Reverse Engineering of Irreducible Polynomials in GF($2^m$) Arithmetic

Cunxi Yu, Daniel Holcomb, Maciej Ciesielski
ECE Department, University of Massachusetts, Amherst, USA
ycunxi@umass.edu, holcomb@engin.umass.edu, ciesiel@ecs.umass.edu

Abstract -
Current techniques for formally verifying circuits implemented in Galois field (GF) arithmetic are limited to those with a known irreducible polynomial $P(x)$. This paper presents a computer algebra based technique that extracts the irreducible polynomial $P(x)$ used in the implementation of a multiplier in GF($2^m$). The method is based on first extracting a unique polynomial in Galois field of each output bit independently, $P_i(x)$, is then obtained by analyzing the algebraic expression in GF($2^m$) of each output bit. We demonstrate that this method is able to reverse engineer the irreducible polynomial of an $n$-bit GF multiplier in $n$ threads. Experiments were performed on Mastrovito and Montgomery multipliers with different $P(x)$, including NIST-recommended polynomials and optimal polynomials for different microprocessor architectures.

Keywords—Reverse Engineering; Formal Verification; Galois Field Arithmetic; Computer Algebra.

I. INTRODUCTION

Galois field (GF) arithmetic is used to implement critical arithmetic components in communication and security-related hardware. It has been extensively applied in many digital signal processing and security applications, such as Elliptic Curve Cryptography (ECC), Advanced Encryption Standard (AES), and others. Multiplication is one of the most heavily used Galois field computations and is a complexity operation. Specifically, in cryptography systems, the size of Galois field circuits can be very large. Therefore, developing a general formal analysis technique of Galois field arithmetic HW/SW implementations becomes critical. Contemporary formal techniques, such as Binary Decision Diagrams (BDDs), Boolean Satisfiability (SAT), Satisfiability Modulo Theories (SMT), etc., are not efficient to either the verification or reverse engineering of Galois field arithmetic. The limitations of these techniques when applied to Galois field arithmetic have been addressed [1].

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 with degree $d \in [0, m]$ and coefficients in GF(2). For example, $P(x)=x^4 + x + 1$ is an irreducible polynomial in GF($2^4$). The multiplication in the field is performed modulo $P(x)$. Theoretically, there is a large number of irreducible polynomials available for constructing the field arithmetic operations in GF($2^m$). However, the choice of irreducible polynomial has great impact on the actual implementation of the resulting GF circuits and the performance of field arithmetic operations. The irreducible polynomials differ in the number of bit-level XOR operations. It is believed that, in general, the irreducible polynomial with minimum number of elements gives the best performance [2]. However, a later work [3] demonstrates that the best irreducible polynomial from circuit performance point of view varies in different scenarios, and depends on a computer architecture in which it is used, such as ARM vs. Intel-Pentium. In other words, 1) for GF($2^m$) multiplication, each irreducible polynomial results in a unique implementation; and 2) for a fixed field size, there exist many irreducible polynomials that could be used for constructing the field in different applications. This provides the main motivation for this work.

Computer algebra techniques with polynomial representations is believed to offer best solution for analyzing arithmetic circuits [1][4][5][6]. These work address the verification problems of Galois field arithmetic and integer arithmetic implementations, including abstractions [4][5][6]. The verification problem is typically formulated as proving that the implementation satisfies the specification. This task is accomplished by performing a series of divisions of the specification polynomial $F$ by the implementation polynomials $B$, representing components that implement the circuit. The techniques based on Gröbner Basis demonstrate that this approach can efficiently transform the verification problem to membership testing of the specification polynomial in the ideals [1][5]. A different approach to arithmetic verification of synthesized gate-level circuits has been proposed, using algebraic rewriting technique, which transforms polynomial of primary outputs to polynomial of primary inputs [6]. The technique proposed in [1] has been specifically applied to large GF($2^m$) arithmetic circuits. However, the knowledge of irreducible polynomial is essential to verify the implementations.

Symbolic computer algebra methods have also been used to reverse engineer the word-level operations for GF circuits and integer arithmetic circuits to speed up the verification process [7][8][9]. In the work of [7], the authors proposed a original spectral method based on analyzing the internal polynomial expressions during the rewriting procedure. Sayed-Ahmed et al. [8] introduced a reverse engineering technique in Algebraic Combinational Equivalence Checking (ACEC) process using Gröbner Basis by converting the function into canonical polynomials. However, both techniques are applicable to integer arithmetic only. In [9], an abstraction technique is introduced by analyzing the polynomial representation...
over $GF(2^m)$. However, similarly to [1], it is limited to the implementation with a known irreducible polynomial. In this work, we present a method that is able to reverse engineer the design by extracting the irreducible polynomial $P(x)$ of the $GF(2^m)$ multiplier, regardless of the $GF(2^m)$ algorithm used (e.g. Mastrovito and Montgomery). This procedure automatically checks the equivalence between the implementation with a golden implementation constructed using the extracted irreducible polynomial $P(x)$.

II. BACKGROUND

Different variants of canonical, graph-based representations have been proposed for arithmetic circuit verification, including Binary Decision Diagrams (BDDs) [10], Binary Moment Diagrams (BMDs) [11], Taylor Expansion Diagrams (TED) [12], and other hybrid diagrams. 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 [6][1]. Alternatively, arithmetic verification problems can be modeled and solved using Boolean satisfiability (SAT) or satisfiability modulo theories (SMT). However, it has been demonstrated that these techniques cannot efficiently solve the verification problem of large arithmetic circuits [1] [13]. Popular in industry are Theorem Provers, user-driven deductive systems for proving that an implementation satisfies the specification, using mathematical reasoning. However, Theorem Provers require manual guidance and in-depth domain knowledge, which makes it difficult to be applied automatically.

A. Computer Algebra Approaches

The most advanced techniques that have potential to solve the arithmetic verification problems are those based on symbolic Computer Algebra. These methods model the arithmetic circuit specification and its hardware implementation as polynomials [1][4][6][9][14]. The verification goal is to prove that implementation satisfies the specification by performing a series of divisions of the specification polynomial $F$ by the implementation polynomials $B = \{f_1, \ldots, f_s\}$, representing components that implement the circuit. The polynomials $f_1, \ldots, f_s$ are called the bases, or generators, of the ideal $J$. Given a set $f_1, \ldots, f_s$ of generators of $J$, a set of all simultaneous solutions to a system of equations $f_1 = 0; \ldots; f_s = 0$ is called a variety $V(J)$. Verification problem is then formulated as testing if the specification $F$ vanishes on $V(J)$. In some cases, the test can be simplified to testing if $F \in J$, which is known in computer algebra as ideal membership testing [1].

There are two basic techniques to reduce polynomial $F$ modulo $B$. A standard procedure to test if $F \in J$ is to divide polynomial $F$ by the elements of $B$: $\{f_1, \ldots, f_s\}$, one by one. The goal is to cancel, at each iteration, the leading term of $F$ using one of the leading terms of $f_1, \ldots, f_s$. If the remainder of the division $r$ is 0, then $F$ vanishes on $V(J)$, proving 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 check if $F$ is reducible to zero, a canonical set of generators, $G = \{g_1, \ldots, g_t\}$, called Gröbner basis is needed. This technique has been successfully applied to Galois field arithmetic [1] and integer arithmetic circuits [5].

Verification work of Galois field arithmetic has been presented in [1] [9]. These works provide significant improvement compared to other techniques, since their formulations rely on certain simplifying properties in Galois field during polynomial reduction. Specifically, the problem reduces to the ideal membership testing over a larger ideal that includes $J_0 = \langle x^2 - x \rangle$ in $F_2$. In this paper, we provide comparison between this technique and our approach.

B. Function Extraction

Function extraction is an arithmetic verification method originally proposed in [6] for integer arithmetic circuits, in $Z_{2^m}$. It extracts a unique bit-level polynomial function implemented by the circuit directly from its gate-level implementation. 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). This technique has been successfully applied to large integer arithmetic circuits, such as 512-bit integer multipliers. However, it cannot be directly applied to large GF multipliers because of exponential size of the intermediate number of polynomial terms before cancellations during rewriting. Fortunately, arithmetic $GF(2^m)$ circuits offer an inherent parallelism which can be exploited in backward rewriting.

In the rest of the paper, we first show 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 element in $F_2^m$ and the function is represented in algebraic expression where all variables are Boolean. Finally, we propose a method to reverse engineer the GF designs by extracting the irreducible polynomial $P(x)$ by analyzing these expressions.

C. Galois Field Multiplication

Galois field (GF) is an algebraic system with a finite number of elements and two main arithmetic operations, addition and multiplication; other operations can be derived from those two [15]. 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 $\{1, 2, \ldots, p - 1\}$, where $p$ is a prime number, with additions and multiplication performed modulo $p$. Binary extension field, denoted $GF(2^m)$ (or $F_2^m$), 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 the coefficients in $GF(2)$. 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 analog to the prime number $p$ in prime fields $GF(p)$. Extension fields are used in many cryptography applications, such as AES and ECC. In this work, we focus on the verification problem of $GF(2^m)$ multipliers.

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 are carried through the carry chain all the way to the most significant bit (MSB). In contrast, there is no carry propagation in $GF(2^m)$ implementations. To represent the result in $GF(2^3)$, the result of the integer multiplication have to be reduced in $GF(2^4)$ to only four output bits. 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 $A(x)=\sum_{n=0}^{n=3} a_n x^n$, $B(x)=\sum_{n=0}^{n=3} b_n x^n$, $Z(x)=\sum_{n=0}^{n=3} z_n x^n$. The functions of $s_i$ (i $\in$ [0, 6]) are represented using polynomials in $GF(2)$, namely: $s_i=s_{0}b_{i}+s_{1}a_{i}b_{i}+s_{2}a_{i}b_{i}+s_{3}a_{i}b_{i}$, up to $s_{6}=s_{2}a_{i}b_{i}$. The outputs $z_n$ (i $\in$ [0, 3]) are computed modulo the irreducible polynomial $P(x)$. Using $P_2(x)=x^4+x+1$, we obtain: $z_0=0s_0+s_1$, $z_1=s_1+s_4+s_5$, $z_2=s_2b_2+a_1b_1+a_2b_1+a_3b_1+a_3b_2$, and $z_3=s_3b_3+a_1b_3+a_3b_3$. 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 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.

\[ P(x_1)=x^4+x^3+1 \]
\[ P(x_2)=x^2+x+1 \]

\[ s_6 \quad s_5 \quad s_4 \quad s_3 \quad s_2 \quad s_1 \quad s_0 \]
\[ a_3 \quad a_2 \quad a_1 \quad a_0 \]
\[ b_3 \quad b_2 \quad b_1 \quad b_0 \]
\[ a_3b_3 \quad a_2b_0 \quad a_1b_1 \quad a_0b_0 \]
\[ a_3b_1 \quad a_2b_2 \quad a_1b_3 \quad a_0b_3 \]
\[ a_3b_0 \quad a_2b_1 \quad a_1b_2 \quad a_0b_2 \]
\[ s_6 \quad s_5 \quad s_4 \quad s_3 \quad s_2 \quad s_1 \quad s_0 \]

\[ P(x_1)x^4 + x^3 + 1 \]
\[ P(x_2)x^2 + x + 1 \]

Fig. 1: Two GF(2) multiplications constructed using $P(x_1)x^4 + x^3 + 1$ and $P(x_2)x^2 + x + 1$.

D. Irreducible Polynomials

For constructing the field $GF(2^m)$, the irreducible polynomial can be either a trinomial, $x^m+x+a+1$, or a pentanomial $x^m+x^a+x^b+x^c+x^d+1$ [16]. In [16], it is stated that the pentanomial is chosen as irreducible polynomial only if an irreducible trinomial doesn’t exist. In order to obtain efficient GF multiplication algorithm, it is required that $m \cdot a \geq w$. However, the work of [3] demonstrates that the trinomials are not always better than pentanomials. It means that for a given field size, there could be various irreducible polynomials used in different implementations.

An example of constructing $GF(2^4)$ multiplication using two different irreducible polynomials is shown in Figure 1. We can see that each polynomial corresponds to a unique multiplication. The performance difference can be evaluated by counting the XOR operations in each multiplication. Since the number of AND and XOR operations for generating partial products (variables $s_i$ in Fig. 1) is always the same, the difference is only caused by the reduction of the corresponding polynomials modulo $P(x)$. The number of XOR operations in reduction process can be counted 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 shown in Figure 1, one can immediately identify the irreducible polynomial $P(x)$. 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 $x^k$ are obtained from the non-zero terms corresponding to the entry $s^m$. 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 $s^3$ in columns $s_3$ and $s_0$. Similarly, for $P_2(x)=x^4+x^3+x^0$, the terms $x^3$ and $x^0$ are obtained by noticing that $s^4$ is placed in columns $s_3$ and $s_0$. The reason for it and the details of this procedure will be explained in the next section.

III. APPROACH

A. Computer Algebraic model

In this approach, the circuit is modeled as a network of logic elements, including: basic logic gates (AND, OR, XOR, INV), and complex standard cell gates (AOI, OAI, etc.) obtained by synthesis and technology mapping. The following algebraic equations are used to describe basic logic gates in $GF(2^m)$ [1]:

\[ \begin{align*}
-\alpha &= a + a \mod 2 \\
\alpha \land b &= a \land b \mod 2 \\
\alpha \lor b &= a \lor b \land a \cdot b \mod 2 \\
\alpha \oplus b &= a \oplus b \mod 2
\end{align*} \]

B. Outline of the Approach

Similarly to the work of [6], the computed function of the circuits is specified by two polynomials, referred to as output signature and input signature. The output signature of a $GF(2^m)$ multiplier is defined as $Sig_{out} = \sum_{i=0}^{m-1} z_i x_i$, and $z_i \in GF(2)$. The input signature of a $GF(2^m)$ multiplier is $Sig_{in} = \sum_{i=0}^{m-1} P_i x_i$, with coefficients (product terms) $P_i \in GF(2)$, and addition operation performed modulo 2. As discussed in Section II and shown in Figure 1, given an irreducible polynomial $P(x)$, the input signature $Sig_{in}$ can be computed easily in $GF(2^m)$. The goal of verification is first to transform the output signature, $Sig_{out}$, using polynomial

\[ \text{For polynomials in } GF(2), \text{ } "a" \text{ is computed as modulo 2.} \]
The rewriting process is described in Algorithm 1. During the rewriting, the polynomial is simplified by applying mod 2 reduction to all its terms. This is unlike in $\mathbb{Z}_{2^m}$ case, where some terms (with opposite signs) would cancel each other.

The rewriting algorithm takes the gate-level netlist of a $GF(2^m)$ circuit as input and first converts each logic gate into equations using Eq. (1). The rewriting process starts with $F_0 = S_{\text{out}}$, proceeds in a topological order of the netlist, and ends when all the variables in $F_i$ are all primary inputs. Each iteration includes two steps: Step 1 (lines 4-6 of the Algorithm) substitute the variable of the gate output using the expression in the inputs of the gate (Eq.1), and name the new expression $F_{i+1}$; Step 2 (line 4 and lines 8-10) simplify the new expression by removing all the monomials and constants that evaluate to 0 in $GF(2)$. The algorithm outputs the function of the design in $GF(2^m)$ after $n$ iterations, where $n$ is the number of gates in the netlist.

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

Input: Gate-level netlist of $GF(2^m)$ multiplier
Output: Output signature $S_{\text{out}}$

1. $P = \{p_1, p_2, ..., p_k\}$: polynomials representing gate-level netlist
2. $F_0 = S_{\text{out}}$
3. for each polynomial $p_i \in P$
4. for output variable $v$ of $p_i$ in $F_i$
5. replace every variable $v$ in $F_i$ by the expression of $p_i$
6. $F_i \rightarrow F_{i+1}$
7. for each monomial $M$ in $F_{i+1}$
8. if the coefficient of $M \mod 2 = 0$
9. or $M$ is constant, $M \mod 2 = 0$
10. remove $M$ from $F_{i+1}$
11. end if
12. end for
13. end for
14. end for
15. return $F_n$

The rewriting process is described in Algorithm 1. During the rewriting, the polynomial is simplified by applying mod 2 reduction to all its terms. This is unlike in $\mathbb{Z}_{2^m}$ case, where some terms (with opposite signs) would cancel each other.

The rewriting algorithm takes the gate-level netlist of a $GF(2^m)$ circuit as input and first converts each logic gate into equations using Eq. (1). The rewriting process starts with $F_0 = S_{\text{out}}$, proceeds in a topological order of the netlist, and ends when all the variables in $F_i$ are all primary inputs. Each iteration includes two steps: Step 1 (lines 4-6 of the Algorithm) substitute the variable of the gate output using the expression in the inputs of the gate (Eq.1), and name the new expression $F_{i+1}$; Step 2 (line 4 and lines 8-10) simplify the new expression by removing all the monomials and constants that evaluate to 0 in $GF(2)$. The algorithm outputs the function of the design in $GF(2^m)$ after $n$ iterations, where $n$ is the number of gates in the netlist.

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

Proof: The proof 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 by inspection. The correctness of the computed signature can be proved by induction on $i$, the step of transforming polynomial $F_i$ into $F_{i+1}$. Assuming that $F_0 = S_{\text{out}}$, and each $F_i \in GF(2^m)$, it is easy to show that $F_{i+1}$ remains in $GF(2^m)$, where each variable in $F_i$ represents output of some logic gate. During the rewriting process, this variable is substituted by a corresponding polynomial in $GF(2^m)$. Hence, the resulting polynomial $F_{i+1}$ correctly represents the function $F_{i+1} \in GF(2^m)$. Proof of the uniqueness of the computed signature follows the same reasoning.

**Example 1** (Figure 2): We illustrate our method using a post-synthesized 2-bit multiplier in $GF(2^5)$, shown in Figure 2. The irreducible polynomial of this design is $P(x) = x^2 + x + 1$.

In addition to verifying the design by comparing the computed polynomial $F_n$ with $S_{\text{out}}$, the expressions of $F_n$ will be used to extract the irreducible polynomial and perform the verification.

An important observation is that the cancellations of polynomial terms take place only within the expression associated with the same degree of polynomial ring ($S_{\text{out}}$ is a polynomial ring). In other words, the cancellations resulting the polynomial reduction happen in a logic cone of every output bit independently of other bits, regardless of logic sharing between the cones.

**Theorem 2 (Parallelizability):** Given a $GF(2^m)$ multiplier with $S_{\text{out}} = F_0 = z_0 z_1^m + z_1 z_1^m + ... + z_m z_1^m$, and $F_i = E_i x^i + E_{i-1} x^{i-1} + ... + E_0 x^0$, where $E_i$ is an algebraic expression in $GF(2^m)$ obtained during rewriting. Then, the polynomial reduction is possible only within a single expression $E_i$ for $i = 1, 2, ..., m$.

Proof: Consider a polynomial $E_i x^{n_i} + E_k x^{n_k}$, where $E_i$ and $E_k$ are simplified in $GF(2)$. That is, $E_i = (c_1 + c_2 + ...)$, and $E_k = (c_1' + c_2' + ...)$. After simplifying each of the two polynomials, there are no common monomials between $E_{i} x^{n_{i}}$ and $E_{k} x^{n_{k}}$. This is because for any element, $c_{1}^{i} x^{n_{i}} \neq c_{1}^{k} x^{n_{k}}$, for any pairs of $(i, k)$ and $(l, j)$.
rewrite G0 and G1. After this, $z_0$ is rewritten to $s_0+s_2$, and $z_1$ is rewritten to $s_1 x + s_2 x + x$. In the rewriting process, we can see that the polynomial reduction happen when there are monomials that are not in GF(2). For example, during the $j^{th}$ iteration of rewriting $z_1$, monomial $2x$ is eliminated. Also, we can see that the reductions happen only within the logic cone of each output bit, as proved in Theorem 2.

In the following, the output-products are the products $a_i b_j$, such that $i+j \geq 2m$. Since these products are associated with bits $s_{i+j}$, they are reduced by $P(x)$.

**Theorem 3**: Given a multiplication in GF($2^m$), let the first output-field product set be $\mathbb{F}_m$. Then, the irreducible polynomial $P(x)$ includes $x^m$ and $x^i$ iff all products in set $\mathbb{F}_m$ exist in the algebraic expression of the $i^{th}$ output bits, where $i \leq m$.

**Proof**: Based on the definition of field arithmetic, the polynomial basis representation of $\mathbb{F}_m$ is $\mathbb{P}_m x^m$. To reduce $\mathbb{F}_m$ into elements in the range $[0, m - 1]$ (with $m$ output bits), the field reductions are performed modulo irreducible polynomial $P(x)$ with highest degree of $m$. Based on the definition of irreducible polynomial, $P(x)$ is either a trinomial or a pentanomial with degree of $m$. Let $P(x)$ be $x^m+P'(x)$. Then,

$$\mathbb{P}_m x^m \mod (x^m + P'(x)) = \mathbb{P}_m P'(x)$$

Hence, if $x^i$ exists in $P'(x)$, it also exists in $P(x)$.

**Example 2** (Figure 2): We illustrate the method of reverse engineering the irreducible polynomial using the 2-bit multiplier in GF($2^m$), shown in Figure 2. The algorithm is shown in Algorithm 2. Using the rewriting technique (Algorithm 1) based on Theorem 1 and 2, we can extract the algebraic expressions of $a_0 b_0$, and $a_1 x = (a_0 b_1 + a_1 b_0 + a_1 b_1) x$, hence $z_1 = a_0 b_1 + a_1 b_0 + a_1 b_1$ (lines 3 - 5). In this example, $m=2$, hence $\mathbb{F}_3 = \{a_1 b_1\}$. We can see that both expressions of $z_0$ and $z_1$ include $\mathbb{F}_3$, which means that $x^3$ and $x^1$ are included in the irreducible polynomial of this design (lines 6 - 7). Based on Theorem 4, we know that $x^6$ is always included (line 2). Hence, irreducible polynomial of this design is $P(x) = x^3 + x + 1$ ($m=2$) (line 10).

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

Input: Gate-level netlist/equations of GF($2^m$) multiplier
Output: Irreducible polynomial $P(x)$

1: $\mathbb{F}_m = \{a_{m-1}, b_1, a_{m-2} b_2, ... a_1 b_{m-1}\}$
2: $P(x) = x^m$: initialize irreducible polynomial
3: for each output bit $z_i$ do
4: apply Algorithm 1 Backward_rewrite(netlist/equations, $z_i$)
5: $\mathbb{F}_P \leftarrow$ Backward_rewrite(netlist/equations, $z_i$)
6: if $\mathbb{F}_m$ exists in $\mathbb{F}_P$, then
7: $P(x) \leftarrow z_i$
8: end if
9: end for
10: return $P(x)$

**IV. RESULTS**

The technique described in this paper was implemented in C++. It reverse engineers the irreducible polynomials of GF($2^m$) multiplications by analyzing the algebraic expressions of each element. The program was tested on a number of combinational gate-level $G\bar{F}(2^m)$ multipliers with different irreducible polynomials including Montgomery multipliers and Mastrovito multipliers. The multiplier generators are taken from [1]. It shows that our technique can successfully reverse engineer the irreducible polynomials of various designs, regardless of the GF($2^m$) algorithm. The experiments were conducted on a PC with Intel(R) Xeon CPU E5-2420 v2 2.20 GHz x12 with 32 GB memory.

We first evaluate our approach using Montgomery and Mastrovito multipliers that are implemented using NIST-recommended irreducible polynomials [16]. The experimental results of Mastrovito multipliers with bit-width varying from 64 to 571 bits is shown in Table I and results of Montgomery multipliers with bit-width varying from 64 to 283 bits is shown in Table II. Note that we use the flattened version Montgomery multipliers, i.e. we have no knowledge of the block boundaries. The bit-width $m$ of the GF($2^m$) multiplier is shown in the first column. The irreducible polynomials used for constructing those multipliers are shown in the second column. The number of equations that represent the implementation is in the third column; it is also the number of iterations of extracting the polynomial expression of each output bit.

Our program takes the netlist/equations of the GF($2^m$) implementations, and the number of threads as inputs. Hence, the users can adjust the parallel effort depending on the hardware resource. In this work, all results are performed in 16 threads. The results in Table I and Table II show that the proposed technique can extract the irreducible polynomial $P(x)$ of large multipliers, regardless of the GF algorithm.
TABLE III: Results of extracting irreducible polynomial of optimized GF(2^m) Mastrovito and Montgomery multipliers.

<table>
<thead>
<tr>
<th>m</th>
<th>Irreducible polynomial</th>
<th>Mastrovito-syn Runtime(s)</th>
<th>Mem</th>
<th>Montgomery-syn Runtime(s)</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>x^408x^405+e^401+e^401+1</td>
<td>12.8</td>
<td>25 MB</td>
<td>5.2</td>
<td>20 MB</td>
</tr>
<tr>
<td>163</td>
<td>x^{105}+x^{102}+x^1+1</td>
<td>67.6</td>
<td>308 MB</td>
<td>21.7</td>
<td>610 MB</td>
</tr>
<tr>
<td>233</td>
<td>x^81x^3+1</td>
<td>149.6</td>
<td>1.2 GB</td>
<td>154.4</td>
<td>2.9 GB</td>
</tr>
<tr>
<td>409</td>
<td>x^2+1</td>
<td>821.8</td>
<td>6.5 GB</td>
<td>855.4</td>
<td>10.3 GB</td>
</tr>
</tbody>
</table>

The complexity of extracting the polynomial expressions becomes easier.

One observation is that in Table II, extracting $P(x)$ of GF($2^{163}$) multiplier requires four times runtime of extracting $P(x)$ of GF($2^{233}$) multiplier. The reason is that the complexity of the GF multiplication using different irreducible polynomials can be very different. The results shown in Table IV compare the performance of the irreducible polynomials of GF($2^{233}$) Mastrovito multipliers for different $P(x)$. Those multipliers are implemented with the polynomials shown in Table IV, which are optimal irreducible polynomials for different computer architectures [3].

We analyze the runtime complexity using various irreducible polynomials. This paper presents a computer algebra based technique that extracts the irreducible polynomial used in the implementation of a multiplier with a given GF($2^m$). The method is based on analyzing the unique polynomial expressions of the output bits in Galois field. The experimental results show that our technique is able to extract the irreducible polynomial up to 571-bit GF multipliers, regardless of the implementation. We analyze the runtime complexity using various irreducible polynomials.

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

REFERENCES


