WORD-LEVEL ABSTRACTION FROM COMBINATIONAL CIRCUITS USING ALGEBRAIC GEOMETRY

by

Tim Pruss

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Electrical and Computer Engineering

The University of Utah

May 2015

Copyright © Tim Pruss 2015

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Tim Pruss

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Priyank Kalla

Ganesh Goplakrishnan

Chris Myers

Kenneth Stevens

Rongrong Chen

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Tim Pruss in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

______________________ ______________________________

Date Priyank Kalla
Chair, Supervisory Committee

Approved for the Major Department

____
Gianluca Lazzi
Chair/Dean

Approved for the Graduate Council

____
David Kieda
Dean of The Graduate School

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Abstraction plays an important role in digital design, analysis, and verification, as it allows for the refinement of functions through different levels of conceptualization. This dissertation introduces a new method to compute a symbolic, canonical, word-level abstraction of the function implemented by a combinational logic circuit. This abstraction provides a representation of the function as a *polynomial* $Z = F(A)$ over the Galois field $\mathbb{F}_{2^k}$, expressed over the k-bit input to the circuit, A. This representation is easily utilized for formal verification (equivalence checking) of combinational circuits.

The approach to abstraction is based upon concepts from commutative algebra and algebraic geometry, notably the Gröbner basis theory. It is shown that the polynomial $F(A)$ can be derived *by computing a Gröbner basis of the polynomials* corresponding to the circuit, using a specific elimination term order based on the circuits topology. However, computing Gröbner bases using elimination term orders is infeasible for large circuits. To overcome these limitations, this work introduces *an efficient symbolic computation* to derive the word-level polynomial. The presented algorithms exploit i) the structure of the circuit, ii) the properties of Gröbner bases, iii) characteristics of Galois fields $\mathbb{F}_{2^k}$, and iv) modern algorithms from symbolic computation.

While the concept is applicable to any arbitrary combinational logic circuit, it is particularly powerful in verification and equivalence checking of hierarchical, custom-designed and structurally dissimilar Galois field arithmetic circuits. In most applications, the field size and the data-path size k in the circuits is very large, up to 1024 bits.

The proposed abstraction procedure can exploit the hierarchy of the given Galois field arithmetic circuits. A custom abstraction tool is designed to efficiently implement the abstraction procedure. Preliminary experiments show that, using the proposed approach, our tool can abstract and verify Galois field arithmetic circuits *up to* 1024 *bits in size*. Contemporary techniques fail to verify these types of circuits beyond 163 bits and cannot abstract a canonical representation beyond 32 bits.

CHAPTER 1

INTRODUCTION

There is an ever-increasing need for secure communication within information technology. Security of sensitive information relies more and more heavily on encryption methodologies implemented in hardware by cryptographic circuits. One of the most prominent of these methodologies is Elliptical Curve Cryptography (*ECC*), which provides more strength per encryption bit than other encryption methodologies. The main building blocks of ECC hardware implementations are fast, *custom-built Galois field arithmetic circuits*. These circuits are notoriously hard to verify, yet their correctness is vitally important in critical applications. In [1], for example, it is shown that a bug in the hardware could lead to the full leakage of the secret cryptographic key, which could compromise the entire system. Thus, formal verification is imperative in Electronic Design Automation (*EDA*) when dealing with cryptographic circuits.

To facilitate this verification, it is highly desirable to obtain a *word-level representation of the datapath of the ECC arithmetic block* from its bit-level implementation. Ideally, this abstraction should be *canonical*, as this allows the it to be directly applicable to equivalence checking. Such a canonical, word-level abstraction of the Galois field arithmetic block would not only make it easier to verify and reason about the cryptographic system as a whole, but also enable the use of higher level abstraction and synthesis tools. As arithmetic circuits are custom-designed, often modularly, using Galois field arithmetic blocks, the abstraction should also exploit the hierarchical nature of the circuitry. Due to the modular circuit structure, abstraction of each arithmetic block becomes the key in verification of the full circuit. Practical applications of ECC dictate a datapath of a minimum of 163-bits, up to 571-bits, as designated by the National Institute for Standards and Technology (NIST). However abstraction of Galois field arithmetic circuits has been infeasible for data-paths beyond 16 bits.

This dissertation proposes an algebraic geometry based approach to abstract canonical, word-level representations of bit-level Galois field arithmetic circuits. The approach is able to abstract representations for circuits up to 571 bits in size, which is the largest NIST standard for datapath size in ECC. Verification of circuits for which this abstraction has been computed is shown to be trivial; thus, the focus is on deriving the abstraction quickly and efficiently.

1.1 Hardware Design and Verification Overview

The typical design flow of a hardware system, as shown in Fig. 1.1, starts with a hardware system specification, which describes the necessary functions and parameters that the system must perform and adhere to. The specification is typically modelled using a transaction-level model (*TLM*), which describes communication details between large circuit modules. The TLM is then translated into a register-transfer-level (*RTL*) description, which is composed of abstracted, interconnected circuit blocks that compose the entire system. RTL is typically implemented in hardware description languages (*HDL*) such as Verilog and VHDL, which are the most popular choices in the industry. Next, the RTL is optimized and converted into a *netlist*, i.e. a large collection of small physical blocks (MOSFET, Boolean logic gates, etc.) and the inter-connections (wires) between them. Lastly, the netlist is further optimized and then mapped onto a physical space on a chip, which is then sent off for fabrication. This entire design flow is automated by Computer-Aided Design (*CAD*) tools.

When moving from one abstraction level of the hardware design process to the next, an important issue arises: how can one ensure that the functionality of the optimized design matches original spec? Bugs in hardware design which are not caught early can have costly effects later, such as the need for a redesign. Bugs in arithmetic circuits can be especially catastrophic. One infamous example is the 1994 floating point division (FDIV) bug that affected the Intel Pentium chip [2], and subsequently cost the company \$475 million because it was discovered after the chip's release. In another more fatal case, during the Gulf war, an American Patriot Missile battery failed to intercept an

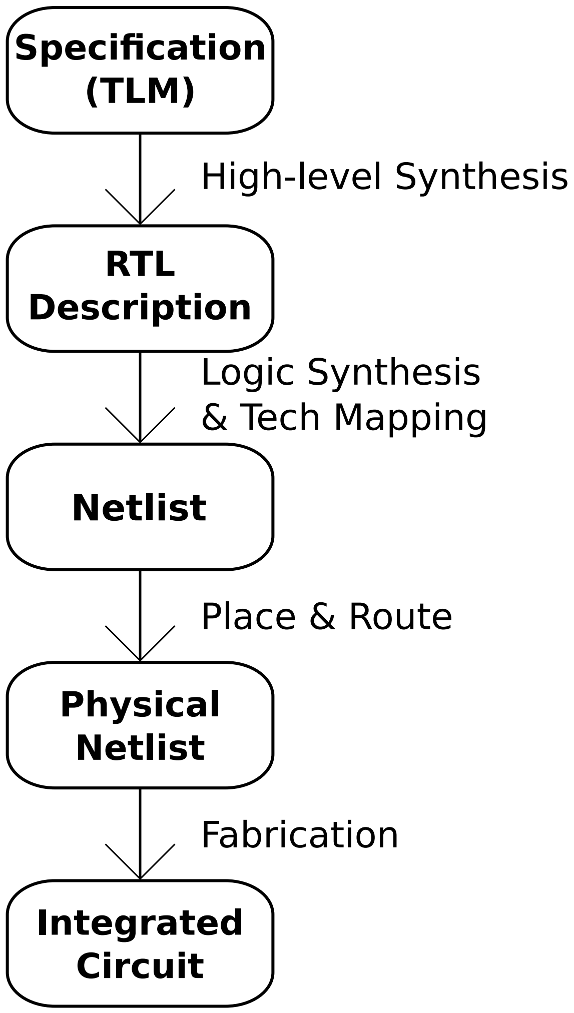


Figure 1.1: Typical hardware design flow.

incoming enemy missile due to an arithmetic error [3]. Since hardware bugs can have significant consequences, there has been extensive work in field of hardware verification to find and eliminate bugs prior to fabrication.

The two main methodologies used in hardware verification are simulation and formal verification. *Simulation* checks correctness by applying exhaustive assignments to the circuit inputs and verifying correctness of the output. This ensures that the circuit performs as designed under all possible inputs. Such exhaustive testing is quite effective for smaller circuits. However, as the size of the circuit increases, it becomes computationally infeasible to simulate all possible test vectors. This is the case with Galois field arithmetic circuits, which are commonly very large in real-world applications. Often for such large circuits, simulations of a smaller and more manageable subset of test vectors are employed to catch bugs. While these tests can increase confidence in the correctness of the design, *they do not guarantee correctness* since every data-flow of the design hasn't been analyzed.

1.2 Formal Verification

Instead of simulating input vectors, *formal verification* utilizes mathematical theory to reason about the correctness of hardware designs. Formal verification has two main forms: property checking and equivalence checking.

Property checking (or property verification) verifies that a design satisfies certain given properties. Property checking is done mainly in the form of theorem proving, model checking, or approaches which combine the two.

1. *Theorem proving* [4] requires the existence of mathematical descriptors of the specification and implementation of the circuit. Theorem provers apply mathematical rules to these descriptors to derive new properties of the specification. In this way, the tool can reduce a proof goal to simpler sub-goals, which can be automatically verified. However, generating the initial proof-goal requires extensive guidance from the user, so there is an overall lack of automation in theorem proving.

2. *Model checking* [5] is an approach to verifying finite-state systems where specification properties are modeled as a system of logic formulas. The design is then traversed to check if the properties hold. If the design is found to violate a particular property, a counter-example is generated which exercises the incorrect behavior in the design. Such counter-examples allow the designer to trace the behavior and find where the error in the design lies. Modern model checking techniques use the result to automatically refine the system and perform further checking. These tools are typically automated, and thus have found widespread use in CAD tool suites.

Equivalence Checking verifies that two different representations of a circuit design have equivalent functionality. An example of equivalence checking as it applies to the hardware design flow is shown in Fig. 1.2.

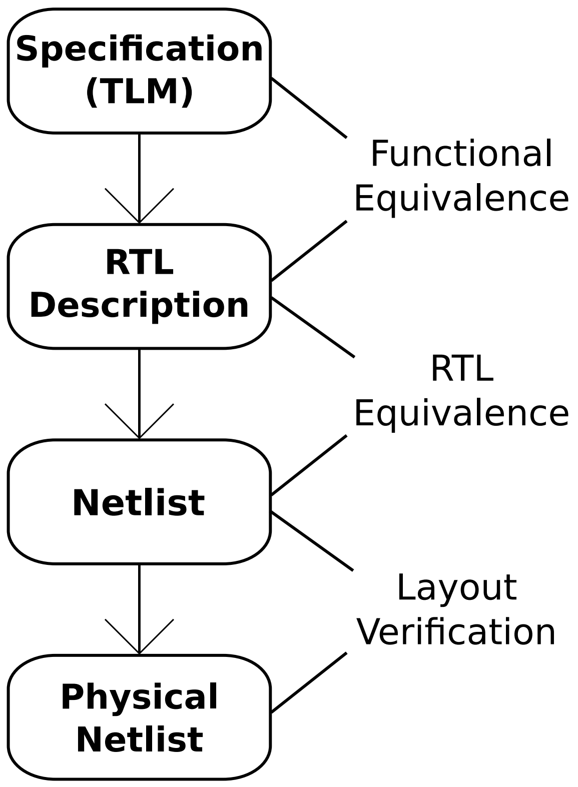


Figure 1.2: Equivalence checking as applied to the hardware design flow.

There are two major equivalence checking techniques: graph-based and satisfiability-based.

1. *Graph-based* techniques construct a canonical graph representation, such as a Binary Decision Diagram (*BDD*) or one of its many variants, of each circuit. A linear comparison is then conducted to determine whether the two graphs are isomorphic. Since the graph representation is canonical, the graphs of the two circuits will be equivalent if and only if the circuits perform the same function.

2. *Satisfiability* techniques construct a miter of the two circuits, typically in a graph such as an And-Inverter graph (*AIG*). A *miter* is a combination of the two circuits with one bit-level output, which is only in a ”1” state when the outputs of the circuits differ given the same given input, as shown in Fig. 1.3. A satisfiability (*SAT*) tool [6] is then employed to simplify the graph and find a solution to the miter, i.e. find an input for which the miter output is ”1”. If a solution is found,

this solution acts as a counter-example of when the circuit outputs differ; otherwise the circuits are functionally equivalent.

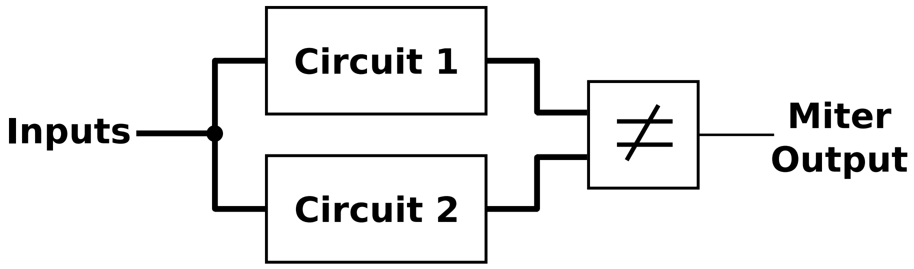


Figure 1.3: A miter of two circuits.

Certain formal verification methods use *computer-algebra* and *algebraic geometry* techniques based on mathematical theories. Unlike SAT-based verification, modern algebraic geometry techniques do not explicitly solve the constraints to find a solution; rather, they reason about the presence or absence of solutions, or explore the geometry of the solutions. These methods [7] [8] [9] transform the circuit design into a polynomial system. Typically, this system of polynomials is then used to compute a Gröbner basis [10]. Computation of Gröbner bases allows for the easy deduction of important properties of a polynomial system, such as the presence or absence of solutions. These properties are then leveraged to perform verification. Unfortunately, such a computation has been shown to be doubly exponential in the worst case, and thus these methods have not been practical for real-world applications. However, recent breakthroughs in computer-algebra hardware verification have shown that it is possible to overcome the complexity of this computation while still utilizing the beneficial properties of a Gröbner bases [11].

1.3 Importance of Word-level Abstraction

Most formal verification techniques can benefit from word-level abstractions of the circuits they verify. Abstraction is defined as state-space reduction, i.e. abstraction reduces state-space by mapping the set of states of a system to a smaller set of states. Because the new representation contains fewer states, it is easier to comprehend and thus easier to use. Word-level abstraction focuses specifically on abstracting a word-level representation of a circuit out of a bit-level representation. For example, a bit-level representation of an integer multiplier is represented by a collection of Boolean inputs and outputs, whereas a word-level abstraction hides the underlying logic and represents the circuit as two integer inputs and one integer output, e.g. $Z = A \cdot B$. As the bit-size of the multiplier increases, the logical implementation of the multiplier grows (typically exponentially) while the word-level abstraction stays the same.

Word-level abstractions have a wide variety applications in formal verification. Theorem proving techniques can leverage abstraction as an automatic decision procedure or as a canonical reduction engine. For example, since RTL is composed of circuit blocks that represent the underlying circuit, RTL verification methods can exploit abstractions of these blocks. This is seen in the following RTL verification methods:

- Model checking [12], where an approximation abstraction of RTL blocks is generated and then refined.
- Graph-based equivalence checking [13] [14], where abstraction methods are used to generate a canonical word-level graph representation of the circuit.
- Satisfiability-based equivalence checking [15], where abstractions are used identify symmetrics and similarities in order to minimize the amount of logic that is sent to the SAT tool.

Other equivalence checking techniques that employ abstractions include satisfiability modulo theory (*SMT*) techniques [16] [17], which are similar to SAT except they operate on higher-level data structures (integers, reals, bit vectors, etc.), as well as constraint solving techniques [18] [19]. In general, RTL equivalence checking approaches would ideally maintain a high-level of abstraction while still retaining sufficient lower-level functional details (such as bit-vector size, precision, etc) [20].

Word-level hardware abstractions also have applications in RTL and datapath synthesis [21] [22] [23]. Abstractions of circuits allow for design reuse, which allows for tool-automated synthesis of larger circuit blocks. Since hardware design specifications tend to be word-level, synthesis tools can use these larger circuit blocks to generate and optimize the datapaths and create the RTL of the system. Thus, in order for a circuit to be used by these automated synthesis tools, its word-level abstraction must be known.

Finally, abstractions can also be applied to detect malicious modifications to a circuit, potentially inserted as a hardware trojan horse. Hardware trojans, a relatively new security concern in the hardware industry, use certain techniques to add incorrect behavior to a design. This behavior is only activated under certain rare circumstances that only the mal-intent designer has knowledge of. The behavior is purposely hidden and is very difficult to encounter during simulation of the design. A manufactured chip with a subsystem that contains a hardware trojan could compromise the entire system in which it is used. In some hardware trojan cases, formal verification techniques may be applied to catch a bug in a design and provide a counter-example which exercises it. However, it can be difficult to tell whether the bug in the design was introduced intentionally of not. On the other hand, word-level abstractions of bit-level circuits *effectively reverse-engineer the true function implemented by the circuit*, which could be used to determine the designer's true intention.

1.4 Dissertation Objective, Motivation, and Contributions

This dissertation focuses on abstracting a canonical, word-level representation of hardware (bit-level) implementations of combinational circuits. The proposed technique is a full abstraction solution which can be applied to any arbitrary acyclic combinational circuit. It is particularly efficient when applied to Galois field arithmetic circuits. Using this technique, if the abstraction of the circuit's implementation and its specification are found, they can be easily compared to determine equivalence. Implementation of a custom software tool, developed to compute the abstractions, is also described.

1.4.1 Motivating Application

The motivation for this work comes from applications of Galois field arithmetic circuits in elliptical curve cryptography (*ECC*) hardware systems. The main operations of encryption, decryption, and authentication in ECC rely on operations performed on elliptic curves, which are implemented in hardware as polynomial functions over Galois fields. To be applicable in real-world situations, ECC data-paths should be a minimum of 163-bits wide, which is the minimum NIST standard, up to a recommended size of 571-bit operand widths. Many non-ECC cryptosystems have datapaths on the order of 1000-bits.

A Galois field arithmetic circuit with a datapath size of k is built as a Boolean function: $\mathbb{B}^k \rightarrow \mathbb{B}^k$. This function is mapped to an operation $f : \mathbb{F}_{2^k} \rightarrow \mathbb{F}_{2^k}$ over the Galois field $\mathbb{F}_{2^k}$. These circuits are custom-built, modular systems which cannot be synthesized due to their complex nature. Thus, formal verification is needed to ensure they operate correctly.

Recent computer-algebra based formal verification techniques have been able to perform verification of Galois field arithmetic circuits with a datapath size up to 163-bits [11]. Word-level abstractions of Galois field arithmetic circuits could be used to further improve these formal verification techniques to allow for verification of larger circuits, as well as provide the other benefits of word-level abstraction. However, there is currently no technique for computing word-level abstractions of Galois field circuits of any practical size.

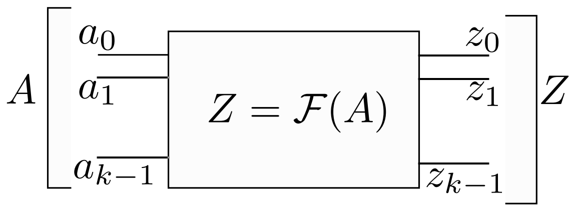


Figure 1.4: Circuit with k-bit input A and k-bit output Z. Abstraction to be derived as $Z = \mathcal{F}(A)$.

While the motivation comes from the need to verify Galois field arithmetic circuits, the presented approach can be generalized to be applicable to any combinational acyclic circuit. Any such circuit with a k-bit input A and a k-bit output Z, such as the one shown in Fig. 1.4, computes $f : \mathbb{B}^k \rightarrow \mathbb{B}^k$ and can thus be analyzed as the function $f : \mathbb{F}_{2^k} \rightarrow \mathbb{F}_{2^k}$. Over $\mathbb{F}_{2^k}$ this function can be represented as the polynomial $Z = \mathcal{F}(A)$. This is trivially generalized when there are multiple k-bit inputs $A_1, A_2, \ldots, A_i$, i.e. $Z = \mathcal{F}(A_1, \ldots, A_i)$. Now assume the word-size of the input differs from the output, that is the circuit computes $f : \mathbb{B}^m \rightarrow \mathbb{B}^n$ for $m \neq n$. This can be represented as a function over Galois fields as $f : \mathbb{F}_{2^m} \rightarrow \mathbb{F}_{2^n}$. This function can be analyzed over the field $\mathbb{F}_{2^k}$ such that $\mathbb{F}_{2^k} \supset \mathbb{F}_{2^m}$ and $\mathbb{F}_{2^k} \supset \mathbb{F}_{2^m}$, where $k = LCM(m, n)$.

1.4.2 Dissertation Contributions

To solve the problem of word-level abstraction, this dissertation proposes a full solution consisting of three main contributions.

1. A theory for finding the word-level abstraction from a bit-level circuit over Galois fields is created. The given bit-level circuit implementation is modelled as a system of polynomials over the field. This theory is derived using techniques from computer-algebra, notably the theory of Gröbner basis [24].

2. Using this theory, new algorithms based on symbolic computation are developed to derive the word-level abstraction. The algorithms are designed to be applicable to industry-size arithmetic circuits over Galois fields [25] [26]. A complexity analysis of the algorithmic approach is also presented. Furthermore, the approach is also generalized to make it applicable to arbitrary combinational circuits. Finally, we show how the approach can be used to exploit the hierarchical structure of large Galois field multipliers designed over composite fields.

3. A custom software tool implementation of the algorithmic approach is described, including an analysis of efficient data structures designed for this purpose [27].

Experiments show that the proposed solution can abstract canonical, word-level, polynomial representations of Galois field arithmetic circuits up to 1024-bits in size, while other contemporary approaches are infeasible beyond a 32-bit designs.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 reviews previous applicable work and highlights their drawbacks with respect to the canonical, word-level abstraction problem. Chapter 3 describes the properties of Galois fields, $\mathbb{F}_{2^k}$, and explains the process of constructing them. It also describes how to design arithmetic circuits over such fields, their complexities, and the role of these circuits in Elliptic

Curve Cryptography. Chapter 4 provides a theoretical background of computer-algebra and Gröbner bases and explains their application to Galois fields. Chapter 5 describes an approach to abstract word-level polynomial representations of combinational circuits using a Gröbner basis computation. Chapter 6 improves on this word-level abstraction approach to make it applicable to much larger circuits. Chapter 7 generalizes the abstraction approach to make it applicable to circuits with varying operand word-lengths. It also describes how the approach can take advantage of the hierarchy of arithmetic circuits designed over composite fields. Chapter 8 describes the implementation details of a custom abstraction tool and gives experimental results of abstracting large Galois field multiplier circuits. Chapter 9 concludes the dissertation and outlines potential future research for continuation of this work.

CHAPTER 2

PREVIOUS WORK

This chapter covers previous work in the area of canonical representations of functions, word-level abstractions and their application to design verification. Since the application of our approach is targeted towards formal equivalence verification, modern combinational equivalence checking techniques are also reviewed. Finally, formal verification techniques using computer algebra, algebraic geometry, and polynomial interpolation are also considered.

2.1 Canonical Decision Diagrams

Canonical representations of Boolean functions have been the subject of extensive investigation for logic synthesis and design verification. The Reduced Ordered Binary Decision Diagram (ROBBD) [28] was the first significant contribution in this area. Efficient implementation of ROBDDs as a software package [29] allowed for efficient formal verification of combinational and sequential circuits. ROBDDs represent a Boolean function as an implicit set of points on a canonical directed acyclic graph (DAG). Manipulation of Boolean functions can then be carried out as composition operations on their respective DAGs. The decomposition principle behind BDDs is one of Shannon's expansion, i.e.

$$f(x, y, \dots) = x f_x + x' f_{x'} \tag{2.1}$$

where $f_x = f(x = 1)$ and $f_{x'} = f(x = 0)$ denote the positive and negative co-factors of f w.r.t. x, respectively. Motivated by the success of BDDs, variants of the Shannon's decomposition principle (Davio, Reed-Muller, etc.) were explored to develop other functional decision diagrams. For example, the AND-OR-NOT logic based Shannon's expansion is transformed into an AND-XOR logic based decomposition, termed as the Davio's decomposition:

$$\begin{aligned} f(x, y, \dots) &= x f_x + x' f_{x'} && (2.2) \\ &= x f_x \oplus x' f_{x'} && (2.3) \\ &= x f_x \oplus (1 \oplus x) f_{x'} && (2.4) \\ &= f_{x'} \oplus x(f_x \oplus f_{x'}) && (2.5) \end{aligned}$$

Decision diagrams based on such decompositions include FDDs [30], ADDs [31], MTBDDs [32], and their hybrid edge-valued counterparts, HDDs [33] and EVBDDs [34]. While these are referred to as *Word-Level Decision Diagrams* [13], the decomposition is still point-wise, binary, w.r.t. each Boolean variable. These representations do not serve the purpose of word-level abstraction from bit-level representations.

Binary Moment Diagrams (BMDs) [35], and its derivatives K*BMDs [36] and *PHDDs [37], depart from the Boolean decomposition and perform the decomposition of a *linear* function based on its two moments. BMDs provide a compact representation for integer arithmetic circuits such as multipliers and squarers. However, these are inapplicable to word-level abstraction of modulo-arithmetic circuits over Galois fields.

Taylor Expansion Diagrams (TEDs) [38] are a word-level canonical representation of a *polynomial expression*, based on the Taylor's series expansion of a polynomial. However, they do not represent a *polynomial function* canonically. For example, $f_1 = 0$ and $f_2 = 2x^2 - 2x \pmod 4$ are two different polynomial representations of the zero function over $\mathbb{Z}_4$; but they are symbolically different polynomials and they have non-isomorphic TED DAGs. While [39] and [40] provide canonical representations of polynomial functions, they do so over finite integer rings $\mathbb{Z}_{2^k}$ and not over Galois fields $\mathbb{F}_{2^k}$.

MODDs [41] [42] are a DAG representation of the characteristic function of a circuit over Galois fields $\mathbb{F}_{2^k}$. MODDs come very close to satisfying our requirements as a canonical word-level representation that can be employed over Galois fields, as it essentially interpolates a polynomial from the characteristic function. However, MODDs do not scale very well for large circuits — this is because every node in the DAG can have up to k children and the normalization operations are very complicated for MODDs.

They also suffer from the size explosion problem during intermediate computations. They are known to be infeasible in representing functions over 32-bit operand words.

2.2 Word-Level Techniques in RTL Synthesis and Verification

Other attempts to derive high-level representations of functions, along with associated decision procedures, can be found in the rich domain of formal model checking [43] [44], theorem proving [14], bit-vector SMT-solvers [16] [45] [46] [47], automated decision procedures for Presburger arithmetic [48] [49], algebraic manipulation techniques [50], or the ones based on term re-writing [51], etc. Polynomial, integer and other non-linear representations have also been researched: Difference Decision Diagrams (DDDs) [52] [53], interval diagrams [54], interval analysis using polynomials [55], etc. Most of these have found application in constraint satisfaction for simulation-based validation: [56] [57] [15] [58] [59] [47]. Among these, [58] [59] [47] have been used to *solve* integer modular arithmetic on linear expressions - a different application from *representing* finite field modulo-arithmetic on polynomials in a canonical form.

2.3 Combinational Equivalence Checking

The verification problem addressed in this dissertation is a manifestation of the combinational equivalence checking (CEC) problem, where the specification (polynomial) and the implementation (circuit) are custom-designed, structurally very dissimilar circuits. To make use of contemporary gate-level CEC tools, we can take the specification circuit ("golden model") and check its equivalence against the implementation circuit. Canonical decision diagrams (BDDs [28] and their word-level variants [13]), And-Invert-Graph (AIG) based reductions [60] [61], circuit-SAT solvers [6], etc., are among the many techniques that can be employed for this CEC. When one circuit is synthesized from the other, this problem can be efficiently solved using AIG-based reductions (e.g. the ABC tool [62]) and circuit-SAT solvers (e.g., CSAT [6]). Synthesized circuits generally contain many sub-circuit equivalences which AIG and CSAT based tools can identify and exploit for verification. However, when the circuits are functionally equivalent but structurally very dissimilar (e.g., Mastrovito [63] versus Montgomery implementations [64] of Galois field circuits), none of the contemporary techniques, including ABC and CSAT, offer a practical solution. Automatic formal verification of large *custom-designed modulo-arithmetic circuits* largely remains unsolved today.

This verification problem is very hard for SAT solvers and also for quantifier-free bit-vector (QF-BV) theory based SMT-solvers, due to the large circuit size, and the presence of AND-XOR-SHIFT structures. Similarly, the Cryptol tool-set [65] also employs AIG-based reductions (SAT-sweeping) and SAT/SMT-solvers for verification of crypto-protocols. For applications where AIGs/SAT/SMT-techniques fail, the *Cryptol* tool-set also does not deliver. As shown in [11], *none of BDDs, SAT, SMT solvers, nor the ABC tool can prove design equivalence beyond* 16*-bit circuits.*

In [66], the authors present a method for verification of integer multipliers using a data-flow approach. This work abstracts a polynomial function of the given multiplier and then solves the network flow problem using algebraic techniques. However, the abstraction is solely bit-level, and is thus not applicable to deriving a word-level representation of a given design.

2.4 Verification of Galois Field Circuits

Symbolic computer algebra techniques have been employed for formal verification of circuits over $\mathbb{Z}_{2^k}$ and also over Galois fields $\mathbb{F}_{2^k}$. The work of [67] shows how to use Gröbner basis techniques to count the zeros of an ideal J over $\mathbb{F}_q$ (i.e. count $V_{\mathbb{F}_q}(J)$). The authors then follow-up with an approach for *quantifier elimination* over Galois fields $\mathbb{F}_q$ [68]. However, computing a Gröbner basis is computationally expensive. While these works present the proper theory and algorithms, efficiency/improvements to the Gröbner basis computation is not addressed. This is also the case with other general verification techniques using Gröbner bases [7] [9] [69], etc.

In [70] [71] [72], the authors present the BLUEVERI tool from IBM for verification of Galois field circuits for error correcting codes against an algorithmic spec. The implementation consists of a set of (pre-designed and verified) circuit blocks that are interconnected to form the error correcting system. The spec is given as a set of design constraints on a “check file”. Their objective is to prove the equivalence of the implementation against this check file. They model the verification instance as a data-flow graph, represent each sub-circuit block with its known (word-level) polynomial over $\mathbb{F}_q$, and formulate the verification problem using the *Weak Nullstellensatz* — i.e. to check if the *variety* of the algebraic system "*spec* $\neq$ *implementation*" is empty for which they employ a Nullstellensatz formulation. Their main contributions are: i) a "term re-writing" to specify the algorithmic description using polynomials (ideal); and ii) integrating an AIG-style [60] Boolean solver with their word-level decision procedure, with lazy signal computations and Boolean reasoning. For final verification, the polynomial system is given to a computer algebra tool (SINGULAR [73]) to *compute* a reduced Gröbner basis. However, improvements to the core Gröbner basis computational engine are not the subject of their work.

In [11] [74] [75] [76] [77], *Lv et al.* present computer algebra techniques for formal verification of Galois field arithmetic circuits. Given a specification polynomial f, and a circuit C, they formulate the verification problems as an ideal membership test using the Strong Nullstellensatz and Gröbner bases. In [75], the authors show that for any combinational circuit, *there exists a term order* $>_1$ *that renders the set of polynomials of the circuit itself a Gröbner basis — and this term order can be easily derived by performing a topological traversal of the circuit.* By exploiting this term order, verification can be significantly scaled to 163-bit (NIST-specified) cryptography circuits. In contrast to the work of [11], we are not given a specification polynomial. Instead, given the circuit C, we want to derive (extract) the word-level specification f. In our work, we borrow and further build upon the results of [67] [68] [75] [11].

2.5 Verification of Integer Arithmetic Circuits using Gröbner Bases

Symbolic computer algebra techniques have been used for verification of integer arithmetic circuits [78] and also for decision procedures over Galois fields [67]. The paper [78] addresses verification of finite precision integer datapath circuits using the concepts of Gröbner bases over the ring $\mathbb{Z}_{2^k}$. This work models the circuit constraints by way of arithmetic-bit-level (ABL) polynomials ($\{G\}$), and formulates the verification test as an equivalent variety subset problem. This problem is solved by deriving a term order that already makes $\{G\}$ a Gröbner basis, then computing a normal form f of the specification g w.r.t. $\{G\}$. Circuit correctness is established by testing whether or not f is a vanishing polynomial over $\mathbb{Z}_{2^k}$ [79]. In [80], the authors further show that the vanishing polynomial test can be omitted by formulating the problem directly over $Q := \mathbb{Z}_{2^k}[X]/\langle x^2 - x : x \in X\rangle$. However, in these works, the problem requires that the word-level abstraction of the circuit be known, whereas our approach derives this abstraction polynomial.

2.6 Polynomial Interpolation in Symbolic Computation

The problem of polynomial interpolation is a fundamental problem in symbolic and algebraic computing which finds application in modular algorithms, such as the GCD computation and polynomial factorization. The problem is stated as follows: Given n distinct data points $x_1, \ldots, x_n$, and their evaluations at these points $y_1, \ldots, y_n$, *interpolate* a polynomial $\mathcal{F}(X)$ of degree $n-1$ (or less) such that $\mathcal{F}(x_i) = y_i$ for $1 \leq i \leq n$. Let t be the number of non-zero terms in $\mathcal{F}$ and let T be the total number of possible terms. When $\frac{t}{T} << 1$, the polynomial $\mathcal{F}$ is *sparse*, otherwise it is *dense*. Much of the work in polynomial interpolation addresses sparse interpolation using the "black-box" model (also called the algebraic circuit model) as shown in Fig. 2.1.

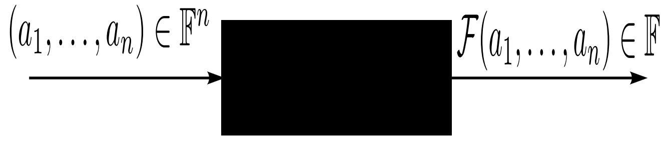

$(a_1,\ldots,a_n) \in \mathbb{F}^n$
$\mathcal{F}(a_1,\ldots,a_n) \in \mathbb{F}$

Figure 2.1: The black-box or the algebraic circuit representation.

Let $\mathcal{F}$ be a multivariate polynomial in n variables $\{x_1, \ldots, x_n\}$, with t non-zero terms ($0 < t < T$), represented with a black-box B. On input $(x_1, \ldots, x_n)$, the black-box evaluates $y_i = \mathcal{F}(x_1, \ldots, x_n)$. Given also a degree bound d on $\mathcal{F}$, the goal is to interpolate the polynomial $\mathcal{F}$ with a minimum number of *probes* to the black-box. The early work of Zippel [81] and Ben-Or/Tiwari [82] require $O(ndt)$ and $O(T \log n)$ probes, respectively, to the black-box. These bounds have since been improved significantly; the recent algorithm of [83] interpolates with $O(nt)$ probes.

Our problem of polynomial abstractions of Galois field circuits falls into the category of dense interpolation, as we require a polynomial that describes the function at each of the q points of the field $\mathbb{F}_q$. Newton's interpolation technique, with the black-box model, bounds the number of probes by $(d+1)^n$ — which exhibits very high complexity. In the logic synthesis area, the work of [84] investigates dense interpolation. Due to this high-complexity, their approach is feasible only for applications over small fields, *e.g.* computing Reed-Muller forms for multi-valued logic over $\mathbb{F}_2$.

For our problem, we can also employ the black-box model by replacing the black-box (algebraic circuit) by the given circuit C; then every *probe* of the black-box would correspond to a *simulation of the circuit*. However, as we desire a polynomial representation of the entire function over the Galois field, exhaustive simulation would be required, which is infeasible.

2.7 Concluding Remarks

For the problem of word-level, canonical, polynomial abstractions of Galois field arithmetic circuits over $\mathbb{F}_{2^k}$, previous related work is either inapplicable or only applicable to circuits no larger than 32-bits in size. Therefore, we propose a *symbolic approach* to polynomial interpolation from a circuit using the Gröbner basis computation. However, the complexity of a Gröbner basis computation is prohibitively expensive; thus, we propose further improvements to this approach by deriving a smaller subset of computations based on a Gröbner basis analysis. These improvements allow for abstractions of flattened Galois field circuits up to 571-bits, which is the largest NIST standard for ECC, or up to 1024-bits when a hierarchy is given. Furthermore, we propose applications of this approach to allow for formal verification of flattened Galois field circuits up to 1024-bits, where current techniques are only applicable for circuits up to 163-bits.

CHAPTER 3

GALOIS FIELDS PRELIMINARIES AND APPLICATION IN HARDWARE DESIGN

This chapter provides a mathematical background for understanding Galois fields and explains how to design Galois field arithmetic circuits. We first introduce the mathematical concepts of groups, rings, fields, and polynomials. We then apply these concepts to create Galois field arithmetic functions and explain how to map them to a Boolean circuit implementation. The material is referred from [85] [86] [87] for Galois field concepts and [63] [88] [64] [89] [90] for hardware design over Galois fields and previous work by *Lv* [11].

3.1 Rings, Fields and Polynomials

Definition 3.1 *An* **abelian group** *is a set* $\mathbb{S}$ *with a binary operation* $'+'$ *which satisfies the following properties:*

- *Closure Law: For every* $a, b \in \mathbb{S}, a + b \in \mathbb{S}$
- *Associative Law: For every* $a, b, c \in \mathbb{S}, (a + b) + c = a + (b + c)$
- *Commutativity: For every* $a, b \in \mathbb{S}, a + b = b + a.$
- *Additive Identity: There is an identity element* $0 \in \mathbb{S}$ *such that for all* $a \in \mathbb{S}$; $a + 0 = a.$
- *Additive Inverse: If* $a \in \mathbb{S}$*, then there is an element* $a^{-1} \in \mathbb{S}$ *such that* $a + a^{-1} = 0.$

The set of integers $\mathbb{Z}$ forms an abelian group under the addition operation.

Definition 3.2 *Given a set* $\mathbb{R}$ *with two binary operations,* $'+'$ *and* $'\cdot'$*, and element* $0 \in \mathbb{R}$*, the system* $\mathbb{R}$ *is called a* **commutative ring with unity** *if the following properties hold:*

- *$\mathbb{R}$ forms an abelian group under the '+' operation with additive identity element* 0.
- *Multiplicative Distributive Law: For all $a, b, c \in \mathbb{R}$, $a \cdot (b + c) = a \cdot b + a \cdot c$.*
- *Multiplicative Associative Law: For every $a, b, c \in \mathbb{R}$, $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.*
- *Multiplicative Commutative Law: For every $a, b \in \mathbb{R}$, $a \cdot b = b \cdot a$*
- *Identity Element: There exists an element $1 \in \mathbb{R}$ such that for all $a \in \mathbb{R}$, $a \cdot 1 = a = 1 \cdot a$*

For the purpose of this dissertation, any time we refer to a **ring**, we are specifically referring to a **commutative ring with unity**. Two common examples of such rings are the set of integers, $\mathbb{Z}$, and the set of rational numbers, $\mathbb{Q}$. Note that while both of these examples are rings with an infinite number of elements, the number of elements in a ring can also be finite.

Definition 3.3 *The* **modular number system** *with base n is a set of positive integers $Z_n = \{0, 1, \ldots, n-1\}$, with the two operations $+$ and $\cdot$ satisfying the properties below:*

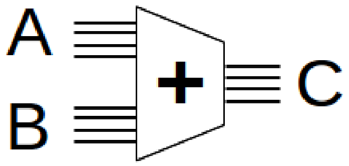
$$\begin{aligned}(a+b) \pmod n &\equiv ((a \pmod n) + (b \pmod n)) \pmod n\\ (a \cdot b) \pmod n &\equiv ((a \pmod n) \cdot (b \pmod n)) \pmod n\\ (-a) \pmod n &\equiv (n-a) \pmod n\end{aligned}$$

Example 3.1 *The set $Z_8 = \{0, 1, \ldots, 7\}$ denotes the modular number system with base* 8*. Examples of some operations performed $\pmod 8$ are:*

$$\begin{aligned}3+6 &= 9 \pmod 8 = 1\\ 3 \cdot 6 &= 18 \pmod 8 = 2\\ (-3) &= 8-3 \pmod 8 = 5\end{aligned}$$

The modular number system $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$, where n is a positive integer, forms a ring. Since this type of ring contains a finite number of elements n, it is termed a *finite integer ring*, where addition and multiplication are computed *modulo n* $\pmod{n}$. In hardware applications, arithmetic over k-bit vectors manifests itself as algebra over the finite integer ring $\mathbb{Z}_{2^k}$, where the k- bit vector represents integer values from $\{0, ..., 2^k - 1\}$.

Example 3.2 *Consider the following arithmetic circuit:*



This circuit takes two 4-bit inputs, A and B, and computes a 4-bit sum C. Since A, B, and C are all bit-vectors of size 4, the addition computation this circuit performs is modulo 2^4. Hence, this circuit exemplifies arithmetic computations over the ring Z_{2^4}.

Some examples of possible inputs and outputs of the circuit:

Addition over $\mathbb{Z}_{2^4}$	Boolean Circuit Implementation
$5+8 = 13 \pmod{16} = 13$	$A = 0101, B = 1000 \rightarrow C = 1101$
$10+9 = 19 \pmod{16} = 3$	$A = 1010, B = 1001 \rightarrow C = 0011$
$12+4 = 16 \pmod{16} = 0$	$A = 1100, B = 0100 \rightarrow C = 0000$

Definition 3.4 *Let $\mathbb{R}$ be a ring. A* **polynomial** *over $\mathbb{R}$ in the indeterminate x is an expression of the form:*

$$a_0 + a_1x + a_2x^2 + \cdots + a_kx^k = \sum_{i=0}^{k} a_ix^i, \forall a_i \in \mathbb{R}. \tag{3.1}$$

The constants a_i are the coefficients and k is the degree of the polynomial. For example, $8x^3 + 6x + 1$ is a polynomial in x over $\mathbb{Z}$, with coefficients 8, 6, and 1 and degree 3.

Definition 3.5 *The set of all polynomials in the indeterminate x with coefficients in the ring $\mathbb{R}$ forms a* **ring of polynomials** *$\mathbb{R}[x]$. Similarly, $\mathbb{R}[x_1, x_2, \cdots, x_n]$ represents the ring of multivariate polynomials with coefficients in $\mathbb{R}$.*

For example, $\mathbb{Z}_{2^4}[x]$ stands for the set of all polynomials in x with coefficients in $\mathbb{Z}_{2^4}$. $8x^3 + 6x + 1$ is an instance of a polynomial contained in $\mathbb{Z}_{2^4}[x]$.

Definition 3.6 *A* **field** $\mathbb{F}$ *is a commutative ring with unity, where every non-zero element in* $\mathbb{F}$ *has a multiplicative inverse; i.e.* $\forall a \in \mathbb{F} - \{0\}$, $\exists \hat{a} \in \mathbb{F}$ *such that* $a \cdot \hat{a} = 1$.

A field is defined as a ring with one extra condition: the presence of a multiplicative inverse for all non-zero elements. Therefore, a field must be a ring while a ring is not necessarily a field. For example, the set $\mathbb{Z}_{2^k} = \{0, 1, \cdots, 2^k - 1\}$ forms a finite ring. However, $\mathbb{Z}_{2^k}$ is not a field because not every element in $\mathbb{Z}_{2^k}$ has a multiplicative inverse. In the ring $\mathbb{Z}_{2^3}$, for instance, the element 5 has an inverse ($5 \cdot 5 \pmod 8 = 1$) but the element 4 does not.

The main concept of field theory is **Field Extensions**. The idea behind a field extension is to take a base field and construct a larger field which contains the base field as well as satisfies additional properties. For example, the set of real numbers $\mathbb{R}$ forms a field; one common extension of $\mathbb{R}$ is the set of complex numbers $\mathbb{C} = \mathbb{R}(i)$. Every element of $\mathbb{C}$ can be represented as $a + b \cdot i$ where $a, b \in \mathbb{R}$, hence $\mathbb{C}$ is a two-dimensional extension of $\mathbb{R}$.

Like rings, fields can also contain either an infinite or a finite number of elements. In this dissertation we focus on finite fields, also known as Galois fields, and the construction of their field extensions.

3.2 Galois Fields

Galois fields, also known as finite fields, find widespread applications in many areas of electrical engineering and computer science such as error- correcting codes, elliptic curve cryptography, digital signal processing, testing of VLSI circuits, among others. In this dissertation, we specifically focus on their application to Elliptic Curve Cryptography as Galois field arithmetic circuits. This section describes the relevant Galois field concepts [85] [86] [87] and hardware arithmetic designs over such fields [63] [88] [64] [89] [90].

Definition 3.7 *A* **Galois field***, denote* $\mathbb{F}_q$*, is a field with a finite number of elements,* q.

The number of elements q of the Galois field is a power of a prime integer, i.e. $q = p^k$, where p is a prime integer, and $k \geq 1$. Thus a Galois field can also be denoted as $\mathbb{F}_{p^k}$.

Fields in the form $\mathbb{F}_{p^k}$ are called Galois extension fields. We are specifically interested in extension fields of type $\mathbb{F}_{2^k}$, where $k > 1$. These are extensions of the binary field $\mathbb{F}_2$.

Example 3.3 *Addition and multiplication operations over $\mathbb{F}_2$:*

$+$	0	1
0	0	1
1	1	0

Addition over $\mathbb{F}_2$

$\cdot$	0	1
0	0	0
1	0	1

Multiplication over $\mathbb{F}_2$

Notice that addition over $\mathbb{F}_2$ is a Boolean XOR *operation, because it is performed modulo* 2. *Similarly, multiplication over $\mathbb{F}_2$ performs a Boolean* AND *operation.*

Algebraic extensions of the binary field $\mathbb{F}_2$ are generally termed as *binary extension fields* $\mathbb{F}_{2^k}$. Where elements in $\mathbb{F}_2$ can only represent 1 bit, elements in $\mathbb{F}_{2^k}$ represent a k-bit vector. This allows them to be widely used in digital hardware applications. In order to construct a Galois field of the form $\mathbb{F}_{2^k}$, an **irreducible polynomial** is required:

Definition 3.8 *A polynomial $P(x) \in \mathbb{F}_2[x]$ is* **irreducible** *if $P(x)$ is non-constant with degree k and cannot be factored into a product of polynomials of lower degree in $\mathbb{F}_2[x]$.*

Therefore, the polynomial $P(x)$ with degree k is irreducible over $\mathbb{F}_2$ if and only if it has no roots in $\mathbb{F}_2$, i.e if $\forall a \in \mathbb{F}_2$, $P(a) \neq 0$. For example, $x^2 + x + 1$ is an irreducible polynomial over $\mathbb{F}_2$ because it has no solutions in $\mathbb{F}_2$, i.e. $(0)^2 + (0) + 1 = 1 \neq 0$ and $(1)^2 + (1) + 1 = 1 \neq 0$ over $\mathbb{F}_2$. Irreducible polynomials exist for any degree ≥ 2 in $\mathbb{F}_2[x]$.

Given an irreducible polynomial $P(x)$ of degree k in the polynomial ring $\mathbb{F}_2[x]$, we can construct a binary extension field $\mathbb{F}_{2^k} \equiv \mathbb{F}_2[x] \pmod{P(x)}$. Let α be a root of $P(x)$, i.e., $P(\alpha) = 0$. Since $P(x)$ is irreducible over $\mathbb{F}_2[x]$, $\alpha \notin \mathbb{F}_2$. Instead, α is an element in $\mathbb{F}_{2^k}$. Any element $A \in \mathbb{F}_{2^k}$ is then represented as:

$$A = \sum_{i=0}^{k-1}(a_i \cdot \alpha^i) = a_0 + a_1 \cdot \alpha + \cdots + a_{k-1} \cdot \alpha^{k-1}$$

where $a_i \in \mathbb{F}_2$ are the coefficients and $P(\alpha) = 0$.

To better understand this field extension, compare its similarities to another commonplace field extension $\mathbb{C}$, the set of complex numbers. $\mathbb{C}$ is an extension of the field of real numbers $\mathbb{R}$ with an additional element $i = \sqrt{-1}$, which is an imaginary root in $\mathbb{R}$. Thus $i \notin \mathbb{R}$, rather $i \in \mathbb{C}$. Every element $A \in \mathbb{C}$ can be represented as:

$$A = \sum_{j=0}^{1}(a_j \cdot i^j) = a_0 + a_1 \cdot i \tag{3.2}$$

where $a_j \in \mathbb{R}$ are coefficients. Similarly, $\mathbb{F}_{2^k}$ is an extension of $\mathbb{F}_2$ with an additional element α, which is the "imaginary root" of an irreducible polynomial P in $\mathbb{F}_2[x]$.

Every element $A \in \mathbb{F}_{2^k}$ has a degree less than k because A is always computed modulo $P(x)$, which has degree k. Thus, $A \pmod{P(x)}$ can be of degree at most $k-1$ and at least 0. For this reason, the field $\mathbb{F}_{2^k}$ can be viewed as a k dimensional vector space over $\mathbb{F}_2$. The equivalent bit vector representation for element A is:

$$A = (a_{k-1}a_{k-2}\cdots a_0) \tag{3.3}$$

Example 3.4 *A 4-bit Boolean vector, $(a_3a_2a_1a_0)$ can be presented over $\mathbb{F}_{2^4}$ as:*

$$a_3 \cdot \alpha^3 + a_2 \cdot \alpha^2 + a_1 \cdot \alpha + a_0 \tag{3.4}$$

For instance, the Boolean vector 1011 is represented as the element $\alpha^3 + \alpha + 1$.

Example 3.5 *Let us construct $\mathbb{F}_{2^4}$ as $\mathbb{F}_2[x] \pmod{P(x)}$, where $P(x) = x^4 + x^3 + 1 \in \mathbb{F}_2[x]$ is an irreducible polynomial of degree $k = 4$. Let α be the root of $P(x)$, i.e. $P(\alpha) = 0$.*

Any element $A \in \mathbb{F}_2[x] \pmod{x^4 + x^3 + 1}$ has a representation of the type: $A = a_3x^3 + a_2x^2 + a_1x + a_0$ (degree < 4) where the coefficients $a_3, \ldots, a_0$ are in $\mathbb{F}_2 = \{0, 1\}$. Since there are only 16 such polynomials, we obtain 16 elements in the field $\mathbb{F}_{2^4}$. Each

Table 3.1: Bit-vector, Exponential and Polynomial representation of elements in $\mathbb{F}_{2^4} = \mathbb{F}_2[x] \pmod{x^4 + x^3 + 1}$

$a_3a_2a_1a_0$	Exponential	Polynomial	$a_3a_2a_1a_0$	Exponential	Polynomial
0000	0	0	1000	α^3	α^3
0001	1	1	1001	α^4	$\alpha^3 + 1$
0010	α	α	1010	α^{10}	$\alpha^3 + \alpha$
0011	α^{12}	$\alpha + 1$	1011	α^5	$\alpha^3 + \alpha + 1$
0100	α^2	α^2	1100	α^{14}	$\alpha^3 + \alpha^2$
0101	α^9	$\alpha^2 + 1$	1101	α^{11}	$\alpha^3 + \alpha^2 + 1$
0110	α^{13}	$\alpha^2 + \alpha$	1110	α^8	$\alpha^3 + \alpha^2 + \alpha$
0111	α^7	$\alpha^2 + \alpha + 1$	1111	α^6	$\alpha^3 + \alpha^2 + \alpha + 1$

element in $\mathbb{F}_{2^4}$ can then be viewed as a 4-bit vector over $\mathbb{F}_2$. Each element also has an exponential α representation. All three representations are shown in Table 3.1.

We can compute the polynomial representation from the exponential representation. Since every element is computed $\pmod{P(\alpha)} = \pmod{\alpha^4 + \alpha^3 + 1}$, we compute the element α^4 as

$$\alpha^4 \pmod{\alpha^4 + \alpha^3 + 1} = -\alpha^3 - 1 = \alpha^3 + 1 \tag{3.5}$$

Recall that all coefficients of $\mathbb{F}_{2^4}$ are in $\mathbb{F}_2$ where $-1 = +1$ modulo 2. The next element α^5 can be computed as

$$\alpha^5 = \alpha^4 \cdot \alpha = (\alpha^3 + 1) \cdot \alpha = \alpha^4 + \alpha = \alpha^3 + \alpha + 1 \tag{3.6}$$

Then α^6 can be computed as $\alpha^5 * \alpha$ and so on.

An irreducible polynomial can also be a primitive polynomial.

Definition 3.9 *A* **primitive polynomial** *$P(x)$ is a polynomial with coefficients in $\mathbb{F}_2$ which has a root $\alpha \in \mathbb{F}_{2^k}$ such that $\{0,\ 1(= \alpha^{2^k-1}),\ \alpha,\ \alpha^2,\ \cdots,\ \alpha^{2^k-2}\}$ is the set of all elements in $\mathbb{F}_{2^k}$, where α is a* **primitive element** *of $\mathbb{F}_{2^k}$.*

A primitive polynomial is guaranteed to generate all distinct elements of a finite field $\mathbb{F}_{2^k}$ while an irreducible polynomial has no such guarantee. Often, there exists more than one irreducible polynomial of degree k. In such cases, any degree k irreducible polynomial can be used for field construction. For example, both x^3+x+1 and x^3+x^2+1 are irreducible in $\mathbb{F}_2$ and either one can be used to construct $\mathbb{F}_{2^3}$. This is due to the following:

Theorem 3.1 *There exist a* **unique** *field $\mathbb{F}_{p^k}$, for any prime p and any positive integer k.*

Theorem 3.1 implies that Galois fields with the same number of elements are **isomorphic** to each other up to the labeling of the elements.

Theorem 3.2 provides an important property for investigating solutions to polynomial equations in $\mathbb{F}_q$.

Theorem 3.2 $[Generalized\ Fermat's\ Little\ Theorem]$ *Given a Galois field $\mathbb{F}_q$, each element $A \in \mathbb{F}_q$ satisfies:*

$$\begin{aligned} A^q &\equiv A \\ A^q - A &\equiv 0 \end{aligned} \tag{3.7}$$

We can extend Theorem 3.2 to polynomials in $\mathbb{F}_q[x]$ as follows:

Definition 3.10 *Let $x^q - x$ be a polynomial in $\mathbb{F}_q[x]$. Every element $A \in \mathbb{F}_q$ is a solution to $x^q - x = 0$. Therefore, $x^q - x$ always vanishes in $\mathbb{F}_q$. Such polynomials are called* **vanishing polynomials** *of the field $\mathbb{F}_q$.*

Example 3.6 *Given $\mathbb{F}_{2^2} = \{0, 1, \alpha, \alpha+1\}$ with $P(x) = x^2 + x + 1$, where $P(\alpha) = 0$.*

$$\begin{aligned} 0^{2^2} &= 0 \\ 1^{2^2} &= 1 \\ \alpha^{2^2} &= \alpha \pmod{\alpha^2+\alpha+1} \\ (\alpha+1)^{2^2} &= \alpha+1 \pmod{\alpha^2+\alpha+1} \end{aligned}$$

3.2.1 Containment of Galois Fields

A Galois field $\mathbb{F}_q$ can be fully contained within a larger field $\mathbb{F}_{q^k}$. That is, $\mathbb{F}_q \subset \mathbb{F}_{q^k}$. For example, Fig 3.1 shows the containment of the fields $\mathbb{F}_2 \subset \mathbb{F}_4 \subset \mathbb{F}_{16}$. It's easy to see that since $\mathbb{F}_4 = \mathbb{F}_{2^2}$, it contains $\mathbb{F}_2$. Likewise $\mathbb{F}_{16} = \mathbb{F}_{4^2} = \mathbb{F}_{2^4}$ contains $\mathbb{F}_4$ and $\mathbb{F}_2$.

The elements $\{0, 1, \alpha, \ldots, \alpha^{14}\}$ designate $\mathbb{F}_{16}$. Of these, $\{0, 1, \alpha^5, \alpha^{10}\}$ create $\mathbb{F}_4$. From these, only $\{0, 1\}$ exist in $\mathbb{F}_2$.

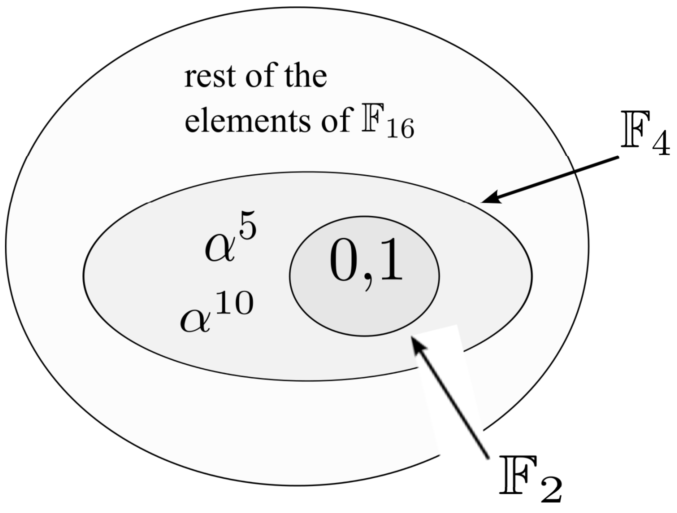


Figure 3.1: Containment of Fields: $\mathbb{F}_2 \subset \mathbb{F}_4 \subset \mathbb{F}_{16}$

Consider the element α^5 of $\mathbb{F}_{16}$. Deriving all $(\alpha^5)^i$ for $i \geq 0$ over $\mathbb{F}_{16}$ gives the following recurrence:

$$\begin{aligned}(\alpha^5)^0 &= 1\\(\alpha^5)^1 &= \alpha^5\\(\alpha^5)^2 &= \alpha^{10}\\(\alpha^5)^3 &= \alpha^{15} = 1\end{aligned} \tag{3.8}$$

The only elements that are generated in this recurrence are $\{1, \alpha^5, \alpha^{10}\}$. Every field contains $\{0, 1\}$, so the elements $\{0, 1, \alpha^5, \alpha^{10}\}$ form $\mathbb{F}_4$. Let $P(x) = x^4 + x^3 + 1$ be the primitive polynomial used to generate $\mathbb{F}_{2^4} = \mathbb{F}_{16}$. A primitive polynomial of degree 2 used to generate $\mathbb{F}_{2^2} = \mathbb{F}_4$ can be found as follows:

$$\begin{aligned}&(x + \alpha^5) \cdot (x + \alpha^{10}) \mod P(x)\\=\ &x^2 + (\alpha^{10} + \alpha^5)x + \alpha^{15} \mod P(x)\\=\ &x^2 + x + 1\end{aligned} \tag{3.9}$$

Theorem 3.3 *$\mathbb{F}_{2^n} \subset \mathbb{F}_{2^m}$ iff $n \mid m$, i.e. if n divides m.*

Therefore:

- $\mathbb{F}_2 \subset \mathbb{F}_{2^2} \subset \mathbb{F}_{2^4} \subset \mathbb{F}_{2^8} \subset \dots$
- $\mathbb{F}_2 \subset \mathbb{F}_{2^3} \subset \mathbb{F}_{2^9} \subset \mathbb{F}_{2^{27}} \subset \dots$
- $\mathbb{F}_2 \subset \mathbb{F}_{2^5} \subset \mathbb{F}_{2^{25}} \subset \mathbb{F}_{2^{125}} \subset \dots$, and so on

Definition 3.11 *The* **algebraic closure** *of the Galois field $\mathbb{F}_{2^k}$, denoted $\overline{\mathbb{F}_{2^k}}$, is the union of all fields $\mathbb{F}_{2^n}$ such that $k \mid n$.*

3.2.2 Polynomial Interpolation over Galois Fields

In the construction of digital circuits, arbitrary mappings between two bit-vectors of size k can be constructed. Each such mapping generates a function $f : \mathbb{B}^k \to \mathbb{B}^k$. As every k-bit vector can be construed as an element in $\mathbb{F}_{2^k}$ (as shown in the previous section), every such function also corresponds to a function over a Galois field: $f : \mathbb{F}_{2^k} \to \mathbb{F}_{2^k}$.

Definition 3.12 *A function $f : \mathbb{R} \to \mathbb{R}$ over a ring R is considered a* **polynomial function** *if there exists a polynomial $\mathcal{F} \in \mathbb{R}[x_1, \dots, x_d]$ such that $\mathcal{F}(x_1, \dots, x_d) = f(x_1, \dots, x_d)$.*

Theorem 3.4 *From [87]: Let $\mathbb{F}_q$ be a Galois field of q elements where q is a power of a prime integer. Given any function $f : \mathbb{F}_q \to \mathbb{F}_q$, there exists a polynomial $\mathcal{F} \in \mathbb{F}_q[x]$ such that $f(a) = \mathcal{F}(a)$, for all $a \in \mathbb{F}_q$. Thus, every function $f : \mathbb{F}_q \to \mathbb{F}_q$ is a polynomial function.*

Thus, since every function over a Galois field, $f : \mathbb{F}_{2^k} \to \mathbb{F}_{2^k}$, is a polynomial function, every mapping between two bit-vectors of size k is a polynomial function over $\mathbb{F}_{2^k}$. Furthermore, every polynomial can be derived using Lagrange interpolation.

Theorem 3.5 *(***Lagrange Interpolation***):*

Given a set of k data points over a function f,

$$(x_0, f(x_0)), \ldots, (x_{k-1}, f(x_{k-1}))$$

where no two $x_i \in \{x_0, \ldots, x_{k-1}\}$ are the same elements, the polynomial representation of f, $\mathcal{F}(x)$, can be interpolated as follows:

$$\mathcal{F}(x) = \sum_{i=0}^{k-1} f(x_i) \cdot L_i(x)$$

$$L_i(x) = \prod_{(0 \le j \le k-1),(j \ne i)} \frac{x - x_j}{x_i - x_j}$$

By applying Lagrange interpolation over every element in the Galois field $\mathbb{F}_{2^k}$, we can derive the polynomial representation $\mathcal{F}$ of any function $f : \mathbb{F}_{2^k} \to \mathbb{F}_{2^k}$. Furthermore, $\mathcal{F}$ is a polynomial of degree at most $2^k - 1$ in x and $\mathcal{F}(a) = f(a)$ for all $a \in \mathbb{F}_{2^k}$.

While every function over a Galois field is a polynomial function, not every function over the integer ring $\mathbb{Z}$ is a polynomial function.

Example 3.7 *Let $A = \{a_2, a_1, a_0\}$ and $Z = \{z_2, z_1, z_0\}$ be 3-bit vectors. Thus, A and $Z \in \mathbb{B}^3$. Consider the following function:*

$$f : Z[2:0] = A[2:0] >> 1$$

f is a **bit-vector right shift** *operation on A. This function can be analyzed as a mapping over different forms: $\mathbb{B}^3 \to \mathbb{B}^3$, $\mathbb{Z}_8 \to \mathbb{Z}_8$, and $\mathbb{F}_{2^3} \to \mathbb{F}_{2^3}$. These mappings from A to Z are:*

$\{a_2a_1a_0\} \in \mathbb{B}^3$	$A \in \mathbb{Z}_8$	$A \in \mathbb{F}_{2^3}$	$\to$	$\{z_2z_1z_0\} \in \mathbb{B}^3$	$Z \in \mathbb{Z}_8$	$Z \in \mathbb{F}_{2^3}$
000	*0*	0	$\to$	*000*	*0*	0
001	*1*	1	$\to$	*000*	*0*	0
010	*2*	α	$\to$	*001*	*1*	1
011	*3*	$\alpha+1$	$\to$	*001*	*1*	1
100	*4*	α^2	$\to$	*010*	*2*	α
101	*5*	α^2+1	$\to$	*010*	*2*	α
110	*6*	$\alpha^2+\alpha$	$\to$	*011*	*3*	$\alpha+1$
111	*7*	$\alpha^2+\alpha+1$	$\to$	*011*	*3*	$\alpha+1$

$f : \mathbb{Z}_8 \to \mathbb{Z}_8$ is not a polynomial function (this can be verified using the results of [91] [92] [93]). However, $f : \mathbb{F}_{2^3} \to \mathbb{F}_{2^3}$ is a polynomial function. By applying Lagrange's

interpolation formula to f over $\mathbb{F}_{2^3}$ for every element in $\mathbb{F}_{2^3}$, we obtain the following polynomial function: $Z = (\alpha^2 + 1)A^4 + (\alpha^2 + 1)A^2$, where $P(\alpha) = \alpha^3 + \alpha + 1 = 0$.

Since every function over $\mathbb{F}_{2^k}$ is a polynomial function, the functional mapping of a Galois field arithmetic circuit over $\mathbb{F}_{2^k}$ must exist in polynomial form. Construction of these arithmetic circuits is described next.

3.3 Hardware Implementations of Arithmetic Operations Over Galois Fields

There are two main applications of hardware implementations of Galois field arithmetic. In the first case, Galois field arithmetic computations, such as ADD OR MUL, are implemented in hardware, and algorithms are then implemented in software (e.g. cryptoprocessors [94] [95]). In other cases, the entire design can be implemented in hardware, such as a one-shot Reed-Solomon encoder-decoder chip [96] [97], or point multiplication circuitry [98] used in elliptic curve cryptosystems. Therefore, there has been extensive research in efficient hardware design of primitive arithmetic computations over Galois fields. In this section, we describe the design principles of such circuits with focus on their architecture and verification complexity.

Addition in $\mathbb{F}_{2^k}$ is performed by correspondingly adding the polynomials together and reducing the coefficients of the result modulo 2.

Example 3.8 *Given $A = \alpha^3 + \alpha^2 + 1 = (1101)$ and $B = \alpha^2 + 1 = (0101)$ in $\mathbb{F}_{2^4}$,*

$$A + B = (\alpha^3 + \alpha^2 + 1) + (\alpha^2 + 1) = (\alpha^3) + (\alpha^2 + \alpha^2) + (1 + 1) = \alpha^3 = (1000).$$

Effectively, the addition operation is only performed on the coefficients, which are in $\mathbb{F}_2$. As addition over $\mathbb{F}_2$ performs an *XOR* operation, constructing an addition circuit over $\mathbb{F}_{2^k}$ is trivial as it only consists of k number of *XOR* gates. A 4-bit adder over $\mathbb{F}_{2^4}$ is shown in Fig. 3.2.

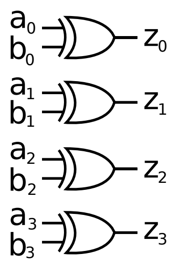


Figure 3.2: 4-bit adder over $\mathbb{F}_{2^4}$.

Multiplication $Z = A \times B \pmod{P(x)}$ in $\mathbb{F}_{2^k}$ conceptually consists of two steps. In the first step, the multiplication $A \times B$ is performed. In the second step, the result is reduced modulo the irreducible polynomial $P(x)$. This multiplication procedure is shown in Example 3.9.

Example 3.9 *Consider the field* $\mathbb{F}_{2^4}$ *with the irreducible polynomial* $P(x) = x^4 + x^3 + 1$ *and* $P(\alpha) = 0$. *We take as inputs:* $A = a_0 + a_1 \cdot \alpha + a_2 \cdot \alpha^2 + a_3 \cdot \alpha^3$ *and* $B = b_0 + b_1 \cdot \alpha + b_2 \cdot \alpha^2 + b_3 \cdot \alpha^3$. *We have to perform the multiplication* $Z = A \times B \pmod{P(x)}$. *The coefficients of* $A = \{a_0, \ldots, a_3\}, B = \{b_0, \ldots, b_3\}$ *are in* $\mathbb{F}_2 = \{0, 1\}$. *This multiplication can be performed as shown:*

			a_3	a_2	a_1	a_0
$\times$			b_3	b_2	b_1	b_0
			$a_3 \cdot b_0$	$a_2 \cdot b_0$	$a_1 \cdot b_0$	$a_0 \cdot b_0$
		$a_3 \cdot b_1$	$a_2 \cdot b_1$	$a_1 \cdot b_1$	$a_0 \cdot b_1$	
	$a_3 \cdot b_2$	$a_2 \cdot b_2$	$a_1 \cdot b_2$	$a_0 \cdot b_2$		
$a_3 \cdot b_3$	$a_2 \cdot b_3$	$a_1 \cdot b_3$	$a_0 \cdot b_3$			
s_6	s_5	s_4	s_3	s_2	s_1	s_0

The result $Sum = s_0 + s_1 \cdot \alpha + s_2 \cdot \alpha^2 + s_3 \cdot \alpha^3 + s_4 \cdot \alpha^4 + s_5 \cdot \alpha^5 + s_6 \cdot \alpha^6$, *where*

$$\begin{aligned}
s_0 &= a_0 \cdot b_0 \\
s_1 &= a_0 \cdot b_1 + a_1 \cdot b_0 \\
s_2 &= a_0 \cdot b_2 + a_1 \cdot b_1 + a_2 \cdot b_0 \\
s_3 &= a_0 \cdot b_3 + a_1 \cdot b_2 + a_2 \cdot b_2 + a_3 \cdot b_1 \\
s_4 &= a_1 \cdot b_3 + a_2 \cdot b_1 + a_3 \cdot b_1 \\
s_5 &= a_2 \cdot b_3 + a_3 \cdot b_2 \\
s_6 &= a_3 \cdot b_3
\end{aligned}$$

Here the multiply "$\cdot$" and add "$+$" operations are performed modulo 2, so they can be implemented in a circuit using AND and XOR gates respectively. Note that unlike integer multipliers, there are no carry-chains in the design, as the coefficients are always reduced modulo 2*. However, the result is yet to be reduced modulo the primitive polynomial* $P(x) = x^4 + x^3 + 1$*. This transforms every exponent representation,* α^d*, to a polynomial representation where* $d \geq k = 4$*.*

α^3	α^2	α	1	
s_3	s_2	s_1	s_0	
s_4	0	0	s_4	$s_4 \cdot \alpha^4 \pmod{P(\alpha)} = s_4 \cdot (\alpha^3 + 1)$
s_5	0	s_5	s_5	$s_5 \cdot \alpha^5 \pmod{P(\alpha)} = s_5 \cdot (\alpha^3 + \alpha + 1)$
s_6	s_6	s_6	s_6	$s_6 \cdot \alpha^6 \pmod{P(\alpha)} = s_6 \cdot (\alpha^3 + \alpha^2 + \alpha + 1)$
z_3	z_2	z_1	z_0	

The final result (output) of the circuit is: $Z = z_0 + z_1\alpha + z_2\alpha^2 + z_3\alpha^3$*; where* $z_0 = s_0 + s_4 + s_5 + s_6$; $z_1 = s_1 + s_5 + s_6$; $z_2 = s_2 + s_6$; $z_3 = s_3 + s_4 + s_5 + s_6$.

The above multiplier design is called the *Mastrovito multiplier* [63] which is the most straightforward way to design a multiplier over $\mathbb{F}_{2^k}$. A logic circuit for a 4-bit *Mastrovito* multiplier over *Galois field* $\mathbb{F}_{2^4}$ is illustrated in Fig. 3.3.

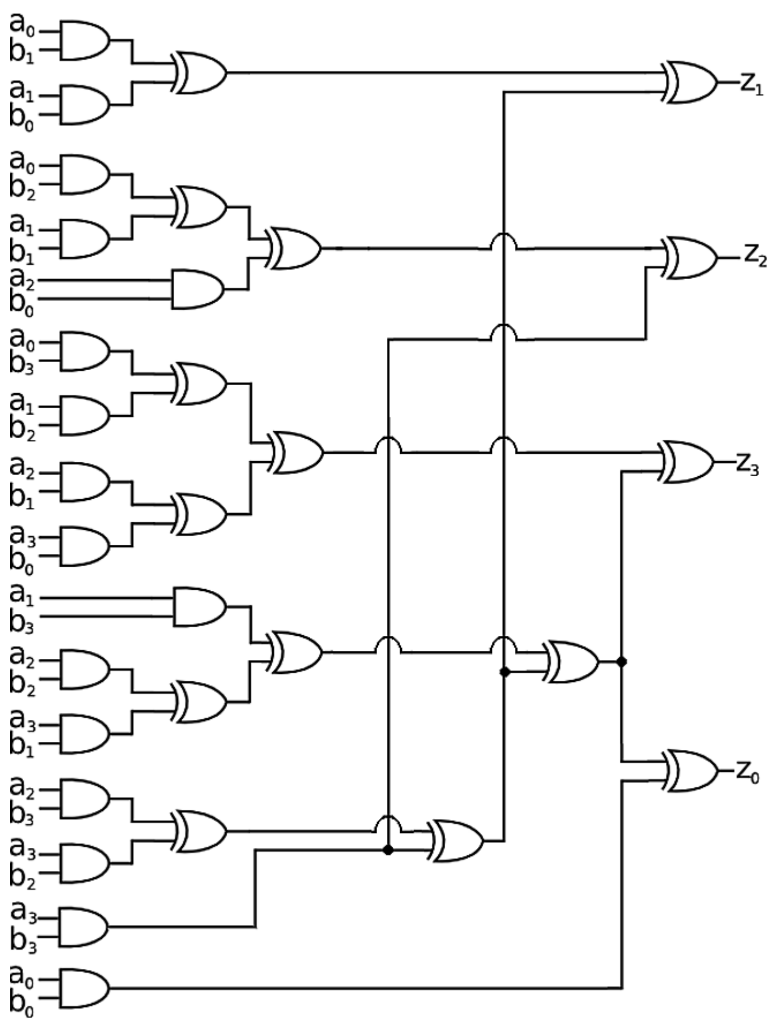


Figure 3.3: Mastrovito multiplier over $\mathbb{F}_{2^4}$.

Modular multiplication is at the heart of many public-key cryptosystems, such as Elliptic Curve Cryptography (ECC) [99]. Due to the very large field size (and hence the data-path width) used in these cryptosystems, the above *Mastrovito* multiplier architecture is inefficient, especially when exponentiation and repeat multiplications are performed. Therefore, efficient hardware and software implementations of modular multiplication algorithms are used to overcome the complexity of such operations. One such algorithm which we will focus on is the Montgomery reduction [88] [64].

3.3.1 Montgomery Multipliers

Montgomery Reduction (MR) computes:

$$G = MR(A, B) = A \cdot B \cdot R^{-1} \pmod{P(x)} \tag{3.10}$$

where A, B are k-bit inputs, $R = \alpha^k$, R^{-1} is multiplicative inverse of R in $\mathbb{F}_{2^k}$, and $P(x)$ is the irreducible polynomial for $\mathbb{F}_{2^k}$. Since Montgomery reduction cannot directly compute $A \cdot B$, we need to pre-compute $A \cdot R$ and $B \cdot R$, as shown in Fig. 3.4.

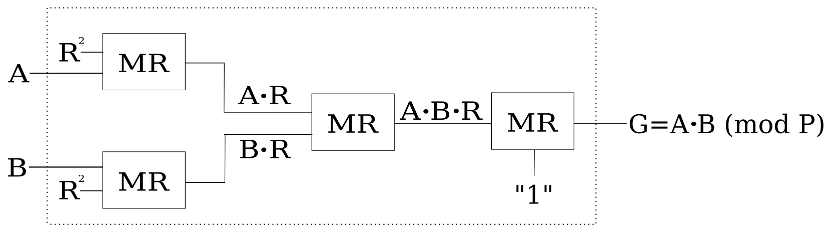


Figure 3.4: Montgomery multiplier over $\mathbb{F}_{2^k}$

Each MR block in Fig. 3.4 represents a Montgomery reduction step which is a hardware implementation of the algorithm shown in Algorithm 1.

Algorithm 1: Montgomery Reduction Algorithm [64]

```
Input: $A(x), B(x) \in \mathbb{F}_{2^k}$; irreducible polynomial $P(x)$.
Output: $G(x) = A(x) \cdot B(x) \cdot x^{-k} \pmod{P(x)}$.
$G(x)$ :=0
for ($i = 0$; $i \leq k - 1$; ++$i$ ) do
    $G(x) := G(x) + A_i \cdot B(x)$ /*$A_i$ is the $i^{th}$ bit of $A$*/;
    $G(x) := G(x) + G_0 \cdot P(x)$ /*$G_0$ is the lowest bit of $G$*/;
    $G(x) := G(x)/x$ /*Right shift 1 bit*/;
end
```

The design of Fig. 3.4 is not efficient to computing $A \cdot B \pmod{P(x)}$ when compared to the Mastrovito implementation. However, when these multiplications are performed repeatedly, such as in iterative squaring, then the Montgomery approach speeds-up the computation. As shown in [89], the critical path delay and gate counts of a

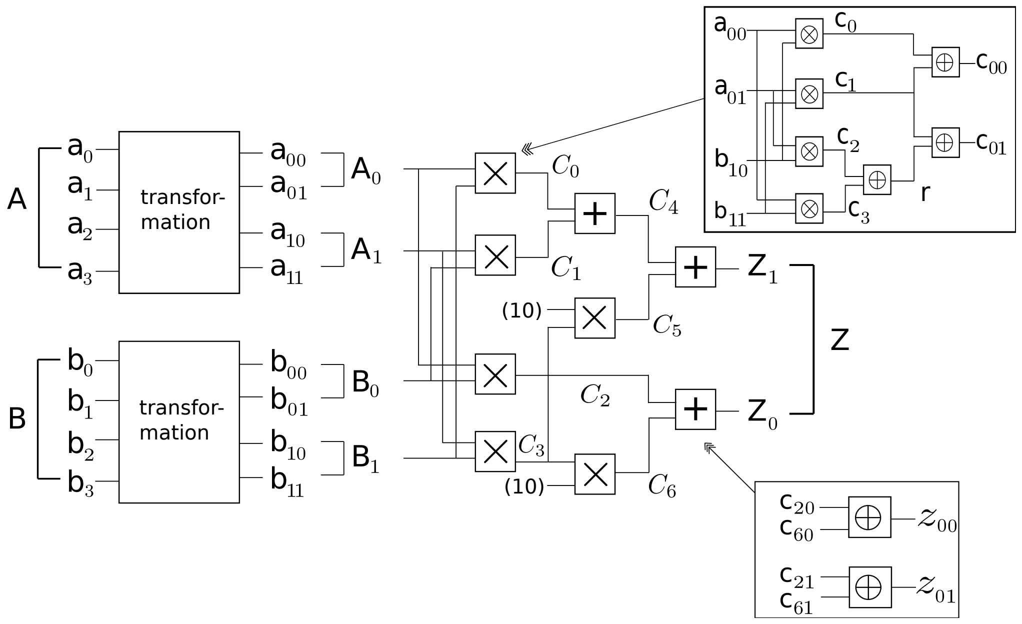


Figure 3.5: 4-bit composite multiplier designed over $\mathbb{F}_{(2^2)^2}$

squarer designed using the Montgomery approach are much smaller than the traditional approaches.

3.3.2 Circuit Designs over Composite Fields

The Galois field $\mathbb{F}_{2^k}$ is a k-dimensional vector space over the sub-field $\mathbb{F}_2$. If $k = m \cdot n$, the field $\mathbb{F}_{2^k}$ can be decomposed as $\mathbb{F}_{(2^m)^n}$. Such a field representation is called a **composite field**, and it is constructed as a n-dimensional extension of the sub-field $\mathbb{F}_{2^m}$. The sub-field $\mathbb{F}_{2^m}$ is called the ground field. Note that we have $\mathbb{F}_2 \subset \mathbb{F}_{2^m} \subset \mathbb{F}_{(2^m)^n}$.

A Galois field arithmetic circuit over $\mathbb{F}_{2^k}$ can thus be composed as circuit over $\mathbb{F}_{(2^m)^n}$ if $k = m \cdot n$. Since the base field is $\mathbb{F}_{2^m}$, this composite field circuit is composed of blocks of m-bit multipliers and adders, along with m-bit buses that act as the inputs and outputs of these blocks. A $\mathbb{F}_{2^4}$ Galois field multiplier designed over the composite field $\mathbb{F}_{(2^2)^2}$ is shown in Fig. 3.5. Design methodologies of these circuits are examined more closely in Chapter 7.

3.3.3 Applications to Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) is one of the most influential applications of Galois fields. ECC is an approach to public-key (or asymmetric-key) cryptography based on the algebraic structure of elliptic curves over Galois fields. Due to the complex nature of these curves, key sizes in ECC can be smaller than other public-key cryptography techniques while providing the same level of security [100]. The main operations of encryption, decryption and authentication in ECC rely on *point multiplications*.

Point multiplication involves a series of addition and doubling of points on the elliptic curve. A drawback of traditional point multiplication is that each point addition and doubling require a multiplicative inverse operation over Galois fields, the computation of which is costly. Modern methods, however, represent the points in projective coordinate systems [98], which has eliminated the need for a multiplicative inverse operation by replacing it with addition and multiplication operations over Galois fields. This has increased the efficiency of point multiplication operations, but it has also increased the need for fast, custom hardware designs of Galois field arithmetic.

In-depth analysis of elliptic curve theory is beyond the scope of this dissertation. Instead, we will look at some examples of point addition and point doubling to give a general idea of the operations involved in ECC and how they apply to Galois field arithmetic. Our experiments use custom Galois field arithmetic designs based on L*ó*pez-Dahab (LD) coordinate system [101], so these examples will use the same coordinate system.

Example 3.10 *Consider point addition in a LD projective coordinate system, as seen in Fig. 3.6.*

Given an elliptic curve: $Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4$ *over* $\mathbb{F}_{2^k}$*, where* X, Y, Z *are* k*-bit vectors that are elements in* $\mathbb{F}_{2^k}$ *and similarly,* a, b *are constants from the field. Let* $P + Q = R$ *represent point addition over the elliptic curve.* $P = (X_1, Y_1, Z_1)$ *and* $Q = (X_2, Y_2, 1)$ *are given. Then* $R = (X_3, Y_3, Z_3)$ *can be computed as follows:*

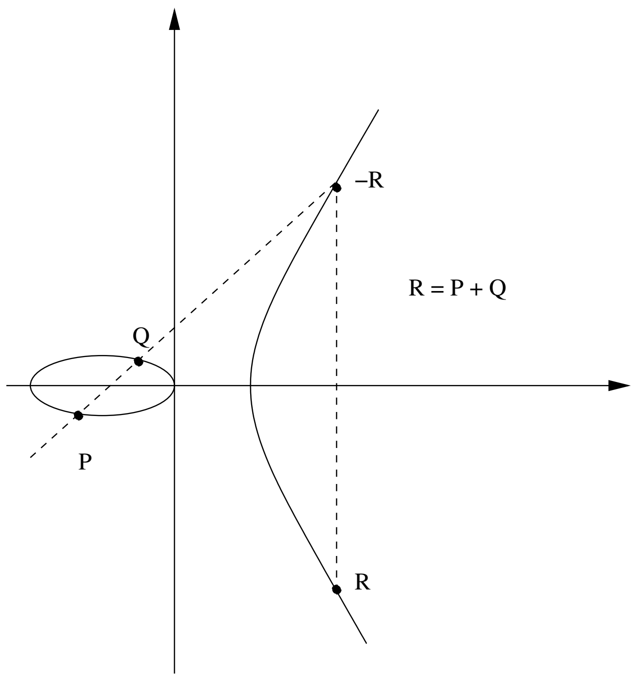


Figure 3.6: Point addition over an Elliptic Curve (R=P+Q)

$$
\begin{aligned}
A &= Y_2 \cdot Z_1^2 + Y_1\\
B &= X_2 \cdot Z_1 + X_1\\
C &= Z_1 \cdot B\\
D &= B^2 \cdot (C + aZ_1^2)\\
Z_3 &= C^2\\
E &= A \cdot C\\
X_3 &= A^2 + D + E\\
F &= X_3 + X_2 \cdot Z_3\\
G &= X_3 + Y_2 \cdot Z_3\\
Y_3 &= E \cdot F + Z_3 \cdot G
\end{aligned}
$$

Example 3.11 *Consider point doubling in a LD projective coordinate system. Given an elliptic curve:* $Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4$. *Let 2(*$X_1$*,* Y_1*,* Z_1*) = (*X_3*,* Y_3*,* Z_3*),* *then*

$$\begin{aligned}
X_3 &= X_1^4 + b \cdot Z_1^4 \\
Z_3 &= X_1^2 \cdot Z_1^2 \\
Y_3 &= bZ_1^4 \cdot Z_3 + X_3 \cdot (aZ_3 + Y_1^2 + bZ_1^4)
\end{aligned}$$

In the above examples, polynomial multiplication and squaring operations can be implemented in hardware using Montgomery reductions over Galois fields $\mathbb{F}_{2^k}$. In practical applications, the field size k of $\mathbb{F}_{2^k}$ is 163, or larger. However, there are no word-level abstraction techniques applicable to circuits of such size, so hardware implementations of Galois field arithmetic circuits cannot benefit from the many advantages of abstraction. Thus, we propose a computer-algebra approach to word-level polynomial abstractions of Galois field arithmetic circuits. Recent computer-algebra formal verification techniques [11] have been able to verify these circuits up to 163 bits. We propose an application of our abstraction approach to improve these techniques. These improvements allow us to perform formal verification of these circuits up to 571 bits. These proposals are described in detail in subsequent chapters.

CHAPTER 4

COMPUTER ALGEBRA FUNDAMENTALS

This chapter reviews fundamental concepts of commutative and computer algebra which are used in this work. Specifically, this chapter covers monomial ordering, polynomial ideals and varieties, and the computation of Gröbner bases. It also overviews elimination theory as well as Hilbert's Nullstellensatz theorems and how they apply to Galois fields. The results of these theorems are used in polynomial abstraction and formal verification of Galois field circuits and are discussed in subsequent chapters. The material of this chapter is mostly referred from the textbooks [102] [10] and previous work by *Lv* [11].

4.1 Monomials, Polynomials, and Term Orderings

Definition 4.1 *A* **monomial** *in variables* $x_1, x_2, \cdots, x_d$ *is a product of the form:*

$$x_1^{\alpha_1} \cdot x_2^{\alpha_2} \cdot \cdots \cdot x_d^{\alpha_d}, \tag{4.1}$$

where $\alpha_i \geq 0, i \in \{1, \cdots, d\}$. *The total degree of the monomial is* $\alpha_1 + \cdots + \alpha_d$.

Thus, $x^2 \cdot y$ is a monomial in variables x, y with total degree 3. For simplicity, we will henceforth denote a monomial $x_1^{\alpha_1} \cdot x_2^{\alpha_2} \cdot \cdots \cdot x_d^{\alpha_d}$ as x^α, where $\alpha = (\alpha_1, \cdots, \alpha_d)$ is a vector size d of integers ≥ 0, i.e., $\alpha \in \mathbb{Z}^d_{\geq 0}$. In the

Definition 4.2 *A* **multivariate polynomial** f *in variables* $x_1, x_2, \ldots, x_d$ *with coefficients in any given field* $\mathbb{K}$ *is a finite linear combination of monomials with coefficients in* $\mathbb{K}$*:*

$$f = \sum_\alpha a_\alpha \cdot x^\alpha, \;\; a_\alpha \in \mathbb{K}$$

The set of all polynomials in $x_1, x_2, \ldots, x_d$ *with coefficients in field* $\mathbb{K}$ *is denoted by* $\mathbb{K}[x_1, x_2, \ldots, x_d]$. *Thus,* $f \in \mathbb{K}[x_1, x_2, \ldots, x_d]$

1. *We refer to the constant $a_\alpha \in \mathbb{K}$ as the* **coefficient** *of the monomial $a_\alpha x^\alpha$.*

2. *If $a_\alpha \neq 0$, we call $a_\alpha x^\alpha$ a term of f.*

As an example, $2x^2 + y$ is a polynomial with two terms $2x^2$ and y, with 2 and 1 as coefficients respectively. In contrast, $x + y^{-1}$ is not a polynomial because the exponent of y is less than 0.

Since a polynomial is a sum of its terms, these terms have to be arranged unambiguously so that they can be manipulated in a consistent manner. Therefore, we need to establish a concept of **term ordering** (also called monomial ordering). A term ordering, represented by $>$, defines how terms in a polynomial are ordered.

Definition 4.3 *Let $\mathbb{T}^d = \{x^\alpha : \alpha \in \mathbb{Z}^d_{\geq 0}\}$ be the set of all monomials in $x_1, \ldots, x_d$. A* **monomial order** *$>$ on $\mathbb{T}^d$ is a total well-ordering satisfying:*

- *For all $x^\alpha, x^\beta \in \mathbb{T}^d$, x^α and x^β are comparable*

- *For any $x^\alpha \in \mathbb{T}^d$, $x^\alpha > 1$*

- *For all $x^\alpha, x^\beta, x^\gamma \in \mathbb{T}^d$, $x^\alpha > x^\beta \Rightarrow x^\alpha \cdot x^\gamma > x^\beta \cdot x^\gamma$*

Term-orderings are totally ordered, i.e. anti-symmetric with constant terms last in the ordering. A total-order ensures that there is no ambiguity with respect to where a term is found in the term-ordering. Total orderings for monomials come in different forms, notably lexicographic orderings (lex), and its variants: degree-lexicographic ordering (deglex) and reverse degree-lexicographic ordering (degrevlex).

A **lexicographic ordering** (lex) is a total-ordering $>$ such that variables in the terms are lexicographically ordered, i.e. simply based on when the variables appear in the ordering. Higher variable-degrees take precedence over lower degrees for equivalent variables (e.g. $a^3 > a^2$ due to $a \cdot a \cdot a > a \cdot a \cdot 1$).

Definition 4.4 **Lexicographic order:** *Let $x_1 > x_2 > \cdots > x_d$ lexicographically. Also let $\alpha = (\alpha_1, \ldots, \alpha_d)$; $\beta = (\beta_1, \ldots, \beta_d) \in \mathbb{Z}^d_{\geq 0}$. Then we have:*

$$x^\alpha > x^\beta \iff \left\{ \begin{array}{l} \textit{Starting from the left, the first co-ordinates of } \alpha_i, \beta_i \\ \textit{that are different satisfy } \alpha_i > \beta_i \end{array} \right. \tag{4.2}$$

A **degree-lexicographic ordering** (deglex) is a total-ordering $>$ such that the total degree of a term takes precedence over the lexicographic ordering. A **degree-reverse-lexicographic ordering** (degrevlex) is the same as a deglex ordering, however terms are lexed in reverse.

Definition 4.5 Degree Lexicographic order: *Let $x_1 > x_2 > \cdots > x_d$ lexicographically. Also let $\alpha = (\alpha_1, \ldots, \alpha_d);\ \beta = (\beta_1, \ldots, \beta_d) \in \mathbb{Z}^d_{\geq 0}$. Then we have:*

$$x^\alpha > x^\beta \iff \begin{cases} \sum_{i=1}^d \alpha_i > \sum_{i=1}^d \beta_i & \textit{or} \\ \sum_{i=1}^d \alpha_i = \sum_{i=1}^d \beta_i \textit{ and } x^\alpha > x^\beta & \textit{w.r.t. lex order} \end{cases} \tag{4.3}$$

Definition 4.6 Degree Reverse Lexicographic order: *Let $x_1 > x_2 > \cdots > x_d$ lexicographically. Also let $\alpha = (\alpha_1, \ldots, \alpha_d);\ \beta = (\beta_1, \ldots, \beta_d) \in \mathbb{Z}^d_{\geq 0}$. Then we have:*

$$x^\alpha > x^\beta \iff \begin{cases} \sum_{i=1}^d \alpha_i > \sum_{i=1}^d \beta_i \textit{ or} \\ \sum_{i=1}^d \alpha_i = \sum_{i=1}^d \beta_i \textit{ and the first co-ordinates} \\ \alpha_i, \beta_i \textit{ from the right, which are different, satisfy } \alpha_i < \beta_i \end{cases} \tag{4.4}$$

Applying these term orderings, we have the following relations, where $a > b > c$.

$$\text{lex:} a^2b > a^2 > abc > ab > ac^2 > ac > b^2c > b^2 > bc^3 > 1 \tag{4.5}$$

$$\text{deglex:} bc^3 > a^2b > abc > ac^2 > b^2c > a^2 > ab > ac > b^2 > 1 \tag{4.6}$$

$$\text{degrevlex:} bc^3 > a^2b > abc > b^2c > ac^2 > a^2 > ab > b^2 > ac > 1 \tag{4.7}$$

The difference between the *lex* and two *deg-* orderings is obvious, while the difference between the two degree-based orderings can be seen by considering from which direction the term is lexed, e.g. $a \cdot c \cdot c > b \cdot b \cdot c$ (deglex, left-to-right) versus $b \cdot b \cdot c > a \cdot c \cdot c$ (degrevlex, right-to-left).

Example 4.1 *Let $f = 2x^2yz + 3xy^3 - 2x^3$. The effects of different term orderings on f are:*

- *lex $x > y > z$: $f = -2x^3 + 2x^2yz + 3xy^3$*
- *deglex $x > y > z$: $f = 2x^2yz + 3xy^3 - 2x^3$*

- *degrevlex* $x > y > z$: $f = 3xy^3 + 2x^2yz - 2x^3$

Definition 4.7 *The* **leading term** *is the first term in a term-ordered polynomial. Likewise, the* **leading coefficient** *is the coefficient of the leading term. Finally, a* **leading monomial** *is the leading term lacking the coefficient. We use the following notation:*

$$lt(f) \quad \text{— Leading Term} \tag{4.8}$$

$$lc(f) \quad \text{— Leading Coefficient} \tag{4.9}$$

$$lm(f) \quad \text{— Leading Monomial} \tag{4.10}$$

$$tail(f) \quad f - lt(f) \tag{4.11}$$

Example 4.2

$$f = 3a^2b + 2ab + 4bc \tag{4.12}$$

$$lt(f) = 3a^2b \tag{4.13}$$

$$lc(f) = 3 \tag{4.14}$$

$$lm(f) = a^2b \tag{4.15}$$

$$tail(f) = 2ab + 4bc \tag{4.16}$$

Polynomial division is an operation over polynomials that is dependent on the imposed monomial ordering. Dividing a polynomial f by another polynomial g cancels the leading term of f to derive a new polynomial.

Definition 4.8 *Let $\mathbb{K}$ be a field and let $f, g \in \mathbb{K}[x_1, x_2, \ldots, x_d]$ be polynomials over the field.* **Polynomial division** *of f by g is computes following:*

$$f - \frac{lt(f)}{lt(g)} \cdot g \tag{4.17}$$

This polynomial division is denoted

$$f \xrightarrow{g} r \tag{4.18}$$

where r is the resulting polynomial of the division. If $\frac{lt(f)}{lt(g)}$ is non-zero, then f is considered divisible by g, i.e. $g \mid f$.

Notice that if $g \nmid f$, that is if f is not divisible by g, then the division operation gives $r = f$.

Example 4.3 *Over $\mathbb{R}[x, y, z]$, set the lex term order $x > y > z$. Let $f = -2x^3 + 2x^2yz + 3xy^3$ and $g = x^2 + yz$.*

$$\frac{lt(f)}{lt(g)} = \frac{-2x^3}{x^2} = -2x \tag{4.19}$$

Since $\frac{lt(f)}{lt(g)}$ is non-zero $g|f$. The division, $f \xrightarrow{g} r$, is computed as:

$$\begin{aligned} r &= f - \frac{lt(f)}{lt(g)} \cdot g = -2x^3 + 2x^2yz + 3xy^3 - (-2x \cdot (x^2 + yz)) \\ &= -2x^3 + 2x^2yz + 3xy^3 - (-2x^3 - 2xyz) = 2x^2yz + 3xy^3 + 2xyz \end{aligned} \tag{4.20}$$

Notice that the division cancels the leading term of f.

4.2 Varieties and Ideals

In computer-algebra based formal verification, it is often necessary to analyze the presence or absence of solutions to a given system of constraints. In our applications, these constraints are polynomials and their solutions are modeled as **varieties**.

Definition 4.9 *Let $\mathbb{K}$ be a field, and let $f_1, \ldots, f_s \in \mathbb{K}[x_1, x_2, \ldots, x_d]$. We call $V(f_1, \ldots, f_s)$ the* **affine variety** *defined by $f_1, \ldots, f_s$ as:*

$$V(f_1, \ldots, f_s) = \{(a_1, \ldots, a_d) \in \mathbb{K}^d : f_i(a_1, \ldots, a_d) = 0, \forall i, 1 \leq i \leq s\} \tag{4.21}$$

$V(f_1, \ldots, f_s) \in \mathbb{K}^d$ is **the set of all solutions** in $\mathbb{K}^d$ of the system of equations: $f_1(x_1, \ldots, x_d) = \cdots = f_s(x_1, \ldots, x_d) = 0$.

Example 4.4 *Given $\mathbb{R}[x, y]$, $V(x^2 + y^2)$ is the set of all elements that satisfy $x^2 + y^2 = 0$ over $\mathbb{R}^2$. So $V(x^2 + y^2) = \{(0, 0)\}$. Similarly, in $\mathbb{R}[x, y]$, $V(x^2 + y^2 - 1) = \{all\ points\ on\ the\ circle : x^2 + y^2 - 1 = 0\}$. Note that varieties depend on which field we are operating on. For the same polynomial $x^2 + 1$, we have:*

- *In $\mathbb{R}[x]$, $V(x^2 + 1) = \emptyset$.*
- *In $\mathbb{C}[x]$, $V(x^2 + 1) = \{(\pm i)\}$.*

The above example shows the variety can be infinite, finite (non-empty set) or empty. It is interesting to note that since we will be operating over finite fields $\mathbb{F}_q$, and any finite set of points is a variety. Likewise, any variety over $\mathbb{F}_q$ is finite (or empty). Consider the points $\{(a_1, \ldots, a_d) : a_1, \ldots, a_d \in \mathbb{F}_q\}$ in $\mathbb{F}_q^d$. Any single point is a variety of some polynomial system: e.g. $(a_1, \ldots, a_d)$ is a variety of $x_1 - a_1 = x_2 - a_2 = \cdots = x_d - a_d = 0$. **Finite unions** and **finite intersections** of varieties are also varieties.

Example 4.5 *Let $U = V(f_1, \ldots, f_s)$ and $W = V(g_1, \ldots, g_t)$ in $\mathbb{F}_q$. Then:*

- $U \cap W = V(f_1, \ldots, f_s, g_1, \ldots, g_t)$
- $U \cup W = V(f_i g_j : 1 \leq i \leq s, 1 \leq j \leq t)$

One important distinction we need to make about varieties is that a variety depends not just on the given system of polynomial equations, but rather on the **ideal** generated by the polynomials.

Definition 4.10 *A subset $I \subset \mathbb{K}[x_1, x_2, \ldots, x_d]$ is an* **ideal** *if it satisfies:*

- $0 \in I$
- *I is closed under addition: $x, y \in I \Rightarrow x + y \in I$*
- *If $x \in \mathbb{K}[x_1, x_2, \ldots, x_d]$ and $y \in I$, then $x \cdot y \in I$ and $y \cdot x \in I$.*

An ideal is generated by its *basis* or *generators*.

Definition 4.11 *Let $f_1, f_2, \ldots, f_s$ be polynomials of the ring $\mathbb{K}[x_1, x_2, \ldots, x_d]$. Let I be an ideal generated by $f_1, f_2, \ldots, f_s$. Then:*

$$I = \langle f_1, \ldots, f_s \rangle = \{h_1 f_1 + h_2 f_2 + \ldots + h_s f_s : h_1, \ldots, h_s \in \mathbb{K}[x_1, \ldots, x_d]\}$$

then, $f_1, \ldots, f_s$ are called the **basis (or generators)** *of the ideal I and correspondingly I is denoted as $I = \langle f_1, f_2, \ldots, f_s \rangle$.*

Example 4.6 *The set of even integers, which is a subset of the ring of integers Z, forms an ideal of Z. This can be seen from the following;*

- 0 *belongs to the set of even integers.*
- *The sum of two even integers x and y is always an even integer.*
- *The product of any integer x with an even integer y is always an even integer.*

Example 4.7 *Given $\mathbb{R}[x, y]$, $I = \langle x, y \rangle$ is an ideal containing all polynomials generated by x and y, such as x^2+y and $x+x \cdot y$. $J = \langle x^2, y^2 \rangle$ is an ideal containing all polynomials generated by x^2 and y^2, such as $x^2 + y^3$ and $x^{10} + x^2 \cdot y^2$. Notice that $J \subset I$ because every polynomial generated by J can be generated by I. But $I \neq J$ because $x + y$ can only be generated by I.*

The same ideal may have many different bases. For instance, it is possible to have different sets of polynomials $\{f_1, \dots, f_s\}$ and $\{g_1, \dots, g_t\}$ that may generate the same ideal, i.e., $\langle f_1, \dots, f_s \rangle = \langle g_1, \dots, g_t \rangle$. Since variety depends on the ideal, these sets of polynomials have the same solutions.

Proposition 4.1 *If $f_1, \dots, f_s$ and $g_1, \dots, g_t$ are bases of the same ideal in $\mathbb{F}[x_1, \dots, x_d]$, so that $\langle f_1, \dots, f_s \rangle = \langle g_1, \dots, g_t \rangle$, then $V(f_1, \dots, f_s) = V(g_1, \dots, g_t)$.*

Example 4.8 *Consider the two bases $F_1 = \{(2x^2 + 3y^2 - 11, x^2 - y^2 - 3\}$ and $F_2 = \{x^2 - 4, y^2 - 1\}$. These two bases generate the same ideal, i.e., $\langle F_1 \rangle = \langle F_2 \rangle$. Therefore, they represent the same variety, i.e.,*

$$V(F_1) = V(F_2) = \{\pm 2, \pm 1\}. \tag{4.22}$$

Ideals and their varieties are a key part of computer-algebra based formal verification. A given hardware design can be transformed into a set of polynomials over a field, $f_1, \dots, f_s \in F$ (we showed how this is done for Galois field arithmetic circuits in the previous chapter). This set of polynomials gives the system of equations:

$$\begin{aligned} f_1 &= 0 \\ &\vdots \\ f_s &= 0 \end{aligned}$$

Using algebra, it is possible to derive new equations from the original system. The ideal $\langle f_1, \ldots, f_s \rangle$ provides a way of analyzing such *consequences* of a system of polynomials.

Example 4.9 *Given two equations in* $\mathbb{R}[x, y, z]$*:*

$$x = z + 1$$
$$y = x^2 + 1$$

we can eliminate x *to obtain a new equation:*

$$y = (z + 1)^2 + 1 = z^2 + 2z + 2$$

Let $f_1, f_2, h \in \mathbb{R}[x, y, z]$ *be polynomials based on these equations:*

$$\begin{aligned} f_1 = x - z - 1 &= 0 \\ f_2 = y - x^2 - 1 &= 0 \\ h = y - z^2 - 2z - 2 &= 0 \end{aligned}$$

If I *is the ideal generated by* f_1 *and* f_2*, i.e.* $I = \langle f_1, f_2 \rangle$*, then we find* $h \in I$ *as follows:*

$$\begin{aligned} g_1 &= x + z + 1 \\ g_2 &= 1 \\ h = g_1 \cdot f_1 + g_2 \cdot f_2 &= y - z^2 - 2z - 2 \end{aligned}$$

where $g_1, g_2 \in \mathbb{R}[x, y, z]$*. Thus, we call* h *a* **member of the ideal** I.

Let $\mathbb{K}$ be any field and let $\mathbf{a} = (a_1, \ldots, a_d) \in \mathbb{K}^d$ be a point, and $f \in \mathbb{K}[x_1, \ldots, x_d]$ be a polynomial. We say that f *vanishes* on $\mathbf{a}$ if $f(\mathbf{a}) = 0$, i.e., $\mathbf{a}$ is in the variety of f.

Definition 4.12 *For any variety* V *of* $\mathbb{K}^d$*, the ideal of polynomials that vanish on* V*, called the vanishing ideal of* V*, is defined as* $I(V) = \{f \in \mathbb{F}[x_1, \ldots, x_d] : \forall \mathbf{a} \in V, f(\mathbf{a}) = 0\}$.

Proposition 4.2 *If a polynomial* f *vanishes on a variety* V*, then* $f \in I(V)$.

Example 4.10 *Let ideal $J = \langle x^2, y^2 \rangle$. Then $V(J) = \{(0,0)\}$. All polynomials in J will obviously agree with the solution and vanish on this variety. However, the polynomials x, y are not in J but they also vanish on this variety. Therefore, $I(V(J))$ is the set of all polynomials that vanish on $V(J)$, and the polynomials x, y are members of $I(V(J))$.*

Definition 4.13 *Let $J \subset \mathbb{K}[x_1, \dots, x_d]$ be an ideal. The radical of J is defined as $\sqrt{J} = \{f \in \mathbb{K}[x_1, \dots, x_d] : \exists m \in \mathbb{N}, f^m \in J\}$.*

Example 4.11 *Let $J = \langle x^2, y^2 \rangle \subset \mathbb{K}[x, y]$. Note neither x nor y belongs to J, but they belong to $\sqrt{J}$. Similarly, $x \cdot y \notin J$, but since $(x \cdot y)^2 = x^2 \cdot y^2 \in J$, therefore, $x \cdot y \in \sqrt{J}$.*

When $J = \sqrt{J}$, then J is said to be a *radical ideal*. Moreover, $I(V)$ is a radical ideal. By analyzing the ideal J, generated by a system of polynomials derived from a hardware design, its variety $V(J)$, and the ideal of polynomials that vanish over this variety, $I(V(J))$, we can reason about the existence of certain properties of the design. To check for the existence of a property, we formulate the property as a polynomial and then perform an **ideal membership test** to determine if this polynomial is contained within the ideal $I(V(J))$. A **Gröbner basis** provides a decision procedure for performing this test, which is described in the following section. A future section focuses on **Hilbert's Nullstellensatz**, which describes the properties of the ideal of a variety, $I(V(J))$.

4.3 Gröbner Bases

As mentioned earlier, different polynomial sets may generate the same ideal. Some of these generating sets may be a better representation of the ideal, and thus provide more information and insight into the properties of ideal. One such ideal representation is a **Gröbner basis**, which has a number of important properties that can solve numerous polynomial decision questions:

- Presence or absence of solutions (varieties)
- Dimension of the varieties
- Ideal membership of a polynomial

In essence, a Gröbner basis is a canonical representation of an ideal. There are many equivalent definitions of Gröbner bases, so we start with the definition that best describes their properties:

Definition 4.14 *A set of non-zero polynomials $G = \{g_1, \ldots, g_t\}$ which generate the ideal $I = \langle g_1, \ldots, g_t \rangle$, is called a* **Gröbner basis** *for I if and only if for all $f \in I$ where $f \neq 0$, there exists a $g_i \in G$ such that $lm(g_i)$ divides $lm(f)$.*

$$G = \textit{GröbnerBasis}(I) \iff \forall f \in I : f \neq 0, \exists g_i \in G : lm(g_i) \mid lm(f) \tag{4.23}$$

The foundation for computing the Gröbner basis of an ideal was laid out by Buchberger [103]. Given a set of polynomials $F = \{f_1, \ldots, f_s\}$ that generate ideal $I = \langle f_1, \ldots, f_s \rangle$, Buchberger gives an algorithm to compute a Gröbner basis $G = \langle g_1, \ldots, g_t \rangle$. This algorithm relies on the notions of S-polynomials and polynomial reduction.

Definition 4.15 *For $f, g \in \mathbb{K}[x_1, \ldots, x_d]$, an* **S-polynomial** *$Spoly(f, g)$ is defined as:*

$$Spoly(f,g) = \frac{L}{lt(f)} \cdot f - \frac{L}{lt(g)} \cdot g \tag{4.24}$$

$$\textit{where } L = lcm\,(lt(f), lt(g))$$

Note lcm denotes least common multiple.

Definition 4.16 *The* **reduction** *of a polynomial f, by another polynomial g, to a reduced polynomial r is denoted:*

$$f \xrightarrow{g} r$$

Reduction is carried out using multivariate, polynomial long division.

For sets of polynomials, the notation

$$f \xrightarrow{F}_{+} r$$

represents the reduced polynomial r resulting from f as reduced by a set of non-zero polynomials $F = \{f_1, \ldots, f_s\}$. The polynomial r is considered **reduced** *if $r = 0$ or no term in r is divisible by a $lm(f_i), \forall f_i \in F$.*

The reduction process $f \xrightarrow{F}_{+} r$, of dividing a polynomial f by a set of polynomials of F, can be modeled as repeated long-division of f by each of the polynomials in F until no further reductions can be made. The result of this process is then r. This reduction process is shown in Algorithm 2.

Algorithm 2: Polynomial Reduction

```
Input: $f, f_1, \dots, f_s$
Output: $r, a_1, \dots, a_s$, such that $f = a_1 \cdot f_1 + \cdots + a_s \cdot f_s + r$.
$a_1 = a_2 = \cdots = a_s = 0$; $r = 0$;
$p := f$;
while $p \neq 0$ do
    i=1;
    divisionmark = false;
    while $i \leq s$ && divisionmark = false do
        if $f_i$ can divide $p$ then
            $a_i = a_i + lt(p)/lt(f_i)$;
            $p = p - lt(p)/lt(f_i) \cdot f_i$;
            divisionmark = true;
        else
            i=i+1;
        end
    end
    if divisionmark = false then
        $r = r + lt(p)$;
        $p = p - lt(p)$;
    end
end
```

The reduction algorithm keeps canceling the leading terms of polynomials until no more leading terms can be further canceled. So the key step is $p = p - lt(p)/lt(f_i) \cdot f_i$, as the following example shows.

Example 4.12 *Given $f = y^2 - x$ and $f_1 = y - x$ in $\mathbb{Q}[x, y]$ with $deglex$: $y > x$, perform $f \xrightarrow{f_1}_{+} r$:*

1. $f = y^2 - x$, $f/f_1 = f - lt(f)/lt(f_1) \cdot f_1 = y^2 - x - (y^2/y) \cdot (y - x) = y \cdot x - x$

2. $f = y \cdot x - x$, $f/f_1 = f - lt(f)/lt(f_1) \cdot f_1 = (y \cdot x - x)/f_1 = x^2 - x$

3. $f = x^2 - x$, *no more operations possible, so* $r = x^2 - x$

With the notions of S-polynomials and polynomial reduction in place, we can now present Buchberger's Algorithm for computing Gröbner bases [103]. Note that a fixed monomial (term) ordering is required for a Gröbner basis computation to ensure that polynomials are manipulated in a consistent manner.

Algorithm 3: Buchberger's Algorithm

```
Input: $F = \{f_1, \ldots, f_s\}$, such that $I = \langle f_1, \ldots, f_s \rangle$
, and term order $>$ Output: $G = \{g_1, \ldots, g_t\}$, a Gröbner basis of $I$
$G := F$;
repeat
    $G' := G$;
    for each pair $\{f_i, f_j\}, i \neq j$ in $G'$ do
        $Spoly(f_i, f_j) \xrightarrow{G'}_+ r$ ;
        if $r \neq 0$ then
            $G := G \cup \{r\}$ ;
        end
    end
until $G = G'$;
```

Buchberger's algorithm takes pairs of polynomials (f_i, f_j) in the basis G and combines them into "S-polynomials" ($Spoly(f_i, f_j)$) to cancel leading terms. The S-polynomial is then reduced (divided) by all elements of G to a remainder r, denoted as $Spoly(f_i, f_j) \xrightarrow{G}_+ r$. This process is repeated for all unique pairs of polynomials, including those created by newly added elements, until no new polynomials are generated; ultimately constructing the Gröbner basis.

Example 4.13 *Consider the ideal $I \subset \mathbb{Q}[x, y]$, $I = \langle f_1, f_2 \rangle$, where $f_1 = yx - y,\ f_2 = y^2 - x$. Assume a degree-lexicographic term ordering with $y > x$ is imposed.*

First, we need to compute $Spoly(f_1, f_2) = x \cdot f_2 - y \cdot f_1 = y^2 - x^2$. Then we conduct a polynomial reduction $y^2 - x^2 \xrightarrow{f_2} x^2 - x \xrightarrow{f_1} x^2 - x$. Let $f_3 = x^2 - x$. Then G is updated as $\{f_1, f_2, f_3\}$. Next we compute $Spoly(f_1, f_3) = 0$. So there is no new polynomial generated. Similarly, we compute $Spoly(f_2, f_3) = x \cdot y^2 - x^3$, followed by $x \cdot y^2 - x^3 \xrightarrow{f_1} y^2 - x^3 \xrightarrow{f_2} x - x^3 \xrightarrow{f_2} 0$. Again, no polynomial is generated. Finally, $G = \{f_1, f_2, f_3\}$.

When computing a Gröbner basis, it's important to note that if $lt(f_i)$ and $lt(f_j)$ have no common variables, the S-poly reduction step in Buchberger's algorithm, $Spoly(f_i, f_j) \xrightarrow{G'}_+$

r, will produce $r = 0$.

Proof. If $lt(f)$ and $lt(g)$ have no common variables, $L = lcm(lt(f), lt(g)) = lt(f) \cdot lt(g)$. Then:

$$Spoly(f, g) = \frac{L}{lt(f)} \cdot f - \frac{L}{lt(g)} \cdot g = \frac{lt(f) \cdot lt(g)}{lt(f)} \cdot f - \frac{lt(f) \cdot lt(g)}{lt(g)} \cdot g = lt(g) \cdot f - lt(f) \cdot g$$

Thus, every monomial in $Spoly(f, g)$ is divisible by either $lt(f)$ or $lt(g)$, so computing $Spoly(f, g) \xrightarrow{f,g}_+ r$ will give $r = 0$. ■

As mentioned previously, a Gröbner basis gives a decision procedure to test for polynomial membership in an ideal. This is explained in the following Theorem.

Theorem 4.1 Ideal Membership Test *Let $G = \{g_1, \cdots, g_t\}$ be a Gröbner basis for an ideal $I \subset \mathbb{K}[x_1, \cdots, x_d]$ and let $f \in \mathbb{K}[x_1, \ldots, x_d]$. Then $f \in I$ if and only if the remainder on division of f by G is zero.*

In other words,

$$f \in I \iff f \xrightarrow{G}_+ 0 \tag{4.25}$$

Example 4.14 *Consider Example 4.13. Let $f = y^2x - x$ be another polynomial. Note that $f = yf_1 + f_2$, so $f \in I$. If we divide f by f_1 first and then by f_2, we will obtain a zero remainder. However, since the set $\{f_1, f_2\}$ is not a Gröbner basis, we find that the reduction $f \xrightarrow{f_2} x^2 - x \xrightarrow{f_1} x^2 - x \neq 0$; i.e. dividing f by f_2 first and then by f_1 does not lead to a zero remainder. However, if we compute the Gröbner basis G of I, $G = \{x^2 - x, yx - y, y^2 - x\}$, dividing f by polynomials in G in any order will always lead to the zero remainder. Therefore, one can decide ideal membership unequivocally using the Gröbner basis.*

A Gröbner basis is not a canonical representation of an ideal, but a **reduced Gröbner basis** is. To compute a reduced Gröbner basis, we first must compute a minimal Gröbner basis.

Definition 4.17 *A* **minimal Gröbner basis** *for a polynomial ideal I is a Gröbner basis G for I such that*

- $lc(g_i) = 1, \forall g_i \in G$
- $\forall g_i \in G$, $lt(g_i) \notin \langle lt(G - \{g_i\}) \rangle$

A **minimal** Gröbner basis is a Gröbner basis such that all polynomials have a coefficient of 1 and no leading term of any element in G divides another in G. Given a Gröbner basis G, a minimal Gröbner basis can be computed as follows:

1. Minimize every $g_i \in G$, i.e $g_i = g_i / lc(g_i)$
2. For $g_i, g_j \in G$ where $i \neq j$, remove g_i from G if $lt(g_i) \mid lt(g_j)$, i.e. remove every polynomial in G whose leading term is divisible by the leading term of some other polynomial in G.

A minimal Gröbner basis can then be further reduced.

Definition 4.18 *A* **reduced Gröbner basis** *for a polynomial ideal I is a Gröbner basis $G = \{g_1, \ldots, g_t\}$ such that:*

- $lc(g_i) = 1, \forall g_i \in G$
- *$\forall g_i \in G$, no monomial of g_i lies in $\langle lt(G - \{g_i\}) \rangle$*

G is a reduced Gröbner basis when no monomial of any element in G divides the leading term of another element. This reduction is achieved as follows:

Definition 4.19 *Let $H = \{h_1, \ldots, h_t\}$ be a minimal Gröbner basis. Apply the following reduction process:*

- $h_1 \xrightarrow{G_1}_+ g_1$*, where g_1 is reduced w.r.t. $G_1 = \{h_2, \ldots, h_t\}$*
- $h_2 \xrightarrow{G_2}_+ g_2$*, where g_2 is reduced w.r.t. $G_2 = \{g_1, h_3, \ldots, h_t\}$*
- $h_3 \xrightarrow{G_3}_+ g_3$*, where g_3 is reduced w.r.t. $G_3 = \{g_1, g_2, h_4, \ldots, h_t\}$*

  $\vdots$

- $h_t \xrightarrow{G_t}_+ g_t$*, where g_t is reduced w.r.t. $G_t = \{g_1, g_2, g_3, \ldots, g_{t-1}\}$*

Then $G = \{g_1, \ldots, g_t\}$ is a **reduced Gröbner basis.**

Subject to the given term order $>$, such a reduced Gröbner basis $G = \{g_1, \ldots, g_t\}$ is a **unique canonical representation of the ideal**, as given by Proposition 4.3 below.

Proposition 4.3 *[10] Let $I \neq \{0\}$ be a polynomial ideal. Then, for a given monomial ordering, I has a unique reduced Gröbner basis.*

Gröbner basis computation depends on the $Spoly$ computation, which in turn depends on the leading terms of polynomials. Thus, different monomial orderings can result in different Gröbner basis computations for the same ideal. Computation using a degrevlex ordering tends to be least difficult, while lex ordering tends to be computationally complex. However, lex ordering used in the computation of Gröbner basis is an **elimination ordering**; that is, the polynomials contained in the resulting Gröbner basis have continuously eliminated variables in the ordering. This is the topic of elimination theory, which is described in the following section.

4.4 Elimination Theory

Elimination theory uses **elimination ordering** to systematically eliminate variables from a system of polynomial equations.

Definition 4.20 *Let I be an ideal in $\mathbb{K}[x_1, \ldots, x_k]$. The i-th* **elimination ideal** *I_i is the ideal of $\mathbb{K}[x_{i+1}, \ldots, x_k]$ defined by*

$$I_k = I \cap \mathbb{K}[x_{i+1}, \ldots, x_k] \tag{4.26}$$

The elimination ideal I_i has eliminated all the variables $x_1, \ldots, x_i$, i.e. it only contains polynomials with variables in $x_{i+1}, \ldots, x_k$. We can generate elimination ideals by computing Gröbner bases using elimination orderings.

Theorem 4.2 [Elimination Theorem] *Let I be an ideal in $\mathbb{K}[x_1, \ldots, x_k]$ and let G be the Gröbner basis of I with respect to the lex order (elimination order) $x_1 > x_2 > \cdots > x_k$. Then, for every $0 \leq i \leq k$,*

$$G_k = G \cap \mathbb{K}[x_{i+1}, \ldots, x_k] \tag{4.27}$$

is a Gröbner basis of the i-th elimination ideal I_i.

This can be better visualized using the following example.

Example 4.15 *Given the following equations in $\mathbb{R}[x, y, z]$*

$$\begin{aligned} x^2 + y + z &= 1 \\ x + y^2 + z &= 1 \\ x + y + z^2 &= 1 \end{aligned}$$

let I be the ideal generated by these equations:

$$I = \langle x^2 + y + z - 1, x + y^2 + z - 1, x + y + z^2 - 1 \rangle$$

The Gröbner basis for I with respect to lex order $x > y > z$ is found to be $G = \{g_1, g_2, g_3, g_4\}$ where

$$\begin{aligned} g_1 &= x + y + z^2 - 1 \\ g_2 &= y^2 - y - z^2 + z \\ g_3 &= 2yz^2 + z^4 - z^2 \\ g_4 &= z^6 - 4z^4 + 4z^3 - z^2 \end{aligned}$$

Notice that while g_1 has variables in $\mathbb{R}[x, y, z]$, g_2 and g_3 only have variables in $\mathbb{R}[y, z]$ and g_4 only has variables in $\mathbb{R}[z]$. Thus, $G_1 = G \cap \mathbb{R}[y, z] = \{g_2, g_3, g_4\}$ and $G_2 = G \cap \mathbb{R}[z] = \{g_4\}$

Also notice that since g_4 only contains variable z, and since $g_4 = 0$, a solution for z can be obtained. This solution can then be applied to g_2 and g_3 to obtain solutions for y, and so on.

Elimination theory provides the basis for our abstraction approach.

4.5 Hilbert's Nullstellensatz

In this section, we further describe some correspondence between ideals and varieties in the context of algebraic geometry. The celebrated results of Hilbert's Nullstellensatz establish these correspondences.

Definition 4.21 *A field $\overline{\mathbb{K}}$ is an* **algebraically closed** *field if every polynomial in one variable with degree at least 1, with coefficients in $\overline{\mathbb{K}}$, has a root in $\overline{\mathbb{K}}$.*

In other words, any non-constant polynomial equation over $\overline{\mathbb{K}}\,[x]$ always has at least one root in $\overline{\mathbb{K}}$. Every field $\mathbb{K}$ is contained in an algebraically closed one $\overline{\mathbb{K}}$. For example, the field of real numbers $\mathbb{R}$ is not an algebraically closed field, because $x^2 + 1 = 0$ has no root in $\mathbb{R}$. However, $x^2 + 1 = 0$ has roots in the field of complex numbers $\mathbb{C}$, which is an algebraically closed field. In fact, $\mathbb{C}$ is the algebra closure of $\mathbb{R}$. Every algebraically closed field is an infinite field.

An interesting result is one of **Strong Nullstellensatz**. The strong Nullstellensatz establishes the correspondence between radical ideals and varieties.

Theorem 4.3 *(The Strong Nullstellensatz [10]) Let $\overline{\mathbb{K}}$ be an algebraically closed field, and let J be an ideal in $\overline{\mathbb{K}}[x_1, \ldots, x_d]$. Then we have $I(V_{\overline{\mathbb{K}}}(J)) = \sqrt{J}$.*

Strong Nullstellensatz holds a special form over Galois fields $\mathbb{F}_q$. Recall the notion of vanishing polynomials over Galois fields from the previous chapter: for every element $A \in \mathbb{F}_q$, $A - A^q = 0$; then the polynomial $x^q - x$ in $\mathbb{F}_q[x]$ vanishes over $\mathbb{F}_q$. Thus, if $J_0 = \langle x^q - x \rangle$ is the ideal generated by the vanishing polynomial, $V(J_0) = \mathbb{F}_q$. Similarly, over $\mathbb{F}_q[x_1, \ldots, x_d]$, J_0 is $\langle x_1^q - x_1, \ldots, x_d^q - x_d \rangle$ and $V(J_0) = (F_q)^d$.

Definition 4.22 *Given two ideals, $I_1 = \langle f_1, \ldots, f_s \rangle$ and $I_2 = \langle g_1, \ldots g_t \rangle$, then the* **sum of ideals** $I_1 + I_2 = \langle f_1, \ldots, f_s, g_1, \ldots g_t \rangle$

Theorem 4.4 *(Strong Nullstellensatz over $\mathbb{F}_q$) For any Galois field $\mathbb{F}_q$, let $J \subset \mathbb{F}_q[x_1, \ldots, x_d]$ be any ideal and let $J_0 = \langle x_1^q - x_1, x_d^q - x_d \rangle$ be the ideal of all vanishing polynomials. Let $V_{\mathbb{F}_q}(J)$ denote the variety of J over $\mathbb{F}_q$. Then, $I(V_{\mathbb{F}_q}(J)) = J + J_0$.*

The proof is given in [67]. Here, we provide a proof outline.

Proof.

1. $\sqrt{J + J_0} = J + J_0$. That is, $J + J_0$ is a radical ideal.

2. $V_{\mathbb{F}_q}(J) = V_{\overline{\mathbb{F}_q}}(J + J_0)$.

3. Due to (2), $I(V_{\mathbb{F}_q}(J)) = I(V_{\overline{\mathbb{F}_q}}(J + J_0))$. By Strong Nullstellensatz, this is equivalent to $\sqrt{J + J_0}$. Finally, due to (1), this is equivalent to $J + J_0$.

■

4.6 Concluding Remarks

Our approach to word-level abstraction of Galois field arithmetic circuits applies concepts of polynomial ideals, varieties, Gröbner basis, and elimination theory to abstract a word-level representation of the circuit. This approach is described in the next chapter. However, a Gröbner basis computation is prohibitively expensive; thus we propose improvements to our original approach in a subsequent chapter.

CHAPTER 5

WORD-LEVEL ABSTRACTION OF COMBINATIONAL CIRCUITS

In this chapter, we introduce our approach to abstract word-level canonical representations of combinational circuits using methods based on computer-algebra and algebraic-geometry. Given a bit-level implementation of a combinational, acyclic circuit C that implements some unknown function $f : \mathbb{F}_{2^k}^n \rightarrow \mathbb{F}_{2^k}$, where Z is the k-bit output and $A_1, \dots, A_n$ are the k-bit inputs, find the canonical word-level representation $Z = \mathcal{F}(A_1, \dots, A_n)$ implemented by C; that is, find a canonical representation of the polynomial $\mathcal{F}$ in terms of $A_1, \dots, A_n$. For example, a combinational circuit with one word-level k-bit input A and k-bit output Z, which computes $Z = \mathcal{F}(A)$ over $\mathbb{F}_{2^k}$, is shown in Fig. 5.1.

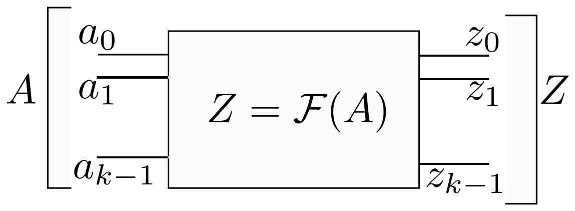


Figure 5.1: Derive the abstraction $Z = \mathcal{F}(A)$

By modelling the the arithmetic circuit as a polynomial system in $\mathbb{F}_{2^k}[x_1, x_2, \cdots, x_d]$, the abstraction problem can be solved using a Gröbner basis computation using an abstraction term order. However, as computing a Gröbner basis is computationally expensive, this approach is directly applicable only to Galois field arithmetic circuits no larger than 40-bits.

5.1 Problem Statement

- Given a gate-level combinational, acyclic circuit C with n word-level k-bit inputs, $A_1, \dots, A_n \in \mathbb{F}_{2^k}$ and one k-bit output Z.
- Pick a primitive, irreducible polynomial $P(x)$ over $\mathbb{F}_2[x]$ of degree k to construct $\mathbb{F}_{2^k}$ (these polynomials are known). Let $P(\alpha) = 0$, where $\alpha \in \mathbb{F}_{2^k}$ is a root of the irreducible polynomial P.
- The bit-level primary inputs of the circuit are denoted $\{a_0^i, a_1^i, \dots, a_{k-1}^i\}$, for $1 \leq i \leq n$; the bit-level primary outputs are $\{z_0, \dots, z_{k-1}\} = Z$. Note that all $a_j^i, z_j \in \mathbb{F}_2$ for $0 \leq j < k$.
- Find the word-level polynomial function $\mathcal{F}$ over $\mathbb{F}_{2^k}$ computed by C in the form of $Z = \mathcal{F}(A_1, \dots, A_n)$.

In order for a combinational circuit C to compute a function f over the Galois field $\mathbb{F}_{2^k}$, C must have any number of k-bit word-level inputs and one k-bit word-level output. Since every function over $\mathbb{F}_{2^k}$ is a polynomial function, C has a word-level polynomial representation over $\mathbb{F}_{2^k}$. The goal is to derive this word-level polynomial representation $\mathcal{F}$ computed by a given combinational circuit over a given $\mathbb{F}_{2^k}$.

For the purpose of explaining the proposed abstraction approach, this chapter explores its application to Galois field multiplier circuits, which are described in Chapter 3, as they form the core of most cryptographic computations in ECC and are notoriously hard to verify. In this case, $P(x)$ is chosen to be the same primitive polynomial over which the circuit was designed.

Example 5.1 *Consider our problem statement as it applies to a multiplier circuit over $\mathbb{F}_{2^k}$. The specification of the circuit is unknown.*

- *Given the Galois field $\mathbb{F}_{2^k}$ and the corresponding irreducible polynomial $P(x)$. Let $P(\alpha) = 0$.*
- *Given a gate-level combinational circuit. The bit-level primary inputs of the circuit are $\{a_0, \dots, a_{k-1},\ b_0, \dots, b_{k-1}\}$, and the bit-level primary outputs are $\{z_0, \dots, z_{k-1}\}$; thus all $a_i, b_i, z_i \in \mathbb{F}_2$.*

- *A and B denote the k-bit word-level inputs and Z is the k-bit word-level output. Therefore, $A = a_0 + a_1\alpha + \cdots + a_{k-1}\alpha^{k-1}$, $B = b_0 + b_1\alpha + \cdots + b_{k-1}\alpha^{k-1}$, and $Z = z_0 + z_1\alpha + \cdots + z_{k-1}\alpha^{k-1}$ with $A, B, Z \in \mathbb{F}_{2^k}$.*

- *Find the word-level polynomial function $\mathcal{F}$ that this circuit implements over $\mathbb{F}_{2^k}$. The polynomial must be in the form of $Z = \mathcal{F}(A, B)$. Since, in this case, the circuit is a multiplier, this resulting polynomial will be $Z = A \cdot B$.*

5.2 Circuit Polynomial Modeling

Given a gate-level implementation of a circuit, we map each gate-level Boolean operator in the circuit (NOT, AND, OR, XOR) to a polynomial over $\mathbb{F}_2$ using the following one-to-one mapping over $\mathbb{B} \rightarrow \mathbb{F}_2$:

$$\begin{aligned}
NOT &: \neg a \rightarrow a + 1 \pmod 2 \\
AND &: a \wedge b \rightarrow a \cdot b \pmod 2 \\
OR &: a \vee b \rightarrow a + b + a \cdot b \pmod 2 \\
XOR &: a \oplus b \rightarrow a + b \pmod 2
\end{aligned} \tag{5.1}$$

where $a, b \in \mathbb{F}_2 = \{0, 1\}$. Note that the equation $c = \mathcal{F}(a, b)$ is written in polynomial form as $c - \mathcal{F}(a, b) = c + \mathcal{F}(a, b) = 0,$ as $-1 \equiv +1 \pmod 2$.

Example 5.2 *Consider the equation with Boolean operators:*

$$z = a \oplus (b \vee c).$$

This equation modeled over $\mathbb{F}_2$ is:

$$z + a + b + c + b \cdot c = 0$$

Notice that the left-hand side expression is a polynomial in $\mathbb{F}_2\,[a, b, c, z] \subset \mathbb{F}_{2^k}\,[a, b, c, z]$

Secondly, we model the k-bit word-level inputs and the k-bit word-level output of the given circuit as polynomial expressions in $\mathbb{F}_{2^k}$ as shown in the problem statement.

If the k-bit word-level output of the circuit is denoted Z which is composed of bit-level outputs $z_0, \ldots, z_{k-1}$, the corresponding equation is:

$$Z = z_0 + z_1\alpha + \cdots + z_{k-1}\alpha^{k-1}$$

Once again, since $-Z = +Z \pmod 2$, this equation is modelled as:

$$Z + z_0 + z_1\alpha + \cdots + z_{k-1}\alpha^{k-1} = 0$$

Likewise, for all word-level inputs $A_1, \ldots, A_n$ we have

$$\begin{aligned} A_1 + a_0^1\alpha + \cdots + a_{k-1}^1 &= 0 \\ &\vdots \\ A_n + a_0^n\alpha + \cdots + a_{k-1}^n &= 0 \end{aligned}$$

Overall, a combinational circuit composed of s Boolean gates with n k-bit inputs, $A_1, \ldots, A_n$, and one k-bit output Z, is modeled as a polynomial system over $\mathbb{F}_{2^k}$ as follows:

$$\begin{aligned} &\left.\begin{aligned} f_1(x_1, x_2, \cdots, x_d) &= 0 \\ f_2(x_1, x_2, \cdots, x_d) &= 0 \\ &\vdots \\ f_s(x_1, x_2, \cdots, x_d) &= 0 \end{aligned}\right\} \quad \textit{Bit-level circuit constraints} \\ &\left.\begin{aligned} f_Z : Z + z_0 + z_1 \cdot \alpha, \cdots, z_{k-1} \cdot \alpha^{k-1} &= 0 \\ f_{A_1} : A_1 + a_0^1 + a_1^1 \cdot \alpha + \cdots + a_{k-1}^1 \cdot \alpha^{k-1} &= 0 \\ &\vdots \\ f_{A_n} : A_n + a_0^n + a_1^n \cdot \alpha + \cdots + a_{k-1}^n \cdot \alpha^{k-1} &= 0 \end{aligned}\right\} \quad \textit{Word-level designation} \end{aligned} \tag{5.2}$$

Example 5.3 *Consider a 2-bit multiplier over $\mathbb{F}_{2^2}$ with $P(x) = x^2 + x + 1$, given in Fig. 5.2. Variables a_0, a_1, b_0, b_1 are primary inputs, z_0, z_1 are primary outputs, and c_0, c_1, c_2, c_3, r_0 are intermediate variables.*

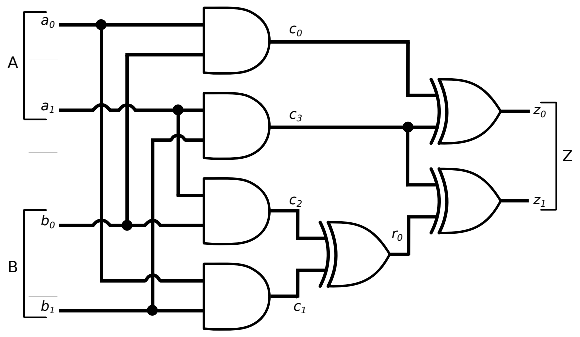


Figure 5.2: A 2-bit multiplier over $\mathbb{F}_(2^2)$.

The circuit can be described using the following Boolean equations:

$$c_0 = a_0 \wedge b_0,$$
$$c_1 = a_0 \wedge b_1,$$
$$c_2 = a_1 \wedge b_0,$$
$$c_3 = a_1 \wedge b_1,$$
$$r_0 = c_1 \oplus c_2,$$
$$z_0 = c_0 \oplus c_3,$$
$$z_1 = r_0 \oplus c_3,$$

With the mapping rules given in Equation 5.1, the above equations are transformed into the following polynomials:

$$c_0 + a_0 \cdot b_0,$$
$$c_1 + a_0 \cdot b_1,$$
$$c_2 + a_1 \cdot b_0,$$
$$c_3 + a_1 \cdot b_1,$$
$$r_0 + c_1 + c_2,$$
$$z_0 + c_0 + c_3,$$
$$z_1 + r_0 + c_3,$$

Therefore, our overall polynomial system is:

$$\left.\begin{aligned} f_1 &: c_0 + a_0 \cdot b_0 \\ f_2 &: c_1 + a_0 \cdot b_1 \\ f_3 &: c_2 + a_1 \cdot b_0 \\ f_4 &: c_3 + a_1 \cdot b_1 \\ f_5 &: r_0 + c_1 + c_2 \\ f_6 &: z_0 + c_0 + c_3 \\ f_7 &: z_1 + r_0 + c_3 \end{aligned}\right\} \quad \textit{Bit-level circuit constraints}$$
$$\left.\begin{aligned} f_A &: A + a_0 + a_1 \cdot \alpha \\ f_B &: B + b_0 + b_1 \cdot \alpha \\ f_Z &: Z + z_0 + z_1 \cdot \alpha \end{aligned}\right\} \quad \textit{Word-Level designation} \tag{5.3}$$

5.3 Abstraction Formulation

Let S be the system of polynomials, $\{f_1, \ldots, f_s, f_{A_1}, \ldots, f_{A_n}, f_Z\} \subset \mathbb{F}_{2^k}$, derived from the hardware implementation of the Galois field arithmetic circuit over $\mathbb{F}_{2^k}$. This circuit performs some unknown function f over $\mathbb{F}_{2^k}$ in the form of $Z = \mathcal{F}(A_1, \ldots, A_n)$, where Z is the k-bit output and $A_1, \ldots, A_n$ are the k-bit inputs. The polynomial representation of $\mathcal{F}$ over $\mathbb{F}_{2^k}$ is thus:

$$f_{\mathcal{F}} : Z + \mathcal{F}(A_1, \ldots, A_n)$$

Since $f_{\mathcal{F}}$ is ultimately derived from the circuit implementation, it agrees with the solution to the system of polynomials $\{S\} = 0$, i.e.:

$$f_1 = \cdots = f_s = f_{A_1} = \cdots = f_{A_n} = f_Z = 0$$

Thus, if we let $J = \langle f_1, \ldots, f_s, f_{A_1}, \ldots, f_{A_n}, f_Z \rangle$ be the ideal generated by S, $f_{\mathcal{F}}$ **vanishes** on the variety $V_{\mathbb{F}_{2^k}}(J)$. Therefore, due to Proposition 4.2, $f_{\mathcal{F}}$ must be contained in the ideal of polynomials that vanish on this variety, $f_{\mathcal{F}} \in I(V_{\mathbb{F}_{2^k}}(J))$.

By applying Strong Nullstellensatz over $\mathbb{F}_{2^k}$ (Theorem 4.3), $I(V_{\mathbb{F}_{2^k}}(J)) = J + J_0$ where J_0 is the ideal generated by all vanishing polynomials in $\mathbb{F}_{2^k}$. Recall that a vanishing polynomial in $\mathbb{F}_{2^k}[x]$ is $x^q - x = x^q + x$. In our case, $\{x_1, \ldots, x_d\} \in \mathbb{F}_2$ and $\{A_1, \ldots, A_n, Z\} \in \mathbb{F}_{2^k}$. Thus, for $\mathbb{F}_{2^k}[x_1, \ldots, x_d, A_1, \ldots, A_n, Z]$:

$$J_0 = \langle x_1^2 + x_1, \ldots, x_d^2 + x_d, A_1^{2^k} + A_1, \ldots, A_n^{2^k} + A_n, Z^{2^k} + Z \rangle$$

The generators of the ideal sum $J + J_0$ are simply the combination of the generators of J and the generators J_0.

Example 5.4 *Let us re-consider Example 5.3. First, polynomials are extracted from the circuit implementation as shown in Example 5.3. These polynomials represent the ideal J. Along with the ideal of vanishing polynomials J_0, the following polynomials represent the generators of $J + J_0$ for the multiplier circuit.*

$$\left.\begin{aligned} f_1 &: c_0 + a_0 \cdot b_0 \\ f_2 &: c_1 + a_0 \cdot b_1 \\ f_3 &: c_2 + a_1 \cdot b_0 \\ f_4 &: c_3 + a_1 \cdot b_1 \\ f_5 &: r_0 + c_1 + c_2 \\ f_6 &: z_0 + c_0 + c_3 \\ f_7 &: z_1 + r_0 + c_3 \end{aligned}\right\} \quad \textit{Bit-level circuit constraints } (\subset J)$$

$$\left.\begin{aligned} f_A &: A + a_0 + a_1 \cdot \alpha \\ f_B &: B + b_0 + b_1 \cdot \alpha \\ f_Z &: Z + z_0 + z_1 \cdot \alpha \end{aligned}\right\} \quad \textit{Word-level designation } (\subset J)$$

$$\left.\begin{aligned} a_0^2 - a_0,\ a_1^2 - a_1,\ b_0^2 - b_0,\ b_1^2 - b_1 \\ c_0^2 - c_0,\ c_1^2 - c_1,\ c_2^2 - c_2,\ c_3^2 - c_3 \\ r_0^2 - r_0,\ z_0^2 - z_0,\ z_1^2 - z_1 \\ A^4 - A,\ B^4 - B,\ Z^4 - Z \end{aligned}\right\} \quad \textit{vanishing polynomials}(J_0)$$

The variety $V_{\mathbb{F}_q}(J)$ is the set of all consistent assignments to the nets (signals) in the circuit C. If we *project this variety on the word-level input and output variables of the circuit C, we essentially generate the function $\mathcal{F}$ implemented by the circuit.* Projection of varieties from d-dimensional space $\mathbb{F}_q^d$ onto a lower dimensional subspace $\mathbb{F}_q^{d-l}$ is equivalent to *eliminating l variables* from the corresponding ideal. This can be done by computing a Gröbner basis of the ideal with elimination ordering, as described in the Elimination Theorem (Theorem 4.2). Thus, we can find the polynomial $f_{\mathcal{F}} : Z + \mathcal{F}(A_1, \dots, A_n)$ by computing the Gröbner basis of $J + J_0$ using the proper elimination ordering.

The proposed elimination order for abstraction is defined as the **abstraction term order**.

Definition 5.1 *Given a circuit C, let $x_1, \dots, x_d$ denote all the bit-level variables, let $A_1, \dots, A_n$ denote the k-bit word-level inputs, and let Z denote the k-bit word-level output. Using the partial variable order $\{x_1, \dots, x_d\} > Z > \{A_1, \dots, A_n\}$, where any refinement of the order will do, impose a lex term order $>$ on the polynomial ring $R = \mathbb{F}_q[x_1, \dots, x_d, Z, A_1, \dots, A_n]$. This elimination term order $>$ is defined as the* **Abstraction Term Order**. *The relative ordering among $x_1, \dots, x_d$ is not important and can be chosen arbitrarily. Likewise, the relative ordering among $A_1, \dots, A_n$ is also unimportant.*

Theorem 5.1 Abstraction Theorem: *Using the setup and notations above, compute a Gröbner basis G of ideal $(J + J_0)$ using the abstraction term order $>$. Then:*
(i) For every word-level input A_i, G must contain the vanishing polynomial $A_i^q - A_i$ as the only polynomial with A_i as its only variable;
(ii) G must contain a polynomial of the form $Z + \mathcal{G}(A_1, \dots, A_n)$; and
(iii) $Z + \mathcal{G}(A_1, \dots, A_n)$ is such that $\mathcal{F}(A_1, \dots, A_n) = \mathcal{G}(A_1, \dots, A_n), \forall A_1, \dots, A_n \in \mathbb{F}_q$. In other words, $\mathcal{G}(A_1, \dots, A_n)$ and $\mathcal{F}(A_1, \dots, A_n)$ are equal as polynomial functions over $\mathbb{F}_q$.

Proof. (i) For A_i, $A_i^q - A_i$ is a given generator of J_0. $A_1, \dots, A_n$ are also the last variables in the abstraction term order. Moreover, A_i is an input to the circuit, so A_i is an independent variable. As a result, $G_{d+1} = G \cap \mathbb{F}_{2^k}[A_1, \dots, A_n] = \{A_1^q - A_1, \dots, A_n^q - A_n\}$.

(ii) Since $f : Z + \mathcal{F}(A_1, \dots, A_n)$ is a polynomial representation of the function of the circuit, $Z + \mathcal{F}(A_1, \dots, A_n) \in J + J_0$, as described above. Therefore, according to the definition of a Gröbner basis, the leading term of $Z + \mathcal{F}(A_1, \dots, A_n)$ (which is Z) should be divisible by the leading term of some polynomial $g_i \in G$. The only way $lt(g_i)$ can divide Z is when $lt(g_i) = Z$ itself. Moreover, due to our abstraction (lex) term order, $Z > A > \cdots > A_n$, so this polynomial must be of the form $Z + \mathcal{G}(A_1, \dots, A_n)$.

(iii) As $Z = \mathcal{F}(A_1, \dots, A_n)$ represents the function of the circuit, $Z+\mathcal{F}(A_1, \dots, A_n) \in J + J_0$. Moreover, $V(J + J_0) \subset V(Z + \mathcal{F}(A_1, \dots, A_n))$. By projecting this variety $V(J + J_0)$ onto the co-ordinates corresponding to $(A_1, \dots, A_n, Z)$, we obtain the *graph of the function* $(A_1, \dots, A_n) \mapsto \mathcal{F}(A_1, \dots, A_n)$ from $\mathbb{F}_{2^k} \to \mathbb{F}_{2^k}$. Since $Z + \mathcal{G}(A)$ is an element of the Gröbner basis of $J + J_0$, $V(J + J_0) \subset V(Z + \mathcal{G}(A_1, \dots, A_n))$. Therefore, $Z = \mathcal{G}(A_1, \dots, A_n)$ gives the same function as $Z = \mathcal{F}(A_1, \dots, A_n)$, for all $A_i \in \mathbb{F}_{2^k}$. ■

As a consequence of the Abstraction Theorem, computing a Gröbner basis G of $J+J_0$ using the abstraction term order finds a polynomial of the form $Z + \mathcal{G}(A_1, \dots, A_n)$ in the Gröbner basis, such that $Z = \mathcal{G}(A_1, \dots, A_n)$ is a polynomial representation of the circuit. However, if the Gröbner basis is not reduced, it is possible to obtain multiple polynomials in G of the form $Z + \mathcal{G}_1(A_1, \dots, A_n), Z + \mathcal{G}_2(A_1, \dots, A_n), \dots$,; all of which correspond to the same function.

Corollary 5.1 *By computing a* **reduced** *Gröbner basis G_r of $J + J_0$, G_r will contain one and only one polynomial in of the form $Z+\mathcal{G}(A_1, \dots, A_n)$, such that $Z = \mathcal{G}(A_1, \dots, A_n)$ is the* **unique, minimal, canonical** *representation of the function $\mathcal{F}$ implemented by the circuit.*

Proof. Any function $f : \mathbb{F}_{2^k}^d \to \mathbb{F}_{2^k}$ has a unique canonical representation as polynomial $P_f \in \mathbb{F}_{2^k}[x_1, ..., x_d]$ such that all its nonzero monomials are of the form $x_1^{i_1} \cdots x_d^{i_d}$ where $0 \le i_j \le q - 1$, for all $j = 1, \dots d$.

Let J_0 be the ideal of all polynomials that vanish over $\mathbb{F}_{2^k}[x_1, \dots, x_d]$. The generators of J_0 are polynomials in the form $x_i^{2^{q_i}} - x_i$, where q_i is the datapath size of x_i. Then these generators also form a reduced Gröbner basis for J_0. This implies that the elements $A_h^{2^k} - A_h, 1 \le h \le n$ will have to be part of the reduced Gröbner basis of $J + J_0$.

Corollary 1.8.6 in [10] shows that the obtained element $\mathcal{F}(A_1, ..., A_n)$ that is reduced modulo $A_h^{2^k} - A_h, 1 \leq h \leq n$. Thus, the polynomial representation of $\mathcal{F}$ in the reduced Gröbner basis is the unique canonical representation. ■

Example 5.5 *Consider the 2-bit multiplier Example 5.4, for which we have already generated $J + J_0$. We apply abstraction term order $>$, i.e a lex order with "bit-level variables" $>$ "Output Z" $>$ "Inputs A, B".*

When we compute the reduced Gröbner basis, G_r, of $\{J + J_0\}$ with respect to this ordering, $G_r = \{g_1, \ldots, g_{14}\}$:

$$g_1 : B^4 + B; \;\; g_2 : b_0 + b_1\alpha + B; \;\; g_3 : a_0 + a_1\alpha + A;$$
$$g_4 : c_0 + c_1\alpha + c_2\alpha + c_3(\alpha + 1) + Z; g_5 : r_0 + c_1 + c_2; \;\; g_6 : z_0 + c_0 + c_3;$$
$$g_7 : z_1 + r_0 + c_3; \;\; \mathbf{g_8} : \mathbf{Z + A \cdot B}; \;\; g_9 : b_1 + B^2 + B; \;\; g_{10} : a_1 + A^2 + A;$$
$$g_{11} : c_3 + a_1 \cdot b_1 g_{12} : c_2 + a_1 \cdot b_1\alpha + a_1 \cdot B; \;\; g_{13} : c_1 + a_1 \cdot b_1\alpha + b_1A; \;\; g_{14} : A^4 + A$$

$g_8 = Z + A \cdot B$ is the **canonical, word-level polynomial** *representing the function performed by the multiplier $Z = A \cdot B$.*

Consolidating our results, the proposed abstraction approach is described as follows:

1. Given a bit-level implementation of a Galois field arithmetic circuit C over a given $\mathbb{F}_{2^k}$, with k-bit output Z and k-bit inputs $A_1, \ldots, A_n$.

2. C performs some unknown function $Z = \mathcal{F}(A_1, \ldots, A_n)$.

3. Model the the circuit as a system of polynomials $\{f_1, \ldots, f_s\} \subset \mathbb{F}_{2^k}[x_1, \ldots, x_d, Z, A_1, \ldots, A_n]$ as described above and let J be the ideal generated by these polynomials.

4. Let J_0 be the ideal generated by all vanishing polynomials of $\mathbb{F}_{2^k}$.

5. By computing a reduced Gröbner basis G_r of ideal $J + J_0$ using **abstraction term order**, the word-level polynomial $Z + \mathcal{F}(A_1, \ldots, A_n)$ will be found in G_r.

5.4 Experimental Results: Validation of the Approach

Our experiments take, as inputs, Mastrovito [63] multiplier circuits of various word sizes k. Each multiplier performs the polynomial function $Z = \mathcal{F}(A, B) = A \cdot B$ over a Galois field, where Z is the k-bit output and A and B are the k-bit inputs. We extract the Boolean gate-level operators J and vanishing polynomials J_0. Then we compute the reduced Gröbner basis G_r of $J + J_0$ with respect to abstraction term order $>$. The resulting G_r contains a polynomial $Z + A \cdot B$.

The experiments were designed as scripts in the computer algebra tool, SINGULAR [73], which provides functionality for polynomial computations over rings and fields. This tool provides a number of efficient polynomial algorithms. The ring $\mathbb{F}_{2^k}[x_1, \dots, x_f, Z, A, B]$ is defined over the abstraction ordering, using the same primitive polynomial ("minpoly" in Singular) $P(X)$ that was used to design the Galois field multiplier. Ideals J and J_0 were provided using their generating polynomials and the the reduced Gröbner basis computation of $J + J_0$ was performed using the "slimgb" command.

The experiments were conducted on a 64-bit Ubuntu machine running a 2.4GHz processor with 8GB of memory. We applied our approach to abstract the canonical, polynomial representation of Mastrovito multipliers of various sizes. Our machine was unable to perform the computations of the Gröbner basis of multipliers beyond 40-bit word inputs.

Table 5.1: Run-time of Gröbner Basis Computation of Mastrovito Multipliers in Singular using Abstraction Term Order $>$.

Word Size (k)	Number of Polynomials (d)	Computation Time (minutes)
16	$1,871$	2.4
24	$3,135$	12
32	$5,549$	22.6
40	$8,587$	266
48	$12,327$	NA (Out of Memory)

5.5 Conclusions

The above approach is guaranteed to find a canonical, word-level representation of the function $\mathcal{F}$ performed by a circuit C over $\mathbb{F}_{2^k}$. However, the Gröbner basis computation is prohibitively complex for circuits of practical sizes. The next chapter proposes a method to overcome the complexity of the Gröbner basis computation in order to make this abstraction approach scalable.

CHAPTER 6

OVERCOMING GROBNER BASIS COMPLEXITY FOR ABSTRACTION

Computing a Gröbner basis is prohibitively expensive for large circuits. The approach from the last chapter is limited only to small circuits, with data-paths no larger than 40-bits. A full Gröbner basis computation results in numerous polynomials, but the abstraction approach "searches" for only one polynomial $(Z + \mathcal{G}(A))$ in the basis. This motivates an investigation into whether it is possible to *guide a sequence of* $Spoly(f, g) \xrightarrow{J+J_0}_+ r$ *computations* to arrive at the desired word-level polynomial. This chapter describes this *smaller subset* of computations, which are derived from a Gröbner basis analysis, to find the word-level polynomial of the function performed by a given circuit. The improved approach can abstract canonical word-level representations of circuits up to 571 bits, corresponding to the largest NIST-specified ECC standard.

6.1 Improving the Abstraction Approach

Consider the word-level abstraction problem formulation from Chapter 5. J is the ideal generated by all polynomials derived from the circuit implementation and J_0 is the ideal of all the vanishing polynomials of every variable in the ring. The computation of the reduced Gröbner basis of $J + J_0$ over $\mathbb{F}_q$ has the following known complexity [67]:

Theorem 6.1 *Let* $J + J_0 = \langle f_1, \dots, f_s,\ x_1^q - x_1, \dots, x_d^q - x_d \rangle \subset \mathbb{F}_q[x_1, \dots, x_d]$ *be an ideal. The time and space complexity of Buchberger's algorithm to compute a Gröbner basis of* $J + J_0$ *is bounded by* $q^{O(d)}$.

In our case $q = 2^k$, and when k and d are large, this complexity makes abstraction infeasible.

Recall that Buchberger's algorithm [103] for computing Gröbner bases depends on the computation of an S-polynomial, which is then reduced by all the polynomials in the basis.

$$Spoly(f_i, f_j) \xrightarrow{G'}_+ r$$

where

$$Spoly(f, g) = \frac{L}{lt(f)} \cdot f - \frac{L}{lt(g)} \cdot g$$

$$L = lcm(lt(f), lt(g))$$

A new polynomial is added to the basis when the remainder of the $Spoly$ reduction, r, is non-zero.

Notice that our approach searches for only one polynomial $f_{\mathcal{F}} : Z + \mathcal{F}(A_1, \dots, A_n)$, and it does by computing the entire reduced Gröbner basis, $G = \{g_1, \dots, g_m\}$ and finding $f_{\mathcal{F}} \in \{g_1, \dots, g_m\}$. This motivates us to investigate whether it's possible to *guide a sequence of* $Spoly(f, g) \xrightarrow{J+J_0}_+ r$ *computations* to arrive at the desired word-level polynomial, without considering other polynomials in the generating set.

Numerous improvements have been introduced to improve the efficiency of Buchberger's algorithm. One of these is the product criterion, the results of which we exploit for our approach.

Lemma 6.1 *[Product Criterion [104]] Let* $\mathbb{F}$ *be any field, and* $f, g \in \mathbb{F}[x_1, \cdots, x_d]$ *be polynomials. If the equality* $lm(f) \cdot lm(g) = LCM(lm(f), lm(g))$ *holds, then* $Spoly(f, g) \xrightarrow{G}_+ 0$.

The above result states that when the leading monomials of f, g are relatively prime then $Spoly(f, g)$ always reduces to 0 modulo G. In this case, $Spoly(f, g)$ need not be considered in Buchberger's algorithm, and thus the computation is avoided. Recall that in the Abstraction Term Order (Definition 5.1), we have "circuit variables $x_1, \dots, x_d$" $>$ "word-level output" $>$ "word-level inputs", where the relative ordering among $x_1, \dots, x_d$ is not important. This ordering is now further refined to exploit the product criteria.

Given an acyclic combinational circuit, an ordering can be applied to the bit-level variables, $\{x_1, \dots, x_d\} \in \mathbb{F}_2$, based on their topological position in the circuit. In a *reverse topological ordering*, the output variable of the gate will always come earlier in the ordering than any of its input variables.

Definition 6.1 Refined Abstraction Term Order (RATO) $>_r$: *Given a circuit C, apply a reverse topological ordering to the bit-level variables $\{x_1, \dots, x_d\}$. Then, impose a lex term order $>_r$ on $\mathbb{F}_q[x_1, \dots, x_d, Z, A_1, \dots, A_n]$ with "circuit variables ordered reverse topologically" $>$ "output word-level variable" $>$ "input word-level variables".*

Let F be the set of polynomials which generate J, i.e. $J =< F >$. When RATO is applied, we find that all *bit-level circuit constraint polynomials* in F have leading terms that are relatively prime to each other. Since we are using a reverse topological variable ordering with lex term ordering, these polynomials are in the form of $f_i = x_i + \text{tail}(f_i)$, where x_i is the output of a gate f_i, and thus the following proposition from *Lv's* work [11] can be applied.

Proposition 6.1 *Let C be any arbitrary combinational circuit. Let $\{x_1, \dots, x_d\}$ denote the set of all variables (signals) in the circuit, i.e. the primary input, intermediate and primary output variables. Perform a* **reverse topological traversal** *of the circuit and order the variables such that $x_i > x_j$ if x_i appears earlier in the reverse topological order. Impose a lex term order to represent the Boolean expression for each gate as a polynomial f_i; then $f_i = x_i + tail(f_i)$. Then the set of all polynomials $\{f_1, \dots, f_s\}$ forms a Gröbner basis, as $lt(f_i)$ and $lt(f_j)$ for $i \neq j$ are relatively prime.*

Example 6.1 *Consider the 2-bit multiplier from Example 5.4. With RATO applied, the bit-level circuit constraint polynomials in F, where $J =< F >$, are:*

$$f_1 : c_0 + a_0 \cdot b_0$$

$$f_2 : c_1 + a_0 \cdot b_1$$

$$f_3 : c_2 + a_1 \cdot b_0$$

$$f_4 : c_3 + a_1 \cdot b_1$$

$$f_5 : r_0 + c_1 + c_2$$

$$f_6 : z_0 + c_0 + c_3$$

$$f_7 : z_1 + r_0 + c_3$$

The leading terms of $f_1, \ldots, f_7$ are relatively prime to each other.

Let F_0 be the set of polynomials which generate J_0, i.e. $J_0 =< F_0 >$. In $F \cup F_0$, for every polynomial $f_i = x_i + \text{tail}(f_i)$ in F there is a vanishing polynomial $v_i = x_i^2 + x_i$ in F_0; f_i and v_i have leading terms which are not relatively prime. In this case, [11] shows that $Spoly(x_i + \text{tail}(f_i), x_i^{2^k} - x_i) \xrightarrow{J,J_0}_+ r$ always produces $r = 0$, and thus can be excluded from the Gröbner basis computation.

Theorem 6.2 *Let $q = 2^k$, and let $\mathbb{F}_q[x_1, \ldots, x_d]$ be a ring on which we have a reverse topological lex order. Let I be a subset of $\{1, \ldots, d\}$. For all $i \in I$, let $f_i = x_i + P_i$ (where $P_i = tail(f_i)$) such that all indeterminates x_j that appear in P_i satisfy $x_i > x_j$. Then the set $G = \{f_i : i \in I\} \cup \{x_1^q - x_1, \ldots, x_d^q - x_d\}$ is a Gröbner basis.*

The proof is given in [11] and is reproduced here:

Proof. Given a system of polynomials derived from a circuit over $\mathbb{F}_q[x_1, \ldots, x_d]$, where $\{x_1, \ldots, x_d\}$ are bit-level variables. Apply a reverse topological lex ordering to $\{x_1, \ldots, x_d\}$. Let x_i be the output of a Boolean logic gate for some $1 \leq i \leq d$. Let $f = x_i + P_i$ be the polynomial derived from this logic gate and $g = x_i^q - x_i$ be vanishing polynomial of x_i. Then $Spoly(f, g) = x_i^{q-1} f - g = x_i^{q-1} P_i + x_i$. In what follows, it is important to note that the indeterminates appearing in P_i are all less than x_i over the given ordering.

First, $x_i^{q-1}P_i + x_i - x_i^{q-2}P_i(x_i + P_i) = x_i^{q-2}P_i^2 + x_i$, which shows that $x_i^{q-1}P_i + x_i \xrightarrow{x_i+P_i} x_i^{q-2}P_i^2 + x_i$.

Next, $x_i^{q-2}P_i^2 + x_i - x_i^{q-3}P_i^2(x_i + P_i) = x_i^{q-3}P_i^3 + x_i$. Continuing in this fashion, we get $x_i P_i^{q-1} + x_i - P_i^{q-1}(x_i + P_i) = x_i + P_i^q$, and finally $x_i + P_i^q - (x_i + P_i) = P_i^q - P_i$. Hence,

$$x_i^{q-1}P_i + x_i \xrightarrow{x_i+P_i} x_i^{q-2}P_i^2 + x_i \xrightarrow{x_i+P_i} x_i^{q-3} + x_i \xrightarrow{x_i+P_i} \cdots$$
$$\cdots \xrightarrow{x_i+P_i} P_i^q + x_i \xrightarrow{x_i+P_i} P_i^q - P_i.$$

Over the finite field $\mathbb{F}_q$, $P_i^q - P_i$ is a vanishing polynomial. Therefore, $P_i^q - P_i \in I(V(J_0)) = \langle x_1^q - x_1, \ldots, x_d^q - x_d \rangle$. Due to the product criterion (Lemma 6.1), $G_0 = \{x_1^q - x_1, \ldots, x_d^q - x_d\}$ is Gröbner basis. Therefore $P_i^q - P_i \xrightarrow{G_0}_+ 0$. ■

Due to RATO, there exists a polynomial $f_{z_i} \in F$ which is the polynomial derived from a Boolean logic gate, where z_i is the first variable in the ordering for some $0 \leq i < k$. That is, $f_{z_i} : z_i + \text{tail}(f_{z_i})$.

Proposition 6.2 *Over RATO, the polynomial pair (f_Z, f_{z_i}) is the only critical pair at the start of the Gröbner basis computation of $J + J_0$, where f_{z_i} is the polynomial derived from the gate*

Proof. Due to Theorem 6.2 and the product criterion, a critical pair must come from a word level designation polynomial, $\{f_Z, f_{A_1}, \ldots, f_{A_n}\}$, and a polynomial derived from a Boolean logic gate. The leading terms of the polynomials $\{f_{A_1}, \ldots, f_{A_n}\}$ are bit-level inputs to the circuit and thus are not the outputs of any gate. Thus, the only critical pair if f_Z and f_{z_i}, where z_i is the first variable in the ordering and is thus the leading monomial of f_Z. ■

Thus, the first computation of the Gröbner basis is guaranteed to be $Spoly(f_Z, f_{z_i}) \xrightarrow{F,F_0}_+ r$. This computation has the following interesting property.

Proposition 6.3 *Normalize f_Z, i.e. f_Z=$f_Z/LC(f_Z)$. Then*

$$Spoly(f_Z, f_{z_i}) \xrightarrow{F,F_0}_+ r \quad \textit{is equivalent to} \quad f_Z \xrightarrow{F-\{f_Z\},F_0} +r \tag{6.1}$$

Proof. Assuming both f_{z_i} and f_Z are minimized, i.e. $LC(f_{z_i}) = LC(f_Z) = 1$, they are both in the form $z_i + P$ for some polynomial P. Let $f = z_i + P_f$ and $g = z_i + P_g$ represent f_{z_i} and f_Z respectively.

(i) For $Spoly(f, g)$

$$L = LCM(LT(f), LT(g)) = LM(f) = LM(g) = z_i \tag{6.2}$$

so

$$\begin{aligned} Spoly(f, g) &= \frac{L}{lt(f)} \cdot f - \frac{L}{lt(g)} \cdot g \\ &= \frac{z_i}{z_i} \cdot f - \frac{z_i}{z_i} \cdot g \\ &= f + g \end{aligned}$$

Thus $Spoly(f_Z, f_{z_i}) \xrightarrow{F,F_0}_+ r$ is equivalent to $f + g \xrightarrow{F,F_0}_+ r$.

(ii) For $f_Z \xrightarrow{F-\{f_Z\},F_0} +r$, since the leading term of f_Z is z_i, the only polynomial in the set $\{F - \{f_Z\}, F_0\}$ which can perform the first division is f_{z_i}. Again, denote f_Z as f and f_{z_i} as g. According to the reduction algorithm, the remainder r of $f \xrightarrow{g}_+ r$ is:

$$\begin{aligned} r &= f - lt(f)/lt(g) \cdot g \\ &= f - z_i/z_i \cdot g \\ &= f - g \\ &= f + g \end{aligned} \tag{6.3}$$

Thus $f_Z \xrightarrow{F-\{f_Z\},F_0} +r$ is equivalent to $(f + g) \xrightarrow{F-\{f_Z\},F_0} +r$

(iii) Consider $(f + g) \xrightarrow{F,F_0} +r$. $f = z_i + P_f$ and $g = z_i + P_g$. So $f + g = 2z_i+P_g+P_f = P_g+P_f$ no longer contains z_i as the leading monomial. As a consequence of RATO, as reduction proceeds the remainder will never again contain z_i since z_i is the first variable in the ordering. Since the leading monomial of f_Z is z_i, f_Z will never be used for reduction. Therefore $(f + g) \xrightarrow{F,F_0} +r$ is equivalent to $(f + g) \xrightarrow{F-\{f_Z\},F_0} +r$

Thus, $Spoly(f_Z, f_{z_i}) \xrightarrow{F,F_0}_+ r$ is equivalent to $f_Z \xrightarrow{F-\{f_Z\},F_0}_+ r$ ■

Thus, $f_Z \xrightarrow{F-\{f_Z\},F_0}_+ r$ is the first computational step of the abstraction. The polynomial remainder r will *not* contain any bit-level variable corresponding to the output of any gate in the design; i.e. primary output bits and intermediate variables of the circuit do not appear in r. To prove this, assume that a non-primary-input variable x_j appears in a monomial term m_j in r. Since there always exists a polynomial f_j such that

$f_j = x_j + \text{tail}(f_j)$, $lt(f_j)$ divides monomial m_j and m_j can be canceled. Therefore, all such terms m_j with non-primary-input bit-level variables can be eliminated.

Two cases need to be considered:

1. Remainder r only contains word-level variables: word-level output Z and the word-level inputs $A_1, \ldots, A_n$. Since RATO is lex with $Z > \{A_1, \ldots, A_n\}$, the remainder r is the desired canonical polynomial representation, $Z + \mathcal{F}(A_1, \ldots, A_n)$.

2. Remainder r contains both the bit-level primary input variables, as well as the word-level variables.

Example 6.2 *Again consider the 2-bit multiplier from Example 6.1. RATO for this example is*

$$z_1 > z_0 > r_0 > c_0 > c_1 > c_2 > c_3 > a_0 > a_1 > b_0 > b_1 > Z > A > B \tag{6.4}$$

$F + F_0$ with RATO applied is:

$$\left.\begin{aligned} f_1 &: c_0 + a_0 \cdot b_0 \\ f_2 &: c_1 + a_0 \cdot b_1 \\ f_3 &: c_2 + a_1 \cdot b_0 \\ f_4 &: c_3 + a_1 \cdot b_1 \\ f_5 &: r_0 + c_1 + c_2 \\ f_6 &: z_0 + c_0 + c_3 \\ f_7 &: z_1 + r_0 + c_3 \end{aligned}\right\} \quad \textit{Bit-level circuit constraints } (\subset J)$$

$$\left.\begin{aligned} f_A &: a_0 + a_1 \cdot \alpha + A \\ f_B &: b_0 + b_1 \cdot \alpha + B \\ f_Z &: z_1 \cdot \alpha + z_0 + Z \end{aligned}\right\} \quad \textit{Word-level designation } (\subset J)$$

$$\left.\begin{aligned} a_0^2 - a_0,\ a_1^2 - a_1,\ b_0^2 - b_0,\ b_1^2 - b_1 \\ c_0^2 - c_0,\ c_1^2 - c_1,\ c_2^2 - c_2,\ c_3^2 - c_3 \\ r_0^2 - r_0,\ z_0^2 - z_0,\ z_1^2 - z_1 \\ A^4 - A,\ B^4 - B,\ Z^4 - Z \end{aligned}\right\} \quad \textit{vanishing polynomials} (J_0)$$

Notice that the leading monomial of f_Z is z_1, which is also the leading monomial of f_7. Minimize f_Z, $f_{Zmin} = f_Z/\alpha = z_1 + z_0 \cdot (\alpha + 1) + Z \cdot (\alpha + 1)$. By computing the S-polynomial of f_{Zmin} and f_7:

$$Spoly(f_{Zmin}, f_7) \xrightarrow{F,F_0}_{+} r$$

the remainder r is $Z + A \cdot B$.

Likewise, if we reduce f_Z by $F - \{f_Z\}, F_0$, that is reduce it by all generators in $J + J_0$ except f_Z:

$$f_Z \xrightarrow{F-\{f_Z\},F_0} +r$$

the remainder r is $Z + A \cdot B$.

Example 6.3 *Now consider a 2-bit multiplier which has a* **bug***. The output lines, z_0 and z_1, have been swapped, as shown in Figure 6.1 below:*

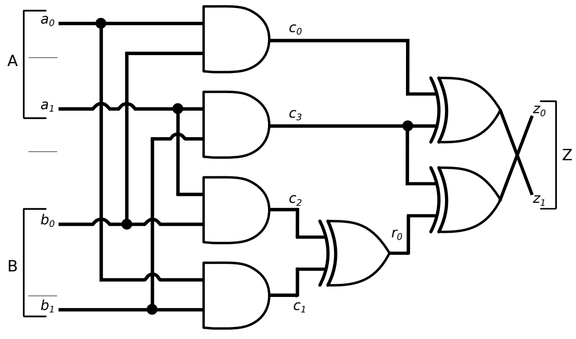


Figure 6.1: A buggy 2-bit multiplier over $\mathbb{F}(2^2)$.

The polynomials in $F + F_0$ are the same as in Example 6.2 except for the following changes:

$$f_6 : z_1 + c_0 + c_3$$

$$f_7 : z_0 + r_0 + c_3$$

$$f_Z : z_1 + z_0 \cdot \alpha + Z$$

f_Z has common leading terms with f_6. Computing

$$Spoly(f_Z, f_6) \xrightarrow{F,F_0}_{+} r$$

gives remainder $r : a_1 \cdot b_1\alpha + a_1 \cdot B\alpha + b_1 \cdot A\alpha + Z + A \cdot B$, which is the same result as computing $f_Z \xrightarrow{F-\{f_Z\},F_0}_+ r$

6.2 Improving Polynomial Division using $F4$-style Reduction

The most intensive computational step in our proposed improvement is that of polynomial division $f_Z \xrightarrow{F-\{f_Z\},F_0}_+ r$. When the circuit C is very large, the polynomial set $\{F - \{f_Z\}, F_0\}$ also becomes extremely large. This division procedure then becomes the bottleneck in our abstraction approach. In principle, this reduction can be performed using contemporary computer-algebra systems — *e.g.*, the SINGULAR [73] tool, which is widely used within the verification community [78] [80] [75]. In our work, we have also performed experiments with SINGULAR. However, as in any "general-purpose" computer algebra tool, the data-structures are not specifically optimized for circuit verification problems. Moreover, SINGULAR also limits the number of variables (d) that it can accommodate in the system to $d < 32767$; this limits its application to large circuits. Recent symbolic computation techniques [11] have shown improvements from employing the concept of $F4$-style polynomial reduction [105]. Therefore, to further improve our approach, we exploit this relatively recent concept, which implements polynomial division using row-reductions on a matrix, to develop a custom verification tool to perform this reduction efficiently.

$Faug\grave{e}re$'s $F4$ approach [105] presents a new algorithm to compute a Gröbner basis. It uses the same mathematical principles as Buchberger's algorithm. However, instead of computing and reducing one S-polynomial at a time, it computes many S-polynomials in one step and reduces them simultaneously using sparse linear algebra on a matrix (triangulation). We can use this efficient reduction technique to perform our reduction, $f_Z \xrightarrow{F-\{f_Z\},F_0}_+ r$, by representing and solving it on a matrix. First, let us consider the following example that demonstrates the main concepts behind the reduction approach of $F4$.

Example 6.4 *Consider the lex term order with $x > y > z$ on the ring $\mathbb{Q}[x, y, z]$. Given $F = \{f_1 = 2x^2 + y, f_2 = 3xy^2 - xy, f_3 = 4y^3 - 1\}$, consider one step of Buchberger's algorithm: $S(f_1, f_2) \xrightarrow{f_1,f_2,f_3}_+ r$. We have, $Spoly(f_1, f_2) = \frac{1}{3}x^2y + \frac{1}{2}y^3 = f_4$. The reduc-*

tion $Spoly(f_1, f_2) \xrightarrow{f_1,f_2,f_3}_+ (-\frac{1}{6}y^2 + \frac{1}{8})$ *is done as follows: Since* $lt(f_1) \mid lt(f_4),\ f_4 \xrightarrow{f_1} h$ *is computed as:*

$$h = f_4 - \frac{lt(f_4)}{lt(f_1)} f_1 = f_4 - \frac{1}{6} y f_1 = \frac{1}{2}y^3 - \frac{1}{6}y^2;$$

Now $lt(f_2)$ *does not divide any term in* h*, but* $lt(f_3) \mid lt(h)$*, so* $f \xrightarrow{f_3} r$*:*

$$r = h - \frac{lt(h)}{lt(f_3)} f_3 = \frac{1}{2}y^3 - \frac{1}{6}y^2 - \frac{1}{8}f_3 = -\frac{1}{6}y^2 + \frac{1}{8}$$

This reduction procedure can also be simulated on a matrix using Gaussian elimination. The reduction above requires the computation of $\frac{1}{6}yf_1$ *and* $\frac{1}{8}f_3$*. Ignoring the coefficients* $\frac{1}{6}, \frac{1}{8}$*, we can generate all the monomials required in the reduction process: i.e. monomials of* f_4, yf_1, f_3*, and setup the problem of cancellation of terms as Gaussian elimination on a matrix. Monomials of* f_4, yf_1, f_3 *are, respectively,* $\{x^2y, y^3\}, \{x^2y, y^2\}, \{y^3, 1\}$*. Let the rows of a matrix* M *correspond to polynomials* $[f_4, yf_1, f_3]$*, and columns correspond to all the monomials (in lex order)* $[x^2y, y^3, y^2, 1]$*. Then the matrix* M *shows the representation of these polynomials where the entry* $M(i,j)$ *is the coefficient of monomial of column* j *present in the polynomial of row* i*.*

$$M = \begin{array}{c|cccc|} & x^2y & y^3 & y^2 & 1 \\ f_4 & \frac{1}{3} & \frac{1}{2} & 0 & 0 \\ yf_1 & 2 & 0 & 1 & 0 \\ f_3 & 0 & 4 & 0 & -1 \end{array}$$

Now, reducing M *to a row echelon form using Gaussian elimination gives:*

$$M = \begin{array}{c|cccc|} & x^2y & y^3 & y^2 & 1 \\ f_4 & \frac{1}{3} & \frac{1}{2} & 0 & 0 \\ h = f_4 - \frac{1}{6}yf_1 & 0 & \frac{1}{3} & -\frac{1}{6} & 0 \\ r = h - \frac{1}{8}f_3 & 0 & 0 & -\frac{1}{6} & \frac{1}{8} \end{array}$$

The last row $(0, 0, -\frac{1}{6}, \frac{1}{8})$ *accounts for polynomial* $-\frac{1}{6}y^2 + \frac{1}{8}y$ *which is equal to the reduction result* r *obtained before.*

This approach generates all the monomial terms that are required in the division process, and the coefficients required for cancellation of terms are accounted for by elementary row reductions in the subsequent Gaussian elimination. Based on the above concepts, a matrix can be constructed for our problem: $f_Z \xrightarrow{F-\{f_Z\},F_0} +r$.

Definition 6.2 *Let $L = [f_1, \ldots, f_m]$ be a list of m polynomials. Let M_L be an ordered list of monomials of elements of L and let n be the number of elements in M_L. Define M as the $m \times n$ matrix which associates the polynomials of L to rows and monomials of M_L to columns. Entry in row i, column j is the coefficient of the j^{th} element of M_L in f_i.*

Algorithm 4: Generating the Matrix for Polynomial Reduction

```
Input: $f_Z, \{F - \{f_Z\}, F_0\}$ as $\{f_1, \ldots, f_s\}$, RATO $>_r$
Output: Remainder $r$ of $f_Z \xrightarrow{f_1,\ldots,f_s}_+ r$
/*$L =$ set of polynomials, rows of $M$*/;
L:=$\{f_Z\}$ ;
/*$M_L =$ the set of monomials, columns of $M$ */;
$M_L$:={ monomials of f} ;
for ($i = 0$; $i \leq$num monomials in $M_L$ ; i++) do
    mon:= the $i^{th}$ monomial of $M_L$;
    Identify $f_k \in F$ satisfying: $lm(f_k)$ can divide $mon$ ;
    /*add polynomial $f_k$ to L as a new row in $M$ */;
    $L := L \cup \frac{mon}{lm(f_k)} \cdot f_k$ ;
    /*Add monomials to $M_L$ as new columns in $M$ */;
    $M_L$:=$M_L \cup \{$monomials of $\frac{mon}{lm(f_k)} \cdot f_k\}$ ;
end
Gaussian Elimination on $M$;
return $r =$ last row of $M$;
```

Algorithm 4 describes our procedure to generate the matrix M of polynomials corresponding to our reduction procedure. The main idea is to setup the rows and columns of the matrix in a way that polynomial division can be subsequently performed by applying Gaussian elimination on M. In the algorithm, the set of polynomials $\{F - \{f_Z\}, F_0\} = \{f_1, \ldots, f_s\}$ correspond to the circuit constraints and RATO is imposed on the polynomials. The output word-level polynomial f_Z is to be reduced w.r.t. $\{f_1, \ldots, f_s\}$. Initially, $L = \{f_Z\}$ is inserted as the first row of the matrix and M_L constitutes the (ordered) list of monomials of f_Z. Then, in every iteration i, a polynomial $f_k \in \{f_1, \ldots, f_s\}$

is identified such that $lm(f_k)$ divides the i^{th} monomial (mon) of M_L; this is to enable cancellation of the corresponding monomial term. The computation $L := L \cup \frac{mon}{lm(f_k)} \cdot f_k$ in the while-loop, generates the polynomials required for reduction.

The list M_L is updated to include monomials of $\frac{mon}{lm(f_k)} \cdot f_k$. Finally, the iteration in the loop terminates when all monomials of M_L have been analyzed. The loop is guaranteed to terminate once mon contains only word-level variables, as no polynomials in $\{f_1, \ldots, f_s\}$ have a leading term that contains a word-level variable over RATO.

Using the set L as rows and M_L as columns, a matrix M is constructed and Gaussian elimination is applied to reduce it to row-echelon form. The last row in the reduced matrix corresponds to the reduction result r. Let us describe the approach using an example.

Example 6.5 *Consider the reduction related to the abstraction of the $\mathbb{F}_{2^2}$ multiplier circuit from Example 6.2. The word-level output designation polynomial f_Z is $z_1\alpha + z_0 + Z$, and the circuit polynomials are*

$$\begin{aligned} f_1 &: a_0 + a_1\alpha + A \\ f_2 &: b_0 + b_1\alpha + B \\ f_3 &: r_0 + a_0b_1 + a_1b_0 \\ f_4 &: z_0 + a_0b_0 + a_1b_1 \\ f_5 &: z_1 + r_0 + a_1b_1 \end{aligned}$$

Here $P(x) = x^2 + x + 1$, and $P(\alpha) = 0$. We have to compute $f_Z \xrightarrow{f_1,\ldots,f_5}_+ r$. Note that, for simplicity, variables c_0, c_1, c_2, c_3 from Example 5.3 have been substituted by functions on primary inputs. Impose RATO on the polynomials as follows:

$$z_1 > z_0 > r_0 > a_0 > a_1 > b_0 > b_1 > Z > A > B \tag{6.5}$$

The algorithm constructs the matrix as follows:

1. *Initialization: $L = \{f_Z\} = \{z_1\alpha + z_0 + Z\}$. $M_L = \{z_1, z_0, Z\}, i = 1, mon = z_1$ (i^{th} monomial of M_L).*

2. *Iteration 1: Identify a polynomial* $f_k \in \{f_1, \ldots, f_s\}$ *s.t.* $lm(f_k) \mid mon$*. Clearly,* $f_k = f_5 = z_1 + r_0 + a_1 b_1$*. Then,* $L = L \cup \frac{mon}{lt(f_k)} \cdot f_k = L \cup f_5$*. Therefore,* $L = \{f, f_5\}$ *and* $M_L = \{z_1, z_0, r_0, a_1b_1, Z\}$*,* $i = 2$ *and* $mon = z_0$*.*

3. *Iteration 2:* $f_k = f_4 = z_0 + a_0b_0 + a_1b_1$ *because* $lm(f_4) \mid mon$*. Therefore,* $L = L \cup f_4$ *and* $M_L = \{z_1, z_0, r_0, a_0b_0, a_1b_1, Z\}, i = 3, mon = r_0$*.*

4. *Iteration 3:* $f_k = f_3 = r_0 + a_0b_1 + a_1b_0$ *as* $lt(f_3) \mid mon$*. Therefore,* $L = L \cup f_3$ *and* $M_L = \{z_1, z_0, r_0, a_0b_0, a_0b_1, a_1b_0, a_1b_1, Z\},\ i = 4, mon = a_0b_0$*.*

5. *Iteration 4:* $f_k = f_1 = a_0 + a_1\alpha + A$ *because* $lm(f_1) \mid mon$*. Then* $L = L \cup \frac{a_0b_0}{a_0} \cdot f_1 = L \cup b_0 \cdot f_1 = \{f_5, f_4, f_3, b_0f_1\}$ *and* $M_L = \{z_1, z_0, r_0, a_0b_0, a_0b_1, a_1b_0, a_1b_1, b_0A, Z\}$*.*

6. *Continuing in this fashion . . .*

7. *Iteration 8:* $L = \{f_Z, f_5, f_4, f_3, b_0f_1, b_1f_1, a_1f_2, Af_2\}$*,*
   $M_L = \{z_1, z_0, r_0, a_0b_0, a_0b_1, a_1b_0, a_1b_1, a_1B, b_0A, b_1A, Z, AB\}$*,*
   $i = 9$*,* $mon = AB$*.*

8. *Iteration 8: Since* $mon = AB$ *contains only the word-level inputs, no polynomial in* F *has a leading term that can cancel* mon*, so the loop terminates. The matrix* M *can be constructed using* L *as rows and* M_L *as columns.*

Figure 6.2a shows the matrix M*, and its subsequent Gaussian elimination is shown in Fig. 6.2b. The last row of the reduced matrix corresponds to the reduction* $f_Z \xrightarrow{f_1,\ldots,f_s}_+ r$*, where* $r = Z + A \cdot B$*.*

6.3 Reducing Bit-Level Inputs

When the remainder r only contains word-level variables, the problem of word-level abstraction is solved. Thus, the focus now is to efficiently obtain a word-level abstraction when r also contains bit-level input variables. In this case, a functional mapping is

$$
M = \begin{array}{c|cccccccccccc}
 & z_1 & z_0 & r_0 & a_0b_0 & a_0b_1 & a_1b_0 & a_1b_1 & a_1B & b_0A & b_1A & Z & AB \\
f_Z & \alpha & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
f_5 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
f_4 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
f_3 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
b_0f_1 & 0 & 0 & 0 & 1 & 0 & \alpha & 0 & 0 & 1 & 0 & 0 & 0 \\
b_1f_1 & 0 & 0 & 0 & 0 & 1 & 0 & \alpha & 0 & 0 & 1 & 0 & 0 \\
a_1f_2 & 0 & 0 & 0 & 0 & 0 & 1 & \alpha & 1 & 0 & 0 & 0 & 0 \\
Af_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \alpha & 0 & 1
\end{array}
$$

(a) Matrix M generated by Algorithm 4

$$
M = \begin{array}{c|cccccccccccc}
 & z_1 & z_0 & r_0 & a_0b_0 & a_0b_1 & a_1b_0 & a_1b_1 & a_1B & b_0A & b_1A & Z & AB \\
f_Z & \alpha & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\alpha f_5 - row1 & 0 & 1 & \alpha & 0 & 0 & 0 & \alpha & 0 & 0 & 0 & 1 & 0 \\
f_4 - row2 & 0 & 0 & \alpha & 1 & 0 & 0 & \alpha+1 & 0 & 0 & 0 & 1 & 0 \\
\alpha f_3 - row3 & 0 & 0 & 0 & 1 & \alpha & \alpha & \alpha+1 & 0 & 0 & 0 & 1 & 0 \\
b_0f_1 - row4 & 0 & 0 & 0 & 0 & \alpha & 0 & \alpha+1 & 0 & 1 & 0 & 1 & 0 \\
\alpha b_1f_1 - row5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \alpha & 1 & 0 \\
Af_2 - row6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}
$$

(b) M reduced to row echelon form via Gaussian Elimination

Figure 6.2: F_4-style polynomial reduction on a matrix for Example 6.5.

needed from each bit-level input variable $\{a_0, \dots, a_{k-1}\} \in \mathbb{F}_2$ to the word-level input variable $A \in \mathbb{F}_{2^k}$.

$$
\begin{aligned}
a_0 &= \mathcal{F}_{a_0}(A) \\
&\vdots \\
a_{k-1} &= \mathcal{F}_{a_{k-1}}(A)
\end{aligned} \tag{6.6}
$$

where each $\mathcal{F}_{a_i}$ is some function of A, which needs to be derived. Here, we present the derivation when $\{a_0, \dots, a_{k-1}\} \in \mathbb{F}_2$ and $A \in \mathbb{F}_{2^k}$. However, this result is applicable from any field $\mathbb{F}_q$ to any extension of the field $\mathbb{F}_{q^k}$, i.e. when $\{a_0, \dots, a_{k-1}\} \in \mathbb{F}_q$ and $A \in \mathbb{F}_{q^k}$. This generalized derivation is presented in Appendix A.1.

These mappings from $\{a_0, \dots, a_{k-1}\}$ to A in Eqn.(6.6) are represented as polynomial functions $f_{a_0}, \dots, f_{a_{k-1}}$ in the following form:

$$\begin{aligned} f_{a_0} &: a_0 + \mathcal{F}_{a_0}(A) \\ &\vdots \\ f_{a_{k-1}} &: a_{k-1} + \mathcal{F}_{a_{k-1}}(A) \end{aligned} \tag{6.7}$$

Due to RATO, $\{a_0, \ldots, a_{k-1}\} > A$, thus the leading terms of $f_{a_0}, \ldots, f_{a_{k-1}}$ are $a_0, \ldots, a_{k-1}$ respectively. Let $F_a = \{f_{a_0}, \ldots, f_{a_{k-1}}\}$. Then computing $r \xrightarrow{F_a, F_0}_+ r_w$ ensures that the new remainder r_w must only contain word-level variables. In other words, r_w must be in the form $Z + \mathcal{F}(A)$ and is thus the word-level polynomial representation of the circuit.

Over $\mathbb{F}_{2^k}$, $A = a_0 + a_1\alpha + \cdots + a_{k-1}\alpha^{k-1}$. To compute A^2, a special property of Galois fields, dealing with powers of elements, can be applied.

Lemma 6.2 *(from [85]) Let $\alpha_1, \ldots, \alpha_t$ be any elements in $\mathbb{F}_{p^k}$. Then*

$$(\alpha_1 + \alpha_2 + \cdots + \alpha_t)^{p^i} = \alpha_1^{p^i} + \alpha_2^{p^i} + \cdots + \alpha_t^{p^i} \tag{6.8}$$

for all integers $i \geq 1$.

Lemma 6.2 can be applied to compute A^2:

$$A^2 = a_0^2 + a_1^2\alpha^2 + \cdots + a_{k-1}^2\alpha^{2(k-1)} \tag{6.9}$$

Since each $a_i \in \mathbb{F}_2$, for $0 \leq i < k$, then $a_i^2 = a_i$. This is applied to find the final form for A^2:

$$A^2 = a_0 + a_1\alpha^2 + \cdots + a_{k-1}\alpha^{2(k-1)} \tag{6.10}$$

Similarly, A^4 can be derived as $(A^2)^2$:

$$A^4 = a_0 + a_1\alpha^4 + \cdots + a_{k-1}\alpha^{4(k-1)} \tag{6.11}$$

Deriving A^{2^j} in this manner for all $0 \leq j < k$ gives a system of k equations. These equations can be represented in matrix form, $\mathbf{A} = \mathbf{M}\mathbf{a}$, where $\mathbf{A} = \{A, A^2, \ldots, A^{2^{k-1}}\}^T$, $\mathbf{M}$ is a k by k matrix of coefficients, and $\mathbf{a} = \{a_0, \ldots, a_{k-1}\}^T$:

$$\begin{bmatrix} A \\ A^2 \\ \vdots \\ A^{2^{k-1}} \end{bmatrix} = \begin{bmatrix} 1 & \alpha & \alpha^2 & \ldots & \alpha^{k-1} \\ 1 & \alpha^2 & \alpha^4 & \ldots & \alpha^{(k-1)\cdot 2} \\ 1 & \alpha^4 & \alpha^8 & \ldots & \alpha^{(k-1)\cdot 4} \\ \vdots & \vdots & \vdots & \cdot & \vdots \\ 1 & \alpha^{2^{k-1}} & \alpha^{2\cdot 2^{k-1}} & \ldots & \alpha^{(k-1)\cdot 2^{k-1}} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{k-1} \end{bmatrix} \tag{6.12}$$

Note that $\mathbf{M}$ is a matrix of constants and $\mathbf{A}$ and $\mathbf{a}$ are vectors of variables. However, by interpreting $\mathbf{a}$ as a vector of unknowns, $\mathbf{M}$ and $\mathbf{A}$ as constants. then F_a can be derived by solving Eqn.(6.12) using Gaussian elimination. However, this system of equations also has a special structure which can be exploited to further simplify the abstraction procedure.

Definition 6.3 Cramer's Rule*: Consider a system of* n *linear equations and* n *unknowns,* $x_1, \dots, x_n$*, expressed in matrix form as* $\mathbf{Mx} = \mathbf{b}$*:*

$$\begin{bmatrix} m_{11} & m_{12} & \dots & m_{1n} \\ m_{21} & m_{22} & \dots & m_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{2n} & \dots & m_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{6.13}$$

If the determinant $|\mathbf{M}|$ *is non-zero, then for* $1 \leq i \leq n$*,*

$$x_i = \frac{|\mathbf{M_i}|}{|\mathbf{M}|} \tag{6.14}$$

where $\mathbf{M_i}$ *is* $\mathbf{M}$ *with the* i*-th column replaced with vector* $\mathbf{b}$*:*

$$\mathbf{M_i} = \begin{bmatrix} m_{11} & m_{12} & \dots & m_{1i-1} & b_1 & m_{1i+1} & \dots & m_{1n} \\ m_{21} & m_{22} & \dots & m_{2i-1} & b_2 & m_{2i+1} & \dots & m_{2n} \\ \vdots & \vdots & \cdot & \vdots & \vdots & \vdots & \cdot & \vdots \\ m_{n1} & m_{n2} & \dots & m_{ni-1} & b_n & m_{ni+1} & \dots & m_{nn} \end{bmatrix} \tag{6.15}$$

Definition 6.4 Vandermonde Matrix*: Let* $V(x_1, \dots, x_n)$ *denote a square* n *x* n *matrix of the form*

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \cdot & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^{n-1} \end{bmatrix} \tag{6.16}$$

where elements of each row are presented in a geometric progression. Then $V(x_1, \dots, x_n)$ *is a* **Vandermonde Matrix***, the determinant of which can be computed as:*

$$|V(x_1, \dots, x_n)| = \prod_{1 \leq i < j \leq n} (x_j - x_i) \tag{6.17}$$

This determinant is non-zero if each $x_i \in \{x_1, \dots, x_n\}$ *is a distinct element.*

Notice that $\mathbf{M}$ in Eqn.(6.12) is a square Vandermonde matrix of the form $V(\alpha, \alpha^2, \ldots, \alpha^{2^{k-1}})$.

Lemma 6.3 *The determinant of* $\mathbf{M}$ *as in Eqn.(6.12) is non-zero.*

Proof. *Since* $\mathbf{M}$ *in Eqn.(6.12) is the Vandermonde matrix* $V(\alpha, \alpha^2, \alpha^4, \ldots, \alpha^{2^{k-1}})$*,*

$$|\mathbf{M}| = \prod_{0\le i<j<k} (\alpha^{2^j} - \alpha^{2^i}) \tag{6.18}$$

Since $\mathbb{F}_{2^k}$ *is constructed from a primitive polynomial, every* α^i *is a distinct element for* $0 \le i < 2^k$*. Thus,* $|\mathbf{M}|$ *is non-zero as it is a product of non-zero elements.* ■

Since $|\mathbf{M}|$ is non-zero, Cramer's rule can be applied to derive an equation for every a_i, $0 \le i < k$:

$$a_i = \frac{|\mathbf{M_i}|}{|\mathbf{M}|} \tag{6.19}$$

where $\mathbf{M_i}$ is $\mathbf{M}$ with the column $\{\alpha^i, \alpha^{i\cdot 2}, \ldots, \alpha^{i\cdot 2^{k-1}}\}^T$ replaced by $\mathbf{A}$.

$$\mathbf{M_i} = \begin{bmatrix} 1 & \alpha & \alpha^2 & \ldots & \alpha^{i-1} & A & \alpha^{i+1} & \ldots & \alpha^{k-1} \\ 1 & \alpha^2 & \alpha^4 & \ldots & \alpha^{(i-1)\cdot 2} & A^2 & \alpha^{(i+1)\cdot 2} & \ldots & \alpha^{(k-1)\cdot 2} \\ \vdots & \vdots & \vdots & . & \vdots & \vdots & \vdots & . & \vdots \\ 1 & \alpha^{2^{k-1}} & \alpha^{2\cdot 2^{k-1}} & \ldots & \alpha^{(i-1)\cdot 2^{k-1}} & A^{2^{k-1}} & \alpha^{(i+1)\cdot 2^{k-1}} & \ldots & \alpha^{(k-1)\cdot 2^{k-1})} \end{bmatrix} \tag{6.20}$$

Lemma 6.4 *Over* $\mathbb{F}_{2^k}$*,* $|\mathbf{M}| = 1$.

Proof. *Since* $\mathbf{M}$ *is a Vandermonde matrix of the form* $V(\alpha, \alpha^2, \alpha^3, \ldots, \alpha^{k-1})$*, then from Eqn.(6.17)*

$$|\mathbf{M}| = \prod_{0\le i<j<k} (\alpha^{2^j} - \alpha^{2^i}) \tag{6.21}$$

Over $\mathbb{F}_{2^k}$*,* $-1 = 1$*, so Equation 6.21 is rewritten as*

$$|\mathbf{M}| = \prod_{0\le i<j<k} (\alpha^{2^j} + \alpha^{2^i}) \tag{6.22}$$

Computing $|\mathbf{M}|^2$ *gives*

$$|\mathbf{M}|^2 = [\prod_{0\le i<j<k} (\alpha^{2^j} + \alpha^{2^i})]^2 \tag{6.23}$$

Applying Lemma 6.2 to Equation 6.23 gives

$$|\mathbf{M}|^2 = \prod_{0\le i<j<k} (\alpha^{2^{j+1}} + \alpha^{2^{i+1}}) \tag{6.24}$$

When $j = k-1$, the product term is in the form $(\alpha^{2^k} + \alpha^{2^{i+1}})$. Since $\alpha^{2^k} = \alpha$ over $\mathbb{F}_{2^k}$, this term equivalent to $(\alpha^{2^{i+1}} + \alpha)$. This gives the property:

$$|\mathbf{M}|^2 = |\mathbf{M}| \tag{6.25}$$

$|\mathbf{M}| \in \mathbb{F}_{2^k}$, and only two elements of $\mathbb{F}_{2^k}$ satisfy Eqn.(6.25): 0 and 1. From Lemma 6.3, $|\mathbf{M}| \neq 0$. So $|\mathbf{M}| = 1$. ■

The proof can be further explained through the help of an example.

Example 6.6 *Over $\mathbb{F}_{2^3}$:*

$$\begin{aligned} A &= a_0 + a_1\alpha + a_2\alpha^2 \\ A^2 &= a_0 + a_1\alpha^2 + a_2\alpha^4 \\ A^4 &= a_0 + a_1\alpha^4 + a_2\alpha^8 \end{aligned} \tag{6.26}$$

From these equations, $\mathbf{M}$ is derived:

$$\mathbf{M} = \begin{bmatrix} 1 & \alpha & \alpha^2 \\ 1 & \alpha^2 & \alpha^4 \\ 1 & \alpha^4 & \alpha^8 \end{bmatrix} \tag{6.27}$$

Since $\mathbf{M}$ is a Vandermonde matrix of the form $V(\alpha, \alpha^2, \alpha^4)$, its determinant is found by applying Eqn.(6.17).

$$|\mathbf{M}| = (\alpha^4 - \alpha^2)\cdot(\alpha^4 - \alpha)\cdot(\alpha^2 - \alpha) \tag{6.28}$$

Over any $\mathbb{F}_{2^k}$, $-1 = 1$, so Equation 6.28 is rewritten as

$$|\mathbf{M}| = (\alpha^4 + \alpha^2)\cdot(\alpha^4 + \alpha)\cdot(\alpha^2 + \alpha) \tag{6.29}$$

Note that $|\mathbf{M}|$ is non-zero since it is a product of non-zero terms. Now compute $|\mathbf{M}|^{\mathbf{2}}$ while applying Lemma 6.2:

$$\begin{aligned} |\mathbf{M}|^2 &= [(\alpha^4 + \alpha^2)\cdot(\alpha^4 + \alpha)\cdot(\alpha^2 + \alpha)]^2 \qquad (6.30) \\ &= (\alpha^8 + \alpha^4)\cdot(\alpha^8 + \alpha^2)\cdot(\alpha^4 + \alpha^2) \end{aligned}$$

Over $\mathbb{F}_{2^3}$, $\alpha^8 = \alpha$, so Equation 6.31 is further simplified.

$$|\mathbf{M}|^2 = (\alpha + \alpha^4) \cdot (\alpha + \alpha^2) \cdot (\alpha^4 + \alpha^2) \tag{6.31}$$

Notice that $|\mathbf{M}|^2 = |\mathbf{M}|$. Since $|\mathbf{M}| \neq 0$, $|\mathbf{M}|$ must equal 1, as no other element of $\mathbb{F}_{2^3}$ can satisfy this condition. Indeed, evaluating Equation 6.29 and minimizing the result based on the primitive polynomial, $P(x)$, that was used to construct $\mathbb{F}_{2^3}$ will always give the result 1 regardless of which $P(x)$ is chosen.

Applying Lemma 6.4 to Eqn.(6.19) gives the equation for a_i,

$$a_i = |\mathbf{M_i}| \tag{6.32}$$

The determinant $|\mathbf{M_i}|$ can be computed symbolically as described below.

6.3.1 Symbolically Computing the Bit-Level Mapping

The refinement of the determinant $|\mathbf{M_i}|$ uses fundamental symmetric polynomials.

Definition 6.5 *For $x_1, \ldots, x_n$ and $0 \leq j \leq n$, let $S_j(x_1, \ldots, x_n)$ be the j-th* **Fundamental Symmetric Polynomial** *in $\{x_1, \ldots, x_n\}$:*

$$S_j(x_1, \ldots, x_n) = \sum_{i_1 < \cdots < i_j} x_{i_1} x_{i_2} \cdots x_{i_j} \tag{6.33}$$

Informally, S_j is the sum of all unique monomials of exactly j variables, with no variable having an exponent greater than 1.

Example 6.7 *The possible fundamental symmetric polynomials over $\{x_1, x_2, x_3\}$ are:*

$$\begin{aligned}
S_0(x_1, x_2, x_3) &= 1 \\
S_1(x_1, x_2, x_3) &= x_1 + x_2 + x_3 \\
S_2(x_1, x_2, x_3) &= x_1 x_2 + x_1 x_3 + x_2 x_3 \\
S_3(x_1, x_2, x_3) &= x_1 x_2 x_3
\end{aligned}$$

Proposition 6.4 *Let $V_i(x_1, \ldots, x_n)$, $0 \leq i \leq n$, be a square Vandermonde-like matrix derived similarly to $V(x_1, \ldots, x_n)$ but with the column $\{x_1^i, \ldots, x_n^i\}^T$ skipped and a column $\{x_1^n, \ldots, x_n^n\}^T$ appended to the end:*

$$V_i(x_1, \dots, x_n) = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{i-1} & x_1^{i+1} & \dots & x_1^n \\ 1 & x_2 & x_2^2 & \dots & x_2^{i-1} & x_2^{i+1} & \dots & x_2^n \\ \vdots & \vdots & \vdots & \cdot & \vdots & \vdots & \cdot & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^{i-1} & x_n^{i+1} & \dots & x_n^n \end{bmatrix} \tag{6.34}$$

It is known that

$$|V_i(x_1, \dots, x_n)| = |V(x_1 \dots, x_n)| \cdot S_{n-i}(x_1, \dots, x_n) \tag{6.35}$$

Computing $|\mathbf{M_i}|$ by interpolating along the $\mathbf{A}^T$ column in Eqn.(6.20) gives

$$|\mathbf{M_i}| = \sum_{j=0}^{k-1} (-1)^{(i+j)} A^{2^j} |V_{i+1}(\alpha, \dots, \alpha^{2^(j-1)}, \alpha^{2^(j+1)}, \dots, \alpha^{2^(k-1)})| \tag{6.36}$$

the final form of which is

$$\begin{aligned} |\mathbf{M_i}| \;\; = \;\; & \sum_{j=0}^{k-1} (-1)^j A^{2^j} \\ & \cdot |V(\alpha, \dots, \alpha^{2^(j-1)}, \alpha^{2^(j+1)}, \dots, \alpha^{2^(k-1)})| \\ & \cdot S_{n-1-i}(\alpha, \dots, \alpha^{2^(j-1)}, \alpha^{2^(j+1)}, \dots, \alpha^{2^(k-1)}) \end{aligned} \tag{6.37}$$

6.4 Overall Approach

The entirety of the word-level abstraction approach for a circuit with k-bit input A and k-bit output Z is summarized as follows:

1. Given a combinational circuit C, with word-level k-bit inputs A and word-level output Z.

2. Select a primitive polynomial $P(x)$ of degree k and construct $\mathbb{F}_{2^k}$.

3. Perform a reverse-topological traversal of C to find RATO, $\{x_1 > x_2 > \cdots > x_d > Z > A\}$, where $\{x_1, \dots, x_d\}$ are bit-level variables with x_i appearing earlier in traversal than x_j if $i < j$.

4. Derive the bit-level polynomials $\{f_1, \dots, f_s\}$ from C. These will be in the form $f_i : x_i + TAIL(f_i)$ where x_i is the output of a Boolean logic gate.

5. Compose the word-level polynomials which correspond bit-level and word-level input and output:

$$f_A : a_0 + a_1\alpha + \cdots + a_{k-1}\alpha^{k-1} + A \tag{6.38}$$

$$f_Z : z_0 + z_1\alpha + \cdots + z_{k-1}\alpha^{k-1} + Z \tag{6.39}$$

6. Compute the reduction $f_Z \xrightarrow{f_1,\ldots,f_s,f_A} +r$.

7. If r does not contain bit-level variables, then r is the word-level abstraction of C over $\mathbb{F}_{2^k}$. Otherwise, continue to step 8.

8. Compute $F_a = \{f_{a_0}, \ldots, f_{a_{k-1}}\}$ as

$$\begin{gathered} f_{a_0} : a_0 + |\mathbf{M_0}| \\ \vdots \\ f_{a_{k-1}} : a_{k-1} + |\mathbf{M_{k-1}}| \end{gathered} \tag{6.40}$$

   where each $|\mathbf{M_i}|$ for $0 \leq i < k$ is given by Eqn.(6.37).

9. Compute $r \xrightarrow{F_a,F_0}_+ r_w$. Then r_w is the word-level abstraction of C over $\mathbb{F}_{2^k}$.

This approach can be easily extended to circuits with multiple word-level inputs as well as circuits with varying word-sizes amongst the word-level inputs and output.

Example 6.8 *Consider, again, the buggy example shown in Example 6.3, corresponding to a buggy version of the multiplier circuit of Fig. 5.2. We already found*

$$r = (\alpha)a_1b_1 + (\alpha+1)a_1B + b_1A + Z + (\alpha+1)AB \tag{6.41}$$

Since r contains the bit-level variable a_1, find $f_{a_1} : a_1 + |\mathbf{M_1}|$. In this example, $f_A : a_0 + a_1\alpha + A$, so

$$\mathbf{M} = \begin{bmatrix} 1 & \alpha \\ 1 & \alpha^2 \end{bmatrix} \tag{6.42}$$

and

$$\mathbf{M_1} = \begin{bmatrix} 1 & A \\ 1 & A^2 \end{bmatrix} \tag{6.43}$$

Computing $|\mathbf{M_1}|$ finds

$$f_{a_1} : a_1 + A^2 + A \tag{6.44}$$

As r also contains the bit-level input b_1, the polynomial f_{b_1} is also required. Since $f_B : b_0 + b_1\alpha + B$ is isomorphic to f_A, f_{b_1} can be derived by performing the corresponding substitutions in f_{a_1}.

$$f_{b_1} : b_1 + B^2 + B \tag{6.45}$$

Now, computing $r \xrightarrow{f_{a_1}, f_{b_1}}_+ r_w$ finds

$$r_w = Z + (\alpha)A^2B^2 + A^2B + (\alpha+1)AB^2 + (\alpha+1)AB \tag{6.46}$$

which is indeed the polynomial representation of the buggy circuit.

6.5 Complexity Analysis

The worst case complexity of abstracting a combinational circuit over $\mathbb{F}_{2^k}[x_1, \dots, x_n]$ using the proposed approach is now analyzed. For simplicity, a generic circuit with a k-bit input A and a k-bit output Z is examined. No assumptions are made about the type of internal gates of the circuit; any gate can have an arbitrary number of inputs and an arbitrary representation over $\mathbb{F}_{2^k}$.

The initial step of the algorithm is to compute:

$$f_z \xrightarrow{F - \{f_z\}, F_0}_+ r \tag{6.47}$$

$$\text{where} \qquad f_z : z_0 + z_1\alpha + \cdots + z_{k-1}\alpha^{k-1} + Z \tag{6.48}$$

where $\{z_0, \dots, z_{k-1}\}$ are the bit-level outputs, F is the set of polynomials derived from the circuit, and F_0 is the set of vanishing polynomials.

Lemma 6.5 *No intermediate polynomial r_i during the reduction process $f_z \xrightarrow{F-\{f_z\}, F_0}_+ r$ will ever contain a variable with a degree larger than* 1.

Proof. Any intermediate result can contain 3 types of variables:

- Bit-level variables: Any intermediate division which results in a polynomial containing a variable $x \in \mathbb{F}_2$ with degree higher than 1 is immediately divided by $\{x^2 + x\} \in F_0$.

- Z: As reduction proceeds, the term Z is never modified since no polynomial other than f_Z contains the variable Z and f_Z is never used during the reduction process.
- A: This term is contained in $f_A : a_0 + a_1\alpha + \cdots + a_{k-1}\alpha^{k-1} + A$. Notice that $LT(f_A) = a_0 \in \mathbb{F}_2$; since an intermediate polynomial will never contain the variable a_0 with a degree higher than 1, reducing by f_A will not create a variable A with degree higher than 1.

■

Lemma 6.6 *Let $C \cdot M$ be a term where C is a coefficient in $\mathbb{F}_{2^k}$ and M is a monomial with variables in $\{x_1, \ldots, x_n\}$. Since the maximum degree of any variable of any intermediate polynomial r_i is 1, the maximum number of terms in any r_i is 2^n. Similarly, the maximum number of terms of any polynomial $\in F$ is 2^n since these polynomials are also guaranteed not to have any variables with degree greater than 1.*

The order in which polynomials in F and F_0 are used to divide r is based on RATO. Each division process divides the leading term of the intermediate polynomial.

Lemma 6.7 *Each term in an intermediate result r_i is reduced at most once.*

Proof. Assume the division is being computed by some $f \in F$. This division is computed as $\frac{LT(r_i)}{LT(f)} \cdot f + r_i$. Since $LT(\frac{LT(r_i)}{LT(f)} \cdot f) = LT(r_i)$, the leading terms are cancelled. The leading term of the resulting polynomial is strictly smaller than $LT(r_i)$ in the ordering. As every subsequent division produces a leading term strictly smaller than the last, $LT(r_i)$ never appears again. Thus, each term is divided at most once. ■

Lemma 6.8 *Since the maximum number of terms in any resulting intermediate polynomial r_i is 2^n and each term is divided at most once, the maximum number of divisions computed during the reduction is 2^n.*

Assume that a monomial multiplication and monomial addition can be computed in constant time. Each division is computed as $\frac{LT(r_i)}{LT(f)} \cdot f + r_i$. Here, f can have at most 2^n terms. $\frac{LT(r_i)}{LT(f)}$ is computed in constant time. Then, $(\frac{LT(r_i)}{LT(f)}) \cdot f$ requires at most 2^n

monomial multiplications. As the result from a monomial multiplication may contain variables with degrees higher than 1, each monomial has its variables minimized, for a maximum of 2^n minimizations. Finally, this result is added to r_i using a maximum of 2^n monomial additions. Thus each division uses at a maximum $2^n + 2^n + 2^n + 1$ monomial operations.

Lemma 6.9 *The complexity of the reduction* $f_z \xrightarrow{F-\{f_z\},J_0}_+ r$ *is* $O(2^{2n})$.

Proof. This is computed as maximum number of divisions multiplied by maximum number of monomial additions/multiplications per division.

$$2^n \cdot (2^n + 2^n + 2^n + 1) = 2^{2n} + 2^{2n} + 2^{2n} + 2^n < 4 \cdot 2^{2n} = O(2^{2n}). \tag{6.49}$$

■

The next step is to derive the bit-level to word-level mapping F_A. Without any optimizations, this can be derived by computing Gaussian elimination of a k by k matrix. The worst case arithmetic complexity here is $O(k^3)$. Furthermore, this is done in parallel with the reduction. As k is much smaller than n, the $O(2^{2n})$ reduction easily absorbs the complexity of deriving F_A, i.e. $O(2^{2n}) >> O(k^3)$.

The last step is computing the final reduction $r \xrightarrow{F_A,F_0}_+ r_w$.

Lemma 6.10 *Any intermediate result* r_j *during the final reduction* $r \xrightarrow{F_A,F_0}_+ r_w$ *will contain at most* $2^{2k} + 1$ *terms.*

Proof. Any intermediate polynomial will only contain the variables $\{a_0, \ldots, a_{k-1}, Z, A\}$. From Lemma 6.5, all variables $\{a_0, \ldots, a_{k-1}\}$ can be at most degree 1. Hence, there can be at most 2^k terms containing only variables $\{a_0, \ldots, a_{k-1}\}$. The variable A can have a degree of at most $2^k - 1$, as any higher degree is immediately divided by $\{A^{2^k} + A\} \in J_0$. Thus there can be at most 2^k terms containing only the variable A, which means that there are $2^k \cdot 2^k = 2^{2k}$ terms containing variables in $\{a_0, \ldots, a_{k-1}, A\}$. The variable Z is only found in 1 term: Z. This term is never divided and never modified since the only polynomial within the set of divisors which contain the variable Z is $\{Z^{2^k} + Z\} \in J_0$. So the maximum number of monomials in an intermediate result is $2^{2k} + 1$. ■

Lemma 6.11 *Due to Lemma 6.7, each term will be divided at most once. Z is never divided, so at most 2^{2k} divisions are computed.*

In each division, $\frac{LT(r_i)}{LT(f)} \cdot f + r_j$, f has at most $2^k + 1$ terms, since it is of the form $a_i + \mathcal{F}(A)$. Thus, there is 1 monomial division, at most $2^k + 1$ monomial multiplications. As before, the degree of each resulting monomial needs to be minimized, which performs a maximum of $2^k + 1$ minimizations. Finally, at most and $2^k + 1$ monomial additions are computed when adding this result to r_j. Thus, the total number of monomial operations per division is:

$$1 + (2^k + 1) + (2^k + 1) + (2^k + 1) = 3 \cdot 2^k + 4 \tag{6.50}$$

Lemma 6.12 *The complexity of the final substitution, which is computed as the reduction $r \xrightarrow{F_a, F_0}_+ r_w$, is $O(2^{3k})$.*

Proof. This is computed as the maximum number of divisions multiplied by the maximum number of monomial operations per division.

$$(2^{2k}) \cdot (3 \cdot 2^k + 4) = 3 \cdot 2^{3k} + \cdot 2^{2k+2} < 3 \cdot 2^{3k+2} + 2^{3k+2} = 2^{3k+5} = O(2^{3k}) \tag{6.51}$$

■

Theorem 6.3 *The worst-case complexity for the abstraction algorithm is*

$$O(2^{2n}) + O(2^{3k}) \tag{6.52}$$

The first reduction is the main bottleneck in the abstraction computation since n is much larger than k.

6.6 Conclusions

This chapter proposed an approach which solves the problem of word-level abstraction from bit-level circuits using symbolic computation. Using an improved ordering (RATO) a reduction is computed to derive a polynomial, r, which contains only bit-level inputs $\{a_0, \ldots, a_{k-1}\}$ and the word-level datapaths A and Z. This reduction is efficiently computed using an $F4$-style reduction engine. Next, an equation of the form $a_i =$

$\mathcal{F}_{a_i}(A)$, which maps each bit-level input to its corresponding word-level representation, is computed using a binomial expansion over $\mathbb{F}_{2^k}$. Substituting variable each a_i in r by $\mathcal{F}_{a_i}(A)$ gives r_w, which is the *canonical, word-level, polynomial representation of the circuit*. Finally, a complexity analysis of the overall approach is provided.

The abstraction approach is directly applicable only to circuits with equivalent datapath sizes amongst the inputs and output, i.e. every word input and output is of size k. The next chapter generalizes the abstraction approach to be applicable to any arbitrary combinational circuit.

CHAPTER 7

GENERALIZING THE APPROACH TO ARBITRARY COMBINATIONAL CIRCUITS

The abstraction approach presented in the previous chapter is directly applicable only when the word-size of the operands of the given circuit are the same, k bits in size. In this case, the circuit computes a function over $\mathbb{F}_{2^k}^n \to \mathbb{F}_{2^k}$, where n is the number of word-level inputs, and the abstraction is thus analyzed over $\mathbb{F}_{2^k}$. When the input and output sizes vary, the analysis must be performed over an overarching field. This chapter shows how to suitably modify the approach for abstracting word-level representations of circuits with varying input and output sizes.

As an application of this generalization, design and verification methodologies for composite field multipliers over $\mathbb{F}_{2^k}$ are also explored. These designs decompose the field $\mathbb{F}_{2^k}$ to $\mathbb{F}_{(2^m)^n}$, where $k = m \cdot n$. Internally, the circuit is then composed of an n-interconnection of m-bit multipliers and adders over $\mathbb{F}_{2^m}$. This chapter describes how this hierarchy can be exploited by the abstraction approach.

7.1 Circuits with Varying Input and Output Sizes

When the word size of the inputs and output of the circuits vary, the functionality of the circuit must be analyzed over an encompassing composite field. Given a circuit with a word-level input A and word-level output Z, let t be the bit size of the A and u be the bit size of Z. Input A can be represented as an element over the field $\mathbb{F}_{2^t}$; likewise, Z is an element of $\mathbb{F}_{2^u}$. Thus, this circuit computes some function $f : \mathbb{F}_{2^t} \to \mathbb{F}_{2^u}$. If $t \neq u$, there is no guarantee that A can be represented over $\mathbb{F}_{2^u}$ and or conversely Z over $\mathbb{F}_{2^t}$. Thus, the analysis must be performed over some $\mathbb{F}_{2^k}$ such that $\mathbb{F}_{2^t} \subset \mathbb{F}_{2^k}$ and $\mathbb{F}_{2^u} \subset \mathbb{F}_{2^k}$. That is, the function is mapped to a larger field $\mathbb{F}_{2^k} \to \mathbb{F}_{2^k}$. The smallest such field $\mathbb{F}_{2^k}$ is constructed where $k = LCM(t, u)$.

Given the primitive polynomial $P(x)$ of degree k, let α be the primitive element of $\mathbb{F}_{2^k}$, i.e. $P(\alpha) = 0$. Let β be some primitive element of $\mathbb{F}_{2^t}$. Then the polynomial f_A is denoted

$$f_A : a_0 + a_1\beta + \cdots + a_{t-1}\beta^{t-1} + A \tag{7.1}$$

Since the analysis is over $\mathbb{F}_{2^k}$, β must be mapped directly to α. Here, a generalized result from [106] gives the relation.

$$\beta = \alpha^{(2^k-1)/(2^t-1)} \tag{7.2}$$

All powers of β in f_A are replaced by their representation in α. Similarly, let γ be the primitive element of $\mathbb{F}_{2^u}$, such that

$$f_Z : z_0 + z_1\gamma + \cdots + z_{u-1}\gamma^{u-1} + Z \tag{7.3}$$

A mapping from γ to α is found in the same way and applied to f_Z. All polynomials in the ideal J now have coefficients in $\mathbb{F}_{2^k}$. The polynomials in J_0 derived from A and from Z are modified to reflect their corresponding fields:

$$A^{2^t} + A = 0 \qquad Z^{2^u} + Z = 0 \tag{7.4}$$

Now the reduction procedure $f_Z \xrightarrow{F-f_z,F_0}_+ r$ is computed normally over $\mathbb{F}_{2^k}$.

Some changes still need to be made to functionally map each $\{a_0, \ldots, a_{t-1}\}$ to A over $\mathbb{F}_{2^k}$. This is accomplished by first computing the bit-level to word-level mapping over $\mathbb{F}_{2^t}$ using the same approach described in Chapter 6, which will provide the polynomials $f_{a_0}, \ldots, f_{a_t}$ with coefficients in β. Then, map each β coefficient in $F_a = \{f_{a_0}, \ldots, f_{a_t}\}$ to α using Eqn.(7.2). Polynomials of F_a are now in $\mathbb{F}_{2^k}$ and the final substitution $r \xrightarrow{F_a+J_0}_+ r_w$ is computed as normal.

This allows the word-level abstraction of any combinational circuit with one word-level input of any size and one word-level output. In the case of multiple word-level inputs, let k be the LCM of all the bit-sizes of the inputs and outputs. Then the abstraction proceeds as normal.

Example 7.1 *Consider the circuit shown in Fig. 7.1. The input, A, is 3 bits wide while the output, Z is 2 bits. Thus, $A \in \mathbb{F}_{2^3}$ and $Z \in \mathbb{F}_{2^2}$. Let β be the primitive element of* $\mathbb{F}_{2^3}$

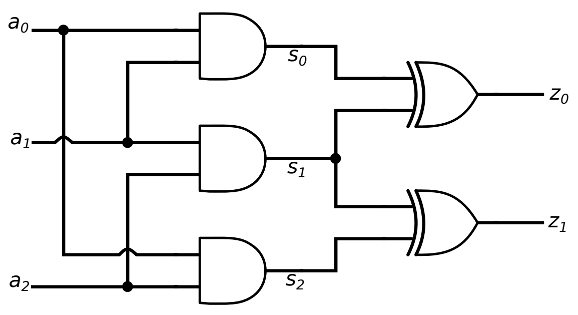


Figure 7.1: Circuit with varying word sizes.

and γ be the primitive element of $\mathbb{F}_{2^2}$, i.e. $A = a_0 + a_1\beta + a_2\beta^2$ and $Z = z_0 + z_1\gamma$. Thus, the circuit computes the function $\mathbb{F}_{2^3} \rightarrow \mathbb{F}_{2^2}$. Since $LCM(2,3) = 6$, then $\mathbb{F}_{2^2} \subset \mathbb{F}_{2^6}$ and $\mathbb{F}_{2^3} \subset \mathbb{F}_{2^6}$. So this function mapped to $\mathbb{F}_{2^6} \rightarrow \mathbb{F}_{2^6}$. Choose $P(X) = X^6 + X + 1$ as the irreducible polynomial to construct $\mathbb{F}_{2^6}$, where $P(\alpha) = 0$. Then β and γ in can be represented in terms of α:

$$\begin{aligned} \beta &= \alpha^{(2^6-1)/(2^3-1)} = \alpha^9 \\ \gamma &= \alpha^{(2^6-1)/(2^2-1)} = \alpha^{21} \end{aligned} \tag{7.5}$$

So the word-level polynomials are:

$$\begin{aligned} &f_A : a_0 + a_1\alpha^9 + a_2\alpha^{18} + A \\ &f_Z : z_0 + z_1\alpha^{21} + Z \end{aligned} \tag{7.6}$$

The rest of the polynomials in F are derived from the circuit:

$$\begin{aligned} &f_1 : z_0 + s_0 + s_1 \quad f_2 : z_1 + s_1 + s_2 \quad f_3 : s_0 + a_0 \cdot a_1 \\ &f_4 : s_1 + a_1 \cdot a_2 \quad f_5 : s_2 + a_0 \cdot a_2 \end{aligned} \tag{7.7}$$

where the RATO ordering is:

$$z_0 > z_1 > s_0 > s_1 > s_2 > a_0 > a_1 > a_2 > Z > A \tag{7.8}$$

F_0 is defined as before, except with a change to the polynomials derived from word-level variables A and Z:

$$\begin{aligned} f_6 &: z_0^2 + z_0 \quad f_7 : z_1^2 + z_1 \quad f_8 : s_0^2 + s_0 \\ f_9 &: s_1^2 + s_1 \quad f_{10} : s_2^2 + s_2 \quad f_{11} : a_0^2 + a_0 \\ f_{12} &: a_1^2 + a_2 \quad f_{13} : a_2^2 + a_2 \quad f_{14} : Z^4 + Z \\ f_{15} &: A^8 + A \end{aligned} \tag{7.9}$$

Computing $f_Z \xrightarrow{F-f_z,F_0}_+ r$ gives:

$$\begin{aligned} r \; = \; & (\alpha^2+\alpha)\cdot a_1 \cdot a_2 + a_1 \cdot A + (\alpha^4+\alpha^3)\cdot a_1 \\ & +(\alpha^5+\alpha^4+\alpha^3+\alpha+1)\cdot a_2 \cdot A + (\alpha^5+\alpha^4+\alpha^2+\alpha)\cdot a_2 + Z \end{aligned} \tag{7.10}$$

This result must be further reduced by $f_{a_1} : a_1 + \mathcal{F}_{a_1}(A)$ and $f_{a_2} : a_2 + \mathcal{F} - a_2(A)$. First, find $\mathbf{M}$ in terms of β as derived from $A = a_0 + a_1\beta + a_2\beta^2$. Then map it to α.

$$\mathbf{M} = \begin{bmatrix} 1 & \beta & \beta^2 \\ 1 & \beta^2 & \beta^4 \\ 1 & \beta^4 & \beta^8 \end{bmatrix} \quad \mathbf{M_1} = \begin{bmatrix} 1 & A & \beta^2 \\ 1 & A^2 & \beta^4 \\ 1 & A^4 & \beta^8 \end{bmatrix} \quad \mathbf{M_2} = \begin{bmatrix} 1 & \beta & A \\ 1 & \beta^2 & A^2 \\ 1 & \beta^4 & A^4 \end{bmatrix} \tag{7.11}$$

Computing the determinants in $f_{a_1} : a_1 + |\mathbf{M_1}|$ and $f_{a_2} : a_2 + |\mathbf{M_2}|$ gives:

$$\begin{aligned} f_{a_1} &: a_1 + A^4 \cdot (\beta^4+\beta^2) + A^2 \cdot (\beta^8+\beta^2) + A \cdot (\beta^8+\beta^4) \\ f_{a_2} &: a_2 + A^4 \cdot (\beta^2+\beta) + A^2 \cdot (\beta^4+\beta) + A \cdot (\beta^4+\beta^2) \end{aligned} \tag{7.12}$$

Replacing all β in f_{a_1} and f_{a_2} with α^9 gives their proper form over $\mathbb{F}_{2^6}$.

$$\begin{aligned} f_{a_1} &: a_1 + A^4 \cdot (\alpha^4+\alpha^3+1) + A^2 \cdot (\alpha^4+\alpha^2+\alpha+1) + A \cdot (\alpha^3+\alpha^2+\alpha) \\ f_{a_2} &: a_2 + A^4 \cdot (\alpha^4+\alpha^2+\alpha+1) + A^2 \cdot (\alpha^3+\alpha^2+\alpha) + A \cdot (\alpha^4+\alpha^3+1) \end{aligned} \tag{7.13}$$

Finally, computing the reduction $r \xrightarrow{f_{a_1},f_{a_2},F_0}_+ r_w$ gives the word-level polynomial abstraction of the circuit.

$$\begin{aligned} r_w : \quad & Z + A^6(\alpha^2+\alpha) + A^5(\alpha^4+\alpha^3+\alpha) + A^4(\alpha^2+\alpha) \\ & +A^3(\alpha^4+\alpha^3+\alpha^2) + A^2(\alpha^4+\alpha^3+\alpha^2) + A(\alpha^4+\alpha^3+\alpha) \end{aligned}$$

This result allows the abstraction to be computed over fields of different sizes. An important application of this result is that of modularly-designed composite field arithmetic circuits. These circuits compute operations over very large fields (i.e. $k = 1024$) by combining operations over smaller sub-fields (i.e. $k = 32$). The abstraction approach can be efficiently applied to these types of circuits by exploiting the hierarchy found in these designs.

7.2 Composite Field Arithmetic Circuits

A Galois field multiplier over $\mathbb{F}_{2^k}$ can be composed over the composite field $\mathbb{F}_{(2^m)^n}$ where $k = m \cdot n$ [107]. Similarly to how $\mathbb{F}_{2^k}$ is a k-dimensional extension of the subfield $\mathbb{F}_2$, $\mathbb{F}_{(2^m)^n}$ is also an n-dimensional extension of $\mathbb{F}_{2^m}$. A **composite field multiplier** lifts the ground field from $\mathbb{F}_2$ to $\mathbb{F}_{2^m}$ and computes the multiplication over $\mathbb{F}_{2^k}$ as a collection of operations over $\mathbb{F}_{2^m}$. Thus, a composite field multiplier over $\mathbb{F}_{2^k}$ is composed internally as a collection of *multipliers* and *adders* over $\mathbb{F}_{2^m}$.

This hierarchy found in composite field multipliers can be exploited by the proposed abstraction approach. Abstraction of these types of multipliers is composed of two steps:

1. Compute the canonical word-level polynomial representation of each $\mathbb{F}_{2^m}$ multiplier and adder.
    - These abstractions are independent of one another. Thus, they are computed in parallel.
    - In the case of an adder, the abstraction is trivial.
2. Compute the overall abstraction of the $\mathbb{F}_{2^k}$ multiplier.

The first step utilizes the proposed abstraction approach over $\mathbb{F}_{2^m}$ with no changes. Once these word-level abstractions are known, they replace the gate-level implementations for the final word-level abstraction of the multiplier over $\mathbb{F}_{2^k}$.

7.2.1 Design of Composite Field Multipliers

The following adapts principles of composite fields from [107] and explains how they are applied to construct Galois field multipliers. Consider the element $A \in \mathbb{F}_{2^k}$ and its representation over $\mathbb{F}_{(2^m)^n}$. Let α be the primitive element of $\mathbb{F}_{2^k}$ and let γ be the primitive element of $\mathbb{F}_{(2^m)^n}$. Then any element $A \in \mathbb{F}_{2^k}$ is represented as:

$$A = a_0 + a_1\alpha + \cdots + a_{k-1}\alpha^{k-1}, \text{where } a_i \in \mathbb{F}_2 \tag{7.14}$$

This same element $A \in \mathbb{F}_{(2^m)^n}$ is represented as:

$$A = A_0 + A_1\gamma + \cdots + A_{n-1}\gamma^{n-1}, \text{where } A_i \in \mathbb{F}_{2^m} \tag{7.15}$$

Let β be the primitive element of $\mathbb{F}_{2^m}$. Then each A_i is represented as

$$A_i = a_{i0} + a_{i1}\beta + \cdots + a_{i\{m-1\}}\beta^{m-1}, \text{where } a_{ij} \in \mathbb{F}_2 \tag{7.16}$$

Since there always exists a unique field with p^k elements, the field $\mathbb{F}_{2^k}$ is isomorphic to the field $\mathbb{F}_{(2^m)^n}$. Due to this, $\gamma = \alpha$. Furthermore, β can be derived from α using Eqn.(7.2) since $\mathbb{F}_{2^m} \subset \mathbb{F}_{2^k}$, i.e. $\beta = \alpha^w$ for some w. Thus, all that is required to construct a composite field multiplier over $\mathbb{F}_{(2^m)^n}$ is the primitive polynomial $P(x)$ which generates $\mathbb{F}_{2^k}$, with $P(\alpha) = 0$, where β is known.

In order to lift the ground field $\mathbb{F}_2$ to $\mathbb{F}_{2^m}$, the variables $\{a_{00}, \ldots, a_{\{n-1\}\{m-1\}}\}$ must be derived in terms of $\{a_0, \ldots, a_{k-1}\}$. Equating the representation of $A \in \mathbb{F}_{2^k}$ with the representation of $A \in \mathbb{F}_{(2^m)^n}$ from Eqns.(7.14) and (7.15) gives the following:

$$\begin{aligned} & a_0 + a_1\alpha + \cdots + a_{k-1}\alpha^{k-1} \\ = \; & A_0 + A_1\alpha + \cdots + A_{n-1}\alpha^{n-1} \\ = \; & \sum_{i=0}^{i=n-1} \Big(\sum_{j=0}^{j=m-1} a_{ij} \cdot \alpha^{wj} \Big) \cdot \alpha^i \end{aligned} \tag{7.17}$$

Here, every $A_0, \ldots, A_{n-1}$ is replaced by its representation over $\mathbb{F}_{2^m}$. Analyzing the coefficients gives $\{a_0, \ldots, a_{k-1}\}$ in terms of $\{a_{00}, \ldots, a_{\{n-1\}\{m-1\}}\}$. This mapping can be depicted as a matrix multiplication.

$$\begin{bmatrix} a_0 \\ \vdots \\ a_{k-1} \end{bmatrix} = \mathbf{T} \begin{bmatrix} a_{00} \\ \vdots \\ a_{\{n-1\}\{m-1\}} \end{bmatrix} \tag{7.18}$$

where $\mathbf{T}$ is a k by k matrix consisting of elements in $\mathbb{F}_2$. Inverting $\mathbf{T}$ gives a mapping from $\{a_{00}, \ldots, a_{\{n-1\}\{m-1\}}\}$ to $\{a_0, \ldots, a_{k-1}\}$.

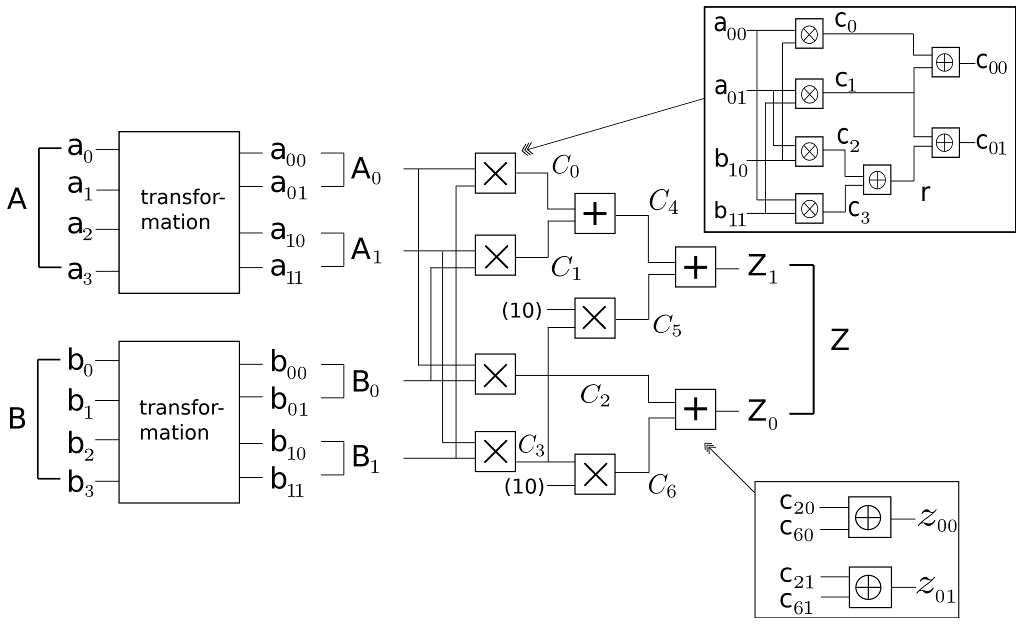


Figure 7.2: 4-bit composite multiplier designed over $\mathbb{F}_{(2^2)^2}$

Example 7.2 *An example composite field multiplier $\mathbb{F}_{(2^2)^2}$, which computes a multiplication over $\mathbb{F}_{2^4}$, is shown in Figure 7.2. Notice that, after the transformation, all additions and multiplications are computed over the base field $\mathbb{F}_{2^2}$.*

Let $P(x) = x^4 + x^3 + 1$ and $P(\alpha) = 0$. Representation of element $A \in \mathbb{F}_{(2^2)^2}$ is:

$$A \quad = A_0 + A_1 \cdot \alpha \tag{7.19}$$

Representation of A_0, A_1 in $\mathbb{F}_{2^m}$ is:

$$\begin{aligned} A_0 &= a_{00} + a_{01} \cdot \beta \\ A_1 &= a_{10} + a_{11} \cdot \beta \end{aligned} \tag{7.20}$$

where $a_{ij} \in \mathbb{F}_2$. From Eqn.(7.2), $\beta = \alpha^5$ Now A_0, A_1 can be substituted into A as follows:

$$A \quad = \quad a_{00} + a_{01} \cdot \alpha^5 + (a_{10} + a_{11} \cdot \alpha^5) \cdot \alpha \tag{7.21}$$

Since $P(x) = x^4 + x^3 + 1$ with $P(\alpha) = 0$,

$$A \pmod{P(\alpha)} = a_{00}+a_{01}+a_{11}+(a_{01}+a_{10}+a_{11})\cdot\alpha+a_{11}\cdot\alpha^2+(a_{01}+a_{11})\cdot\alpha^3 \tag{7.22}$$

The same element $A \in \mathbb{F}_{2^4}$ is represented as:

$$A = a_0 + a_1 \cdot \alpha + a_2 \cdot \alpha^2 + a_3 \cdot \alpha^3 \tag{7.23}$$

Since Eqns. (7.22) and (7.23) represent the same element, we can match the coefficients of the the polynomials to obtain:

$$\begin{aligned} a_0 &= a_{00} + a_{01} + a_{11} \\ a_1 &= a_{01} + a_{10} + a_{11} \\ a_2 &= a_{11} \\ a_3 &= a_{01} + a_{11} \end{aligned}$$

This mapping can also be reversed and represented as a matrix T^{-1}:

$$\begin{bmatrix} a_{00} \\ a_{01} \\ a_{10} \\ a_{11} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{7.24}$$

Thus, A is represented in $\mathbb{F}_{(2^2)^2}$ as:

$$\begin{aligned} A &= A_0 + A_1 \cdot \alpha \\ A_0 &= a_{00} + a_{01} \cdot \alpha^5 \\ A_1 &= a_{10} + a_{11} \cdot \alpha^5 \\ a_{00} &= a_0 + a_3 \\ a_{01} &= a_2 + a_3 \\ a_{10} &= a_1 + a_3 \\ a_{11} &= a_2 \end{aligned}$$

B is similarly represented in $\mathbb{F}_{(2^2)^2}$.

7.2.2 Abstraction of Composite Field Multipliers

The first step in abstracting the word-level polynomial representation of the composite field multiplier is to abstract every internal $\mathbb{F}_{2^m}$ computational block. In the case of a $\mathbb{F}_{2^m}$ adder block ($\{Z_0, Z_1, C_4\}$ from Fig.(7.2)), this abstraction is trivial, as the adder is just a bit-wise XOR computation. In the case of a multiplier block ($\{C_0, C_1, C_2, C_3, C_5, C_6\}$ from Fig.(7.2)), the abstraction approach presented in the previous chapter is directly applicable. Each abstraction can be computed independently, so these computations can be performed in parallel. This set of abstracted polynomials, F, generates the ideal J.

Once the word-level abstraction of each $\mathbb{F}_{2^m}$ sub-block is known, the final abstraction of the entire design can be computed using only word-level variables. Set a RATO ordering. Then reduce the word level polynomial by $F + F_0$:

$$f_Z : Z_0 + Z_1\alpha + \cdots + Z_{n-1}\alpha^{n-1} + Z \tag{7.25}$$

$$f_Z \xrightarrow{F+F_0}_+ r \tag{7.26}$$

Here, F is the set of word-level polynomial abstractions from each $\mathbb{F}_{2^m}$ block and F_0 is the set of vanishing polynomials. Every variable X, apart from the word-level inputs A and B and word-level output Z, is an element of $\mathbb{F}_{2^m}$, so its corresponding vanishing polynomial is $X^{2^m} + X$. Computing the reduction gives remainder r containing the elements

$$A_0, A_1, \ldots, A_{n-1}, B_0, B_1, \ldots, B_{n-1}, Z \tag{7.27}$$

Similarly, the mapping from $\{A_0, \ldots, A_{n-1}\}$ to A now needs to be derived (along with the mapping from $\{B_0, \ldots, B_{n-1}\}$ to B). That is, the next step is to find

$$\begin{aligned} & F_A = \{F_{A_0}, F_{A_1}, \ldots, F_{A_{n-1}}\} \\ \text{where} \qquad & F_{A_i} = A_i + \mathcal{F}_{A_i}(A) \end{aligned} \tag{7.28}$$

F_A is derived from the polynomial

$$A = A_0 + A_1\alpha + \cdots + A_{n-1}\alpha^{n-1} \tag{7.29}$$

Compute A^{2^m} as follows:

$$\begin{aligned} A^{2^m} : \quad & (A_0 + A_1\alpha + \cdots + A_{n-1}\alpha^{n-1})^{2^m} \\ & = A_0^{2^m} + A_1^{2^m}\alpha^{2^m} + \cdots + A_{n-1}^{2^m}\alpha^{2^m(n-1)} \\ & = A_0 + A_1\alpha^{2^m} + \cdots + A_{n-1}\alpha^{2^m(n-1)} \end{aligned} \tag{7.30}$$

Here, the property is $A_i^{2^m} = A_i$ is exploited. Continually raise this result by 2^m to obtain $A^{2^{jm}}$ for all $0 \le j < n$. This gives a system of n equations and n unknowns

$\{A_0, \ldots, A_{n-1}\}$. As before, this system of equations can be represented in matrix form, $\mathbf{A} = \mathbf{Ma}$, where $\mathbf{A} = \{A, A^{2^m}, \ldots, A^{2^{(n-1)m}}\}^T$, $\mathbf{M}$ is a n by n matrix of coefficients $\in \mathbb{F}_{2^k}$, and $\mathbf{a} = \{A_0, \ldots, A_{n-1}\}^T$:

$$\begin{bmatrix} A \\ A^{2^m} \\ \vdots \\ A^{2^{(n-1)m}} \end{bmatrix} = \begin{bmatrix} 1 & \alpha & \alpha^2 & \ldots & \alpha^{n-1} \\ 1 & \alpha^{2^m} & \alpha^{2 \cdot 2^m} & \ldots & \alpha^{(n-1)\cdot 2^m} \\ 1 & \alpha^{2^{2m}} & \alpha^{2 \cdot 2^{2m}} & \ldots & \alpha^{(n-1)\cdot 2^{2m}} \\ \vdots & \vdots & \vdots & \cdot & \vdots \\ 1 & \alpha^{2^{(n-1)m}} & \alpha^{2 \cdot 2^{(n-1)m}} & \ldots & \alpha^{(n-1)\cdot 2^{(n-1)m}} \end{bmatrix} \begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{n-1} \end{bmatrix} \tag{7.31}$$

The matrix $\mathbf{M}$ is the Vandermonde matrix $V(\alpha, \alpha^{2^m}, \ldots, \alpha^{2^{(n-1)m}})$. Since α is a primitive element, all elements $\alpha, \ldots, \alpha^{2^{k-1}}$ are unique, where $k = m \cdot n$. Thus, all elements $\alpha, \alpha^{2^m}, \ldots, \alpha^{2^{(n-1)m}}$ are unique, so $|\mathbf{M}| \neq 0$. Then, Cramer's rule can be applied to derive each F_{A_i} as

$$F_{A_i} = A_i + \frac{|\mathbf{M_i}|}{|\mathbf{M}|} \tag{7.32}$$

where $\mathbf{M_i}$ is $\mathbf{M}$ with the i-th column replaced by $\mathbf{A}$. Here, $|\mathbf{M}|$ is not guaranteed to be equal to 1, so it must be computed. $F_B = \{F_{B_0}, \ldots, F_{B_{n-1}}\}$ is similarly derived. Computing $r \xrightarrow{F_A, F_B, J_0}_+ r_w$ gives $r_w : Z + \mathcal{F}(A, B)$ which is the canonical word-level polynomial abstraction of the composite field design.

Example 7.3 *Consider the composite field multiplier over $\mathbb{F}_{2^{2^2}}$ from Example 7.2. Abstracting every multiplier and adder over the base field $\mathbb{F}_{2^2}$ gives the following polynomials:*

$$\begin{aligned} f_1 &: Z_0 + C_6 + C_2 & f_2 &: Z_1 + C_5 + C_4 & f_3 &: C_6 + \alpha^5 C_3 \\ f_4 &: C_5 + \alpha^5 C_3 & f_5 &: C_4 + C_1 + C_0 & f_6 &: C_3 + A_1 \cdot B_1 \\ f_7 &: C_2 + A_0 \cdot B_0 & f_8 &: C_1 + A_1 \cdot B_0 & f_9 &: C_0 + A_0 \cdot B_1 \end{aligned} \tag{7.33}$$

Set the following RATO ordering:

$$\begin{aligned} Z_0 > Z_1 > C_6 > C_5 > C_4 > C_3 > C_2 > C_1 > C_0 \\ > A_0 > A_1 > B_0 > B_1 > Z > A > B \end{aligned} \tag{7.34}$$

The vanishing polynomials are:

$$\begin{array}{lll} f_{10}: Z_0^4 + Z_0 & f_{11}: Z_1^4 + Z_1 & f_{12}: C_6^4 + C_6 \\ f_{13}: C_5^4 + C_5 & f_{14}: C_4^4 + C_4 & f_{15}: C_3^4 + C_3 \\ f_{16}: C_2^4 + C_2 & f_{17}: C_1^4 + C_1 & f_{18}: C_0^4 + C_0 \\ f_{19}: A_0^4 + A_0 & f_{20}: A_1^4 + A_1 & f_{21}: B_0^4 + B_0 \\ f_{22}: B_1^4 + B_1 & f_{23}: Z^{16} + Z & f_{24}: A^{16} + A \\ f_{25}: B^{16} + B & & \end{array} \tag{7.35}$$

Then $F = f_1, \ldots, f_9$ *and* $F_0 = f_{10}, \ldots, f_{25}$*. Here,* $f_Z : Z_0 + Z_1\alpha + Z$*. Computing* $f_Z \xrightarrow{F,F_0}_+ r$ *gives:*

$$r = A_0 \cdot B_0 + \alpha A_0 \cdot B_1 + \alpha A_1 \cdot B_0 + \alpha^2 A_1 \cdot B_1 + Z \tag{7.36}$$

Next, $F_A = \{F_{A_0}, F_{A_1}\}$ *needs to be derived. Here,* $A = A_0 + A_1\alpha$ *and* $A^4 = A_0 + A_1\alpha^4$*, which gives*

$$\mathbf{M} = \begin{bmatrix} 1 & \alpha \\ 1 & \alpha^4 \end{bmatrix}; \quad \mathbf{M_0} = \begin{bmatrix} A & \alpha \\ A^4 & \alpha^4 \end{bmatrix}; \quad \mathbf{M_1} = \begin{bmatrix} 1 & A \\ 1 & A^4 \end{bmatrix} \tag{7.37}$$

After minimizing by the primitive polynomial $P(x) = x^4 + x^3 + 1$*, the determinants of these matrices are:*

$$|\mathbf{M}| = \alpha^3 + \alpha + 1; \quad |\mathbf{M_0}| = \alpha A^4 + (\alpha^3 + 1)A; \quad |\mathbf{M_1}| = A^4 + A \tag{7.38}$$

Now F_{A_0} *and* F_{A_1} *are derived:*

$$F_{A_0} = \quad A_0 + \tfrac{|\mathbf{M_0}|}{|\mathbf{M}|} \quad = A_0 + (\alpha^3 + \alpha^2 + 1)A^4 + (\alpha^3 + \alpha^2)A \tag{7.39}$$

$$F_{A_1} = \quad A_1 + \tfrac{|\mathbf{M_1}|}{|\mathbf{M}|} \quad = A_1 + (\alpha^3 + \alpha)A^4 + (\alpha^3 + \alpha)A \tag{7.40}$$

Since $F_B = \{F_{B_0}, F_{B_1}\}$ *is derived from* $B = B_0 + B_1\alpha$*, which is isomorphic to* $A = A_0 + A_1\alpha$*, then* F_B *can be derived from* F_A *by substituting corresponding variables.*

$$F_{B_0} = B_0 + (\alpha^3 + \alpha^2 + 1)B^4 + (\alpha^3 + \alpha^2)B \tag{7.41}$$

$$F_{B_1} = B_1 + (\alpha^3 + \alpha)B^4 + (\alpha^3 + \alpha)B \tag{7.42}$$

Finally, computing $r \xrightarrow{F_A, F_B, F_0}_+ r_w$ *gives the word-level abstraction of the circuit.*

$$r_w : Z + A \cdot B \tag{7.43}$$

7.3 Conclusion

This chapter described how to generalize the abstraction approach to any arbitrary combinational circuit. This method works best when the derived operand-width k, from the LCM of the word-lengths of all inputs and output, is not very large, for instance, when the output size is a multiple of the input sizes. When all word-sizes are relatively prime, k is a product of all sizes, which can make the analysis overly bulky.

An abstraction technique for exploiting the hierarchy of composite field multiplier circuits over $\mathbb{F}_{(2^m)^n}$ was also examined. The approach abstracts all internal multipliers and adders over $\mathbb{F}_{2^m}$ in parallel and uses these abstractions to compute the final abstraction completely over word-level variables. Experimental results for abstractions of composite field multipliers using a custom-built tool is presented in the next chapter.

CHAPTER 8

IMPLEMENTATION OF THE CUSTOM ABSTRACTION SOFTWARE AND EXPERIMENTAL RESULTS

The abstraction procedure can be fully scripted using the computer algebra tool SINGULAR. However, SINGULAR has limitations which make abstraction of large circuits *impossible*. This is due to:

- A limit on the number of ring variables allowed. As of SINGULAR release 4.0.1, a ring cannot be declared with more than $32,767$ variables. This limits the number of gates that can be present in the design.
- The size of an exponent (n) of a variable x, is limited $n < 2^{32}$.
- Large amount of memory usage and slow computation time. SINGULAR uses a dense-distributive structure for polynomials, which is a poor representation of sparse polynomials over rings with many variables.

The limit on the size of exponents prohibits an abstraction beyond 32-bit circuits, as larger circuits require manipulating word-level variables with exponents larger than 2^{32}. Even if this limitation is overcome, Singular uses an enormous amount of memory. For instance, preliminary experiments show that using Singular to compute the initial reduction of a 163-bit Mastrovito multiplier uses 41.6 GB of memory! Thus, deriving abstractions using Singular is infeasible on desktop workstations.

In order to overcome these limitations, a custom C++ tool is developed to compute the word-level abstraction of circuits quickly and efficiently. This chapter describes the implementation details of this tool.

8.1 Data Structures and Algorithms

Computing a word-level abstraction requires the representation and manipulation of polynomials in $\mathbb{F}_{2^k}[x_1, \ldots, x_d]$ over a lex ordered ring. Intermediate polynomials can become very large during the abstraction procedure, so the data structure to represent them is designed to conserve memory while allowing for fast manipulation during the reduction procedures. The backbone of the tool is a custom library composed of three main sections:

1. Galois field elements
2. Monomials and rings
3. Polynomials and division procedures

These are built on top of each other. The starting point is the custom Galois field element section, which facilitates the construction of Galois field elements over $\mathbb{F}_{2^k}$.

8.1.1 Galois Field Elements

The Galois field section of the library is initialized by parsing a given primitive polynomial $P(x)$ of degree k which constructs $\mathbb{F}_{2^k}$. Any element $C \in \mathbb{F}_{2^k}$ can be represented in the form

$$C = c_{k-1} \cdot \alpha^{k-1} + c_{k-2} \cdot \alpha^{k-2} + \cdots + c_2 \cdot \alpha^2 + c_1 \cdot \alpha + c_0 \tag{8.1}$$

where $\{c_0, \ldots, c_{k-1}\} \in \mathbb{F}_2$ and α is the primitive element. This *structure* is stored as an unsigned byte array containing $\{c_0, \ldots, c_{k-1}\}$, as shown in Figure 8.1. Thus, each element uses $\left\lfloor \frac{k-1}{8} \right\rfloor + 1$ bytes of memory. Any leading bits after c_{k-1} in the last byte are set to 0.

Addition between two elements

$$C = c_{k-1} \cdot \alpha^{k-1} + \cdots + c_2 \cdot \alpha^2 + c_1 \cdot \alpha + c_0 \tag{8.2}$$

$$D = d_{k-1} \cdot \alpha^{k-1} + \cdots + d_2 \cdot \alpha^2 + d_1 \cdot \alpha + d_0 \tag{8.3}$$

is simply a combination of like terms

$$C + D = (c_{k-1} + d_{k-1}) \cdot \alpha^{k-1} + \cdots + (c_2 + d_2) \cdot \alpha^2 + (c_1 + d_1) \cdot \alpha + (c_0 + d_0) \tag{8.4}$$

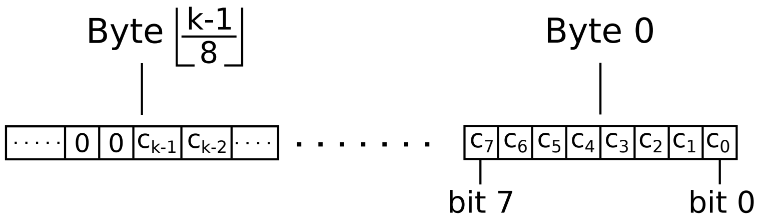


Figure 8.1: Object structure of a Galois field element

Since addition over $\mathbb{F}_2$ is computed as a bit-wise XOR, the library's Galois field element structure allows addition to be trivially performed as a byte-wise XOR operation. Furthermore, this structure makes it easy to check if a given element is equal to 0 or 1, which is used in deciding when a term is to be removed (0) and when a division can be ignored (1).

During *library initialization*, the elements $\alpha^k, \alpha^{k+1}, \ldots, \alpha^{2k-2}$ are pre-computed and cached. First, α^k is derived directly from the given primitive polynomial,

$$P(x) = x^k + c_{k-1} \cdot x^{k-1} + \cdots + c_1 \cdot x + 1 \tag{8.5}$$

for $\{c_1, \ldots, c_{k-1}\} \in \mathbb{F}_2$. Since $P(\alpha) = 0$,

$$\alpha^k = c_{k-1} \cdot \alpha^{k-1} + \cdots + c_1 \cdot \alpha + 1 \tag{8.6}$$

To compute α^{k+1}, first compute a 1-bit left shift of α^k.

$$\alpha^{k+1} = c_{k-1} \cdot \alpha^k + c_{k-2} \cdot \alpha^{k-1} + \cdots + c_1 \cdot \alpha^2 + \alpha \tag{8.7}$$

This element can contain the term α^k, which must be minimized by the primitive polynomial. Thus, if c_{k-1} is 1, the $c_{k-1}\alpha^k$ term is removed and the minimized form of α^k is added. This derives the minimized form for α^{k+1}. Computation continues in this fashion (shift by 1, minimize if needed) until all $\alpha^k, \alpha^{k+1}, \ldots, \alpha^{2k-2}$ have been derived. These elements are later used during the multiplication procedure.

Example 8.1 *Given the primitive polynomial $P(x) = x^4 + x^3 + 1$, initialize the library by computing α^4, α^5, and α^6. Here, $k = 4$, and each element of $\mathbb{F}_{2^4}$ can be represented as*

$$c_3 \cdot \alpha^3 + c_2 \cdot \alpha^2 + c_1 \cdot \alpha + c_0 \tag{8.8}$$

which is stored as one byte in the following form.

$$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 0 & 0 & 0 & \mathbf{c_3} & \mathbf{c_2} & \mathbf{c_1} & \mathbf{c_0} \\ \hline \end{array} \tag{8.9}$$

Notice that there are 4 *leading bits which are unused; these are always set to* 0*.*

$P(\alpha) = \alpha^4 + \alpha^3 + 1 = 0$*. Hence,* $\alpha^4 = \alpha^3 + 1$*, which is stored as*

$$\alpha^4 \quad = \begin{array}{cccccccc} & & & & c_3 & c_2 & c_1 & c_0 \\ \hline \multicolumn{1}{|c|}{0} & \multicolumn{1}{c|}{0} & \multicolumn{1}{c|}{0} & \multicolumn{1}{c|}{0} & \multicolumn{1}{c|}{\mathbf{1}} & \multicolumn{1}{c|}{\mathbf{0}} & \multicolumn{1}{c|}{\mathbf{0}} & \multicolumn{1}{c|}{\mathbf{1}} \\ \hline \end{array} \tag{8.10}$$

To compute α^5*, take the* α^4 *element and shift the result left 1 bit. The* c_3 *term is dropped since the leading* 4 *bits are always* 0*.*

$$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 0 & 0 & 0 & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \hline \end{array} \tag{8.11}$$

Then, since c_3 *was* 1*, add* α^4*.*

$$\begin{array}{r} \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 0 & 0 & 0 & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \hline \end{array} \\ + \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \hline \end{array} \\ \hline \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \hline \end{array} \end{array} \tag{8.12}$$

This gives $\alpha^5 = \alpha^3 + \alpha + 1$*. Similarly,* α^6 *is derived as*

$$\begin{array}{r} \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 0 & 0 & 0 & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ \hline \end{array} \\ + \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \hline \end{array} \\ \hline \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ \hline \end{array} \end{array} \tag{8.13}$$

Multiplication requires temporarily increasing the size of the byte-array to store the intermediate result, which can have values up to $\alpha^{2(k-1)}$.

$$c_{2k-2} \cdot \alpha^{2k-2} + \cdots + c_k \cdot \alpha^k + c_{k-1} \cdot \alpha^{k-1} + \cdots + c_2 \cdot \alpha^2 + c_1 \cdot \alpha + c_0 \tag{8.14}$$

This result needs to be divided by the given minimum polynomial. Each α term with an exponent of k or larger is replaced by its minimized equivalent, which was computed during initialization. That is, for $i \geq k$, each term for which $c_i = 1$ is removed and the minimized form of α^i added in.

Example 8.2 *Consider again the setup for $\mathbb{F}_{2^4}$ from Example 8.1. Compute the product of the following two elements:*

$$\alpha^3 + \alpha^2 + 1 \tag{8.15}$$

$$\alpha^2 + \alpha \tag{8.16}$$

These elements are stored respectively as

$$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 0&0&0&0&\mathbf{1}&\mathbf{1}&\mathbf{0}&\mathbf{1}\\\hline\end{array} \tag{8.17}$$

$$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 0&0&0&0&\mathbf{0}&\mathbf{1}&\mathbf{1}&\mathbf{0}\\\hline\end{array} \tag{8.18}$$

The intermediate result is computed using the basic shift-and-add procedure.

$$\begin{array}{ccccccccc}
 & & & & & 1&1&0&1\\
 & & & & x & 0&1&1&0\\
\hline
 & & & & & 0&0&0&0\\
 & & & & 1&1&0&1& \\
 & & & 1&1&0&1& & \\
+ & & 0&0&0&0& & & \\
\hline
 & 0&\mathbf{0}&\mathbf{1}&\mathbf{0}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{0}
\end{array} \tag{8.19}$$

The intermediate result of the multiplication is $\alpha^5+\alpha^3+\alpha^2+\alpha$, which needs to be further minimized. The value of α^5 was determined during initialization to be $\alpha^3+\alpha+1$. The α^5 term from the intermediate result is removed and the minimized form is added.

$$\begin{array}{c|c|c|c|c|c|c|c|c|}
\cline{2-9} & 0&0&0&0&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{0}\\
\cline{2-9} + & 0&0&0&0&\mathbf{1}&\mathbf{0}&\mathbf{1}&\mathbf{1}\\
\hline
\multicolumn{9}{c}{}\\
\cline{2-9} & 0&0&0&0&\mathbf{0}&\mathbf{1}&\mathbf{0}&\mathbf{1}\\
\cline{2-9}
\end{array} \tag{8.20}$$

So the minimized result of the product is α^2+1.

Division of two Galois field elements, $C = \frac{B}{A}$, requires finding the multiplicative inverse of the divisor: $C = B \cdot A^{-1}$. To find the inverse, the library implements the extended Euclidean algorithm over $\mathbb{F}_{2^k}$, depicted in Algorithm 5. The algorithm requires a non-minimized representation of the element $P(\alpha)$, so the size of object is temporarily increased to allow the storage of the α^k bit. The function DIV returns the quotient and remainder of a Euclidean division; that is, $DIV(A, B)$ returns $\{Q, R\}$, where $A = B \cdot Q + R$. This procedure is described in Algorithm 6; here, DEG returns the highest degree of a given element in $\mathbb{F}_{2^k}$, i.e. $DEG(\alpha^4 + \alpha^3 + 1)$ would return 4.

Algorithm 5: Inverse of an element over $\mathbb{F}_{2^k}$

```
**Input**: $M := P(\alpha)$ where $P(x)$ was used to generate $\mathbb{F}_{2^k}$, $A \in \mathbb{F}_{2^k}$
**Output**: $A^{-1}$ over $\mathbb{F}_{2^k}$
$\{Q_0, Q_1\} := \{0, 0\}$;
$\{R_0, R_1\} := \{M, A\}$;
$\{U_0, U_1\} := \{0, 1\}$;
$i := 1$;
**while** $R_i \neq 1$ **do**
    **if** $R_i == 0$ **then**
        **ERROR**: No inverse exists
    **end**
    $\{Q_{i+1}, R_{i+1}\} := DIV(R_{i-1}, R_i)$;
    $U_{i+1} := (Q_{i+1} \cdot U_i) + U_{i-1}$;
    $i := i + 1$;
**end**
**return** $U_i$;
```

Algorithm 6: DIV (Euclidean Division over $\mathbb{F}_{2^k}$)

```
**Input**: $A, B \in \mathbb{F}_{2^k}$
**Output**: $\{Q, R\}$ such that $A = B \cdot Q + R$
$\{Q, R\} := \{0, A\}$;
**while** $DEG(R) \geq DEG(B)$ **do**
    $S := \alpha^{DEG(R) - DEG(B)}$;
    $Q := Q + S$;
    $R := R + S \cdot B$;
**end**
**return** $\{Q, R\}$;
```

Example 8.3 *Given $P(x) = x^8 + x^4 + x^3 + x + 1$ which generates $\mathbb{F}_{2^8}$, find A^{-1} where $A = \alpha^6 + \alpha^4 + \alpha + 1$. Table 8.1 shows the steps Algorithm 6 goes through to find the inverse.*

Table 8.1: Steps to derive the inverse of $\alpha^6 + \alpha^4 + \alpha + 1$

i	$\mathbf{Q_i}$	$\mathbf{R_i}$	$\mathbf{U_i}$
0	0	$\alpha^8 + \alpha^4 + \alpha^3 + \alpha + 1$	0
1	0	$\alpha^6 + \alpha^4 + \alpha + 1$	1
2	$\alpha^2 + 1$	α^2	$\alpha^2 + 1$
3	$\alpha^4 + \alpha^2$	$\alpha + 1$	$\alpha^6 + \alpha^2 + 1$
4	α	1	$\alpha^7 + \alpha^6 + \alpha^3 + \alpha$

The derived inverse is $A^{-1} = \alpha^7 + \alpha^6 + \alpha^3 + \alpha$. Correctness can be checked by computing $A \cdot A^{-1}$ and verifying that the result is 1.

8.1.2 Rings and Monomials over Galois Fields

A monomial M over the ring $\mathbb{F}_{2^k}[x_1, \ldots, x_d]$ is a power-product of variables from the ring along with a coefficient $C \in \mathbb{F}_{2^k}$.

$$M = C \cdot {x_1}^{e_1} \cdot {x_2}^{e_2} \cdots {x_d}^{e_d}; \quad e_i \geq 0 \tag{8.21}$$

Ring variables can either be bit-level (representing a single wire within a circuit) or word-level (representing a word input or output). If x_i is a bit-level variable then $x_i \in \mathbb{F}_2$; thus it has the property $x_i^2 = x_i$, so its exponent $e_i \in \{0, 1\}$. If the variable is word-level, then $e_i < 2^k$ due to the property $x_i^{2^k} = x_i$.

Lex ordering is the only monomial ordering used for abstraction, and hence it is the only ordering implemented in the tool. Ring variables are added as strings, one at a time, along with an argument stating whether the variable is bit-level or word-level. Each variable is given a unique unsigned integer id, which is continuously incremented with each added variable. Thus, ids of two variables can be compared to quickly distinguish which variable appears earlier in the ordering.

Three static objects are created during initialization of the ring:

- **strToId** - a map of each variable name (string) to its id (unsigned int)
- **idToStr** - a map of each id (unsigned int) to its variable name (string)
- **wordSet** - a set of ids (unsigned int) of all variables which are word-level

Here, “map” and “set” are classes of the standard C++ library. It’s important to note that a C++ “set” is a container of unique, ordered elements (this property is exploited later). The “strToId” map object is used when constructing monomials to quickly find the id of a parsed variable name. The “idToStr” map object allows a monomial object to be printed to the user. The “wordSet” object is used to quickly check whether a given variable is word-level, which determines how it is handled during monomial operations.

Once the ring has been initialized, monomials can be generated and manipulated. Internally, all monomial variables are manipulated using their ids. Each monomial object contains the following:

- **coef** - a Galois field object (as described in the previous subsection)
- **idSet** - a set of ids (unsigned int) of all variables in the monomial
- **idToExp** - a map of variable ids (unsigned int) to their exponents (BigUnsigned)

During monomial creation, string variable names are parsed and mapped to their corresponding ids, which are then added to the set. The exponent map is only filled for variables that are word-level. As most variables are bit-level in a circuit, most monomials will have a completely empty map. Since exponents can be much larger than what can be stored in a primitive data structure, each exponent is stored as a BigUnsigned object of the open source library *BigInt* [108]. This is a library which provides basic functionality for signed and unsigned integers of unbounded size.

Monomial comparison is required for proper monomial ordering, which is necessary for implementation of polynomial procedures such as reduction. The comparison procedure compares the id sets of two monomials, one variable at a time. If the variables differ, the smaller id appears earlier in the ordering. If they are the same, exponents are checked if the variable is word-level. This procedure is shown in Algorithm 7.

Algorithm 7: Monomial Comparison

```
Input: Monomials $M_1$ and $M_2$
Output: $< 0$ if $M_2 > M_1$, $> 0$ if $M_1 > M_2$, 0 if $M_1 == M_2$
id$_1$ := $M_1$.idSet.begin();
id$_2$ := $M_2$.idSet.begin();
while $id_1 \neq \varnothing$ && $id_2 \neq \varnothing$ do
    if $id_1 \neq id_2$ then
        return id$_2$-id$_1$;
    end
    if $id_1 \in wordSet$ then
        if $M_1$.idToExp[id$_1$] $\neq$ $M_2$.idToExp[id$_2$] then
            return $M_1$.idToExp[id$_1$] - $M_2$.idToExp[id$_2$];
        end
    end
    id$_1$ := $M_1$.idSet.next();
    id$_2$ := $M_2$.idSet.next();
end
if $id_1$ == $\varnothing$ && $id_2$ == $\varnothing$ then
    return $0$;
end
if $id_1$ == $\varnothing$ then
    return $-1$;
end
return $1$;
```

Multiplication of two monomials is the main function of this portion of the tool. First, the two Galois field objects are multiplied together using the previously described method. Then, the two sets of ids are merged together. Since sets can only contain unique values, duplicates are discarded; this is done automatically using the standard set::insert operation. In the common case, both monomials only contain bit-level variables and the multiplication would be complete. If there are word-level variables in the monomials, the mapped exponents of each such variable would be added together and then minimized if the resulting exponent is $\geq 2^k$.

Example 8.4 *Consider again the setup for $\mathbb{F}_{2^4}$ from Example 8.1. Construct the ring $\mathbb{F}_{2^4}[a, b, c, Z]$ with the lex ordering $a > b > c > Z$, where $\{a, b, c\}$ are bit-level variables and Z is a word-level variable. The initialized monomial static library objects are:*

strToId			**idToStr**			**wordSet**
“a”	$\rightarrow$	0	0	$\rightarrow$	“a”	
“b”	$\rightarrow$	1	1	$\rightarrow$	“b”	$\{3\}$
“c”	$\rightarrow$	2	2	$\rightarrow$	“c”	
“Z”	$\rightarrow$	3	3	$\rightarrow$	“Z”	

(8.22)

Let M_1 and M_2 be the following monomials:

$$M_1 = (\alpha^3 + \alpha^2 + 1)abZ^{10} \qquad M_2 = (\alpha^2 + \alpha)bcZ^7 \tag{8.23}$$

These are stored internally by the tool as:

$\mathbf{M_1}$

coef								**idSet**	**idToExp**
0	0	0	0	**1**	**1**	**0**	**1**	$\{0, 1, 3\}$	$3 \rightarrow 10$

(8.24)

$\mathbf{M_2}$

coef								**idSet**	**idToExp**
0	0	0	0	**0**	**1**	**1**	**0**	$\{1, 2, 3\}$	$3 \rightarrow 7$

(8.25)

It’s easy to see that $M_1 > M_2$ in the given ordering, since the first element of “idSet” in M_1 is 0 while in M_2 it is 1. Multiplying M_1 by M_2 is computed by first multiplying the Galois field elements (coef) together (as shown in Example 8.2). Then, the two sets (idSet) are merged (union). Notice that, although variable b appears in both monomials, $b^2 = b$ due to it being a bit-level variable. This is handled automatically by the set class (duplicates thrown out). Finally, the two corresponding exponents of variable Z (idToExp) are added together. Since this new exponent of Z is 17, and $17 \geq 2^4$, the exponent is minimized by the property $Z^{16} = Z$. Thus, the new exponent of Z is 2.

$\mathbf{M_1 \cdot M_2}$

coef								**idSet**	**idToExp**
0	0	0	0	**0**	**1**	**0**	**1**	$\{0, 1, 2, 3\}$	$3 \rightarrow 2$

(8.26)

So $M_1 \cdot M_2 = (\alpha^2 + 1)abcZ^2$

Monomial division is a procedure mainly used during polynomial reduction. Given two monomials M_1 and M_2, compute $\frac{M_1}{M_2}$. That is, find a monomial M_3 such that $M_1 = M_2 \cdot M_3$. Monomial division is described in Algorithm 8. The division procedure loops over all variables in M_2 and removes them from M_1 if they are bit-level. For word-level variables, exponents are subtracted from each other. If a variable exists in M_2 but not in M_1, or if the exponent of a word-level variable in M_2 is larger than in M_1, the division is 0. Finally at the end, the Galois field elements are divided by each other.

Algorithm 8: Monomial Division

```
Input: Monomials $M_1$ and $M_2$
Output: $\frac{M_1}{M_2}$ if it exists
$M_3$ := $M_1$;
foreach $id \in M_2.idSet$ do
    if $id \notin M_3$ then
        return NULL;
    end
    if $id \notin wordSet$ then
        $M_3$.idSet.erase(id);
    else
        if $M_2.idToExp[id] > M_3.idToExp[id]$ then
            return NULL;
        end
        $M_3$.idToExp[id] -= $M_2$.idToExp[id];
        if $M_3.idToExp[id] == 0$ then
            $M_3$.idSet.erase(id);
        end
    end
end
$M_3$.coef /= $M_2$.coef;
return $M_3$;
```

8.1.3 Polynomials and Polynomial Division

With the monomial structure defined, a polynomial is simply a C++ vector of monomial objects. These monomial objects are ordered by the given ring ordering, which is imposed at all times, using monomial comparisons.

Polynomial addition is computed by simply merging the vector lists of two polynomials together, since the two polynomial vectors are already sorted. If two monomials are found to be equal, their Galois field coefficients are added together and the resulting monomial added to the sum if the new coefficient is not 0.

Multiplication of a polynomial by a monomial, $P_2 = M_1 \cdot P_1$ is detailed in Algorithm 9. Each monomial in P_1 is iteratively multiplied by M_1 to derive a temporary monomial M_{temp}, which is added to P_2. Let M_{last} denote the last monomial in P_2 at any given time. Due to the ordering, typically $M_{last} \geq M_{temp}$ during the procedure. In cases where it is not, which only happens when exponents have been minimized, M_{temp} falls not far earlier than M_{last}. Thus, to add M_{temp} to P_2, M_{temp} is compared to monomials in P_2 in reverse order.

Algorithm 9: Multiplication of a Polynomial by a Monomial

```
Input: Monomial $M_1$, Polynomial $P_1$.
Output: $M_1 \cdot P_1$
$P_2$ := ∅;
foreach Monomial $M_p \in P_1$ do
    $M_{temp}$ := $M_p \cdot M_1$;
    foreach Monomial $M_{p2} \in P_2$ in reverse order do
        if $M_{p2} > M_{temp}$ then
            $P_2$.insertAfter($M_{p2}$,$M_{temp}$);
            break;
        end
        if $M_{p2} == M_{temp}$ then
            $M_{p2}$.coef += $M_{temp}$.coef;
            if $M_{p2}$.coef == $0$ then
                $P_2$.pop();
            end
            break;
        end
    end
    if $M_{temp}$ hasn't been inserted then
        $P_2$.insertToFront($M_{temp}$);
    end
end
return $P_2$;
```

Multiplication of polynomials, $P_3 = P_1 \cdot P_2$, is computed as numerous monomial-by-polynomial multiplications. Each monomial in P_1 is multiplied by the entire polynomial P_2 to derive a temporary polynomial P_{temp}. Then, P_{temp} is added to a growing P_3. Order is maintained by these sub-procedures, so no further ordering logic is needed.

Example 8.5 *Assume the environment has been set up over the ring $\mathbb{F}_{2^4}[a, b, c, Z]$ as in Example 8.4. Let P_1 and P_2 be the following polynomials*

$$P_1 = (\alpha)ab + bZ^3 \qquad P_2 = abc + ab + b \tag{8.27}$$

$P_1 + P_2$ is computed by merging the polynomials together. Two monomials exist with the same order, $(\alpha)ab$ in P_1 and ab in P_2, so here only the coefficients are merged.

$$P_1 + P_2 = abc + (\alpha + 1)ab + bZ^3 + b \tag{8.28}$$

$P_1 \cdot P_2$ is computed by taking each monomial of P_1 and multiplying it by P_2. The first temporary polynomial generated is:

$$(\alpha)ab \cdot P_2 = (\alpha)ab \cdot (abc + ab + b) = (\alpha)abc + (\alpha)ab + (\alpha)ab = (\alpha)abc \tag{8.29}$$

Notice that, since two equivalent terms were generated, their coefficients were added together creating $(0) \cdot ab$, so this term was removed. The second multiplication is

$$bZ^3 \cdot P_2 = bZ^3 \cdot (abc + ab + b) = abcZ^3 + abZ^3 + bZ^3 \tag{8.30}$$

Finally, these two polynomials are added together to obtain the final result.

$$P_1 \cdot P_2 = abcZ^3 + (\alpha)abc + abZ^3 + bZ^3 \tag{8.31}$$

Polynomial reduction is the main procedure computed by the abstraction tool. To reduce a polynomial P_1 by polynomial P_2, one reduction step is computed as

$$P_1 \xrightarrow{P_2} P_1 + \frac{LT(P_1)}{LT(P_2)} \cdot P_2 \tag{8.32}$$

where LT is the first monomial object of the given polynomial. $\frac{LT(P_1)}{LT(P_2)}P_2$ modifies P_2 so that it has the same leading term as P_1. Thus, when the two polynomials are added together, the leading terms are cancelled out. Note that reduction is only possible when $LT(P_1)$ is divisible by $LT(P_2)$. However, one reduction step may not be sufficient to compute a full reduction, i.e. it's possible that the resulting polynomial could be reduced further by P_2. One can reapply the reduction steps until no more reductions are possible. A more efficient method is to collect all monomials in P_1 that are divisible by $LT(P_2)$, add the results of each division to P_{temp}, and then compute $P_1 + P_{temp} \cdot P_2$. Due to the ordering, if $LT(P_2) > M_i$ where M_i is the i^{th} monomial in P_1, then all $M_j \in P_1$ for $j \geq i$ are not divisible by $LT(P_2)$. Thus, the divisions are performed in monomial order and stopped as soon as this condition holds. The overall procedure is described in Algorithm 10. Polynomial reduction makes use of monomial division, monomial-by-polynomial multiplication, and polynomial addition, which were all detailed previously.

Algorithm 10: Polynomial Reduction

```
Input: Polynomial $P_1$, Polynomial $P_2$.
Output: $r$ where $P_1 \xrightarrow{P_2}_+ r$
$P_{temp}$ := $\varnothing$;
foreach Monomial $M_p \in P_1$ do
    if $LT(P_2) > M_p$ then
        break;
    end
    $M_{div}$ := $M_p / LT(P_2)$;
    if $M_{div} \mathrel{!}= \varnothing$ then
        $P_{temp}$.push_back($M_{div}$);
    end
end
return $P_1 + (P_{temp} \cdot P_2)$;
```

Example 8.6 *Consider again the $\mathbb{F}_{2^4}[a, b, c, Z]$ setup from Example 8.5 with*

$$P_1 = (\alpha)ab + bZ^3 \qquad P_2 = abc + ab + b \tag{8.33}$$

Compute $P_2 \xrightarrow{P_1}_+ r$. Attempt to divide each monomial of P_2 by $LT(P_1) = (\alpha)ab$. The first monomial division is:

$$\frac{abc}{(\alpha)ab} = (\alpha^3 + \alpha^2)c \tag{8.34}$$

Note that over this field, $(\alpha)^{-1} = (\alpha^3 + \alpha^2)$. The next monomial division is:

$$\frac{ab}{(\alpha)ab} = (\alpha^3 + \alpha^2) \tag{8.35}$$

The last monomial division is not possible,

$$\frac{b}{(\alpha)ab} = \varnothing \tag{8.36}$$

so $P_{temp} = (\alpha^3 + \alpha^2)c + (\alpha^3 + \alpha^2)$. *The final reduction is then computed as*

$$\begin{aligned}
&P_2 + (P_{temp} \cdot P_1) \\
&= abc + ab + c + ((\alpha^3 + \alpha^2)c + (\alpha^3 + \alpha^2)) \cdot ((\alpha)ab + bZ^3) \\
&= abc + ab + c + (abc + ab + (\alpha^3 + \alpha^2)bcZ^3 + (\alpha^3 + \alpha^2)bZ^3) \\
&= (\alpha^3 + \alpha^2)bcZ^3 + (\alpha^3 + \alpha^2)bZ^3 + c \qquad (8.37)
\end{aligned}$$

Notice that the two monomials of P_2 *that were divisible by* $LT(P_1)$ *are cancelled out.*

8.2 Abstraction Tool Flow and Results

The tool takes the circuit as input and applies the approach presented in Chapter 6 to derive the polynomial representation of the circuit. The most computationally intensive procedures in the approach are

1. Initial reduction of the word-level polynomial, $f_z \xrightarrow{F-\{f_z\},F_0}_+ r$
2. Derivation of the polynomials for the second reduction procedure, $F_a = \{a_0 = \mathcal{F}_0(A), a_1 = \mathcal{F}_1(A), \dots\}$
3. Computation of the second reduction (substitution), $r \xrightarrow{F_A,F_0}_+ Z + \mathcal{F}(A)$

Of these, the first two are computed in parallel as they are independent of each other. Step 1 orders the polynomials in $J + J_0$ by their monomial order and reduces f_Z in that order. In our experiments, this step typically takes longer than step 2.

All experiments are run on a 64-bit Linux desktop with a 3.5GHz Intel Core™ i7 Quad-core CPU and 16 GB of RAM. Table 8.2 depicts the time and memory required to derive the polynomial abstraction from bug-free and buggy Mastrovito multiplier circuits using our custom tool. This circuit is provided as a bit-blasted/flattened gate-level netlist. These circuits compute $Z = A \cdot B$ over some field $\mathbb{F}_{2^k}$, so the analysis is performed over this same field, abstracting this word-level representation. The bug introduced is a swapping of two output nodes of the given circuit, ensuring that the effect propagates down during the reduction process. Similarly, Table 8.3 depicts the results for abstracting flattened Montgomery multipliers.

Table 8.2: Abstraction of Mastrovito multipliers. Time given in seconds, memory given in MB. TO = 3 days ($259,200$ seconds.)

Size (k)		163	233	283	409	571
# of Gates		153K	167K	399K	508K	1.6M
Time (s)	Bug Free	1,443	1,913	11,116	17,848	192,032
	Buggy	1,487	2,106	11,606	20,263	204,194
Max Memory (MB)		213	269	561	845	2,855

Montgomery multipliers are typically designed hierarchically, as shown in Fig. 3.4. If the hierarchy is known, it can be exploited by computing the abstraction of each MR block in parallel, as shown in Table 8.4. In this table, 'BLK A' and 'B' denote the input MR blocks, 'BLK Mid' denotes the middle block and 'BLK Out' is the output block. While each block is an MR block, some have been simplified by constant-propagation, hence they have different sizes. First, a polynomial is extracted for each MR block (gate-level to word-level abstraction), and then the approach is re-applied at word-level to derive the input-output relation (solved trivially in < 1 second). Our approach can extract the word-level polynomial for up to 571-bit circuits!

Table 8.3: Abstraction of flat Montgomery multipliers. Time given in seconds, memory given in MB. TO = 3 days ($259,200$ seconds.

Size (k)		163	233	283	409	571
# of Gates		184K	329K	488K	1.0M	1.97M
Time	Bug Free	6,897	63,805	TO	TO	TO
	Buggy	6,961	64,009	TO	TO	TO
Max Memory		153	325	505	971	2,240

To abstract a word-level representation of the composite field multiplier $\mathbb{F}_{(2^m)^n}$ [Fig. 7.2], we first apply our approach to abstract a word-level representation of each m-bit block. In the case of an adder, this abstraction is trivially computed in 1 second. In the case of a multiplier, refer to our experimental results for Mastrovito multipliers for comparison (Table 8.2) as these are designed as m-bit Mastrovito blocks. Each abstraction can be computed independently. Once these word-level abstractions are known, the final abstraction over $\mathbb{F}_{(2^m)^n}$ is performed using only word-level variables.

Table 8.4: Abstraction of Montgomery blocks. Time given in seconds, memory is given in MB. TO = 3 days (259,200 seconds)

Circuit Size (k)			163	233	283	409	571
# of Gates		Blk A	33K	55K	82K	168K	330K
		Blk B	33K	55K	82K	168K	330K
		Blk Mid	85K	163K	241K	502K	980K
		Blk Out	32K	54K	81K	168K	328K
Time	Bug Free	Blk A	25	142	330	1,322	5,371
		Blk B	25	141	329	1,335	5,241
		Blk Mid	73	408	883	4,471	19,942
		Blk Out	24	140	321	1,338	5,532
	Buggy	Blk A	26	142	331	1,323	5,372
		Blk B	26	141	330	1,336	5,421
		Blk Mid	111	580	1,411	6,829	37,804
		Blk Out	25	141	322	1,339	5,539
Max Mem Per Blk			80	168	254	538	1,129

The results of this final word-level abstraction of buggy and bug-free multipliers over composite fields are shown in Table 8.5.

The above experiments also demonstrate that we can perform equivalence checking between Mastrovito (golden model) and Montgomery multiplier (implementation) circuits, by deriving a canonical polynomial (Z_1, Z_2)from each circuit independently and then checking if $Z_1 = Z_2$, for up to 571-bit circuits. Our experiments have shown that contemporary approaches (BDDs, SAT, SMT, and AIG/ABC) show some success in bug-catching for these kinds of circuits (particularly ABC). These experiments are shown in Table 8.7. Note that ABC uses random simulation for FRAIGing (functionally reducing AIGs), which can catch a bug early if it is lucky. However, full equivalence checking using any of these techniques failed even for 16-bit circuits, as shown in Table 8.8.

128					256				
m	n	Time		Max	m	n	Time		Max
		Bug Free	Buggy	Mem			Bug Free	Buggy	Mem
2	64	1	1	4	2	128	15	15	23
4	32	1	1	2	4	64	2	2	4
8	16	1	1	2	8	32	1	1	3
16	8	1	1	2	16	16	1	1	2
32	4	1	1	2	32	8	1	1	2
64	2	1	1	3	64	4	1	1	2
-	-	-	-	-	128	2	1	1	2
-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-

512					1024				
m	n	Time		Max	m	n	Time		Max
		Bug Free	Buggy	Mem			Bug Free	Buggy	Mem
2	256	406	408	90	2	512	$11,883$	$12,050$	414
4	128	53	53	25	4	256	$1,520$	$1,536$	106
8	64	8	8	4	8	128	209	211	29
16	32	2	2	4	16	64	38	37	10
32	16	1	1	3	32	32	10	10	5
64	8	1	1	3	64	16	4	4	3
128	4	1	1	2	128	8	2	2	3
256	2	1	1	2	256	4	1	1	3
-	-	-	-	-	512	2	1	1	3

Table 8.5: Abstraction of bug-free Mastrovito multipliers over $\mathbb{F}_{(2^m)^n}$. Time is given in seconds. Memory is given in MB. TO = more than 24 hours = $86,400$ seconds. Note that abstractions of the m-bit blocks which compose the circuits are already known.

Table 8.6: Statistics of Designs over $\mathbb{F}_{(2^m)^n}$

n	2	4	8	16	32
# of $\mathbb{F}_{2^m}$ Multipliers	6	36	168	720	2976
# of $\mathbb{F}_{2^m}$ Adders	3	27	147	675	2883

Table 8.7: Bug-catching between a golden-model Mastrovito and buggy Montgomery circuit. Time given in seconds. TO = 3 days (259,200 seconds)

Circuit Size (k)	64	163	233	283	409	571
ABC	1	32	6	96	217	401
Lingeling	1	8	362	12,728	3,323	23,298
Picosat	15,235	TO	TO	TO	TO	TO
Boolector	4	30	41	105	152	19,113
CVC4	2	11	64	8,660	280	TO
Z3	1	12	55	10,169	335	TO
Yices	1	6	7	618	578	11,568

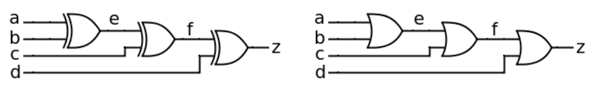
Table 8.8: Equivalence checking between a golden-model Mastrovito and a bug-free Montgomery circuit. TO = 3 days (259,200 seconds)

Circuit Size (k)	16
ABC	TO
Lingeling	TO
Picosat	TO
Boolector	TO
CVC4	TO
Z3	TO
Yices	TO

8.3 Limitations of the Abstraction Approach

Our tools and approach performs very favorable for $\mathbb{F}_{2^k}$ multiplier circuits and other functions designed over fields. These types of circuits are based on AND-XOR gate logic. Thus, the polynomials derived during the reduction procedures are very sparse. Since the complexity of the algorithm heavily depends on the density of the polynomials, the worst-case is avoided in such designs. In the case of bugs within these circuits, these polynomials increase in size, but are still easily manageable. This is why the approach is applicable to very large Galois field multipliers, both buggy and bug-free.

However, for random logic, especially logic containing chains of OR gates, the polynomial becomes very dense. As a result, the algorithm begins to encounter the computational worst-case. This is best shown with a small example.



(a) XOR logic (b) OR logic

Figure 8.2: Logic comparisons

Example 8.7 *Consider the circuit in Figure 8.2a, which performs a 4-input XOR function. Due to RATO, the monomial ordering of the variables is $z > f > e > d > c > b > a$. Thus, z will be reduced in terms of the rest of the variables. The polynomials derived from the design are:*

$$f_1 : z + f + d \quad f_2 : f + e + c \quad f_3 : e + b + a$$

The reduction procedure $z \xrightarrow{f_1, f_2, f_3}_{+} r$ will be computed as follows:

1. $z \xrightarrow{z+f+d} f + d$

2. $(f + d) \xrightarrow{f+e+c} e + d + c$

3. $(e + d + c) \xrightarrow{e+b+a} d + c + b + a$

In each reduction, the output gate variable is removed and one copy of each input variable is added, leaving a sparse polynomial. Now consider the same circuit with the XOR gates replaced by OR gates, as shown in Figure 8.2b.

The monomial ordering stays the same, but the polynomials derived from each gate have changed:

$$f_1 : z + fd + f + d \quad f_2 : f + ec + e + c \quad f_3 : e + ba + b + a$$

The reduction procedure, $z \xrightarrow{f_1, f_2, f_3}_+ r$ is now computed as:

1. $z \xrightarrow{z+fd+f+d} fd + f + d$

2. $(fd + f + d) \xrightarrow{f+ec+e+c} f + edc + ed + dc + d$;
   $(f + edc + ed + dc + d) \xrightarrow{f+ec+e+c} edc + ed + ec + e + dc + d + c$

3. $(edc + ed + ec + e + dc + d + c) \xrightarrow{e+ba+b+a}_+$
   $dcba + dcb + dca + dba + dc + db + da + d + cba + cb + ca + c + ba + b + a$

Each pass removes an output variable of the gate, but replaces it with two instances of each input variable. This increases the density of the resulting polynomial exponentially.

8.4 Conclusions

This chapter examined the data structures and algorithms implemented in a custom software abstraction tool. Abstraction of Galois field circuits using the custom tool has greatly better performance compared to using SINGULAR scripts. With it, we can abstract circuits up to 1024-bits, buggy or bug-free. Although a bug will generally substantially inflate the size of the resulting polynomial abstraction, the tool is not greatly hindered by the presence of bugs in a circuit. However, for random logic, especially OR-based logic, the size of the polynomials tends to grow exponentially during the abstraction. Thus, the approach is infeasible for circuits with OR-gate chains.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

A combinational circuit with k-inputs and k-outputs implements Boolean functions $f : \mathbb{B}^k \to \mathbb{B}^k$, where $\mathbb{B} = \{0, 1\}$. The function can also be construed as a mapping $f : \mathbb{F}_{2^k} \to \mathbb{F}_{2^k}$, where $\mathbb{F}_{2^k}$ denotes the Galois field of 2^k elements. A circuit with differing input and output sizes computes $f : \mathbb{B}^m \to \mathbb{B}^n$, which can be represented as a function over Galois fields $f : \mathbb{F}_{2^m} \to \mathbb{F}_{2^n}$. This circuit can also be analyzed as the function $f : \mathbb{F}_{2^k} \to \mathbb{F}_{2^k}$, where $\mathbb{F}_{2^k} \supset \mathbb{F}_{2^m}$ and $\mathbb{F}_{2^k} \supset \mathbb{F}_{2^m}$.

Every function f over $\mathbb{F}_{2^k}$ is a polynomial function — i.e., there exists a unique, minimal, canonical polynomial $\mathcal{F}$ that describes f. This dissertation presented novel techniques based on computer-algebra and algebraic-geometry to derive the canonical (word-level) polynomial representation from the circuit as $Z = \mathcal{F}(A)$ over $\mathbb{F}_{2^k}$, where A and Z denote, respectively, the input and output bit-vectors of the circuit.

A theory for word-level polynomial abstraction of bit-level circuits over Galois fields is first developed. This theory is derived using techniques from computer-algebra, notably the theory of Gröbner basis. However, due to the computational complexity of computing a Gröbner basis, the solution is not scalable to large designs. In order to overcome these limitations, new symbolic computational algorithms are developed and refined. The algorithms employ techniques from the binomial expansion over $\mathbb{F}_{2^k}$ and $\mathbb{F}_4$-style reduction and can exploit hierarchy in a given circuit. Finally, an efficient implementation of the algorithmic approach is presented.

Experiments show that the proposed approach works exceptionally well for abstracting word-level Galois field arithmetic circuits. It has been shown that the approach can abstract and verify these types of circuits with up to 1024-bit datapaths. Other contemporary techniques cannot to verify these types for circuits beyond 163-bits and fail to abstract them beyond 32-bits.

However, in cases of random logic, the abstraction approach can generate high-degree polynomials:

$$X^{q-1} + X^{q-2} \dots \tag{9.1}$$

In these cases, the polynomials derived during the computation are dense, and the computational complexity of manipulating such polynomials makes abstraction infeasible.

9.1 Future Work

Due to the modular nature of the proposed solution, there are many potential future research directions that can be explored.

9.1.1 Hardware Acceleration

The first reduction step, $f_Z \xrightarrow{F-f_{z_i},F_0}_+ r$ is the most computationally complex part of the proposed abstraction approach. This reduction could be implemented using a hardware accelerator. Significant speed-ups have been observed in GPU implementations of circuit simulation algorithms [109]. Furthermore, this work has shown cases where multiple, independent abstractions need to be computed at the same time, such as when abstracting a word-level representation of a composite field multiplier.

These abstractions can be computed in parallel with one another, and this parallelism could then be exploited using a GPU. Furthermore, our approach to compute the abstractions uses an $F4$-style reduction procedure, which performs many complex computations over a large matrix. Operations over matrices can be suitably implemented using a GPU. Lastly, the substitution by $a_i = \mathcal{F}(A)$ is trivially parallelized. Thus, further study is proposed to implement word-level abstraction on a general purpose GPU.

9.1.2 Integration with EDA Tools

The proposed canonical word-level abstraction approach is a full, self-contained solution. It can thus be integrated into other EDA tools. There are direct applications of word-level abstractions to design synthesis. For instance, the approach can compute a functional decomposition of a logic or it could be used in high-level RTL synthesis. Since the derived abstraction is canonical, it can also be used in verification engines such as SMT solvers. The abstraction approach most efficiently handles AND/XOR logic, so it could be used to complement approaches in the mentioned tools that are efficient over AND/OR logic.

9.1.3 Polynomial Reductions using Data-Structures

The abstraction approach poorly handles chains of OR gates due to their representation as polynomials of Galois fields. Other polynomial-based tools [110] have shown that it can be beneficial to represent polynomials internally as decision diagrams. Thus, it is worthwhile to explore whether it is possible to implement the algorithmic approach in a different data-structure that is better-suited for handling this type of logic. One candidate data-structure is the And-Invert-Graph, as this structure efficiently handles OR gates. The widely-used tool ABC [62] provides a very efficient, flexible, and open-source implementation of the AIG data structure.

Recall that a one-step reduction of the polynomial f by polynomial g, $f \xrightarrow{g} r$, is computed as:

$$r = f - \frac{LT(f)}{LT(g)} \cdot g \tag{9.2}$$

Over Boolean circuits, $\mathbb{B} \equiv \mathbb{F}_2$. Since the leading term of any monomial in $\mathbb{B}$ is 1, and $-1 \equiv +1$, then

$$f - (\frac{LT(f)}{LT(g)} \cdot g) \equiv f + (\frac{LM(f)}{LM(g)} \cdot g) \tag{9.3}$$

which computes an XOR operation

$$f \oplus (\frac{LM(f)}{LM(g)} \cdot g) \tag{9.4}$$

while the $\cdot$ operator acts as an AND operation $\wedge$. So one step of the reduction procedure can be computed as the following AND/XOR operation:

$$r = f \oplus \frac{LM(f)}{LM(g)} \wedge g \tag{9.5}$$

Thus, we propose an investigation into implementing the algorithms presented in this dissertation over AIGs.

9.1.4 Application to Sequential Circuit Verification

Sequential Galois field arithmetic circuits over $\mathbb{F}_{2^k}$ take k-bit inputs and produce a k-bit result after k-clock cycles of operation. Formal verification of sequential arithmetic circuits with large datapath sizes is beyond the capabilities of contemporary verification techniques. To address this problem, we described a verification method in [26] which uses the presented abstraction approach to implicitly unroll the sequential arithmetic circuit over multiple (k) clock-cycles. The resulting function computed by the state-registers of the circuit is represented canonically as a multi-variate word-level polynomial over $\mathbb{F}_{2^k}$. While directly applicable to sequential Galois field arithmetic circuits, this work needs to be further generalized in order to make applicable to any sequential state machine.

9.1.5 Application to Formal Software Verification

Computer algebra techniques based on Gröbner basis theory have been used in formal software verification [69]. In this work, a Gröbner basis computation is used to derive *loop invariants*. However, the derived invariants are not bit-precise, so not every invariant that is computed can be applied to the verification. As our approach maintains the input-output relationship in the abstraction, it could be applied to find bit-precise invariants.

9.1.6 Application to Integer Arithmetic Circuits

The abstraction approach derives a word-level representation of circuits over Galois fields, $\mathbb{F}_{2^k}$. In order to expand its usability, we conjecture whether it is possible to apply concepts from this approach to **abstract word-level representations of circuits over integer rings**, $\mathbb{Z}_{2^k}$. As any function over a Galois field $\mathbb{F}_{2^k}$ is a polynomial function, there exists a polynomial which describes the word-level function of a given circuit over $\mathbb{F}_{2^k}$. However, not every function over an integer ring $\mathbb{Z}_{2^k}$ is a polynomial function. Thus, a single polynomial which describes the function of a circuit over $\mathbb{Z}_{2^k}$ is not guaranteed to exist. Even though there may not exist a single polynomial which describes the entire function over $\mathbb{Z}_{2^k}$, elimination theory and Gröbner basis still applies over this ring. Thus, it may be possible to modify the theory and implementation of our word-level abstraction approach in order to abstract a set of word-level polynomials over $\mathbb{Z}_{2^k}$.

APPENDIX

A.1 Representations of Base Field Elements over Extension Fields

Consider a field $\mathbb{F}_q$ and a k-bit extension of this field, $\mathbb{F}_{q^k}$. This extension is created using a primitive, irreducible polynomial $P(x)$ of degree k over $\mathbb{F}_q[x]$. Any element $A \in \mathbb{F}_{q^k}$ can be represented as

$$A = a_0 + a_1\alpha + \cdots + a_{k-1}\alpha^{k-1} \tag{A.1}$$

where $\{a_0, \ldots, a_{k-1}\} \in \mathbb{F}_q$ and α is the primitive element of $\mathbb{F}_{q^k}$, i.e. $P(\alpha) = 0$. The goal is to derive the polynomial functions $a_i = \mathcal{F}(A)$ for all $0 \leq i < k$. Note that $\mathbb{F}_q$ could itself be an extension field of a base field $\mathbb{F}_p$ where $q = p^l$ for some $l \geq 1$.

Each a_i has the following property:

$$a_i = a_i^q \tag{A.2}$$

Element A has a similar property:

$$A = A^{q^k} \tag{A.3}$$

Examine what happens when Equation A.1 is raised by the power q. The following lemma is applied.

Lemma A.1 *[85] Let $\alpha_1, \ldots, \alpha_t$ be any elements in $\mathbb{F}_{p^k}$. Then*

$$(\alpha_1 + \alpha_2 + \cdots + \alpha_t)^{p^i} = \alpha_1^{p^i} + \alpha_2^{p^i} + \cdots + \alpha_t^{p^i} \tag{A.4}$$

for all integers $i \geq 0$.

Corollary A.1 *Since $q = p^l$ for some $l \geq 1$, Lemma A.1 is applicable to $\mathbb{F}_{q^k}$. Let $\alpha_1, \ldots, \alpha_t$ be any elements in $\mathbb{F}_{q^k}$. Then*

$$(\alpha_1 + \alpha_2 + \cdots + \alpha_t)^{q^i} = \alpha_1^{q^i} + \alpha_2^{q^i} + \cdots + \alpha_t^{q^i} \tag{A.5}$$

for all integers $i \geq 0$.

Applying Corollary A.1 to Equation gives:

$$A^{q^i} = a_0^{q^i} + a_1^{q^i}\alpha^{q^i} + \cdots + a_{k-1}^{q^i}\alpha^{(k-1)q^i} \tag{A.6}$$

Then, applying Equation A.2 to this result gives:

$$A^{q^i} = a_0 + a_1\alpha^{q^i} + \cdots + a_{k-1}\alpha^{(k-1)q^i} \tag{A.7}$$

In this way, k unique equations are derived, $\{A, A^q, \ldots, A^{q^{k-1}}\}$, along with k unknowns, $\{a_0, \ldots, a_{k-1}\}$. These equations can be represented in matrix form, $\mathbf{A} = \mathbf{M}\mathbf{a}$, where $\mathbf{A} = \{A, A^q, \ldots, A^{q^{k-1}}\}^T$, $\mathbf{M}$ is a k by k matrix of coefficients $\in \mathbb{F}_{q^k}$, and $\mathbf{a} = \{a_0, \ldots, a_{k-1}\}^T$:

$$\begin{bmatrix} 1 & \alpha & \alpha^2 & \ldots & \alpha^{k-1} \\ 1 & \alpha^q & \alpha^{2q} & \ldots & \alpha^{(k-1)q} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \alpha^{q^{k-1}} & \alpha^{2q^{k-1}} & \ldots & \alpha^{(k-1)q^{k-1}} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{k-1} \end{bmatrix} = \begin{bmatrix} A \\ A^q \\ \vdots \\ A^{q^{k-1}} \end{bmatrix} \tag{A.8}$$

Treat $\mathbf{a}$ as a vector of unknowns, $\mathbf{M}$ and $\mathbf{A}$ as constants. This system of equations can be solved using Gaussian elimination. However, this system also has a special structure which can be exploited. $\mathbf{M}$ is a k by k Vandermonde matrix of the form $V(\alpha, \alpha^q, \ldots, \alpha^{q^{k-1}})$. Due to the property of Vandermonde matrices:

$$|\mathbf{M}| = \prod_{0 \le i < j < k} (\alpha^{q^j} - \alpha^{q^i}) \tag{A.9}$$

Since the elements $\{\alpha, \alpha^q, \ldots, \alpha^{q^{k-1}}\}$ are distinct:

$$|\mathbf{M}| \neq 0 \tag{A.10}$$

Thus, Cramer's rule can be applied to derive an equation for every a_i:

$$a_i = \frac{|\mathbf{M_i}|}{|\mathbf{M}|} \tag{A.11}$$

where $\mathbf{M_i}$ is $\mathbf{M}$ with the column $\{\alpha^i, \alpha^{iq}, \ldots, \alpha^{iq^{k-1}}\}^T$ replaced by $\mathbf{A}$.

$$\mathbf{M_i} = \begin{bmatrix} 1 & \alpha & \ldots & \alpha^{i-1} & A & \alpha^{i+1} & \ldots & \alpha^{k-1} \\ 1 & \alpha^q & \ldots & \alpha^{(i-1)q} & A^q & \alpha^{(i+1)q} & \ldots & \alpha^{(k-1)q} \\ \vdots & \vdots & . & \vdots & \vdots & \vdots & . & \vdots \\ 1 & \alpha^{q^{k-1}} & \ldots & \alpha^{(i-1)q^{k-1}} & A^{q^(k-1)} & \alpha^{(i+1)q^(k-1)} & \ldots & \alpha^{(k-1)q^(k-1)} \end{bmatrix} \tag{A.12}$$

Computing $|\mathbf{M_i}|$ by interpolating along the A' column gives

$$|\mathbf{M_i}| = \sum_{j=0}^{k-1}(-1)^{(i+j)}A^{q^j}|V_{i+1}(\alpha, \dots, \alpha^{q^{(j-1)}}, \alpha^{q^{(j+1)}}, \dots, \alpha^{q^{(k-1)}})| \tag{A.13}$$

where $V_i(x_1, \dots, x_n)$ is the Vandermonde matrix $V(x_1, \dots, x_n)$ with the i-th column skipped and an extra column added to the end.

$$V_i(x_1, \dots, x_n) = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{i-1} & x_1^{i+1} & \dots & x_1^n \\ 1 & x_2 & x_2^2 & \dots & x_2^{i-1} & x_2^{i+1} & \dots & x_2^n \\ \vdots & \vdots & \vdots & . & \vdots & \vdots & . & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^{i-1} & x_n^{i+1} & \dots & x_n^n \end{bmatrix} \tag{A.14}$$

The computation of determinant of V_i is known to the linear algebra community to be

$$|V_i(x_1, \dots, x_n)| = |V(x_1 \dots, x_n)| \cdot S_{n-i}(x_1, \dots, x_n) \tag{A.15}$$

where $S_i(x_1, \dots, x_n)$ is the i-th fundamental symmetric polynomial in $\{x_1, \dots, x_n\}$. Thus, $|\mathbf{M_i}|$ is computed as.

$$\begin{aligned} |\mathbf{M_i}| \quad = \quad & \sum_{j=0}^{k-1}[(-1)^j A^{q^j} \\ & \cdot|V(\alpha, \dots, \alpha^{q^{(j-1)}}, \alpha^{q^{(j+1)}}, \dots, \alpha^{q^{(k-1)}})| \\ & \cdot S_{n-1-i}(\alpha, \dots, \alpha^{q^{(j-1)}}, \alpha^{q^{(j+1)}}, \dots, \alpha^{q^{(k-1)}})] \end{aligned} \tag{A.16}$$

Lastly, this work has discovered the following proposition for the determinant $|\mathbf{M}|$.

Proposition A.1 *The determinant $|\mathbf{M}|$ has the property*

$$|\mathbf{M}|^q = (-1)^{k-1}|\mathbf{M}| \tag{A.17}$$

Proof. *Since $\mathbf{M}$ is a Vandermonde matrix of the form $V(\alpha, \alpha^q, \dots, \alpha^q k - 1)$, then*

$$|\mathbf{M}| = \prod_{0\leq i<j<k} (\alpha^{q^j} - \alpha^{q^i}) \tag{A.18}$$

Computing $|\mathbf{M}|^q$ gives

$$|\mathbf{M}|^q = [\prod_{0\leq i<j<k} (\alpha^{q^j} - \alpha^{q^i})]^q \tag{A.19}$$

Applying Corollary A.1 to Equation A.19 gives

$$|\mathbf{M}|^q = \prod_{0 \le i < j < k} (\alpha^{q^{j+1}} - \alpha^{q^{i+1}}) \tag{A.20}$$

When $j = k - 1$*, the product term is in the form* $(\alpha^{q^k} - \alpha^{q^{i+1}})$*. Since* $\alpha^{q^k} = \alpha$ *over* $\mathbb{F}_{q^k}$*, this term equivalent to* $-(\alpha^{q^{i+1}} - \alpha)$*. This gives the property:*

$$|\mathbf{M}|^q = (-1)^{k-1} |\mathbf{M}| \tag{A.21}$$

■

REFERENCES

[1] E. Biham, Y. Carmeli, and A. Shamir, "Bug Attacks", *in Proceedings on Advances in Cryptology*, pp. 221–240, 2008.

[2] Thomas R. Nicely, "Pentium FDIV Flaw", `http://www.trnicely.net/pentbug/pentbug.html`.

[3] Douglas N. Arnold, "The Patriot Missile Failure", `http://www.ima.umn.edu/~arnold/disasters/patriot.html`.

[4] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, First edition, 1991.

[5] E. Clarke, O. Grumberg, and D. Peled, *The Temporal Logic of Reactive and Concurrent Systems*, The MIT Press, 1999.

[6] F. Lu, L. Wang, K. Cheng, and R. Huang, "A Circuit SAT Solver With Signal Correlation Guided Learning", *in IEEE Design, Automation and Test in Europe*, pp. 892–897, 2003.

[7] G. Avrunin, "Symbolic Model Checking using Algebraic Geometry", *in Computer Aided Verification Conference*, pp. 26–37, 1996.

[8] C. Condrat and P. Kalla, "A Gröbner Basis Approach to CNF formulae Preprocessing", *in International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 618–631, 2007.

[9] Y. Watanabe and *et al*, "Application of Symbolic Computer Algebra to Arithmetic Circuit Verification", *in IEEE International Conference on Computer Design*, pp. 25–32, October 2007.

[10] W. W. Adams and P. Loustaunau, *An Introduction to Gröbner Bases*, American Mathematical Society, 1994.

[11] J. Lv, *Scalable Formal Verification of Finite Field Arithmetic Circuits using Computer Algebra Techniques*, PhD thesis, Univ. of Utah, Aug. 2012.

[12] H. Jain, D. Kroening, N. Sharygina, and E. Clarke, "Word Level Predicate Abstraction and Refinement Techniques for Verifying Rtl Verilog", *in Design Automation Conf.*, 2005.

[13] S. Horeth and Drechsler, "Formal Verification of Word-Level Specifications", *in IEEE Design, Automation and Test in Europe*, pp. 52–58, 1999.

[14] L. Arditi, "*BMDs can Delay the use of Theorem Proving for Verifying Arithmetic Assembly Instructions", in Srivas, editor, *In Proc. Formal methods in CAD*. Springer-Verlag, 1996.

[15] Z. Zeng, P. Kalla, and M. J. Ciesielski, "LPSAT: A Unified Approach to RTL Satisfiability", *in Proc. DATE*, 2001.

[16] R. Brummayer and A. Biere, "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays", *in TACAS 09, Volume 5505 of LNCS*. Springer, 2009.

[17] R. Brant, D. Kroening, and *et al*, "Deciding Bit-Vector Arithmetic with Abstraction", *in Proc. TACAS*, pp. 358–372, 2007.

[18] D. Babic and M. Musuvathi, "Modular Arithmetic Decision Procedure", Technical Report TR-2005-114, Microsoft Research, 2005.

[19] N. Tew, P. Kalla, N. Shekhar, and S. Gopalakrishnan, "Verification of Arithmetic Datapaths using Polynomial Function Models and Congruence Solving", *in Proc. Intl. Conf. on Computer-Aided Design (ICCAD)*, pp. 122–128, 2008.

[20] A. Gupta, "Formal Hardware Verification Methods: A Survey", *Formal Methods in System Design*, vol. 1, pp. 151–238, 1992.

[21] J. Smith and G. DeMicheli, "Polynomial methods for component matching and verification", *in Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 1998.

[22] J. Smith and G. DeMicheli, "Polynomial Methods for Allocating Complex Components", *in IEEE Design, Automation and Test in Europe*, 1999.

[23] A. Peymandoust and G. DeMicheli, "Application of Symbolic Computer Algebra in High-Level Data-Flow Synthesis", *IEEE Transactions CAD*, vol. 22, pp. 1154–11656, 2003.

[24] T. Pruss, P. Kalla, and F. Enescu, "Word-Level Abstraction from Bit-Level Circuits using Gröbner Basis", *in International Workshop on Logic and Synthesis*, 2013.

[25] T. Pruss, P. Kalla, and F. Enescu, "Equivalence Verification of Large Galois Field Arithmetic Circuits using Word-Level Abstraction via Gröbner Bases", *in Design Automation Conference*, 2014.

[26] X. Sun, P. Kalla, T. Pruss, and F. Enescu, "Fquivalenceormal Verification of Sequential Galois Field Arithmetic Circuits using Algebraic Geometry", *in Design Automation for Test in Europe*, 2015.

[27] T. Pruss, P. Kalla, and F. Enescu, "Efficient Symbolic Computation for Word-Level Abstraction from Combinational Circuits for Verification over Galois Fields", *IEEE Transactions on CAD (in review)*, 2015.

[28] R. E. Bryant, "Graph Based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers*, vol. C-35, pp. 677–691, August 1986.

[29] Randal E. Bryant Karl S. Brace, Richard L. Rudell, "Efficient implementation of a bdd package", *in DAC, pp40-45*, 1990.

[30] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M.A. Perkowski, "Efficient Representation and Manipulation of Switching Functions based on Ordered Kronecker Functional Decision Diagrams", *in Design Automation Conference*, pp. 415–419, 1994.

[31] I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic Decision Diagrams and their Applications", *in Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 188–191, Nov. 93.

[32] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping", *in DAC*, pp. 54–60, 93.

[33] E. M. Clarke, M. Fujita, and X. Zhao, "Hybrid Decision Diagrams - Overcoming the Limitation of MTBDDs and BMDs", *in Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 159–163, 1995.

[34] Y-T. Lai, M. Pedram, and S. B. Vrudhula, "FGILP: An ILP Solver based on Function Graphs", *in ICCAD*, pp. 685–689, 93.

[35] R. E. Bryant and Y-A. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams", *in Proceedings of Design Automation Conference*, pp. 535–541, 1995.

[36] R. Dreschler, B. Becker, and S. Ruppertz, "The K*BMD: A Verification Data Structure", *IEEE Design & Test of Computers*, vol. 14, pp. 51–59, 1997.

[37] Y. A. Chen and R. E. Bryant, "*PHDD: An Efficient Graph Representation for Floating Point Verification", *in Proc. ICCAD*, 1997.

[38] M. Ciesielski, P. Kalla, and S. Askar, "Taylor Expansion Diagrams: A Canonical Representation for Verification of Data-Flow Designs", *IEEE Transactions on Computers*, vol. 55, pp. 1188–1201, 2006.

[39] N. Shekhar, *Equivalence Verification of Arithmetic Datapaths using Finite Ring Algebra*, PhD thesis, Univ. of Utah, Dept. of Electrical and Computer Engineering, Aug. 2007.

[40] B. Alizadeh and M. Fujita, "Modular Datapath Optimization and Verification based on Modular-HED", *IEEE Transactions CAD*, pp. 1422–1435, Sept. 2010.

[41] A. Jabir and Pradhan D., "MODD: A New Decision Diagram and Representation for Multiple Output Binary Functions", *in IEEE Design, Automation and Test in Europe*, 2004.

[42] A. Jabir, D. Pradhan, T. Rajaprabhu, and A. Singh, "A Technique for Representing Multiple Output Binary Functions with Applications to Verification and Simulation", *IEEE Transactions on Computers*, vol. 56, pp. 1133–1145, 2007.

[43] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vencentelli, F. Somenzi, A. Aziz, S-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, G. Shiple, S. Swamy, and T. Villa, "VIS: A System for Verification and Synthesis", *in Computer Aided Verification*, 1996.

[44] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[45] C. Barrett and C. Tinelli, "CVC3", *in Computer Aided Verification Conference*, pp. 298–302. Springer, July 2007.

[46] L. Moura and N. Bjrner, "Z3: An Efficient SMT Solver.", *in International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 4963. Springer, 2008.

[47] C. W. Barlett, D. L. Dill, and J. R. Levitt, "A Decision Procedure for bit-Vector Arithmetic", *in DAC*, June 1998.

[48] H. Enderton, *A mathematical introduction to logic*, Academic Press New York, 1972.

[49] T. Bultan and et al, "Verifying systems with integer constraints and boolean predicates: a composite approach", *in In Proc. Int'l. Symp. on Software Testing and Analysis*, 1998.

[50] S. Devadas, K. Keutzer, and A. Krishnakumar, "Design verification and reachability analysis using algebraic manipulation", *in Proc. ICCD*, 91.

[51] Z. Zhou and W. Burleson, "Equivalence Checking of Datapaths Based on Canonical Arithmetic Expressions", *in DAC*, 95.

[52] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard, "Difference decision diagrams", *in Computer Science Logic*, The IT University of Copenhagen, Denmark, Sep. 1999.

[53] Jesper Møller and Jakob Lichtenberg, "Difference decision diagrams", Master's thesis, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, Aug. 1998.

[54] K. Strehl, "Interval Diagrams: Increasing Efficiency of Symbolic Real-Time Verification", *in Intl. Conf. on Real Time Computing systems and Applications*, 1999.

[55] P. Sanchez and S. Dey, "Simulation-Based System-Level Verification using Polynomials", *in High-Level Design Validation & Test Workshop, HLDVT*, 1999.

[56] G. Ritter, "Formal Verification of Designs with Complex Control by Symbolic Simulation", in Springer Verlag LCNS, editor, *Advanced Research Working Conf. on Correct Hardware Design and Verification Methods (CHARME)*, 1999.

[57] F. Fallah, S. Devadas, and K. Keutzer, "Functional Vector Generation for HDL models using Linear Programming and 3-Satisfiability", *in Proc. DAC*, '98.

[58] R. Brinkmann and R. Drechsler, "RTL-Datapath Verification using Integer Linear Programming", *in Proc. ASP-DAC*, 2002.

[59] C.-Y. Huang and K.-T. Cheng, "Using Word-Level ATPG and Modular Arithmetic Constraint Solving Techniques for Assertion Property Checking", *IEEE Trans. CAD*, vol. 20, pp. 381–391, 2001.

[60] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, pp. 1377–1394, Nov. 2006.

[61] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to Combinational Equivalence Checking", *in Proc. Intl. Conf. on CAD (ICCAD)*, pp. 836–843, 2006.

[62] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool", *in Computer Aided Verification*, vol. 6174, pp. 24–40. Springer, 2010.

[63] E. Mastrovito, "VLSI Designs for Multiplication Over Finite Fields GF(2^m)", *Lecture Notes in Computer Science*, vol. 357, pp. 297–309, 1989.

[64] C. Koc and T. Acar, "Montgomery Multiplication in GF(2^k)", *Designs, Codes and Cryptography*, vol. 14, pp. 57–69, Apr. 1998.

[65] L. Erkök, M. Carlsson, and A. Wick, "Hardware/Software Co-verification of Cryptographic Algorithms using Cryptol", *in Proc. Formal Methods in CAD (FMCAD)*, pp. 188–191, 2009.

[66] M. Ciesielski, W. Brown, D. Liu, and A. Rossi, "Function Extraction from Arithmetic Bit-level Circuits", *ISVLSI*, 2014.

[67] S. Gao, "Counting Zeros over Finite Fields with Gröbner Bases", Master's thesis, Carnegie Mellon University, 2009.

[68] S. Gao, A. Platzer, and E. Clarke, "Quantifier Elimination over Finite Fields with Gröbner Bases", *in Intl. Conf. Algebraic Informatics*, 2011.

[69] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, "Non-linear Loop Invariant Generation using Grobner Bases", *SIGPLAN Not.*, vol. 39, pp. 318–329, 2004.

[70] L. Lastras-Montaño, P. Meany, E. Stephens, B. Trager, J. O'Conner, and L. Alves, "A new class of array codes for memory storage", *in Proc. Information Theory and Applications Workshop*, pp. 1–10, 2011.

[71] L. Lastras, A. Lvov, B. Trager, S. Winograd, V. Paruthi, A. El-Zhein, R. Shadowen, and G. Janssen, "New Formal Verification Techniques for Algorithms over Finite Fields", Presented at Intl. Workshop on Internation Theory and Applications. Abstract of the paper available at: http://ita.ucsd.edu/workshop/12/talks, 2012.

[72] A. Lvov, L. Lastras-Montaño, V. Paruthi, R. Shadowen, and A. El-Zein, "Formal Verification of Error Correcting Circuits using Computational Algebraic Geometry", *in Proc. Formal Methods in Computer-Aided Design (FMCAD)*, pp. 141–148, 2012.

[73] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann, "SINGULAR 3-1-3 — A computer algebra system for polynomial computations", 2011, http://www.singular.uni-kl.de.

[74] J. Lv, P. Kalla, and F. Enescu, "Efficient Grobner Basis Reductions for Formal Verification of Galois Field Arithmetic Circuits", *IEEE Transactions CAD*, vol. 32, pp. 1409–1420, Sept. 2013.

[75] J. Lv, P. Kalla, and F. Enescu, "Efficient Groebner Basis Reductions for Formal Verification of Galois Field Multipliers", *in IEEE Design, Automation and Test in Europe*, 2012.

[76] J. Lv, P. Kalla, and F. Enescu, "Verification of Composite Galois Field Multipliers over GF($(2^m)^n$) using Computer Algebra Techniques", *in IEEE High-Level Design Validation and Test Workshop*, pp. 136–143, 2011.

[77] J. Lv, P. Kalla, and F. Enescu, "Formal Verification of Galois Field Multipliers using Computer Algebra", *in 25th IEEE International Conference on VLSI Design*, 2012.

[78] O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G. Gruel, "An Algebraic Approach to Proving Data Correctness in Arithmetic Datapaths", *in Computer Aided Verification Conference*, pp. 473–486, 2008.

[79] N. Shekhar, P. Kalla, and F Enescu, "Equivalence Verification of Polynomial Datapaths using Ideal Membership Testing", *IEEE Transactions on CAD*, vol. 26, pp. 1320–1330, July 2007.

[80] E. Pavlenko, M. Wedler, D. Stoffel, W. Kunz, A. Dreyer, F. Seelisch, and G.-M. Greuel, "STABLE: A New QBF-BV SMT Solver for Hard Verification Problems Combining Boolean Reasoning with Computer Algebra", *in IEEE Design, Automation and Test in Europe Conference*, pp. 155–160, 2011.

[81] R. Zippel, "Probabilistic algorithms for sparse interpolation", *in Proc. Symp. Symbolic and Algebraic Computation*, pp. 216–226, 1979.

[82] M. Ben-Or and P. Tiwari, "A deterministic algorithm for sparse multivariate polynomial interpolation", *in Proc. Symp. Theory of Computing*, pp. 301–309, 1988.

[83] S. Javadi and M. Monagan, "On sparse polynomial interpolation over finite fields", *in Intl. Symp. Symbolic and Algebraic Computing*, 2010.

[84] Z. Zilic and Z. Vranesic, "A deterministic multivariate interpolation algorithm for small finite fields", *IEEE Trans. Computers*, vol. 51, Sept. 2002.

[85] Robert J. McEliece, *Finite Fields for Computer Scientists and Engineers*, Kluwer Academic Publishers, 1987.

[86] S. Roman, *Field Theory*, Springer, 2006.

[87] R. Lidl and H. Niederreiter, *Finite Fields*, Cambridge University Press, 1997.

[88] P. Montgomery, "Modular Multiplication Without Trial Division", *Mathematics of Computation*, vol. 44, pp. 519–521, Apr. 1985.

[89] H. Wu, "Montgomery Multiplier and Squarer for a Class of Finite Fields", *IEEE Transactions On Computers*, vol. 51, May 2002.

[90] M. Knežević, K. Sakiyama, J. Fan, and I. Verbauwhede, "Modular Reduction in GF(2^n) Without Pre-Computational Phase", *in Proceedings of the International Workshop on Arithmetic of Finite Fields*, pp. 77–87, 2008.

[91] D. Singmaster, "On Polynomial Functions (mod m)", *J. Number Theory*, vol. 6, pp. 345–352, 1974.

[92] Z. Chen, "On polynomial functions from Z_n to Z_m", *Discrete Math.*, vol. 137, pp. 137–145, 1995.

[93] Z. Chen, "On polynomial functions from $\mathrm{Z}_{n_1} \times \mathrm{Z}_{n_2} \times \cdots \times \mathrm{Z}_{n_r}$ to Z_m", *Discrete Math.*, vol. 162, pp. 67–76, 1996.

[94] ST Microelectronics, *ST23YLxx series Microcontroller for Smart Cards*.

[95] K. Kobayashi, *Studies on Hardware Assisted Implementation of Arithmetic Operations in Galois Field*, PhD thesis, Nagoya University, Japan, 2009.

[96] S. Morioka and Y. Katayama, "Design methodology for a one-shot reed-solomon encoder and decoder", *in IEEE International Conference on Computer Design*, pp. 60–67, 1999.

[97] Y. Lee, K. Sakiyama, L. Batina, and I. Verbauwhede, "Elliptic-Curve-Based Security Processor for RFID", *IEEE Transactions on Computers*, vol. 57, pp. 1514–1527, Nov. 2008.

[98] Darrel Hankerson, Julio Hernandez, and Alfred Menezes, "Software Implementation of Elliptic Curve Cryptography over Binary Fields", in CetinK. Koc and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems CHES 2000*, vol. 1965 of *Lecture Notes in Computer Science*, pp. 1–24. Springer Berlin Heidelberg, 2000.

[99] V. Miller, "Use of Elliptic Curves in Cryptography", *in Lecture Notes in Computer Sciences*, pp. 417–426, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[100] B. A. Forouzan, *Cryptography and Network Security.*, A. R. Apt, 2008.

[101] J. López and R. Dahab, "Improved Algorithms for Elliptic Curve Arithmetic in GF(2^n)", *in Proceedings of the Selected Areas in Cryptography*, pp. 201–212, London, UK, UK, 1999. Springer-Verlag.

[102] D. Cox, J. Little, and D. O'Shea, *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*, Springer, 2007.

[103] B. Buchberger, *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*, PhD thesis, University of Innsbruck, 1965.

[104] B. Buchberger, "A criterion for detecting unnecessary reductions in the construction of a groebner bases", *in EUROSAM*, 1979.

[105] J-C. Faugẽre, "A New Efficient Algorithm for Computing Gröbner Bases (F_4)", *Journal of Pure and Applied Algebra*, vol. 139, pp. 61–88, June 1999.

[106] B. Sunar, E. Savas, and C. Ko, "Constructing Composite Field Representations for Efficient Conversion", *IEEE Transactions on Computers*, vol. 52, pp. 1391–1398, November 2003.

[107] C. Paar, *Efficient VLSI Architecture for Bit-Parallel Computation in Galois Fields*, PhD thesis, University of Essen, Germany, 1994.

[108] McCutchen M., "C++ big integer library", https://mattmccutchen.net/bigint/, 2010.

[109] Z. Feng, Z. Zeng, and P. Li, "Parallel On-Chip Power Distribution Network Analysis on Multicore GPU Platforms", *IEEE Transactions VLSI*, 2011.

[110] Michael Brickenstein and Alexander Dreyer, "Polybori: A Framework for Gröbner Basis Computations with Boolean Polynomials", *Journal of Symbolic Computation*, vol. 44, pp. 1326–1345, September 2009.