Formal Analysis of Galois Field Arithmetics – Parallel Verification and Reverse Engineering

Article *in* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems · February 2018

DOI: 10.1109/TCAD.2018.2808457

2 authors, including:

Maciej J. Ciesielski
University of Massachusetts Amherst
148 PUBLICATIONS **1,785** CITATIONS

Formal Analysis of Galois Field Arithmetic Circuits

- Parallel Verification and Reverse Engineering

Cunxi Yu *Student Member, IEEE,* and Maciej Ciesielski, *Senior Member, IEEE*

***Abstract*—Galois field (GF) arithmetic circuits find numerous applications in communications, signal processing, and security engineering. Formal verification techniques of GF circuits are scarce and limited to circuits with known bit positions of the primary inputs and outputs. They also require knowledge of the irreducible polynomial $P(x)$, which affects final hardware implementation. This paper presents a computer algebra technique that performs verification and reverse engineering of GF(2^m) multipliers directly from the gate-level implementation. The approach is based on extracting a unique irreducible polynomial in a parallel fashion and proceeds in three steps: 1) determine the bit position of the output bits; 2) determine the bit position of the input bits; and 3) extract the irreducible polynomial used in the design. We demonstrate that this method is able to reverse engineer GF(2^m) multipliers in *m* threads. Experiments performed on synthesized *Mastrovito* and *Montgomery* multipliers with different $P(x)$, including NIST-recommended polynomials, demonstrate high efficiency of the proposed method.**

***Index Terms*—Galois field arithmetic, computer algebra, formal verification, reverse engineering, parallelism.**

I. Introduction

DESPITE considerable progress in verification of random and control logic, advances in formal verification of arithmetic circuits have been lagging. This can be attributed to the difficulty in efficient modeling of arithmetic circuits and datapaths, without resorting to computationally expensive Boolean methods. Contemporary formal techniques, such as *Binary Decision Diagrams* (BDDs), *Boolean Satisfiability* (SAT), *Satisfiability Modulo Theories* (SMT), etc., are not directly applicable to verification of integer and finite field arithmetic circuits [1] [2]. This paper concentrates on formal verification and reverse engineering of finite (Galois) field arithmetic circuits.

Galois field (GF) is a number system with a finite number of elements and two main arithmetic operations, addition and multiplication; other operations can be derived from those two [3]. GF arithmetic plays an important role in coding theory, cryptography, and their numerous applications. Therefore, developing formal techniques for hardware implementations of GF arithmetic circuits, and particularly for finite field multiplication, is essential.

The elements in field GF(2^m) can be represented using polynomial rings. The field of size m is constructed using *irreducible polynomial* $P(x)$, which includes terms of degree with $d \in [0, m]$ with coefficients in GF(2). The arithmetic operation in the field is then performed modulo $P(x)$. The choice of the irreducible polynomial has a significant impact on the hardware implementation of the GF circuit and its performance. Typically, the irreducible polynomial with a minimum number of elements gives the best performance [4], but it is not always the case.

Due to the rising number of threats in hardware security, analyzing finite field circuits becomes important. Computer algebra techniques with polynomial representation seem to offer the best solution for analyzing arithmetic circuits. Several works address the verification and functional abstraction problems, both in Galois field arithmetic [1] [5] [6] and integer arithmetic implementations [7] [2] [8] [9] [10]. Symbolic computer algebra methods have also been used to reverse engineer the word-level operations for GF circuits and integer arithmetic circuits to improve verification performance [11] [12] [5]. The verification problem is typically formulated as proving that the implementation satisfies the specification, and is accomplished by performing a series of divisions of the specification polynomial by the implementation polynomials. In the work of Yu *et al.* [11], the authors proposed an original spectral method based on analyzing the internal algebraic expressions during the rewriting procedure. Sayed-Ahmed et al. [12] introduced a reverse engineering technique in Algebraic Combinational Equivalence Checking (ACEC) process by converting the function into canonical polynomials and using *Gröbner Basis*.

However, the above mentioned algebraic techniques have several limitations. Firstly, they are restricted to implementations with a known binary encoding of the inputs and outputs. This information is needed to generate the specification polynomial that describes the circuit functionality regarding its inputs and outputs, necessary for the polynomial reduction process (described in Section II-D). Secondly, these methods are unable to explore parallelism (inherent in GF circuits), as they require that the polynomial division is applied iteratively using reverse-topological order [2] [9] [6]. Thirdly, the approaches applied specifically to GF(2^m) arithmetic circuits [5] [6], require knowledge of the irreducible polynomial $P(x)$ of the circuit.

In this work, we present a formal approach to reverse engineer the gate-level finite field arithmetic circuits that exploit inherent parallelism of the GF circuits. The method is based on a parallel algebraic rewriting approach [13] and applied specifically to multipliers. The objective of reverse engineering is as follows: given the netlist of a gate-level GF multiplier, extract the bit positions of input and output

C. Yu and M. Ciesielski are with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA, 01375. The related tools and benchmarks are released publicly on Github, *ycunxi.github.io/Parallel_Formal_Analysis_GaloisField*
E-mail: ycunxi@umass.edu

bits and the irreducible polynomial used in constructing the GF multiplication; then extract the specification of the design using this information. Bit position i indicates the location of the bit in the binary word according to its significance (LSB vs MSB). Our approach solves this problem by transforming the algebraic expressions of the output bits into an algebraic expression of the input bits (specification), and is done in parallel for each output bit. Specifically, it includes the following steps[1]:

- Extract the algebraic expression of each output bit.
- Determine the bit position of the outputs.
- Determine the bit position of the inputs.
- Extract the irreducible polynomial $P(x)$.
- Extract the specification by algebraic rewriting.

We demonstrate the efficiency of our method using GF(2^m) *Mastrovito* and *Montgomery* multipliers of up to 571-bit width in a bit-blasted format (i.e., flattened to bit-level), implemented using various irreducible polynomials.

II. Background

A. Canonical Diagrams

Several approaches have been proposed to check an arithmetic circuit against its functional specification. Different variants of canonical, graph-based representations have been proposed, including Binary Decision Diagrams (BDDs) [14], Binary Moment Diagrams (BMDs) [15] [16], Taylor Expansion Diagrams (TED) [17], and other hybrid diagrams. While BDDs have been used extensively in logic synthesis, their application to verification of arithmetic circuits is limited by the prohibitively high memory requirement for complex arithmetic circuits, such as multipliers. BDDs are being used, along with many other methods, for local reasoning, but not as monolithic data structure [18]. BMDs and TEDs offer a better space complexity but require word-level information of the design, which is often not available or is hard to extract from bit-level netlists. While the canonical diagrams have been used extensively in logic synthesis, high-level synthesis, and verification, their application to verify large arithmetic circuits remains limited by the prohibitively high memory requirement of complex arithmetic circuits [2] [1].

B. SAT, ILP and SMT Solvers

Arithmetic verification problems have been typically modeled using Boolean satisfiability (SAT). Several SAT solvers have been developed to solve Boolean decision problems, including ABC [19], MiniSAT [20], and others. Some of them, such as CryptoMiniSAT [21], target specifically XOR-rich circuits, and are potentially useful for arithmetic circuit verification, but are all based on a computationally expensive DPLL (Davis, Putnam, Logemann, Loveland) decision procedure [22]. Some techniques combine automatic test pattern generation (ATPG) and modular arithmetic constraint-solving techniques for the purpose of test generation and assertion checking [23]. Others integrate linear arithmetic constraints with Boolean SAT in a unified algebraic domain [24], but their effectiveness is limited by constraint propagation across the Boolean and word-level boundary. To avoid this problem, methods based on ILP models of arithmetic operators have been proposed [25] [26], but in general ILP techniques are known to be computationally expensive and not scalable to large scale systems. *SMT solvers* depart from treating the problem in a strictly Boolean domain and integrate different well-defined theories (Boolean logic, bit vectors, integer arithmetic, etc.) into a DPLL-style SAT decision procedure [27]. Some of the most effective SMT solvers, potentially applicable to our problem, are Boolector [28], Z3 [29], and CVC [30]. However, SMT solvers still model functional verification as a decision problem and, as demonstrated by extensive experimental results, neither SAT nor SMT solvers can efficiently solve the verification problem of large arithmetic circuits [1] [10].

C. Theorem Provers

Another class of solvers include Theorem Provers, deductive systems for proving that an implementation satisfies the specification, using mathematical reasoning. The proof system is based on a large and strongly problem-specific database of axioms and inference rules, such as simplification, term rewriting, induction, etc. Some of the most popular theorem proving systems are: HOL [31], PVS [32], ACL2 [33], and the term rewriting method described in [34]. These systems are characterized by high abstraction and powerful logic expressiveness, but they are highly interactive, require intimate domain knowledge, extensive user guidance, and expertise for efficient use. The success of verification using theorem provers depends on the set of available axioms and rewrite rules, and on the choice and order in which the rules are applied during the proof process, with no guarantee for a conclusive answer [35].

D. Computer Algebra Approaches

The most advanced techniques that have potential to solve the arithmetic verification problems are those based on Symbolic Computer Algebra. The verification problem is typically formulated as a proof that the implementation satisfies the specification [2] [1] [8] [7] [9]. This is accomplished by performing a series of divisions of the specification polynomial F by a set of implementation polynomials B, representing circuit components, the process referred to as reduction of F modulo B. Polynomials $f_1, ..., f_s \in B$ are called the bases, or *generators*, of the ideal J. Given a set $f_1, ..., f_s$ of generators of J, a set of all simultaneous solutions to a system of equations f_1=0; ... ,f_s=0 is called a *variety* $V(J)$. Verification problem is then formulated as a test if the specification F vanishes on $V(J)$, i.e., if $F \in V(J)$. This is known in computer algebra as *ideal membership* testing problem [1].

Standard procedure to test if $F \in V(J)$ is to divide polynomial F by the polynomials $\{f_1, ..., f_s\}$ of B, one by one. The goal is to cancel, at each iteration, the leading term of F using one of the leading terms of $f_1, ..., f_s$. If the remainder r of the division is 0, then F vanishes on $V(J)$, proving

[1]Our tool and benchmarks used in this journal paper are released publicly at our project website at https://ycunxi.github.io/Parallel_Formal_Analysis_GaloisField

that the implementation satisfies the specification. However, if $r \neq 0$, such a conclusion cannot be made; B may not be sufficient to reduce F to 0, and yet the circuit may be correct. To reliably check if F is reducible to zero, a *canonical* set of generators, $G = \{g_1, ..., g_t\}$, called *Gröbner basis*, is needed. It has been shown that for combinational circuits with no feedback, certain conditions automatically make the set B a Groebner basis [36]. Specifically, if the polynomials $f_1, ..., f_s \in B$ are ordered in reverse topological order of logic gates, from primary outputs to primary inputs, and the leading term of each polynomial is the output of a logic gate, then set B is automatically a Groebner basis. Some of the authors use Gaussian elimination, rather than explicit polynomial division, to speed up the polynomial reduction process [1] [8]. The polynomials corresponding to fanout-free logic cones can be precomputed to reduce the size of the problem [8].

The polynomial reduction technique has been successfully applied to both integer arithmetic circuits [9] and Galois field arithmetic [1]. Verification work of Galois field arithmetic has been presented in [1] [5]. Formulation of problems in GF arithmetic takes advantage of known properties of Galois field during polynomial reductions. Specifically, the problem reduces to the ideal membership testing over a larger ideal that includes ideal $J_0 = \langle x^2 - x \rangle$ in $\mathbb{F}_2$, for each internal signal x of the circuit. Inclusion of this ideal basically assures that each signal assumes a binary value. In this paper, we provide comparison between this technique and our approach.

E. Function Extraction

Function extraction is an arithmetic verification method originally proposed in [2] for arithmetic circuits in modular integer arithmetic $\mathbb{Z}_{2^m}$. It extracts a unique bit-level polynomial function implemented by the circuit directly from its gate-level implementation. Instead of expensive polynomial division, extraction is done by *backward rewriting*, i.e., transforming the polynomial representing encoding of the primary outputs (called the *output signature*) into a polynomial at the primary inputs (the *input signature*) using algebraic models of the logic gates of the circuit. That is, the rewriting is performed in a reverse topological order. This technique has been successfully applied to large integer arithmetic circuits, such as 512-bit integer multipliers. However, it is not directly applicable to large Galois Field multipliers because of potentially exponential number of polynomial terms, before the internal term cancellations takes place during rewriting. Fortunately, arithmetic GF(2^m) circuits offer an inherent parallelism which can be exploited in backward rewriting, without memory explosion.

In the rest of the paper, we first describe how to apply such parallel rewriting in GF(2^m) circuits while avoiding memory explosion experienced in integer arithmetic circuits. Using this approach, we extract the function of each output bit in $\mathbb{F}_{2^m}$ and the function is represented in a *pseudo-Boolean polynomial* expression, where all variables are Boolean. Finally, we propose a method to reverse engineer the GF(2^m) designs by analyzing these expressions.

III. Galois Field Multiplication

Galois field (GF) is a number system with a finite number of elements and two main arithmetic operations, addition and multiplication; other operations such as division can be derived from those two [3]. Galois field with p elements is denoted as GF(p). The most widely-used finite fields are *Prime Fields* and *Extension Fields*, and particularly *Binary Extension Fields*. Prime field, denoted GF(p), is a finite field consisting of finite number of integers $\{1, 2, ..., p-1\}$, where p is a prime number, with additions and multiplication performed *modulo p*. Binary extension field, denoted GF(2^m) (or $\mathbb{F}_{2^m}$), is a finite field with 2^m elements. Unlike in prime fields, however, the operations in extension fields are not computed *modulo* 2^m. Instead, in one possible representation (called *polynomial basis*), each element of GF(2^m) is a *polynomial ring* with m terms with coefficients in GF(2), modulo $P(x)$. Addition of field elements is the usual addition of polynomials, with coefficient arithmetic performed modulo 2. Multiplication of field elements is performed modulo *irreducible polynomial* $P(x)$ of degree m and coefficients in GF(2). The irreducible polynomial $P(x)$ is analogous to the prime number p in prime fields $GF(p)$. In this work, we focus on the verification problem of GF(2^m) multipliers that appear in many cryptography and in some DSP applications.

A. GF Multiplication Principle

Two different GF multiplication structures, constructed using different irreducible polynomials $P_1(x)$ and $P_2(x)$, are shown in Figure 1. The integer multiplication takes two n-bit operands as input and generates a $2n$-bit word, where the values computed at lower significant bits ripple through the carry chain all the way to the most significant bit (MSB). In contrast, in GF(2^m) implementations the number of outputs is reduced to n using irreducible polynomial P(x). The product terms are added for each column (output bit position) modulo 2, hence there is no carry propagation. For example, to represent the result in GF(2^4), with only four output bits, the four most significant bits in the result of the integer multiplication have to be reduced to GF(2^4). The result of such a reduction is shown in Figure 1. In GF(2^4), the input and output operands are represented using polynomials $A(x)$, $B(x)$ and $Z(x)$, where $A(x)$=$\sum_{n=0}^{n=3} a_n \cdot x^n$, $B(x)$=$\sum_{n=0}^{n=3} b_n \cdot x^n$, and $Z(x)$=$\sum_{n=0}^{n=3} z_n \cdot x^n$, respectively.

Example 1: The function of each multiplication bit s_i ($i \in$ [0, 6]) is represented using polynomials in GF(2), namely: s_0=a_0b_0, s_1=a_1b_0+a_0b_1, etc. ..., up to s_6=a_3b_3[2]. The output bits z_n ($n \in$ [0, 3]) are computed modulo the irreducible polynomial $P(x)$. Using $P_2(x)$=x^4+x+1, we obtain : z_0=s_0+s_4, z_1=s_1+s_4+s_5, z_2=a_0b_2+a_1b_1+a_2b_0+a_2b_3+a_3b_2+a_3b_3, and z_3=a_0b_3+a_1b_2+a_2b_1+a_3b_0+a_3b_3. The coefficients of the multiplication results are shown in Figure 2. In digital circuits, partial products are implemented using AND gates, and addition modulo 2 is done using XOR gates. Note that, unlike in integer multiplication, in GF(2^m) circuits there is

[2]For polynomials in GF(2), "+" are computed as modulo 2.

no carry out to the next bit. For this reason, as we can see in Figure 1, the function of each output bit can be computed independently of other bits.

			a_3	a_2	a_1	a_0
			b_3	b_2	b_1	b_0
			a_3b_0	a_2b_0	a_1b_0	a_0b_0
		a_3b_1	a_2b_1	a_1b_1	a_0b_1	
	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
a_3b_3	a_2b_3	a_1b_3	a_0b_3			
s_6	s_5	s_4	s_3	s_2	s_1	s_0

$P_1(x)$=$x^4 + x^3 + 1$

s_3	s_2	s_1	s_0
s_4	0	0	s_4
s_5	0	s_5	s_5
s_6	s_6	s_6	s_6
z_3	z_2	z_1	z_0

$P_2(x)$=$x^4 + x + 1$

s_3	s_2	s_1	s_0
0	0	s_4	s_4
0	s_5	s_5	0
s_6	s_6	0	0
z_3	z_2	z_1	z_0

Figure 1: Two GF(2^4) multiplications constructed using $P_1(x)$=x^4+x^3+1 and $P_2(x)$=x^4+x+1.

output	polynomial expression
z_0	(a_0b_0)+a_1b_3+a_2b_2+a_3b_1
z_1	$(a_0b_1$+$a_1b_0)$+a_1b_3+a_2b_2+a_2b_3+a_3b_1+a_3b_2
z_2	$(a_0b_2$+a_1b_1+$a_2b_0)$+a_2b_3+a_3b_2+a_3b_3
z_3	$(a_0b_3$+a_1b_2+a_2b_1+$a_3b_0)$+a_3b_3

Figure 2: Extracted algebraic expressions of the four output bits of GF(2^4) multiplier for $P(x) = x^4 + x + 1$.

B. Irreducible Polynomials

In general, there are various irreducible polynomials that can be used for a given field size, each resulting in a different multiplication result. For constructing efficient arithmetic functions over GF(2^m), the irreducible polynomial is typically chosen to be a trinomial, x^m+x^a+1, or a pentanomial x^m+x^a+x^b+x^c+1 [37]. For efficiency reason, coefficients m, a are chosen such that m - $a \geq m/2$.

An example of constructing GF(2^4) multiplication using two different irreducible polynomials is shown in Figure 1. We can see that each polynomial produces a unique multiplication result. The size of the corresponding multiplier can be estimated by counting the number of XOR operations in each multiplication. Since the number of AND and XOR operations for generating partial products (variables s_i in Figure 1) is the same, the difference is only caused by the reduction of the corresponding polynomials modulo $P(x)$. The number of two-input XOR operations introduced by the reduction with $P(x)$ can be obtained as the number of terms in each column minus one. For example, the number of XORs using $P_1(x)$ is 3+1+2+3=9; and using $P_2(x)$, the number of XORs is 1+2+2+1=6.

As will be shown in the next section, given the structure of the GF(2^m) multiplication, such as the one shown in Figure 1, one can readily identify the irreducible polynomial $P(x)$ used during the GF reduction. This can be done by extracting the terms s_k corresponding to the entry s_m (here s_4) in the table and generating the irreducible polynomial beyond x^m. We know that $P(x)$ must contain x_m, and the remaining terms x^k of $P(x)$ are obtained from the non-zero terms corresponding to the entry s_m. For example, for the irreducible polynomial $P_1(x) = x^4 + x^3 + x^0$, the terms x^3 and x^0 are obtained by noticing the placement of s_4 in columns z_3 and z_0. Similarly, for $P_2(x) = x^4 + x^1 + x^0$, the terms x^1 and x^0 are obtained by noticing that s_4 is placed in columns z_1 and z_0. The reason for it and the details of this procedure will be explained in the next section.

IV. Parallel Extraction in Galois Field

In this section, we introduce our method for extracting the unique algebraic expressions of the output bits (e.g. Figure 2) using computer algebraic method. This can be used to verify the GF(2^m) multipliers when the binary encoding of inputs and output and the irreducible polynomial are given. We introduce a parallel function extraction framework in GF(2^m), which allows us to individually extract the algebraic expression of each output bit. This framework is used for reverse engineering, since our reverse engineering approach is based on analyzing the algebraic expression of output bits in GF(2), as introduced in Section I.

A. Computer Algebraic model

The circuit is modeled as a network of logic elements of arbitrary complexity, including basic logic gates (AND, OR, XOR, INV) and complex standard cell gates (AOI, OAI, etc.) generated by logic synthesis and technology mapping. We extend the algebraic model of Boolean operators developed in [10] for integer arithmetic to finite field arithmetic in $GF(2)$, i.e., modulo 2. For example, the pseudo-Boolean model of XOR(a, b)=$a + b - 2ab$ is reduced to $(a + b + 2ab)$ mod 2 = $(a + b)$ mod 2. The following algebraic equations are used to describe basic logic gates in $GF(2^m)$ [1]:

$$\begin{aligned} &\neg a = 1 + a \\ &a \wedge b = a \cdot b \\ &a \vee b = a + b + a \cdot b \\ &a \oplus b = a + b \end{aligned} \tag{1}$$

B. Outline of the Approach

Similarly to the work of [2] and [10], the arithmetic function computed by the circuits is obtained by transforming (rewriting) the polynomial representing the encoding of the primary outputs (called *output signature*) into the polynomial at the primary inputs, the *input signature*. The output signature of a $GF(2^m)$ multiplier, $Sig_{out} = \sum_{i=0}^{m-1} z_i x^i$, with $z_i \in GF(2)$. The input signature of a $GF(2^m)$ multiplier, Sig_{in} = $\sum_{i=0}^{m-1} \mathbb{P}_i x^i$, with coefficients $\mathbb{P}_i \in GF(2)$ being product terms, and addition operation performed modulo 2. If the irreducible polynomial $P(x)$ is provided, Sig_{in} is know; otherwise, it will be computed by backward rewriting from Sig_{out}. The goal is to transform the output signature, Sig_{out}, using polynomial representation of the internal logic

elements (1), into an input signature Sig_{in} in $GF(2^m)$, which determines the arithmetic function (specification) computed by the circuit.

Theorem 1: *Given a combinational arithmetic circuit in $GF(2^m)$, composed of logic gates, described by Eq. 1, input signature Sig_{in} computed by backward rewriting is unique and correctly represents the function implemented by the circuit in $GF(2^m)$.*

Proof: The proof of correctness relies on the fact that each transformation step (rewriting iteration) is correct. That is, each internal signal is represented by an algebraic expression, which always evaluates to a *correct value* in $GF(2^m)$. This is guaranteed by the correctness of the algebraic model in Eq. (1), which can be proved easily by inspection. For example, the algebraic expression of *XOR(a,b)* in $\mathbb{Z}_{2^m}$ is $a+b-2ab$. When implemented in $GF(2^m)$, the coefficients in the expression must be in $GF(2)$, hence *XOR(a,b)* in $GF2^m$ is represented by $a+b$. The proof of uniqueness is done by induction on i, the step of transforming polynomial F_i into F_{i+1}. A detailed induction proof of this theorem is provided in [2] for expressions in $\mathbb{Z}_{2^m}$.

□

Algorithm 1 Backward Rewriting in $GF(2^m)$

Input: Gate-level netlist of $GF(2^m)$ multiplier
Input: Output signature Sig_{out}, and (optionally) input signature, Sig_{in}
Output: GF function of the design; return Sig_{out}==Sig_{in}

1: $\mathcal{P}$={$p_0, p_1, ..., p_n$}: polynomials representing gate-level netlist
2: F_0=Sig_{out}
3: **for** each polynomial $p_i \in \mathcal{P}$ **do**
4: **for** output variable v of p_i in F_i **do**
5: replace every variable v in F_i by algebraic expression of p_i
6: $F_i \rightarrow F_{i+1}$
7: **for** each monomial M in F_{i+1} **do**
8: **if** the coefficient of M%2==0
9: or M is a constant, M%2==0 **then**
10: remove M from F_{i+1}
11: **end if**
12: **end for**
13: **end for**
14: **end for**
15: **return** F_n and F_n =?Sig_{in}

Theorem 1, together with the algebraic model of Boolean gates (1), provide the basis for polynomial reduction using backward rewriting. This is described by Algorithm 1. The method takes the gate-level netlist of a GF(2^m) multiplier as input and first converts each logic gate into an algebraic expression using Eq. (1). The rewriting process starts with the output signature $F_0 = Sig_{out}$ and performs rewriting in reverse topological order, from outputs to inputs. It ends when all the variables in F_i are primary inputs, at which point it becomes the input signature Sig_{in} [2].

Each iteration includes two basic steps: 1) substitute the variable of the gate output using the expression in the inputs of the gate (Eq.1), and name the new expression F_{i+1} (lines 3 - 6); and 2) simplify the new expression in two ways: a) by eliminating terms that cancel each other (as in the integer arithmetic case [2]), and b) by removing all the monomials (including constants) that reduce to 0 in GF(2) (line 3 and lines 7 - 10). The algorithm outputs the arithmetic function of the design in GF(2^m) after n iterations, where n is the number of gates in the netlist. The final expression $F_n = Sig_{in}$

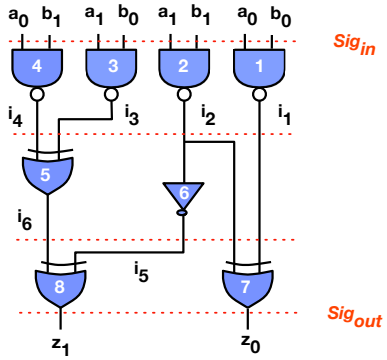

Figure 3: The gate-level netlist of post-synthesized and mapped 2-bit multiplier over GF(2^2). The irreducible polynomial is $P(x) = x^2 + x + 1$.

Sig_{out}: F_{init}=z_0+xz_1	Eliminating terms
G8: F_8=z_0+x(i_5+i_6)	-
G7: F_7=i_1+i_2+x(i_5+i_6)	-
G6: F_6=i_1+i_2+x(i_3+i_4+i_5)	-
G5: F_5=i_1+i_2+x(i_3+i_4+i_2+1)	-
G4: F_4=i_1+i_2+x(i_2+i_3+a_0b_1)+2x	$2x$
G3: F_3=i_1+i_2+x(i_2+a_1b_0+a_0b_1+1)	-
G2: F_2=i_1+a_1b_1+1+x(a_1b_1+a_1b_0+a_0b_1)+2x	$2x$
G1: F_1=a_0b_0+a_1b_1+2+x(a_1b_1+a_1b_0+a_0b_1)	2
Sig_{in}: a_0b_0+a_1b_1+x(a_1b_1+a_1b_0+a_0b_1)	-

Figure 4: Function extraction of a 2-bit GF multiplier shown in Figure 3 using backward rewiring from PO to PI.

can be used to verify if the circuit performs the desired arithmetic function by checking if the computed polynomial Sig_{in} matches the expected specification, if known. This equivalence check can be readily performed using canonical word-level representations, such as BMD [15] or TED [17] which can efficiently check equivalence of two polynomials. Alternatively, if the specification is not known, the computed signature can serve as the specification extracted from the circuit.

Example 2 (Figure 3): We illustrate our method using a post-synthesized 2-bit multiplier in $GF(2^2)$, shown in Figure 3. The irreducible polynomial is $P(x) = x^2 + x + 1$. The output signature is $Sig_{out} = z_0$+z_1x, and input signature is $Sig_{in} = (a_0b_0$+$a_1b_1)$+$(a_1b_1$+a_1b_0+$a_0b_1)x$. First, $F_{init} = Sig_{out}$ is transformed into F_8 using polynomial of gate $g8$, z_1=$i_5 + i_6$ and simplified to $F_8 = z_0 + i_5x + i_6x$. Then, the polynomials F_i are successively derived from F_{i+1} and checked for a possible reduction. The first reduction happens when F_5 is transformed into F_4, where i_4 (at gate g_4) is replaced by $(1 + a_0b_0)$. After simplification, a monomial $2x$ is identified and removed by modulo 2 from F_4. Similar reductions are applied during the transformations $F_3 \rightarrow F_2$ and $F_2 \rightarrow F_1$. Finally, the function of the design is extracted as expression F_1. A complete rewriting process is shown in Figure 4. We can see that $F_1 = Sig_{in}$, which indicates that the circuit indeed implements the $GF(2^2)$ multiplication with $P(x)$=$x^2 + x + 1$.

An important observation is that the potential reductions

take place only within the expression associated with the same degree of polynomial ring (Sig_{out}). In other words, the reductions happen in a logic cone of every output bit *independently* of other bits, regardless of logic sharing between the cones. For example, the reductions in F_4 and F_2 happen within the logic cone of output z_1 only. Similarly, in F_1, the reduction is within logic cone of z_0. Details of the proof are provided in [13].

C. Implementation

This section describes the implementation of our parallel verification method for Galois field multipliers. Our approach takes the gate-level netlist as input, and outputs the extracted function of the design. It includes four steps:

Step1: Convert netlist to equations. Parse the gate-level netlist into algebraic equations based on Equation 1. The equations are listed in topological order, to be rewritten by backward rewriting in the next step. m copies of this equation file will be made for a GF(2^m) multiplier.

Step2: Generate signatures. Split the output signature of GF(2^m) multipliers into m polynomials, with Sig_{out_i}=z_i. Insert the new *signatures* into the m copies of the equation file generated from Step1. Each signature represents a single output bit.

Step3: Parallel extraction. Apply Algorithm 1 to each equation file to extract the polynomial expression of each output in parallel. In contrast to work on integer arithmetic [2], the internal expression of each output bit does not offer any polynomial reduction (*monomial cancellations*) with other bits. Ideally, our approach can extract GF(2^m) multiplier in m threads. However, due to the limited computing resources, it is impossible to extract GF(2^m) multipliers in m threads when m is very large. Hence, our approach puts a limit on the number of parallel threads T (T = 5, 10, 20 and 30 have been tested in this work). This process is illustrated in Figure 5. The m extraction tasks are organized into several task sets, ordered from LSB to MSB. In each set, the extractions are performed in parallel. Since the runtime of each extraction within the set can differ, the tasks in the next set will start as soon as any previous task terminated.

Step4: Finalization. Compute the final function of the multiplier. Once the algebraic expression of each output bit in GF(2) is computed, our method computes the final function by constructing the Sig_{out} using the rewriting process in step 3.

Our algorithm uses a data structure that efficiently implements iterative substitution and elimination during backward rewriting. It is similar to the data structure employed in function extraction for integer arithmetic circuits [2], suitably modified to support simplifications in finite fields algebra. Specifically, in addition to cancellation of terms with opposite signs, it performs modulo 2 reduction of monomials and constants. The data structure maintains the record of the terms (monomials) in the expression that contain the variable to be substituted. It reduces the cost of finding the terms that will have their coefficients changed during substitution. Each element represents one monomial consisting of the variables

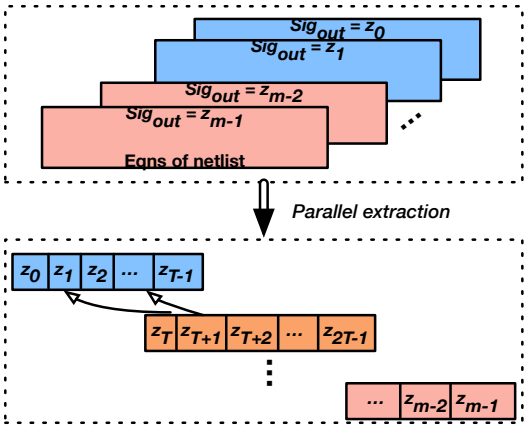

Figure 5: Step3: parallel extraction of a GF(2^m) multiplier with T threads.

in the monomials and its coefficient. The expression data structure is a C++ object that represents a pseudo-Boolean expression, which contains of all the elements in the data structure. It supports both fast addition and fast substitution with two C++ maps, implemented as binary search trees, a terms map, and a substitution map. This data structure includes two cases of simplifications: 1) after substitution the coefficients of all the monomials are updated and the monomials with coefficient zero are eliminated; 2) the monomials whose coefficient modulo 2 evaluate to 0 are eliminated. The second case is applied after each substitution.

Sig_{out0}=z_0	elim	Sig_{out1}=x·z_1	elim
G8: z_0	-	G8: i_5x+i_6x	-
G7: i_1+i_2	-	G7: i_5x+i_6x	-
G6: i_1+i_2	-	G6: i_2x+x+i_6x	-
G5: i_1+i_2	-	G5: i_2x+x+i_3x+i_4x	-
G4: i_1+i_2	-	G4: i_2x+**x**+i_3x+a_0b_1x+**x**	2x
G3: i_1+i_2	-	G3: i_2x+a_1b_0x+x+a_0b_1x	-
G2: i_1+a_1b_1+1	-	G2: a_1b_1x+**x**+a_1b_0x+**x**+a_0b_1x	2x
G1: **1**+a_0b_0+a_1b_1+**1**	2	G1: x(a_1b_1+a_1b_0+a_0b_1)	-
z_0=a_0b_0+a_1b_1, z_1=x(a_1b_1+a_1b_0+a_0b_1)			

Figure 6: Extracting the algebraic expression of z_0 and z_1 in Fig. 4.

Example 3 (Figure 6): We illustrate our parallel extraction method using a 2-bit multiplier in GF(2^2) in Figure 3. The output signature Sig_{out} = z_0+z_1x is split into two signatures, $Sig_{out0} = z_0$ and $Sig_{out1} = z_1$. Then, the rewriting process is applied to Sig_{out0} and Sig_{out1} in parallel. When Sig_{out0} and Sig_{out1} have been successfully extracted, the two signatures are merged into Sig_{out0} + x·Sig_{out1}, resulting in the polynomial Sig_{in}. In Figure 4, we can see that elimination happens three times (F_4, F_2, and F_1). As expected, this happens within each element in GF(2^n). In Figure 6 one elimination in Sig_{out0} and two eliminations in Sig_{out1} have been done independently, as shown earlier (refer to Example 2).

V. Reverse Engineering

In this section, we present our approach to perform reverse engineering of GF(2^m) multipliers. Using the *extraction* technique presented in the previous section, we can extract the

algebraic expression of each output bit. In contrast to the algebraic techniques of [6] [10], our extraction technique can extract the algebraic expression of each output bit independently. This means that the extraction can be done without the knowledge of the bit position of the inputs and outputs. Two theorems are provided and proved to support this claim.

In a GF(2^m) multiplication, let s_i ($i \in \{0,1,...,2m\text{-}1\}$) be a set of partial products generated by AND gates and combined with an XOR operations. For example, in Figure 1, there are six product sets, s_0, s_1, ..., s_6, where s_1=a_1b_0+a_0b_1; or written as a set: s_1=$\{a_1b_0, a_0b_1\}$, etc. These product sets are divided into two groups: those with index $i \leq m-1$, called *in-field* product sets; and those with index $i \geq m$, called *out-of-field* product sets. The in-field product sets s_i, in this case s_0, s_1, s_2, s_3, correspond to the output bits z_i. The out-of-field product sets will be reduced into the field GF(2^m) using mod $P(x)$ operation, and assigned to the respective output bit, as determined by $P(x)$. In the case of Figure 1, the out-of-field sets are s_4, s_5, s_6. In general, for a GF(2^m) multiplication, m product sets are *in-field*, and m-1 product sets are *out-of-field* [38].

A. Output Encoding Determination

We will now demonstrate how to determine the encoding, and hence bit position, of the outputs.

Theorem 2: Given a GF(2^m) multiplication, the in-field product sets (s_0, s_1, ..., s_{m-1}) appear in exactly one element of GF(2^m) each, and the out-of-field product sets (s_m, s_{m+1}, ..., s_{2m-1}) appear in at least two elements (outputs) of GF(2^m), as a result of reduction mod $P(x)$.

Proof: An irreducible polynomial in GF(2^m) has the standard form $P(x) = x^m + P'(x)$, where the tail polynomial $P'(x)$ contains at least two monomials x^d with degree $d < m$. For example, there are two such monomials for a trinomial, four for pentanomial, etc. Since $P(x) = 0$ we have $x^m = P'(x)$ in GF(2^m). Hence the variable x^m, associated with the first out-of-field partial product set s^m will appear in at least two outputs, determined by $P'(x)$. Other variables, x^k, associated with out-of-field partial product set s_k, for $k > m$, can be expressed as $x^k = x^{k-m}x^m = x^{k-m}P'(x)$ and will contain at least two elements. QED □

In fact, the number of outputs in which the out-of-field set s_k will appear is equal to the number of monomials in the above product $x^{k-m}P'(x)$, provided that every monomial x^j with $j > m$ is recursively reduced mod $P'(x)$, i.e., by using relation $x^m = P'(x)$. We illustrate this fact with an example of multiplication in GF(2^4) using irreducible polynomial $P_1(x) = x^4 + x^3 + 1$ shown in the left side of Figure 1. The in-field sets, associated with outputs z_0, z_1, z_2, z_3, are s_0, s_1, s_2, s_3. Since $P_1(x) = x^4 + x^3 + 1 = 0$, we obtain $x^4 = x^3 + 1$. This means that set s_4 appears in two output columns, z_3 and z_0. Then

$$x^5 = x \cdot x^4 = x(x^3 + 1) = x^4 + x = x^3 + x + 1,$$

which means that s_5 appears in three outputs: z_3, z_1, z_0. Finally,

$$x^6 = x \cdot x^5 = x(x^3 + x + 1) = x^4 + x^2 + x = x^3 + x^2 + x + 1,$$

that is, s_6 will appear in four outputs: z_3, z_2, z_1, z_0. As expected, this matches the left Table in Figure 1. Note the recursive derivation of x^k for $k > m$, which increases the number of columns to which a given set s_k is assigned.

Based on Theorem 2, we can find the in-field product sets, s_0, s_1, ..., s_{m-1}, by searching the unique products in the resulting algebraic expressions of the output bits. In this context, *unique products* are the products that exist in only one of the extracted algebraic expressions. Since the in-field product set indicates the bit position of the output, we can determine the bit positions of the output bits as soon as all the in-field product sets are identified.

Example 4 (Figure 2): We illustrate the procedure of determining bit positions with an example of a GF(2^4) multiplier implemented using irreducible polynomial $P_2(x)$=x^4+x+1 (see Figure 1). Note that in this process the labels do not offer any knowledge of the bit positions of inputs and outputs. The extracted algebraic expressions of the four output bits are shown in Figure 2. The labels of the variables do not indicate any binary encoding information. We first identify the unique products that include set s_0=a_0b_0 in algebraic expression of z_0; set s_1=(a_0b_1+a_1b_0) in z_1; set s_2=(a_0b_2+a_1b_1+a_2b_0) in z_2; and set s_3=(a_0b_3+a_1b_2+a_2b_1+a_3b_0) in z_3. Note that the number of products in the in-field product set s_i is i. Hence, we find all the in-field product sets and their relation to the extracted algebraic to be as follows:

s_0 = a_0b_0, z_0 → Least significant bit (LSB)
s_1 = a_0b_1+a_1b_0, z_1 → 2^{nd} output bit
s_2 = a_0b_2+a_1b_1+a_2b_0, z_2 → 3^{rd} output bit
s_3 = a_0b_3+a_1b_2+a_2b_1+a_3b_0, z_3 → Most significant bit (MSB)

B. Input Encoding Determination

Algorithm 2 Input encoding determination for $GF(2^m)$

Input: a set of algebraic expressions represent the in-field product sets S
Output: bit position of input variables

```
1: S={s_0, s_1, ..., s_{m-1}}
2: initialize a vector of variables V ← {}
3: for i=0, i≤m-1, i++ do
4:     for each variable v in algebraic expression of s_i do
5:         if v does not exist in V then
6:             assign bit position value of v = i
7:             store v in variable set V
8:         end if
9:     end for
10: end for
11: return V
```

We can now determine the bit position of the input variables using the procedure outlined in Algorithm 2. The input bit position can be determined by analyzing the in-field product sets, obtained in the previous step. Based on the GF multiplication algorithm, we know that s_0 is generated by an AND function with two LSBs of the two inputs; and the two products in s_1 are generated by the AND and XOR operations using two LSBs and two 2^{nd} input bits, etc. For example in a GF(2^4) multiplication (Figure 1), s_0=a_0b_0, where a_0 and b_0 are LSBs; s_1=a_1b_0+a_0b_1, where a_0, b_0 are LSBs; a_1, b_1 are 2^{nd} LSBs. This allows us to determine the bit position of the input bits recursively by analyzing the algebraic expression of s_i. We

illustrate this with the GF(2^4) multiplier implemented using $P_2(x) = x^4+x+1$ (Figure 2).

Example 5 (Algorithm 2): The input of our algorithm is a set of algebraic expressions of the in-field product sets, s_0, s_1, s_2, s_3 (line 1). We initialize vector V to store the variables in which their bit positions are assigned (line 2). The first algebraic expression is s_0. Since the two variables, a_0 and b_0 are not in V, the bit positions of these two variables are assigned index $i = 0$ (line 4-8). In the second iteration, V=$\{a_0, b_0\}$, and the input algebraic expression is s_1, including variables a_0, b_0, a_1 and b_1. Because a_1 and b_1 are not in V, their bit position is $i = 1$. The loop ends when all the algebraic expressions in S have been visited, and returns V=$\{(a_0, b_0)_0, (a_1, b_1)_1, (a_2, b_2)_2, (a_3, b_3)_3\}$. The subscripts are the bit position values of the variables returned by the algorithm. Note that this procedure only gives the bit position of the input bits; the information of how the input words are constructed is unknown. There are 2^{m-1} combinations from which the words can be constructed using the information returned in V. For example, the two input words can be W_0=a_0+2a_1+4b_2+8a_3 and W_1=b_0+2b_1+4a_2+8b_3; or they can be W'_0=a_0+2a_1+4b_2+8b_3 and W'_1=b_0+2b_1+4a_2+8a_3. Although there may be many combinations for constructing the input words, the specification of the GF(2^m) is unique.

C. Extraction of the Irreducible Polynomial

Theorem 3: *Given a multiplication in GF(2^m), let the first out-of-field product set be s_m. Then, the irreducible polynomial $P(x)$ includes monomials x^m and $\{x^i\}$ iff all products in the set s_m appear in the algebraic expression of the i^{th} output bits, for all $i < m$.*

Proof: Based on the definition of field arithmetic for GF(2^m), the polynomial basis representation of s_m is $x^m s_m$. To reduce s_m into elements in the range [0, $m-1$], the field reductions are performed modulo irreducible polynomial $P(x)$ with highest degree m ($c.f.$ the proof of Theorem 2). As before, let $P(x) = x^m + P'(x)$. Then,

$$x^m s_m \ mod \ (x^m + P'(x)) \ = \ s_m P'(x)$$

Hence, if x^i exists in $P'(x)$, it also exists in $P(x)$. Therefore, x^i exists in $P(x)$, iff $x^i s_m$ exists in $x^m s_m$ mod $P(x)$. □

Even though the input bit positions have been determined in the previous step, we cannot directly generate s_m since the combination of the input bits for constructing the input words is still unknown. In Example 5 (m=4), we can see that s_m=$\{a_1b_3, a_2b_2, a_3b_1\}$ when input words are W_0 and W_1; but s_m=$\{a_1a_3, a_2b_2, b_1b_3\}$ when inputs words are W'_0 and W'_1. To overcome this limitation, we create a set of products s'_m, which includes all the possible products that can be generated based on all input combinations. The set s'_m includes the $true$ products, i.e., those that exist in the first out-of-field product set; and it also includes some $dummy$ products. The dummy products are those that never appear in the resulting algebraic expressions. Hence, we first generate the set s'_m and eliminate the dummy products by searching the algebraic expressions. After this, we obtain s_m. Then, we use s_m to extract the irreducible polynomial $P(x)$ using Algorithm 3.

Example 6: We illustrate the method of reverse engineering the irreducible polynomial using the $GF(2^4)$ multiplier of Fig. 1. The procedure is outlined in Algorithm 3. The extracted algebraic expressions S (line 1 at Algorithm 3) is shown in Figure 2. The bit position of input bits is determined by Algorithm 2 (line 2). Based on the result of Algorithm 2, we generate s'_m=$\{a_1a_3, b_1b_3, a_2b_2, a_3b_1, a_1b_3\}$. To eliminate the dummy products from s'_m, we search all algebraic expressions in S, and eliminate the products that cannot be part of the resulting products. In this case, we find that a_1a_3 and b_1b_3 are the dummy products. Hence, we get s_m=$\{a_3b_1, a_2b_2, a_1b_3\}$ (line 3). Based on the definition of irreducible polynomial, $P(x)$ must include x^m; in this example $m = 4$ (line 4). While looping over all the algebraic expressions, the expressions for z_0 and z_1 contain all the products of s_m. Hence, x^0 and x^1 are included in $P(x)$, so that $P(x) = x^4+x^1+x^0$. We can see that it is the same as $P_2(x)$ in Figure 1.

Algorithm 3 Extracting irreducible polynomial in $GF(2^m)$

Input: the algebraic expressions of output bits S
Input: the first out-of-field product set s_m
Output: Irreducible polynomial $P(x)$
1: S = $\{exp_0, exp_1, ..., exp_{m-1}\}$
2: $V \leftarrow$ Algorithm 2(S)
3: $s_m \leftarrow eliminate\_dummy(s'_m \leftarrow V, S)$
4: $P(x)$=x^m: initialize irreducible polynomial
5: **for** i=0, i≤m-1, i++ **do**
6: **if** all products in s_m exist in exp_i **then**
7: $P(x)$ += x^i
8: **end if**
9: **end for**
10: **return** $P(x)$

In summary, using the framework presented in Section IV-C, we first extract the algebraic expressions of all output bits. Then, we analyze the algebraic expressions to find the bit position of the input bits and the output bits, and extract the irreducible polynomial $P(x)$. In the example of the GF(2^4) multiplier implemented using $P(x) = x^4+x+1$, shown in Figure 1, the final results returned by our approach gives the following: 1) the input bits set V= $\{(a_0, b_0)_0, (a_1, b_1)_1, (a_2, b_2)_2, (a_3, b_3)_3\}$, where the subscripts represent the bit position; 2) z_0 is the least significant bit (LSB), z_1 is the 2^{nd} output bit, z_2 is the 3^{rd} output bit, and z_3 is the most significant bit (MSB); 3) irreducible polynomial is $P(x)$ = x^4+x+1; 4) the specification can be verified using the approach presented in Section IV with the reverse engineered information.

VI. Results

The experimental results of our method are presented in two subsections: 1) evaluation of parallel verification of GF(2^m) multipliers; and 2) evaluation of reverse engineering of GF(2^m) multipliers. The results given in this section include data (total time and maximum memory) for the entire verification or reverse engineering process, including translating the gate-level verilog netlist to the algebraic equation, performing backward rewriting and other required functions.

A. Parallel Verification of GF(2^m) Multipliers

The verification technique for GF(2^m) multipliers presented in Section IV was implemented in C++. It performs backward

Mastrovito		[5]		This work					
Op size	# equations	Runtime (sec)	Mem (MB)	Runtime (s)					Mem*
				T=1	*T=5*	*T=10*	*T=20*	*T=30*	*T=1**
32	5,482	1	3	5	2	1	1	1	10 MB
48	12,228	8	13	9	6	3	3	2	21 MB
64	21,814	29	21	19	11	8	7	7	37 MB
96	51,412	195	45	68	38	26	20	23	84 MB
128	93,996	924	91	153	91	63	55	57	152 MB
163	153,245	3546	161	336	192	137	121	113	248 MB
233	167,803	4933	168	499	294	212	180	171	270 MB
283	399,688	30358	380	1580	890	606	550	530	642 MB
571	1628,170	*TO*	-	13176	7980	5038	*MO*	*MO*	2.6 GB

TABLE I: Results of verifying Mastrovito multipliers using our parallel approach. T is the number of threads. MO=Memory out of 32 GB. TO=Time out of 18 hours.
(**T=1* shows the maximum memory usage of a single thread.)

Montgomery		[5]		This work					
Op size	# equations	Runtime (sec)	Mem (MB)	Runtime (s)					Mem*
				T=1	*T=5*	*T=10*	*T=20*	*T=30*	*T=1**
32	4,352	2	3	5	3	2	1	2	8 MB
48	9,602	14	13	34	18	11	9	6	16 MB
64	16.898	63	21	80	45	31	28	27	27 MB
96	37,634	554	45	414	234	157	133	142	59 MB
128	66,562	1924	68	335	209	121	115	110	95 MB
163	107,582	12063	101	2505	1616	1172	1095	1008	161 MB
233	219,022	*TO*	168	1240	722	565	457	480	301 MB
283	322,622	*TO*	380	32180	19745	17640	15300	14820	488 MB

TABLE II: Results of verifying *Montgomery* multipliers using our parallel approach. T is the number of threads. TO=Time out of 18 hours.
(**T=1* shows the maximum memory usage of a single thread.)

rewriting with variable substitution and polynomial reductions in Galois field in parallel fashion. The program was tested on a number of combinational gate-level $GF(2^m)$ multipliers taken from [6], including the Montgomery multipliers [39] and Mastrovito multipliers [40]. The bit-width of the multipliers varies from 32 to 571 bits. The verification results for various Galois field multipliers obtained using SAT, SMT, ABC [41], and Singular [42], have already been presented in works of [1] and [6]. They clearly demonstrate that techniques based on computer algebra perform significantly better than other known techniques. Hence, in this work, we only compare our approach to those two, and specifically to the tool described in [6]. However, in contrast to the previous work on Galois field verification, all the GF(2^m) multipliers used in this paper are bit-blasted gate-level implementations. The bit-level multipliers are taken from [6] and mapped onto gate-level circuits using ABC [41]. Our experiments were conducted on a PC with Intel(R) Xeon CPU E5-2420 v2 2.20 GHz $\times$12 with 32 GB memory. As described in the next section, our technique can verify Galois field multipliers in multiple threads by applying Algorithm 1 to each output bit in parallel. The number of threads is given as input to the tool.

The experimental results of our approach and comparison with [6] are shown in Table I for gate-level Mastrovito multipliers with bit-width varying from 32 to 571 bits. These multipliers are directly mapped using ABC without any optimization. The largest circuit includes over 1.6 million gates. This is also the number of polynomial equations and the number of rewriting iterations (see Section IV). The results generated by the tool, presented in [6] are shown in columns 3 and 4 of Table I. We performed four different series of experiments, with the number of threads T varying from 5 to 30. The table shows CPU runtime and memory usage for different values of T. The timeout limit (TO) was set to 12 hours and memory limit (MO) to 32 GB. The experimental results show that our approach provides on average 26.2$\times$, 37.8$\times$, 42.7$\times$, and 44.3$\times$ speedup, for T = 5, 10, 20, and 30 threads, respectively. Our approach can verify the multipliers up to 571 bit-wide multipliers in 1.5 hours, while that of [6] fails after 12 hours.

The reported memory usage of our approach is the maximum memory usage *per thread*. This means that our tool experiences maximum memory usage with all T threads running in the process; in this case, the memory usage is $T \cdot Mem$. This is why the 571-bit Mastrovito multipliers could be successfully verified with T = 5 and 10, but failed with T = 20 and 30 threads. For example, the peak memory usage of 571-bit Mastrovito multiplier with $T = 20$ is $2.6 \times 20 = 52$ GB, which exceeds the available memory limit.

We also tested Montgomery multipliers with bit-width varying from 32 to 283 bits; the results are shown in Table II. These experiments are different than those in [6]. In our work, we first flatten the Montgomery multipliers before applying our verification technique. That is, we assume that only the positions of the primary inputs and outputs are known, without the knowledge of any high-level structure. In contrast, [6] verifies the Montgomery multipliers that are represented with four hierarchical blocks. For 32- to 163-bit Montgomery multipliers, our approach provides on average a 9.2$\times$, 15.9$\times$, 16.6$\times$, and 17.4$\times$ speedup, for $T = 5$, 10, 20, and 30, respectively. Notice that [6] cannot verify the flattened Montgomery multipliers larger than 233 bits in 12 hours.

Analyzing Table I we observe that the rewriting technique of our approach when applied to Montgomery multipliers require

significantly more time than for Mastrovito multipliers. The main reason for this difference is the internal architecture of the two multiplier types. Mastrovito multipliers are obtained directly from the standard multiplication structure, with the partial product generator followed by an XOR-tree structure, as in modular arithmetic. Since the algebraic model of XOR in GF arithmetic is linear, the size of the polynomial expressions generated during rewriting of this architecture is relatively small. In contrast, in a Montgomery multiplier the two inputs are first transformed into Montgomery form; the products of these Montgomery forms are called *Montgomery products*. Since the polynomial expressions in Montgomery forms are much larger than partial products, the increase in size of intermediate expressions is unavoidable.

1) ***Dependence on*** $P(x)$*:* In Table II, we observe that CPU runtime for verifying a 163-bit multiplier is greater than that of a 233-bit multiplier. This is because the computational complexity depends not only on the bit-width of the multiplier, but also on the irreducible polynomial $P(x)$ used in constructing the multiplier.

We illustrate this fact using two GF(2^4) multiplications implemented using two different irreducible polynomials (c.f. Figure 1). We can see that for $P_1(x)$=x^4+x^3+1, the longest logic paths for z_3 and z_0, include ten and seven products that need to be generated using XORs, respectively. However, when $P_2(x)$=x^4+x+1, the two longest paths, z_1 and z_2, have only seven and six products. This means that the GF(2^4) multiplication requires 9 XOR operations using $P_1(x)$ and 6 XOR operations using $P_2(x)$. In other words, the gate-level implementation of the multiplier implemented using $P_1(x)$ has more gates compared to $P_2(x)$. In conclusion, we can see that irreducible polynomial $P(x)$ has significant impact on both design cost and the verification time of the GF(2^m) multipliers.

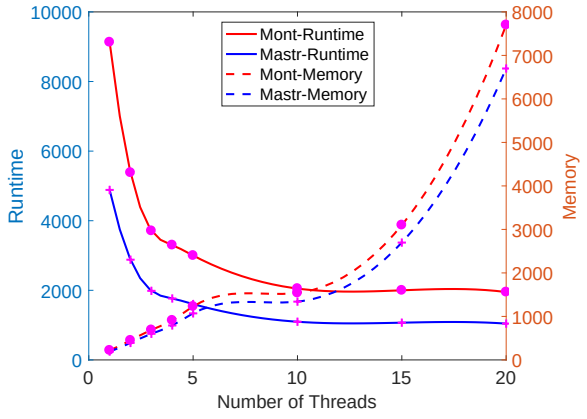

Figure 7: Runtime and memory usage of parallel verification approach as a function of the number of threads T.

2) ***Runtime vs. Memory****:* In this section, we discuss the tradeoff of runtime and memory usage of our parallel approach to Galois Field multiplier verification. The plots in Figure 7 show the average runtime and memory usage for different number of threads, over the set of multipliers shown in Tables I and II (32 to 283 bits). The vertical axis on the left is CPU runtime (in seconds), and on the right is memory usage (MB). The horizontal axis represents the number of threads T, ranging from 1 to 30. The runtime is significantly improved for T ranging from 5 to 15. However, there is not much speedup when T is greater than 20, most likely due to the memory management synchronization overhead between the threads. Similarly to the results for Mastrovito multipliers (Table I), our approach is limited here by the memory usage when the size of the multiplier and the number of threads T are large. In our work, $T = 20$ seems to be the best choice. Obviously, T varies for different platforms, depending on the number of cores and the memory.

We also analyzed the runtime complexity of our verification algorithm for a single thread (T=1) computation; it is shown in Figure 8. The y-axis shows the total runtime of rewriting the polynomial expressions, and x-axis indicates the size of the Mastrovito multiplier. The result shows that the overall speedup is roughly the same for each value of T. Montgomery multipliers exhibit similar behavior, regardless of the choice of the irreducible polynomial.

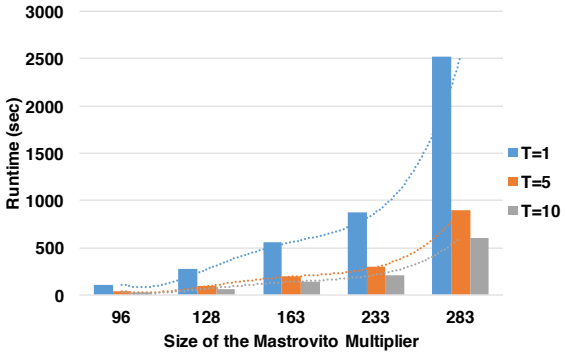

Figure 8: Single thread runtime analysis for Mastrovito multipliers.

3) ***Effect of Synthesis on Verification****:* In [10] the authors conclude that highly bit-optimized integer arithmetic circuits are harder to verify than their original, pre-synthesized netlists. This is because the efficiency of the rewriting technique relies on the amount of cancellations between the different terms of the polynomial, and such cancellations strongly depend on the order in which signals are rewritten. A good ordering of signals is difficult to achieve in highly bit-optimized synthesized circuits.

To see the effect of synthesis on parallel verification of GF circuits, we applied our approach to *post-synthesized* Galois field multipliers with operands up to 409 bits (571-bit multipliers could not be synthesized in a reasonable time). We synthesized *Mastrovito* and *Montgomery* multipliers using *ABC* tool [41]. We repeatedly used the commands *resyn2* and *dch*[3] until the number of AIG levels or nodes could not be reduced anymore. The synthesized multipliers were mapped using a 14nm technology library. The verification experiments shown in Table III are performed by our tool with $T = 20$ threads. Our tool was able to verify both 409-bit *Mastrovito* and *Montgomery* multipliers within just 13 minutes. We observed that in our parallel approach Galois field multipliers are easier to be verified after optimization

[3]"dch" is the most efficient bit-optimization function in ABC.

than in their original form. For example, the verification of a 283-bit Montgomery multiplier takes 15,300 seconds for T =20. After optimization, the runtime dropped to just 169.2 seconds, which means that such a verification is 90x faster than of the original implementation. The memory usage has also been reduced from 488 MB to 194 MB. In summary, in contrast to verification problems of integer multipliers [10], the bit-level optimization actually reduces the complexity of backward rewriting process. This is because extracting the function of an output bit of a GF multiplier depends only on the logic cone of that bit and does not require logic expression from other bits to be simplified (c.f. Theorem 3). Hence, the complexity of function extraction is naturally reduced if the logic cone is minimized.

Op size	*Mastrovito*			*Montgomery*		
	# eqn	Runtime(s)	Mem	# eqn	Runtime(s)	Mem
64	11499	4	21 MB	9471	15	38 MB
96	25632	11	44 MB	20306	41	54 MB
128	45983	29	77 MB	35082	27	78 MB
163	73483	62	123 MB	56408	205	153 MB
233	121861	135	201 MB	110947	141	199 MB
283	120877	168	198 MB	111006	169	194 MB
409	385974	776	635 MB	340076	751	597 MB

TABLE III: Runtime and memory usage of synthesized *Mastrovito* and *Montgomery* multipliers (T=20).

B. Reverse Engineering of GF(2^m) Multipliers

The reverse engineering technique presented in this paper was implemented in the framework described in Section V in C++. It reverse engineers bit-blasted GF(2^m) multipliers by analyzing the algebraic expressions of each element using the approach presented in Section IV. The program was tested on a number of gate-level $GF(2^m)$ multipliers with different irreducible polynomials, including Montgomery multipliers and Mastrovito multipliers. The multiplier generator, taken from [1], takes the bit-width and the irreducible polynomial as inputs and generates the multipliers in the equation format. The experimental results show that our technique can successfully reverse engineer various GF(2^m) multipliers, regardless of the GF(2^m) algorithm and the irreducible polynomials. We set the number of threads to 16 for all the reverse engineering evaluations in this section. This is dictated by the fact that T=16 gives most promising performance (runtime) and scalability (memory usage) metrics on our platform, based on the analysis presented in Section VI-A2 (Figure 7).

m	$P(x)$	*Mastrovito-syn*		*Montgomery-syn*	
		T(s)	Mem	T(s)	Mem
64	$x^{64}+x^{21}+x^{19}+x^4+1$	13	25 MB	5	20 MB
163	$x^{163}+x^{80}+x^{47}+x^9+1$	69	508 MB	221	610 MB
233	$x^{233}+x^{74}+1$	152	1.2 GB	154	2.9 GB
409	$x^{409}+x^{87}+1$	825	6.5 GB	855	10.3 GB

TABLE IV: Results of reverse engineering synthesized and technology mapped Mastrovito and Montgomery multipliers.

Our program takes the netlist/equations of the GF(2^m) implementations, and the number of threads as input. Hence, the users can adjust the parallel efforts depending on the limitation of the machines. In this work, all results are performed in 16 threads. Typical designs that require reverse engineering are those that have been bit-optimized and mapped using a standard-cell library. Hence, we apply our technique to the bit-optimized Mastrovito and Montgomery multipliers (Table IV). For the purpose of our experiments, the multipliers are optimized and mapped using ABC [41]. Compared to the verification runtime of synthesized multipliers (Table III), the CPU time spent on analyzing the extracted expressions for reverse engineering is less than 10% of the extraction process. This is because most computations of reverse engineering approach are associated with those for extracting the algebraic expressions, as presented in Section VI-A2, Table III.

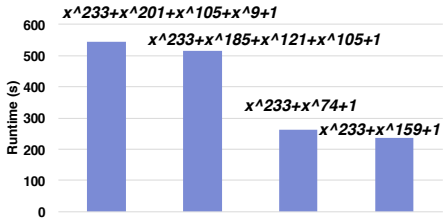

Figure 9: Result of reverse engineering GF(2^{233}) Mastrovito multipliers implemented with different P(x).

The reverse engineering approach has been further evaluated using four Mastrovito multipliers, each implemented with a different irreducible polynomial $P(x)$ in GF(2^{233}). The polynomials are obtained from [43] and optimized using ABC synthesis tool. The results are shown in Figure 9. We can see that the multipliers implemented with trinomial $P(x)$ are much easier to be reverse engineered than those based on a pentanomial $P(x)$. This is because the multipliers implemented with pentanomial $P(x)$ contain more gates and have longer critical path, since the reduction over pentanomial requires more XOR operations. The CPU runtime for irreducible polynomial of the same class (trinomials or pentanomials) is almost the same. As discussed in Section III-B, comparison of the two trinomials shows that the efficient trinomial irreducible polynomial, x^m+x^a+1, typically satisfies m-a>$m/2$.

The results for designs synthesized with 14nm technology library are shown in Figure 10. It shows that the area and delay of the Mastrovito multiplier implemented with $P(x)$=x^{233}+x^{74}+1 are 5.7% and 7.4% less than for $P(x)$=x^{233}+x^{159}+1, respectively.

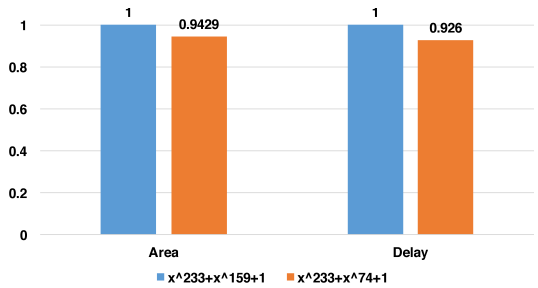

Figure 10: Evaluation of the design cost using GF($2^2$33) Mastrovito multipliers with irreducible polynomials x^{233}+x^{159}+1 and x^{233}+x^{74}+1.

VII. Conclusion

This paper presents a parallel approach to verification and reverse engineering of gate-level Galois Field multipliers using

computer algebraic approach. It introduces a parallel rewriting method that can efficiently extract functional specification of Galois Field multipliers as polynomial expressions. We demonstrate that compared to the best known algorithms, our approach tested on T=30 threads provides on average 44$\times$ and 17$\times$ speedup in verification of Montgomery and Mastrovito multipliers, respectively. We presented a novel approach that reverse engineers the gate-level Galois Field multipliers, in which the irreducible polynomial, as well as the bit position of the inputs and outputs are unknown. We demonstrated that our approach can efficiently reverse engineer the Galois Field multipliers implemented using different irreducible polynomials. Future work will focus on formal verification of prime field arithmetic circuits and complex cryptography circuits.

ACKNOWLEDGMENT

The authors would like to thank Prof. Kalla, University of Utah, for his valuable comments and the benchmarks; and Dr. Arnaud Tisserand, University Rennes 1 ENSSAT, for his valuable discussion. This work has been funded by NSF grants, CCF-1319496 and CCF-1617708.

REFERENCES

[1] J. Lv, P. Kalla, and F. Enescu, "Efficient Grobner Basis Reductions for Formal Verification of Galois Field Arithmatic Circuits," *IEEE Trans. on CAD*, vol. 32, no. 9, pp. 1409–1420, September 2013.

[2] M. Ciesielski, C. Yu, W. Brown, D. Liu, and A. Rossi, "Verification of Gate-level Arithmetic Circuits by Function Extraction," in *52nd DAC*. ACM, 2015, pp. 52–57.

[3] C. Paar and J. Pelzl, *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[4] M. Ciet, J.-J. Quisquater, and F. Sica, "A short note on irreducible trinomials in binary fields," in *23rd Symposium on Information Theory in the BENELUX*, 2002.

[5] T. Pruss, P. Kalla, and F. Enescu, "Equivalence Verification of Large Galois Field Arithmetic Circuits using Word-Level Abstraction via Gröbner Bases," in *DAC'14*, 2014, pp. 1–6.

[6] ——, "Efficient symbolic computation for word-level abstraction from combinational circuits for verification over finite fields," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 35, no. 7, pp. 1206–1218, 2016.

[7] E. Pavlenko, M. Wedler, D. Stoffel, W. Kunz, A. Dreyer, F. Seelisch, and G. Greuel, "Stable: A new qf-bv smt solver for hard verification problems combining boolean reasoning with computer algebra," in *DATE*, 2011, pp. 155–160.

[8] F. Farahmandi and B. Alizadeh, "Groebner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction," *Microprocessors and Microsystems*, vol. 39, no. 2, pp. 83–96, 2015.

[9] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, "Formal verification of integer multipliers by combining grobner basis with logic reduction," in *DATE'16*, 2016, pp. 1–6.

[10] C. Yu, W. Brown, D. Liu, A. Rossi, and M. J. Ciesielski, "Formal verification of arithmetic circuits using function extraction," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 2131–2142, 2016.

[11] C. Yu and M. J. Ciesielski, "Automatic word-level abstraction of datapath," in *IEEE International Symposium on Circuits and Systems, ISCAS 2016, Montréal, QC, Canada*, 2016, pp. 1718–1721.

[12] A. Sayed-Ahmed, D. Große, M. Soeken, and R. Drechsler, "Equivalence checking using grobner bases," *FMCAD'2016*, 2016.

[13] C. Yu and M. J. Ciesielski, "Efficient parallel verification of galois field multipliers," *ASP-DAC'17*, 2017.

[14] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. on Computers*, vol. 100, no. 8, pp. 677–691, 1986.

[15] R. E. Bryant and Y. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *Proceedings of the 32st Conference on Design Automation, San Francisco, California, USA, Moscone Center, June 12-16, 1995.*, 1995, pp. 535–541.

[16] Y.-A. Chen and R. Bryant, "*PHDD: An Efficient Graph Representation for Floating Point Circuit Verification," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-97-134, 1997.

[17] M. Ciesielski, P. Kalla, and S. Askar, "Taylor Expansion Diagrams: A Canonical Representation for Verification of Data Flow Designs," *IEEE Trans. on Computers*, vol. 55, no. 9, pp. 1188–1201, Sept. 2006.

[18] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, E. R. V. Frolov, and A. Naik., "Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation," in *Computer Aided Verification (CAV)*. Springer, 2009, pp. 414–429.

[19] A. Mishchenko *et al.*, "ABC: A System for Sequential Synthesis and Verification," *URL http://www. eecs. berkeley. edu/˜ alanmi/abc*, 2007.

[20] N. Sorensson and N. Een, "Minisat v1. 13-a sat solver with conflict-clause minimization," *SAT*, vol. 2005, p. 53, 2005.

[21] M. Soos, "Enhanced Gaussian Elimination in DPLL-based SAT Solvers." in *POS@ SAT*, 2010, pp. 2–14.

[22] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[23] C.-Y. Huang and K.-T. Cheng, "Using Word-level ATPG and Modular Arithmetic Constraint-Solving Techniques for Assertion Property Checking," *IEEE Trans. on CAD*, vol. 20, no. 3, pp. 381–391, March 2001.

[24] F. Fallah, S. Devadas, and K. Keutzer, "Functional vector generation for hdl models using linear programming and 3-satisfiability," in *Design Automation Conference (DAC)*. IEEE, 1998, pp. 528–533.

[25] R. Brinkmann and R. Drechsler, "RTL-datapath Verification using Integer Linear Programming," in *Proceedings of the 2002 Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE Computer Society, 2002, p. 741.

[26] Z. Zeng, K. R. Talupuru, and M. Ciesielski, "Functional Test Generation Based on Word-level SAT," *Journal of Systems Architecture*, vol. 51, no. 8, pp. 488–511, 2005.

[27] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. ios press, 2009, vol. 185.

[28] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, 2015.

[29] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[30] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer aided verification (CAV)*. Springer, 2011, pp. 171–177.

[31] M. J. Gordon and T. F. Melham, "Introduction to HOL A Theorem Proving Environment for Higher Order Logic," in *Cambridge University Press*, 1993.

[32] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A Prototype Verification System," in *Automated Deduction - CADE-11*. Springer, 1992, pp. 748–752.

[33] B. Brock, M. Kaufmann, and J. S. Moore, "Acl2 theorems about commercial microprocessors," in *Formal Methods in Computer-Aided Design (FMCAD)*. Springer, 1996, pp. 275–293.

[34] S. Vasudevan, V. Viswanath, R. W. Sumners, and J. A. Abraham, "Automatic Verification of Arithmetic Circuits in RTL using Stepwise Refinement of Term Rewriting Systems," *IEEE Trans. on Computers*, vol. 56, no. 10, pp. 1401–1414, 2007.

[35] D. Kapur and M. Subramaniam, "Mechanical Verification of Adder Circuits using Rewrite Rule Laboratory," *Formal Methods in System Design (FMCAD)*, vol. 13, no. 2, pp. 127–158, 1998.

[36] D. Stoffel and W. Kunz, "Equivalence Checking of Arithmetic Circuits on the Arithmetic Bit Level," *IEEE Trans. on CAD*, vol. 23, no. 5, pp. 586–597, May 2004.

[37] NIST, "Recommended elliptic curves for federal government use," 1999.

[38] C. Yu, D. Holcomb, and M. Ciesielski, "Reverse engineering of irreducible polynomials in gf (2 m) arithmetic," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017, pp. 1558–1563.

[39] C. K. Koc and T. Acar, "Montgomery multiplication in gf (2k)," *Designs, Codes and Cryptography*, vol. 14, no. 1, pp. 57–69, 1998.

[40] B. Sunar and Ç. K. Koç, "Mastrovito multiplier for all trinomials," *Computers, IEEE Transactions on*, vol. 48, no. 5, pp. 522–527, 1999.

[41] A. Mishchenko *et al.*, "Abc: A system for sequential synthesis and verification," *URL http://www. eecs. berkeley. edu/˜ alanmi/abc*, 2007.

[42] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann, "SINGULAR 3-1-6 A Computer Algebra System for Polynomial Computations," Tech. Rep., 2012, http://www.singular.uni-kl.de.

[43] M. Scott, "Optimal irreducible polynomials for gf (2m) arithmetic." *IACR Cryptology ePrint Archive*, vol. 2007, p. 192, 2007.