therefore, we should describe circuits well-hierarchically when verifying huge circuits. On the other hand, there are some cryptographic hardware where the circuit function has very high degree even at word-level (e.g., AES decryption hardware), and/or hierarchical description is unavailable (e.g., masked AES hardware). Therefore, formula evaluation methods whose computation time does not heavily depend on the degree and number of variables is quite necessary.

[*3] In this dissertation, we consider only reduced GBs, which are defined as GBs containing no redundant polynomial. A GB can be easily transformed into the corresponding reduced GB.

3.2.3 Example

In this subsection, we show formal design of two-input parallel full-tree multipliers over PB-based $GF(2^m)$ [63]. We first describe algebraic description of the multipliers. Let a and b be elements of $GF(2^m)$ and the inputs to the multipliers. Let c be an element of $GF(2^m)$ and the output. Let $F(\beta)$ be an mth degree irreducible polynomial over $GF(2)$, and $F(\beta)$ is given by $\beta^m + f_{m-1}\beta^{m-1} + f_{m-2}\beta^{m-2} + \cdots + f_0$, where f_i is an element of $GF(2)$. The function of multiplier is represented by $c = a \times b$. Full-tree multipliers are composed of two blocks: a Partial Product Generator (PPG) and an ACCumulator (ACC), The functions of PPG and ACC are represented by

$$\sum_{i=0}^{n-1} w_i = a \times b, \tag{3.14}$$

$$c = \sum_{i=0}^{n-1} w_i, \tag{3.15}$$

respectively. Here, w_i is the ith partial product generated by PPG, and is given by $a \times b_i\beta^i$, where b_i ($\in GF(2)$) denotes the ith bit of b. In other words, the function of multipliers is realized by arithmetic circuits with functions represented by Eqs. 3.14–3.15.

Let $w_{i,\hat{i}}$ ($\in GF(2)$) be the ith bit of w_i. Since b is represented by $b_{m-1}\beta^{m-1} + b_{m-2}\beta^{m-2} + \ldots b_0$ in the lower-level of abstraction (and a is also represented in the same manner), the function of PPG is represented by

$$w'_{i,\hat{i}} = a_{\hat{i}} \times b_i, \tag{3.16}$$

$$w_{i,\hat{i}} = w'_{i,\hat{i}+i} + \sum_{\iota=1}^{i} \left(w'_{i,m-\iota} \times f'_{\hat{i},m+i-\iota} \right), \tag{3.17}$$

where $w'_{i,\hat{i}}$ denotes the intermediate result ($w'_{i,\hat{i}+i} = 0$ when $\hat{i}+i \geq m$), and $f'_{\hat{i},m+i-\iota}$ is a constant over $GF(2)$ which is determined as the $\hat{i}$th coefficient of a polynomial given by dividing $\beta^{m+i-\iota}$ by $F(\beta)$. Equation 3.16 represents multiplication excluding reduction by $F(\beta)$, and is realized by $GF(2)$ multipliers (i.e., AND gates). Equation 3.17 represents reduction by $F(\beta)$, and is realized by $GF(2)$ adders (i.e., XOR gates). In other words, in order to perform $w_i = a \times b_i\beta^i$, we perform the multiplication $a \times b_i$ by Eq. 3.16, and then perform the multiplication of β and $a \times b_i$ by Eq. 3.17.

On the other hand, Eq. 3.15 is implemented using an array of $GF(2^m)$ adders, which are given by bit-parallel XOR gates.

Figure 3.3 shows a GF-ACG for $GF(2^2)$ full-tree multiplier at four-levels of abstraction. The nodes in Fig. 3.3(a), (b), and (c) correspond to the respective shaded parts in Fig. 3.3(b), (c),

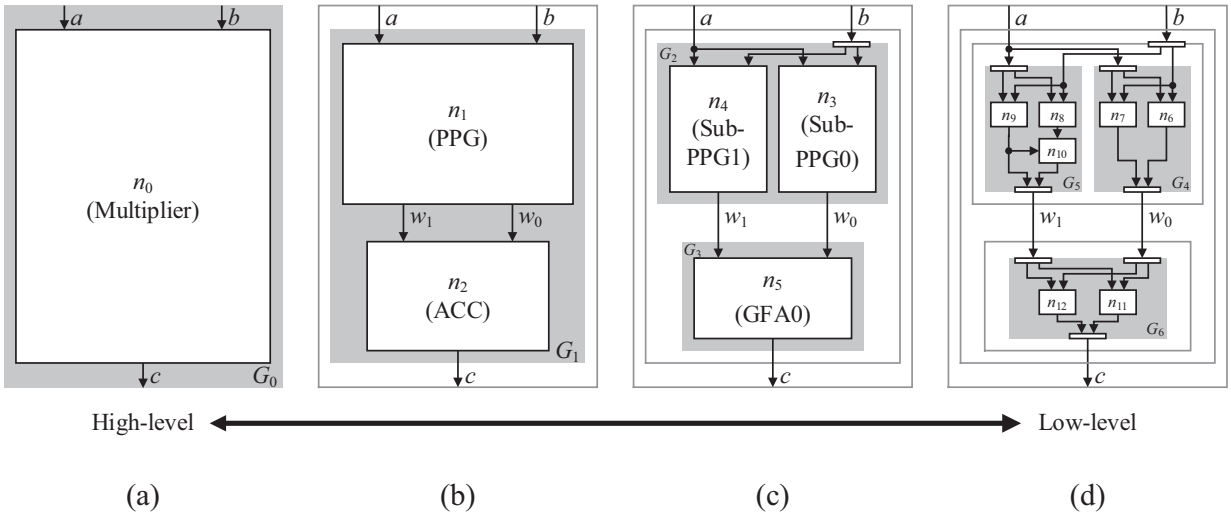


Fig. 3.3 GF-ACG for full-tree multiplier over PB-based $GF(2^2)$: (a) highest- to (d) lowest-level of abstraction.

and (d). Figure 3.3(d) denotes the lowest-level of abstraction, which represents a logic circuit. Table 3.1 shows nodes, GFs, and GF variables in Fig. 3.3, in which the decomposition and composition nodes are omitted.

Nodes "PPG" and "ACC" have functional assertions given by Eqs. 3.14 and 3.15, respectively. Nodes "SubPPG0" and "SubPPG1" have functional assertion given by Eq. 3.16. In addition, a node "GFA0" represents an adder over $GF(2^2)$, Note that we can design the multipliers for different bit length and irreducible polynomials in the same manner.

We then measure verification time of the $GF(2^m)$ multipliers for different m. Figure 3.4 shows the time of the formal verification, where the verification procedures are performed using an open-source computer algebra software Risa/Asir [2] on a Linux personal computer with an Intel Xeon E5450 3.00GHz processor and 32GB RAM. For a comparison, we also show the verification time of the corresponding Verilog HDL description using a typical Logic simulator Verilog-XL. We did not verify multipliers larger than $GF(2^{16})$ using the logic simulator in this experiment because verification time increases exponentially to the extension degree (i.e., input bit length). On the other hand, we could verify large multipliers by using formal verification. For example, a $GF(2^{128})$ multiplier was completely verified within one minute. Thus, we could confirm the efficiency of GF-ACG.

The verification time of formal verification increases proportionally to the square of extension degree because the GF-ACG for $GF(2^m)$ full-tree multipliers have one, two, and $2m^2 + m - 1$ nodes at highest-, second-, and third-level hierarchies, respectively. Algorithm 3.1 applies the

Table 3.1 Nodes, GFs, and GF variables in Fig. 3.3

Node
[Multiplier]
$n_0 = (\{c = a \times b\}, G_1)$
[PPG]
$n_1 = (\{\sum_{i=0}^{1} w_i = a \times b\}, G_2)$,
[SubPPG0]
$n_3 = (\{w_0 = a \times b_0\}, G_4)$
[GF(2) Multiplier]
$n_6 = (\{w_{0,0} = a_0 \times b_0\}, nil)$,
$n_7 = (\{w_{0,1} = a_1 \times b_0\}, nil)$,
[SubPPG1]
$n_4 = (\{w_1 = a \times b_1\}, G_5)$
[GF(2) Multiplier]
$n_8 = (\{w'_{1,0} = a_0 \times b_1\}, nil)$,
$n_9 = (\{w_{1,1} = a_1 \times b_1\}, nil)$,
[GF(2) Adder]
$n_{10} = (\{w_{1,0} = w_{1,1} + w'_{1,0}\}, nil)$,
[Accumulator]
$n_2 = (\{c = \sum_{i=0}^{2} w_i\}, G_3)$
[GFA0]
$n_5 = (\{w_0 = w_0 + w_1\}, G_6)$
[GF(2) Adder]
$n_{11} = (\{c_0 = w_{0,1} + w_{0,0}\}, nil)$
$n_{12} = (\{c_1 = w_{1,1} + g_{1,0}\}, nil)$
GFs
$GF_{(2^2)} = (\left(x^2, x^1, x^0\right), (\{0,1\}, \{0,1\}, \{0,1\}), (x^2 + x + 1, x + 1))$
$GF_{(2)} = ((1), (\{0,1\}), nil)$
GF variables
$a, b, c = (GF_{(2^2)}, (1,0))$
$w_0, w_1 = (GF_{(2^2)}, (1,0))$
$a_0, a_1, b_0, b_1, c_0, c_1 = (GF_{(2)}, (0,0))$
$w_{0,0}, w_{0,1}, w_{1,0}, w_{1,1}, w'_{1,0} = (GF_{(2)}, (0,0))$

formula evaluation to $2m^2 + m + 1$ nodes. Since the degree and number of variables of functional assertion of each node is not so large, formula evaluation of each node is performed in less than one second. Thus, the hierarchical description of GF-ACG is quite important for fast verification.

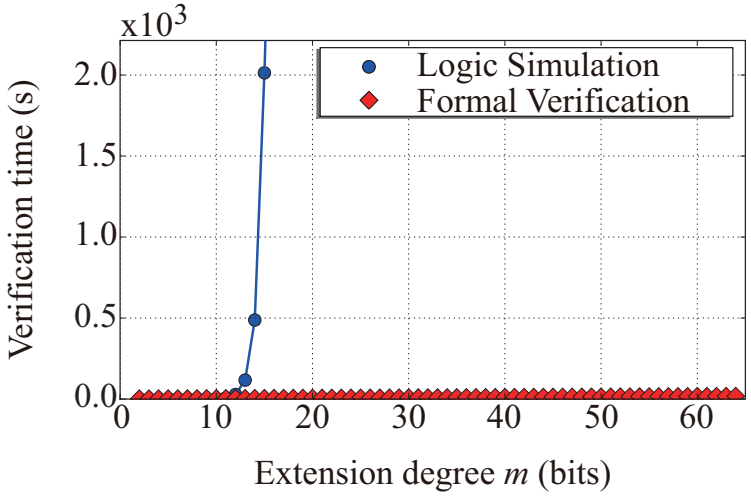


Fig. 3.4 Verification time of $GF(2^m)$ full-tree multipliers for $2 \leq m \leq 64$.

3.3 Proposed Method

3.3.1 Proposed Formal Description

This subsection presents a new GF-ACG which can handle both redundant and non-redundant GF representations and pipelined circuits.

Formal representation of GFs

The conventional GF-ACG gives $GF_{(q^m)}$ by a basis $\boldsymbol{B}$, coefficient vector $\boldsymbol{C}$, and an irreducible polynomial IP. On the other hand, in the proposed GF-ACG $GF_{(q^m)}$ is represented using a vector $\boldsymbol{X}$ which consists of arbitrary elements of $GF(q^m)$, and $\boldsymbol{X}$ is given by

$$\boldsymbol{X} = (X_{n-1}, X_{n-2}, \ldots, X_j, \ldots, X_0). \tag{3.18}$$

If $GF_{(q^m)}$ represents a PRR-based $GF(q^m)$ X^j is given by x^j. If $GF_{(q^m)}$ represents an RRB-based $GF(q^m)$, X^j is given by β^j. In addition, the number of elements in $\boldsymbol{C}$ increases to n. On the other hand, redundant representations (i.e., PB and RRB) can be also represented by the same manner as the conventional one.

The conventional $GF_{(q^m)}$ uses an irreducible polynomial to represent the indeterminate element. On the other hand, in PRR-based GFs, the modular polynomial is given by a product of an mth degree irreducible polynomial $F(x)$ and $(n-m)$th degree polynomial $G(x)$, the indeterminate x is no longer a root of the modular polynomial, and all elements are divisible by $G(x)$. To represent such properties of PRR-based GFs, we explicitly represent the modular polynomial of PRR-based GFs by two polynomials $F(x)$ and $G(x)$. By contrast, since RRB-based GF uses

a root of mth degree irreducible polynomial to represent its elements as well as PB and NB, we give the definition polynomial of RRB-based GFs for by only $F(\beta)$ as same as PB and NB. Thus, in the proposed GF-ACG, $GF_{(q^m)}$ is defined as

$$(\boldsymbol{X}, \boldsymbol{C}, (F, G)). \tag{3.19}$$

If $GF_{(q^m)}$ represents a PRR-based $GF(q^m)$, let G be a polynomial defining the GF. If $GF_{(q^m)}$ represents a PB-, an NB-, or an RRB-based $GF(q^m)$, let G be 1. For example, let $GF^{(PB)}_{(2^2)}$, $GF^{(NB)}_{(2^2)}$, $GF^{(PRR)}_{(2^2)}$, and $GF^{(RRB)}_{(2^2)}$ be formal representations of a PB-, an NB-, a PRR-, and an RRB-based $GF(2^2)$ with $F(\beta) = \beta^2 + \beta + 1$ ($F(x) = x^2 + x + 1$), respectively. They are represented by

$$GF^{(PB)}_{(2^2)} = ((\beta^1, \beta^0), (\{0,1\}, \{0,1\}), (\beta^2 + \beta^1 + \beta^0, 1)), \tag{3.20}$$

$$GF^{(NB)}_{(2^2)} = ((\alpha^{2^1}, \alpha^{2^0}), (\{0,1\}, \{0,1\}), (\beta^2 + \beta^1 + \beta^0, 1)), \tag{3.21}$$

$$GF^{(PRR)}_{(2^2)} = ((x^2, x^1, x^0), (\{0,1\}, \{0,1\}, \{0,1\}), (x^2 + x + 1, x + 1)), \tag{3.22}$$

$$GF^{(RRB)}_{(2^2)} = ((\beta^2_0, \beta^1_0, \beta^0_0), (\{0,1\}, \{0,1\}, \{0,1\}), (\beta^2 + \beta^1 + \beta^0, 1)), \tag{3.23}$$

where $\alpha = \beta^1$.

PRR-based $GF(q^m)$ is equivalent to an m-dimensional subspace of n-dimensional linear space over $GF(q)$, which indicates that arithmetic circuits over PRR-based GF have don't-care inputs in contrast to those based on other representations. In words, $q^n - q^m$ elements in the entire space q^n are not included in PRR-based $GF(q^m)$. Because these $q^n - q^m$ elements are not applied to the GF arithmetic, the above algebraic verification cannot be directly applied to the GF arithmetic circuits described in digit-level, that is, nodes whose internal structures include decomposition (or composition) nodes of $GF(q^m)$ (or $GF(q)$) elements into $GF(q)$ (or $GF(q^m)$) elements. Note that we can verify any node whose internal structure consists of word-level arithmetic functions.

The basic idea for valid verification is to represent the don't-care condition by algebraic equations. Because PRR-based $GF(q^m)$ is equivalent to an m-dimensional subspace of n-dimensional linear space, we can consider PRR-based $GF(q^m)$ as an (n, m) linear code over $GF(q)$, where n and m denote code length and information symbol length, respectively. In addition, since all elements of PRR-based $GF(q^m)$ can divisible by $G(x)$, PRR-based $GF(q^m)$ defined with $G(x)$ (and $H(x)$) is equivalent to a (quasi-)cyclic code with a generator polynomial $G(x)$. The (n, m) cyclic code is a linear code where all (i.e., polynomials whose degree is up to $n-1$) are close to addition, multiplication, and cyclic shift (i.e., arithmetic operations modulo $P(x) = x^n - 1$). On the other hand, an (n, m) quasi-cyclic code is derived by shortening an (n', m) cyclic code ($n' > n$) where all codewords are closed to addition and multiplication modulo $P(x)$ ($= G(x) \times F(x)$). Thus, PRR-based $GF(q^m)$ is equivalent to the (n, m) cyclic code if the modular polynomial $P(x)$ is

binomial; otherwise, PRR-based $GF(q^m)$ is equivalent to an (n, m) quasi-cyclic code. The necessary and sufficient condition that a polynomial is a codeword of a (quasi-)cyclic code is given by linear equations called Linear Recurrence Relation (LRR). We can represent the don't-care condition of PRR-based GF arithmetic circuits using LRR.

The LRR of cyclic code is derived from the generator polynomial $G(x)$. (The LRR of quasi-cyclic code is also derived from a similar method.) We first calculate an mth degree polynomial $H(x) = x^m + h_{m-1}x^{m-1} + h_{m-2}x^{m-2} + \cdots + h_0$ ($h_i \in GF(q)$) called parity check polynomial as follows:

$$H(x) = \frac{x^n - 1}{G(x)}. \tag{3.24}$$

$(x^n - 1)/G(x)$ in the rhs is always divisible by $G(x)$ since $G(x)$ is given as a factor polynomial of $x^n - 1$. Here, a polynomial of up to nth degree $A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n_2} + \cdots + a_0$ ($a_j \in GF(q)$) is a code word of the (n, m) cyclic code if and only if $a_0, a_1, \ldots, a_{n-1}$ satisfy

$$h_0 a_{\hat{j}+m} + h_1 a_{\hat{j}+m-1} + \cdots + h_{m-1} a_{\hat{j}+1} + a_{\hat{j}} = 0, \tag{3.25}$$

for $0 \leq \hat{j} \leq n - 1$, and Eq. 3.25 is the LRR of the (n, m) cyclic code. While we described the LRR of cyclic code, we can also derive those of quasi-cyclic codes by replacing n with n' and letting $a_n = a_{n+1} = \cdots = a_{n'-1} = 0$ in Eq. 3.25.

For valid (i.e., sound and complete) verification, in the proposed GF-ACG, decomposition nodes of PRR-based $GF(q^m)$ into $GF(q)$ has LLR as its functional assertion in order to guarantee that the decomposed variable is a valid element of PRR-based $GF(q^m)$ and satisfies don't-care condition. On the other hand, when verifying nodes whose internal structure contains composition nodes of PRR-based $GF(q)$ into $GF(q^m)$, we check whether the LRR of composed variable can be derived from the internal structure in order to guarantee the outputs' validity.

In addition, PRR-based GFs use a formal variable x instead of a root of irreducible polynomial (e.g., β). This indicates that we should describe the correspondence relation between x and β explicitly in verifying GF arithmetic circuits based on a combination of PRR and other representations. We then describe the correspondence relation between x and β, where β is a root of $F(x)$. Let $E(x)$ be the multiplicative unit element of PRR-based $GF(q^m)$, and $E(x)$ corresponds to 1 in $GF(q^m)$ based on other representations. According to [60], an element of PRR-based GF $A(x)$ is mapped to the corresponding element of PB-based GF by substituting β into x in $A(x)$. Therefore, $E(\beta) = 1$. Then, let us consider a polynomial $xE(x) \bmod P(x)$. Since $E(x)$ is divisible by $G(x)$, $xE(x)$ is also divisible by $G(x)$, which indicates $xE(x)$ is an element of PRR-based $GF(q^m)$. Here, since $\beta E(\beta) = \beta$ (which is followed by $E(\beta) = 1$), $xE(x) \bmod P(x)$ corresponds to β. Hence, the correspondence relation between x and β is given by

$$(xE(x) \bmod P(x)) = \beta. \tag{3.26}$$

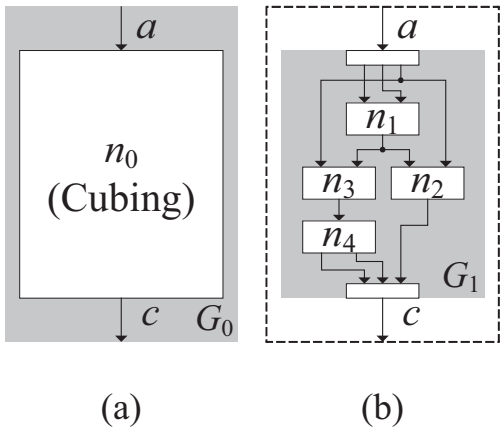


Fig. 3.5 GF-ACG for PRR-based $GF(2^2)$ cubing circuit.

Table 3.2 Nodes, GFs, and GF variables in Fig. 3.5

Node
[Cubing] $n_0 = (\{c = a^3, c_2 + c_1 + c_0 = 0\}, G_1)$
[Decomposition] $n_{decomp} = (\{a_2x^2 + a_1x + a_0 = a, a_2 + a_1 + a_0 = 0\}, nil)$
[AND] $n_1 = (\{w_0 = a_2 \times a_1\}, nil)$
[AND] $n_2 = (\{c_0 = w_0 \times a_0\}, nil)$
[XOR] $n_3 = (\{c_1 = w_0 + a_0\}, nil)$
[wiring] $n_4 = (\{b_2 = c_1, b_1 = c_1\}, nil)$
[Composition] $n_{comp} = (\{c = c_2x^2 + c_1x + c_0\}, nil)$
GFs
$GF_{(2^2)}^{(PRR)} = (\left(x^2, x^1, x^0\right), (\{0,1\}, \{0,1\}, \{0,1\}), (x^2 + x + 1, x + 1))$
$GF_{(2)} = ((1), (\{0,1\}), nil)$
GF variables
$a, b = (GF_{(2^2)}^{(PRR)}, (2, 0))$
$a_2, a_1, a_0, b_2, b_1, b_0, c_1, c_0 = (GF_{(2)}, (0, 0))$

By adding Eq. 3.26 to functional assertion of decomposition/composition nodes for PRR-based GFs, we can validly verify GF arithmetic circuits based on a combination of GF representations. Thus, we can represent and verify GF arithmetic circuits based on various GF representation. Note that $E(x)$ can be easily derived using the extended Euclidean algorithm. See [60] for detail.

In the following, we show two examples to validate our method.

The first example of extended GF-ACG is a PRR-based $GF(2^2)$ cubing circuit based on combined representations. Figure 3.5 depicts a GF-ACG for the cubing circuit at two levels of abstraction, and Tab. 3.2 shows the nodes, variables, and GFs in Fig. 3.5. In this example, we have $G(x) = x + 1$, $F(x) = x^2 + x + 1$ (the second degree AOP) and $P(x) = x^3 + 1$. The functional

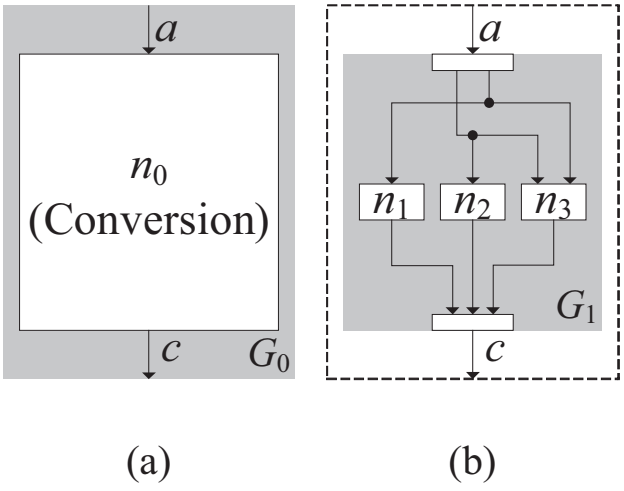


Fig. 3.6 GF-ACG for circuit converting from PB-based $GF(2^2)$ to PRR-based one.

assertion is given by $c = a^3$. Here, c is represented by $c = c_2x^2 + c_1x + c_0$, and the LRR of c of PRR-based $GF(2^2)$ is given by $c_2 + c_1 + c_0 = 0$. (a and its LRR are also given in the same manner.) We can efficiently implement such non-linear function using bit-level operations exploiting the don't-care condition (i.e., LRR). The node "Cubing" has the functional assertion $c = a^3$ and the LRR of c (i.e., $c_2 + c_1 + c_0 = 0$) because its internal structure contains a composition node for c. Its internal structure represents a logic circuit consisting of a decomposition node, a composition node, two AND gate nodes, and an XOR gate node. Note here that the functional assertion decomposition node includes an additional GF equation representing the LRR for the input variable a (i.e., $a_2 + a_1 + a_0 = 0$).

When we verify the node "Cubing" without the LRR, the formula evaluation returns a value of false because the cubing circuit is valid only for elements of the PRR-based $GF(2^2)$. On the other hand, the formula evaluation considering the LRR returns a value of true because the LRR provides the condition that a is always an element of PR-based $GF(2^2)$. Furthermore, the formula evaluation considering the LRR also guarantees that the output variable c is always an element of $GF(2^2)$ because the corresponding LRR ($c_2 + c_1 + c_0 = 0$) is derived from the GB.

The second example is a conversion circuit from the PB-based $GF(2^2)$ to PRR-based one. Figure 3.6 shows the corresponding GF-ACG and Tab. 3.3 shows the nodes, GFs, and GF variables in Fig. 3.6. Node "Conversion" in Fig. 3.6(a) is the top-level node that consists of a composition node n_{comp}, decomposition node n_{decomp}, nodes for wiring n_1, n_2, and a node representing an XOR gate n_3. Figure 3.6(b) shows the internal structure of "Conversion." Nodes in Fig. 3.6(b) does not have internal structure because the GF-ACG of Fig. 3.6(b) represents a logic circuit. In this example, the unit element $E(x)$ is given by $x^2 + x$, and the algebraic relation between the

Table 3.3 Nodes, GFs, and GF variables in Fig. 3.5

Node
[Conversion] $n_0 = (\{c = a, c_2 + c_1 + c_0 = 0\}, G_1)$
[Decomposition] $n_{decomp} = (\{a_1\beta + a_0 = a\}, nil)$
[wiring] $n_1 = (\{c_0 = a_1\}, nil)$
[wiring] $n_2 = (\{c_1 = a_0\}, nil)$
[XOR] $n_3 = (\{c_2 = a_1 + a_0\}, nil)$
[Composition] $n_{comp} = (\{c = c_2x^2 + c_1x + c_0, x^2 + x = \beta\}, nil)$
GFs
$GF^{(PB)}_{(2^2)} = (\left(\beta^1, \beta^0\right), (\{0,1\}, \{0,1\}), (\beta^2 + \beta + 1, 1))$
$GF^{(PRR)}_{(2^2)} = (\left(x^2, x^1, x^0\right), (\{0,1\}, \{0,1\}, \{0,1\}), (x^2 + x + 1, x + 1))$
$GF_{(2)} = ((1), (\{0,1\}), nil)$
GF variables
$a = (GF^{(PB)}_{(2^2)}, (2, 0))$
$b = (GF^{(PRR)}_{(2^2)}, (2, 0))$
$a_1, a_0, b_2, b_1, b_0 = (GF_{(2)}, (0, 0))$

indeterminate x and the root β (satisfying $\beta^2 + \beta + 1 = 0$) is represented by $x(x^2 + x) = \beta$. The functional assertion of "Conversion" is given by $b = a$, which cannot be derived from only the five equations representing its internal structure. On the other hand, using the relation $x^2 + 1 = \beta$, we can correctly derive $b = a$ from its internal structure equations. The LRR of c (i.e., $c_2 + c_1 + c_0 = 0$) is also derived, which guarantees that c is an element of PRR-based $GF(2^2)$.

Formal representation of sequential arithmetic circuits

We then describe new GF-ACG which can represent sequential arithmetic circuits that perform the computation with plural clock cycles, such as pipelined ones.

In order to represent time modality, we introduce Linear-time Temporal Logic (LTL) to the GF-ACG representation. LTL is a kind of modal logic and has modality for linear and discrete time [84,118]. So far, LTL has been used for several formal verification such as behavior verification of microprocessors [38, 73]. In the proposed method, we represent the circuit functions using LTL, and then transform them to GF equations representing the same functions in order to perform formula evaluation by the conventional computer algebra software (or formula evaluation method presented in Section 3.3).

First, we introduce two modal operators: $\mathcal{X}$ and $\mathcal{N}$. The operator $\mathcal{X}$ is applied to an equation, and $\mathcal{X}(\Pi)$ (Π is an equation such as $c = a \times b$) indicates that Π holds for every clock cycle in a synchronous circuit. Note here that, for equations Π_0 and Π_1, $(\mathcal{X}(\Pi_0) \wedge \mathcal{X}(\Pi_1)) \rightarrow \mathcal{X}(\Pi_0 \wedge \Pi_1)$, which expresses that the identical clock is applied to two blocks with respective functions Π_0 and

Π_1 in a circuit. The operator $\mathcal{N}$ is applied to a variable. $\mathcal{N}v$ denotes the value of a variable v after one clock cycle for the reference (e.g., initial) cycle. Here, v and $\mathcal{N}v$ basically represent the identical variable, but have different modal (i.e., clock cycles). In addition, $\mathcal{N}$ can be applied repeatedly to one variable. For example, $\mathcal{N}(\mathcal{N}v)$ denotes the value of a variable v after two clock cycles. To represent delays of ϑ clock cycles (where ϑ is a natural number), we define $\mathcal{N}^{\vartheta}$ as follows:

$$\mathcal{N}^{\vartheta}v = \begin{cases} \mathcal{N}(\mathcal{N}^{\vartheta-1}v) & \text{if} \quad \vartheta \geq 2 \\ \mathcal{N}v & \text{if} \quad \vartheta = 1 \end{cases} . \tag{3.27}$$

Thus, $\mathcal{N}^{\vartheta}v$ denotes the value of v after ϑ clock cycles. Note that, in this representation, v without $\mathcal{N}$ is considered as $\mathcal{N}^0 v$ and represents the value of v at the reference clock cycle.

We then describe pipelined circuits using the above modal operators. We first define the pipeline register. A pipeline register has the functionality which outputs the input of the previous clock cycle. In addition, the functionality holds for every clock cycle. Hence, the functional assertion of pipeline register is represented by

$$\mathcal{X}\left(\mathcal{N}v_{out} = v_{in}\right), \tag{3.28}$$

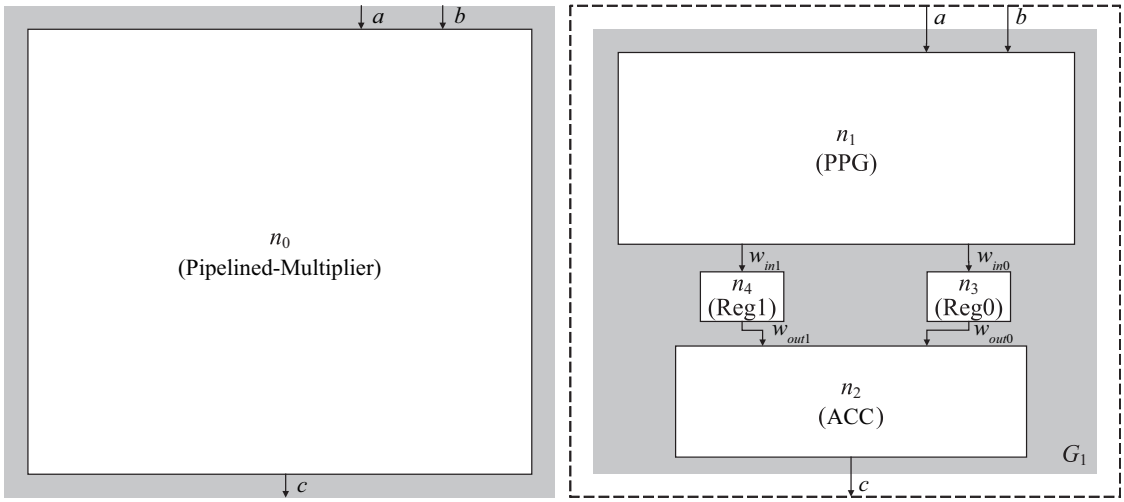
where v_{in} and v_{out} denote the input and output of pipeline register, respectively. Let n_{reg} be a node for pipeline register. Since standard cell libraries usually have cells for register and delay element, n_{reg} has no internal structure and is given as follows:

$$n_{reg} = (\{\mathcal{X}\left(\mathcal{N}v_{out} = v_{in}\right)\}, nil). \tag{3.29}$$

Similarly, we can also represent other functional delay elements and registers (e.g., scan flip-flops) using $\mathcal{X}$ and $\mathcal{N}$.

We can represent pipelined circuit using n_{reg}. For example, Fig. 3.7 shows a pipeline $GF(2^2)$ multiplier at (a) top- and (b) second-level of abstraction. Table 3.4 shows nodes, GF, and GF variables in Fig. 3.7. The basic structure of the multiplier is the same as one in Fig. 3.3 excluding pipeline registers inserted at boundary between PPG and ACC. The functional assertion of the highest-level node is given by $\mathcal{X}(\mathcal{N}c = a \times b)$, which expresses that the product of a and b is computed one clock cycle after the input, and this function holds for every clock cycle. Note that we omit the internal structures of PPG and ACC because they are given in the similar manner to ones in Fig. 3.3.

Thus, we can describe any kind of GF arithmetic circuits based on pipeline. However, since the above representation includes the modal operators of LTL, the following formal verification cannot be applied to the representation. Therefore, we transform the LTL-based representation



(a) (b)

Fig. 3.7 GF-ACG for pipelined $GF(2^2)$ multiplier: (a) top- and (b) second-level of abstraction.

Table 3.4 Nodes, GF, and GF variables in Fig. 3.7

Nodes
[Pipelined-Multiplier] $n_0 = (\{\mathcal{X}(\mathcal{N}c = a \times b)\}, G_1)$
[PPG] $n_1 = (\{\mathcal{X}(w_{in0} + w_{in1} = x \times y)\}, G_2)$
[GFA] $n_2 = (\{\mathcal{X}(c = w_{out0} + w_{out1})\}, G_3)$
[Reg0] $n_3 = (\{\mathcal{X}(\mathcal{N}w_{out0} = w_{in0})\}, nil)$
[Reg0] $n_4 = (\{\mathcal{X}(\mathcal{N}w_{out10} = w_{in1})\}, nil)$

GFs
$GF(2^2) = (\left(\beta^1, \beta^0\right), (\{0,1\}, \{0,1\}), \beta^2 + \beta + 1)$
$GF(2) = (\left(\beta^0\right), (\{0,1\}), nil)$

GF variables
$x, y, z = (GF(2^2), (1,0))$
$w_i, r_i = (GF(2^2), (1,0)),\ (0 \leq i \leq 1)$

to GF equations which represent the identical functionality without modal operators. For the transformation, we introduce time (i.e., clock cycle) indices.

Let v^t be the value of v in the tth clock cycle (t is an element of $\mathbb{N}$). The formula $\mathcal{X}(\Pi)$ is equal to $\forall t(\Pi)$ because $\mathcal{X}$ indicates "for all clock cycles." By contrast, for the t_0th clock cycle, $\mathcal{N}v^{(t_0)}$ is equal to $v^{(t_0+1)}$ because $v^{(t_0+1)}$ indicates the value after one clock cycle. It is clear that $\mathcal{N}^{\vartheta}v^{(t_0)}$ is equal to $v^{(t_0+\vartheta)}$. Thus, we can describe any LTL-based representation without temporal operators $\mathcal{X}$ and $\mathcal{N}$. The transformation can be performed in a systematic manner. Note

that the GF equations after transformation should be generated through LTL-based representation in order to express data dependency between different clock cycles.

Then, we describe the formal verification of Pipelined-Multiplier in Fig. 3.7. The functional assertion of Pipelined-Multiplier is transformed into

$$\forall t(c^{(t+1)} = a^{(t)} \times b^{(t)}), \tag{3.30}$$

and substitute a reference clock cycle t_0 into the bound variable t as follows:

$$c^{(t_0+1)} = a^{(t_0)} \times b^{(t_0)}, \tag{3.31}$$

which is the GF equation which should be derived from its internal structure. Note that, in this case, there is no constraint about the free variable t_0. On the other hand, the functional assertions in its internal structure are composed of $\forall t(w_{in0}^{(t)} + w_{in1}^{(t)} = a^{(t)} \times b^{(t)})$, $\forall t(c^{(t)} = w_{out0}^{(t)} + w_{out1}^{(t)})$, $\forall t(w_{out0}^{(t+1)} = w_{in0})$, $\forall t(w_{out1}^{(t)} = w_{in1})$. Since we consider only synchronous circuits, all blocks in the circuit should have the reference (i.e., initial) clock cycle. Since $\forall t(\Pi_0) \wedge \forall t(\Pi_1) \rightarrow \forall t(\Pi_0 \wedge \Pi_1)$, the simultaneous equations representing internal structure is represented by

$$\begin{cases} w_{in0}^{(t)} + w_{in1}^{(t)} &= a^{(t)} \times b^{(t)} \\ w_{out0}^{(t+1)} &= w_{in0}^{(t)} \\ w_{out1}^{(t+1)} &= w_{in1}^{(t)} \\ c^{(t+1)} &= w_{out0}^{(t+1)} + w_{out1}^{(t+1)} \end{cases}, \tag{3.32}$$

for all t. Therefore, we verify the function of Pipelined-Multiplier by checking whether Eq. 3.31 is satisfied if Eq. 3.33 holds for all t.

Let ϑ_{max} be the maximum value of exponents of $\mathcal{N}$ in Eq. 3.31. Here, ϑ_{max} represents the number of pipeline stages, that is, the number of clock cycles required for the computation of Pipelined-Multiplier. Since simultaneous equations derived by substituting t_0 into t in Eq. 3.33 represent the circuit function in t_0th clock cycle, simultaneous equations derived by substituting $t_0, t_0 + 1, \ldots, t_0 + \vartheta_{max}$ into t in Eq. 3.33 represent all functions required for the computation. In the case of Eq. 3.31, because $\vartheta_{max} = 1$, we can check whether Eq. 3.31 can be derived from

$$\begin{cases} w_{in0}^{(t_0)} + w_{in1}^{(t_0)} &= a^{(t_0)} \times b^{(t_0)} \\ w_{out0}^{(t_0+1)} &= w_{in0}^{(t_0)} \\ w_{out1}^{(t_0+1)} &= w_{in1}^{(t_0)} \\ c^{(t_0+1)} &= w_{out0}^{(t_0+1)} + w_{out1}^{(t_0+1)} \\ w_{in0}^{(t_0+1)} + w_{in1}^{(t_0+1)} &= a^{(t_0+1)} \times b^{(t_0+1)} \\ w_{out0}^{(t_0+2)} &= w_{in0}^{(t_0+1)} \\ w_{out1}^{(t_0+2)} &= w_{in1}^{(t_0+1)} \\ c^{(t_0+2)} &= w_{out0}^{(t_0+2)} + w_{out1}^{(t_0+2)} \end{cases}, \tag{3.33}$$

in order to verify the function of Pipelined-Multiplier. Because there is no longer temporal operators nor quantifier (i.e., $\forall$), we can perform formula evaluation based on polynomial reduction by GB (or other formula evaluation methods represented in the following section). While we described only the case where $\vartheta_{max} = 1$, we can also verify any kind of GF arithmetic circuit with ϑ_{max} pipelined stage by substituting $t_0, t_0 + 1, \ldots, t_0 + \vartheta_{max}$ into t. On the other hand, the number of variables in the formula evaluation increases linearly to the number of pipeline stages (i.e., ϑ_{max}), and ϑ_{max} would have an impact on verification time. Therefore, we confirm the effectiveness of proposed method through experimental verification of DPA-resistant cryptographic hardware utilizing pipeline.

In the above verification, the circuit function and input/output variables are fixed and initialization is not required, which indicates that no constraint on the free variable t_0. Therefore, the functional assertion of pipelined circuit (e.g., Eq. 3.31) holds for all t_0 in accordance with the $\forall$ introduction rule. On the other hand, if the circuit requires initialization or input/output variables change in clock cycles, there is constraints on t_0. To verify such circuits (e.g., multiprecision multipliers), methods for representing operand-scheduling and initial condition would be required.

3.3.2 Proposed Formal Verification

This subsection presents new formal verification method of GF-ACG. The verification of GF-ACG is performed by checking equivalence between hierarchies. So far, a formula evaluation method based on polynomial reduction by GB (i.e., Algorithms 3.2 and 3.3) have been used for the equivalence checking. However, the method has difficulties in verifying circuits with higher-degree function (e.g., AES decryption hardware) and those with logic-level optimization (e.g., masked AES hardware). To address these issues, we first present two formula evaluation methods. First, we present a method based ND for the first order predicate logic, which can rapidly verify circuits of higher-degree functions. We then present a method combining PPRM expansion and GB-based evaluation, which can be useful for verifying logic-level optimized circuits. Finally, we propose new verification algorithm which combines three formula evaluation method for fast and complete verification.

ND-based evaluation method

A major advantage of GF-ACG is that we can describe sophisticated datapaths, such as those found in many cryptographic hardware. However, they sometimes include high-level nodes with higher-degree arithmetic functions (e.g., exponentiation of polynomials). GB-based verification has difficulty handling such functions, as mentioned above. On the other hand, the functional assertion of such nodes is often given as a simple equation, substituting input variables for internal

$$\frac{}{\tau = \tau} \qquad \frac{P[\tau_0/var] \quad \tau_0 = \tau_1}{P[\tau_1/var]}$$

(a) (b)

Fig. 3.8 Proof figures for inference rules of proposed method: (a) axiom of equal sign and (b) rule of equal sign.

structure (function). This is because many higher-level functions of cryptography, which consist of several sub-functions, are implemented by simple serial and/or parallel combinations of sub-functions. The correctness of such substitution can be verified deductively without algebraic manipulation. In other words, we use only substitution to verify whether the connection of the internal nodes is valid for the function.

To verify nodes whose functional assertions are efficiently given by substitutions alone, we introduce a natural deduction technique for first-order predicate logic with equal sign [121]. Accordingly, we treat the equations (i.e., functional assertions and simultaneous equations of internal structure) as symbol strings in order to apply the inference rules of natural deduction, which derives an equation from other equations. In predicate logic, if formula Π can be derived from a set of formulae $\boldsymbol{T}$ by applying inference rules a finite number of times, it can be said "Π is provable from $\boldsymbol{T}$." This means that A is correct if all of the elements of $\boldsymbol{T}$ are correct. Using the proof method, we solve simultaneous algebraic equations of mathematical substitutions by natural deduction. In general, we generate a proof figure to give the natural deduction proof. The proof figure has a tree structure consisting of formulae, and shows that the formula at the bottom (i.e., the conclusion) is provable from the formulae at the top (i.e., the premises). We apply inference rules repeatedly to the formulae from the bottom up in order to generate such a proof figure. The inference rules themselves are given as a proof figure. Figure 3.8 shows proof figures for the two inference rules of natural deduction used in the proposed method. The set of rules are equivalent to equality axioms that give the attributes of the general equal sign. Figure 3.8(a) shows one inference rule, that $\tau = \tau$ for any term τ is true without any premise or/and any proof. Figure 3.8(b) shows the other inference rule, which defines mathematical substitution, where $P[\tau_0/var]$ and $P[\tau_1/var]$ indicate the formulae P where every variable var is replaced by term τ_0 and term τ_1, respectively. The rule illustrated in Fig. 2(b) says that an equation $P[\tau_1/var]$ is provable from two equations, $P[\tau_0/var]$ and $\tau_0 = \tau_1$. The natural deduction system with these two rules guarantees that the conclusion at the bottom of the proof figure is true if all of the premises are true.

The new ND-based method treats a functional assertion (an equation) and internal structure

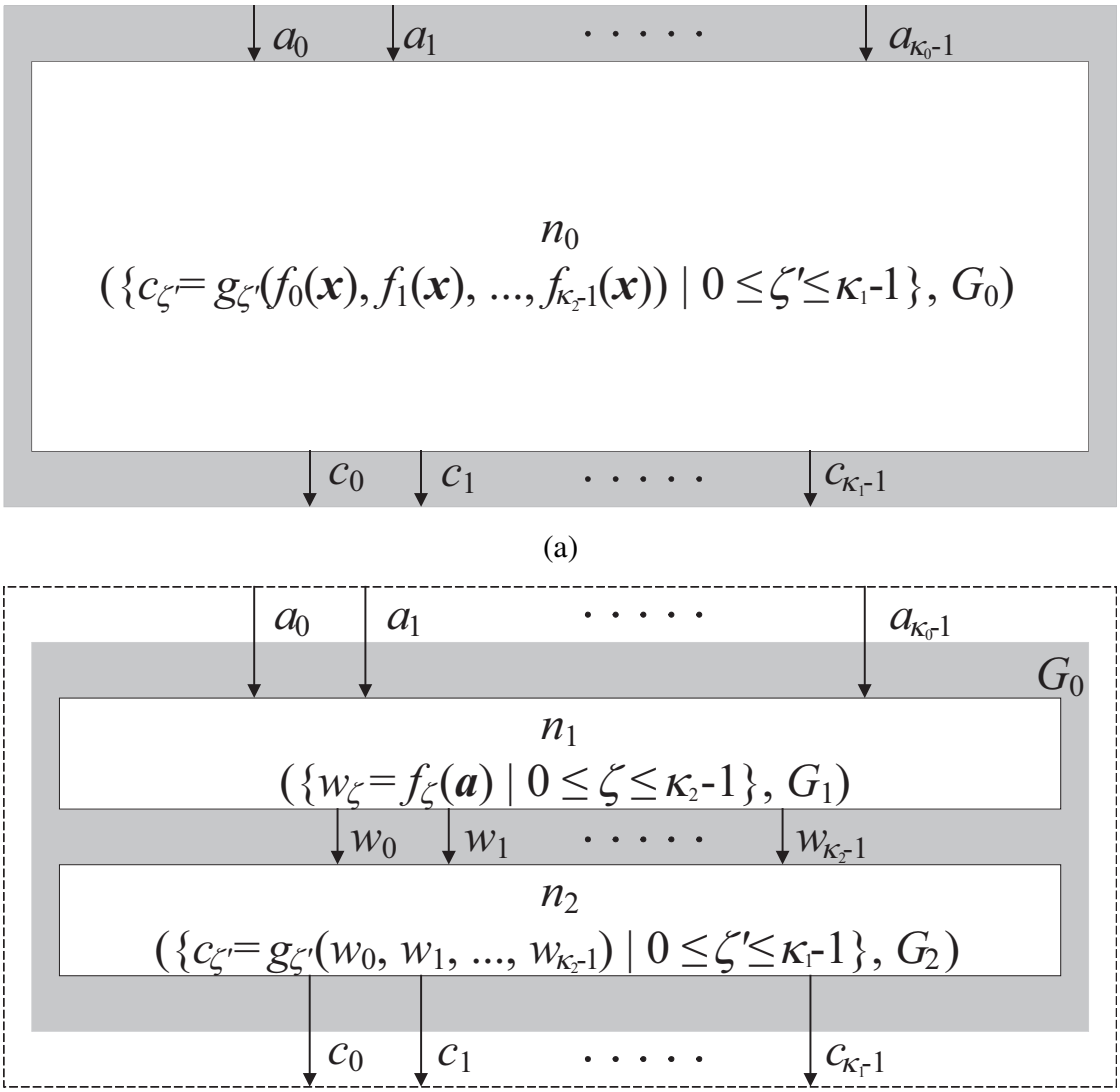


Fig. 3.9 Example of GF-ACG for ND-based method: (a) top-level and (b) second-level description.

(i.e., simultaneous equations) as a "conclusion" and "premises" in the proof figure, respectively. When we successfully generate a proof figure that has the functional assertion at the bottom and consists of inference rules and/or premises at the higher steps, we can say that the node function is correctly derived from the internal structure.

Note here that the new method exploits arithmetic logic and essentially differs from the existing verification methods based on DDs. Our new method describes predicates (i.e., GF equations) by word-level variables whose domain consists of elements of GF and gives a proof by the applica-

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{c_{\zeta'}=c_{\zeta'}} \quad c_{\zeta'}=g_{\zeta'}(w_0, w_1, \ldots, w_{\kappa_2-1})}{c_{\zeta'}=g_{\zeta'}(w_0, w_1, \ldots, w_{\kappa_2-1})} \quad w_0=f_0(\boldsymbol{a})}{c_{\zeta'}=g_{\zeta'}(f_0(\boldsymbol{a}), w_1, \ldots, w_{\kappa_2-1})} \quad w_1=f_1(\boldsymbol{a})}{c_{\zeta'}=g_{\zeta'}(f_0(\boldsymbol{a}), f_1(\boldsymbol{a}), \ldots, w_{\kappa_2-1})} \quad w_{\kappa_2-1}=f_{\kappa_2-1}(\boldsymbol{a})}{c_{\zeta'}=g_{\zeta'}(f_0(\boldsymbol{a}), f_1(\boldsymbol{a}), \ldots, f_{\kappa_2-1}(\boldsymbol{a}))}$$

Fig. 3.10 Proof figure for verification of n_0 in Fig. 3.9.

tion of predicates (i.e., inference rules), while DD-based methods describe a circuit specification by using logic variables and quantifiers $\forall$ and $\exists$ and check the equivalence between target and reference circuits. Thus, the new method is easily applied to GF-ACGs.

Figure 3.9 shows an example of GF-ACG which is suitable for the new verification method. Let n_0 in Fig. 3.9(a) be a node that has κ_0 inputs and κ_1 outputs. The functional assertion is given by

$$c_{\zeta'} = g_{\zeta'}\left(f_0(\boldsymbol{a}), f_1(\boldsymbol{a}), \ldots, f_{\kappa_2-1}(\boldsymbol{a})\right),\ 0 \le \zeta' \le \kappa_1 - 1, \tag{3.34}$$

where $\boldsymbol{x}$ represents κ_0 variables $a_0, a_1, \ldots, a_{\kappa_0-1}$, and $f_0, f_1, \ldots, f_{\kappa_2-1}$ represent functions of κ_2 and $g_0, g_1, \ldots, g_{\kappa_1-1}$ represent functions of κ_1 variables, respectively. In addition, let n_1, n_2 in Fig. 3.9(b) be nodes of the internal structure of n_0 with functional assertions

$$w_\zeta = f_\zeta\left(\boldsymbol{a}\right),\ 0 \le \zeta \le \kappa_2 - 1, \tag{3.35}$$

$$c_{\zeta'} = g_{\zeta'}(w_0, w_1, \ldots, w_{\kappa_2-1}),\ 0 \le \zeta' \le \kappa_1 - 1, \tag{3.36}$$

respectively. Figure 3.10 shows a proof figure to verify the functional assertion of n_0, which is generated by the verification method based on natural deduction. We generate such proof figures for all ζ' $(0 \le \zeta' \le q-1)$. The double line in Fig. 3.10 represents the application of the same inference rules more than once. Such a proof figure is certainly generated if the function is correct and the simultaneous algebraic equations are given as mathematical substitutions. Fig. 3.9 illustrates an example where the internal structure is composed of two nodes connected in series, but the verification method can be applied to any internal structure if the simultaneous equations are given as mathematical substitutions.

Algorithm 3.4 shows the proposed ND-based method, where the node given by $(\boldsymbol{F}, G)$ is the input and the verification result $res \in \{\text{true, false}\}$ is the output. Lines 3–5 obtain simultaneous

Algorithm 3.4 Formula evaluation based on ND

Input: Node $n = (\boldsymbol{F}, G_{in})$
Output: Formula evaluation result $res \in \{\text{true}, \text{false}\}$

```
 1: function ND-BASEDEVALUATION(n)
 2:     set S ← ∅; Bool res ← true;
 3:     for all n = (F', G'_in) ∈ N'(∈ G_in) do
 4:         S ← S ∪ F';
 5:     end for
 6:     list L ← SortEquationsByLength(S);
 7:     for all f ∈ F do
 8:         str c ← f; int ζ ← 0; list L' ← L;
 9:         while ζ < length(L') do
10:             if IsSubstitutable(S, L'[i]) = true then
11:                 c ← Substitution(c, L'[i]);
12:                 delete L'[i];  ζ ← 0;
13:             else
14:                 ζ ← ζ + 1;
15:             end if
16:         end while
17:         res ← res& IsAxiom(S);
18:     end for
19:     return res;
20: end function
```

equations (i.e., functional assertions) that forms the internal structure G as well as Algorithm 3.2. The function "SortEquationsByLength" in Line 6 sorts the obtained equations by the length of rhs in descending order to perform the following substitutions uniquely. Using the sorted list, we then apply the inference rule in Fig. 3.8(b) repeatedly to each equation in $\boldsymbol{F}$. The function "IsSubstitutable" in Line 10 examines whether the ζth element of L can be substituted for any term in c. If it is possible, the function "Substitution" at line 11 performs mathematical substitution by using the rule of equal sign, where the output c, the inputs c, and $L'[i]$ correspond to $P[\tau_0/var]$, $P[\tau_1/var]$, and $\tau_0 = \tau_1$ in Fig. 3.8(b), respectively. After the substitution, the equation used in "Substitution" is deleted from list L' since an equation is never used again to give proof of an equation and the value of ζ is reset to examine the substituted equation again from the first equation in L'. After all substitutions, the function "IsAxiom" in Line 17 examines whether the obtained equation is the same as axiom $(\tau = \tau)$. Finally, we return true if and only if all the equations in $\boldsymbol{F}$ are transformed to the axiom.

The verification time of this algorithm is basically proportional to the number of performed substitution operations, that is, the number of equations that comprise the functional assertion and the internal edges in the internal structure Note that each substitution operation is performed by

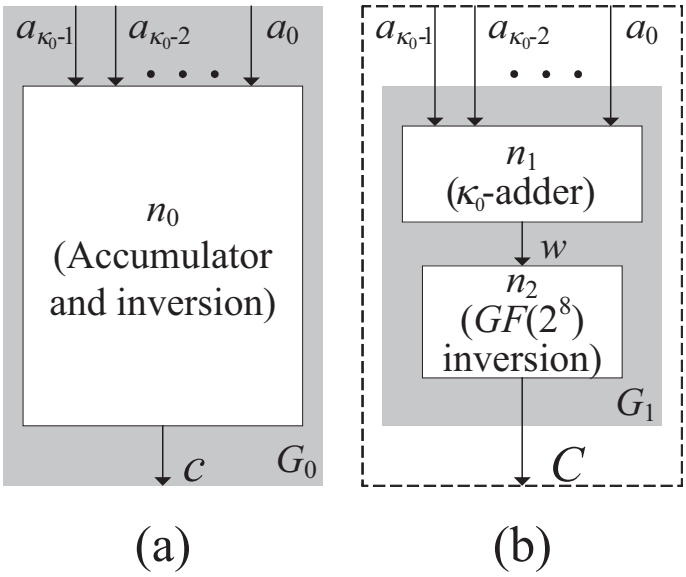


Fig. 3.11 GF-ACG for circuit connecting κ_0-input adder and inversion in serial.

a symbolic computation. Therefore, the verification method is efficient when the GF-ACG has a higher-degree function given by substitutions.

The algorithm stops when "IsSubstitutable($S, L'[i]$) = false" for every equation. The termination condition of this algorithm is that the internal structure (i.e., GF-ACG) does not include directed cycle graphs, that is, substitutions are not repeated more than once. This property is also implemented by **delete** $L'[i]$ on Line 8. The "false" suggests that the number of equations obtained from the internal structure is finite. In addition, Algorithm 3.4 simply consists of substitution operations to examine a proof figure. The correctness of Algorithm 3.4 is given from the proven soundness of the natural deduction system [121], which means that the conclusion (i.e., functional assertion) is correct if the premises (i.e., internal structure) are correct in the proof figure.

The above algorithm returns false if the node is wrong or the proof must consider arithmetic rules (e.g., addition and multiplication) because it does not exploit arithmetic rules and other inference rules of natural deduction (e.g., rules about $\vee, \wedge$ and $\supset$). In other words, the algorithm sometimes returns false even if the circuit is correct when the function is not derived deductively. Therefore, it is better to apply the ND-based method only to nodes derived deductively and those that would required to take an enormous amount of time to verify by the GB-based method.

To evaluate the efficiency of the ND-based method, we show the design and verification of the $GF(2^8)$ circuits consisting of an n-input adder and an inversion which are sometimes used in ECC decoders and cryptographic processors.

Table 3.5 Nodes, GF, and GF variables in Fig. 3.11

Nodes
[Accumulator and inversion]
$n_0 = (\{c = \left(\sum_{\zeta=0}^{\kappa_0-1} a_\zeta\right)^{254}\}, G_1)$
[Accumulator] $n_1 = (\{w = \sum_{\zeta=0}^{\kappa_0-1} a_i\}, G_2)$
[Inversion] $n_2 = (\{c = w^{254}\}, G_3)$
GF
$GF(2^8) = (\left(\beta^7, \beta^6, \ldots, \beta^0\right), (\{0,1\}, \{0,1\}, \ldots, \{0,1\}), \beta^8 + \beta^4 + \beta^3 + \beta^2 + 1)$
GF variables
$a_\zeta = (GF(2^8), (7,0)), 0 \leq \zeta \leq \kappa_0 - 1$
$c, w = (GF(2^8), (7,0))$

Table 3.6 Verification time of n_0 in Fig. 3.11 (s)

# of variables κ_0	2	3	4	5	6	7	8
Logic simulation	0.56	30.16	10,032.98	TO	TO	TO	TO
GB-based	0.33	0.49	130.84	TO	TO	TO	TO
ND-based	0.37	0.32	0.28	0.27	0.35	0.33	0.32

Figure 3.11 shows a GF-ACG for the target circuit and Table 3.5 shows the nodes, GFs, and GF variables in Fig. 3.11. The node "Accumulator and inversion" in Fig. 3.11(a) is the top-level node. Its functional assertion is given by

$$b = \left(\sum_{\zeta=0}^{\kappa_0-1} a_\zeta\right)^{254}. \tag{3.37}$$

This function is higher-degree and has a large number of terms in the expanded form because it performs an exponentiation of a multivariate polynomial. Fig. 3.11(b) shows the internal structure of the "Accumulator and inversion" node, which consists of two nodes: "κ_0-adder" and "$GF(2^8)$ inversion." The κ_0-adder node is described by 2-input adders (i.e., bit-parallel XOR) and $GF(2^8)$ inversion is described by an addition-chain exponentiation to compute $c = w^{254}$. (We describe the details of the inversion circuit in the following parts of this dissertation.)

The verification time of such circuits for $2 \leq \kappa_0 \leq 8$ was measured by open-source computer algebra system Risa/Asir on a Linux PC with Intel Xeon E5450 3.00 GHz processor and 32 GB RAM. Table 3.6 shows the results using the conventional RTL simulation, GB-based method, and ND-based method (the proposed method) where "TO" indicates that the verification did not finish

Table 3.7 Conversion rules to obtain PPRM form

Before	After
$\forall((var1 = exp_1)\ \&\ f(var1))$	$\forall(f(exp_1))$
$exp_1 = exp_2$	$\neg(exp_1 \oplus exp_2)$
$exp_1 \rightarrow exp_2$	$(\neg exp_1) \mid exp_2$
$exp_1 \mid exp_2$	$\neg((\neg exp_1)\ \&\ (\neg exp_2))$
$\neg exp_1$	$exp_1 \oplus 1$
$(exp_1 \oplus exp_2)\ \&\ exp_3$	$(exp_1\ \&\ exp_3)$ $\oplus(exp_2\ \&\ exp_3)$
$exp_1\ \&\ exp_1$	exp_1
$exp_1 \oplus exp_1$	0
$exp_1 \oplus 0$	exp_1
$0 \oplus 1$	1

in a day (i.e., 8.64×10^4 sec). Note that Table 3.6 shows the verification time only for the top-level node. The RTL simulation could not verify the circuits completely if the number of operands was more than five. This is because the verification requires 2^{40} test patterns when $\kappa_0 = 5$. The GB-based method also failed to verify the circuits when $\kappa_0 \geq 5$ because of the memory limitations caused by the large number of terms. More precisely, the number of terms becomes $\binom{254+\kappa_0-1}{\kappa_0-1}$ ($\approx 2^{62}$ in the case of $\kappa_0 = 5$). On the other hand, the ND-based method successfully verified such circuits in a fixed time independently of κ_0 because the verification time only depends on the number of equations representing the internal structure. Thus, we confirmed that the proposed method is efficient in verifying GF arithmetic circuits of high degrees.

PPRM-based evaluation method

A PPRM-based verification method was first introduced in [95], demonstrating very fast correctness proof of combinational circuits of AES components (S-box, MixColumns, etc.). This method is applicable to $GF(2)$-level verification, and cannot handle $GF(2^m)$-level verification.

PPRM stands for Positive Polarity Reed-Muller (form) and is a form of Boolean logic formulae, where they are represented by single stage AND-XOR expressions with no negation [126]. Any Boolean logic formulae can be converted to PPRM form by applying transformation rules shown in Tab. 3.7. Because PPRM is a canonical form [126], any logic formulae will be converted to the identical PPRM formula if and only if their corresponding truth tables are the same.

This verification method consists of the following steps: (Step 1) Describe predicates of both combinational circuit specification, called *reference*, and its implementation. A predicate $\Pi_{circuit}$

represents the input-output relation of a circuit and are in the form of

$$\Pi_{circuit}(in_0, \ldots, in_{\kappa_{in}-1}, out_0, \ldots, out_{\kappa_{out}-1}) \equiv$$
$$out_0 = f_0(in_0, \ldots, in_{\kappa_{in}-1}) \,\&\, \ldots \,\&\, out_{\kappa_{out}-1} = f_{\kappa_{out}}(in_0, \cdots, in_{\kappa_{in}}), \tag{3.38}$$

where κ_{in} and κ_{out} denote the number of input and output variables, respectively. Variables in_ρ and out_ϱ ($0 \leq \rho \leq \kappa_{in}$ and $0 \leq \varrho \leq \kappa_{out}$) represent the circuit's primary input and output, respectively. Boolean function $f_\varrho^{(b)}$ represents the circuit function.
(Step 2) Describe the following verification condition

$$\begin{aligned}\forall variables \,\{ & P_{ref}(in_0^{(r)}, \ldots, in_{\kappa_{in}-1}^{(r)}, out_0^{(r)}, \ldots, out_{\kappa_{out}-1}^{(r)}) \\ & \,\&\, P_{impl}(in_1^{(m)}, \ldots, in_{\kappa'_{in}-1}^{(m)}, out_0^{(m)}, \ldots, out_{\kappa_{out}-1}^{(m)}) \\ & \,\&\, in_0^{(r)} = in_0^{(m)} \,\&\, \ldots \,\&\, in_{\kappa_{in}-1}^{(r)} = in_{\kappa_{in}-1}^{(m)} \\ & \,\&\, out_0^{(r)} = out_0^{(m)} \,\&\, \cdots \,\&\, out_{\kappa_{out}-1}^{(r)} = out_{\kappa_{out}-1}^{(m)} \,\},\end{aligned} \tag{3.39}$$

where κ'_{in} is the number of primary inputs of the target circuit, and Π_{ref} and Π_{impl} denote the predicates corresponding to reference and target circuits, respectively. Variables $in_{\rho'}^{(r)}$ ($0 \leq \rho' \leq \kappa'_{in}-1$)and $in_\varrho^{(m)}$ denote the primary inputs of reference and target circuit, respectively. Variables $out_\rho^{(r)}$ and $out_\varrho^{(m)}$ denote the primary outputs of reference and target circuit, respectively. The meaning of this formula is that the values of all primary outputs in the implementation should equal to those in the reference, for any values of primary inputs. Please note that the number of primary inputs in the implementation can be larger than that in the reference ($\kappa'_{in} \geq \kappa_{in}$) [*4].
(Step 3) Convert the body of the verification condition to PPRM form, in order to determine the truth. The implementation circuit is equivalent to the reference, namely. it is correct, if and only if a constant 1 is obtained as a result[*5]. The Steps 2 and 3 can be mechanized.

Needless to say, this verification method is meaningful only when the "correct" reference circuits (i.e., golden model) can be constructed easily and the implementation circuits are mostly different from their reference because of various optimizations. For example, in the case of AES, this condition is applicable and a practical method of making reference AES components was introduced in [95]. Making correct GF adder and multiplier is easy because the adder is just a set of parallel XORs and the multiplier has a simple regular structure if implemented as Mastrovito or full-tree multiplier. Once correct GF adder and multiplier are built, correct S-box, MixColumns

[*4] In this case, the primary inputs $in_{\kappa_{in}}^{(m)}$ through $in_{\kappa'_{in}}^{(m)}$ should have no effect to primary outputs.

[*5] Determining the truth of this verification condition enables to check all PPRM representations of outputs simultaneously, while it is described in the above as if the check of each output is done separately.

and other AES components can be built up easily in a bottom-up manner. For instance, a correct $GF(2^8)$ inversion circuit on can be made by cascading multiple multipliers and squarers, by computing a^{254}, where a is input of the inverter. Although these straightforward circuits cannot be used in the practical AES implementations, because of too large size and too long critical path delay, it is sufficient as references. The real AES designs use much more optimized and/or masked components and the design tasks are difficult and complicated.

While the judgment of the truth of verification condition in step 3 is not always impossible by conventional logic manipulation methods such as BDD, Free binary DD (FDD) and conversions to the other canonical forms [94], this PPRM method is not only significantly faster [95], but also applicable to much larger circuits with large number of input variables[*6].

In the following, a new PPRM-based formula evaluation method is proposed. Algorithm 3.5 is used for verifying $GF(2)$-level circuit very fast using reference circuit generated the GB-based method.

Given a node over $GF(2)$, we first translate it into the corresponding logical expressions in the PPRM expression, then apply the PPRM-based evaluation to the PPRM expression with the equivalent reference expression stored in the library (denoted by **Lib**). If the given node is a fundamental arithmetic circuit such as adders and multipliers, the reference can be automatically retrieved from the library. If there is no adequate reference in the library, we (i.e., users) need to prepare a GF-ACG description in advance for the reference. The reference description is verified by the GB-based method in the reference generation step. The verified reference is transformed to logical expressions, and is then used in the PPRM-based evaluation.

In summary, the major new features of the proposed method are (1) to switch the verification method depending on the GF (i.e., $GF(2^m)$ or $GF(2)$), and (2) the semi-automatic reference generation for the PPRM-based method. The details of these features will be explained in the following. The basic concept of such verification method combining GB- and PPRM-based methods was shown in [111]. In this dissertation, we show in-depth quantitative and experimental evaluation of verification time of tower-field arithmetic circuits, how to describe correctness criteria of tower-field and masked circuits for the verification, and a systematic design flow exploiting the verification method in accordance with the results, while the previous work showed only verification of several circuits and application to a masked AES hardware.

[*6] Conventional logic manipulation methods are unsuitable for handling GF arithmetic circuits, because the GF circuits mostly consist of XOR gates (addition on $GF(2)$) and AND gates (multiplication on $GF(2)$). The use of many XOR gates often causes exponential growth of BDD and/or Sum-Of-Products/Products-Of-Sum logic formula size. In addition, it is difficult to apply SAT (SATisfiability) solvers to GF arithmetic circuits. The previous result in [81] showed that any SAT solver did not succeed in verifying multipliers over $GF(2^{16})$, which means that SAT solvers have difficulty in handling practical GF arithmetic circuits with more-than 32-bit operands.

Algorithm 3.5 Formula evaluation based on PPRM expansion with reference library or generation by GB-based method

Input: $GF(2)$-level node $n = (\boldsymbol{F}, G_{in})$, Reference library **Lib**
Output: Formula evaluation result $res \in \{\text{true}, \text{false}\}$

```
1: function PPRM-BasedEvaluation(n)
2:     list Υ ← LogicalExp(n);
3:     if Logical expression for function F in Lib then
4:         list ref ← GetReference(Lib, F);
5:         Bool res ← PPRM-Expansion(Υ, ref);
6:     else
7:         list G_ref ← GF-ACG-synthesis(F);
8:         node n_ref ← Top-levelNodeOf(G_ref);
9:         if GB-BasedEvaluation(n_ref) then
10:            list ref ← LogicalExp(n_ref);
11:            Bool res ← PPRM-Expansion(Υ, ref);
12:        else
13:            Bool res ← false;
14:        end if
15:    end if
16:    return res;
17: end function
```

In the following, we describe design of tower-field circuits which is frequently used in practical cryptographic hardware, especially based on $GF(2^m)$ arithmetic [7, 101, 123, 128]. In general, arithmetic circuits over tower fields are much smaller than those over standard extension fields, and many side channel countermeasures are based on this tower field technique. Tower-field circuits are highly optimized in logic-level, and GB-based evaluation sometimes requires huge-time to verify tower-field circuits. Thus, tower-field circuits can be main applications and examples of PPRM-based evaluation.

Figure 3.12 shows the overview of the κ_{in}-input κ_{out}-output combinational circuits based on this technique. Main computation is performed on a tower field, and an isomorphic mapping δ and its inverse mapping δ^{-1} are attached to the main computation. These functions are used to convert the data from/to the original extension field.

One weakness of the GB-based evaluation is that it is not suitable for component verification on $GF(2)$-level. We have conducted an experiment of component verification, by using Mastrovito and tower-field multipliers as design examples described in GF-ACG. The experimental setup is the same as that in Sect. 3.2.3.

Table 3.8 shows the description and verification results of the Mastrovito and tower-field multipliers, where "# vars." and "# eqs." in the columns "Highest-level" indicate the number of

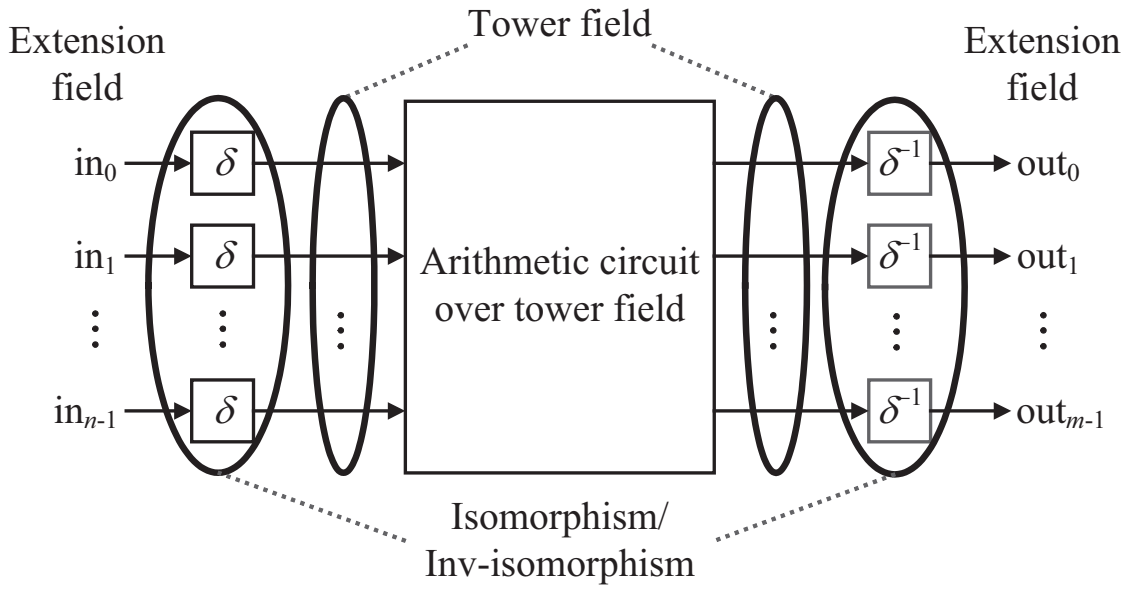


Fig. 3.12 Typical circuit structure of tower-field operation.

Table 3.8 Description and verification of $GF(2^m)$ multipliers

m	Mastrovito multipliers			Tower field multipliers		
	GB veri. time (s)	Highest level # vars.	Highest level # eqs.	GB veri. time (s)	Highest level # vars.	Highest level # eqs.
4	2.18	7	2	1.53	36	49
8	2.85	11	2	55.30	72	97
16	4.34	19	2	TO	144	193
32	7.98	35	2	TO	288	381
64	15.66	67	2	TO	576	769
128	39.21	131	2	TO	1,152	1,537

variables and equations in verifying the most time-consuming highest-level node, respectively. The GB-based verification time (i.e. GB veri. time) for Mastrovito multipliers basically increased by the square of the operand bit length m since the number of nodes in a GF-ACG was given by $O(m^2)$. Each node was verified in less than 1 second thanks to the well-hierarchical description. On the other hand, the GB-based verification times for tower field multiplier ran out from the 16-bit one due to the flattened description of isomorphic mappings[*7].

It is known that the numbers of variables and equations have a significant impact on the compu-

[*7] There is a difference between the verification results of this dissertation and the previous paper [64]. This is because the verification time in this dissertation includes isomorphic mappings from/to extension fields while that in [64] does not.

Table 3.9 Verification time for Mastrovito multipliers over $GF(2^m)$ (s)

m	PPRM veri. time		Veri. time in [81]		GB veri. time
	Bug-free	Bugs	Bug-free	Bugs	for ref. circuit
16	0.02	0.02	0.04	0.04	5.49
32	0.06	0.06	1.41	1.43	9.36
64	0.34	0.33	112.13	114.86	18.30
128	2.29	2.33	3,054.00	3,061.00	44.05
160	4.27	4.32	9,361.00	9,384.00	66.24
163	4.52	4.50	16,170.00	16,368.00	67.79
233	7.52	7.44	No data	No data	138.43
283	22.36	22.66	No data	No data	217.08

tational cost of Gröbner basis in general. According to [8], the worst-case complexity of Gröbner-basis computation is at least $O(2^{2^{\kappa/10}})$, that is, the double exponential of number of variables κ. In addition, it is shown that the arithmetic complexity bound in the cases of zero-dimensional ideal is $deg^{O(\kappa^2)}$ and that of the homogenized system having finitely many solutions is $\ell^{O(1)}deg^{O(k)}$, where deg is the maximum degree of equations and ℓ is the number of equations. Our verification is not always such cases, but we require at worst the above computational cost. The numbers of variables and equations for verifying tower-field circuits are respectively given by $\kappa = 3\kappa_{io}m$ and $\ell = 4\kappa_{io}m + 1$, where κ_{io} denotes the total number of inputs and outputs. In the case of multipliers, the numbers of variables and equations are $\kappa = 9m$ and $\ell = 12m + 1$ while those of Mastrovito multipliers are at most $\kappa = m + 3$ and $\ell = 2$, respectively. As a result, we had a difficulty in verifying the highest-level node in larger tower-field multipliers by the GB-based evaluation.

As another preliminary experiment, Tab. 3.9 shows the comparison of PPRM-based evaluation and the previous work [81] by verifying Mastrovito multipliers. In PPRM-based evaluation, the implementation and reference in GF-ACG are translated into those in conventional Boolean expressions. In addition, Tab. 3.9 also shows the GB-based evaluation time of the corresponding full-tree multiplier [63], which is simply composed of a partial product generator and an accumulator, and therefore is employed for references. Note here that the circuit architectures of full-tree multiplier is different from that of Mastrovito multiplier. Even when the architectures of the target and reference multiplier are different, the PPRM-based evaluation can rapidly verify Mastrovito multipliers using the reference multipliers. This means that the PPRM-based evaluation can efficiently verify tower-field multipliers using the corresponding extension field multiplier generated by means of GF-ACG. Thus, we can confirm the efficiency of the combination of GB- and PPRM-based evaluation methods. We have found also that the PPRM-based evaluation was much faster

Algorithm 3.6 Transformation from GF-ACG to logical expression

Input: GF-ACG $G = (\boldsymbol{N}, \boldsymbol{E})$
Output: Logical expression $\boldsymbol{\Lambda}$

```
1: function LOGICALEXP(G)
2:     Λ' ← ∅;
3:     for all (F, G_in) ∈ N do
4:         Λ' ← Λ' ∪ LogicalExp(G_in);
5:     end for
6:     Λ ← Compression(Λ', E);
7:     return Λ;
8: end function
```

than the GB-based evaluation and was applicable to larger designs. In addition, the PPRM-based evaluation was significantly faster than the previous verification report [81] and was able to verify multipliers over larger fields such as $GF(2^{233})$ and $GF(2^{283})$.

Through the above preliminary experiments, we have decided to use the PPRM-based for $GF(2)$-level component verification, instead of the GB-based method. In order to do this, we constructed a translation algorithm from a GF-ACG to the corresponding logical expressions. Algorithm 4.1 is necessary to verify a GF-ACG description of a component implementation. In this algorithm, when a GF-ACG G is given, we first extract a set of logic expressions from G recursively. Then, the set of logic expressions is minimized by compression operations according to the connections of internal edges.

Regarding to the reference circuits which are required in the PPRM-based method, these are constructed in advance, as already described before. In many cases, such as inversion circuits, we can easily generate and verify reference circuits, which are implemented with a simple architecture (e.g., without tower-field techniques), by the GB-based evaluation method. The PPRM-based method can efficiently verify practical (or highly-optimized) circuits using such reference circuits. Thus, the PPRM-based evaluation method is applicable to a wider variety of circuits than either of the methods solely even if an extra cost generating reference circuits is required.

Note that, since the reference circuits for the PPRM-based evaluation is generated through algebraic procedures (i.e., GB-based evaluation), the additional properties of cryptographic hardware such as validity of tower-field implementation are implicitly verified by the PPRM-based evaluation. In addition, because PPRM-based method examines inclusion relation of the reference and target circuits, it can be used even when the reference and target circuits have different bit length (e.g., verification of masked implementation) by describing the relation between additional input/output. We show how to apply the PPRM-based method to an masked implementation [114] in Section 3.3.5.

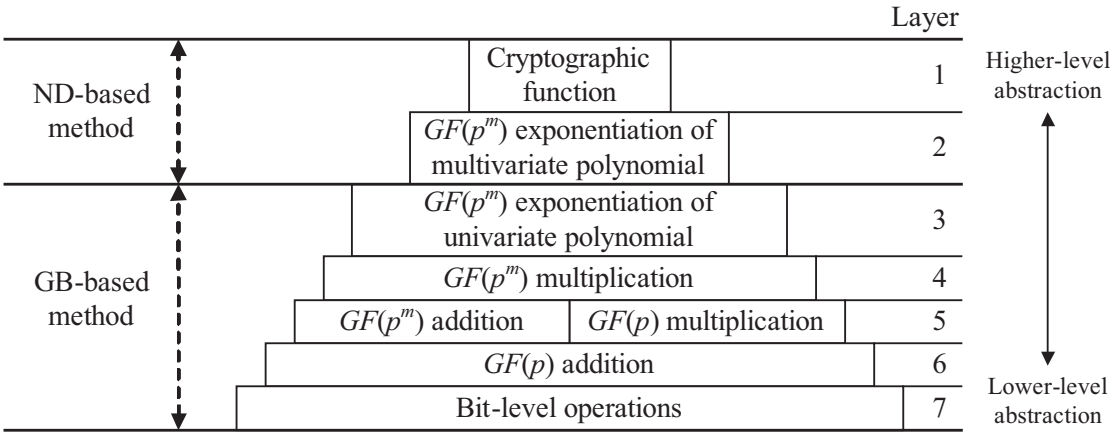


Fig. 3.13 Classification of circuit functions.

Proposed formal verification combining three formula evaluation methods

The ND-based method can verify nodes in which the functional assertion can be derived deductively from their internal structures. The PRRM-based method can verify nodes with logic-level (i.e., $GF(2)$-level) optimization if the nodes represent basic components where the reference circuit is easily prepared (using the GB-based method). On the other hand, the conventional GB-based method can be applied to any kind of GF arithmetic circuits, and can efficiently verify circuits with hierarchical structure and lower-degree functions. Therefore, the proposed verification method combines the above formula evaluation method to verify any kind of GF arithmetic circuits efficiently. First, we apply the PPRM-based method to $GF(2)$-level nodes. In addition, we apply the ND-based method to nodes derived deductively from their internal structures. Finally, we apply the GB-based method to the rest of nodes in order to perform complete verification.

We then describe how to use the three evaluation methods in a combination. It is clear that the PPRM-based method should be applied to $GF(2)$-level nodes. On the other hand, to use GB- and NB-based methods in a combination, we first divide arithmetic functions into seven layers to distinguish functions derived deductively. We then apply the ND-based method to functions belonging to the top two layers and the GB-based method to the rest. Fig. 3.13 shows the hierarchical structure of arithmetic functions that is commonly given by GF-ACG. This hierarchy consists of the Cryptographic function, $GF(p^m)$ exponentiation of multivariate polynomials, $GF(p^m)$ exponentiation of polynomials with one variable, $GF(p^m)$ multiplication, $GF(p^m)$ addition, $GF(p^m)$ multiplication, $GF(p^m)$ addition, and Bit-level operations from higher to lower levels of abstraction. In practical circuit design, such hierarchical structure of arithmetic functions are usually considered in order to provide rational description and to achieve high-performance.

The top layer includes the node for the whole Cryptographic hardware function and the sec-

ond layer includes the nodes for exponentiation of multivariate polynomials. These functions are necessarily implemented by simple serial or/and parallel combinations of subfunctions (e.g., adders, multipliers, and inversion). Therefore, such functions can be derived deductively and are suitable for the ND-based method. Note again that the GB-based method has difficulty in verifying functions such as those that belong to the top two layers since these functions usually have higher degrees. On the other hand, the rest of the functions, which belong to the lower layers, perform basic arithmetic operations, and most of them are low-degree functions even though there are many implementations of such functions. In other words, most of them are not derived deductively. Therefore, these functions should be verified by the GB-based method. Furthermore, several circuits with $GF(2)$-level optimization is not classified according to Fig. 3.13. For example, although a masked inversion circuit in [114] belongs to the second layer, but it is directly realized by $GF(2)$-level (i.e., bit-level) operations. Therefore, all nodes where their internal structures is given at $GF(2)$-level should be verified by the PPRM-based method, regardless of their belonging layer described in Fig. 3.13

In summary, our design method consists of three steps, as follows:

(Step 1) Describe the target circuit in the form of GF-ACG $(\boldsymbol{N}, \boldsymbol{E})$.

(Step 2) For each node n given by $(\boldsymbol{F}, G_{in}) \in \boldsymbol{N}$, determine which verification method should be applied.

(Step 3) Verify each node by either the GB-, ND-, or PPRM-based method.

In Step 1, it is essential to describe the circuit hierarchically according to Fig. 3.13 to ensure fast verification.

Algorithm 3.7 shows the entire verification flow used in Steps 2 and 3, which is obtained by extending Algorithm 3.1 for combining the GB-, ND-, PPRM-based formula evaluation methods. Given a GF-ACG, the formula evaluation based on either GB-, ND-, or PPRM-based method is applied to all the nodes having functional assertions and internal structures. In Line 4, for nodes with internal structures (i.e., $G \neq nil$), we recursively apply "NewVerify" to G and verify the node using either GB-, ND-, or PPRM-based method. On the other hand, for nodes without any internal structure (i.e., $G = nil$), "Verify" always returns true since the node represents a logic element (i.e., a pre-defined gate in a standard cell library). Line 6 checks whether or not the given node is $GF(2)$-level. This check can be easily realized by checking whether there are directed edges with $GF(2)$ variables in its internal structure. If it is true, the PPRM-based method is applied to the node. Line 8 corresponds to Step 2. "LayerOf" returns the layer level of the function, as shown in Fig. 3.13. LayerOf can be implemented easily by looking at the form of the function and whether G_{in} consists of decomposition/composition nodes. Consequently, each node is verified properly in Lines 9 or 11. If there are nodes whose functions are fully given as mathematical substitutions

Algorithm 3.7 Proposed GF-ACG verification

Input: GF-ACG $G = (\boldsymbol{N}, \boldsymbol{E})$
Output: Verification result $res \in \{\text{true}, \text{false}\}$

```
1: function NEWVERIFY(G)
2:     Bool res ← true;
3:     for all (F, G_in) ∈ N do
4:         if G' ≠ nil then
5:             res ← res & VerifyG_in;
6:             if G_in is GF(2)-level then
7:                 res ← res & PPRM-BasedEvaluation(F, G_in);
8:             else if LayerOf(F) ≤ 2 then
9:                 res ← res & ND-BasedEvaluation(F, G_in);
10:            else
11:                res ← res & GB-BasedEvaluation(F, G_in);
12:            end if
13:        end if
14:    end for
15:    return res;
16: end function
```

with higher degrees and large nodes verified by PPRM-based method, we can significantly reduce the total verification time.

3.4 Applications

3.4.1 $GF(2^m)$ parallel multipliers

In order to demonstrate the effectiveness and efficiency of the proposed method, this subsection shows its applications to design various GF arithmetic circuits and cryptographic hardware. In this section, we represent GF-ACGs without temporal operators when designing circuit without pipeline.

First, we describe formal design of full-tree multipliers over PRR-based $GF(2^m)$. PRR-based full-tree multipliers can be designed in the same manner as PB-based ones described in Section 3.2.3. Figure 3.14 shows a PRR-based $GF(2^2)$ full-tree multiplier with a modular polynomial $P(x) = x^3 + 1$. Table 3.11 shows nodes, GFs, and GF variables in Fig. 3.14. Since the modular polynomial is given by a binomial, reduction by $P(x)$ in nodes "SubPPB0," "SubPPB1," and "SubPPB2" in the third-level abstraction (i.e., Fig. 3.14(c)) can be performed by solely bit-wise permutation without logic gates. Generally, if there are less terms in $P(x)$, the number of logic gates in SubPPGs decreases, which motivates the usage of redundant GF representations.

Figure 3.15 shows the verification time of PRR-based multipliers for $2 \leq m \leq 32$. Table 3.12

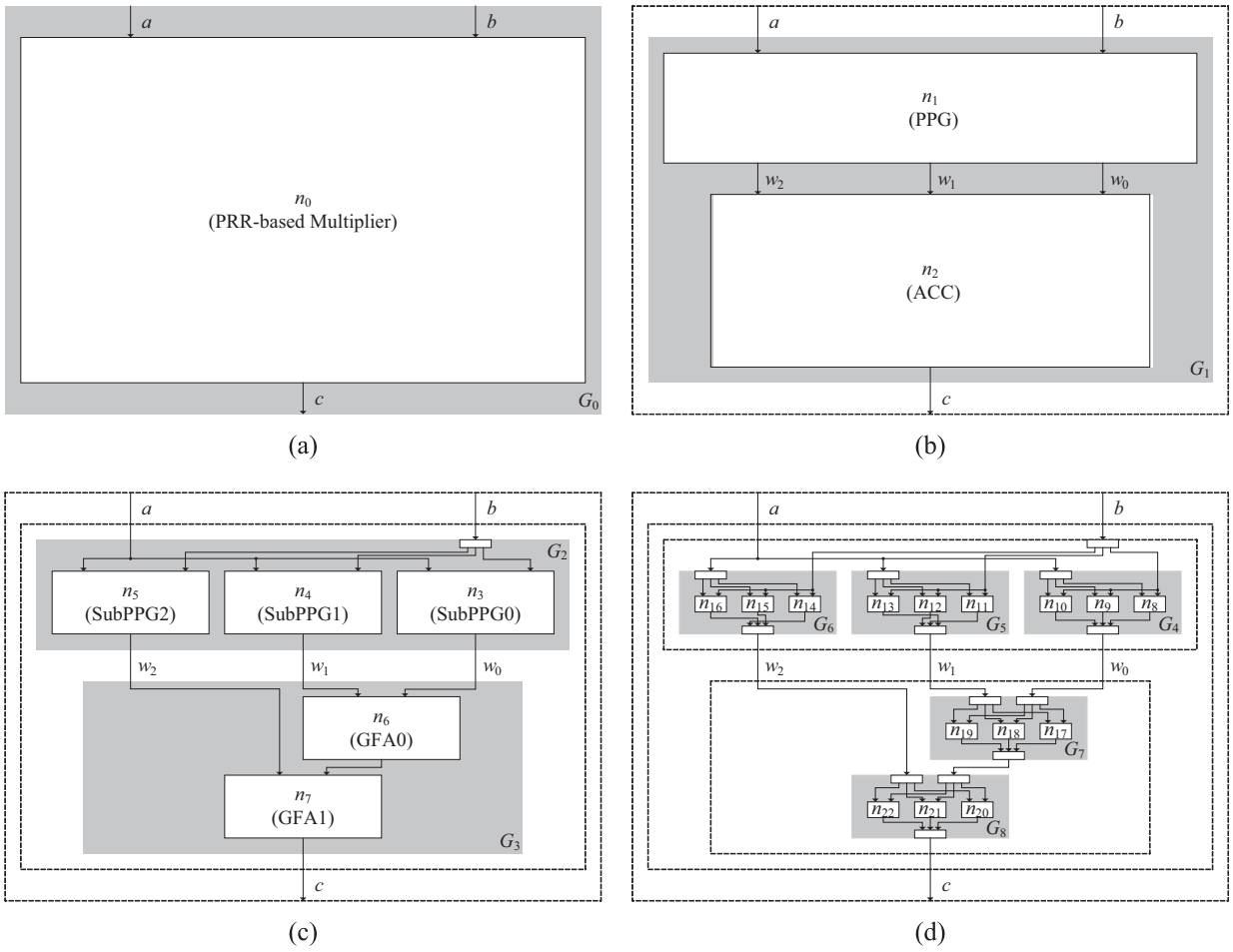


Fig. 3.14 GF-ACG for PRR-based $GF(2^2)$ full-tree multiplier: (a) top- to (d) lowest-level of abstraction.

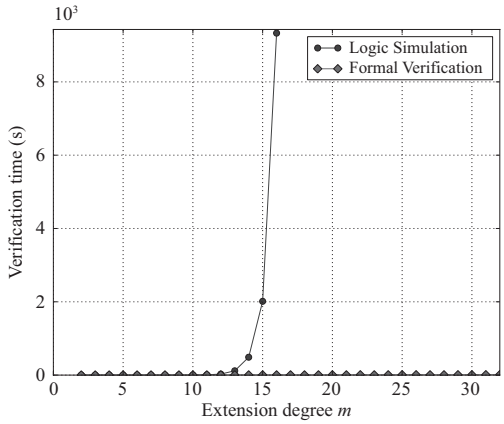


Fig. 3.15 Verification time of PRR-based full-tree multipliers.

Table 3.10 Nodes, GFs, and GF variables in Fig. 3.14.

Node
[PRR-based Multiplier] $n_0 = (\{c = a \times b\}, G_1)$
[PPG] $n_1 = (\{\sum_{j=0}^{2} w_j = a \times b\}, G_2)$,
[SubPPG0]
$n_3 = (\{w_0 = a \times b_0\}, G_4)$
[GF(2) Multiplier]
$n_8 = (\{w_{0,0} = a_0 \times b_0\}, nil)$,
$n_9 = (\{w_{0,1} = a_1 \times b_0\}, nil)$,
$n_{10} = (\{w_{0,2} = a_2 \times b_0\}, nil)$,
[SubPPG1]
$n_4 = (\{w_0 = a \times b_1\}, G_5)$
[GF(2) Multiplier]
$n_{11} = (\{w_{1,0} = a_0 \times b_1\}, nil)$,
$n_{12} = (\{w_{1,1} = a_1 \times b_1\}, nil)$,
$n_{13} = (\{w_{1,2} = a_2 \times b_1\}, nil)$,
[SubPPG2]
$n_5 = (\{w_2 = a \times b_2\}, G_6)$
[GF(2) Multiplier]
$n_{14} = (\{w_{2,0} = a_0 \times b_2\}, nil)$,
$n_{15} = (\{w_{2,1} = a_1 \times b_2\}, nil)$,
$n_{16} = (\{w_{2,2} = a_2 \times b_2\}, nil)$,
[Accumulator] $n_2 = (\{c = \sum_{j=0}^{2} w_j\}, G_3)$
[GFA0]
$n_6 = (\{u_0 = w_0 + w_1\}, G_7)$
[GF(2) Adder]
$n_{17} = (\{u_{0,0} = w_{0,0} + w_{1,0}\}, nil)$
$n_{18} = (\{u_{0,1} = w_{0,1} + w_{1,1}\}, nil)$
$n_{19} = (\{u_{0,2} = w_{0,2} + w_{1,2}\}, nil)$
[GFA1]
$n_7 = (\{c = u_0 + w_2\}, G_8)$
[GF(2) Adder]
$n_{20} = (\{c_0 = u_{0,0} + w_{2,0}\}, nil)$
$n_{21} = (\{c_1 = u_{0,1} + w_{2,1}\}, nil)$
$n_{22} = (\{c_2 = u_{0,2} + w_{2,2}\}, nil)$
GFs
$GF_{(2^2)} = (\left(x^2, x^1, x^0\right), (\{0,1\}, \{0,1\}, \{0,1\}), (x^2 + x + 1, x + 1))$
$GF_{(2)} = ((1), (\{0,1\}), nil)$
GF variables
$a, b, c = (GF_{(2^2)}^{(PRR)}, (2, 0))$
$w_0, w_1, w_2 = (GF_{(2^2)}^{(PRR)}, (2, 0))$
$u_0 = (GF_{(2^2)}^{(PRR)}, (2, 0))$
$a_0, a_1, a_2, b_0, b_1, b_2, c_0, c_1, c_2 = (GF_{(2)}, (0, 0))$
$w_{0,0}, w_{0,1}, w_{0,2}, w_{1,0}, w_{1,1}, w_{1,2}, w_{2,0}, w_{2,1}, w_{2,2} = (GF_{(2)}, (0, 0))$
$u_{0,0}, u_{0,1}, u_{0,2} = (GF_{(2)}, (0, 0))$

also shows typical multipliers for $m = 4$, 8, 16, 32, 64, and 128, where verification procedures are performed using an open-source computer algebra software Risa/Asir on a Linux personal

Table 3.11 Nodes, GFs, and GF variables in Fig. 3.14.

Node
[PRR-based Multiplier] $n_0 = (\{c = a \times b\}, G_1)$
[PPG] $n_1 = (\{\sum_{j=0}^{2} w_j = a \times b\}, G_2)$,
[SubPPGj] $n_{3+j} = (\{w_j = a \times b_j\}, G_{4+j})$
[GF(2) Multiplier]
$n_{4+3j+\hat{j}} = (\{g_{j,j+\hat{j} \bmod 3} = a_{\hat{j}} \times b_j\}, nil)$,
[Accumulator] $n_2 = (\{c = \sum_{j=0}^{2} w_j\}, G_3)$
[GFA0] $n_6 = (\{u_0 = w_0 + w_1\}, G_7)$
[GF(2) Adder]
$n_{17+j} = (\{u_{0,j} = w_{0,\hat{j}} + g_{1,\hat{j}}\}, nil)$
[GFA1] $n_7 = (\{c = u_0 + w_0\}, G_8)$
[GF(2) Adder]
$n_{20+j} = (\{c_j = u_{0,\hat{j}} + w_{0,\hat{j}}\}, nil)$
GFs
$GF_{(2^2)} = (\left(x^2, x^1, x^0\right), (\{0,1\}, \{0,1\}, \{0,1\}), (x^2 + x + 1, x + 1))$
$GF_{(2)} = ((1), (\{0,1\}), nil)$
GF variables
$a, b, c = (GF_{(2^2)}^{(PRR)}, (2,0))$
$w_0, w_1, w_2 = (GF_{(2^2)}^{(PRR)}, (2,0))$
$u_0 = (GF_{(2^2)}^{(PRR)}, (2,0))$
$a_0, a_1, a_2, b_0, b_1, b_2, c_0, c_1, c_2 = (GF_{(2)}, (0,0))$
$w_{0,0}, w_{0,1}, w_{0,2}, w_{1,0}, w_{1,1}, w_{1,2}, w_{2,0}, w_{2,1}, w_{2,2} = (GF_{(2)}, (0,0))$
$u_{0,0}, u_{0,1}, u_{0,2} = (GF_{(2)}, (0,0))$

Table 3.12 Verification time of PRR-based $GF(2^m)$ full-tree multipliers (s)

m	4	8	16	32	64	128
Logic simulation	0.37	0.41	9,329.98	N/A	N/A	N/A
This study	1.81	3.53	4.84	10.84	16.54	20.01

computer with an Intel Xeon E5450 3.00GHz processor and 32GB RAM. In order to compare our results, we also show the verification time of the corresponding Verilog HDL description using a typical logic simulator (Verilog-XL). We did not verify multipliers other than the $GF(2^{16})$ multipliers using the logic simulator in this experiment because verification time increases exponentially with an increase in the extension degree m. On the other hand, we completely verified the $GF(2^{128})$ multiplier. Thus, we confirmed the validity and efficiency of the proposed method. Note that other conventional verification methods would not be applicable to PRR-based multipliers because they did not show how to model PRR-based circuits and its don't-care condition.

3.4.2 $GF(2^8)$ inversion circuits

The multiplicative inversion of a ($\in GF(2^8)$) is given by $a^{-1} = a^{254}$ because any element of $GF(\omega)$ satisfies $a^{\omega} = a$. In cryptographic algorithms, the inversion of 0 is usually defined to be 0. The inversion operation is widely used in cryptographic algorithms such as AES. In particular, since $GF(2^m)$ inversions are known as one of the most useful functions for resistance against some cryptanalytic techniques [108] such as differential and linear cryptanalyses [15, 88], many byte-oriented ISO/IEC standard ciphers employ $GF(2^8)$ inversion. Thus, investigating $GF(2^m)$ inversion circuit implementation is quite valuable from academic and practical viewpoints; and therefore, various inversion circuits have been proposed because performance in the previously mentioned applications can be critical [33, 96, 97, 101, 106, 127].

In this paper, we design $GF(2^8)$ inversion circuits based on various representations including redundant ones using GF-ACG. The inversion circuits designed in this paper is based on tower-field arithmetic produced by repeating the field extension, which is denoted by $GF((2^4)^2)$ or $GF(((2^2)^2)^2)$. The use of tower field is a promising approach to implementing efficient $GF(2^m)$ inversion [66]. Note that there is a one-to-one mapping (i.e., isomorphism) between the elements of $GF(2^8)$ and those of the composite fields. The basic idea underlying the composite field approach is to reduce hardware cost by exploiting smaller arithmetic operations over subfield $GF((2^2)^2)$ or $GF(2^4)$ instead of $GF(2^8)$. This GF inversion over a composite field is efficiently implemented in the Itoh-Tsujii Algorithm (ITA) [66]. Figure 3.16 shows the typical datapath of tower-field inversion circuit based on the ITA. Each component denotes an arithmetic circuit over a subfield (i.e., $GF(2^4)$ or $GF((2^2)^2)$). See Section 5.2 for details of tower-field inversion.

The tower-field inversion circuits can be designed using GF-ACG. Figure 3.17 shows a GF-ACG for the inversion circuit combining NB, PRR, and RRB representations at the top two levels of abstraction, and Tab. 3.13 shows nodes, variables, and GFs in Fig. 3.17. (Note that figures of GF-ACG for lower-level of abstraction is omitted for the simplicity.) In Tab. 3.13, γ is a primitive element of the $GF(2^4)$. Node "$GF((2^4)^2)$ inversion" in Fig. 3.17(a) is the top-level node, which consists of five nodes including decomposition and composition nodes. Fig. 3.17(b) shows the internal structure of "$GF((2^4)^2)$ inversion." Here, the inputs (i.e., a_1 and a_0) are given by an NB, the intermediate value w_0 is given by a PR, and the outputs (i.e. b_1 and b_0) and w_1 are given by an RRB. Node n_1 computes w_0 and converts from NB to PRR representation, and its internal structure is given by a logic circuit. Node n_2 performs $GF(2^4)$ inversion and conversion from PR to RRB, and its internal structure is also given by a logic circuit. Finally, node n_3 performs multiplication over the RRB by two RRB-based multipliers over $GF(2^4)$.

Since nodes n_0, n_1, and n_2 are associated with variables of PRR-based $GF(2^4)$, these nodes cannot be verified by GF-ACG without the above extension. In this example, the algebraic relation

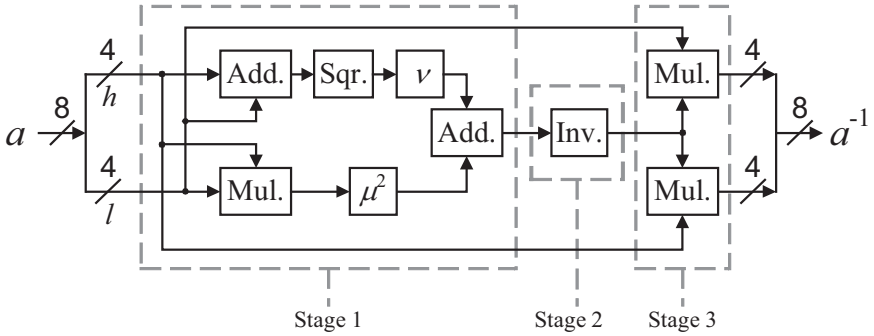
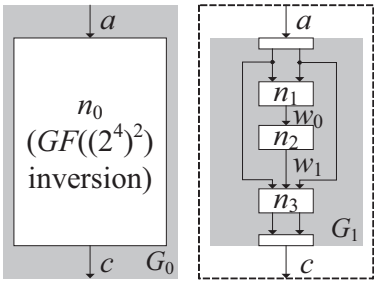


Fig. 3.16 Typical datapath of $GF(2^8)$ composite field inversion.

between x and β_0 is given by $\beta_0 = x^4 + x^3 + x^2 + 1$, which are required to verify n_0 and n_1 correctly. In addition, since w_0 is given by PRR, n_1 contains a composition node to PRR while n_2 contains a decomposition node from PRR. There, the LRR of w_0 should be correctly derived from the internal structure of n_1, and is required for the complete verification of n_2. The LRR of w_0 is given by $w_{0,0} + w_{0,1} + w_{0,2} + w_{0,3} + w_{0,4} = 0$, where $w_{0,j}$ $(0 \leq j \leq 4)$ is the jth bit of w_0.

Table 3.14 shows the verification time of $GF(2^8)$ inversion circuits base on various representations. Note that $GF(2^8)$ inversion circuits based on other representation(s) over $GF((2^4)^2)$ and $GF(((2^2)^2)^2)$ can be designed in the manner similar to the above. We could successfully verify them in several seconds. The verification time basically depended on the number of nodes (i.e., call of GB-, ND-, or PPRM-BasedEvaluation) in each internal structure of n_1, n_2, and n_3, because each node could be verified in a short time by the GB-based method. For example, Nogamis' circuits, which exploited both PB- and NB-based multipliers in a well-hierarchical manner, required the longest time among them. On the other hand, circuits of this work could be verified in the shortest time because the circuit was described in a flattened manner (i.e., the internal structures of n_1, n_2, and n_3 were given by logic circuits), although these nodes can be efficiently verified even by the GB-based method. To the authors' best knowledge, this is the first study to verify inversion circuits utilizing a combination of GF representations (including redundant ones) in a formal manner.

Table 3.14 also shows the area-time product of the eight $GF(2^8)$ inversion circuits using Synopsys Design Compiler with a TSMC 65-nm cell library. The result suggests that the inversion circuit based on a combination of redundant and non-redundant representations (This study (Section 5.2)) achieved the highest efficiency, which can be efficiently designed and verified by using the extended GF-ACG, while the conventional formal methods could not verify it because of redundant representation. Thus, we can confirm the effectiveness of the proposed GF-ACG sup-



(a) (b)

Fig. 3.17 GF-ACG for $GF((2^4)^2)$ inversion circuit based on combination of redundant and non-redundant GF arithmetic: (a) top- and (b) second-levels of abstraction.

Table 3.13 Nodes, GFs, and GF variables in Fig. 3.17

Nodes
[$GF((2^4)^2)$ Inversion] $n_0 = (\{c = a^{254}\}, G_1)$
[Decomposition] $n_{decomp} = (\{a_1\beta_1^{16} + a_0\beta_1 = a\}, nil)$
[$GF(2^4)$ 17th power] $n_1 = (\{w_0 = a_1a_0(x^4+x)^2 + (a_1+a_0)^2(x^4+x^3+x^2+1)\}, G_2)$,
[$GF(2^4)$ Inversion] $n_2 = (\{w_1 = w_0^{14}\}, G_3)$
[$GF(2^4)$ Multipliers] $n_3 = (\{c_1 = w_1 \times a_0, c_0 = w_1 \times c_1\}, G_4)$
[Composition] $n_{comp} = (\{c = c_1\beta_1^{16} + c_0\beta_1\}, nil)$
GF
$GF((2^4)^2) = (\left(b_1^{16}, b_1^1\right), \left(\{0, \gamma_0^1, \ldots, \gamma_0^{14}\}, \{0, \gamma_0^1, \ldots, \gamma_0^{14}\}\right), (\beta_1^2 + (\beta_0^4 + \beta_0)\beta_1 + \beta_0, 1))$
$GF(2^4)^{(NB)} = ((\beta_0^{2^3}, \beta_0^{2^2}, \ldots, \beta_0^{2^0}), (\{0,1\}, \{0,1\}, \ldots, \{0,1\}), (\beta_0^4 + \beta_0^3 + \beta_0^2 + \beta_0^1 + \beta_0^0, 1))$
$GF(2^4)^{(PRR)} = ((x^4, x^3, \ldots, x^0), (\{0,1\}, \{0,1\}, \ldots, \{0,1\}), (x^4 + x^3 + x^2 + x^1 + x^0, x + 1))$
$GF(2^4)^{(RRB)} = ((\beta_0^4, \beta_0^3, \ldots, \beta_0^0), (\{0,1\}, \{0,1\}, \ldots, \{0,1\}), (\beta_0^4 + \beta_0^3 + \beta_0^2 + \beta_0^1 + \beta_0^0, 1))$
GF variables
$a, c, = (GF((2^4)^2), (1,0))$
$a_1, a_0 = (GF_{(2^4)}^{(NB)}, (3,0))$
$c_1, c_0 = (GF_{(2^4)}^{(RRB)}, (4,0))$
$w_0 = (GF_{(2^4)}^{(PRR)}, (4,0))$
$w_1 = (GF_{(2^4)}^{(RRB)}, (4,0))$

porting both redundant and non-redundant representations. See Section 5.2 for detail of This study (Section 5.2).

Table 3.14 Verification time of $GF(2^8)$ tower-field inversion circuits

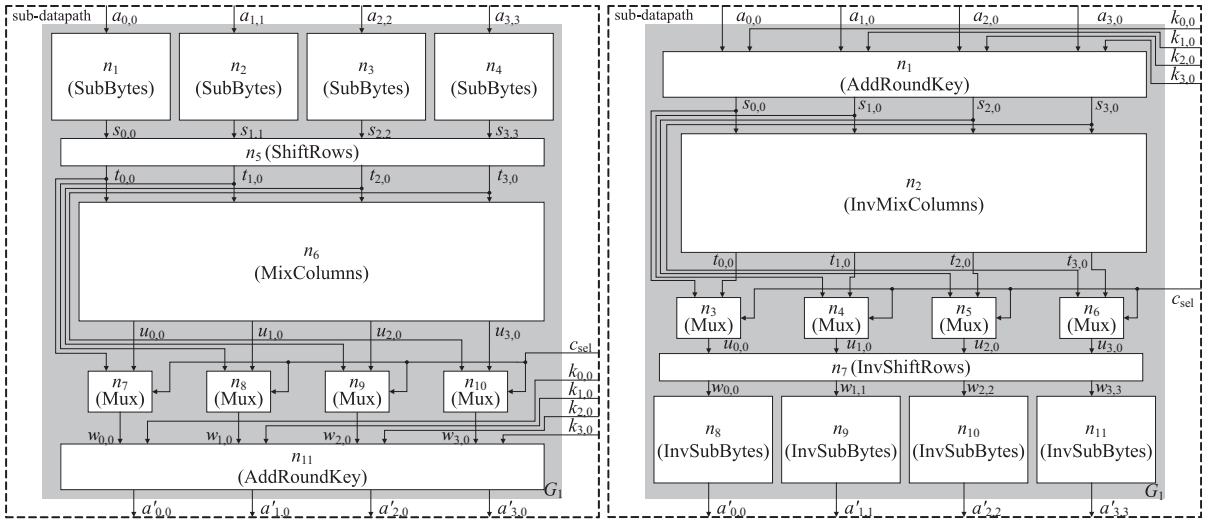
	Field	Representation		Verification time [s]
		Tower field	Indeterminate field	
Satoh et al. [127]	$GF(((2^2)^2)^2)$	PB	PB	3.12
Canright [33]	$GF(((2^2)^2)^2)$	NB	PB	3.92
Nogami et al. [106]	$GF(((2^2)^2)^2)$	PB and NB	PB and NB	5.45
Rudra et al. [123]	$GF((2^4)^2)$	PB	PB	2.93
Joen et al. [67]	$GF((2^4)^2)$	PB	PB	2.93
Nekado et al. [101]	$GF((2^4)^2)$	NB	RRB	2.00
This study	$GF((2^4)^2)$	NB	PR	3.55
This study (Section 5.2)	$GF((2^4)^2)$	NB	NB, PRR, and RRB	1.20

3.4.3 AES hardware

In this subsection, we focus on the design and verification of AES encryption and decryption hardware which perform one round per clock cycle. Note that we do not consider the key scheduling function to generate round keys. However, AES hardware with round key generation on-the-fly can also be designed in the same manner.

The datapath of AES hardware is divided into four sub-datapaths, and each of them encrypts (or decrypts) four bytes independently. Figure 3.18 shows the GF-ACGs of the first sub-datapaths for the AES (a) encryption and (b) decryption hardware at the second and third levels of abstraction. The top-level nodes (i.e., the entire datapaths) are composed of sub-datapaths connected in parallel. (The figures and descriptions of the top-level nodes are omitted for simplicity.) Tables 3.15 and 3.16 show the nodes of first sub-datapaths of the AES encryption and decryption hardware, respectively. Note that the second, third, and fourth sub-datapaths are designed in the same manner.

Here, we explain the details of the encryption hardware. Node "sub-datapath" is the second-level node, whose inputs are four-byte data, a four-byte round key, and one-bit control signal c_{sel}. The functional assertion, which is derived deductively from its four sub-functions and Mux, is very high degree. Nodes "SubBytes," "ShiftRows," "MixColumns," and "AddRoundKey" in the internal structure have functional assertions corresponding to the sub-functions. As mentioned above, "SubBytes" has a high-degree function with one variable and "MixColumns" has four linear functions with four variables. Therefore, their combination also has high-degree functions with a large number of terms. Lower-level abstractions are detailed in [63]. The decryption hardware is also designed in the same manner as the encryption hardware. "InvSubBytes," "InvShiftRows," and "InvMixColumns" perform the inverse functions of "SubBytes," "ShiftRows,"



(a) (b)

Fig. 3.18 GF-ACG for sub-datapath of AES (a) encryption and (b) decryption hardware.

Table 3.15 Nodes, GF, and variables in AES encryption hardware (Fig. 3.18(a))

Nodes
[sub-datapath]
$n_0 = (\{a'_{\epsilon,0} = k_{\epsilon,0} + c_{\rm sel}(\sum_{d_r=0}^{7} c_{d_r} a_{\epsilon,\epsilon}^{-2^{d_r}} + c_8) + (1-c_{\rm sel}) \sum_{e_r=0}^{3} v_{e_r-\epsilon}(\sum_{d_r=0}^{7} c_{d_r} a_{e_r,e_r}^{-2^{d_r}} + c_8) \mid 0 \le \epsilon \le 3\}, G_1)$
[SubBytes] $n_{\epsilon+1} = (\{s_{\epsilon,\epsilon} = \sum_{d_r=0}^{7} c_{d_r} a_{\epsilon,\epsilon}^{-2^{d_r}} + c_8\}, G_{\epsilon+2})$
[ShiftRows] $n_5 = (\{t_{\epsilon,0} = s_{\epsilon,\epsilon} \mid 0 \le \epsilon \le 3\}, G_6)$
[MixColumns] $n_6 = (\{u_{\epsilon,0} = \sum_{e_r=0}^{3} v_{e_r-\epsilon} t_{e_r,0} \mid 0 \le \epsilon \le 3\}, G_7)$
[Mux] $n_{\epsilon+7} = (\{w_{\epsilon,0} = c_{\rm sel} t_{\epsilon,0} + (1-c_{\rm sel}) u_{\epsilon,0}\}, G_{\epsilon+8})$
[AddRoundKey] $n_{11} = (\{a'_{\epsilon,0} = k_{\epsilon,0} + w_{\epsilon,0} \mid 0 \le \epsilon \le 3\}, G_{12})$
GF
$GF(2^8) = ((\beta^7, \beta^6, \ldots, \beta^0), (\{0,1\}, \{0,1\}, \ldots, \{0,1\})\beta^8 + \beta^4 + \beta^3 + \beta^1 + \beta^0)$
GF variables
$a_{\epsilon,\epsilon}, a'_{\epsilon,0}, s_{\epsilon,\epsilon}, t_{\epsilon,0}, u_{\epsilon,0}, k_{\epsilon,0}, w_{\epsilon,0} = (GF(2^8), (7,0)),\ (0 \le \epsilon \le 3)$
$c_{\rm sel} = (GF(2^8), (0,0))$

and "MixColumns," respectively. Note that "AddRoundKey" has the same function as in the encryption since it is a simple key addition (i.e., bit-parallel XOR with a round key). See [63] for how to obtain algebraic description of AES hardware.

For comparison, the AES encryption and decryption hardware were verified by the GB-based method and the proposed verification method. Tables 3.17 and 3.18 show the verification

Table 3.16 Nodes, GF, and variables in AES decryption hardware (Fig. 3.18(b))

Nodes
[sub-datapath]
$n_0 = (\{a'_{\epsilon,0} = (\sum_{d_r=0}^{7} c_{d_r}(c_{\rm sel}(k_{\epsilon,0} + a_{\epsilon,0}) + (1 - c_{\rm sel}) \times \sum_{e_r=0}^{3}(v'_{e_r-\epsilon}(k_{e_r,-\epsilon} + a_{e_r,-\epsilon})))^{2^{d_r}} + c_8)^{254} \mid 0 \le \epsilon \le 3\}, G_1)$
[AddRoundKey] $n_{11} = (\{s_{\epsilon,-\epsilon} = k_{\epsilon,-\epsilon} + a_{\epsilon,-\epsilon} \mid 0 \le \epsilon \le 3\}, G_{12})$
[InvMixColumns] $n_6 = (\{t_{\epsilon,-\epsilon} = \sum_{e_r=0}^{3} v'_{e_r-\epsilon} s_{e_r,0} \mid 0 \le \epsilon \le 3\}, G_7)$
[InvShiftRows] $n_5 = (\{w_{\epsilon,0} = u_{\epsilon,-\epsilon} \mid 0 \le \epsilon \le 3\}, G_6)$
$[InvSubBytes] n_{\epsilon+1} = (\{a'_{\epsilon,0} = (\sum_{d_r=0}^{7} c_{d_r} w_{\epsilon,0}^{2^{d_r}} + c_8)^{254}\}, G_{\epsilon+2}), 0 \le \epsilon \le 3$
[Mux] $n_{\epsilon+7} = (\{w_{\epsilon,0} = c_{\rm sel} t_{\epsilon,0} + (1 - c_{\rm sel}) u_{\epsilon,0}\}, G_{\epsilon+8}), 0 \le \epsilon 3$
GF
$GF(2^8) = ((\beta^7, \beta^6, \ldots, \beta^0), (\{0,1\}, \{0,1\}, \ldots, \{0,1\}), \beta^8 + \beta^4 + \beta^3 + \beta^1 + \beta^0)$
GF variables
$a_{\epsilon,-\epsilon}, a'_{\epsilon,0}, s_{\epsilon,-\epsilon}, t_{\epsilon,0}, u_{\epsilon,0}, k_{\epsilon,0}, w_{\epsilon,0} = (GF(2^8), (7,0)),\ (0 \le \epsilon \le 3)$
$c_{\rm sel} = (GF(2^8), (0,0))$

Table 3.17 Verification time of AES encryption hardware

Nodes	Verification Time (s)	
	GB-based	This study
AESencryption	0.44	0.07
sub-datapath	0.25	0.07
SubBytes	3.72	3.72
ShiftRows	0	0
MixColumns	0.84	0.69
Mux	0.63	0.35
AddRoundKey	0.24	0.21
Total	6.12	5.07

times of the encryption and decryption hardware, respectively, where "AESencryption" and "AESdecryption" indicate the top-level nodes of the encryption and decryption hardware, respectively. The experimental setup was the same as Section III. In this verification, the ND-based method was applied to "AESencryption" and "sub-datapath" since "AESencryption" belongs to the ECC/cryptographic layer and "sub-datapath" belongs to the $GF(p^m)$ exponentiation of multivariate polynomial layer in Fig. 3.13. Note that "ShiftRows" and "InvShiftRows" were not verified since they were implemented only by wiring. As a result, the GB-based method failed to verify "AESdecryption," but could verify "AESencryption." This was caused by the order of SubBytes and MixColumns (InvSubBytes and InvMixColumns). The function of "AESencryption" does not have fewer terms than "AESdecryption" since the encryption hardware

Table 3.18 Verification time of AES decryption hardware

Nodes	Verification Time (s)	
	GB-based	This study
AESdecryption	N/A	0.07
sub-datapath	N/A	0.07
AddRoundKey	0.24	0.21
InvMixColumns	1.71	1.71
Mux	0.63	0.35
InvShiftRows	0	0
InvSubBytes	4.58	4.58
Total	N/A	6.99

performs SubBytes before MixColumns. Therefore, when verifying "AESdecryption," the GB-based method required an enormous amount of memory and computational time because of the huge number of terms in the polynomials. On the other hand, the proposed method successfully verified "AESdecryption" in less than a second since the ND-based method could verify the function without expanding the polynomials. This result confirms the efficiency of the proposed method in verifying cryptographic hardware.

3.4.4 Masked AES hardware

We have applied the proposed method to a practical 128-bit AES hardware with a masking-based countermeasure against side-channel attack [114]. The above hardware is designed with GF-ACG in the same manner as [111], where each node has a function described by algebraic equations and an internal structure. Table 3.19 shows the nodes, GFs and variables using in GF-ACGs of top three levels of abstraction for the masked datapath. The Masked Rand sub-datapath is one of four 32bit data randomization nodes consisting of one MaskedDataPath, one MaskPath and one MaskedRoundKey at the highest level of the hierarchy. The MaskedDataPath then consists of four MaskedSBox units, one ShiftRows, one MixColumns, four Mux units and one AddRoundKey. The MaskPath is represented by four ShiftRows units, one MixColumns and four Mux units, and the Maskedroundkey is represented by four 3-input/1-output adders.

The critical component designed here is Masked inversion in the MaskedSBox. The functional assertion of this masked inversion is given by

$$b = (a + x)^{254} + y. \tag{3.40}$$

Note again that, although 3.40 belongs to the second layer of Fig. 3.13, MaskedSBox is a $GF(2)$-

Table 3.19 Nodes, GF, and GF variables for Masked Rand sub-datapath

Nodes
[Masked Rand sub-datapath]
$n_0 = (\{a'_{\epsilon,0} = k_{\epsilon,0} + z_{\epsilon,0} + (1 - c_{\rm sel})y_{\epsilon,\epsilon} + c_{\rm sel}\sum_{e_r=0}^{3} v_{e_r-\epsilon}y_{e_r,e_r}$ $+c_{\rm sel}((\sum_{d_r=0}^{7} c_{d_r}(a_{\epsilon,\epsilon} + z_{\epsilon,\epsilon})^{-2^{d_r}} + c_8) + y_{\epsilon,\epsilon})$ $+(1 - c_{\rm sel})\sum_{e_r=0}^{3} v_{e_r-\epsilon}((\sum_{d_r=0}^{7} c_{d_r}(a_{e_r,e_r} + z_{e_r,e_r})^{-2^{d_r}} + c_8) + y_{e_r,e_r}) \mid 0 \le \epsilon \le 3\}, G_1)$
[MaskedDataPath]
$n_1 = (\{a'_{\epsilon,0} = k'_{\epsilon,0} + c_{\rm sel}((\sum_{d_r=0}^{7} c_{d_r}(a_{\epsilon,\epsilon} + z_{\epsilon,\epsilon})^{-2^{d_r}} + c_8) + y_{\epsilon,\epsilon})$ $+(1 - c_{\rm sel})\sum_{e_r=0}^{3} v_{e_r-\epsilon}((\sum_{d_r=0}^{7} c_{d_r}(a_{e_r,e_r} + z_{e_r,e_r})^{-2^{d_r}} + c_8) + y_{e_r,e_r}) \mid 0 \le \epsilon \le 3\}, G_2)$
[MaskedSBox] $n_{\epsilon+4} = (\{s_{\epsilon,\epsilon} = (\sum_{d_r=0}^{7} c_{d_r}(a_{\epsilon,\epsilon} + z_{\epsilon,\epsilon})^{-2^{d_r}} + c_8) + y_{\epsilon,\epsilon}\}, G_{\epsilon+5}),\ (0 \le \epsilon \le 3)$
[ShiftRow] $n_8 = (\{t_{\epsilon,0} = s_{\epsilon,\epsilon} \mid 0 \le \epsilon \le 3\}, G_9)$
[MixColumn] $n_9 = (\{u_{\epsilon,0} = \sum_{e_r=0}^{3} v_{e_r-\epsilon}t_{e_r,0}, \mid 0 \le \epsilon \le 3\}, G_{10})$
[Mux] $n_{\epsilon+10} = (\{w_\epsilon = c_{\rm sel}t_{\epsilon,0} + (1 - c_{\rm sel})u_{\epsilon,0}\}, G_{\epsilon+33}),\ (0 \le \epsilon \le 3)$
[AddRoundKey] $n_{14} = (\{a'_{\epsilon,0} = k_{\epsilon,0} + w_\epsilon \mid 0 \le \epsilon \le 3\}, G_{11})$
[MaskMixColumn] $n_2 = (\{y'_{\epsilon,0} = (1 - c_{\rm sel})y_{\epsilon,\epsilon} + c_{\rm sel}\sum_{e_r=0}^{3} v_{e_r-\epsilon}y_{e_r,e_r} \mid 0 \le \epsilon \le 3\}, G_3)$
[MaskedKeyScheduling] $n_3 = (\{k'_{\epsilon,0} = k_{\epsilon,0} + z_{\epsilon,0} + y'_{\epsilon,0} \mid 0 \le \epsilon \le 3\}, G_4)$
GF
$GF(2^8) = ((\beta^7, \beta^6, \ldots, \beta^0), (\{0,1\}, \{0,1\}, \ldots, \{0,1\}), \beta^8 + \beta^4 + \beta^3 + \beta^1 + \beta^0)$
GF variables
$a_{\epsilon,\epsilon}, a'_{\epsilon,0}, z_{\epsilon,\epsilon}, z_{\epsilon,0}, y_{\epsilon,\epsilon}, y'_{\epsilon,0}, s_{\epsilon,\epsilon}, t_{\epsilon,0}, u_{\epsilon,0}, k_{\epsilon,0}, k'_{\epsilon,0}, w_\epsilon = (GF(2^8), (7,0)),\ (0 \le \epsilon \le 3)$
$c_{\rm sel} = (GF(2^8), (0,0))$

level circuit.

To verify the MaskedSBox validly, we obtain an additional criterion (i.e., relation between mask and masked values) for masked components which is used in the masked implementations [45, 114]. Typical implementation of masked component $f_m(md, m_0, ..., m_k)$ receives mask value(s) m_0 ... m_k ($k \ge 0$) and a masked input $md = mask(d, m_0, ..., m_k)$ (d is an unmasked input), and returns masked output. If there is a corresponding unmasked reference $f_r(d)$, the f_m is correct when the following condition holds (Fig. 3.19):

$$\forall d, m_0, ..., m_k \ \{f_r(d) = mask^{-1}(f_m(mask(d, m_0, ..., m_k), m_0, ..., m_k), m_0, ..., m_k)\}. \tag{3.41}$$

For comparison, we verified the MaskedSBox by both the GB-based and the PPRM-based verification methods. Using the GB-based method, we could not succeed in verifying maskedSBox, since the composite-field circuit has three (i.e., more-than-one) isomorphism functions. On the other hand, by the PPRM-based method, we could succeed in verification in approximately 1 second.

Table 3.20 shows the verification time for the whole masked datapath by the proposed method, where the experimental setup is the same as that in Section 3.2. In Table 3.20 the verification times of the MixColumns and MaskedRoundKey include those of the lower-level design. In summary,

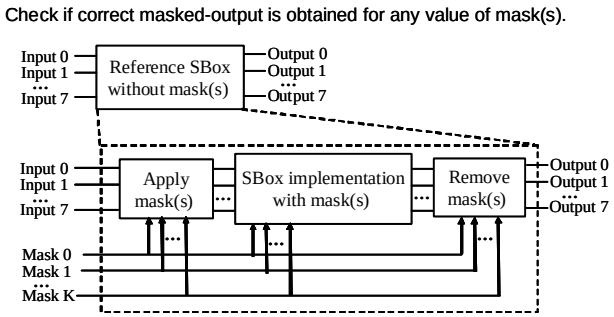


Fig. 3.19 Additional property of DPA-resistant operations.

Table 3.20 Verification times for Masked Rand datapath

Node	Num.	Veri. time [sec]	Procedure
Masked Data Randomization	1	0.14	ND
Masked sub-datapath	4	0.10	ND
MaskedDataPath	4	0.11	ND
MaskedSBox	16	0.93	PPRM
ShiftRow	4	0.00	GB
MixColumn	4	0.83	GB
Mux	16	0.01	PPRM
AddRoundKey	4	0.03	PPRM
MaskMixColumn	4	0.21	GB
MaskedRoundKey	4	0.41	GB
Total	61	2.77	

the proposed method could verify all of the datapath circuit in all design levels within 3 seconds thanks to the combination of three formula evaluation method. Note that, since GB-based method requires a larger time for verifying higher-level nodes (i.e., Masked Data Randomization, Masked sub-datapath, and MaskedDataPath), the ND-based method makes the verification about 76 times faster than that without the ND-based method [111].

3.4.5 LED hardware resistant to DPAs based on pipeline

The LED block cipher [57] (i.e., LED) is a 64-bit lightweight block cipher based on $GF(2^4)$ arithmetic. LED is well known as a smaller implementation in compared to other conventional block ciphers. In this study, we consider the 64-bit-key LED that operates on a 4×4 matrix of nibbles (one nibble = four bits), whose elements are variables over $GF(2^4)$. The intermediate state M in the encryption and the secret key K are represented by the nibbles $m_{\epsilon,\varepsilon}$ and $k_{\epsilon,\varepsilon}$ ($0 \leq \epsilon \leq 3$ and $0 \leq \varepsilon \leq 3$).

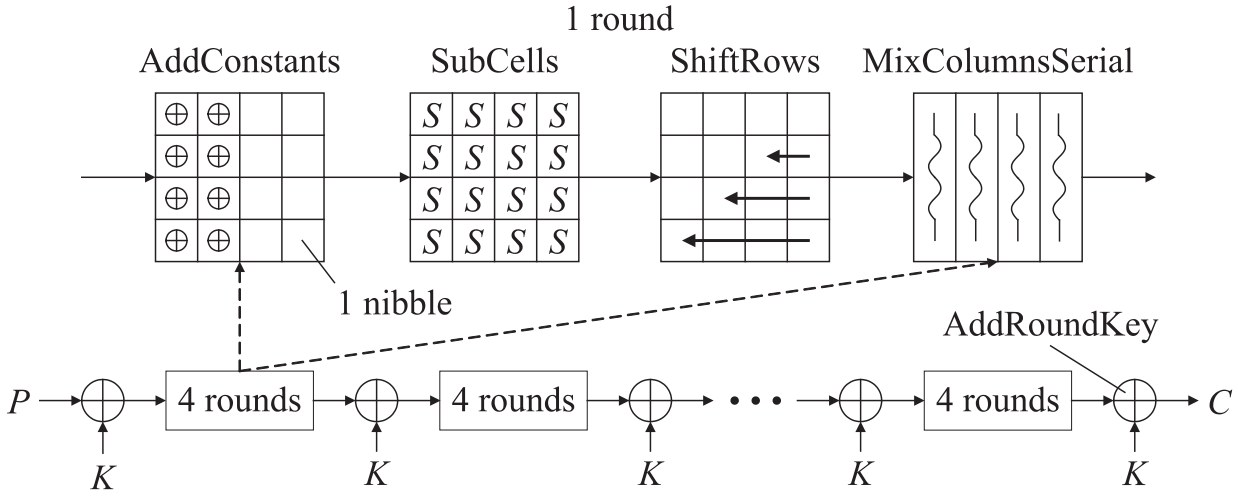


Fig. 3.20 LED encryption flow.

Figure 3.20 provides an overview of the LED encryption flow. The 64-bit-key LED encryption comprises 32 rounds. Each round consists of four sub-functions: AddConstants, SubCells, ShiftRows, and MixColumnsSerial similarly to AES. The intermediate value and key are added at the AddRoundKey step every four rounds. Each sub-function is performed over $GF(2^4)$ with an irreducible polynomial $x^4 + x + 1$. Let $a_{\epsilon,\varepsilon}, s_{\epsilon,\varepsilon}, t_{\epsilon,\varepsilon}, w_{\epsilon,\varepsilon}$, and $a'_{\epsilon,\varepsilon}$ be the output of the sub-functions and AddRoundKey, respectively. The algebraic representation of each sub-function is given as follows.

1. AddConstants: $a_{\epsilon,\varepsilon} = m_{\epsilon,\varepsilon} + rcon_{\epsilon,\varepsilon}$,
   where $rcon_{\epsilon,\varepsilon}$ is a constant value.

2. SubCells: $s_{\epsilon,\varepsilon} = \sum_{d_r=0}^{14} c_{d_r} a_{\epsilon,\varepsilon}^{d_r}$,
   where c_{d_r} is a constant of $GF(2^4)$.

3. ShiftRows: $t_{\epsilon,\varepsilon} = s_{\epsilon,\varepsilon-\epsilon}$,
   where the εth row is cyclically shifted to the left.

4. MixColumnsSerial: $w_{\epsilon,\varepsilon} = \sum_{h=0}^{3} e_{\epsilon,\epsilon'} t_{\epsilon',\varepsilon}$,
   where $e_{\epsilon,\varepsilon}$ is the (ϵ,ε)th element of a constant matrix and $w_{\epsilon,\varepsilon}$ corresponds to the input to the next round when AddRoundKey is not performed.

5. AddRoundKey: $a'_{\epsilon,\varepsilon} = w_{\epsilon,\varepsilon} + k_{\epsilon,\varepsilon}$,
   where $a'_{\epsilon,\varepsilon}$ corresponds to the input to the next round.

We then apply a countermeasure against DPAs using pipelining to LED hardware. This sub-

section focuses on a design of Threshold Implementation (TI), which is a state-of-the-art countermeasure. TI combines dynamic hazard reduction by pipelining and a cryptographic protocol called Multi-Party Computation (MPC). In MPC, an m-bit secret data $a(\in GF(2^m))$ is divided into σ shares additively (i.e., $a = a_0 + a_1 + \cdots + a_\varphi + \cdots + a_{\sigma-1}, a_l \in GF(2^m)$) to prevent attackers from estimating secret data a even if they obtain partial shares. In addition, TI randomly divides a into σ shares in order for DPA-leakage to disappear. Here, because TI does not cause any information leakage by means of dynamic hazard as a result of pipelining, TI is said to be superior to other sharing-based countermeasures.

DPA-leakage is mainly caused by a non-linear operation (i.e., S-box). We then design a TI-based S-box. (Please see [103] for details about TI.) Let $a(\in GF(2^4))$ and $S(a)$ be the input to the S-box and a bijective function (i.e., S-box) in LED. In addition, let s be the output of the S-box. We cannot compute $s = S(a)$ directly without generating DPA-leakage. To prevent such leakages, we divide a and s into three shares, where $a = a_0 + a_1 + a_2$ and $s = s_1 + s_2 + s_3$, respectively. Instead of S, we then compute a 12-bit input and 12-bit output function S', which meets $(s_0, s_1, s_2) = S'(a_0, a_1, a_2)$. Here, $S(a)$ is divided into F and G, which satisfy $S(a) = F(G(a))$ because S' should be implemented with a two-stage pipeline architecture to reduce dynamic hazard and its DPA-leakage. The crucial feature of TI is that F and G are realized by partial functions F_l and G_l $(0 \leq l \leq 2)$ that satisfy the following conditions:

1. *Correctness*
   The sum of shares becomes original data. In other words, $a = a_0+a_1+a_2, g = g_0+g_1+g_2$, and $s = s_0 + s_1 + s_2$, where g is the output of function G and g_l is its share.

2. *Noncompleteness*
   A sub-function G_l is independent of an input share g_l: $g_0 = G_0(a_1, a_2), g_1 = G_1(a_0, a_2), g_2 = G_2(a_0, a_1)$. Function F has the same construction.

3. *Uniformity*
   The variables of a, g, and s and the shares a_l, g_l, and s_l are uniformly distributed. In other words, for arbitrary value $v \in GF(2^4)$ and $u \in GF(2^4)^3$, the probability of $s_l = v$ and $s = u$ is $1/|GF(2^4)|$ and $1/|GF(2^4)^3|$, respectively. The other variables a_l, g_l, a, and g have the same probabilities.

Fig. 3.21 shows the entire architecture of a TI-based LED hardware and S-box designed in this study. The architecture has a 192-bit datapath because 64-bit secret data (i.e., plaintext) is divided into three shares. A plaintext is first divided into three shares, and a ciphertext is given as the sum of encrypted shares. SubCells consists of the aforementioned TI-based S-boxes and other

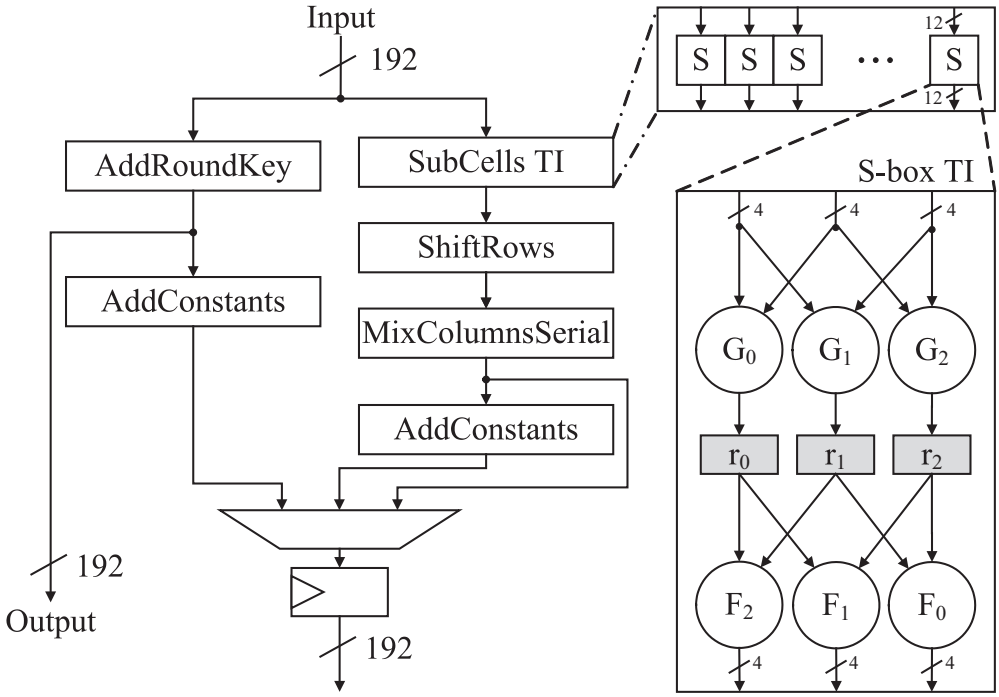


Fig. 3.21 Architecture of TI-based LED hardware and S-box.

operations are constructed by simply duplicating original ones.

Figure 3.22 shows GF-ACGs for the TI-based LED hardware of Fig. 3.21. Table 3.21 shows the nodes, GFs, and GF variables. Figure 3.22(a) shows the top-level node, where $a_{i,\varepsilon,l}$ and $b_{i,\varepsilon,l}$ are the divided input and output, respectively. The secret key $k_{i,\varepsilon}$ and round constant $rcon_{i,\varepsilon}$ are also divided. Figure 3.22(a) shows the internal structure of the "LED TI datapath." The functional assertions of "AddConstants," "ShiftRows," "MixColumnsSerial," and "AddRoundKey" are previously described. The functional assertion of "SubCells TI" is derived as functions that compute $s_{i,\varepsilon,l}$ from $a_{i,\varepsilon,0}$, $a_{i,\varepsilon,1}$, and $a_{i,\varepsilon,2}$ with one stage pipeline. Figure 3.22(b) shows the internal structure of "SubCells TI," which consists of 16 "S-box TIs." Finally, Fig. 3.22(c) shows the internal structure of a "S-box TI," which is implemented by "Function F_l," "Function G_l," and pipeline registers "Reg r_l." The functional assertions of "Function F_l" and "Function G_l" are given by the aforementioned method and their internal structures are logic circuit.

Table 3.22 shows the verification time of the GF-ACG shown in Fig. 3.22, in which the proposed verification procedures are performed using an open-source computer algebra software Risa/Asir on a Linux PC with an Intel Xeon E5450 3.00GHz processor and 32GB RAM. The pipelined tamper-resistant LED hardware was successfully verified in approximately 1 h whereas the conventional formal verification method would fail because of the pipeline architecture. The logic simulation could not also verify it in a day. Thus, the LTL-based representation and GB-

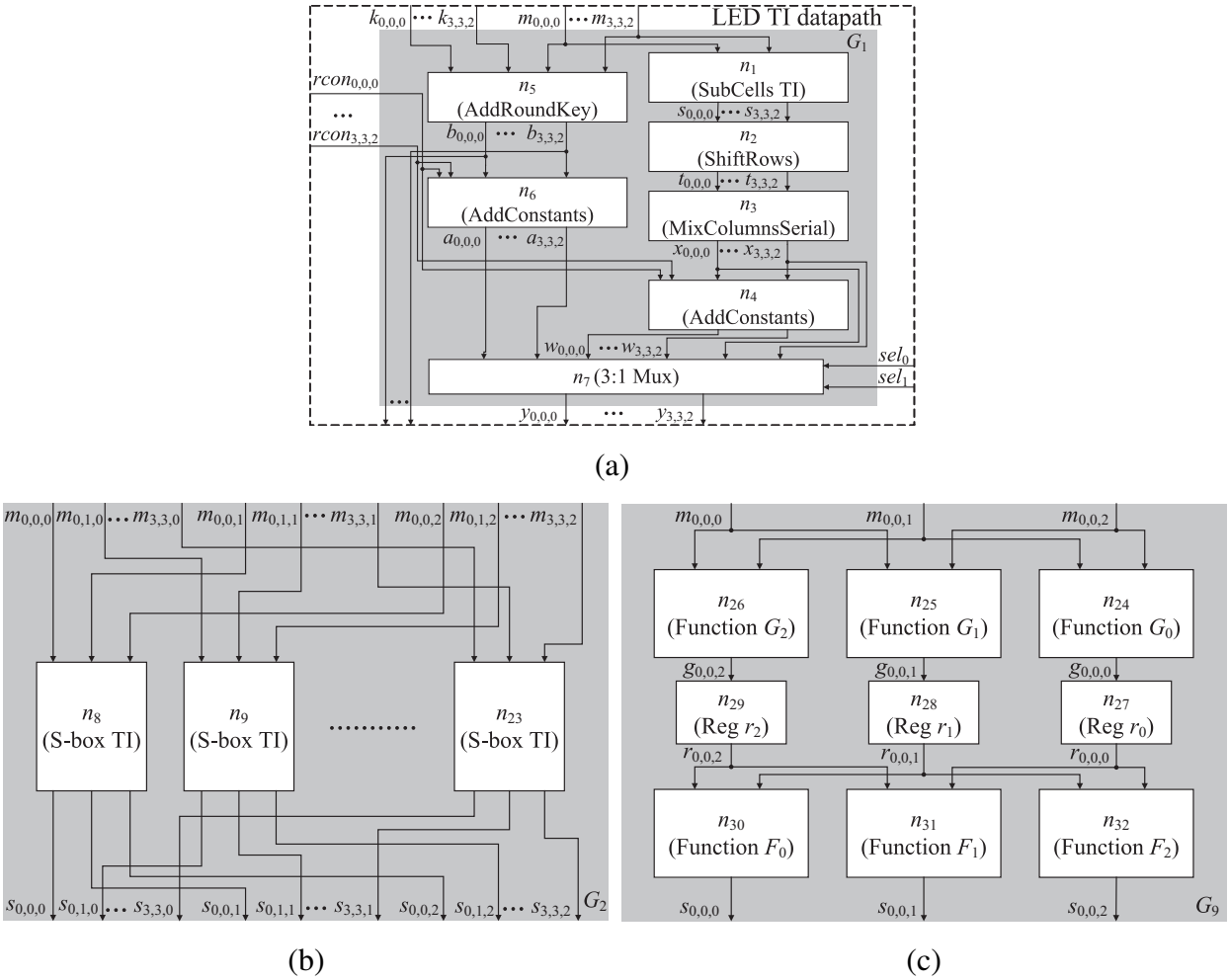


Fig. 3.22 GF-ACG for TI-based LED hardware: (a) top-level node and internal structure of (b) SubCells TI and (c) S-box TI.

based method are useful for designing and verifying such practical pipelined circuits. Note that we used only GB-based method in this experiment (i.e., without ND- and PPRM-based methods) in order to evaluate the influence of LTL-based representation on GB computation, but ND- and PPRM-based methods can be used for speeding up the verification. In particular, the usage of ND-based method would make it possible to verify the node "LED TI datapath" within a second, considering the above results about AES hardware.

Table 3.21 Nodes, GF, and GF variables in Fig. 3.22

Nodes
[LED TI datapath]
$n_0 = (\{\mathcal{X}(\mathcal{N}a'_{0,0,0} = (1-\mathcal{N}sel_0)(1-\mathcal{N}sel_1)(a_{0,0,0}+k_{0,0,0}+$ $rcon_{0,0,0})+\mathcal{N}sel_0(1-\mathcal{N}sel_1)(\sum_{e_r=0}^{3} v_{0,e_r}F_0(G_1(a_{e_r,0,2},$ $a_{e_r,0,0}), G_2(a_{e_r,0,0}, a_{e_r,0,1}))+rcon_{0,0,0})+(1-\mathcal{N}sel_0)sel_1$ $\sum_{e_r=0}^{3} v_{0,e_r}F_0(G_1(a_{e_r,0,2}, a_{e_r,0,0}), G_2(a_{e_r,0,0}, a_{e_r,0,1}))),$ $\vdots$ $\}, G_1)$
[SubCells TI]
$n_1 = (\{\mathcal{X}(\mathcal{N}s_{\epsilon,j,0} = F_0(G_1(a_{\epsilon,\varepsilon,2}, a_{\epsilon,\varepsilon,0}), G_2(a_{\epsilon,\varepsilon,0}, a_{\epsilon,\varepsilon,1}))),$ $\mathcal{X}(\mathcal{N}s_{\epsilon,\varepsilon,1} = F_1(G_2(a_{\epsilon,\varepsilon,0}, a_{\epsilon,\varepsilon,1}), G_0(a_{\epsilon,\varepsilon,1}, a_{\epsilon,\varepsilon,2}))),$ $\mathcal{X}(\mathcal{N}s_{\epsilon,\varepsilon,2} = F_2(G_0(a_{\epsilon,\varepsilon,1}, a_{\epsilon,\varepsilon,2}), G_1(a_{\epsilon,\varepsilon,2}, a_{\epsilon,\varepsilon,0}))$ $\mid 0 \le \epsilon \le 3, 0 \le \varepsilon \le 3\}, G_2)$
[S-box TI]
$n_{24} = (\{\mathcal{X}(g_{0,0,0} = G_0(a_{0,0,1}, a_{0,0,2}))\}, G_{25})$
$n_{25} = (\{\mathcal{X}(g_{0,0,1} = G_1(a_{0,0,2}, a_{0,0,0}))\}, G_{26})$
$n_{26} = (\{\mathcal{X}(g_{0,0,2} = G_2(a_{0,0,0}, a_{0,0,1}))\}, G_{27})$
$n_{27+l} = (\{\mathcal{X}(\mathcal{N}r_{0,0,l} = g_{0,0,l})\}, nil)\ (0 \le l \le 3)$
$n_{30} = (\{\mathcal{X}(s_{0,0,0} = F_0(r_{0,0,1}, r_{0,0,2}))\}, G_{28})$
$n_{31} = (\{\mathcal{X}(s_{0,0,1} = F_1(r_{0,0,2}, r_{0,0,0}))\}, G_{29})$
$n_{32} = (\{\mathcal{X}(s_{0,0,2} = F_2(r_{0,0,0}, r_{0,0,1}))\}, G_{30})$
$\vdots$
[ShiftRows]
$n_2 = (\{\mathcal{X}(t_{\epsilon,\varepsilon,l} = s_{\epsilon,\varepsilon-\epsilon,l}) \mid (0 \le \epsilon \le 3, 0 \le \varepsilon \le 3, 0 \le l \le 2)\}, G_3)$
[MixColumnsSerial]
$n_3 = (\{\mathcal{X}(w_{\epsilon,\varepsilon,l} = \sum_{e_r=0}^{3} v_{\epsilon,e_r}t_{e_r,\varepsilon,l}) \mid (0 \le \epsilon \le 3, 0 \le \varepsilon \le 3, 0 \le l \le 2)\}, G_4)$
[AddConstants]
$n_4 = (\{\mathcal{X}(w_{\epsilon,\varepsilon,l} = w_{\epsilon,\varepsilon,l} + rcon_{\epsilon,\varepsilon,l}) \mid (0 \le \epsilon \le 3, 0 \le \varepsilon \le 3, 0 \le l \le 2)\}, G_5)$
$n_6 = (\{\mathcal{X}(y_{\epsilon,\varepsilon,l} = w_{\epsilon,\varepsilon,l} + rcon_{\epsilon,\varepsilon,l}) \mid (0 \le \epsilon \le 3, 0 \le \varepsilon \le 3, 0 \le l \le 2)\}, G_7)$
[AddRoundKey]
$n_5 = (\{\mathcal{X}(\mathcal{N}z_{\epsilon,\varepsilon,l} = a_{\epsilon,\varepsilon,l} + k_{\epsilon,\varepsilon,l}) \mid (0 \le \epsilon \le 3, 0 \le \varepsilon \le 3, 0 \le l \le 2)\}, G_6)$
[3:1 Mux]
$n_7 = (\{\mathcal{X}(a'_{\epsilon,\varepsilon,l} = (1-sel_0)(1-sel_1)y_{\epsilon,\varepsilon,l} + sel_0(1-sel_1)w_{\epsilon,\varepsilon,l} + (1-sel_0)sel_1w_{\epsilon,\varepsilon,l})$ $\mid (0 \le \epsilon \le 3, 0 \le \varepsilon \le 3, 0 \le l \le 2)\}, G_8)$
GFs
$GF(2^4) = (\left(\beta^3, \beta^2, \beta^1, \beta^0\right), (\{0,1\}, \{0,1\}, \{0,1\}, \{0,1\}), \beta^4 + \beta + 1)$
$GF(2) = (\left(\beta^0\right), (\{0,1\}), nil)$
GF variables
$a_{\epsilon,\varepsilon,l}, a'_{\epsilon,\varepsilon,l}, s_{\epsilon,\varepsilon,l}, t_{\epsilon,\varepsilon,l}, w_{\epsilon,\varepsilon,l}, w_{\epsilon,\varepsilon,l}, y_{\epsilon,\varepsilon,l}, z_{\epsilon,\varepsilon,l}, f_{\epsilon,\varepsilon,l}, g_{\epsilon,\varepsilon,l}, rcon_{\epsilon,\varepsilon,l}, k_{\epsilon,\varepsilon,l}$ $= (GF(2^4), (3,0)), (0 \le \epsilon \le 3, 0 \le \varepsilon \le 3, 0 \le l \le 2)$
$sel_0, sel_1 = (GF(2^4), (0,0))$

3.5 Discussion

3.5.1 Comparison with conventional formal verification methods

Although we showed the verification result of logic synthesis for a comparison with conventional method, there are other types of formal verification methods, as descrbed in Section 2.5. In

Table 3.22 Verification time of GF-ACG for TI-based LED hardware

Graph name	Verification time
LED TI datapath	3,153.19
AddConstants	0.24
SubCells	7.69
ShiftRows	0.19
MixColumnsSerial	1.10
AddRoundKey	0.57
3:1Mux	0.90
Total	3,163.88

earlier related works, functional verification methods were primarily based on decision diagrams (DDs) and binary moment diagrams (BMDs) [28, 30]. However, as described in Sect. 2.5, these methods are basically limited to integer arithmetic and focus on only small component circuits such as adders and multipliers. Although there are some DDs for GFs [82, 135], handling practical GFs such as $GF(2^{16})$ and $GF(2^{32})$ is difficult. According to [81], a BDD- (Binary DD-)based method required 1,899.69 s to verify a $GF(2^{16})$ multiplier, and could not verify a $GF(2^{22})$ multiplier in 10 h. In addition to DD-based methods, SAT- and SMT-solvers are also used for the functional verification of arithmetic circuits in some commercial EDA tools. Many SAT-solvers can efficiently solve satisfiability of CNF (Conjunctive Normal Form) formulae, but they have difficulty in verifying GF circuits, which are usually described in an AND-XOR expression. Although a SAT-solver called CryptoMiniSAT has been developed for cryptographic applications, it cannot verify a PB-based Mastrovito multiplier over $GF(2^{16})$ in 10 h [81]. Applying SMT-solvers to GF arithmetic is also difficult because SMT-solvers have also been developed basically for integer arithmetic. In addition, such methods require reference circuits (i.e., golden models), which are often not prepared in advance. On the other hand, recently, algebraic methods based on a GB were also presented in [79–81], which verified up to 163-bit GF multipliers effectively without any reference model. However, these methods were only applied to multipliers based on polynomial basis (PB). It seems difficult to apply them to larger and practical circuits commonly used in cryptographic hardware (e.g., multipliers over $GF(2^{233})$ and $GF(2^{283})$ in elliptic curve cryptography and circuits based on an NB and redundant representations for compact and efficient implementation). In contrast, the proposed method supports various arithmetic algorithms (e.g., full-tree, Mastrovito, and Massey-Omura) and representations including both redundant and non-redundant representations (i.e., PB, NB, PRR, and RRB) in addition to PB- and PRR-based multipliers used for our experimental evaluation in this dissertation. Thus, we can confirm the effectiveness of the proposed method.

Table 3.23 Classification of modern ciphers

	Applicable		Not applicable
GF	$GF(2^8)$	$GF(2^4)$	$GF(2)$
Ciphers	AES, Camellia, MUGI, SNOW, KCipher-2	Piccolo, LED, TWINE	PRESENT, PRINCE, SIMON

3.5.2 Applicability and generality

The AES encryption hardware designed in this paper performs the subfunctions in the order of SubBytes, ShiftRows, MixColumns, and AddRoundKey, which is a naive implementation. Such naive hardware can be verified even by the conventional GB-based method as shown in the previous paper [63]. On the other hand, in many previous works on AES hardware implementation, resister-retiming and subfunction-reordering techniques heve been proposed to reduce the circuit delay and area [59, 87, 128]. We should consider these optimized techniques for designing practical AES hardware. The conventional GB-based method would require huger time to verify such practical AES encryption datapaths that perform AddRouneKey or MixColumns prior to SubBytes because the function contains the 254th power of multivariate polynomials as a result of the reordering. On the other hand, the proposed method can efficiently verify such practical datapaths where the function can be derived deductively even after the resister-retiming and subfunction-reordering. Thus, the applications of GF-ACG-based design can be extended by the proposed method.

A further application of the proposed method to other symmetric key algorithms would be possible because many modern ciphers are represented by $GF(2^m)$ $(m \geq 4)$ arithmetic. Table 3.23 shows a classification of some modern ciphers by GF used. Some of the ISO/IEC 18033 standard ciphers with $GF(2^8)$ arithmetic, such as Camellia [7], SNOW [53], MUGI [61], and KCipher-2 [132], can be designed and verified using the proposed method. Some of lightweight ciphers with $GF(2^4)$ arithmetic, such as LED [57], Piccolo [131], and TWINE [136], are also possible applications. On the other hand, it would be difficult to apply for ciphers which are not based on $GF(2^m)$ arithmetic and/or cannot be described in a hierarchical manner such as PRESENT [19], PRINCE [24], and SIMON [117].

3.6 Conclusion

This section proposed a new formal design methodology for cryptographic hardware. The proposed GF-ACG has the following advantages: to represent various GF representations including

Table 3.24 Summary of verification time of various GF arithmetic circuits (s)

	PB-based $GF(2^{16})$ mult.	PRR-based $GF(2^{16})$ mult.	AES enc. hardware	AES dec. hardware	Masked AES hardware	TI-based LED hardware
Logic simulation	9,330.00	TO	TO	TO	TO	TO
BDD	1,899.69	N/A	TO	TO	TO	N/A
SAT solver	TO	N/A	TO	TO	TO	N/A
Existing GF-ACG	5.52	N/A	6.12	TO	TO	N/A
This study	5.50	4.84	5.07	6.99	2.77	3,163.88

redundant ones and to handle sequential arithmetic circuits such as pipelined ones. In addition, we proposed two new formula evaluation methods for verifying GF arithmetic circuits which cannot be verified by the conventional GB-based formula evaluation in a practical time, and then proposed the formal verification based on a combination of three formula evaluation methods. The proposed formal verification can verify many practical cryptographic hardware, which cannot be verified by any other conventional method.

The proposed formal design method has a far larger design space than conventional methods, which covers many practical cryptographic hardware. Table 3.24 shows a brief summary of the verification results of various GF arithmetic circuits and cryptographic hardware by the conventional and proposed method, where "N/A" means "Not Applicable." The values for BDD and SAT solver are derived from [81]. Table 3.24 clearly indicates the advantage of the proposed method in designing and verifying GF arithmetic circuits and cryptographic hardware.

4

Automatic Generation System for Cryptographic Hardware

4.1 Introduction

Chapter 2 presented the formal design methodology of cryptographic hardware. Although the proposed GF-ACG is useful for designing and verifying various GF arithmetic circuits, GF-ACG cannot be used with the conventional EDA tools such as logic synthesizer. Therefore, Chapter 3 provides an automatic generation system of GF arithmetic circuits for cryptographic hardware in order to unify the proposed design methodology and conventional EDA tools. While the proposed system focuses only GF multipliers, the synthesis methods and transformation from GF-ACG to HDL are useful for other GF arithmetic circuits. In addition, since $GF(q^m)$ adders can be implemented by digit-parallel $GF(q)$ adders, automatic generation of $GF(q^m)$ multipliers is useful for designing a wider variety of $GF(q^m)$ circuit, many of which is constructed by a combination of adders and multipliers over $GF(q^m)$.

In this chapter, we first introduce the overview of proposed systems. We then describe automatic

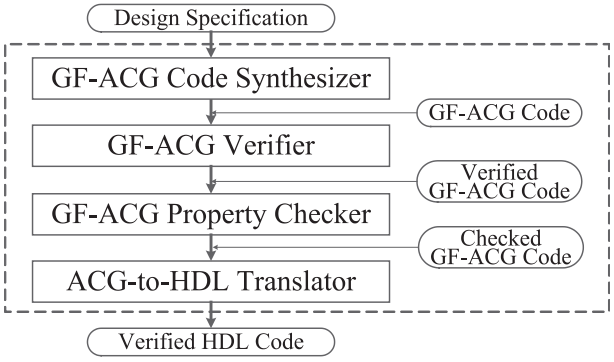


Fig. 4.1 Block diagram of GF-AMG.

Algorithm 4.1 Translate GF-ACG to HDL

Input: GF-ACG $G = (\boldsymbol{N}, \boldsymbol{E})$
Output: HDL Description $\boldsymbol{D}$

```
1: function MAPPING(G)
2:     D' ← ∅; str S;
3:     for all (F, G') ∈ N do
4:         if G' ≠ nil then
5:             D' ← D' ∪ Mapping(G');
6:         end if
7:     end for
8:     S ← GFACGtoHDLmodule(G);
9:     D ← D' ∪ {S};
10:    return D
11: end function
```

generation of $GF(p^m)$ multipliers where $p = 2, 3, 5, 7$, and 11. In addition, we present automatic generation of formally-proven DPA-resistant GF multipliers based on GMS. In addition to the circuit function, the DPA-resistance is formally verified by the algorithms proposed in Section 4.4. The capability and performance are evaluated through experimental generation of various multipliers.

4.2 System Overview

Figure 4.1 is a block diagram of GF-AMG. It consists of (i) the GF-ACG Code Synthesizer, (ii) the GF-ACG Verifier, and (iii) the ACG-to-HDL Translator. The GF-ACG Code Synthesizer generates GF-ACG code according to the users' design specification, which includes characteristic, multiplication algorithm, modular polynomial, and logic type. Table 4.1 shows a list of characteristics, multiplication algorithms, modular polynomial degrees, logic types, and security order that can be generated by the GF-AMG system. Here, security order inidicates the order of GMS

Table 4.1 Specification supported by GF-AMG

Characteristic p	Algorithm	Degree for IP	Logic type	Security order
2	Full-tree	2–256	binary	0–5
	Mastrovito	2–256		
	Massey-Omura	2–64		
3, 5, 7, 11	Full-tree	2-256	binary p-valued logic	0

Table 4.2 Mapping of GF values onto logic values

(a) An example of $GF(2)$

$GF(2)$ value	Logic value
0	0
1	1

(b) An example of $GF(3)$

$GF(3)$ value	Logic value
0	00
1	01
2	10

described in Section 4.3. In multiplication algorithms, full-tree and Mastrovito are algorithms for PB, PRR, and RRB multipliers while Massey-Omura is an algorithm for NB-based multiplier. The GF-ACG Verifier proceeds to formally verify the generated GF-ACG code by the method described in Chapter 3. The ACG-to-HDL Translator then translates the verified GF-ACG code into the equivalent Verilog-HDL code, using the algorithm shown in Algorithm 4.1. In addition, when generating a DPA-resistant multiplier based on GMS, the GF-ACG property checker checks whether it satisfies GMS-properties using algorithms proposed in the following section. Given a GF-ACG G, we extract a set of relations of internal edges at the lowest level of abstraction from G recursively. The relations of internal edges are then translated into the corresponding HDL format by one-to-one mapping.

4.3 Automatic generation of $GF(p^m)$ multipliers

In this section, we first show design of $GF(p^m)$ arithmetic circuits by GF-ACG, which is extended in order to handle GF arithmetic circuit with a characteristic greater than two and multiple-valued logic devices (e.g., ternary gate). We then show the automatic generation of $GF(p^m)$ multipliers on the basis of the extended GF-ACG.

4.3.1 Extension of GF-ACG to $GF(p^m)$ arithmetic circuit

An extension of the GF-ACG is presented for describing a $GF(p^m)$ arithmetic circuit, enabling it to be implemented in multiple-valued logic as well as in binary logic. In the above GF-ACG, a

mapping from a GF variable to a logic variable at the lowest level description is implicitly given (i.e., 0 and 1 in $GF(2)$ are mapped into 0 and 1 in binary logic, respectively) because it focuses only on $GF(2^m)$ arithmetic circuits and their binary implementations. Therefore, such mapping is done without the need for any additional procedure. In order to describe and verify nodes with $GF(p)$ ($p \geq 2$) variables and their R-valued ($R \geq 2$) implementation, however, we need to give an explicit mapping at the lowest level of abstraction. Our plan is to provide a mapping function, called an encoding function, for transforming $GF(p)$ variables into R-valued logic variables in the form of a functional assertion (i.e., a GF equation) for the lowest-level nodes.

We first describe an encoding function for transforming $GF(p)$ variables into binary logic variables. Each GF variable in C_i (a coefficient set of degree i) is encoded by at least $\lceil \log_2 |C_i| \rceil$ logic variables. Table 4.2 gives examples of such mappings, showing encodings of (a) $GF(2) \in \{0, 1\}$ and (b) $GF(3) \in \{0, 1, 2\}$ into binary logic variables. Note that for cases having characteristic $p > 2$, any encoding is possible, including non-minimum-length encoding. Such encoding can be represented by a specific equation, referred to as an *encoding equation*. Let x and L_η ($0 \leq \eta \leq k-1$) be a GF variable over $GF(p)$ and a logic variable used for encoding, respectively. Let $\boldsymbol{\alpha} = (\alpha_0, \alpha_1, \ldots, \alpha_{k-1}) \in \{0, 1\}^k$ be a k-bit logic value. The general form of the encoding equation is then given as

$$a = \sum_{\boldsymbol{\alpha} \in \{0,1\}^k} \left(f(\boldsymbol{\alpha}) \times \Pi_{\eta=0}^{k-1} L_\eta^{\alpha_\eta} \right), \tag{4.1}$$

where $f(\boldsymbol{\alpha})$ is the GF value corresponding to $\boldsymbol{\alpha}$, and $L_\eta^{\alpha_\eta}$ is the η-th literal, defined as

$$L_\eta^{\alpha_\eta} = \begin{cases} 1 - L_\eta & (\alpha_\eta = 0) \\ L_\eta & (\alpha_\eta = 1) \end{cases}. \tag{4.2}$$

For example, the encoding equations for Tab. 4.2(a) and (b) are given as $a = L_0$ and $a = (1 - L_1)L_0 + 2L_1(1 - L_0)$, respectively.

Figure 4.2 shows GF-ACGs for a 2-input multiplier over $GF(3)$ implemented in binary logic [115], where the node in Fig. 4.2(a) corresponds to the shaded part in Fig. 4.2(b). This indicates that node n_0 has an internal structure consisting of lower-level nodes in the corresponding shaded part. Table 4.3 shows the nodes, GFs and variables used in Fig. 4.2. The nodes of n_1, n_2, and n_9 in Fig. 4.2(b) perform the mapping between GF variables and logic variables. More precisely, the functions of n_1 and n_2 are to translate GF variables into logic variables, while the function of n_9 is to translate logic variables into GF variables. Note that the functional assertions of such nodes require equation(s) that represent unused inputs. In this example, one such equation is given as $a_{L0}a_{L1} = 0$ because $(a_{L0}, a_{L1}) = (1, 1)$ is not used. Thus, any $GF(p^m)$ arithmetic circuit will be implemented by binary logic circuits in a uniform manner.

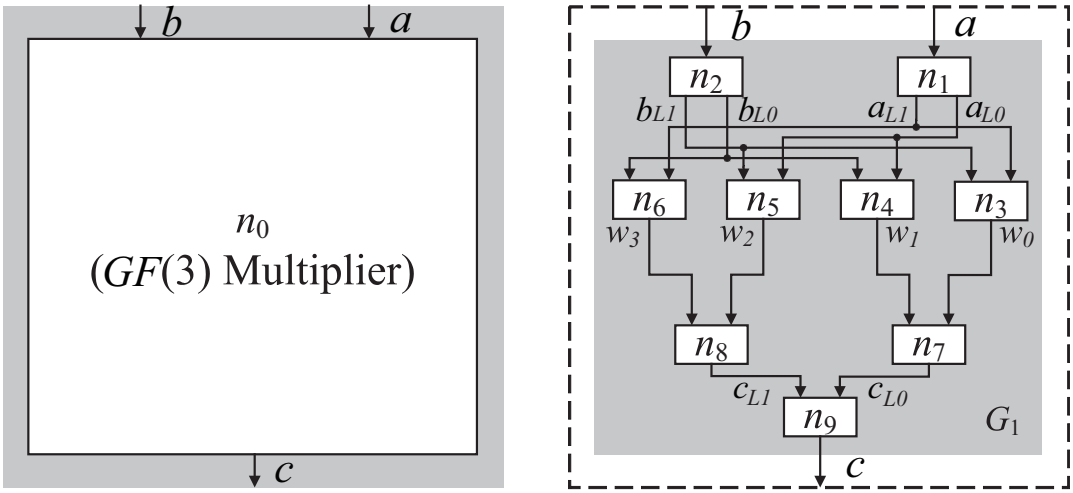


Fig. 4.2 GF-ACGs for $GF(3)$ multiplier.

Table 4.3 Nodes, GFs and GF variables in Fig. 4.2(b)

Nodes
$n_0 = (\{c = a \times b\}, G_1)$
$n_1 = (\{(1-a_{L1})a_{L0} + 2a_{L1}(1-a_{L0}) = a, a_{L0}a_{L1} = 0\}, nil)$
$n_2 = (\{(1-b_{L1})b_{L0} + 2b_{L1}(1-b_{L0}) = b, b_{L0}b_{L1} = 0\}, nil)$
$n_3 = (\{w_0 = AND(a_{L1}, b_{L1})\}, nil)$
$n_4 = (\{w_1 = AND(a_{L0}, b_{L0})\}, nil)$
$n_5 = (\{w_2 = AND(a_{L0}, b_{L1})\}, nil)$
$n_6 = (\{w_3 = AND(a_{L1}, b_{L0})\}, nil)$
$n_7 = (\{c_{L0} = OR(w_0, w_1)\}, nil)$
$n_8 = (\{c_{L1} = OR(w_2, w_3)\}, nil)$
$n_9 = (\{c = (1-c_{L1})c_{L0} + 2c_{L1}(1-c_{L0}), c_{L1}c_{L0} = 0\}, nil)$

GFs
$GF(3) = (\left(\beta^0\right), (\{0,1,2\}), nil)$
$Logic = (\left(\beta^0\right), (\{0,1\}), nil)$

GF variables	
$a = (GF(3), (0,0))$	$a_{L0}, a_{L1} = (Logic, (0,0))$
$b = (GF(3), (0,0))$	$b_{L0}, b_{L1} = (Logic, (0,0))$
$c = (GF(3), (0,0))$	$c_{L0}, c_{L1} = (Logic, (0,0))$
$w_0, w_1, w_2, w_3 = (Logic, (0,0))$	

Next, we then describe an extension of the above encoding equation for the case of R-valued implementation. Table 4.4 shows examples of the mapping of (a) GF(3) $\in \{0,1,2\}$ and (b) GF(5) $\in \{0,1,2,3,4\}$ into ternary logic. Let a and L_η $(0 \leq \eta \leq k-1)$ be a GF variable over $GF(p)$ and an R-valued logic variable used for encoding, respectively. Let $\boldsymbol{\alpha} = (\alpha_0, \alpha_1, \ldots, \alpha_{k-1}) \in$

Table 4.4 Mapping of GF values onto 3-valued logic values

(a) An example of $GF(3)$

$GF(3)$ value	3-valued logic value
0	0
1	1
2	2

(b) An example of $GF(5)$

$GF(5)$ value	3-valued logic value
0	00
1	01
2	02
3	10
4	11

$\{0, 1, \ldots, R-1\}^k$ be a k-bit R-valued logic value; the encoding equation is then given as

$$a = \sum_{\boldsymbol{\alpha} \in \{0,1,\ldots,R-1\}^k} \left(f(\boldsymbol{\alpha}) \times \Pi_{\eta=0}^{k-1} L_\eta^{\alpha_\eta} \right), \tag{4.3}$$

and $L_\eta^{\alpha_\eta}$ is represented by

$$L_\eta^{\alpha_\eta} = \prod_{\substack{l \in \{0,1,\cdots,R-1\} \\ l \neq \alpha_\eta}} \frac{L_\eta - l}{\alpha_\eta - l}, \tag{4.4}$$

where Eqs. 4.3 and 4.4 are based on arithmetic operations over $GF(p)$.

For example, the encoding equations for Tab. 4.4 (a) and (b) are given by $a = L_0$ and $a = (2L_1 + 3L_1 + 1)L_0 + 2L_1^2 + L_1$, respectively.

4.3.2 Generation of $GF(p^m)$ parallel multipliers

This section focuses on the design and generation of $GF(p^m)$ parallel multipliers in GF-AMG. For the conventional design of $GF(2^m)$ parallel multipliers also generated by GF-AMG, see [110], [58], and [85]. Let a and $b \in GF(p^m)$ be the inputs and let $c \in GF(p^m)$ be the output. The multiplication over $GF(p^m)$ is first divided into the following two functions:

$$\sum_{i=0}^{m-1} w_i = a \times b, \tag{4.5}$$

$$c = \sum_{i=0}^{m-1} w_i, \tag{4.6}$$

where $w_i \in GF(p^m)$ $(0 \leq i \leq n-1)$ is the i-th partial product. We then consider the internal structure of the nodes corresponding to Eqs. 4.5 and 4.6 to obtain its hierarchical GF-ACG description. Each w_i is given by

$$w_i = a \times b_i, \tag{4.7}$$

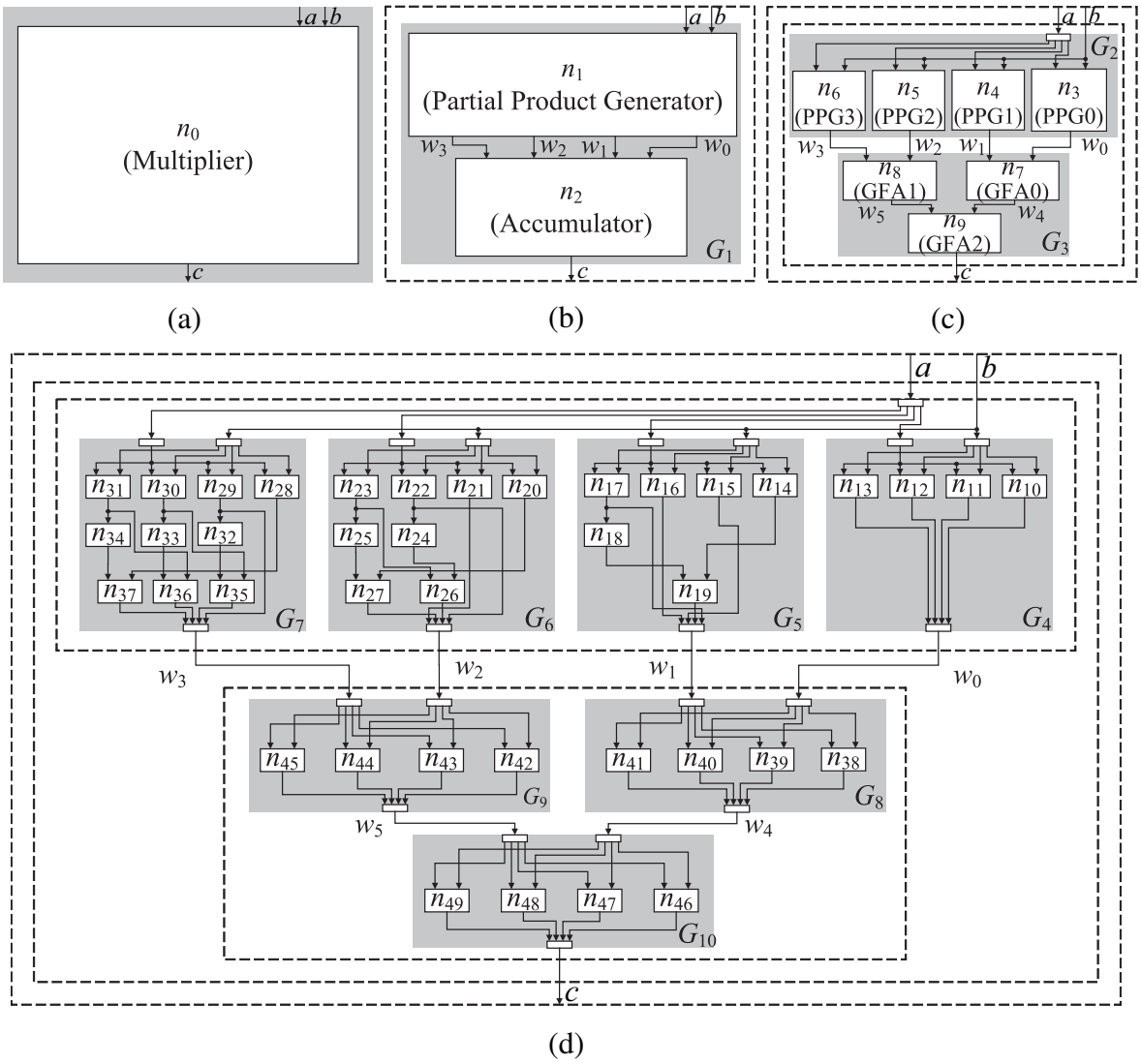


Fig. 4.3 GF-ACGs for $GF(3^4)$ parallel multipliers of (a) the top-level to (d) the 4th-level.

where b_i is the i-th element obtained by the decomposition of $b = \sum_{i=0}^{m-1} b_i$ and $a \in GF(p^m)$. This means that the internal structure of the node performing Eq. 4.5 is composed of m nodes performing Eq. 4.7. The nodes corresponding to Eq. 4.6 are composed of $m-1$ 2-input 1-output adders over $GF(p^m)$, which are given by m 2-input 1-output adders over $GF(p)$.

For example, Fig. 4.3 shows the GF-ACGs for the $GF(3^4)$ parallel multiplier at the top four levels of abstraction. Table 4.5 shows the corresponding nodes, GFs and GF variables. Note that the decomposition and composition nodes are not shown in Tab. 4.5. The nodes in Fig. 4.3 (a), (b), and (c) correspond to the shaded parts in Fig. 4.3 (b), (c), and (d), respectively. The 2nd-level nodes "Partial Product Generator" (PPG) and GF "Accumulator" (GFA) in Fig. 4.3 (b) have

Table 4.5 Nodes, GFs and GF variables in Fig. 4.3

Nodes	
[Multiplier] $n_0 = (\{c = a \times b\}, G_1)$	
[Partial Product Generator] $n_1 = (\{w_0 + w_1 + w_2 + w_3 = a \times b\}, G_2)$	
[PPG0] $n_3 = (\{w_0 = a \times b_0\}, G_4)$	
$n_{10} = (\{w_{0,0} = a_0 \times b_{0,0}\}, G_{11})$	$n_{11} = (\{w_{0,1} = a_1 \times b_{0,0}\}, G_{12})$
$n_{12} = (\{w_{0,2} = a_2 \times b_{0,0}\}, G_{13})$	$n_{13} = (\{w_{0,3} = a_3 \times b_{0,0}\}, G_{14})$
[PPG1] $n_4 = (\{w_1 = a \times b_1\}, G_5)$	
$n_{14} = (\{w_6 = a_0 \times b_{1,1}\}, G_{15})$	$n_{15} = (\{w_{1,2} = a_1 \times b_{1,1}\}, G_{16})$
$n_{16} = (\{w_{1,3} = a_2 \times b_{1,1}\}, G_{17})$	$n_{17} = (\{w_{1,0} = a_3 \times b_{1,1}\}, G_{18})$
$n_{18} = (\{w_7 = -w_{1,0}\}, G_{19})$	$n_{19} = (\{w_{1,1} = w_6 + w_7\}, G_{20})$
[PPG2] $n_5 = (\{w_2 = a \times b_2\}, G_6)$	
$n_{20} = (\{w_8 = a_0 \times b_{2,2}\}, G_{21})$	$n_{21} = (\{w_{2,3} = a_1 \times b_{2,2}\}, G_{22})$
$n_{22} = (\{w_{2,0} = a_2 \times b_{2,2}\}, G_{23})$	$n_{23} = (\{w_9 = a_3 \times b_{2,2}\}, G_{24})$
$n_{24} = (\{w_{10} = -w_{2,0}\}, G_{25})$	$n_{25} = (\{w_{11} = -w_9\}, G_{26})$
$n_{26} = (\{w_{2,1} = w_{10} + w_9\}, G_{27})$	$n_{27} = (\{w_{2,2} = w_8 + w_{11}\}, G_{28})$
[PPG3] $n_6 = (\{w_3 = a \times b_3\}, G_7)$	
$n_{28} = (\{w_{12} = a_0 \times b_{3,3}\}, G_{29})$	$n_{29} = (\{w_{3,0} = a_1 \times b_{3,3}\}, G_{30})$
$n_{30} = (\{w_{13} = a_2 \times b_{3,3}\}, G_{31})$	$n_{31} = (\{w_{14} = a_3 \times b_{3,3}\}, G_{32})$
$n_{32} = (\{w_{15} = -w_{3,0}\}, G_{33})$	$n_{33} = (\{w_{16} = -w_{13}\}, G_{34})$
$n_{34} = (\{w_{17} = -w_{14}\}, G_{35})$	$n_{35} = (\{w_{3,1} = w_{13} + w_{15}\}, G_{36})$
$n_{36} = (\{w_{3,2} = w_{14} + w_{16}\}, G_{37})$	$n_{37} = (\{w_{3,3} = w_{12} + w_{17}\}, G_{38})$
[Accumulator] $n_2 = (\{c = w_0 + w_1 + w_2 + w_3\}, G_3)$	
[GFA0] $n_7 = (\{w_4 = w_0 + w_1\}, G_8)$	
$n_{38+i} = (\{w_{4,i} = w_{0,i} + w_{1,i}\}, G_{39+i})$	
[GFA1] $n_8 = (\{w_5 = w_2 + w_3\}, G_9)$	
$n_{42+i} = (\{w_{5,i} = w_{2,i} + w_{3,i}\}, G_{43+i})$	
[GFA2] $n_9 = (\{c = w_4 + w_5\}, G_{10})$	
$n_{46+i} = (\{c_i = w_{4,i} + w_{5,i}\}, G_{47+i})$	

GFs
$GF(3^4) = (\left(\beta^3, \beta^2, \beta^1, \beta^0\right), (\{0,1,2\}, \{0,1,2\}, \{0,1,2\}, \{0,1,2\}), \beta^4 + \beta + 2)$
$GF(3) = (\left(\beta^0\right), (\{0,1,2\}), nil)$

GF variables	
$a, b, c = (GF(3^4), (3,0))$	$a_i, b_i, c_i = (GF(3), (0,0)),\ (0 \le i \le 3)$
$b_{i,i} = (GF(3), (0,0))$	$w_\varsigma = (GF(3^4), (3,0)),\ (0 \le \varsigma \le 5)$
$w_{\varsigma,i} = (GF(3), (0,0)),\ (0 \le \varsigma \le 5)$	

functional assertions corresponding to Eqs. 4.5 and 4.6, respectively. The 3rd-level nodes "PPGi" in Fig. 4.3 (c) have the functional assertion corresponding to Eq. 4.7. The nodes "GFAi" in Fig. 4.3 (c) indicate 2-input 1-output adders over $GF(3^4)$ to construct "Accumulator". In addition, the nodes in Fig. 4.3 (d) indicate $GF(3)$ arithmetic circuits, and these are described as given in Fig. 4.2, which showed the binary implementation case. Thus, we have the GF-ACGs for the $GF(p^m)$ parallel multiplier represented in a hierarchical manner.

Algorithm 4.2 displays an algorithm for synthesizing $GF(p^m)$ multipliers. Given a design specification (i.e., an irreducible polynomial and an implementation logic), the algorithm generates a GF-ACG. The function "Degree" in Line 2 obtains the degree of the irreducible

Algorithm 4.2 Synthesize GF-ACG

Input: Irreducible Polynomial IP, Logic L
Output: GF-ACG $G = (\boldsymbol{N}, \boldsymbol{E})$

```
1: function FULLTREE(IP, Logic)
2:     int m ← Degree(IP); str eq;
3:     for i = 0 to m − 1 do
4:         int eq ← GetEquation(i, IP);
5:         int l ← CountSubOperator(eq);
6:         int m ← CountAddOperator(eq);
7:         for η = 0 to m − 1 do
8:             G_{m+η} ← GenerateGFpMultiplier(L);                ▷ 4th-level
9:         end for
10:        for η = 0 to l − 1 do
11:            G_{2m+η} ← GenerateGFpAdditiveInv(L);              ▷ 4th-level
12:        end for
13:        for η = 0 to m − 1 do
14:            G_{2m+l+η} ← GenerateGFpAdder(L);                  ▷ 4th-level
15:        end for
16:        G_i ← GeneratePPGi(eq, G_m, ..., G_{2m+l+m−1});        ▷ 3rd-level
17:    end for
18:    G' ← GeneratePPG(G_0, ..., G_{d−1});                       ▷ 2nd-level
19:    for i = 0 to m − 2 do
20:        for η = 0 to m − 1 do
21:            G_{m+η−1} ← GenerateGFpAdder(L);                   ▷ 4th-level
22:        end for
23:        G_i = GenerateGFAi(G_{m−1}, ..., G_{2m−2});            ▷ 3rd-level
24:    end for
25:    G'' = GenerateACC(G_0, ..., G_{m−2});                      ▷ 2nd-level
26:    G = GenerateMultiplier(G', G'');                           ▷ top-level
27:    return G
28: end function
```

polynomial. According to the value obtained, its internal structure is generated in a recursive manner. The function "GetEquation" in Line 4 obtains an equation of the "PPGi" expressions that are represented in Eq. 4.7. The functions "CountSubOperator" and "CountAddOperator" count the numbers of "-" and "+" operators in the equation, respectively. Using the above numbers and the degree, we generate 4th-level GF-ACGs for $GF(p)$ arithmetic circuits. The functions "GenerateGFpMultiplier","GenerateGFpAdditiveInv", and "GenerateGFpAdder" return GF-ACGs for $GF(p)$ multipliers, additive inverters, and adders, respectively. Their internal structures are determined by the given logic L. If L is binary logic, the internal structure is given by the netlist corresponding to $GF(p)$ multiplier, additive inverter, and adder which are

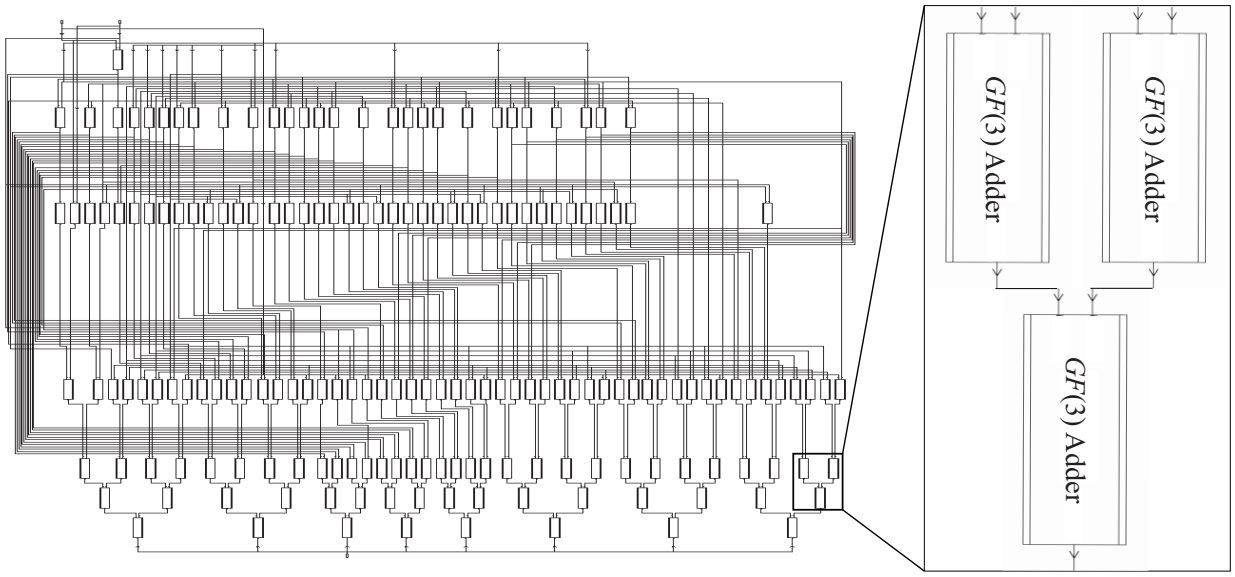


Fig. 4.4 Schematic of $GF(3^8)$ multiplier obtained from GF-AMG.

designed in a manner similar to Fig. 2. If L is not binary logic, the internal structure is given as nil in order for designers to use custom logic cells designed by themselves. The function "GeneratePPGi" in Line 16 generates "PPGi" expressions $(0 \leq i \leq m-1)$ from the 4th-level GF-ACGs, where the $GF(p)$ adders are placed as a tree. The function "GeneratePPG" in Line 18 generates a GF-ACG for "Partial Product Generator" from the 3rd-level GF-ACGs of "PPGi". Similarly, "Accumulator" is generated from the 3rd-level GF-ACGs of "GFAi" consisting of m GF-ACGs of $GF(p)$ adders. In "Accumulator", $m-1$ "GFAi" expressions are placed as a tree. Finally, the function "GenerateMultiplier" in Line 26 generates a GF-ACG for the $GF(p^m)$ multiplier from the 2nd-level GF-ACG.

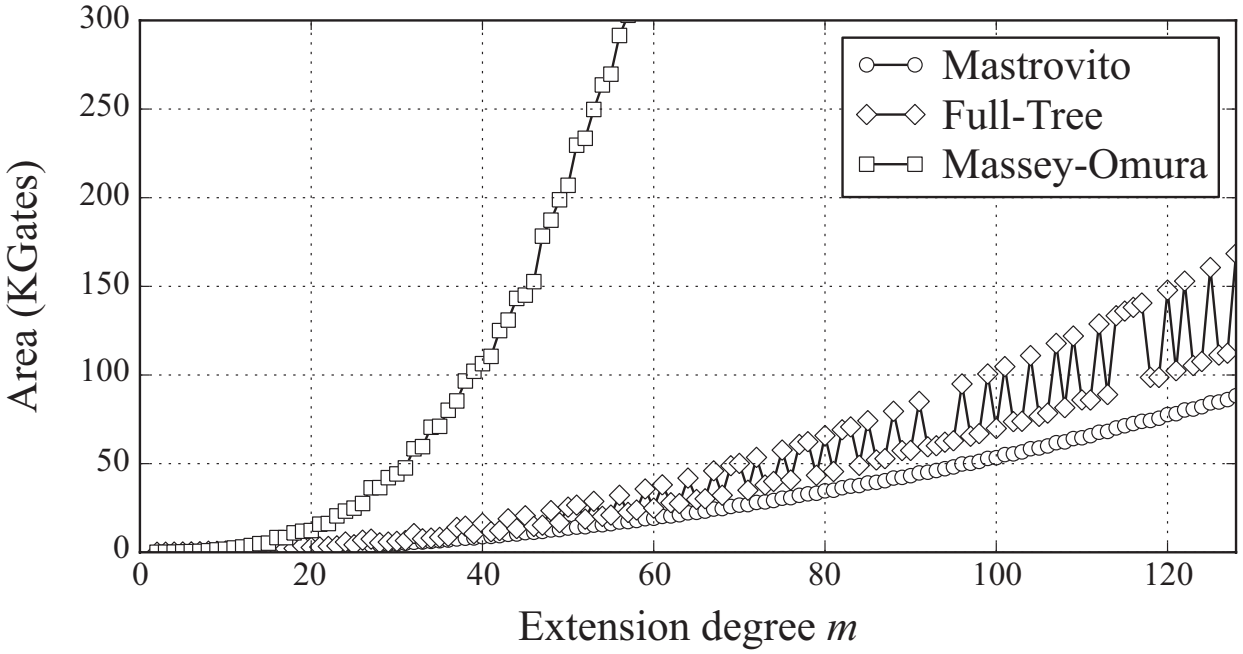
The HDL code generated for $GF(p^m)$ multipliers may be applied in not only a binary implementation but also an L-valued implementation. ($GF(2^m)$ multipliers, by contrast, are implemented only in binary logic.) For a binary implementation, we can apply the HDL code to the standard back-end design flow including logic synthesis and placement and routing (P&R) with the standard cell library. For an L-valued implementation, we would implement the HDL code by a technology mapping with a custom-made library in an L-valued logic. Thus, we see that GF-AMG generates verified HDL codes for both multiple-valued logic and binary logic.

As an example, Fig. 4.4 shows a schematic of a $GF(3^8)$ multiplier generated by GF-AMG, where the lowest level component indicates an arithmetic circuit over $GF(3)$. We can implement this multiplier in ternary logic by applying a ternary logic circuit to the component.

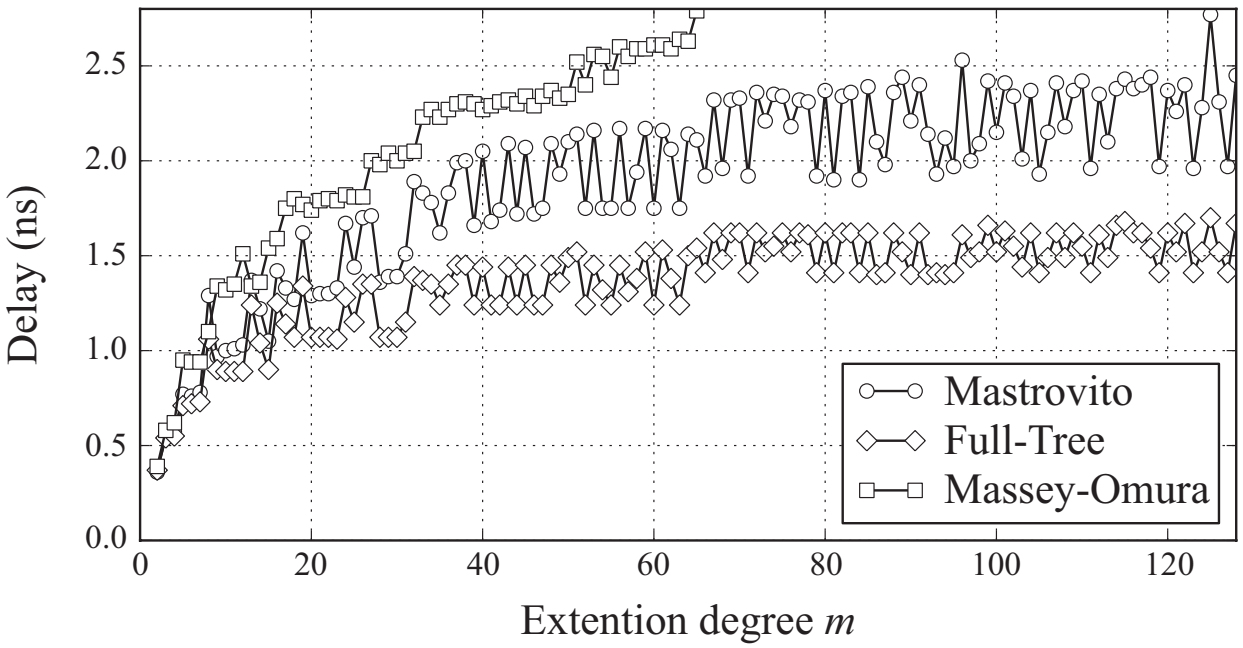
Table 4.6 Generation times of multipliers over $GF(p^m)$ (s)

Extended degree m	8					16					32				
Characteristic p	2	3	5	7	11	2	3	5	7	11	2	3	5	7	11
GF-ACG synthesis	0.07	0.07	0.07	0.07	0.07	0.08	0.08	0.08	0.09	0.09	0.12	0.14	0.14	0.15	0.15
Formal verification	2.59	2.78	2.77	2.78	2.77	4.06	4.26	4.24	4.25	4.24	7.40	7.70	7.62	7.70	7.66
ACG-to-HDL	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.02	0.02	0.10	0.11	0.11	0.11	0.11
Total	2.66	2.86	2.84	2.86	2.85	4.17	4.37	4.35	4.36	4.36	7.62	7.95	7.87	7.96	7.92

Extended degree m	64					128					256				
Characteristic p	2	3	5	7	11	2	3	5	7	11	2	3	5	7	11
GF-ACG synthesis	0.29	0.35	0.44	0.37	0.39	0.99	1.23	1.30	1.33	1.39	3.90	4.99	5.18	5.32	5.58
Formal verification	16.39	17.26	18.37	17.25	17.26	48.25	54.14	54.09	54.08	54.22	235.62	289.79	293.42	292.79	292.34
ACG-to-HDL	0.58	0.66	0.81	0.66	0.67	3.63	4.63	4.33	4.56	4.40	27.57	33.05	32.88	33.34	33.33
Total	17.26	18.27	19.62	18.28	18.32	52.87	60.00	59.72	59.97	60.01	267.09	327.83	331.48	331.45	331.24



(a) Area.



(b) Delay.

Fig. 4.5 Comparison of three types of $GF(2^m)$ multiplier for different extension degrees.

4.3.3 Experimental generation

The performance of our system was evaluated through the experimental generation of $GF(p^m)$ parallel multipliers. We first generated a set of $GF(p^m)$ parallel multipliers of typical degrees.

Table 4.7 Performance of $GF(p^m)$ multipliers for different characteristics and degrees

	Area (KGates)						Delay (ns)					
m	4	8	16	32	64	128	4	8	16	32	64	128
$p = 3$	0.6	2.3	9.7	39.3	158.1	685.8	1.48	2.36	2.81	3.25	3.68	4.13
$p = 5$	2.22	9.67	40.28	164.34	663.78	2,667.98	2.83	3.74	4.63	5.54	6.43	7.34
$p = 7$	5.26	23.02	96.71	399.90	1,572.58	N/A	5.28	6.58	9.09	9.24	11.78	N/A
$p = 11$	14.06	61.99	259.32	1,043.41	4,247.72	N/A	9.95	12.44	17.09	19.43	21.88	N/A

The generation was conducted with the same experimental setup as the above.

Table 4.6 shows the generation times, consisting of GF-ACG synthesis, verification, and GF-ACG-to-HDL translation times, for each of the degrees investigated. Using our method, we achieved complete verification even for a 1024-bit multiplier over $GF(11^{256})$. Note here that the verification time decreases even in the case of a larger p because the computation time of algebraic operation with the software used in the experiment is sometimes dependent on the machine condition such as parallel-executed processes. As a comparison to evaluate the advantage of the verifier, we also performed the Verilog-XL simulation using the corresponding HDL descriptions. With this method, we were not able to complete the simulation of $GF(3^{10})$ or larger multipliers because the simulation time increases exponentially as the extension degree increases. As described above, GFs with at most characteristic seven are used for pairing-based cryptography so far. Thus, the experimental result suggests that our system is sufficient and available for such applications.

We then generated a set of $GF(2^m)$ parallel multipliers for three types of multiplication algorithm in order to assess the performance variation. The performance was evaluated with the Synopsys Design Compiler and the TSMC 65-nm cell library.

Figure 4.5 shows the area and delay of the three types of $GF(2^m)$ multiplier for different value of m, where the vertical axis indicates the (a) area or (b) delay, and the horizontal axis indicates the extension degree. We can confirm here that Mastrovito and Full-Tree have the advantage in area and delay, respectively. Massey-Omura, which is a typical multiplication algorithm using a normal basis (NB), did not demonstrate any advantage in area or delay over the other two algorithms. However, Massey-Omura is useful for more sophisticated arithmetic NB circuits; for example, we can design efficient exponential circuits based on an NB since the squaring operation is performed only by wiring.

Table 4.7 shows the performance of $GF(p^m)$ multipliers implemented in a binary logic, for different characteristics and degrees. For $GF(p^m)$ multipliers with $p = 5$, 7, and 11, we implemented $GF(p)$ arithmetic circuits (i.e., adder, multiplier, and constant multipliers over $GF(p)$) using the corresponding lookup-table. The Synopsys Design Compiler could not synthesize the

$GF(7^{128})$ and $GF(11^{128})$ multipliers under our experimental condition due to the memory overflow. This would be because the circuit area (i.e., the number of logic gates) for the multipliers are too large ($\sim$ 6 M gates). However, the results suggest that designers can generate a variety of practical GF multipliers from given design specifications by the proposed GF-AMG system.

4.4 Automatic generation of DPA-resistant $GF(2^m)$ multipliers based on GMS

GMS is a state-of-the-art masking-based countermeasure against higher-order DPAs [122]. A dth-order GMS can describe any kind of arithmetic circuits over $GF(2^m)$ resistant to dth-order DPAs including advanced ones that utilize glitch induced in power consumption [83, 92]. In this section, we first describe the attack model considered in this thesis called dth order probing model, and describe three security properties of GMS. Then, we show a construction of GMS-based circuits using GF parallel multipliers as examples.

4.4.1 Attack model

GMS considers a modified dth-order probing model [65] to attackers' advantage. It was proven that the dth probing model is essentially equivalent to dth-order DPA model [48] while the probing model is useful for discussing countermeasures against DPAs using glitch.

In this model, the attacker can probe d wires in the circuit within a time window. The values of the probed wires is determined by the combinational logic circuit performing the GF arithmetic function before the wire. This indicates that the attacker obtain the value of synchronous points (i.e., registers) connected to the wires. For example, Fig. 4.6 illustrates the circuit for describing the first-order probing model. The attacker would probe the wire g (i.e., output of the function G) to obtain the values of registers $f'_0, f'_1, \ldots$, and $f'_{\sigma-1}$ (i.e., all input of G). In other words, the attacker can retrieve all intermediate values to calculate F'. On the other hand, the attacker cannot obtain the values of reg1 and intermediate values in the functions $F_0, F_1, \ldots$, and $F_{\sigma-1}$ by probing f. Though we describe the first-order probing here, the dth probing model where the attacker probes d wires is much alike.

4.4.2 GMS properties

The working principle of masking schemes including GMS is to represent a secret value $a \in GF(2^m)$ by $a_0 + a_1 + \cdots + a_\varphi + \cdots + a_{\sigma-1}$, where $a_\varphi \in GF(2^m)$ $(0 \leq \varphi \leq \sigma - 1)$ is initially given by a random mask. Each element a_φ is called a share. Let a be the input and $a_0, a_1, \ldots, a_\varphi, \ldots, a_{\sigma-1}$ be the share of a, where σ is the number of input share. Let $r \in GF(2^m)$

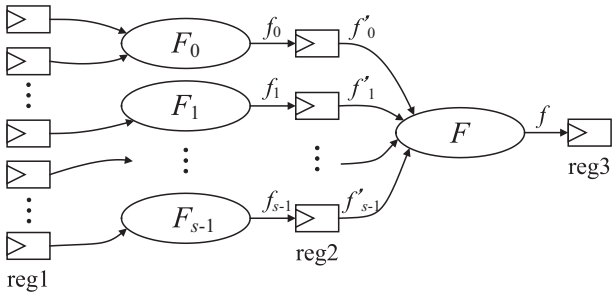


Fig. 4.6 Example of dth-order probing model.

be the output of combinational circuit and $r_0, r_1, \ldots, r_{\varphi'}, \ldots, r_{\sigma'-1}$ be the share of r, where $r_\varphi \in GF(2^m)$ $(0 \leq \varphi' \leq \sigma' - 1)$ denotes the φ'th share, and σ' is the number of output shares.

(i) *Correctness*: the first property implies that the sum of shares is equal to the secret value at the input and output of the circuit, namely, $a = a_0 + a_1 + \cdots + a_{\sigma-1}$ and $r = r_0 + r_1 + \cdots + r_{\sigma'-1}$. This property indicates that the shared circuit correctly performs the original (i.e., nonshared) function.

(ii) *dth-order noncompleteness*: the second property implies that any d output shares are independent of at least one input share. The number of input shares (i.e., σ) required to meet the dth-order noncompleteness is dependent on d and the degree of the circuit function. Typically, σ and σ' are respectively given by $dt + 1$ and $\binom{\sigma}{t}$, where t is the degree.

(iii) *Uniformity*: the third property indicates that the input and output values are uniformly distributed.

Correctness and dth-order noncompleteness can be realized for any GF function, while some functions cannot satisfy uniformity under the constraints of two above properties. However, the uniformity criterion can be satisfied by the addition of fresh mask(s) to the non-uniform outputs [103].

4.4.3 Construction of GMS-based circuit

As an example, we describe the construction of GMS-based $GF(2^m)$ multipliers. The number of input shares σ is given by $2d+1$ because the degree of a multiplication is two (the GMS-based two-operand multiplier has 2σ inputs in total). The number of output shares is given by $\sigma' = \binom{\sigma}{2}$ $(= d(2d+1))$. Figure 4.7 shows the (a) first- and (b) second-order GMS-based multipliers over $GF(2^m)$, where a_φ and $b_{\hat{\varphi}}$ $(0 \leq \hat{\varphi} \leq \sigma - 1)$ are input shares and c_φ is output share of the multiplier. When $m \geq 2$, AND and XOR gates in Fig. 4.7 denote multiplier and adder over $GF(2^m)$, respectively. Hence, each GMS-based multiplier in Fig. 4.7 performs $c = a \times b$ where $a = \sum_\varphi a_i$, $b = \sum_{\hat{\varphi}} b_{\hat{\varphi}}$, and $c = \sum_\varphi c_\varphi$. The GMS-based multipliers are composed of nonlinear,

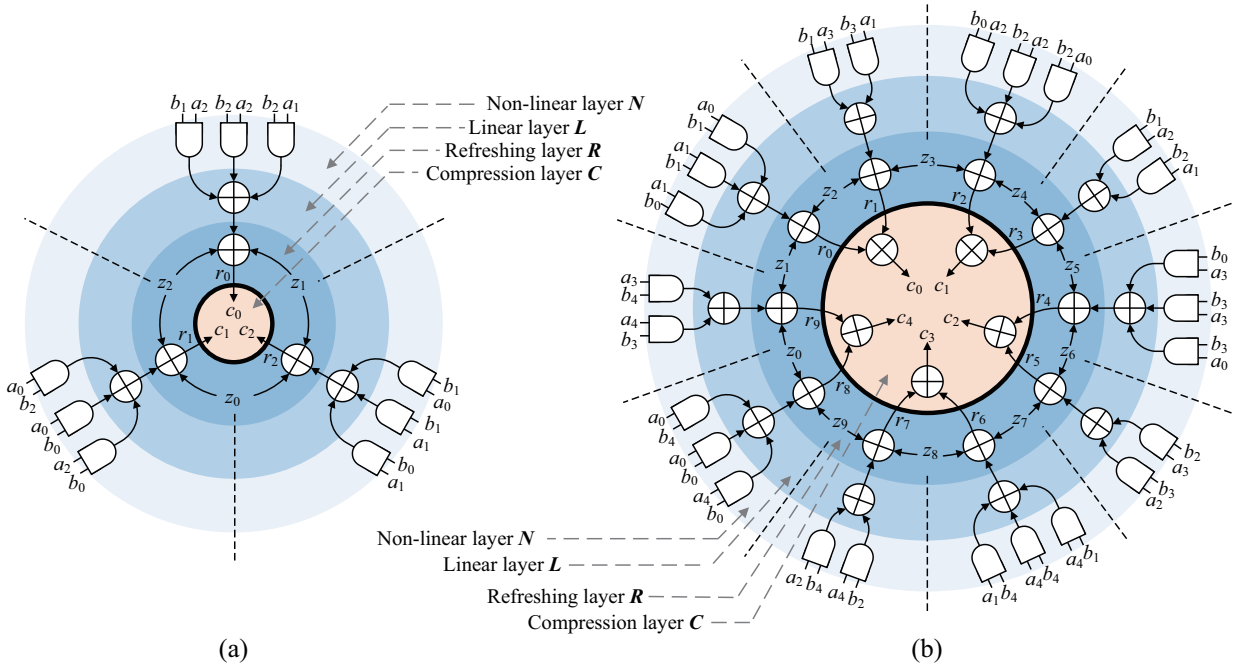


Fig. 4.7 GMS-based $GF(2^m)$ multipliers in [122]: (a) first- and (b) second-order.

linear, refreshing, and compression layers and are denoted by $\boldsymbol{N}$, $\boldsymbol{L}$, $\boldsymbol{R}$, and $\boldsymbol{C}$, respectively. Note that the bold line separating $\boldsymbol{R}$ and $\boldsymbol{C}$ denotes registers and σ' indicates the number of output shares for $\boldsymbol{R}$ (i.e., $r_{\varphi'}$).

The nonlinear layer $\boldsymbol{N}$ computes $a_{\varphi}b_{\hat{\varphi}}$ ($0 \leq \varphi \leq \sigma - 1$ and $0 \leq \hat{\varphi} \leq \sigma - 1$) using multipliers over $GF(2^m)$. The number of inputs and outputs for $\boldsymbol{N}$ is given by 2σ and σ^2, respectively.

The linear layer $\boldsymbol{L}$ consists of adders to reduce the number of elements from σ^2 to σ'. The adders should be constructed to satisfy dth-order noncompleteness. In other words, any d outputs of $\boldsymbol{L}$ are independent of at least one input of $\boldsymbol{N}$.

The refreshing layer $\boldsymbol{R}$ adds fresh masks to the outputs of $\boldsymbol{L}$ to meet uniformity[*1]. The ring-shaped addition shown in Fig. 4.7 makes it possible to add fresh masks while maintaining correctness and noncompleteness.

The compression layer $\boldsymbol{C}$ consists of adders to reduce the number of shares from σ' to σ. Note that, if $\sigma' = \sigma$, the $\boldsymbol{C}$ layer is composed of only wiring as Fig. 4.7(a). To satisfy dth-order noncompleteness, the registers are located at the boundary between $\boldsymbol{R}$ and $\boldsymbol{C}$.

We then design a DPA-resistant multiplier based on the dth-order GMS using GF-ACG. Figure

[*1] In the first-order GMS, $\boldsymbol{R}$ can be omitted if the outputs of $\boldsymbol{L}$ meet the required uniformity criterion. However, since it is known that two-input multiplication has no construction to satisfy the uniformity [103], $\boldsymbol{R}$ is always required for first-order GMS multipliers, as shown in Fig. 4.7(a). In other words, the uniformity is satisfied by $\boldsymbol{R}$ in the first-order GMS.

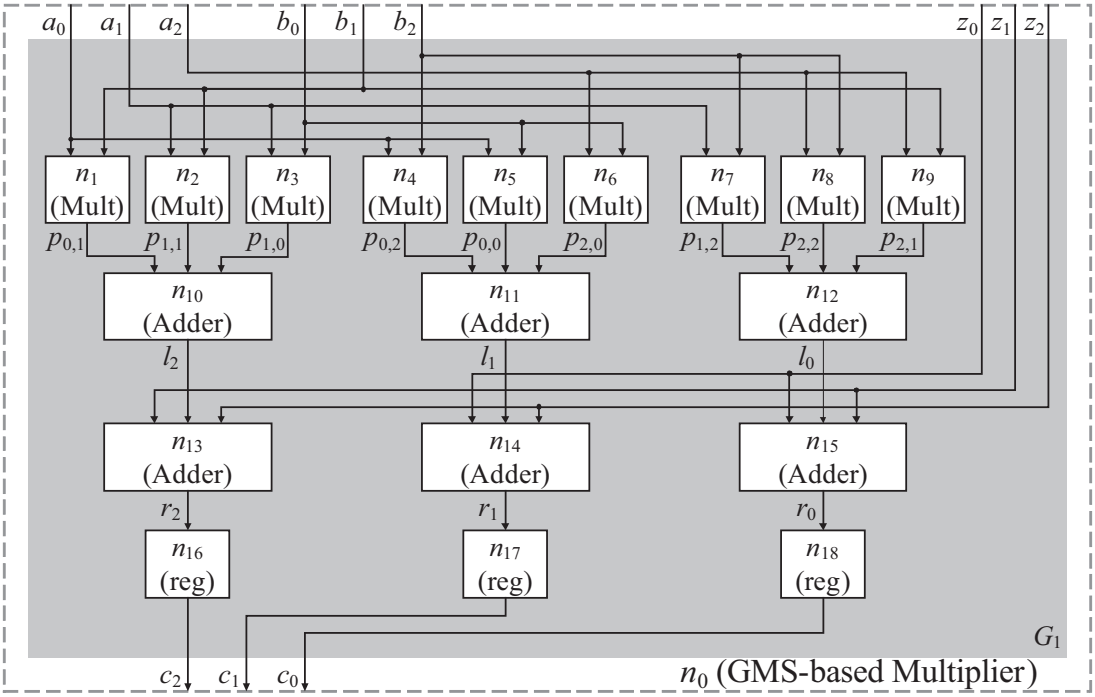


Fig. 4.8 GF-ACG for GMS-based GF parallel multiplier: top- and second-levels of abstractions.

4.8 shows a GF-ACG for the GMS-based $GF(2^m)$ multiplier with $d = 1$ and an irreducible polynomial IP, which corresponds to the multiplier of Fig. 4.7(a). Table 4.8 shows nodes, GF, and variables in Fig. 4.8. Note here that the subscripts of p and z are members of $\mathbb{Z}/\sigma\mathbb{Z}$ and $\mathbb{Z}/\sigma'\mathbb{Z}$, respectively. The variables a_i, b_j, and c_i denote the shares of input and output a, b, and c, respectively (i.e., $a = \sum_{\varphi} a_{\varphi}, b = \sum_{\hat{\varphi}} b_{\hat{\varphi}}$, and $c = \sum_{\varphi} c_{\varphi}$ and the multiplier performs $c = a \times b$). Nodes $n_1, n_2, \ldots$, and n_9 denote the multipliers in the $\boldsymbol{N}$. (Note that we can use any type of multipliers for them including ones shown in this dissertation.) Nodes n_{10}, n_{11}, and n_{12} denote the adders in the $\boldsymbol{L}$, which are designed to satisfy noncompleteness. Nodes n_{13}, n_{14}, and n_{15} denote the adders in the $\boldsymbol{R}$, where fresh masks z_0, z_1, and z_2 are added to l_0, l_1, and l_2. After the $\boldsymbol{R}$, output shares $r_{\varphi'}$ for the $\boldsymbol{R}$ are stored in registers n_{16}, n_{17}, and n_{18}. Finally, the register outputs are added together to reduce the number of shares from σ' to σ. In Fig. 4.8, we do not perform addition in the $\boldsymbol{C}$ because $\sigma = \sigma' = 3$. Note that higher-order GMS-based multipliers can be also described in the same manner.

Algorithm 4.3 synthesizes GF-ACG to represent a GMS-based multiplier from an irreducible polynomial IP and a GMS-order d. Function "DegreeOf" in Line 2 obtain the extension degree m from IP. In Line 3, we calculate the numbers of input and output shares (i.e., s and

Table 4.8 Nodes, GFs, and variables in Fig. 4.8

Nodes
[GMS-based Multiplier]
$n_0 = (\{\mathcal{X}(c_0 = a_1b_2 + a_2b_2 + a_2b_1 + z_1 + z_2),$ $\mathcal{X}(c_1 = a_0b_2 + a_0b_0 + a_2b_0 + z_0 + z_2),$ $\mathcal{X}(c_2 = a_0b_1 + a_1b_1 + a_1b_0 + z_0 + z_1)\}, G_1)$
[Mult]
$n_1 = (\{\mathcal{X}(p_{0,1} = a_0b_1)\}, G_2)$ $\quad n_2 = (\{\mathcal{X}(p_{1,1} = a_1b_1)\}, G_3)$
$n_3 = (\{\mathcal{X}(p_{1,0} = a_1b_0)\}, G_4)$ $\quad n_4 = (\{\mathcal{X}(p_{0,2} = a_0b_2)\}, G_5)$
$n_5 = (\{\mathcal{X}(p_{0,0} = a_0b_0)\}, G_6)$ $\quad n_6 = (\{\mathcal{X}(p_{2,0} = a_2b_0)\}, G_7)$
$n_7 = (\{\mathcal{X}(p_{1,2} = a_1b_2)\}, G_8)$ $\quad n_8 = (\{\mathcal{X}(p_{2,2} = a_1b_1)\}, G_9)$
$n_9 = (\{\mathcal{X}(p_{2,1} = a_2b_1)\}, G_{10})$
[Adder]
$n_{10} = (\{\mathcal{X}(l_2 = p_{0,1} + p_{1,1} + p_{1,0})\}, G_{11})$
$n_{11} = (\{\mathcal{X}(l_1 = p_{0,2} + p_{0,0} + p_{2,0})\}, G_{12})$
$n_{12} = (\{\mathcal{X}(l_0 = p_{1,2} + p_{2,2} + p_{2,1})\}, G_{13})$
$n_{13} = (\{\mathcal{X}(r_0 = l_2 + z_1 + z_2)\}, G_{14})$
$n_{14} = (\{\mathcal{X}(r_1 = l_1 + z_0 + z_2)\}, G_{15})$
$n_{15} = (\{\mathcal{X}(r_2 = l_0 + z_0 + z_1)\}, G_{16})$
[reg]
$n_{16} = (\{\mathcal{X}(\mathcal{N}c_0 = r_2)\}, G_{17})$
$n_{17} = (\{\mathcal{X}(\mathcal{N}c_1 = r_1)\}, G_{18})$
$n_{18} = (\{\mathcal{X}(\mathcal{N}c_2 = r_0)\}, G_{19})$
GF
$GF(2^m) = (\left(\beta^{m-1}, \beta^{m-2}, \ldots, \beta^0\right), (\{0,1\}, \{0,1\}, \ldots, \{0,1\}), IP)$
GF variables
$a_\varphi, b_{\hat{\varphi}}, p_{\varphi,\hat{\varphi}}, c_{\varphi'}, l_{\varphi'}, r_{\varphi'}, z_{\varphi'} = (GF(2^m), (m-1, 0))$

s', respectively). In Lines 4–8, we generate GF-ACGs for submultipliers in the $\boldsymbol{N}$. Function "GenSubMult" obtains a GF-ACG $G^{(N)}_{\sigma\varphi+\hat{\varphi}}$ for a submultiplier to calculate a partial product $a_\varphi b_{\hat{\varphi}}$. This function can be implemented using Algorithm 4.2. In Lines 9–19, we generate adders in the $\boldsymbol{L}$. Functions "GenAdderL0(m, o_0, o_1)" and "GenAdderL1(m, x, y)" obtain GF-ACGs $G^{(L)}_k$ for adders that calculate $a_{o_0}b_{o_1} + a_{o_0}b_{o_0} + a_{o_1}b_{o_0}$ and $a_{o_0}b_{o_1} + a_{o_1}b_{o_0}$, respectively, where o_0 and o_1 are integers. An adder in the $\boldsymbol{L}$ has two input shares of a and b. Therefore, the d outputs are dependent on at most $2d$ input shares, which indicates that the $\boldsymbol{L}$ would be satisfied with the dth-order noncompleteness. The GF adders are easily synthesized because they consist of bit-parallel XORs. In Lines 20–22, we generate adders in the $\boldsymbol{R}$ and registers for the outputs of the $\boldsymbol{R}$. Function "GenAdderR" obtains a GF-ACG $G^{(R)}_\varphi$ for adders to compute $r_{\varphi'} = l_{\varphi'} + z_{\varphi'+1} + z_{\varphi'+2}$, and a register to store $r_{\varphi'}$. Lines 23–25 generate adders in the $\boldsymbol{C}$. Function "GenAdderC" obtains a GF-ACG $G^{(C)}_\varphi$ for adders to compute $c_\varphi = \sum_{e=0}^{d-1} r_{\varphi d+e}$. Lastly, function "GenGMSMult" in Line 26 generates a GF-ACG G_0 for the GMS-based multiplier, where $G^{(\boldsymbol{N})}$, $G^{(\boldsymbol{L})}$, $G^{(\boldsymbol{R})}$, and $G^{(\boldsymbol{C})}$ denote the sets of all $G^{(N)}_{\sigma\varphi+\hat{\varphi}}$, $G^{(L)}_{\varphi'}$, $G^{(R)}_{\varphi'}$, and $G^{(C)}_\varphi$, respectively.

Algorithm 4.3 GF-ACG synthesis

Input: Irreducible polynomial IP, GMS-order d
Output: GF-ACG $G_0 = (\boldsymbol{N_0}, \boldsymbol{E_0})$
function GF-ACGSYNTHESIS(IP, d)
  int $m \leftarrow$ DegreeOf(IP);
  int $\sigma \leftarrow 2d+1$; **int** $\sigma' \leftarrow d(2d+1)$;
  for φ from 0 to $\sigma - 1$ **do**
    for $\hat{\varphi}$ from 0 to $\sigma - 1$ **do**
      $G^{(N)}_{\sigma\varphi+\hat{\varphi}} \leftarrow$ GenSubMult(m, φ, φ', IP);
    end for
  end for
  for φ from 1 to $\sigma - 2$ **do**
    $G^{(L)}_{\sigma\varphi} \leftarrow$ GenAdderL0($m, \varphi, 0$);
    for $\hat{\varphi}$ from 1 to $\varphi - 1$ **do**
      $G^{(L)}_{\sigma\varphi+\hat{\varphi}} \leftarrow$ GenAdderL1($m, \varphi, \hat{\varphi}$);
    end for
  end for
  $G^{(L)}_{\sigma'-\sigma} \leftarrow$ GenAdderL0($m, 0, \sigma - 1$);
  $G^{(L)}_{\sigma'-\sigma+1} \leftarrow$ GenAdderL0($m, \sigma - 1, 1$);
  for $\hat{\varphi}$ from 2 to $\sigma - 2$ **do**
    $G^{(L)}_{\sigma'-\sigma+\hat{\varphi}+1} \leftarrow$ GenAdderL1($m, \sigma - 1, \hat{\varphi}$);
  end for
  for φ' from 0 to $\sigma' - 1$ **do**
    $G^{(R)}_{\varphi'} \leftarrow$ GenAdderR(m, φ');
  end for
  for φ from 0 to $\sigma - 1$ **do**
    $G^{(C)}_{\varphi} \leftarrow$ GenAdderC(m, φ);
  end for
  $G_0 \leftarrow$ GenGMSMult($G^{(\boldsymbol{N})}, G^{(\boldsymbol{L})}, G^{(\boldsymbol{R})}, G^{(\boldsymbol{C})}$);
  return G_0;
end function

Algorithm 4.4 Correctness checking for multiplier

Input: A GF-ACG $G_0 = (\boldsymbol{N}_0, \boldsymbol{E}_0)$
Output: Verifcation result $res \in \{\text{true}, \text{false}\}$
function CHECKCORRECTNESS($n_0 = (\boldsymbol{F}_0, G_1) \in \boldsymbol{N}$)
  Bool $result \leftarrow$ true;
  set $\boldsymbol{M} \leftarrow \{c - ab\}$;
  set $\boldsymbol{T} \leftarrow$ MaskingRelation($\boldsymbol{E}_0$);
  set $\boldsymbol{G} \leftarrow \boldsymbol{F}_0 \cup \boldsymbol{T}$;
  set $\boldsymbol{GB} \leftarrow$ GröbnerBasisOf($\boldsymbol{G}$);
  $res \leftarrow res$ & IsIdealMember($\boldsymbol{M}, \boldsymbol{GB}$);
  return res;
end function

4.4.4 Functional verification and GMS property checking

The functionality of synthesized GF-ACG code is verified by algorithms in Chapter 3. In addition, in this dissertation, we present a novel formal method for checking whether the GF-ACG

Algorithm 4.5 Noncompleteness checking

Input: A GF-ACG $G_1 = (\boldsymbol{N}_1, \boldsymbol{E}_1)$, Order d
Output: Veirifcation $res \in \{\text{true}, \text{false}\}$

```
1: function CHECKNONCOMPLETENESS(G_1)
2:     set U ← ∅; set S ← ∅;
3:     int σ' ← d(2d + 1); Bool res ← true;
4:     for all (F'_in, G'_in) ∈ N_1 do
5:         S ← S ∪ F'_in;
6:     end for
7:     set GB ← GröbnerBasisOf(S);
8:     for φ' from 0 to σ' − 1 do
9:         if ExtractPolyWithHT(r_k, GB) = ∅ then
10:            res ← res & false;
11:        else
12:            U ← U∪ExtractPolyWithHT(r_φ', GB);
13:        end if
14:    end for
15:    set V ← CombinationsOfPolyIn(U, d);
16:    for all v ∈ V do
17:        if v contains all input shares then
18:            res ← res & false;
19:        end if
20:    end for
21:    return res;
22: end function
```

satisfies the GMS properties. Algorithm 4.4 checks the correctness property of GF-ACG for a GMS-based multiplier. The algorithm focuses on the relation between the secret variable a and its shares a_φ. (Other variables b and c can also be handled similarly.) In other words, since the correctness of multiplication indicates that the secret variables a, b, and c meet $c = a \times b$, the algorithm verifies whether $c = a \times b$ is derived from the equation of the internal structure, $a = \sum_\varphi a_\varphi$, $b = \sum_{\hat{\varphi}} b_{\hat{\varphi}}$, and $c = \sum_\varphi c_\varphi$, Function "MaskingRelation" in Line 4 derives the above equation representing the relation between secret variable and its shares from the set of edges. Then, the equivalence checking can be performed by means of GB and polynomial reduction in Lines 6 and 7, respectively. Algorithm 4.4 is less time consuming than the functional verification of GF-ACG because the computation of GB, which is a time-consuming procedure, is required only once. In addition, the proposed method can handle a reduced number of variables for computing GB owing to a hierarchical GF-ACG description.

Algorithm 4.5 verifies the dth-order noncompleteness by checking whether any d output shares from the $\boldsymbol{R}$ (i.e., $r_0, r_1, \ldots, r_{\sigma'-1}$) are independent of at least one input share (i.e., a_φ or $b_{\varphi'}$). Noncompleteness can be checked with a symbolic manipulation after a formula manipulation because a GF-ACG represents the relation of input, output, and intermediate variables by GF equations. More precisely, we first compute the relationship between $r_{\varphi'}$ and input shares. In Lines

3–5, we generate a polynomial set $\boldsymbol{S}$ that contains all the functional assertions of internal nodes for the GMS-based multiplier. Set $\boldsymbol{S}$ represents its function with intermediate variables $p_{\varphi,\hat{\varphi}}$, $l_{\varphi'}$, and $r_{\varphi'}$. In Line 6, we compute the Gröbner basis corresponding to $\boldsymbol{S}$ with a lexicographical order $\preceq$ induced by $a_\varphi \preceq b_{\hat{\varphi}} \preceq z_{\varphi'} \preceq r_{\varphi'} \preceq p_{\varphi,\hat{\varphi}} \preceq l_{\varphi'} \preceq c_\varphi$. Thus, in Lines 8–14, we can obtain the relationship between $r_{\varphi'}$ and input shares in accordance with the elimination theorem, where the function "ExtractPolyWithHT($r_{\varphi'}, \boldsymbol{GB}$)" extracts a polynomial whose head term is $r_{\varphi'}$ in $\boldsymbol{GB}$. If we fail to extract such a polynomial the GF-ACG is not satisfied with the dth-order noncompleteness. In Line 15, we create a set $\boldsymbol{V}$ including all d-combinations of polynomials in $\boldsymbol{U}$, where each element is a set of d polynomials. Set $\boldsymbol{V}$ has $\binom{d(2d+1)}{d}$ sets because $\boldsymbol{V}$ consists of σ' $(= d(2d+1))$ polynomials according to the number of outputs of the $\boldsymbol{R}$ layer. In Lines 16–20, we check if each element (i.e., set) $\boldsymbol{v}$ of $\boldsymbol{V}$ does not contain all input shares. The time or memory complexity of Alg. 4.5 increases at least proportionally to the factorial of d because $d(2d+1)$ elements have to be examined.

Finally, we can easily verify the uniformity by looking at the $\boldsymbol{R}$ because an $\boldsymbol{R}$ can guarantee uniformity.

4.4.5 Experimental generation

The performance of the proposed system is demonstrated through experimental generations of GMS-based multipliers. We measured the processing time of each step in Fig. 4.1 by generating GMS-based Mastrovito multipliers for different extension degrees m $(= 8, 16, 32, 64, 128,$ and $256)$ and GMS orders d $(1 \leq d \leq 5)$.

Table 4.9 shows the experimental result. Note that the conventional logic simulation cannot verify even the smallest multiplier (i.e., when $m = 8$ and $d = 1$) because its total input size is 72 bits (including fresh masks for the $\boldsymbol{R}$ layer). When $d \leq 4$, our system could generate all the multipliers including the 256-bit ones within 8 min. In the experiment, the verification time was given by $O(m^2)$ because the number of nodes in the GMS-based multiplier increased proportionally to m^2, and each node was verified in less than 1 s. Because the time for functional verification dominated the total generation time, we could significantly reduce it using GF-ACG. On the other hand, when $d \geq 5$, the GMS-property checking dominated the generation time. This is because the computation time of noncompleteness checking by Alg. 4.5 increases at least proportionally to the factorial of d. However, we confirmed that the proposed system successfully generated such a huge multiplier with $m = 256$ and $d = 5$ in only 15 min.

Table 4.9 Generation time of GMS-based GF multipliers (s)

Extension degree m	8					16					32				
GMS order d	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
GF-ACG synthesis	0.07	0.07	0.07	0.07	0.08	0.09	0.09	0.09	0.09	0.10	0.16	0.15	0.16	0.16	0.16
Functional verification	2.97	3.18	3.24	3.67	4.44	4.53	4.73	4.81	5.28	6.22	8.19	8.41	8.49	8.94	9.96
Property checking	0.18	0.18	0.23	4.17	331.96	0.18	0.18	0.24	3.89	330.02	0.18	0.19	0.24	4.11	330.61
GF-ACG to HDL	0.01	0.01	0.01	0.01	0.01	0.03	0.03	0.03	0.03	0.03	0.14	0.14	0.14	0.15	0.15
Total	3.23	3.43	3.56	7.92	336.50	4.82	5.03	5.17	9.29	336.36	8.67	8.90	9.03	13.36	340.88

Extension degree m	64					128					256				
GMS order d	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
GF-ACG synthesis	0.41	0.41	0.41	0.41	0.42	1.48	1.49	1.49	1.50	1.50	5.95	5.94	5.92	5.94	6.02
Functional verification	18.40	18.87	19.01	19.28	21.32	63.76	63.92	64.33	64.94	69.73	405.29	406.63	406.21	407.29	426.09
Property checking	0.19	0.20	0.26	4.03	336.20	0.24	0.24	0.30	3.93	364.15	0.41	0.42	0.48	5.70	406.74
GF-ACG to HDL	0.83	0.85	0.85	0.83	0.85	5.67	5.69	5.63	5.66	5.66	44.71	44.91	44.96	45.17	46.34
Total	19.83	20.33	20.53	24.56	358.78	71.16	71.34	71.76	76.03	441.04	456.35	457.91	457.57	464.10	885.19

4.4.6 Discussion

The result shows that functional verification occupies a large portion of the generation time when $d \leq 4$, which indicates that the limitation of our system is determined by functional verification. Thus, we can significantly reduce the generation time by employing fast formal verification based on GF-ACG, as discussed in Section 3.4.1.

A discussion point of importance is the applicability of our method. For example, a $GF(2^{128})$ multiplier is used in AES-GCM. Because the multiplication can be targeted by a side-channel attacker [12], there is a high demand for designing tamper-resistant multipliers for AES-GCM, which can be efficiently generated by our system as shown in Table 4.9. In addition, we could generate larger multipliers such as $m = 233$ and $m = 283$ that can be used in elliptic curve cryptography, which indicates that our system is also useful for designing tamper-resistant elliptic curve cryptographic processors.

Our system supports GMS-based multipliers whose order is not greater than five. However, the DPA order discussed in many literatures (e.g., [91]) is at most five. As the DPA order increases, attackers find retrieving the key more difficult because of noise. Until now, DPA-leakages have been evaluated with at most fifth-order even in a state-of-the-art laboratory setting [91]. Owing to the abovementioned reasons, the proposed system would be considerably useful in many practical or commercial devices considering tamper-resistance.

Finally, it was shown that side-channel-resistant circuits based on dth-order GMS could be constructed with $d+1$ shares at minimum, which would be useful for designing more compact circuits than conventional ones with $dt + 1$ shares [39]. Our system would be extended to multipliers with $d + 1$ shares, while our system would easily synthesize and verify the multipliers in the manner similar to the above.

4.5 Conclusion

This chapter presented an automatic generation system of GF arithmetic circuits for cryptographic hardware. Since the system generates verified HDL description from design specifications, the generated HDL can be used in the conventional EDA tools. Thus, the system can be considered as an implementation and interface of the proposed formal design methodology, and unifies the proposed methodology and conventional EDA tools. As a result of experimental generation, we confirmed that the system can generate more than 10,000 GF multipliers including large ones such as fifth-order GMS-based $GF(2^{256})$ multipliers in a practical time. This indicates that our system is useful for designing (DPA-resistant) cryptographic hardware (e.g., AES-GCM and PBC).

5 Design of Efficient AES Hardware

5.1 Introduction

This chapter designs highly efficient AES hardware in order to demonstrate the significance of redundant GF arithmetic, higher-degree functions, logic-level optimization, and pipelining in designing cryptographic hardware. First, we design an efficient $GF(2^8)$ inversion circuit and AES S-box based on a combination of redundant and non-redundant GF arithmetic. We then design an efficient AES hardware that supports both encryption and decryption. The functional assertion of proposed architecture is given as higher-degree functions because of optimizations to compress encryption and decryption datapaths. Moreover, we design an efficient DPA-resistant AES hardware based on GMS. The performance of the architectures designed in this chapter are evaluated through logic synthesis and gate-level timing simulation for power estimation in comparison with the conventional ones. As a result, our architectures have approximately 20–50% higher area-time efficiency than the conventional best ones.

5.2 Efficient $GF(2^8)$ inversion circuit and AES S-box

5.2.1 Overview

Inversion functions over $GF(2^m)$ are known as a useful component for m-bit substitution functions [108]. Therefore, for modern ciphers, the substitution function based on $GF(2^8)$ inversion is one of the most integral parts to be resistant against major cryptanalytic techniques such as differential and linear cryptanalyses [15,88]. Many ISO/IEC standard ciphers (e.g., AES and Camellia) employ an inversion function over $GF(2^8)$ in substitution functions [7,100]. For example, SubBytes of AES consists of an inversion over $GF(2^8)$ (i.e., S-box) and an affine transformation over $GF(2)$. The hardware performance of such ciphers heavily depends on the inversion circuits used. As a result of the explosive increase in resource-constrained devices in the context of Internet of Things (IoT) applications, there is currently substantial demand for lightweight implementation of inversion functions.

Many approaches to reducing the hardware cost of $GF(2^8)$ inversion circuits have been proposed. While it has been shown that direct mapping-based approaches (e.g., table-lookup, PPRM, and BDD [96,97,125]) are useful for low-latency implementation, the tower field approach, which calculates a^{-1} $(= a^{254})$ $(a \in GF(2^8))$ using the equivalent tower field, is a promising approach for achieving the compact and efficient implementation. This technique converts the original field $GF(2^8)$ into a tower field such as $GF(((2^2)^2)^2)$ and $GF((2^4)^2)$ in the middle of the inversion. Researchers have previously shown that the tower field approach is efficient because the subfields $GF((2^2)^2)$ and $GF(2^4)$ operations are designed more compactly than the original field operations. Satoh et al. [127] were the first to present a compact implementation of the AES S-box by the tower field $GF(((2^2)^2)^2)$ represented by PBs. Canright [33] further reduced the gate count of the AES S-box using NBs and optimizing the isomorphic mappings. Canright's implementation was the smallest for a long time. Nogami et al. [105] recently mixed polynomial and normal bases to achieve the most efficient implementation. They showed that the product of gate count and critical delay for the inversion circuit could be reduced by the Mixed Bases (MB). Some implementations using $GF((2^4)^2)$ have also been proposed by researchers such as Rudra et al. [123] and Jeon et al. [67], who presented PB-based $GF((2^4)^2)$ inversion circuit designs. These results suggest that such field representations have a significant impact on hardware performance. In addition, while the aforementioned ones are based on non-redundant representations, Wu et al. [144] and Nekado et al. [101] showed that RRB-based designs were useful for designing efficient inversion circuits.

This dissertation presents a technique in which non-redundant and redundant GF arithmetic are combined to achieve a compact and efficient $GF(2^8)$ inversion circuit design. The key idea un-

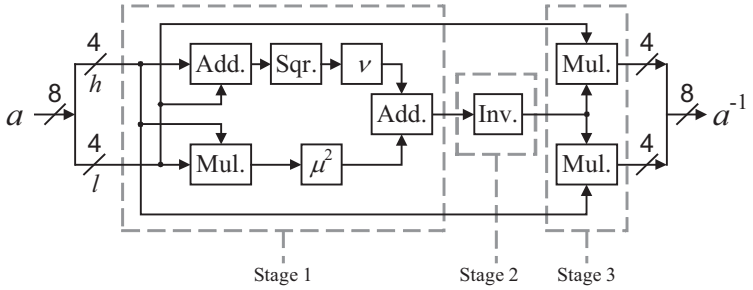


Fig. 5.1 Inversion circuit over $GF(((2^2)^2)^2)$ in [33] (Same as Fig. 3.16).

derlying the proposed circuit is calculation of the inversion of the tower field $GF((2^4)^2)$ by the NB, PRR, and RRB combination. The former part for the 16th and 17th powers of the input is calculated by an NB with a symmetric property. This is followed by calculation of the latter parts for $GF(2^4)$ inversion and $GF(2^4)$ multiplication by PRR and RRB, respectively. The mapping from NB to PRR/RRB is efficiently implemented by the symmetric property of the NB. The efficacy of the proposed circuit is evaluated by means of gate counts and logic synthesis results using a TSMC 65-nm CMOS standard cell library. The proposed circuit is approximately 25% higher efficiency (i.e., area-time product) excluding isomorphic mappings than any other conventional circuits, including those with the tower field $GF(((2^2)^2)^2)$. In addition, the flexibility of redundant representations in the proposed circuit enables it to have the best efficiency even including isomorphic mappings from/to $GF(2^8)$. To the best of our knowledge, the proposed circuit is the most efficient tower field arithmetic-based implementation for the AES S-box.

In the following, we fisrt review the tower-field $GF(2^8)$ inversion circuits based on Itoh-Tsujii Algorithm (ITA). Then, we show the proposed tower-field inversion circuits and their performance evaluation by logic synthesis in comparison with the conventional ones. In addition, we apply the proposed inversion circuit to AES S-box design.

5.2.2 Related works

This section briefly describes previous work on the design of $GF(2^8)$ inversion circuits based on tower field arithmetic. The inverse element of a non zero $a \in GF(2^8)$ is given by $a^{-1} = a^{254}$ because any non zero element of $GF(2^8)$ satisfies $a = a^{256}$. (The inverse element of zero is usually defined to be zero for common cryptographic applications.) The basic idea underlying the tower field approach is reduction of hardware cost by exploiting smaller arithmetic operations over subfield $GF((2^2)^2)$ or $GF(2^4)$ instead of $GF(2^8)$. There is a one-to-one mapping (i.e., an isomorphism) between the elements of $GF(2^8)$ and those of the tower field. This GF inversion over a tower field is efficiently implemented in the Itoh-Tsujii Algorithm (ITA) [66].

Figure 5.1 illustrates a $GF(2^8)$ inversion circuit presented in [33], where the datapath is divided into upper and lower 4 bits and each component denotes an arithmetic circuit over subfield $GF((2^2)^2)$. Let $a \in GF(((2^2)^2)^2)$ be the input given by $h\alpha^{16} + l\alpha$ in an NB $\{\alpha^{16}, \alpha\}$, where h and l $(\in GF((2^2)^2))$ are respectively the upper and lower 4 bits of a, and α is a root of a second degree irreducible polynomial over $GF((2^2)^2)$ (i.e., a modular polynomial for extending $GF((2^2)^2)$ to $GF(((2^2)^2)^2)$). The inversion of a is calculated in the following three stages: (1) Calculation of the 16th and 17th powers, (2) Subfield inversion, and (3) Final multiplication. Note that the above $GF((2^2)^2)$ operators are replaced with the $GF(2^4)$ operators in the case of the tower field $GF((2^4)^2)$.

The performance of this inversion circuit depends on the tower field and its basis representation. Three of the best known circuit structures are based on the tower field of $GF(((2^2)^2)^2)$. Satoh et al. first designed this kind of $GF(((2^2)^2)^2)$ inversion circuit using PB [127]. Canright then designed a more compact circuit based on NB [33]. The hardware cost of inversion and exponentiation operations can be reduced by NB because the squaring operation is performed solely by wiring. Nogami et al. presented the possibility of MB, which employs both polynomial and normal bases for the input and output data, respectively [105]. Their method exhibited improved performance in the product of gate count and critical delay for the $GF(((2^2)^2)^2)$ inversion circuit and the AES S-box, including isomorphic mappings. In addition to $GF(((2^2)^2)^2)$, it is possible to design efficient inversion circuits using another tower field of $GF((2^4)^2)$. Rudra et al. [123] and Jeon et al. [67] designed $GF((2^4)^2)$ inversion circuits based on PB with smaller critical delay than those of $GF(((2^2)^2)^2)$ inversion circuits.

5.2.3 Proposed $GF(2^8)$ inversion circuit

This section presents our proposed $GF(2^8)$ inversion circuit that takes full advantage of the above redundant GF arithmetic. The important ideas are to employ the tower field $GF((2^4)^2)$ inside the circuit and perform the subfield (i.e., $GF(2^4)$) operations using redundant GF arithmetic. We introduce PRR for the $GF(2^4)$ inversion because we can exploit a modular polynomial, $P(x) = x^5 + 1$, thanks to the irreducible fourth degree AOP. We also introduce RRB for the $GF(2^4)$ multiplication. In addition, we employ an NB for the input in order to exploit the Frobenius mapping feature, which performs the 16th power of input solely by wiring.

In accordance with ITA, our inversion circuit consists of three stages, as shown in Fig. 5.1. Here, we represent the inputs of Stages 1, 2, and 3 by NB, PRR, and RRB, respectively. In particular, we employ an NB that has a symmetric property, which makes it possible to convert the elements from NB to PRR without increasing the circuit delay.

Figure 5.12 shows a block diagram of our proposed circuit, where components H, L, and F

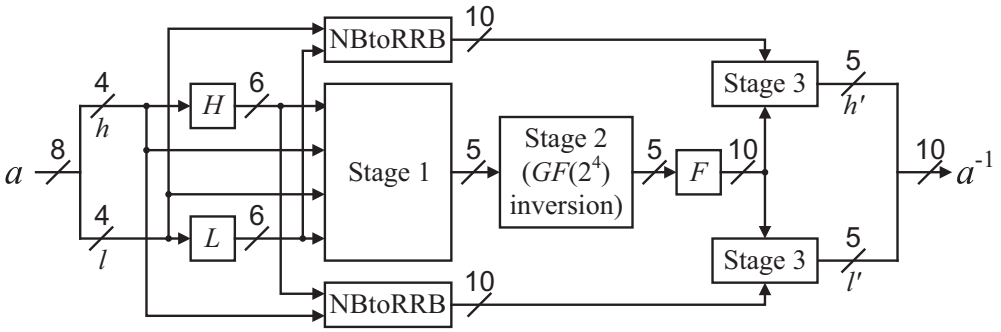


Fig. 5.2 Proposed inversion circuit.

respectively calculate $H_{i,j}$, $L_{i,j}$, and $F_{i',j'}$ described in the following. When input a is represented by $h\alpha^{16} + l\alpha$, components NB2RRB convert h and l from NB to RRB solely by wiring. Note that H and L are shared with Stages 1 and 3. The stages in the proposed circuit are designed as follows:

1. Calculation of the 16th and 17th powers

Stage 1 performs the 16th and 17th powers of input, where input a is given by NB, and outputs a^{16} and a^{17} are given by RRB and PRR, respectively. Let α be a root of a second degree irreducible polynomial over $GF(2^4)$. The irreducible polynomial is given by $\alpha^2 + \mu\alpha + \nu$, where μ and ν are the constants of $GF(2^4)$. When input a is represented by $a = h\alpha^{16} + l\alpha$ in an NB $\{\alpha^{16}, \alpha\}$, a^{16} and a^{17} are respectively given by

$$a^{16} = l\alpha^{16} + h\alpha, \tag{5.1}$$

$$a^{17} = hl\mu^2 + (h+l)^2\nu. \tag{5.2}$$

Eq. 5.1 indicates that a^{16} is performed by twisting wires.

The isomorphic mapping from NB to RRB does not require any additional gates because the NB (e.g., $\{\beta^4, \beta^3, \beta^2, \beta^1\}$) can be considered as a reduced version of RRB (e.g., $\{\beta^4, \beta^3, \beta^2, \beta^1, \beta^0\}$) with the identical root of the 4th degree AOP. Conversely, the isomorphic mapping from NB to PRR requires some gates. However, the symmetric property of the NB used in our circuit provides a mapping that does not increase the circuit delay.

Let us now look at the isomorphic mapping from NB to PRR. Here, an isomorphic mapping is represented by $z' = \Gamma(z)$, where an element z in one GF representation is converted into an element z' in another GF representation. In the binary vector form, the output $\boldsymbol{z}'$ is obtained from the product of a conversion matrix γ and the transposed input (i.e., $\boldsymbol{z}' = \gamma \boldsymbol{z}^T$) when the conversion matrix γ represents the isomorphism Γ. The PRR-based $GF(2^4)$ is given with the modular polynomial $P(x) = x^5 + 1$ ($G(x) = x + 1$ and $H(x) = x^4 + x^3 + x^2 + x + 1$) and the

conversion matrix from NB to PRR is as follows:

$$\phi = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}, \tag{5.3}$$

where the least significant bits are in the upper left corner. (See Appendix for an explanation of how to obtain the matrix.) Let d $(= d_4x^4 + d_3x^3 + \cdots + d_0)$ be the output of Stage 1 (i.e., the 17th power of input in PRR), where $d_0, d_1, \ldots,$ and d_4 are the elements of $GF(2)$. The output is provided by applying the isomorphism Φ from NB to PRR to a^{17} (i.e., the product of the conversion matrix ϕ and the transposed vector form of a^{17}). However, the multiplication of ϕ and the output of Eq. 5.2 requires an additional circuit with $2T_X$ delay if the multiplication is performed explicitly. To avoid such additional circuit, we derive another output equation from Eq. 5.2 as follows:

$$\begin{aligned} d &= \Phi(hl\mu^2 + (h+l)^2\nu) \\ &= \Phi(\mu^2(hl)) + \Phi(\nu((h+l)^2)) \\ &= \Phi'(hl) + \Phi''((h+l)^2), \end{aligned} \tag{5.4}$$

where Φ' and Φ'' are the linear functions obtained by merging Φ with the constant multiplications of μ^2 and ν, respectively. Note that constant multiplications over GF can also be given as linear functions represented by conversion matrices. When $\mu = \beta^4 + \beta$ and $\nu = \beta$, the resulting matrices ϕ' and ϕ'' representing respectively Φ' and Φ'' are given as

$$\phi' = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}, \ \phi'' = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix}, \tag{5.5}$$

where the least significant bits are in the upper left corners.

To design the circuit defined by Eq. 5.4, we exploit an NB with the symmetric property that h and l are given by $h = h_4\beta^4 + h_3\beta^3 + h_2\beta^2 + h_1\beta$ and $l = l_4\beta^4 + l_3\beta^3 + l_2\beta^2 + l_1\beta$ with a common NB $\{\beta^4, \beta^3, \beta^2, \beta^1\}$, where $h_1, \ldots, h_4$ and $l_1, \ldots, l_4$ are the elements of $GF(2)$ [*1]. As

[*1] The notation of bases in the NB is frequently given by $(\beta^{2^3}, \beta^{2^2}, \beta^{2^1}, \beta^{2^0}) = (\beta^3, \beta^4, \beta^2, \beta^1)$. In this subsection, we employ the notation $(\beta^4, \beta^3, \beta^2, \beta^1)$ to simplify the correspondence between the NB and PRR/RRB.

a result, the outputs $d_0, d_1, \ldots,$ and d_4 are given by

$$d_0 = (h_1l_2 + h_2l_1 + h_3l_4 + h_4l_3 + h_1l_1 + h_4l_4) + (h_1 + l_1 + h_3 + l_3 + h_4 + l_4), \tag{5.6}$$

$$d_1 = (h_1l_2 + h_2l_1 + h_1l_3 + h_3l_1 + h_2l_2 + h_4l_4) + (h_1 + l_1 + h_2 + l_2 + h_3 + l_3 + h_4 + l_4), \tag{5.7}$$

$$d_2 = (h_1l_3 + h_3l_1 + h_1l_4 + h_4l_1 + h_2l_3 + h_3l_2 + h_2l_2) + (h_1 + l_1 + h_2 + l_2 + h_4 + l_4), \tag{5.8}$$

$$d_3 = (h_1l_4 + h_4l_1 + h_2l_3 + h_3l_2 + h_2l_4 + h_4l_2 + h_3l_3) + (h_2 + l_2 + h_3 + l_3 + h_4 + l_4), \tag{5.9}$$

$$d_4 = (h_2l_4 + h_4l_2 + h_3l_4 + h_4l_3 + h_1l_1 + h_3l_3) + (h_1 + l_1 + h_2 + l_2 + h_3 + l_3), \tag{5.10}$$

respectively. Here, the symmetric property enables us to factor Eqs. 5.6–5.10 as follows:

$$d_0 = H_{1,2} \vee L_{1,2} + H_{3,4} \vee L_{3,4} + h_2 \vee l_2 + h_3l_3, \tag{5.11}$$

$$d_1 = H_{1,2} \vee L_{1,2} + H_{1,3}L_{1,3} + h_3 \vee l_3 + h_4 \vee l_4, \tag{5.12}$$

$$d_2 = H_{1,3} \vee L_{1,3} + H_{1,4}L_{1,4} + H_{2,3} \vee L_{2,3} + h_4 \vee l_4, \tag{5.13}$$

$$d_3 = H_{1,4} \vee L_{1,4} + H_{2,3} \vee L_{2,3} + H_{2,4}L_{2,4} + h_1 \vee l_1, \tag{5.14}$$

$$d_4 = H_{2,4} \vee L_{2,4} + H_{3,4} \vee L_{3,4} + h_1 \vee l_1 + h_2l_2, \tag{5.15}$$

where $H_{i,j} = h_i + h_j$, $L_{i,j} = l_i + l_j$ $(1 \leq i < j \leq 4)$, and $\vee$ denotes the OR operator (i.e., $a \vee b = a + b + ab$). The component denoted by Stage 1 in Fig. 5.12 performs the computations corresponding to Eqs. 5.11–5.15. Therefore, the proposed Stage 1 is performed with only $T_A + 3T_X$ (or $T_O + 3T_X$) delay, whereas conventional $GF(((2^2)^2)^2)$ inversion implementations are performed with at least $6T_X$ delay, where $T_A, T_O,$ and T_X denote the delays of the AND, OR, and XOR gates, respectively.

2. Subfield inversion

Stage 2 performs the inversion over the subfield $GF(2^4)$, where the input and output are given by PRR and RRB, respectively. We first describe the architecture of the PRR-based $GF(2^4)$ inversion, and then show the isomorphic mapping from PRR to RRB below.

The inversion over $GF(2^4)$ is performed by the 14th power of the input. The input (i.e., the output of Stage 1) d $(= d_4x^4 + d_3x^3 + \cdots + d_0)$ is given as an element of the PRR-based $GF(2^4)$. The input is satisfied with the condition (called the linear recurrence relation) $d_0 + d_1 + d_2 + d_3 + d_4 = 0$ [*2] because it is equivalent to the codeword of a CRC generated by $G(x)$ $(= x + 1)$, which makes it possible to perform the exponentiation by bit-wise operations over the PRR-based $GF(2^4)$ in an efficient manner.

[*2] The linear recurrence relation is used for error detection in CRC. A polynomial is a codeword of a CRC iff the relation is satisfied.

Let e $(= e_4x^4 + e_3x^3 + \cdots + e_0)$ be the inverse element of d in PRR, where $e_0, e_1, \ldots,$ and e_4 are the elements of $GF(2)$. Using the linear recurrence relation, we can derive $e_0, e_1, \ldots,$ and e_4 as follows:

$$e_0 = (d_1 \vee d_4)(d_2 \vee d_3), \tag{5.16}$$

$$e_1 = ((d_4 + 1)(d_1 + d_2)) \vee (d_0 d_4 (d_2 \vee d_3)), \tag{5.17}$$

$$e_2 = ((d_3 + 1)(d_2 + d_4)) \vee (d_0 d_3 (d_1 \vee d_4)), \tag{5.18}$$

$$e_3 = ((d_2 + 1)(d_1 + d_3)) \vee (d_0 d_2 (d_1 \vee d_4)), \tag{5.19}$$

$$e_4 = ((d_1 + 1)(d_3 + d_4)) \vee (d_0 d_1 (d_2 \vee d_3)). \tag{5.20}$$

According to Eqs. 5.16–5.20, the proposed Stage 2 requires $T_A + T_O + T_X$ delay, whereas the conventional structures [33, 101, 105, 127] require at least $T_A + 3T_X$. Note that when the multiplicative unit element $E(x)$ $(= x^4 + x^3 + x^2 + x)$ is given as the input, the output becomes not $E(x)$ but 1. However, the output is acceptable in Stage 3 (i.e., $GF(2^4)$ multiplication) because both $E(x)$ and 1 are the idempotent elements in the residue ring modulo $P(x)$.

Let us now look at the PRR-to-RRB mapping. To provide it uniquely, we focus on the definition of PRR in [47], in which the mapping Ψ from PRR defined by x to another representation defined by β is isomorphism. According to [60], the change-of-basis from PRR can be performed by substituting a root of $H(x)$ (i.e., β) to elememts. Let $f = f_4\beta^4 + f_3\beta^3 + \cdots + f_0$ be the output of Stage 2 in RRB, where $f_0, f_1, \ldots,$ and f_4 are elements of $GF(2)$. The output can be given by

$$f = \Psi(e) = e_4\beta^4 + e_3\beta^3 + e_2\beta^2 + e_1\beta + e_0,$$

because the RRB is defined using $H(x)$. This means that the PRR-to-RRB mapping is performed without any additional circuit, assuming that $f_0 = e_0, \ldots,$ and $f_4 = e_4$. As a result, the PRR-based design provides inversion and isomorphic mapping with fewer logic gates.

3. Final multiplication

Stage 3 generates the final output using two $GF(2^4)$ multiplication operations, where both the inputs and output are given by RRB. As stated above, the RRB-based $GF(2^4)$ multiplier is known to be one of the most efficient multipliers [101].

Let h' $(= h'_4\beta^4 + h'_3\beta^3 + \cdots + h'_0)$ be the upper 5 bits of the final output a^{-1} in RRB, where $h'_0, h'_1, \ldots,$ and h'_4 are the elements of $GF(2)$. Multiplying f and l, we can calculate elements

$h'_0, h'_1, \ldots,$ and h'_4 as follows:

$$h'_0 = L_{1,4}F_{1,4} + L_{2,3}F_{2,3}, \tag{5.21}$$

$$h'_1 = l_1F_{0,1} + L_{2,4}F_{2,4}, \tag{5.22}$$

$$h'_2 = l_2F_{0,2} + L_{3,4}F_{3,4}, \tag{5.23}$$

$$h'_3 = l_3F_{0,3} + L_{1,2}F_{1,2}, \tag{5.24}$$

$$h'_4 = l_4F_{0,4} + L_{1,3}F_{1,3}, \tag{5.25}$$

where $F_{i',j'}$ denotes $f_{i'} + f_{j'}$ $(0 \leq i' < j' \leq 4)$. The lower five bits of a^{-1} (denoted by l') are also obtained in the same manner as in Eqs. 5.21–5.25. The component denoted by Stage 3 in Fig. 5.12 performs the computations corresponding to Eqs. 5.21–5.25. Note here that the calculations for $F_{i',j'}$ can be shared within Stage 3. As a result, the number of circuit components for the two multipliers in Stage 3 is reduced.

The above inversion circuit achieves the shortest critical delay among tower field inversion circuits as evaluated in Section 4. In the following, we describe a variation of the proposed circuit with a smaller critical delay at the cost of a slight area overhead. We focus on Stage 2 and $F_{i',j'}$ computation prior to Stage 3. Since Stage 2 is given by one-to-one mapping and $F_{i',j'}$ is computed by XORing the output of Stage 2, we can unify Stage 2 and $F_{i',j'}$ computation by deriving $F_{i',j'}$ directly from $d_0, d_1, \ldots,$ and d_4 as follows:

$$F_{0,1} = d_2(d_1d_3 + 1) + d_0d_1 + d_3d_4, \tag{5.26}$$

$$F_{0,2} = d_4(d_1d_2 + 1) + d_0d_2 + d_1d_3, \tag{5.27}$$

$$F_{0,3} = d_1(d_3d_4 + 1) + d_0d_3 + d_2d_4, \tag{5.28}$$

$$F_{0,4} = d_3(d_2d_4 + 1) + d_0d_4 + d_1d_2, \tag{5.29}$$

$$F_{1,2} = d_1(d_0d_2 + 1) + d_0d_4 + d_2d_3, \tag{5.30}$$

$$F_{1,3} = d_3(d_0d_1 + 1) + d_0d_2 + d_1d_4, \tag{5.31}$$

$$F_{1,4} = d_0(d_2d_3 + 1) + d_1d_3 + d_2d_4, \tag{5.32}$$

$$F_{2,3} = d_0(d_1d_4 + 1) + d_1d_2 + d_3d_4, \tag{5.33}$$

$$F_{2,4} = d_2(d_0d_4 + 1) + d_0d_3 + d_1d_4, \tag{5.34}$$

$$F_{3,4} = d_4(d_0d_3 + 1) + d_0d_1 + d_2d_3. \tag{5.35}$$

The above computation for each $F_{i',j'}$ requires $T_A + 2T_X$ delay while the critical delay of the original circuit in Fig. 5.12 requires $T_A + T_O + 2T_X$. Thus, we can further reduce the critical delay of the inversion circuit at the expense of a few additional gates.

Table 5.1 Critical delay and gate count of inversion circuits over tower fields

	Field	Representation		Gate count	Critical delay
		Tower field	Intermediate field		
Satoh et al. [127]	$GF(((2^2)^2)^2)$	PB	PB	(30, 0, 96, 0, 0, 6, 0)	$4T_A + 17T_X$
Canright [33]	$GF(((2^2)^2)^2)$	NB	NB	(0, 0, 56, 0, 0, 34, 6)	$4T_A + 15T_X$
Nogami et al. [105]	$GF(((2^2)^2)^2)$	PB and NB	PB and NB	(36, 0, 95, 0, 0, 0, 0)	$4T_A + 14T_X$
Rudra et al. [123]	$GF((2^4)^2)$	PB	PB	(60, 0, 72, 0, 0, 0, 0)	$4T_A + 10T_X$
Jeon et al. [67]	$GF((2^4)^2)$	PB	PB	(58, 2, 67, 0, 0, 0, 0)	$4T_A + 10T_X$
Nekado et al. [101]	$GF((2^4)^2)$	NB	RRB	(42, 0, 68, 2, 0, 0, 0)	$4T_A + 7T_X$
This study (low-power)	$GF((2^4)^2)$	NB	NB, PRR, and RRB	(38, 16, 51, 0, 4, 0, 0)	$3T_A + T_O + 6T_X$
This study (high-speed)	$GF((2^4)^2)$	NB	NB, RR, and RRB	(45, 10, 57, 0, 10, 0, 0)	$3T_A + 6T_X$

5.2.4 Performance evaluation

Table 5.1 shows the circuit delay and gate count of the proposed inversion circuit, where $(g_0, g_1, g_2, g_3, g_4, g_5, g_6)$ in the Gate count column respectively indicate the number of AND, OR, XOR, XNOR, NOT, NAND and NOR gates, and Representation indicates the GF representation(s) used in the circuit. In the Representation column, "Tower field of $GF(2^8)$" and "Intermediate field" denote the representations for $GF(((2^2)^2)^2)$ or $GF((2^4)^2)$ and $GF((2^2)^2)$ or $GF(2^4)$, respectively. "This study (low-power)" denotes the inversion circuit in Fig. 5.12, and "This study (high-speed)" denotes the circuit where Stage 2 and $F_{i',j'}$ computation are unified as described in the last paragraph of Section 3. For comparison, Tab. 5.1 also shows those of the conventional inversion circuits. The critical delay of all the conventional ones were given by reference to [101]. On the other hand, the gate counts of the conventional ones were individually given because there was no single reference data covering all of them. The gate count of [33] was given from the original paper, that of [127] was given from a public source code by the authors [139], and those of [67, 101, 123][*3] were given by reference to [101]. The gate count of [105] was given from a straightforward structure designed by us according to [105] since there was neither public data nor source code.

The critical paths of Stages 1, 2, and 3 in the proposed circuit require T_A+3T_X, $T_A+T_O+T_X$, and $T_A + 2T_X$ delay, respectively. In our high-speed version, that of Stages 2 and 3 is at most $3T_A+2T_X$. As a result, the total delay of our inversion circuit is $3T_A+T_O+6T_X$ (or $3T_A+6T_X$), which is the fastest compared with the other inversion circuits. The gate count in "This study (high-speed)" is smaller or comparable to the conventional ones because the number of additional

[*3] The paper [123] focused on efficinet software implementation of AES. However, in the paper, tower-field arithmetic and bit-slicing techniques were used for implementing $GF(2^8)$ inversion on software. In other words, they first proposed a tower-field $GF(2^8)$ inversion circuit (i.e., hardware implementation) and then proposed a software implementation based on the circuit. Note that the authors in [123] mentioned that their technique could be applied to both hardware and software implementation. Thus, we can compare the performance of the proposed circuit with that of [123] according to its circuit description.

Table 5.2 Performance evaluation of inversion circuits over tower fields

	Area [GE]	Time [ns]	Power [μW]	Area-Time product	Power-Time product
Satoh et al. [127]	210.50	3.02	38.0	635.72	114.76
Canright [33]	178.00	2.92	38.6	519.75	112.71
Nogami et al. [105]	291.50	3.67	70.9	1,069.81	260.20
Rudra et al. [123]	219.00	2.32	35.6	508.08	82.59
Jeon et al. [67]	221.25	2.19	34.1	474.54	72.68
Nekado et al. [101]	204.50	1.89	33.4	386.51	63.13
This study (low-power)	174.75	1.81	20.7	316.30	37.47
This study (high-speed)	187.50	1.55	21.4	290.63	33.17

XOR and XNOR gates due to unification is trivial. In total, the high-speed is more efficient than any other circuits including our low-power version.

To conduct a detailed evaluation, the above tower-field $GF(2^8)$ inversion circuits were synthesized using Synopsys Design Compiler with a TSMC 65-nm CMOS standard cell library. Table 5.7 shows the synthesis results, where Area indicates the circuit area estimated based on a two-way NAND equivalent gate size (i.e., gate equivalents (GE)), Time indicates the circuit delay under the worst-case conditions, Power indicates the power consumption estimated by the dynamic gate-level simulation-based method[*4] at the operation frequency of 100 MHz, and Area-Time product indicate the product of Area and Time. For the best performance comparison, an area optimization option (which maximizes the effort of minimizing the number of gates without flattening the description) was applied. Note that the results were consistent even when the following speed optimization (which searches for the minimum timing without increasing the area obtained from the prior area optimization) options was applied. The conventional inversion circuits were also synthesized and evaluated using the same tool, option, and power estimation method. The source codes of [127] and [33] were obtained from authors' websites [139] and [35] , respectively. (Like them, we also applied gate-reduction techniques to our inversion circuit.) The source codes of [105], [123], [67], and [101] were described by us according to the structures given in the papers.

Our inversion circuit of low-power version achieved the smallest area although the total gate count of the proposed circuit was roughly the same as the convetional ones [33, 101]. The less XOR gates in our circuit would lead to the smaller circuit area because an XOR and XNOR gates require larger area than a NAND and NOR gate in standard cell libraries. The power consumption

[*4] This method performed a timing simulation (or delay simulation) at the gate-level using a set of test input data, and logged the switching activity of all internal gates. In this subsection, we checked all patterns of circuit inputs and internal states, and then estimated the circuit power from the simulation log and the cell information of the target library.

was basically proportional to the circuit area. However, the power consumption of Canright's circuit was relatively larger in spite of its small area. This is because Stage 1 of Canright's circuit has two paths whose delays are different as shown in Fig. 5.1. Note that the Stage 2 also has such paths owing to the recursive construction of $GF(((2^2)^2)^2)$ inversion. Such difference causes glitches that usually make the power consumption larger. On the other hand, Satoh's circuit based on $GF(((2^2)^2)^2)$ had lower power than Canright's one because the two paths in the Stage 1 (and 2) were compressed in Satoh's source code. Consequently, we confirmed that the low-power version of the proposed circuit achieved the smallest area of 174.75 GE, the smaller circuit delay of 1.81 ns compared with conventional other circuits, and the lowest power consumption of 20.7 μW that was 38% lower than the conventional best. In addition, the high-speed version of the proposed circuit achieved the smallest critical delay of 1.55 ns, and the area-time product was 24.8% smaller than that of the conventional best circuit, respectively.

5.2.5 Application to AES S-box design

The proposed inversion circuit was efficiently applied to the AES S-box design. The AES S-box consists of a $GF(2^8)$ inversion and an affine transformation over $GF(2)$. Here, the $GF(2^8)$ is represented in a PB with an irreducible polynomial $x^8 + x^4 + x^3 + x + 1$. Therefore, a change-of-basis between $GF(2^8)$ and $GF((2^4)^2)$ is required if the inversion over $GF((2^4)^2)$ is applied. Figure 5.3 shows an overview of the AES S-box with tower field arithmetic. In an S-box (Fig. 5.3(a)), the input (in the PB-based $GF(2^8)$) is initially mapped to the tower field by applying an change-of-basis Δ_f which is given by an isomorphim. After the inversion operation over the tower field, the inverse mapping and affine transformation are finally performed in series. Here, we can merge the inverse mapping into the affine transformation because both of them are represented in the form of constant matrices over $GF(2)$. The merged mapping is denoted by Δ_l. This merging reduces the delay and gate counts. On the other hand, in an inverse S-box (Fig. 5.3(b)), the inverse affine transformation is performed prior to change-of-basis and tower field inversion. Hence, The inverse affine transformation and change-of-basis are unified as Λ_f, and the inverse change-of-basis Λ_l is solely performed after the tower field inversion.

The matrices for the change-of-basis Δ_f and Δ_l (Λ_f and Λ_l) have an impact on the performance of S-box. When tower field $GF((2^4)^2)$ is used, the matrices are defined by the bases of $GF(2^4)$ and the defining polynomials for the extension of $GF(2^4)$ to $GF((2^4)^2)$. The efficiency of the two matrices for change-of-basis Δ_f and Δ_l (Λ_f and Λ_l) is determined by the largest Hamming weight in the columns. For example, if the largest Hamming weight in the columns is four, the critical path becomes $2T_X$ delay. If it is five, the critical path becomes $3T_X$ delay. Therefore, the matrices should be selected with a view to minimizing the largest Hamming weight in the

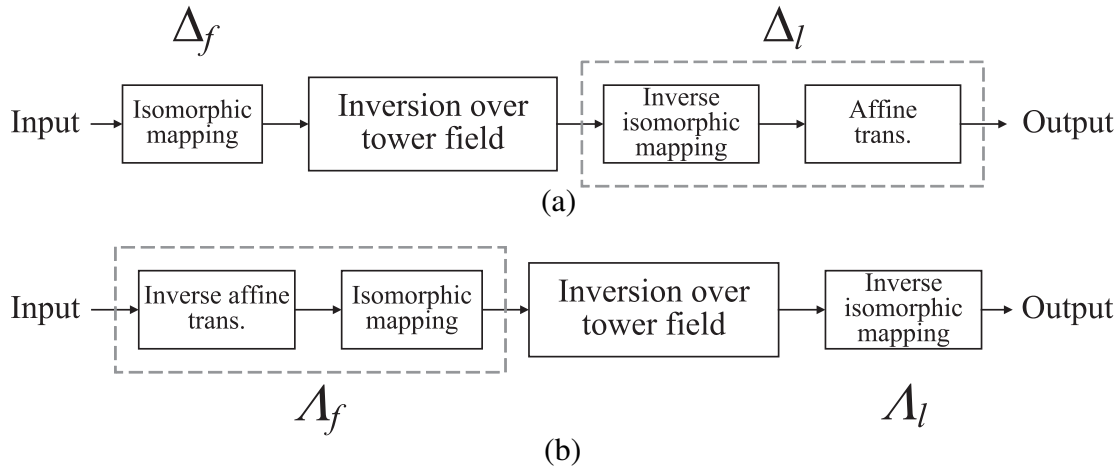


Fig. 5.3 Overview of AES (a) S-box and (b) inverse S-box based on tower field arithmetic.

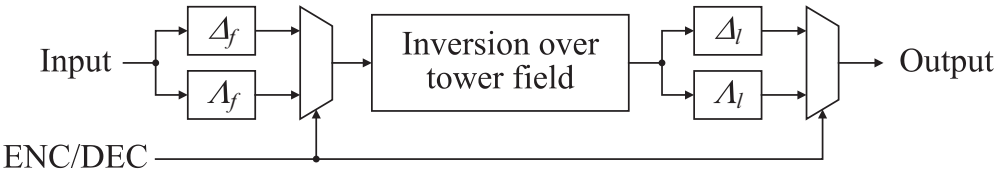


Fig. 5.4 Typical architecture of unified S-box.

columns.

Some techniques for optimizing change-of-basis between AES field and $GF((2^4)^2)$ have been reported in [86, 87, 101]. In [86, 87], an algorithm which searches all construction of isomorphic mappings from/to PB-based $GF((2^4)^2)$ was used. In addition, a technique in [101] used More Miscellaneously Mixed Bases (MMMB) to expand search space of the isomorphim for change-of-basis by utilizing asymmetric property of input of inversion circuit given by RRB. However, these technique cannot be applied to our inversion circuits because the irreducible polynomial of $GF(2^4)$ should be given by the AOP of degree 4 and the input of inversion circuit should be given by symmetric NB.

We design a unified S-box that supports both encryption and decryption shown in Fig. 5.4 in addition to the S-box. There are efficient conversion matrices for either encryption or decryption in the proposed inversion circuit. However, we found that there is no efficient conversion matrix for both encryption and decryption in the structure of Fig. 5.4 due to the deficiency of search space of isomorphim for change-of-basis. Therefore, we introduce a new technique for expanding the search space. Figure 5.5 illustrates the AES S-box structures with the proposed technique,

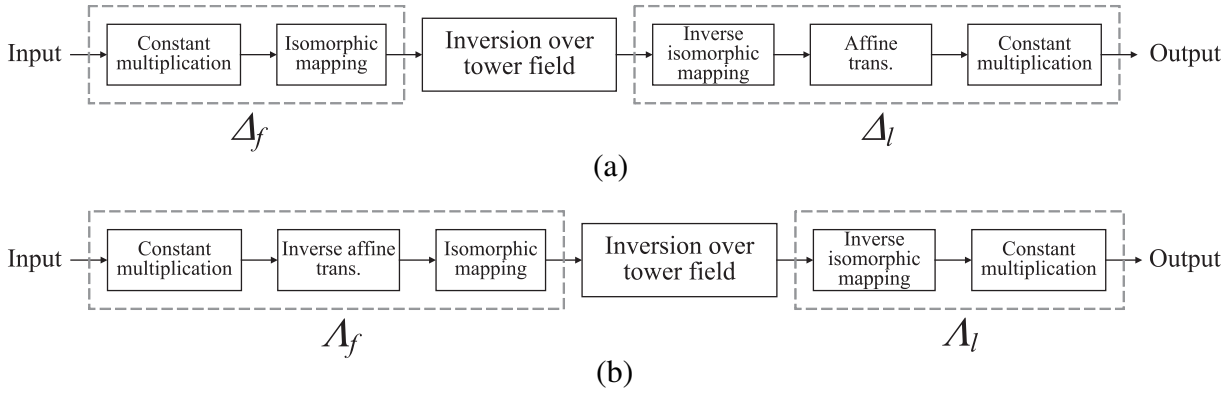


Fig. 5.5 AES (a) S-box and (b) inverse S-box with proposed technique for optimizing linear mappings.

where the component "constant multiplication" perform a multiplication over PB-based $GF(2^8)$ with a fixed value. The S-box based on tower field arithmetic (denoted by S) is represented by $S(a) = A(\Theta'((\Theta(a))^{-1})) + 0\text{x}63$, where Θ is a change-of-basis from the PB-based $GF(2^8)$ to a tower field, Θ' is its inverse change-of-basis, and A denotes the linear mapping of the affine transformation. We can rewrite the equation using a non-zero fixed value c (in the PB-based $GF(2^8)$) as follows:

$$S(a) = A(c(\Theta'((\Theta(c(a)))^{-1}))) + 0\text{x}63.$$

Because the multiplication with c is a linear mapping, we can unify c and Θ as Δ_f, and unify Θ', c, and A as Δ_l. The fixed value c can take one of 255 elements over $GF(2^8)$. Thus, we can increase the variety of conversion matrices by 255 times. The proposed technique can be also applied to the inverse S-box S' as follows:

$$S'(a) = c'(\Theta'((((\Theta(c'(A'(a)))))^{-1})) + \Theta(c'(0\text{x}05)),$$

where A' is the linear mapping of the inverse affine transformation and c' is a non-zero fixed value for decryption. Note that c and c' do not need to be equal in general.

With the proposed technique, we successfully found efficient conversion matrices δ_f, δ_l, λ_f, and λ_l respectively for Δ_f, Δ_l, Λ_f, and Λ_l when the $GF(2^4)$ elements of Stage 1 are represented in an NB $\{\beta^4, \beta^3, \beta^2, \beta^1\}$ and the defining polynomial for the extension is given by $\alpha^2 + (\beta^4 +$

$\beta)\alpha + \beta$. As a concrete example, the matrices δ_f, δ_l, λ_f, and λ_l are given by

$$\delta_f = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}, \delta_l = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}, \quad (5.36)$$

$$\lambda_f = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}, \lambda_l = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}, \quad (5.37)$$

where the least significant bits are in the upper left corners. The vector of the inverse affine transformation is given by $\Theta(c'(0\text{x}05)) = 0\text{x}29$. Here, the largest Hamming weight in each column of δ_f and λ_f is at most four while that of δ_l and λ_l is at most six. This means that the former and latter mappings are implemented with at most $2T_X$ and $3T_X$ delays, respectively.

Tables 5.3 and 5.4 show the critical delay and the number of XOR gates required for the mappings of the proposed AES S-box and unified S-box compared with the conventional implementations, respectively. Here, the numbers of XOR gates for Canright's and Jeon's S-boxes, which were achieved by factorizing XOR gates, were given according to their papers. We applied the similar factorization technique to our S-boxes. On the other hand, those for other S-boxes were derived directly from conversion matrices without any factorization. Our circuits achieve $3T_A + T_O + 11T_X$ or $3T_A + 11T_X$ delay, which is smaller than the conventional S-boxes with tower field arithmetic[*5]. Tables 5.3 and 5.4 also show the synthesis results (area, delay time, and power) obtained from the same tool, synthesis options, and method as the above, where ENC and

[*5] According to [26] and [27], a logic minimization method can further reduce the total gates or critical delay of [33] and [105]. However, the same minimization can also be applied to other circuits including ours. Therefore, we did not apply the minimization in this dissertation. Note that, in our environment, the critical delay and area-time product of our S-boxes without the minimization technique are smaller than those in [26] and [27].

Table 5.3 Performance comparison of S-boxes based on tower field arithmetic

	Critical delay		# XOR gates		Area	Time	Power	Area-Time	Power-Time
	Δ_f	Δ_l	Δ_f	Δ_l	[GE]	[ns]	[μW]	product	product
Satoh et al. [127]	$3T_X$	$3T_X$	24	21	283.50	4.41	72.4	1,250.24	319.28
Canright [33]	$3T_X$	$3T_X$	24 (in total)		236.75	4.30	72.8	1,018.04	313.04
Nogami et al. [105]	$2T_X$	$2T_X$	20	19	392.00	4.78	113.0	1,873.77	540.14
Rudra et al. [123]	$3T_X$	$3T_X$	20	23	283.39	3.65	60.0	1,034.36	219.00
Jeon et al. [67]	$3T_X$	$3T_X$	10	31	312.25	4.82	73.2	1,505.03	352.82
Nekado et al. [101]	$2T_X$	$3T_X$	17	36	289.50	3.29	64.6	952.46	212.53
This study (low-power)	$2T_X$	$3T_X$	15	21	249.00	3.04	43.1	756.96	131.02
This study (high-speed)	$2T_X$	$3T_X$	15	21	261.50	2.78	44.4	726.98	123.43

Table 5.4 Performance comparison of unified S-boxes based on tower field arithmetic

	Critical delay		# XOR gates		Area	Time	Power [μW]		Area-Time
	Δ_f or Λ_f	Δ_l or Λ_l	$\Delta_f + \Lambda_f$	$\Delta_l + \Lambda_l$	[GE]	[ns]	ENC	DEC	product
Satoh et al. [127]	$3T_X$	$3T_X$	46	44	366.75	4.94	97.0	95.2	1,811.75
Canright [33]	$3T_X$	$3T_X$	20	18	311.25	4.97	109	111	1,546.91
Rudra et al. [123]	$3T_X$	$3T_X$	42	44	367.00	4.06	80.6	79.9	1,490.01
Jeon et al. [67]	$3T_X$	$3T_X$	22	31	381.00	5.93	112	124	2,259.33
Nekado et al. [101]	$2T_X$	$3T_X$	32	68	372.50	3.66	89.0	96.2	1,363.36
This study (low-power)	$2T_X$	$3T_X$	32	56	334.75	3.33	59.9	67.5	1,114.71
This study (high-speed)	$2T_X$	$3T_X$	32	56	347.50	3.05	61.4	69.4	1,059.87

DEC in the Power column in Tab. 5.4 indicate the power consumption for encryption and decryption, respectively. The source codes were given from the same methods as Tab. 5.7. Note here that Canright's design in [33] supports both encryption and decryption, and we have slightly changed it to support only encryption to allow a fair comparison to our design (for Tab. 5.3). On the other hand, since Satoh's design supports either encryption or decryption, we have also changed it to support both encryption and decryption as described in Fig. 5.4 (for Tab. 5.4). As a result, the area-time products of our AES S-boxes unified S-boxes were respectively 28.1% and 31.5% better than Canright's ones, which had been the smallest for a long time, and were also respectively 23.2% and 22.3 % better than Nekado's latest ones. The power consumption of our S-boxes and unified S-boxes were respectively 40.8% and 42.1% better than Canright's ones, and were also respectively 33.2% and 31.2 % better than Nekado's ones. Note that the synthesis results and power consumption of inverse S-boxes would be consistent with Tab. 5.3.

A further discussion point when applying the proposed method to cryptographic cores is the well-known side channel issue. In particular, the resource sharing of Stages 1 and 3 would cause glitches during the computation. To apply our method to masking-based countermeasures with pipelining such as threshold implementation [17, 103] and generalized masking scheme [39, 122] to defeat sophisticated (higher-order) attacks utilizing glitches, we need to decompose shared

resources, which results in the increase of 12 XOR gates for the low-power version or 17 XOR gates for high-speed version in total. Note however that such increase would also happen in other works (e.g., [33, 101]) using the similar optimization. In contrast, our method is more suitable for multiplexing-based countermeasures, such as WDDL [138], due to the high efficiency.

5.3 High throughput/gate AES hardware

5.3.1 Overview

Nowadays, many cryptographic algorithms are required to be implemented in resource-constrained devices and embedded systems with a high throughput and efficiency. Since 2001, many hardware implementations for AES have been proposed and evaluated for CMOS logic technologies. Studies of AES design are important from both practical and academic perspectives since AES employs an SPN structure and the major components (i.e., an 8-bit S-box and permutation used in ShiftRows and MixColumns) followed by many other security primitives.

AES encryption and decryption are commonly used in block-chaining modes such as CBC, CMAC, and CCM (e.g., for SSL/TLS, IEEE802.11 wireless LAN, and IEEE802.15.4 wireless sensor networks). Therefore, AES hardware that efficiently perform both encryption and decryption in the above block-chaining modes are highly demanded. However, many conventional hardware architectures employ pipelining techniques to enhance the throughput and efficiency [76, 78, 87], although such block-wise parallelism is not available in the above block-chaining modes. For example, the highest throughput of 53 Gbps was achieved in the previous best encryption/decryption architecture [87], but it only worked in the ECB mode. In addition, these previous studies assumed offline key scheduling owing to the difficulty of on-the-fly scheduling. On-the-fly key scheduling should be implemented in most resource-constrained devices because an offline key scheduling implementation requires additional memory to store expanded round keys. Thus, it is valuable to investigate an efficient AES architecture with on-the-fly key scheduling without any pipelining technique.

In this subsection, we design a new round-based AES hardware for both encryption and decryption with on-the-fly key scheduling, which achieves the lowest critical path delay (the least number of serially connected gates in the critical path) with less area overhead compared to conventional architectures with tower-field S-boxes. Our architecture employs new operation-reordering and register-retiming techniques to unify the inversion circuits for encryption and decryption without any selectors. In addition, these techniques make it possible to unify the affine transformation and linear mappings (i.e., the isomorphism and constant multiplications) to reduce the total number of logic gates. The proposed and conventional AES encryption/decryption datapaths are synthe-

sized and evaluated with the TSMC standard-cell and NanGate open-cell libraries. The evaluation results show that our architecture can perform both (CBC-)encryption and decryption more efficiently. For example, the throughput per gate of the proposed architecture in the NanGate 15-nm process is 72% larger than that of the best conventional architecture.

5.3.2 Related works to unified AES detapath for encryption and decryption

Architectures that perform one round of encryption or decryption per clock cycle without pipelining are the most typical for AES design and are called round-based architectures in this dissertation. Round-based architectures can be implemented more efficiently in terms of throughput per area than other architectures by utilizing the inherent parallelism of symmetric key ciphers. For example, the byte-serial architecture [86, 93] is intended for the most compact and low-power implementations such as in RFID but is not intended for the high throughput and efficiency. In contrast, round-based architectures are suitable for a high throughput per gate, which leads to a low-energy implementation [127].

To design such round-based encryption/decryption architectures in an efficient manner, we consider how to unify the resource-consuming components such as the inversion circuits in SubBytes/InvSubBytes for the encryption and decryption datapaths. There are two conventional approaches for designing such unified datapaths. The first approach is to place two distinct datapaths for encryption and decryption and select one of the datapaths with multiplexers as in [78]. Figure 5.6 shows an overview of the datapath flow in [78], where the inversion circuit is shared by both paths, and additional multiplexers are used at the input and output of the encryption and decryption paths. In [78], a reordered decryption operation was introduced as shown in Fig. 5.7. The intermediate value is stored in a register after InvMixColumns instead of AddRoundKey. Such register retiming was suitable for pipelined architectures. The main drawbacks of such approaches are the false critical path delay and the required area and delay overheads caused by four multiplexers. The critical path of the datapath in Fig. 5.6 is denoted in bold, which would never be active because it passes from the decryption path to the encryption path. This false critical path reduces the maximum operation frequency owing to logic synthesis due to the false longest logic chain. The overhead caused by the multiplexers is also nonnegligible for common standard-cell-based designs.

The second approach is to unify the circuits of the functions SubBytes, ShiftRows, and MixColumns with their inverse functions, respectively. Figure 5.8 shows the datapath in [127] where encryption and decryption paths are combined using the second approach, where the reordering technique is given in Fig. 5.9. The order of the decryption operations is changed to be the same

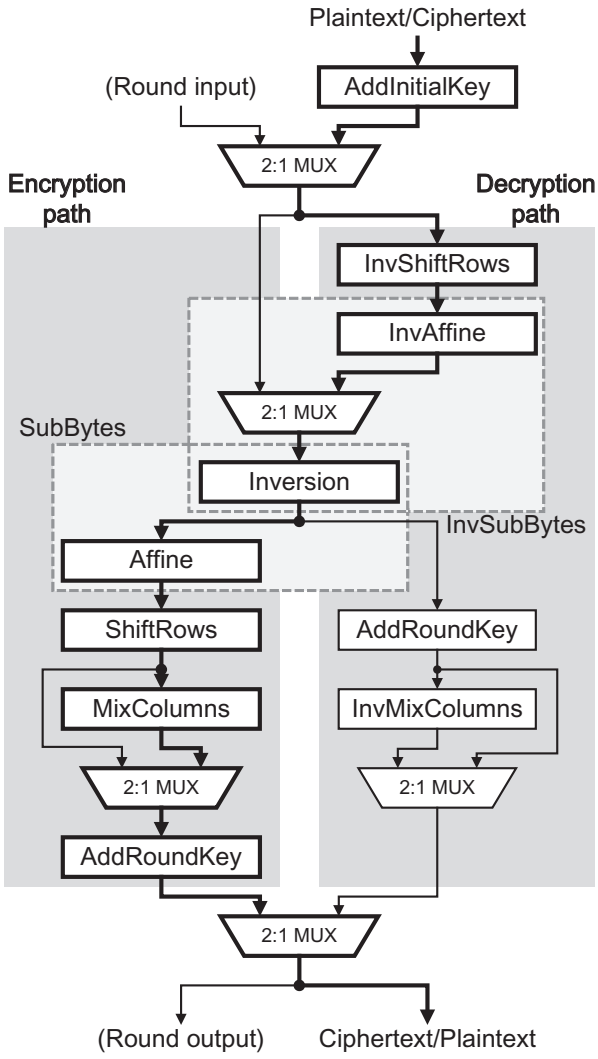


Fig. 5.6 Conventional parallel datapath in [78].

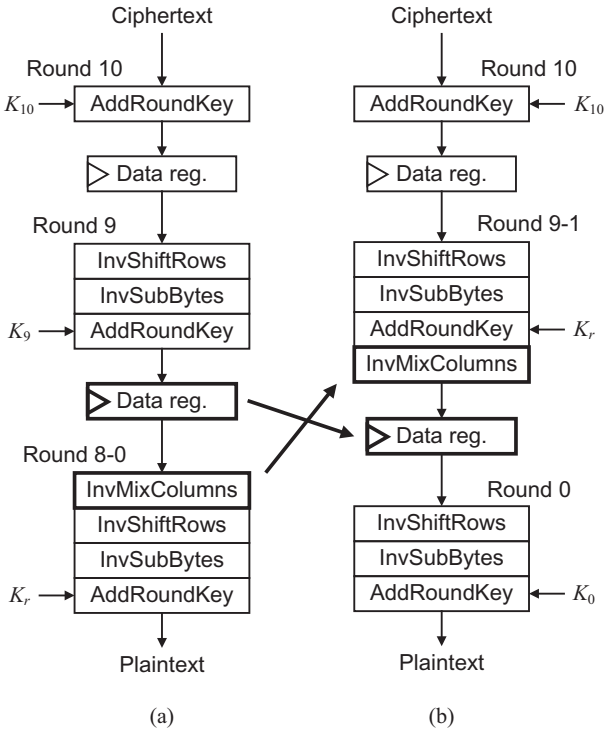


Fig. 5.7 Register-retiming techniques in [78]: (a) original and (b) resulting decryption flows.

as that of the encryption operations. Note that the order of (Inv)SubBytes and (Inv)ShiftRows can be changed without any overhead, and the datapath in [127] changes the order of SubBytes and ShiftRows in the encryption. The reordering of AddRoundKey and InvMixColumns utilizes the linearity of InvMixColumns as follows: $MC^{-1}(M_r + K_r) = MC^{-1}(M_r) + MC^{-1}(K_r)$, where MC^{-1} is the function InvMixColumns, and M_r and K_r are the intermediate value after InvShiftRows and the round key at the r-th round, respectively. Here, InvMixColumns requires the round keys, whereas MixColumns and InvMixColumns can be unified to reduce the area. Therefore, this type of architecture requires an additional InvMixColumns to compute $MC^{-1}(K_r)$ for decryption. In addition, the false path and multiplexer overhead exist because each function and its inverse function are implemented in a partially serial manner with multiplexers like SubBytes

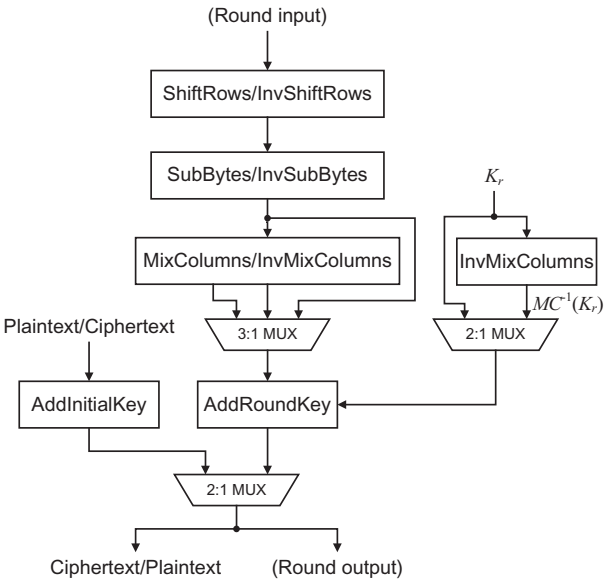


Fig. 5.8 Conventional datapath in [127], where encryption and decryption paths are combined.

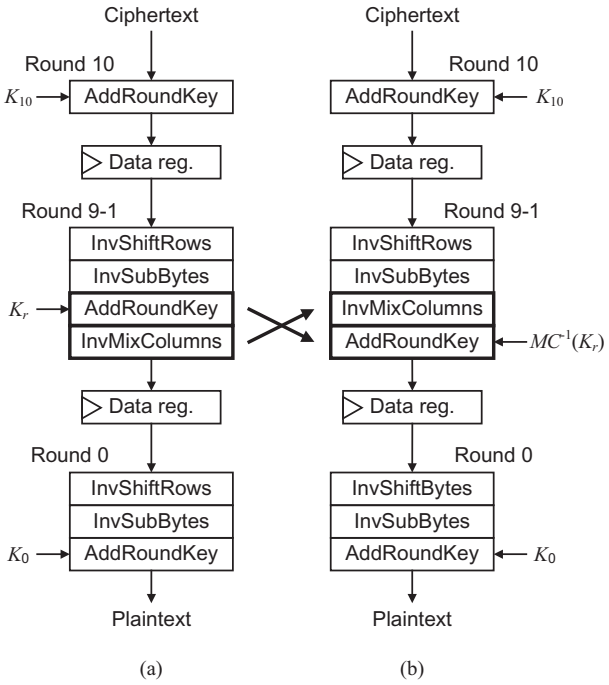


Fig. 5.9 Reordering technique in [127]: decryption flows (a) before and (b) after reordering.

and InvSubBytes in Fig. 5.6, where the critical path consists of Affine, Inversion, InvAffine, and an additional multiplexer.

The architecture in [87] employs a reordering technique similar to [127]. The major difference is the intermediate value stored in the register. The architecture in [77] also employs the same approach that combines the encryption and decryption datapaths, but does not change the order of AddRoundKey and InvMixColumns to remove InvMixColumns to compute $MC^{-1}(K_r)$. As a result, an additional selector is required to unify MixColumns and InvMixColumns.

As described above, sharing inversion circuits is essential for designing efficient AES hardware. Although a hardware T-box architecture such as that in [97] is also useful for a high-throughput implementation, it is not applicable to the above shared datapath owing to the lack of sharable components between the encryption and decryption paths.

The design of the inversion circuit used in (Inv)SubBytes has a significant impact on the performance of AES implementations as described in Section 5.2. Since we focus on efficient AES hardware in terms of throughput/gate, we focus on inversion circuits based on tower-field arithmetic. On the other hand, some architectures perform all of the AES subfunctions (i.e., SubBytes

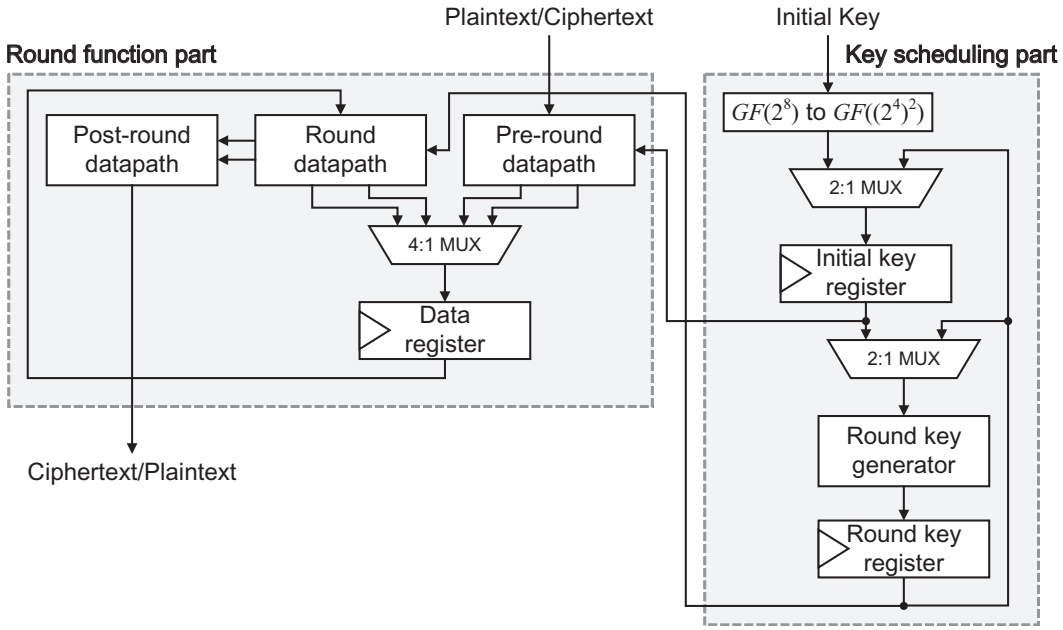


Fig. 5.10 Overall architecture of proposed AES hardware.

as well as ShiftRows, MixColumns, and AddRoundKey) over the tower field, where isomorphic mapping and its inverse mappings are performed at the timings of the data (i.e., plaintext and ciphertext) input and output, respectively [59, 86, 87, 93, 123]. In other words, the cost of field conversion is suppressed when the conversion is performed only once during encryption or decryption. However, the cost of constant multiplications in MixColumns over a tower field is worse than that over the AES field while inversion is efficiently performed over the tower field. More precisely, in tower-field architectures, such linear mappings including constant multiplications usually require $3T_{XOR}$ delay, where T_{XOR} indicates the delay of an XOR gate [101]. The XOR gate count used in (Inv)MixColumns over a tower field is also worse than that over AES field.

5.3.3 Designed AES hardware

This subsection designs a new round-based AES architecture that unifies the encryption and decryption paths in an efficient manner. The key ideas for reducing the critical path delay are summarized as follows: (1) to merge linear mappings such as MixColumns and isomorphic mappings as much as possible by reordering subfunctions, (2) to minimize the number of selectors to unify the encryption and decryption paths by the above merging and a register retiming, and (3) to perform isomorphic mapping and its inverse mappings only once in the pre- and post-round datapaths. We can reduce the number of linear mappings to at most one for each round operation as the

effect of (1). Moreover, we can reduce the number of selectors to only one (4-to-1 multiplexer) in the unified datapath as the effect of (2) while the inversion circuit is shared by the encryption and decryption paths. From the idea of (3), we can remove the isomorphic mapping and its inverse mappings from the critical path. Figure 5.10 shows the overall architecture that consists of the round function and key scheduling parts. Our architecture performs all of the subfunctions over a tower field for both the round function and key scheduling parts and therefore applies isomorphic mappings between the AES and tower fields in the datapaths of the pre- and post-round operations, which are represented as the blocks "Pre-round datapath" and "Post-round datapath" in Fig. 5.10. "Round datapath" performs one round operation for either encryption or decryption.

Round function part

The designed architecture in this subsection employs a unified datapath for encryption and decryption as in [78] and applies new operation-reordering and register-retiming techniques to address the conventional issues of a false critical path and additional multiplexers. Using our operation-reordering technique and then merging linear mappings, we can reduce the number of linear mappings on the critical path of the round datapath to at most one. Our reordering technique also allows to unify the linear mappings and affine transformation in a round. The unification of these mappings can drastically reduce the critical path delay and the XOR-gate count of linear mappings, even in a tower-field architecture.

The new operation reordering is derived as follows. First, the original round operation of AES encryption is represented by the following equation:

$$\begin{aligned} m_{\epsilon,\varepsilon}^{(r+1)} &= u_{-\epsilon}S(m_{0,\epsilon+\varepsilon}^{(r)}) + u_{1-\epsilon}S(m_{1,\epsilon+\varepsilon}^{(r)}) + u_{2-\epsilon}S(m_{2,\epsilon+\varepsilon}^{(r)}) + u_{3-\epsilon}S(m_{3,\epsilon+\varepsilon}^{(r)}) + k_{\epsilon,\varepsilon}^{(r)} \\ &= \sum_{e=0}^{3}(u_{e-\epsilon}S(m_{e,\epsilon+\varepsilon}^{(r)})) + k_{\epsilon,\varepsilon}^{(r)}, \end{aligned} \tag{5.38}$$

where $m_{\epsilon,\varepsilon}^{(r)}$ and $k_{\epsilon,\varepsilon}^{(r)}$ are the ϵth row and εth column intermediate value and round key at the rth round, except for the final round. Note that the subscripts of each variable are a member of $\mathbb{Z}/4\mathbb{Z}$. The function S indicates the 8-bit S-box, and u_0, u_1, u_2, and u_3 are the coefficients of the matrix of MixColumns and respectively given by $\beta, \beta+1, 1$, and 1, where β is the indeterminate of $GF(2^8)$ satisfying $\beta^8 + \beta^4 + \beta^3 + \beta + 1 = 0$. We can rewrite Eq. 5.38 by decomposing S into inversion and affine transformation as follows:

$$m_{\epsilon,\varepsilon}^{(r+1)} = \sum_{e=0}^{3}(u_{e-\epsilon}(A(\left(m_{e,\epsilon+\varepsilon}^{(r)}\right)^{-1}) + \mathrm{const})) + k_{\epsilon,\varepsilon}^{(r)}, \tag{5.39}$$

where A is the linear mapping of the affine transformation, and const $(= \beta^6 + \beta^5 + \beta + 1)$ is a

constant. In the case of tower-field architectures, Eq. 5.39 is represented by

$$m_{\epsilon,\varepsilon}^{(r+1)} = \sum_{e=0}^{3}(u_{e-\epsilon}(A(\Delta'(\left(\Delta(m_{e,\epsilon+\varepsilon}^{(r)})\right)^{-1})) + \text{const})) + k_{\epsilon,\varepsilon}^{(r)}, \tag{5.40}$$

where Δ is the isomorphic mapping from the AES field to a tower field, and Δ' is the inverse isomorphic mapping.

The linear mappings, which include an isomorphism and constant multiplications over the GF, are performed by the constant multiplication of the corresponding matrix over $GF(2)$. Therefore, we can merge such mappings to reduce the critical path delay and the number of XOR gates. In addition, we consider the variable $d_{\epsilon,\varepsilon}^{(r)}$ of the tower field derived from $m_{\epsilon,\varepsilon}^{(r)}$. Substituting $m_{\epsilon,\varepsilon}^{(r)}$ with $\Delta'(d_{\epsilon,\varepsilon}^{(r)})$ $(= m_{\epsilon,\varepsilon}^{(r)})$, we can merge the linear mappings as follows:

$$d_{\epsilon,\varepsilon}^{(r+1)} = \sum_{e=0}^{3}(U_{e-\epsilon}(\left(d_{e,\epsilon+\varepsilon}^{(r)}\right)^{-1})) + \Delta(\text{const}) + \Delta(k_{\epsilon,\varepsilon}^{(r)}), \tag{5.41}$$

where $U_\epsilon(x) = \Delta(u_\epsilon(A(\Delta'(x))))$. Note that an arbitrary linear mapping L satisfies $L(a+b) = L(a) + L(b)$. Thus, the linear mappings of a round in Eq. 5.41 can be merged into at most one, even with a tower-field S-box, whereas the linear mappings in Eq. 5.40 cannot be.

On the other hand, the corresponding equation for AES decryption with tower-field arithmetic is given by

$$d_{\epsilon,\varepsilon}^{(r-1)} = \sum_{e=0}^{3}(\Delta(v_{e-\epsilon}(\Delta'(\left(\Delta(A'(\Delta'(d_{e,\varepsilon-\epsilon}^{(r)}))) + \Delta(\text{const}')\right)^{-1} + \Delta(k_{e,\varepsilon-\epsilon}^{(r)}))))), \tag{5.42}$$

where A' indicates the linear mapping of the inverse affine transformation. The coefficients v_0, v_1, v_2, and v_3 are respectively given by $\beta^3+\beta^2+\beta$, $\beta^3+\beta+1$, $\beta^3+\beta^2+1$, and β^3+1, and const' $(= \beta^2+1)$ is a constant. Here, the linear mappings cannot be merged into one because they are performed both before and after the inversion operation. In addition, if we construct an encryption/decryption datapath based on Eqs. 5.41 and 5.42, the inversion circuit cannot be shared by encryption and decryption without a selector because the timings of the inversion operations are different from each other. Therefore, we consider a register retiming to store the intermediate value $s_{\epsilon,\varepsilon}^{(r)}$ given after the inverse affine transformation over the tower-field. Here, $s_{\epsilon,\varepsilon}^{(r)}$ is given by $s_{\epsilon,\varepsilon}^{(r)} = \Delta(A'(\Delta'(d_{\epsilon,\varepsilon}^{(r)}))) + \Delta(\text{const}')$. In the decryption, we store $s_{\epsilon,\varepsilon}^{(r)}$ in the data register instead of $d_{\epsilon,\varepsilon}^{(r)}$. Using $s_{\epsilon,\varepsilon}^{(r)}$ and $s_{\epsilon,\varepsilon}^{(r-1)}$, we rewrite Eq. 5.42 as follows:

$$s_{\epsilon,\varepsilon}^{(r-1)} = \sum_{e=0}^{3}(V_{e-\epsilon}(\left(s_{e,\varepsilon-\epsilon}^{(r)}\right)^{-1} + \Delta(k_{e,\varepsilon-\epsilon}^{(r)}))) + \Delta(\text{const}'), \tag{5.43}$$

where $V_\epsilon(x) = \Delta(A'(v_\epsilon(\Delta'(x))))$.

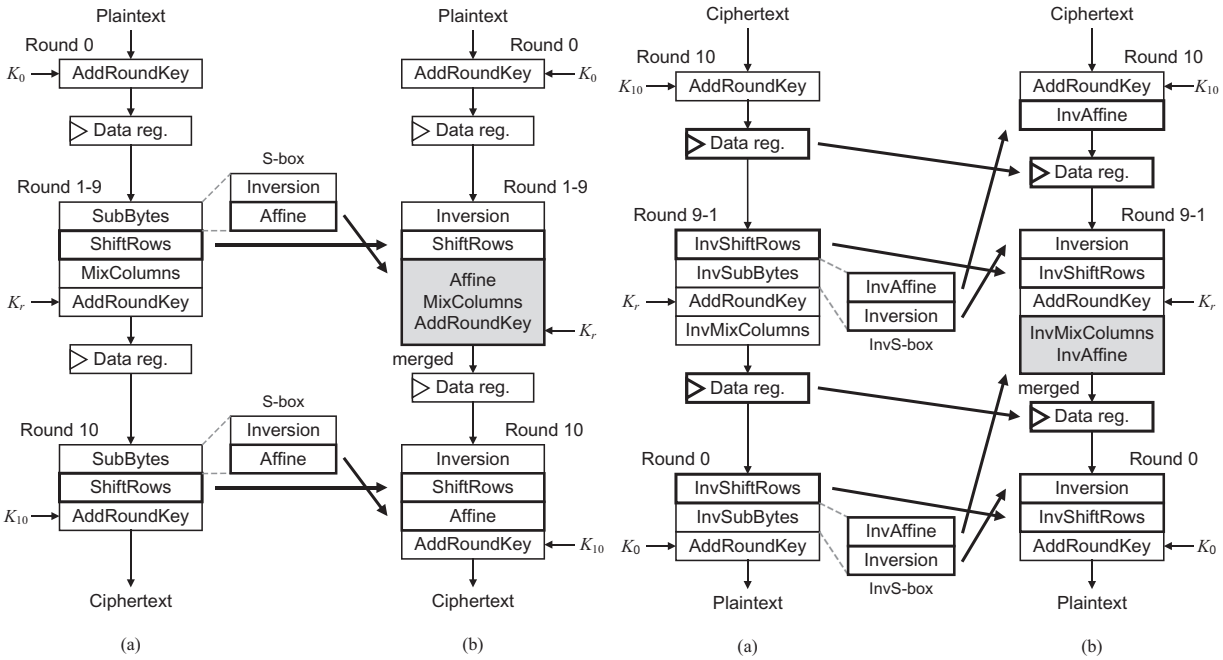


Fig. 5.11 (i) Encryption and (ii) decryption flows (a) before and (b) after our operation-reordering and register-retiming techniques.

Our round datapath is constructed with a minimal critical path delay according to Eqs. 5.41 and 5.43. Here, we further reorder the sequence of operations (i.e., subfunctions) to share inversion circuits without additional selectors and to unify the linear mappings. Figure 5.11 shows the proposed reordering technique. We first decompose SubBytes into the inversion and (Inv)Affine. In the encryption, Affine, MixColumns, and AddRoundKey can be merged by exchanging Affine and ShiftRows. In the decryption, the inversion circuit is located at the beginning of the round by exchanging the inversion and InvShiftRows. Thus, additional selectors for sharing the inversion circuit are not required thanks to the operation-reordering and register-retiming techniques. This is because both inversion operations are performed at the beginning of the round, which means that the data register output can be directly connected to the inversion circuit.

Figure 5.12 illustrates our round function datapath with the unification of linear mappings. Our architecture employs only one 128-bit 4-in-1 multiplexer, whereas conventional ones employ several 128-bit multiplexers. For example, the datapath in [77] employs seven 128-bit multiplexers[*6]. Fewer selectors can reduce the critical path delay and circuit area and solve the false critical path problem. Unified affine and Unified affine^{-1} in Fig. 5.12 perform the unified linear mappings (i.e.,

[*6] The selectors in SubBytes/InvSubBytes are included in the seven multiplexers.

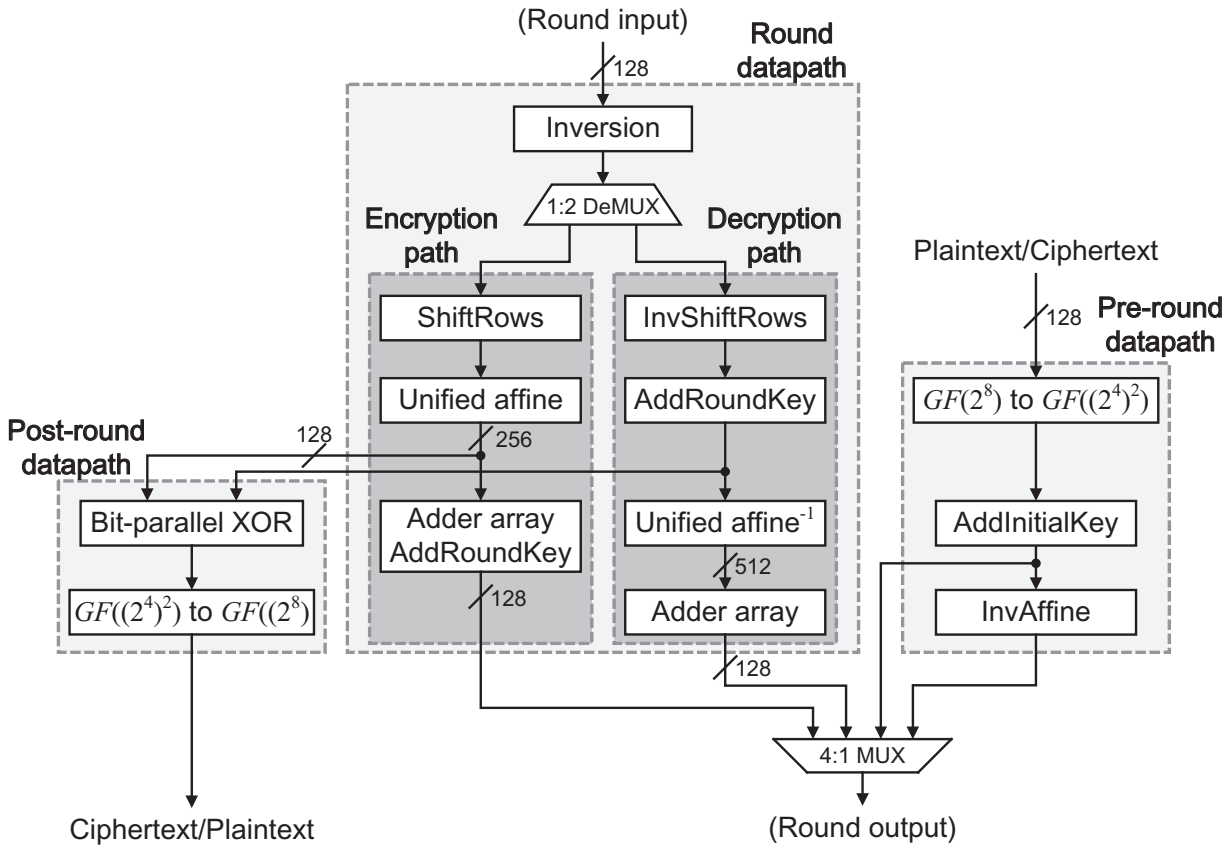


Fig. 5.12 Proposed round function part.

$U_0, \ldots, U_3$ and $V_0, \ldots, V_3$) and constant addition. The number of linear mappings on the critical path is at most one in our architecture, whereas that of the conventional architectures is not. We can also suppress the overhead of constant multiplication over the tower field by the unification. Adder arrays in Fig. 5.12 consist of four 4-input 8-bit adders in MixColumns or InvMixColumns. In the encryption, the factoring technique for MixColumns and AddRoundKey [101] is available for Unified affine, which makes the circuit area smaller without a delay overhead. As a result, the data width between Unified affine and Adder array in Encryption path is reduced from 512 to 256 bits because the calculations of U_1 and U_3 are not performed in Encryption path. In addition, Adder array and AddRoundKey are unified in Encryption path because both of them are composed of 8-bit adders[*7]. On the other hand, since there is no factoring technique for InvMixColumns without delay overheads, the data width from Unified affine^{-1} to Adder array in Decryption path is 512 bits. Finally, an inactive path can be disabled using a demultiplexer since our datapath is fully parallel after the inversion circuit. Thanks to the disabling, a multiplexer and AddRoundKey

[*7] Some architectures such as [77, 127] unify AddInitialKey and AddRoundKeys. We did not unify them to avoid increasing the number of selectors.

are unified as Bit-parallel XOR. (The addition of $\Delta(c)$ in Unified affine should be active only when encryption.) In addition, the demultiplexer would suppress power consumption due to a dynamic hazard. Although tower-field inversion circuits are known to be power-consuming owing to dynamic hazards [96], these hazards can be terminated at the input of the inactive path.

Our datapath employs the inversion circuit presented in the previous section because it has the highest area-time efficiency among inversion circuits including one using a logic minimization technique [26]. We can merge the isomorphic mappings in order to reduce the linear function on the round datapath to only one, even if the inversion circuit has different GF representations at the input and output. Since the output is given by an RRB, the data width from Inversion to Unified affine (or Unified affine^{-1}) is given by 160 bits. However, AddRoundKey in the decryption path and Bit-parallel XOR in the post-round datapath are implemented respectively by only 128 XOR gates because the NB used as the input is equal to the reduced version of the RRB. In addition, a 1:2 DeMUX is implemented with NOR gates thanks to the redundancy, whereas nonredundant representations require AND gates.

Key scheduling part

The on-the-fly key scheduling part is shared by the encryption and decryption processes. For the encryption, the key scheduling part first stores the initial key in the initial key register (in Fig. 5.10) and then generates the round keys during the following clock cycles. For the decryption, the final round key should be calculated from the initial key and stored in the initial key register in advance. The key scheduling part then generates the round keys in the reverse order by the round key generator (in Fig. 5.10). However, conventional key scheduling datapaths such those as in [77, 127] are not applicable to our round datapath because they have a loop with a false path and/or a longer true critical path than our datapath.

To address the above issue, we introduce a new architecture for the key scheduling datapath. For on-the-fly implementation, the subkeys are calculated for each of the four subkeys (i.e., 128 bits) in a clock cycle. Therefore, the on-the-fly key scheduling for the encryption is expressed as

$$\begin{cases} k_0^{(r+1)} &= k_0^{(r)} + KeyEx(k_3^{(r)}) \\ k_1^{(r+1)} &= k_0^{(r)} + k_1^{(r)} + KeyEx(k_3^{(r)}) \\ k_2^{(r+1)} &= k_0^{(r)} + k_1^{(r)} + k_2^{(r)} + KeyEx(k_3^{(r)}) \\ k_3^{(r+1)} &= k_0^{(r)} + k_1^{(r)} + k_2^{(r)} + k_3^{(r)} + KeyEx(k_3^{(r)}) \end{cases}, \tag{5.44}$$

where $k_0^{(r)}, k_1^{(r)}, k_2^{(r)}$, and $k_3^{(r)}$ are a 32-bit subkey at the rth round and $KeyEx$ is the key expansion function that consists of a round constant addition, RotWord, and SubWord. The inverse key

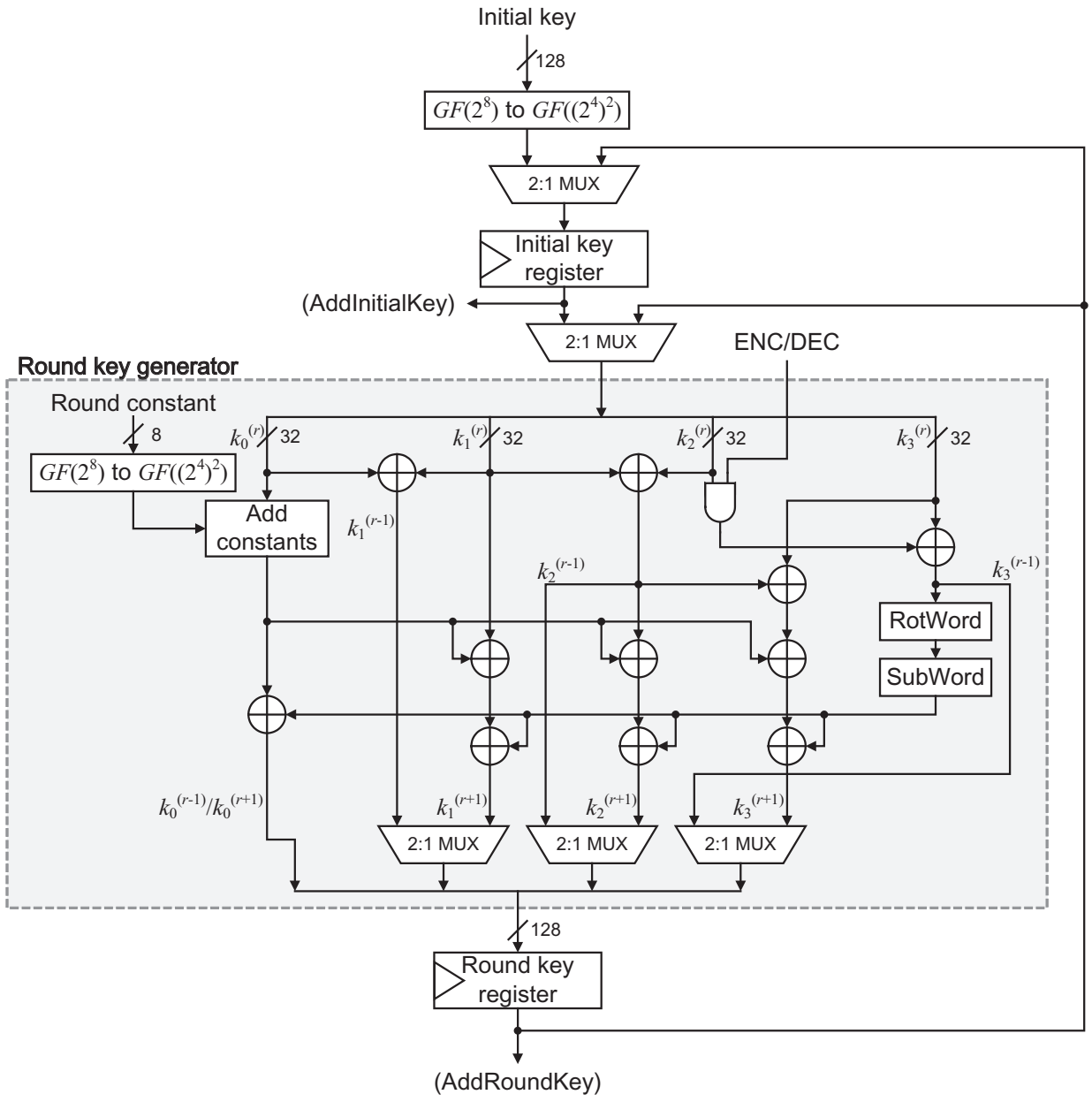


Fig. 5.13 Key scheduling part.

scheduling for the decryption is represented by

$$\begin{cases} k_0^{(r-1)} &= k_0^{(r)} + KeyEx(k_2^{(r)} + k_3^{(r)}) \\ k_1^{(r-1)} &= k_0^{(r)} + k_1^{(r)} \\ k_2^{(r-1)} &= k_1^{(r)} + k_2^{(r)} \\ k_3^{(r-1)} &= k_2^{(r)} + k_3^{(r)} \end{cases} . \tag{5.45}$$

Figure 5.13 shows the key scheduling datapath architecture, where the $KeyEx$ components are unified for encryption and decryption. Note here that most of adders (i.e., XOR gates) for com-

Table 5.5 Synthesis results for conventional and our AES hardware architectures with area optimization

	Area (GE)	Latency (ns)	Max. freq. (MHz)	Throughput (Gbps)	Efficiency (Kbps/GE)
TSMC 65-nm					
Satoh et al. [127]	13,671.75	78.10	140.85	1.64	119.88
Lutz et al. [78]	20,380.50	68.50	145.99	1.87	91.69
Liu et al. [77]	12,538.75	85.25	129.03	1.50	119.75
Mathew et al. [87]	20,639.50	97.68	112.61	1.31	63.49
This study	15,242.75	46.97	234.19	2.73	178.78
NanGate 45-nm					
Satoh et al. [127]	12,560.99	31.57	348.43	4.05	322.78
Lutz et al. [78]	20,000.66	20.30	492.61	6.31	315.26
Liu et al. [77]	11,829.34	34.43	319.49	3.72	314.28
Mathew et al. [87]	17,573.33	41.80	263.16	3.06	174.25
This study	13,814.69	16.94	649.35	7.56	546.96
NanGate 15-nm					
Satoh et al. [127]	14,526.01	4.36	2,524.17	29.37	2,022.04
Lutz et al. [78]	23,391.49	4.57	2,185.84	25.44	1,087.37
Liu et al. [77]	13,847.25	4.74	2,321.05	27.01	1,950.46
Mathew et al. [87]	21,361.00	5.32	2,066.93	24.05	1,125.95
This study	15,468.97	2.65	4,144.22	48.22	3,117.44

puting $k_1^{(r+1)}, k_2^{(r+1)}$, and $k_3^{(r+1)}$ should be nonintegrated to make the critical path shorter than that of the round function part. The input key is initially mapped to the tower field, and all of the computations (including AddRoundKey) are performed over the tower field. The ENC/DEC signal controls the input to RotWord and SubWord using a 32-bit AND gate. The upper 2-in-1 multiplexer selects an initial key or a final round key as the input to Initial key register, the middle 2-in-1 multiplexer selects a key stored in Initial key register or a round key as the input to Round key generator, and the lower 2-in-1 multiplexers select encryption or decryption path. The round constant addition is performed separately from RotWord and SubWord to reduce the critical path delay. As a result, the critical path delay of the key scheduling part becomes shorter than that of the round function part.

5.3.4 Performance evaluation

Tables 5.5 and 5.6 summarize the synthesis results of our AES encryption/decryption architecture by Synopsys Design Compiler (Version D2010-3) with the TSMC 65-nm and NanGate 45- and 15-nm standard-cell libraries [4, 5] under the worst-case conditions, where Area indicates the

Table 5.6 Synthesis results for conventional and our AES hardware architectures with area-speed optimization

	Area (GE)	Latency (ns)	Max. freq. (MHz)	Throughput (Gbps)	Efficiency (Kbps/GE)
TSMC 65-nm					
Satoh et al. [127]	14,516.50	56.87	193.42	2.25	155.05
Lutz et al. [78]	22,883.25	33.90	294.99	3.78	165.00
Liu et al. [77]	13,970.50	60.17	182.82	2.13	152.27
Mathew et al. [87]	23,298.49	65.45	168.07	1.96	83.94
This study	15,807.00	34.10	322.58	3.75	237.47
NanGate 45-nm					
Satoh et al. [127]	13,386.67	24.42	450.45	5.24	391.55
Lutz et al. [78]	22,417.01	14.40	694.44	8.89	396.52
Liu et al. [77]	12,443.66	28.27	389.11	4.53	363.86
Mathew et al. [87]	19,243.67	31.90	344.83	4.01	208.51
This study	14,582.99	13.53	813.01	9.46	648.73
NanGate 15-nm					
Satoh et al. [127]	16,924.74	3.31	3,322.26	38.66	2,284.17
Lutz et al. [78]	25,692.49	2.08	4,799.85	61.44	2,391.28
Liu et al. [77]	15,768.43	3.65	3,014.14	35.07	2,224.29
Mathew et al. [87]	23,789.48	4.03	2,729.18	31.76	1,334.95
This study	17,232.00	1.80	6,117.70	71.19	4,131.14

circuit area estimated on the basis of a two-way NAND equivalent gate size (i.e., gate equivalents (GEs)); Latency indicates the latency for encryption, which is estimated by the circuit path delay of the datapath under the worst low condition; Max. freq. indicates the maximum operation frequency obtained from the critical path delay; Throughput indicates the throughput at the maximum operation frequency; and Efficiency indicates the throughput per area, which corresponds to the product of the area and latency in this nonpipelined design[*8]. To perform a practical performance comparison, an area optimization (which maximizes the effort of minimizing the number of gates without flattening the description) was applied in Tab. 5.5, and an area-speed optimization (where an asymptotical search with a set of timing constraints was performed after the area optimization) was applied in Tab. 5.6.

In these tables, the conventional representative datapaths [77,78,87,127] were also synthesized using the same optimization conditions. The source codes for these syntheses were described by

[*8] Design Compiler generated a static power consumption report for each architecture. However, the report dose not consider the effect of glitches while tower-field inversion circuits are known to include non-trivial glitches [96]. Therefore, we did not mention the power consumption report to avoid misleading.

the authors referring to [77, 78, 87, 127], except for the source codes of Satoh's and Canright's S-boxes in [33, 127] that can be obtained from their websites [35, 139]. For a fair comparison, the datapaths of [78] and [87] were adjusted to the round-based nonpipelined architecture corresponding to the our datapath. Note that only the inversion circuit over a PB-based $GF((2^4)^2)$ in [87] was not described faithfully according to the paper[*9]. Latency and Throughput were calculated assuming that the datapath of [78] requires 10 clock cycles to perform each encryption or decryption and the others require 11 clock cycles. This is because the initial key addition and first-round computation are performed with one clock cycle for [78]. Area was calculated without the initial key, round key, and data registers to compare the datapaths more clearly. Note also that the key scheduling parts of [78] and [87] were implemented with the one presented in this subsectionbecause there is no description for the key scheduling parts. (For [78], the isomorphic mapping from $GF(2^8)$ to $GF((2^4)^2)$ was removed for applying to the round function part.)

The results in Tab. 5.5 show that our datapath achieves the lowest latency (i.e., highest throughput) compared with the conventional ones with tower-field inversion circuits owing to the lower critical path delay. Moreover, the circuit area is not the largest owing to fewer selectors. Note that the latency is consistent with the throughput because these circuits are not pipelined. Although all operations are translated to the tower field in our architecture, the area and delay overheads of MixColumns and InvMixColumns are suppressed by the unification technique. In addition, even with a tower-field S-box, our architecture has an advantage with regard to the latency over Lutz's one with table-lookup-based inversion, as indicated in Tab. 5.6. As a result, our architecture is more efficient in terms of the throughput per area than any conventional architecture. More precisely, our datapath is approximately 53–72% more efficient than any conventional architecture under the conditions of the three CMOS processes. The results also suggest that our architecture would perform an AES encryption or decryption with the smallest energy. Moreover, the cutoff of an inactive path by a demultiplexer would further reduce the power consumption caused by a dynamic hazard, but this could not be evaluated by the logic synthesis and still remains for the future study.

The performance of the architecture in [87] was relatively lower for our experimental conditions because its critical path includes InvMixColumns to compute $MC^{-1}(K_r)$ and therefore becomes longer than those of other designs. In addition, InvMixColumns over a tower-field is more area-consuming than that over an AES field. This suggests that the architecture in [87] is not suitable

[*9] According to [87], the $GF(2^4)$ inversion in the circuit can be implemented with a $T_{XOR} + 3T_{NAND}$ delay, where T_{XOR} and T_{NAND} are the delays of the XOR and NAND gates, respectively. However, there is no detailed description to realize such a circuit. Therefore, using the best of our knowledge, we described the circuit by a direct mapping based on the PPRM expansion, which is an algebraic normal form frequently used for designing GF arithmetic circuits [96, 125].

for an on-the-fly key scheduling implementation. The architectures in [77,127] have smaller areas than our architecture; however, our architecture has a higher throughput. The increasing ratio of the throughput is larger than that of the circuit area because the architectures in [127] and [77] use InvMixColumns to compute $MC^{-1}(K_r)$ and require several additional selectors, respectively.

The above comparative evaluation was done with ours and some conventional but representative datapaths. There are other previous works focusing on efficiency (i.e., throughput per gate) by round-based architectures. However, such previous works do not provide a concrete implementation and/or exhibit better performance than the abovementioned conventional datapaths. For example, a hardware AES implementation with a short critical path was presented in [101], which employed an RRB to reduce the critical path delay of SubBytes/InvSubBytes and MixColumns/InvMixColumns. However, we could not evaluate the efficiency by ourselves because of the lack of a detailed description. Another AES encryption/decryption architecture with a high throughput was presented in [77]. However, the architecture had a lower throughput/area efficiency compared to the architecture in [127] according to that paper. Moreover, AES architectures that support either encryption or decryption such as in [97, 142] are not evaluated in this subsection.

The design in this section employs a round-based architecture without block-wise parallelism such as pipelining. The modes of operations with block-wise parallelism (e.g., the ECB and CTR modes) are also available owing to the trade-off between the area and the throughput by pipelining [62]. A simple way to obtain a pipelined version of our round datapath is to unroll the rounds and insert pipeline registers between them. The datapath can be further pipelined by inserting registers into the round datapath. Our round datapath can be efficiently pipelined by placing the pipeline register at the output of the inversion with a good delay balance between the inversion and the following circuit. For example, the synthesis results for our round datapath using the area-speed optimization with the NanGate 45-nm standard-cell library indicated that the inversion circuit had a delay of 0.63 ns, and the remainder had a delay of 0.67 ns. As a result, pipelining would achieve a throughput of 17.37 Gbps, which is nearly twice that without pipelining. Thus, our round datapath is also suitable for such a pipelined implementation.

5.4 Efficient DPA-resistant AES hardware

5.4.1 Overview

In this subsection, we first design a compact AES S-box based on GMS and then design a more efficient DPA-resistant AES hardware architecture. The GMS-based AES S-box is designed with a combination of the state-of-the-art GMS construction [122] and the algebraic characteristics of

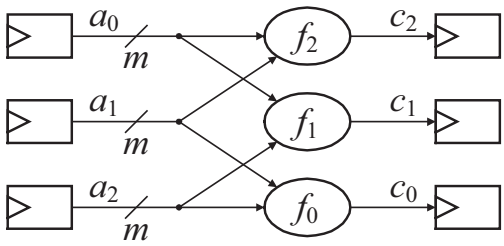


Fig. 5.14 Overview of circuit of function $t = 2$ meeting first-order non-completeness.

AES S-box[*10]. The designed hardware employs a byte-serial architecture commonly used for GMS-based AES in order to tolerate the overheads of circuit area and random number generation [18, 39, 93]. In such architectures, the latency overhead caused by the pipeline registers of GMS is also non-negligible. The conventional works perform SubBytes, ShiftRows, and MixColumns at serial timings despite of pipelined SubBytes, which indicates that an extra latency occurs in every round due to the pipelining, and results in the loss of energy. In contrast, the AES hardware designed in this section exploits a new register-retiming technique to perform the above operations in a partially parallel manner with a modest increase of circuit area. In addition, our AES hardware performs all the operations over tower field for a further reduction of latency (i.e., pipeline-stage). Furthermore, our architecture saves the cost of GMS applied to the key scheduling unit according to the report of [120] which shows that it has no DPA-leaks. With the results of logic synthesis, we confirm that the AES hardware designed in this section has a smaller area and 11–21% lower latency than conventional architectures. In addition, we evaluate the DPA-leakage of our AES hardware implemented on an FPGA. The t-test result shows that there is no obvious first-order DPA-leak from the proposed architecture within 500,000 traces. While this section focuses on the first-order security, the concept of our design can be applied to the higher-order security.

5.4.2 GMS with $d + 1$ input shares

As mentioned before, GMS was presented as a provably-secure countermeasure against DPAs including advanced DPAs exploiting power consumption caused by glitches [83, 92], and can be considered as a subset of GMS. While many conventional countermeasures require to use specific tools and/or libraries (e.g., symmetric layout) [138], the usage of GMS makes it possible to design DPA-resistant hardware with the standard design tools including standard cells and automatic layout. In recent years, some related works on GMS have been reported. They include its extension to higher-order DPAs [17, 122], DPA-resistant cryptographic hardware designs based

[*10] In this subsection, we call TI GMS because TI is a subset of GMS.

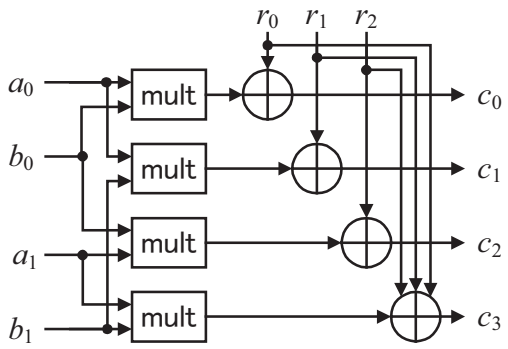


Fig. 5.15 First-order GMS-based $GF(2^m)$ multiplier with two input shares.

on GMS. [18, 39, 93, 120], and GMS-friendly (i.e., GMS-friendly) cryptography where GMS can be efficiently applied to the S-box. [25].

There were two known methods to construct circuits satisfying dth-order non-completeness. The difference of the two is the numbers of input shares. The first construction method was proposed in [103], where the number of input shares is given by $td + 1$, where t is the algebraic degree of circuit function, and the number of output shares is given by $\sigma' = \binom{\sigma}{t}$ (e.g., $\sigma' = \sigma$ when $d = 1$), as described before. For example, Fig. 5.14 shows an overview of a circuit satisfying the first-order non-completeness when $t = 2$ (e.g., an two-input AND gate and a multiplier), where f_φ indicates the function of the circuit that computes c_φ. The number of input shares is 3 $(= 1\times2+1)$, and each f_φ has 2 inputs. Given that c_φ is independent of a_φ in Fig. 5.14, the circuit is thought to be secure under first-order DPAs from the viewpoint of first-order non-completeness. Note that, in this case, the numbers of input and output shares are the same (i.e., $\sigma = \sigma' = 3$); however, they become different when $d \geq 2$.

It was shown that the above GMS with $td + 1$ input shares was useful especially for designing hardware architectures of lightweight ciphers [120] such as PRESENT [19] and LED [57] (as described in Section 1.4.5). This is because such ciphers employ an S-box whose algebraic degree is at most three, which leads to an efficient GMS construction with simple one-stage pipelining where the number of input and output are equal. In addition, all the output shares can be satisfied with the uniformity property in the GMS-based S-box in PRESENT and LED, which means that any on-the-fly random number generation is not required during one block encryption.

The second construction method was recently proposed in [122], where the number of input shares is given by $d + 1$. To construct GMS-based S-box of higher algebraic degree, such as in AES, a multi-stage pipelining and an on-the-fly random number generation are required because it is known that there is no GMS construction satisfying the uniformity property [18, 93]. The GMS with $d + 1$ input shares is more useful for designing compact and efficient hardware architecture

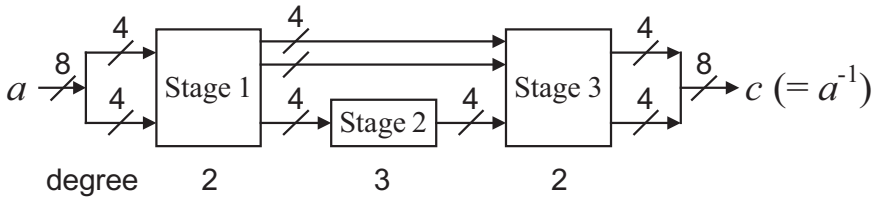


Fig. 5.16 Three-stage expression of tower-field $GF(2^8)$ inversion circuit.

for such a cipher. Actually, in [39], a more compact GMS-based AES hardware was designed with $d+1$ input shares. For example, Fig. 5.15 shows a first-order GMS-based $GF(2^m)$ multiplier with 2 $(= d+1)$ input shares, where a_0, a_1, b_0, and b_1 are the input shares, c_0, c_1, c_2, and c_3 are the output shares, "mult" denotes a nonshared $GF(2^m)$ multiplier, and r_0, r_1, and r_2 are fresh masks for remasking. The multiplier performs $c = a \times b$ under the first-order non-completeness, where $a = a_0 + a_1$, $b = b_0 + b_1$, and $c = c_0 + c_1 + c_2 + c_3$. More precisely, each output share c_j is independent of either a_0 or a_1 (b_0 or b_1), which means the multiplier meets the first-order non-completeness. The number of output share for the GMS is given by at least $(d+1)^t$.

5.4.3 Our design

We first design a GMS-based AES S-box, which mainly determines the performance and security of AES hardware. We then design an efficient byte-serial AES hardware architecture which employs new register-retiming and tower-field arithmetic to reduce the latency without large area overhead.

First-order GMS-based AES S-box

Until now, some GMS-based inversion (i.e., S-box) circuits were proposed in the literature [18, 39, 93]. In order to describe the GMS-based inversion ciruits, Fig. 5.16 shows an abstract block diagram of tower-field inversion consisting of three stages. The algebraic degrees of Stages 1, 2, and 3 are given by two, three, and two, respectively. The major difference of the conventional circuits is the numbers of pipeline-stages and input shares. The inversion circuit in [93] was based on the GMS with $td+1$ input shares and a four-stage pipeline architecture inserting pipeline registers inside Stage 2 in addition to boundaries between Stages 1, 2, and 3. While the inversion circuit in [18] also employed the GMS with $td+1$ input shares, it was based on a three pipeline-stage architecture where Stage 2 was given as a combinational circuit with algebraic degree three. Since the tower-field inversion is efficiently decomposed to three stages in terms of the algebraic expression [96], such three-stage pipeline architecture also makes the GMS-based inversion circuit more efficient. On the other hand, a more compact GMS-based inversion circuit with four-stage

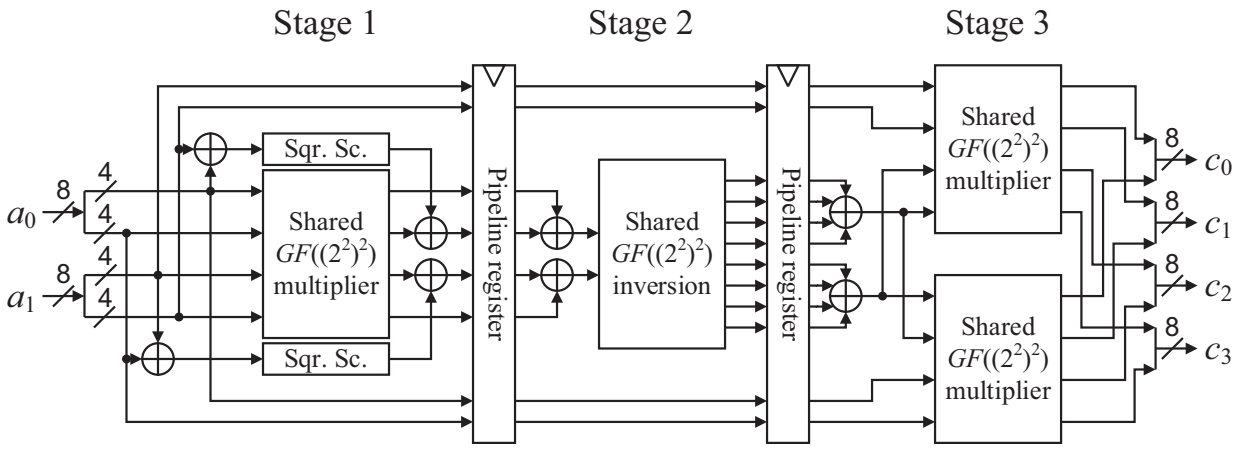


Fig. 5.17 Proposed first-order GMS-based tower-field inversion circuit.

pipelining was designed in [39] on the basis of GMS with $d+1$ input shares. Note here that two more pipeline stages between the inversion and each mapping (i.e., Δ_f and Δ_l) are inserted for GMS-based inversion circuits in order to satisfy dth-order non-completeness.

This section designs a further compact and efficient GMS-based inversion circuit based on a combination of GMS with $d+1$ input shares and the above algebraic characteristics of tower-field inversion. More precisely, the inversion circuit designed here exploits a three-stage pipeline technique similar to [18], and the input of each stage is given by two shares. While the possibility of such a design was mentioned in [39], there was neither description nor evaluation in the literature. Figure 5.17 illustrates the proposed first-order GMS-based S-box which performs $(c_0+c_1+c_2+c_3) = (a_0+a_1)^{-1}$ using three clock cycles where Stages 1, 2, and 3 correspond to those of Fig. 5.1, respectively. Note here that paths with fresh masks for remasking are omitted for simplicity, but Stages 1, 2, and 3 require 12-, 28-, and 24-bit random numbers for remasking [39], respectively. The block "Shared $GF((2^2)^2)$" multiplier denotes the GMS-based multiplier shown in Fig. 5.15. The block "Sqr. Sc." performs squaring and scaling operations over $GF((2^2)^2)$ in a nonshared manner. Finally, the block "Shared $GF((2^2)^2)$ inversion" performs $GF((2^2)^2)$ inversion under the first-order non-completeness by a combinational circuit. As stated in [39], the number of output shares is given by $(d+1)^t$, which would have an impact on the circuit area. However, each subfunction of Shared $GF((2^2)^2)$ inversion can be efficiently factored and implemented using OR (or NOR) gates, which makes Stage 2 smaller. An example of logical expression for Shared $GF((2^2)^2)$ inversion is described in Appendix.

Our GMS-based S-box was evaluated with Synopsys Design Compiler version D-2010.03 and TSMC 65-nm standard CMOS technology. Table 5.7 shows the synthesis results of the conventional and our first-order GMS-based S-boxes, where Area denotes circuit area estimated by

Table 5.7 Performance evaluation of first-order GMS-based AES S-boxes

	Area [GE]		Clock cycles		Area-Latency	Randomness
	`compile`	`_ultra`	S-box	Inversion	product	[bit]
Moradi et al. [93]	No data	4,244	5	4	21,220	44
Bilgin et al. [18]	2,835	2,224	4	3	8,896	32
Cnudde et al. [39]	1,977	1,872	6	4	11,232	54
This study	1,425	1,342	5	3	6,710	64

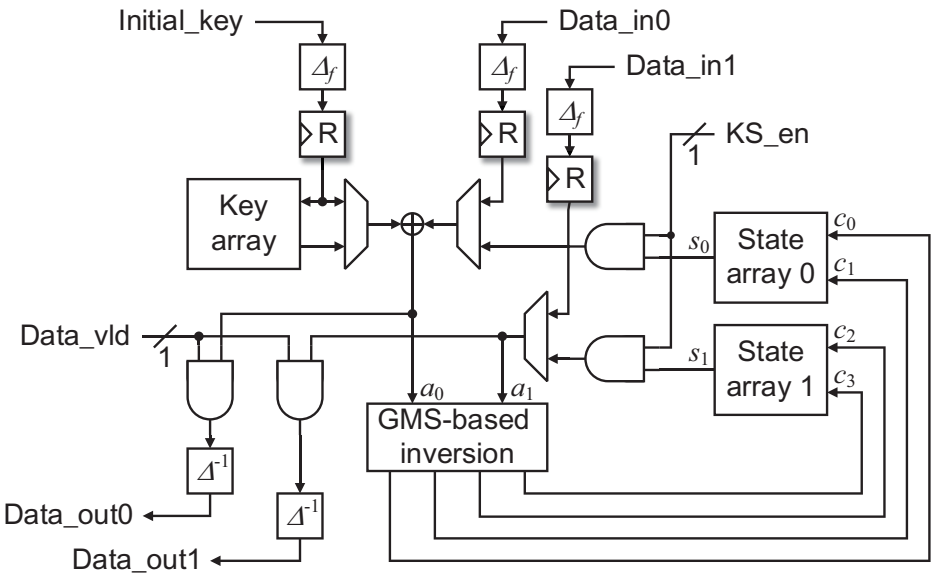


Fig. 5.18 Proposed byte-serial AES hardware architecture.

two-input NAND equivalent gate size (GE: Gate Equivalents), Clock cycles denotes the number of clock cycles required to perform S-box and inversion operations, Area-Latency product denotes the product of Area and Clock cycles (of S-box), and Randomness denotes the number of random bits required in a clock cycle. The columns `compile` and `_ultra` in Area were obtained by the commands `compile` and `compile_ultra`, respectively. For comparison, the values of conventional methods were derived from a table in [39]. We can confirm that our S-box achieved the smallest area without latency overhead while more randomness is consumed. In other words, our S-box is especially effective if random number generation is not critical.

New byte-serial AES hardware architecture

Figure 5.18 shows the proposed byte-serial AES hardware architecture with the above 1st-order GMS-based inversion circuit. The proposed architecture basically has an eight-bit datapath. In Fig. 5.18, the arrow without bit-width information denotes an eight-bit data flow. The blocks

"State array" and "Key array" denote register arrays to store the intermediate values and round keys, respectively. The block "GMS-based inversion" denotes the GMS-based inversion circuits. Note that paths of random numbers for remasking are omitted. The blocks "Δ_f" and "Δ^{-1}" perform the isomorphic mapping and inverse mapping, respectively. Note that the output of Δ_f should be stored into the register "R" for satisfying non-completeness in the following inversion. Two State arrays are required to store $(d+1)$-shared intermediate value. On the other hand, since it is known that the key scheduling function has no DPA-leaks [120], we do not apply GMS to the Key array. SubWord in the key scheduling function is performed by a nonshared S-box in [33]. Here, the S-box should be gated using AND gates to reduce dynamic power consumption, where the gating is controlled by one-bit signal "KS_en." Note that though the SubWord can also be performed using GMS-based inversion, we use a distinct non-pipelined S-box to suppress the latency due to the pipelined Inversion. Note also that our architecture would be designed with a higher-order GMS-based inversion circuit in the similar manner.

Our architecture performs all operations (i.e., AddRoundkey, SubBytes, Shift-Rows, Mix-Columns, and key scheduling) over the tower field to reduce the number of pipeline stages (i.e., latency). Therefore, the isomorphic mappings are performed only at the input and output of the circuit. In addition, a new register-retiming technique, where the affine transformation in SubBytes is performed in State array, is introduced to further reduce the latency of the pipeline architecture. Consequently, the latency for two clock cycles are reduced by the above architectural design.

Figure 5.19 shows the internal structure of State array, which mainly consists of eight-bit registers and logic circuits for affine transformation and MixColumns. (Key array is omitted because it can be implemented in the same manner as [93].) We applied the above register-retiming techniques to our State array. The State array in Fig. 5.19 is different from that of [93] because of the following three features: (i) the SubBytes of the last byte and ShiftRows are simultaneously performed in one clock cycle, (ii) MixColumns of the second and third colomns and the next round SubBytes are executed in parallel, and (iii) the SubBytes of the last four bytes and MixColumns are simultaneously performed. While the conventional State array has distinct paths only for ShiftRows, our State array performs ShiftRows and one-byte shift simultaneously using a unified path indicated in gray by allowed lines in Fig. 5.19 thanks to the feature (i). The output of S2 is given to GMS-based inversion instead of S0 according to the feature (ii). Finally, by the feature (iii), affine transformation is performed at "Aff" in parallel during the byte shift (for the 0th–11th bytes) or MixColumns (for 12th–15th bytes). It is possible to unify the affine transformation and the MixColumns in the same manner as the previous section; however, we do not apply the unification technique to our architecture it does not contribute to the increase of efficiency (i.e., the product of circuit area and latency). Note here that MixColumns should be gated as well as S-box

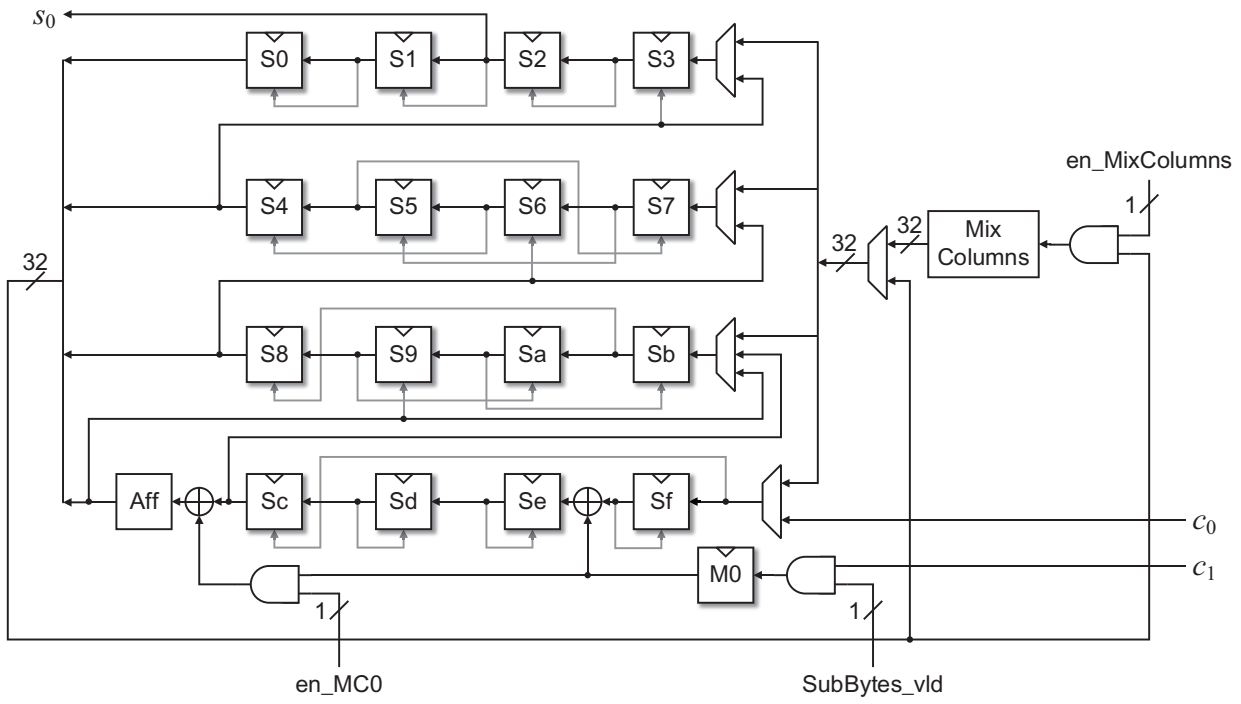


Fig. 5.19 State array.

for SubWord.

Figure 5.20 shows the timing diagrams of (a) conventional [93] and (b) our byte-serial AES hardware architectures, where "SubBytes l," "Inversion l," "Aff l," "SR," "MC ε," and "KS ε" denote the lth SubBytes, lth byte inversion, lth byte affine transformation, ShiftRows, εth column MixColumns, and εth byte SubWord in key scheduling, respectively. The blocks in gray denote operations of the previous or next round executed in parallel to the round of interest. From Fig. 5.20, we can confirm that our architecture achieves 20 clock cycles for one round operation while the conventional one requires 25 clock cycles because of the effect of the above resister-retiming and tower-field arithmetic techniques.

5.4.4 Evaluation

Performance evaluation

To conduct a performance evaluation, we synthesized our AES hardware with Synopsys Design Compiler and TSMC 65 nm standard CMOS as above. Table 5.8 shows the synthesis result of our AES hardware in Fig. 5.18. For comparison, Tab. 5.8 also shows those of the conventional ones derived in the same manner as Tab. 5.7. Note that Area of This study includes the area required for all the components in Fig. 5.18 and a control unit implemented with a 10-bit shift register and a five-bit counter. From Tab. 5.8, we confirmed that our hardware achieved 11–21% lower

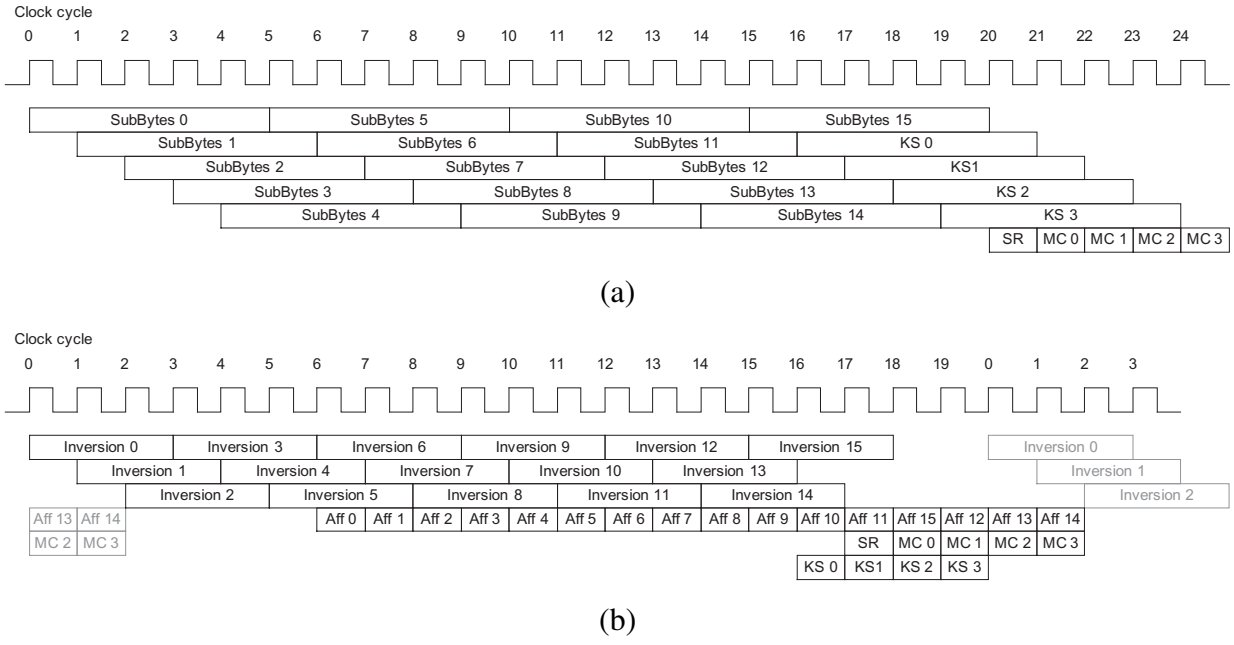


Fig. 5.20 Timing diagrams of (a) Conventional [93] and (b) proposed byte-serial AES hardware architectures.

latency than the conventional ones. Though additional path selectors for register-retiming and MixColumns over tower fields would have an influence on the circuit area [33], our AES hardware achieved the smallest circuit area because of the proposed S-box and nonshared Key array. This also indicates that the circuit area can be further reduced by performing ShiftRows in a distinct clock cycle and/or replacing the tower-field MixColumns with AES-field one in exchange for increasing 10 clock cycles. Table 5.8 also shows the estimated power consumption based on gate-level timing simulation, where Power-Latency product indicates the product of Power and Clock cycles. The values of the conventional works were calculated using a table in [93]. The scaled values of Power and Power-Latency product in the parentheses are derived by dividing the original ones by the square of process rate (i.e., $(180/65)^2$). (The architecture in [93] was synthesized with 180 nm standard CMOS.) Note that it is quite difficult to compare power consumption estimation of hardware architectures in a fair manner, which heavily depends on the used technology and estimation method. However, the results roughly indicate that the lower latency would directly lead to lower energy of one block encryption. Thus, we confirmed the effectiveness of our design.

Experimental evaluation of DPA-leakage

The DPA-resistance capability of our S-box was evaluated with an experiment using an FPGA implementation.

Table 5.8 Performance of AES hardware architecture based on first-order GMS

	Area [GE]		Clock cycles	Area-Latency product	Power [μW]	Power-Latency product
	`compile`	`_ultra`				
Moradi et al. [93]	11,114	11,031	266	2,956 K	24.12 (3.14)	6,415 (835)
Bilgin et al. [18]	8,119	7,282	246	1,997 K	No data	
Cnudde et al. [39]	6,681	6,340	276	1,844 K	No data	
This study	6,321	6,053	219	1,376 K	3.06	670

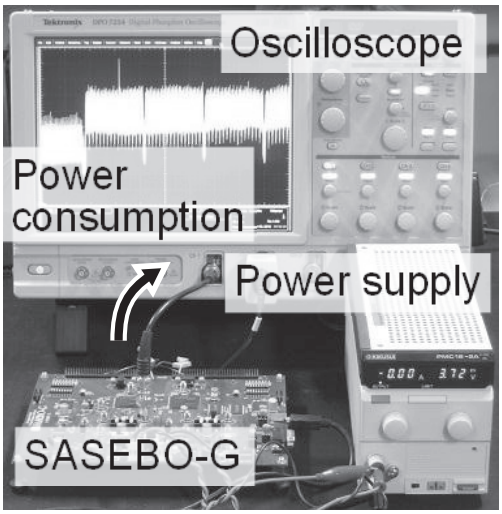


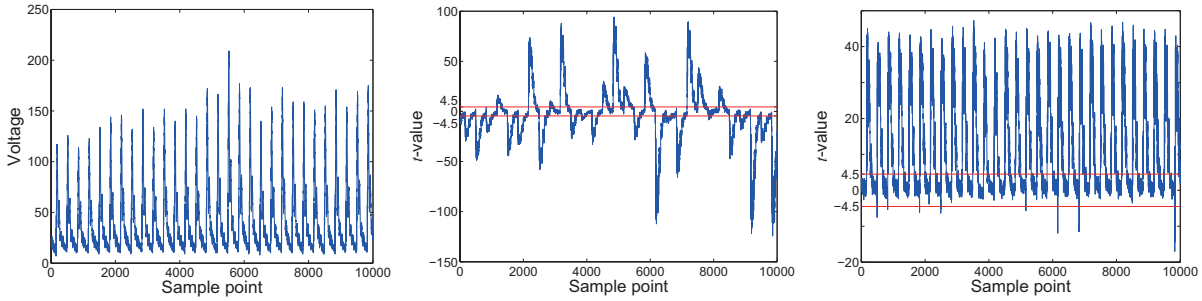
Fig. 5.21 Experimental setup.

Figure 5.21 shows the experimental setup consisting of a Side-Channel Attack Standard Evaluation Board (SASEBO-G) [3] and an oscilloscope Tektronix DPO7254. The designed AES hardware was implemented on an FPGA (Xilinx Virtex II Pro) on the SASEBO-G, and the power variation was sampled with the sampling rate of 1GS/s.

We evaluated the resistance and vulnerability of the AES hardware by Test Vector Leakage Assessment (TVLA) based on Welch's t-test (a.k.a. non-specific t-test) [130]. The TVLA examines t-values which indicate the existence of dth-order DPA-leakage exploitable by the attackers.

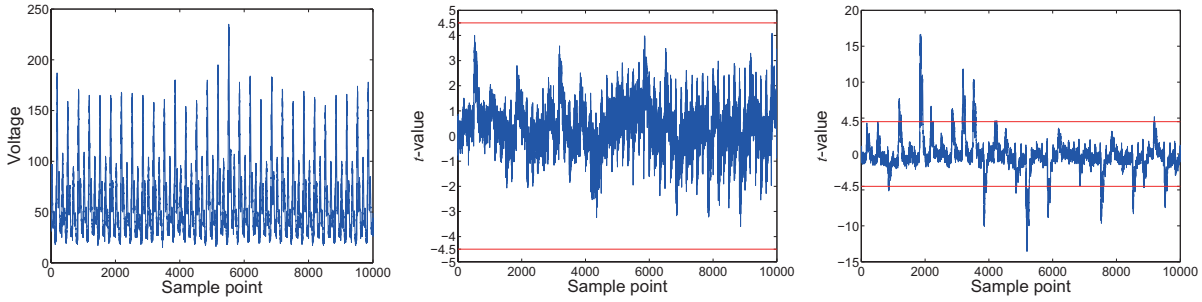
Figures 5.22(a) and 5.23(a) show examples of power traces at around the nineth with and without a pseudo random number generator (PRNG) implemented on the FPGA, respectively. When the PRNG is turned on, the GMS works. We can find the small spikes between the big spikes in Fig. 5.23(a) because AES and PRNG are asynchronously active. Thus, the PRNG would not have a significant impact on the following TVLA result.

Figures 5.22 and 5.23 show the (b) first-order and (c) second-order TVLA results. We used 10,000 and 500,000 traces for Figs. 5.22 and 5.23, respectively. It is known that the absolute



(a) Raw trace. (b) First-order. (c) Second-order.

Fig. 5.22 Measurement and TVLA results without PRNG.



(a) Raw trace. (b) First-order. (c) Second-order.

Fig. 5.23 Measurement and TVLA results with PRNG.

t-value of more than 4.5 indicates a high confidence in the existence of exploitable DPA-leakage. The results suggest that our design is resistant to the first-order DPAs under the condition of 500,000 traces by means of the first-order GMS. On the other hand, we can see the second-order leakage in both Figs. 5.22 and 5.23 due to the limitation of the first-order GMS. Thus, we could validate the DPA resistance of the designed AES hardware n the experimental condition with 500,000 traces.

5.5 Conclusion

This chapter designed highly efficient AES hardware. Since the designed hardware utilizes redundant GF arithmetic, higher-degree functions, logic-level optimizations, and/or pipelining, the conventional methods including the existing GF-ACG cannot be applied to them. On the ohter hand, the proposed formal design method can design and verify them. Thus, the proposed formal design method is useful even for designing such state-of-the-art and high-performance cryptographic hardware.

6 Conclusion

This dissertation studied a formal design of cryptographic hardware from three viewpoints of theory, implementation, and application.

Chapter 2 introduced the basics on cryptographic hardware design. We described that modern cryptographic algorithms are closely related to GF arithmetic, and therefore the design of GF arithmetic circuits are quite important in designing cryptographic hardware. We also described the conventional design and verification methods of arithmetic circuits, and their difficulties in designing and verifying GF arithmetic circuits. Finally, we introduced the existing formal design method based on GF-ACG, and described the issue on its application to practical cryptographic hardware.

Chapter 3 presented a new formal design methodology of cryptographic hardware. We proposed the new GF-ACG which can handle a wider variety of GF arithmetic circuits including practical ones based on redundant GF representations and with pipelining. We then introduced two equivalence checking methods for GF-ACG based on ND for the first order predicate logic and a PPRM expansion, respectively. They can be efficiently applied to GF arithmetic circuits which cannot be completely verified by the conventional algebraic formula evaluation method.

Furthermore, we presented a new verification algorithm combining the two equivalence checking methods with the conventional algebraic method. We demonstrated the effectiveness and efficiency of the proposed methodology through its applications to parallel multipliers, AES hardware, and tamper-resistant cryptographic hardware.

Chapter 4 produced an automatic generation systems of GF arithmetic circuits for cryptographic hardware. The proposed system could generate verified multipliers over $GF(p^m)$, where $p = 2, 3, 5, 7, 11$ and $2 \leq m \leq 256$. In addition, the system could also generate SCA-resistant GF multipliers based on GMS and verify the SCA resistance property formally by newly proposed algorithms proposed in this dissertation. The performance of the proposed system was evaluated through experimental multiplier generations. As a result, we confirmed that the proposed system could synthesize more than 10,000 GF multipliers including large ones with 256-bit inputs within a practical time, and the proposed system would be useful for designing practical cryptographic hardware such as symmetric key ciphers, ECC, and PBC.

Chapter 5 designed highly efficient AES hardware as applications of the propose design methodology. We first designed a highly efficient $GF(2^8)$ inversion circuit and an AES S-box based on a combination of redundant and non-redundant GF representations. We then designed a high throughput/gate AES hardware compressing encryption and decryption detapaths. The designed hardware architecture could be applied and useful to other modern ciphers because it exploits datapath optimization techniques. We also designed an SCA-resistant AES hardware based on GMS, which achieves a higher efficiency than the conventional ones by the optimization techniques similar to the above. The above hardware can be designed using the proposed formal design methodology while they utilized redundant GF arithmetic, higher-degree function, logic-level optimization, and/or pipelining. Thus, we confirmed that the formal design methodology would be useful for such state-of-the-art and practical cryptographic hardware design.

Although this dissertation focused on AES as applications, our methodology can be applied to other cryptographic algorithms based on GF arithmetic. Further applications to other ciphers would remain in the future work. In addition, our methodology would be useful for designing cryptographic algorithm because the proposed methodology uses GF equations to representing GF arithmetic algorithms. Moreover, it is also desirable to develop cryptographic hardware resistant to implementation attacks other than SCAs (e.g., FIAs) using the proposed methodology. The circuit description using GF equations would also be helpful for formalization and verification of resistance against other implementation attacks.

Bibliography

[1] Arithmetic module generator for GF parallel multipliers, http://www.aoki.ecei.tohoku.ac.jp/arith/gfamg/

[2] Risa/Asir (Kobe distribution) download page, http://www.math.kobe-u.ac.jp/Asir/asir.html

[3] Side-channel attack standard evaluation board (SASEBO), http://www.rcis.aist.go.jp/special/SASEBO

[4] NanGate FreePDK15 open cell library (Jan 2016), http://www.nangate.com/?page_id=2328

[5] NanGate FreePDK45 open cell library (Jan 2016), http://www.nangate.com/?page_id=2325

[6] Al Fardan, Nadhem J.and Paterson, K.G.: Lucky thirteen: Breaking the TLS and DTLS record protocols. In: IEEE Symposium on Security and Privacy (S&P). pp. 526–540. IEEE (2013)

[7] Aoki, K., Ichikawa, T., Kanda, M., Matsui, M., Moriai, S., Nakajima, J., Tokita, T.: *Camallia*: A 128-bit block cipher suitable for multiple platforms—design and analysis. In: Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 2012, pp. 39–56. Springer (2001)

[8] Bardet, M.: On the complexity of a Gröbner basis algorithm. Algorithms Seminar 2002–2004 38, 85–92 (2005)

[9] Barreto, P.S.L.M., Lynn, B., Scott, M.: Efficient implementation of pairing-based cryptosystems. Journal of Cryptology 17, 321–334 (2004)

[10] Bassham III, L.E.: The advanced encryption standard algorithm validation suite. N.I.S.T. Information Technology Laboratory, Computer Security Division (2002)

[11] Beaulieu, R., Treatman-Clark, S., Shors, D., Weeks, B., Smith, J., Wingers, L.: The SIMON and SPECK lightweight block ciphers. In: Design Automation Conference (DAC). pp. 1–6. ACM/EDAC/IEEE, IEEE (2015)

[12] Belaïd, S., Coron, J.S., Fouque, P.A., Gérard, B., Kammerer, J.G., Prouff, E.: Improved side-channel analysis of finite-field multiplication. In: Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Scienece, vol. 6917, pp. 395–415 (2015)

[13] Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. Journal of Cryptographic Engineering 2, 77–89 (2012)

[14] Biham, E., Carmeli, Y., Shamir, A.: Bug attacks. In: Advances in Cryptology—CRYPTO 2008. Lecture Notes in Computer Science, vol. 5157, pp. 221–240. Springer (2008)

[15] Biham, E., Shamir, A.: Differential Cryptanalysis of the Data Encryption Standard. Springer (1993)

[16] Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Advances in Cryptology—CRYPTO 1997. Lecture Notes in Computer Science, vol. 1294, pp. 513–525 (1997)

[17] Bilgin, B., Gierlichs, B., Nikov, V., Rijmen, V.: Higher-order threshold implementations. In: Advances in Cryptology—ASIACRYPT 2014. Lecture Notes in Computer Science, vol. 8874, pp. 326–343. Springer (2014)

[18] Bilgin, B., Gierlichs, B., Nikova, S., Nikov, V., Rijmen, V.: Trade-offs for threshold implementations illustrated on AES. IEEE Transactions on Computer-Aided Design of Integrated and Systems 34(7), 1188–1200 (2015)

[19] Bogdanov, A., Knudsen, L., Leander, G., Paar, C., Poschmann, A., Robshaw, M., Seurin, Y., Vikkelsoe, C.: PRESENT: An ultra-lightweight block cipher. In: Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 4727, pp. 450–466. Springer (2007)

[20] Boneh, D., DeMillo, R., Lipton, R.: On the importance of checking cryptographic protocols for faults. In: Advances in Cryptology—EUROCRYPTO 1997. Lecture Notes in Computer Science, vol. 1223, pp. 37–51 (1997)

[21] Boneh, D., Franklin, M.: Identity-based encryption from the weil pairing. Advances in Crypto;pgy—CRYPTO '01, Lecture Notes in Computer Science 2139, 213–229 (2001)

[22] Boneh, D., Di Cresenzo, G., Ostrovsky, R., Persiano, G.: Public key encryption with keyword search. In: Advances in Cryptology—EUROCRYPT 2004. Lecture Notes in Computer Science, vol. 3027, pp. 506–522. Springer (2004)

[23] Boneh, D., Lynn, B., Shacham, H.: Short signatures from the Weil pairing. Journal of Cryptology 17, 297–319 (2004)

[24] Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E.B., Knezevic, M., Knudsen, L.R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S.S., Yalçin, T.:

PRINCE—a low-latency block cipher for pervasive computing applications. In: Advances in Cryptology–ASIACRYPT 2013. LNCS, vol. 7658, pp. 208–225 (2012)

[25] Boss, E., Grosso, V., Güneysu, T., Leander, G., Moradi, A., Schneider, T.: Strong 8-bit Sboxes with efficient masking in hardware. In: International Conference on Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 9813, pp. 171–193. Springer (2016)

[26] Boyer, J., Matthews, P., Peralta, P.: Logic minimization techniques with applications to cryptology. Journal of Cryptology 47, 280–312 (2013)

[27] Boyer, J., Peralta, R.: A small depth-16 circuit for the AES S-box. In: Information Secutiry and Privacy Research. IFIP Advances in Information and Communication Technology, vol. 376, pp. 287–298. Springer (2012)

[28] Bryant, R.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers C-35(8), 677–691 (Aug 1986)

[29] Bryant, R.E.: Graph-based algorithms for Boolean function manipularion. IEEE Transsactions on Computers C-35(8), 677–691 (1986)

[30] Bryant, R.E., Chen, Y.A.: Verifcation of arithmetic circuits with binary moment diagrams. In: 32nd Design Automation Conf. pp. 535–541. IEEE/ACM (1995)

[31] Buchberger, B.: Some properties of Gröbner-bases for polynomial ideals. SIGSAM Bull. 10(4), 19–24 (1976)

[32] C. Kocher, P.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Advances in Cryptology—CRYPTO 1996. Lecture Notes in Computer Science, vol. 1109, pp. 104–113. Springer (1996)

[33] Canright, D.: A very compact S-box for AES. In: Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 3659, pp. 441–455. Springer (2005)

[34] Canright, D., Batina, L.: A very compact "perfectly masked" S-box for AES (corrected). IACR Cryptology ePrint Archive 2009, 11 (2009)

[35] Canright, D.: Canright web page, \url{http://faculty.nps.edu/drcanrig/}

[36] Chen, Y., Bryant, R.: ACV: an arithmetic circuit verifier. Proc. of the 1996 IEEE/ACM Int. Conf. on Computer-Aided Design pp. 361–365 (1997)

[37] Chen, Z., Zhou, Y.: Dual-rail random switching logic: A countermeasure to reduce side channel leakage. In: International Workshop on Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 4249, pp. 242–254. Springer (2006)

[38] Clarke, E., Emerson, E., Sistla, A.: Automatic verification of finite-state concurrent systems

using temporal logic specifications. ACM Transactions on Programming Languages and Systems 8(2), 244–263 (1986)

[39] Cnudde, T.D., Reparaz, O., Bilgin, B., Nikova, S., Nikov, V., Rijmen, V.: Masking AES with $d+1$ shares in hardware. In: International Conference on Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 9813, pp. 194–212. Springer (2016)

[40] Cox, D.A., Little, J.B., O'Shea, D.: Ideals, Varieties, and Algorithms. Springer-Verla, 2nd edn. (1996)

[41] Cox, D.A., Little, J.B., O'Shea, D.: Using Algebraic Geometry, Graduate Texts in Mathematics, vol. 185. Springer-Verlag (1998)

[42] Cryptographic competitions: Caesar: Competition for authenticated encryption: Security, applicability, and robustness (2016), `https://competitions.cr.yp.to/caesar.html`

[43] Cryptography Research and Evaluation Commitees (CRYPTREC): CRYPTREC Homepage (2017), `\url{http://www.cryptrec.go.jp/english/index.html}`

[44] Ding, J., Yang, B.Y.: Multivariate public key cryptography. In: Bernstein, D.J., Buchmann, J., Dahmen, E. (eds.) Post-Quantum Cryptography. pp. 193–241. Springer

[45] Doulcier-Verdier, M., Dutertre, J.M., Fournier, J., Rigaud, J.B., Robisson, B., Tria, A.: A side-channel and fault-attack resistant AES circuit working on duplicated complemented values. In: Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International. pp. 274–276. IEEE (2011)

[46] Drechsler, R. (ed.): Advanced Formal Verification. Kluwer Academic Publishers (2004)

[47] Drolet, G.: A new representation of elements of finite fields $GF(2^m)$ yielding small complexity arithmetic circuits. IEEE Transactions on Computers 47(9), 938–946 (1997)

[48] Duc, A., Dziembowski, S., Faust, S.: Unifying leakage models: From probing attacks to noisy leakage. In: Advances in Cryptology—EUROCRYPT 2014. Lecture Notes in Computer Science, vol. 8441, pp. 423–440. Springer (2014)

[49] Duong, T., Rizzo, J.: Here come the $\oplus$ ninjas (2011), `\url{https://www.nist.gov/}`

[50] Duursma, I., Lee, H.S.: Tate pairing implementation for hyperelliptic curves $y^2 = x^p - x + d$. In: Advances in Cryptology—ASIACRYPT 2003. Lecture Notes in Computer Science, vol. 2894, pp. 111–123. Springer (2003)

[51] Duursma, I., Sakurai, K.: Efficient algorithms for the Jacobian variety of hyperelliptic curves $y^2 = x^p - x + 1$ over a finite field of odd characteristic p. In: Coding Theory, Cryptography and Related Areas. pp. 73–89. Springer (1998)

[52] Dworlin, M.: NIST special publication 800-38D—recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. Tech. rep., National Institute of Standards and Technology (NIST) (2007), http://dl.acm.org/citation.cfm?id=2206251

[53] Ekdahl, P., Johansson, T.: A new version of the stream cipher SNOW. In: Selected Areas in Cryptography (SAC). Lecture Notes in Computer Science, vol. 2595, pp. 47–61. Springer (2001)

[54] Gao, S.: Normal bases over finite fields. Citeseer (1993)

[55] Goyal, V., Pandey, O., Sahai, A., Waters, B.: Attribute-based encryption for fine-grained access control of encrypted data. In: ACM conference on Computer and Communication Security (ACM-CCS). pp. 89–98. ACM (2006)

[56] Grabher, P., Page, D.: Hardware acceleration of Tate pairing in characteristic three. In: International Workshop on Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 3659, pp. 398–411. Springer (2005)

[57] Guo, J., Peyrin, T., Poschmann, A., Robshaw, M.: The LED block cipher. In: Cryptographic Hardware and Embedded Systems (CHES). Lecture Note in Computer Science, vol. 6917, pp. 326–341 (2011)

[58] Halbutogullari, A., Koç, C.: Mastrovito multiplier for general irreducible polynomials. IEEE Transactions on Computers 49(5), 503–518 (May 2000)

[59] Hammad, I., El-Sankary, K., El-Masry, E.: High-speed AES encryptor with efficient merging techniques. IEEE Embedded Systems Letters 2, 67–71 (2010)

[60] Hirotomo, M., Mouri, K., Morii, M.: Generalized polynomial ring representation over $GF(2^m)$ and its application. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences (Japanese Edition) J89-A(10), 790–800 (2006)

[61] Hitachi Ltd.: MUGI pseudorandom numbner generator specification ver 1.2 (2001), http://www.hitachi.com/rd/yrl/crypto/mugi/mugi_spe.pdf

[62] Hodjat, A., Verbauwhede, I.: Area-throughput trade-offs for fully pipelined 30 to 70 Gbits/s AES processors. IEEE Transactions on Computers 50(4), 366–372 (2006)

[63] Homma, N., Saito, K., Aoki, T.: A formal approach to designing cryptographic processors based on $GF(2^m)$ arithmetic circuits. IEEE Transactions on Information Forensics and Security 7(1), 3–13 (Feb 2012)

[64] Homma, N., Saito, K., Aoki, T.: Toward formal design of practical cryptographic hardware based on Galois field arithmetic. IEEE Transactions on Computers 63(10), 2604–2613 (Jun 2014)

[65] Ishai, Y., Sahai, A., Wagner, D.: Private circuits: Securing hardware againgst probing

attacks. In: Advances in Cryptology—CRYPTO 2003. Lecture Notes in Computer Science, vol. 2729, pp. 461–481. Springer (2003)

[66] Itoh, T., Tsujii, S.: A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. Information and Computation 78, 171–177 (1988)

[67] Jeon, Y., Kim, Y., Lee, D.: A compact memory-free architecture for the AES algorithm using resource sharing methods. Journal of Circuits, Systems, and Computers 19(5), 1109–1130 (2010)

[68] Katti, R., Brennan, J.: Low complexity multiplication in a finite field using ring representation. IEEE Transactions on Computers 52(4), 418–427 (2003)

[69] Katz, J., Sahai, A., Waters, B.: Predicate encryption supproting disjunctions, polynomial equations, and inner products. In: Advances in Cryptology—EUROCRYPT 2008. Lecture Notes in Computer Science, vol. 4965, pp. 146–162. Springer (2008)

[70] Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation 48(177), 203–209 (1987)

[71] Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Advances in Cryptology—CRYPTO 1999. Lecture Notes in Computer Science, vol. 1666, pp. 388–397. Springer (1999)

[72] Koren, I.: Computer arithmetic algorithms 2nd Edition. A K Peters (2001)

[73] Lamport, L.: "sometime" is sometimes "not never": on the temporal logic of programs. In: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles Of Programming Languages. pp. 174–185. ACM, New York, NY, USA (1980)

[74] Langley, A.: RFC 7539 - ChaCha20 and Poly1305 for IETF protocols - IETF tools (2015), `\url{https://tools.ietf.org/html/rfc7539}`

[75] Lee, E., Lee, H.S., Lee, Y.: Eta pairing computation on general divisors over hyperelliptic curves $y^2 = x^p - x + d$. Journal of Synbolic Computation 43, 452–474 (2005)

[76] Lin, S.Y., Huang, C.T.: A high-throughput low-power AES cipher for network applications. In: The 12th Asia and South Pacific Design Automation Conference (ASP-DAC 2007). pp. 595–600. IEEE (2007)

[77] Liu, P.C., Chang, H.C., Lee, C.Y.: A 1.69 Gb/s area-efficient AES crypto core with compact on-the-fly key expansion unit. In: 41th European Solid-State Circuits Conference (ESSCIRC 2009). pp. 404–407. IEEE (2009)

[78] Lutz, A., Treichler, J., Gürkaynak, F., Kaeslin, H., Basler, G., Erni, A., Reichmuth, S., Rommens, P., Oetiker, P., Fichtner, W.: 2Gbit/s hardware realizations of RIJNDAEL and SERPENT: A comparative analysis. In: Workshop on Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 2523, pp. 144–158.

Springer (2002)

[79] Lv, J., Kalla, P.: Formal verification of Galois field multipliers using computer algebra techniques. In: VLSI Design (VLSID), 2012 25th International Conference on. pp. 388–393. IEEE (2012)

[80] Lv, J., Kalla, P., Enescu, F.: Verification of composite Galois field multipliers over $GF((2^m)^n)$ using computer algebra techniques. In: IEEE International High Level Design Validation and Test Workshop (HLDVT). pp. 136–143. IEEE (2011)

[81] Lv, J., Kalla, P., Enescu, F.: Efficient Gröbner basis reductions for formal verification of Galois field multipliers. In: Proceedings of the Conference on Design, Automation and Test in Europe. pp. 899–904. EDA Consortium (2012)

[82] M. Jabir, A., K. Pradhan, D.: A graph-based unified technique for computing and representing coefficients over finite fields. IEEE Transactions on Computers 56(8), 1119–1132 (Aug 2007)

[83] Mangrad, S., Pramstaller, N., Oswald, E.: Successfully attacking masked AES hardware implementations. In: Workshop on Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 3659, pp. 157–171. Springer (2005)

[84] Manna, Z., Pnueli, A.: Temporal verification of reactive systems (1987)

[85] Massey, J., Omura, J.: Computational method and apparatus for finite field arithmetic (1986), uS Patent

[86] Mathew, S., Satpathy, S., Suresh, V., Anders, M., Himanshu, K., Amit, A., Hsu, S., Chen, G., Krishnamurthy, R.K.: 340 mV–1.1V, 289 Gbps/W, 2090-gate nanoAES hardware accelerator with area-optimized encrypt/decrypt $GF(2^4)^2$ polynomials in 22 nm tri-gate CMOS. IEEE Journal of Solid-State Circuits 50, 1048–1058 (2015)

[87] Mathew, S.K., Sheikh, F., Kounavis, M.E., Gueron, S., Agarwal, A., Hsu, S.K., Himanshu, K., Anders, M.A., Krishnamurthy, R.K.: 53 Gbps native $GF(2^4)^2$ composite-field AES-encrypt/decrypt accelerator for content-protection in 45 nm high-performance microprocessors. IEEE Journal of Solid-State Circuits 46, 767–776 (2011)

[88] Matsui, M.: The first experimental cryptoanalysis of the Data Encryption Standard. In: Advances in Cryptology—CRYPTO 1994. Lecture Notes in Computer Science, vol. 839, pp. 1–11. Springer (1994)

[89] McGrew, D.A., Viega, J.: The Galois/Counter Mode of operation (GCM) (2005), `http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/gcm-revised-spec.pdf`

[90] Micciancio, D., Regev, O.: Lattice-based cryptography. In: Bernstein, D.J., Buchmann, J., Dahmen, E. (eds.) Post-Quantum Cryptography. pp. 147–191. Springer

[91] Moradi, A., Alexander, W.: Assessment of hiding the higher-order leakages in hardware—What are the achievements versus overheads? In: Workshop on Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 9293, pp. 453–474. Springer (2015)

[92] Moradi, A., Mischke, O., Eisenbarth, T.: Correlation-enhanced power analysis collision attack. In: Workshop on Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 6225, pp. 125–139. Springer (2010)

[93] Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the limits: A very compact and a threshold implementation of AES. In: Advances in Cryptology—EUROCRYPT 2011. Lecture Notes in Computer Science, vol. 6632, pp. 59–88. Springer (2011)

[94] Morioka, S., Katayama, Y., Yamane, T.: Towards efficient verification of arithmetic algorithms over Galois fields $GF(2^m)$. Proc. 13th Conf. on Computer Aided Verification LNCS 2102, 465–477 (2001)

[95] Morioka, S.: Verification of AES component H/W optimization using a PPRM-based formal method. In: Computer Security Symposium 2012. pp. 765–772. No. 3, CSEC (2012), Japanese edition

[96] Morioka, S., Satoh, A.: An optimized S-Box circuit architecture for low power AES design. In: Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 2523, pp. 172–186. Springer (2002)

[97] Morioka, S., Satoh, A.: A 10 Gbps full-AES crypto design with a twisted-BDD S-Box architecture. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 12, 686–691 (2004)

[98] Mukhopadhyay, D., Sengar, G., Chowdhury, R.D.: Hierarchical verification of Galois field circuits. IEEE Transaction on Computer Aided Design of Integraded Circuits and Systems 26(10), 1893–1898 (2007)

[99] Mullin, R.C., Onyszchuk, I.M., Vanstone, S.A., Wilson, R.M.: Optimal normal bases in $GF(p^n)$. Discrete Applied Mathematics 22(2), 149–161 (1989)

[100] National Institute of Standards and Technology (NIST): Advanced Encryption Standard (AES) FIPS Publication 197 (2001), `\url{http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf}`

[101] Nekado, K., Nogami, Y., Iokibe, K.: Very short critical path implementation of AES with direct logic gates. In: Advances in Information and Computer Security (IWSEC). Lecture Notes in Computer Science, vol. 7631, pp. 51–68. Springer (2012)

[102] New European Schemes for Signatures, Integrity, and Encryption (NESSIE): The homepage of the NESSIE project (2017), `\url{https://www.cosic.esat.`

kuleuven.be/nessie/}

[103] Nikova, S., Rijmen, V., Schläffer, M.: Secure hardware implementation of nonlinear functions in the presence of glitches. Journal of Cryptology 24, 292–321 (2011)

[104] NIST: National Institute of Standards and Technology—U.S. Department of Commerce (2017), \url{https://www.nist.gov/}

[105] Nogami, Y., Nekado, K., Toyota, T., Hongo, N., Morikawa, Y.: Mixed bases for efficient inversion in $\mathbb{F}_{((2^2)^2)^2}$ and conversion matrices of SubBytes of AES. In: Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 6225, pp. 234–247. Springer (2010)

[106] Nogami, Y., Nekado, K., Toyota, T., Hongo, N., Morikawa, Y.: Mixed bases for efficient inversion in $\mathbb{F}_{((2^2)^2)^2}$ and conversion matrices of SubBytes of AES. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences 94(6), 1318–1327 (2011)

[107] Nogami, Y., Saito, A., Morikawa, Y.: Finite extension field with modulus of all-one polynomial field and representation of its elements of for fast arithmetnic operations. IEICE transactions on fundamentals of electronics, communications and computer sciences E86-A(9), 2376–2387 (2003)

[108] Nyberg, K.: Differentially uniform mappings for cryptography. In: Advances in Cryptology—EUROCRYPT 1993. Lecture Notes in Computer Science, vol. 765, pp. 55–64. Springer (1993)

[109] Okamoto, K., Homma, N., Aoki, T.: A hierarchical graph-based approach to generating formally-proofed Galois-field multipliers. In: Workshop on Security Proofs for Embedded Systems (PROOFS). pp. 98–109 (2013)

[110] Okamoto, K., Homma, N., Aoki, T.: A graph-based approach to designing parallel multipliers over Galois fields based on normal basis representations. In: 43th International Symposium on Multiple-Valued Logic (ISMVL 2013). pp. 54–59. IEEE (2015)

[111] Okamoto, K., Homma, N., Aoki, T., Morioka, S.: A hierarchical formal approach to verifying side-channel resistant cryptographic processors. In: Hardware-Oriented Security and Trust (HOST). pp. 76–79. IEEE (2014)

[112] Okamoto, T., Takashima, K.: Fully secure functional encryption with general relations from the decisional linear assumption. In: Advances in Cryptology—CRYPTO 2010. Lecture Notes in Computer Science, vol. 6223, pp. 191–208. Springer (20010)

[113] Omondi, A.R.: Computer Arithmetic Systems: Algorithms, Architecture and Implementations. Prentice Hall (1994)

[114] Oswald, E., Mangard, S., Pramstaller, N., Rijmen, V.: A side-channel analysis resistant

description of the AES S-box. In: Fast Software Encryption. Lecture Notes in Computer Science, vol. 3557, pp. 413–423. Springer (2005)

[115] Page, D., Smart, N.P.: Hardware implementation of finite fields of characteristic three. CHES 2002, Lecture Notes in Computer Science 2523, 529–539 (Aug 2002)

[116] Parhami, B.: Computer Arithmetic: Algorithms and Hardware Designs. Oxford University Press (2000)

[117] Pavlenko, E., Wedler, M., Stoffel, D., Kunz, W., Dreyer, A., Seelisch, F., Greuel, G.M.: STABLE: A new QF-BV SMT solver for hard verification problems combining Boolean reasoning with computer algebra. In: Conference on Design, Automation and Test in Europe (DATE). pp. 1–6. EDA Consortium, IEEE (2011)

[118] Pnueli, A.: The temporal logic of programs. In: Symposium on Foundations of Computer Science. pp. 46–67. IEEE Computer Society Press (1977)

[119] Popp, T., Mangard, S.: Masked dual-rail pre-charge logic: DPA-resistance without routing constraints. In: International Workshop on Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 3659, pp. 172–186. Springer (2005)

[120] Poschmann, A., Moradi, A., Khoo, K., Lim, C.W., Wang, H., Ling, S.: Side-channel resistant crypto for less than 2,300 GE. Journal of Cryptology 24, 322–334 (2011)

[121] Prowitz, D.: NATURAL DEDUCTION—A Proof-Theoretical Study. Almqvist & Wiksell (1965)

[122] Reparaz, O., Bilgin, B., Nikova, S., Gierlichs, B., Verbauwhede, I.: Consolidating masking schemes. In: Advances in Cryptology—CRYPTO 2015. Lecture Notes in Computer Science, vol. 9215, pp. 764–783. Springer (2015)

[123] Rudra, A., Dubey, P.K., Jutla, C.S., Kumar, V., Rao, J., Rohatgi, P.: Efficient Rijndael encryption implementation with composite field arithmetic. In: Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 2162, pp. 171–184 (2001)

[124] Sahai, A., Waters, B.: Fuzzy indentity-based encryption. In: Advances in Cryptology—EUROCRYPT 2005. Lecture Notes in Computer Science, vol. 3494, pp. 457–473. Springer (2005)

[125] Sasao, T.: And-Exor expressions and their optimization. In: Sasao, T. (ed.) Logic Synthesis and Optimization. The Kluwer International Series in Engineering and Computer Science, vol. 212, pp. 287–312. Kluwer Academic Publishers (1993)

[126] Sasao, T.: Representations of logic functions using EXOR operators. In: Representations of discrete functions, pp. 29–54. Springer (1996)

[127] Satoh, A., Morioka, S., Takano, K., Munetoh, S.: A compact Rijndael hardware architecture with S-box optimization. In: Advances in Cryptology—ASIACRYPT 2001. Lecture Notes in Computer Science, vol. 2248, pp. 239–254. Springer (2001)

[128] Satoh, A., Morioka, S., Takano, K., Munetoh, S.: A compact Rijndael hardware architecture with S-box optimization. In: Advances in Cryptology—ASIACRYPT 2001. Lecture Notes in Computer Science, vol. 2248, pp. 239–254. Springer (2001)

[129] Savas, E., Koç, K.C.: Finite field arithmetic for cryptography. IEEE Circuits and Systems Magazine 10(2), 40–56 (Aug 2010)

[130] Schneider, T., Moradi, A.: Leakage assesment methodology—A clear roadmap for side-channel evaluations. In: Workshop on Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 9293, pp. 495–513. Springer (2015)

[131] Shibutani, K., Isobe, T., Hiwatiri, H., Mitsuda, A., Akishita, T., Shirai, T.: *Piccolo*: An ultra-lightweight blockcipher. In: Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 6917, pp. 342–357. Springer (2011)

[132] Shinsaku, K., Toshiaki, T., Kouichi, S.: K2: a stream cipher algorithm using dynamic feedback control. In: International Conference on Security and Cryptography (SECRYPT). vol. 1, pp. 204–213 (2007)

[133] Silverman, J.H.: Fast multiplication in finite fields $GF(2^N)$. In: Workshop on Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 1717, pp. 122–134. Springer (1999)

[134] Srandaert, F.X., Örs, S.B., Preneel, B.: Power analysis of an FPGA—implementation of Rijndael: Is pipelining a DPA countermeasure? In: International Workshop on Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 3156, pp. 30–44. Springer (2004)

[135] Stankovic, R., Drechsler, R.: Circuit design from Kronecker Galois field decision diagrams for multiple-valued functions. In: 27th International Symposium Multiple-Valued Logic. pp. 275–280. IEEE (1997)

[136] Suzaki, T., Minematsu, K., Morioka, S., Kobayashi, E.: TWINE: A lightweight block cipher for multiple platforms. In: Selected Areas in Cryptography (SAC). Lecture Notes in Computer Science, vol. 7707, pp. 339–354. Springer (2012)

[137] Suzuki, D., Saeki, M., Ichilawa, T.: DPA leakage models for CMOS logic circuits. In: Workshop on Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 3659, pp. 366–382. Springer (2005)

[138] Tiri, K., Verbauwhede, I.: A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In: Design, Automation and Test in Europe Conference

and Exhibition (DATE). vol. 1, pp. 246–251 (2004)

[139] Tohoku University: Cryptographic hardware project, `http://www.aoki.ecei.tohoku.ac.jp/crypto/`

[140] Trichina, E.: Combinational logic design for AES SubBytes transformation on masked data (2003), `http://eprint.iacr.org/2003/236`, Cryptology ePrint Archive, Report 2003/236

[141] Tromer, E., Osvik, D.A., Shamir, A.: Efficient cache attacks on AES, and countermeasures. Journal of Cryptology 23, 37–71 (2010)

[142] Verbauwhede, I., Schaumont, P., Kuo, H.: Design and performance testing of a 2.29-GB/s Rijndael processor. IEEE Journal of Solid-State Circuits 38, 569–572 (2003)

[143] Wu, H.: Low complexity bit-parallel finite field arithmetic using polynomial basis. In: Workshop on Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 1717, pp. 280–291. Springer (1999)

[144] Wu, H., Hasan, A., Blake, I.F.: Highly regular architectures for finite field computation using redundant basis. In: Workshop on Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, vol. 1717, pp. 269–279. Springer (1999)

[145] Yashima, J., Takenaka, M., Shimoyama, T.: On validation tests for block cipher modules. In: 26th CSEC Group Meeting. pp. 83–89. No. 75, IEICE (2004), Japanese edition

List of Publications

Journals

1. Rei Ueno, Naofumi Homma, and Takafumi Aoki, "Automatic generation system for multiple-valued Galois-field parallel multipliers," *IEICE Transactions on Information and Systems*, Vol. E100-D, No. 8, pp. 1603–1610, August 2017.
2. Rei Ueno, Naofumi Homma, Takafumi Aoki, and Sumio Morioka, "Hierarchical formal verification combining algebraic transformation with PPRM expansion and its application to masked cryptographic processors," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science*, Vol. E100-A, No. 7, pp. 1396–1408, July 2017.
3. Rei Ueno, Naofumi Homma, Yukihiro Sugawara, and Takafumi Aoki, "Formal approach to verifying Galois field arithmetic circuits of higher Degrees," *IEEE Transactions on Computers*, Vol. 66, No. 3, March 2017.
4. Rei Ueno, Naofumi Homma, and Takafumi Aoki, "A formal verification method of error correction code processors over Galois-field arithmetic," *Journal of Multiple-Valued Logic and Soft Computing*, Old City Publishing, Vol. 26, Issue 1/2, pp. 55–73, February 2016.
5. Rei Ueno, Naofumi Homma, and Takafumi Aoki, "Efficient DFA on SPN-based block ciphers and its application to the LED block cipher," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science*, Vol. E98-A, No. 1, pp. 182–191, January 2015.

International conferences

6. Akira Ito, Rei Ueno, Naofumi Homma, and Takafumi Aoki, "On the detectability of hardware Trojans embedded in parallel multipliers," *IEEE 48th International Symposium on Multiple-Valued Logic (ISMVL 2018)*. (to appear)
7. Manami Suzuki, Rei Ueno, Naofumi Homma, and Takafumi Aoki, "Quaternary debiasing for physically unclonable functions," *IEEE 48th International Symposium on Multiple-*

Valued Logic (ISMVL 2018). (to appear)

8. Hirokazu Oshida, Rei Ueno, Naofumi Homma, and Takafumi Aoki, "On masked Galois-field multiplication for authenticated encryption resistant to side channel analysis," *9th International Workshop on Constructive Side-channel Analysis and Secure Design (COSADE 2018)*. (to appear)
9. Rei Ueno, Naofumi Homma, and Takafumi Aoki, "Design of highly efficient tamper-Resistant AES processor based on 1st order TI," *12th International Workshop on Security (IWSEC)*, Hiroshima, Japan, May 2017. (Invited talk)
10. Rei Ueno, Naofumi Homma, and Takafumi Aoki, "A systematic design of tamper-resistant Galois-field arithmetic circuits based on threshold implementation with $(d + 1)$ input shares," *IEEE 47th International Symposium on Multiple-Valued Logic (ISMVL 2017)*, pp. 136–141, Novi Sad, Serbia, May 2017.
11. Wataru Kawai, Rei Ueno, Naofumi Homma, Takafumi Aoki, Kazuhide Fukushima, and Shinsaku Kiyomoto, "Practical power analysis on KCipher-2 software on low-end microcontrollers," *Security for Embedded and Mobile Systems (SEMS), IEEE Euro Security and Privacy Workshops (EuroS&P Workshops)*, pp. 113–121, Paris, France, April 2017.
12. Rei Ueno, Naofumi Homma, and Takafumi Aoki, "Toward more efficient DPA-resistnat AES hardware architecture based on threshold implementation," *8th International Workshop on Constructive Side-channel Analysis and Secure Design (COSADE 2017)*, Lecture Notes in Computer Science 10348, Springer, pp. 50–64, Paris, France, April 2017.
13. Manami Suzuki, Rei Ueno, Naofumi Homma, and Takafumi Aoki, "Multiple-valued debiasing for physically unclonable functions and its application to fuzzy extractors," *8th International Workshop on Constructive Side-channel Analysis and Secure Design (COSADE 2017)*, Lecture Notes in Computer Science 10348, Springer, pp. 248–263, Paris, France, April 2017.
14. Rei Ueno, Naofumi Homma, and Takafumi Aoki, "Automatic generation of formally-proven tamper-resistant Galois-field multipliers based on generalized masking scheme," *IEEE/ACM 20th Design, Automation and Test in Europe Conference and Exhibition (DATE 2017)*, pp. 973–983, Lausanne, Switzerland, March 2017.
15. Rei Ueno, Sumio Morioka, Naofumi Homma, and Takafumi Aoki, "A high throughput/gate AES hardware architecture by compressing encryption and decryption datapaths—Towared efficient CBC-mode implementation," *18th International Conference on Cryptographic Hardware and Embedded Systems (CHES 2016)*, Lecture Notes in Computer Science 9813, Springer, pp. 538–558, Santa Barbara, CA, USA, August 2017.

16. Rei Ueno, Yukihiro Sugawara, Naofumi Homma, and Takafumi Aoki, "Formal design of pipelined GF arithmetic circuits and its application to cryptographic processors," *IEEE 46th International Symposium on Multiple-Valued Logic (ISMVL 2016)*, pp. 217–222, Sapporo, Japan, May 2016.
17. Wataru Kawai, Rei Ueno, Naofumi Homma, Takafumi Aoki, Kazuhide Fukushima, and Shinsaku Kiyomoto, "Side channel security evaluation for KCipher-2 software on smart cards," *25th International Workshop on Post-Binary ULSI Systems*, pp. 9–12, Sapporo, Japan, May 2016.
18. Rei Ueno, Naofumi Homma, Yukihiro Sugawara, Yasuyuki Nogami, and Takafumi Aoki, "Highly efficient $GF(2^8)$ inversion circuit based on redundant GF arithmetic and its application to AES design," *17th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2015)*, Lecture Notes in Computer Science 9293, Springer, pp. 63–80, Saint-Malo, France, September 2015.
19. Rei Ueno, Naofumi Homma, Yukihiro Sugawara, and Takafumi Aoki, "Formal design of Galois-field arithmetic circuits based on polynomial ring representation," *IEEE 45th International Symposium on Multiple-Valued Logic (ISMVL 2015)*, pp. 48–53, Waterloo, Canada, May 2015.
20. Yukihiro Sugawara, Rei Ueno, Naofumi Homma, and Takafumi Aoki, "System for automatic generation of parallel multipliers over Galois field," *IEEE 45th International Symposium on Multiple-Valued Logic (ISMVL 2015)*, pp. 54–59, Waterloo, Canada, May 2015.
21. Rei Ueno, Kotaro Okamoto, Naofumi Homma, and Takafumi Aoki, "An efficient approach to verifying Galois-field arithmetic circuits of higher degrees and its application to ECC decoders," *IEEE 44th International Symposium on Multiple-Valued Logic (ISMVL 2014)*, pp. 144–149, Bremen, Germany, May 2014.

Awards

22. Kenneth C. Smith Early Career Award in Microelectronics (May 23, 2017), Rei Ueno; "Formal design of pipelined GF arithmetic circuits and its application to cryptographic processors," *IEEE 46th International Symposium on Multiple-Valued Logic (ISMVL 2016)*. (Authors: Rei Ueno, Yukihiro Sugawara, Naofumi Homma, and Takafumi Aoki)
23. IEEE student travel-award (ISMVL 2015), Rei Ueno; "Formal design of Galois-field arithmetic circuits based on polynomial ring representation," *IEEE 45th International Symposium on Multiple-Valued Logic (ISMVL 2015)*. (Authors: Rei Ueno, Naofumi Homma, Yukihiro Sugawara, and Takafumi Aoki)

Formal Design of Cryptographic Hardware

by

Rei Ueno

Graduate School of Information Sciences, Tohoku University

January 2018