Efficient Parallel Verification of Galois Field Multipliers

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

Abstract - Galois field (GF) arithmetic is used to implement critical arithmetic components in communication and security-related hardware, and verification of such components is of prime importance. Current techniques for formally verifying such components are based on computer algebra methods that proved successful in verification of integer arithmetic circuits. However, these methods are sequential in nature and do not offer any parallelism. This paper presents an algebraic functional verification technique of gate-level $GF(2^m)$ multipliers, in which verification is performed in bit-parallel fashion. The method is based on extracting a unique polynomial in Galois field of each output bit independently. We demonstrate that this method is able to verify an n-bit GF multiplier in n threads. Experiments performed on pre- and post-synthesized *Mastrovito* and *Montgomery* multipliers show high efficiency up to 571 bits.

Keywords— Formal verification; Galois field arithmetic circuits; 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 it is a high complexity operation. Specifically, in cryptography systems, the size of Galois field circuits can be very large. Therefore, developing 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 directly applicable 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 in [1].

The most successful techniques for verifying arithmetic circuits use computer algebra techniques with polynomial representations [1][2][3][4]. The verification problem is typically formulated as proving that the implementation satisfies the specification. This is accomplished by performing a series of divisions of the specification polynomial F by the implementation polynomials $B = \{f_1, \dots, f_s\}$, representing components that implement the circuit. The technique based on *Gröbner Basis* demonstrated that this approach can efficiently reduce the complexity of the verification problem to *membership testing* of the specification polynomial in the ideals [1][3]. This technique has been applied successfully to large Galois Field arithmetic circuits [1]. Symbolic computer algebra methods have been used to derive word-level operation for GF circuits and integer arithmetic circuits to improve the verification performance [5][6]. A different approach to arithmetic verification of synthesized gate-level circuits has been proposed in [4]. This method uses algebraic rewriting of the polynomials at the primary outputs to extract specification as a polynomial at the primary inputs.

However, a common limitation to all these works is that they are not applicable to parallel verification. This is because the verification problem based on computer algebra technique expresses the *specification* as polynomial in all output bits. In this approach, the polynomial division can be done only in a single thread. In principle, multiple specifications (called *output signature* in [4]) can be generated by splitting the output signature. However, we examined this method and found the performance to be really poor. The reason is that the technique of [4] needs to rewrite the entire output signature in all the output bits to benefit from large monomial cancellations during rewriting. In other works [1][5], the verification problem of post-synthesized Galois field multipliers have not been addressed.

In this work, we extend the verification technique of [4] to verification of Galois field multipliers, while applying bit-level parallelism. Specifically:

- We propose an algorithm for Galois field arithmetic verification, which significantly reduces the internal expression size during algebraic rewriting.
- We evaluate our approach using benchmarks used in [1][5], including *Mastrovito* and *Montgomery* multipliers, up to 571 bits. The results show that efficiency of our approach surpasses that of [1] and [5].
- We demonstrate that for the verification problem for an n-bit Galois field multiplier can be accomplished ideally in n parallel threads. In this work, we set the number of threads to 5, 10, 20, and 30. We also analyze the efficiency of our parallel approach by studying the tradeoff between CPU runtime and memory usage.
- We address the verification of *synthesized* Galois field multipliers, while previous work dealt only with the verification of structural representation (arithmetic netlists) prior to synthesis.

II. Background

Different variants of canonical, graph-based representations have been proposed for arithmetic circuit verification, including Binary Decision Diagrams (BDDs) [7], Binary Moment Diagrams (BMDs) [8], Taylor Expansion Diagrams (TED) [9], 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 for complex arithmetic circuits [4][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] [10]. Another class of solvers include Theorem Provers, deductive systems for proving that an implementation satisfies the specification, using mathematical reasoning. However, this technique requires manual guidance, 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][2][4][5][11].

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 $f_1, ..., f_s$ 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 $f_1(x_1, ..., x_n)$=0; ...,$f_s(x_1, ..., x_n)$=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 checking 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: $f_1, ..., f_s$, one by one. The goal is to cancel, at each iteration, the leading term of F using one of the leading terms of $f_1, ..., f_s$. If the remainder of the division is $r = 0$, 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, $G = \{g_1, ..., g_t\}$, called *Gröbner basis* is needed. This technique has been successfully applied to Galois field arithmetic [1] and integer arithmetic circuits [3]. A different approach has been proposed in [4], where a gate-level network is described by a system of equations and proved by *backward rewriting*. Starting with the known output signature (polynomial) in primary output variables, it rewrites the signature from the primary outputs to primary inputs, to extract an arithmetic function (specification). The specific verification work of Galois field arithmetic has been presented in [1] [5]. These works provide significant improvement compared to other techniques, since their formulations relies on certain simplifying properties in Galois field during polynomial reductions. Specifically, the problem reduces to the ideal membership testing over a larger ideal that includes $J_0 = \langle x^2 - x \rangle$ in $\mathbb{F}_2$. In this paper, we provide a comparison between this technique and our approach.

B. 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 can be derived from those two [12]. 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 is 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. For example, a 2-bit vector A=$\{a_0,\ a_1\}$ in $GF(2^2)$, is $A(x)$=a_0+a_1x, where $a_i \in GF(2)$={0,1}. Multiplication of field elements is performed modulo *irreducible polynomial* $P(x)$ of degree m and coefficients in $GF(2)$. For example, P=$x^2 + x + 1$ is an irreducible polynomial in $GF(2^2)$. 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.

An example of multiplication in $GF(2^2)$ is shown in Figure 1. The left part of the figure shows a standard 2-bit integer multiplication with four output bits. To represent the result in $GF(2^m)$, which can contain only two bits, the bits r_3 and r_2 are reduced in $GF(2^2)$. This result of such a reduction is shown on the right part of the figure. The input and output operands are represented using polynomials $A(x)$, $B(x)$ and $Z(x)$. The functions of s_0, s_1 and s_2 are represented using polynomials in $GF(2)$: s_0=a_0b_0, s_1=a_1b_0+a_0b_1, and s_2=a_1b_1[1]. Hence, z_0=s_0+s_2 and z_1=s_1+s_2. As a result, the coefficients of the multiplication are: z_0=a_0b_0+a_1b_1, z_1 = a_0b_1+a_1b_0+a_1b_1. In digital circuits, partial products can be implemented using AND gates, and addition modulo 2 using XOR gates. Note that, unlike in the 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 is computed independently of other bits.

$$\begin{array}{cccc}
 & & a_1 & a_0 \\
 & & b_1 & b_0 \\
\hline
 & & a_1b_0 & a_0b_0 \\
 & a_1b_1 & a_0b_1 & \\
\hline
r_3 & r_2 & r_1 & r_0
\end{array}
\qquad
\begin{array}{ccc}
 & a_1 & a_0 \\
 & b_1 & b_0 \\
\hline
 & a_1b_0 & a_0b_0 \\
a_1b_1 & a_0b_1 & \\
\hline
s_2 & s_1 & s_0 \\
 & s_2 & s_2 \\
\hline
 & z_1 & z_0
\end{array}$$

a) b)

Fig. 1: 2-bit multiplication: a) standard integer multiplication with 4-bit result; b) multiplication in $GF(2^2)$ with $A(x)$ = a_0+a_1x, $B(x)$ = b_0+b_1x and result $Z(x)$ = z_0+z_1x $\equiv$ $A(x)$·$B(x)$ mod $P(x)$; irreducible polynomial $P(x) = x^2 + x + 1$.

C. Function Extraction

Function extraction is an arithmetic verification method proposed in [4] for integer arithmetic circuits, in $\mathbb{Z}_{2^m}$. 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 show how to apply such parallel rewriting in $GF(2^m)$ circuits while avoiding memory explosion experienced in integer arithmetic circuits.

III. Preliminaries

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.) obtained by synthesis and technology mapping. Instead of modeling Boolean operators using pseudo-Boolean equations, we use the algebraic models 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 $GF(2^m)$, according to [1]:

[1]For polynomials in $GF(2)$, "+" is computed modulo 2.

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

B. Outline of the Approach

Similarly to the work of [4], the computed function of the circuits is specified by two polynomials. 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, with addition operation performed modulo 2 (e.g. $(a_0b_0 + a_1b_1)$ mod 2). For a $GF(2^m)$ multiplier, if the irreducible polynomial $P(x)$ is provided, Sig_{in} is known. Our goal is to transform the output signature, Sig_{out}, using polynomial representation of the internal logic elements, into the input signature Sig_{in} in $GF(2^m)$. The the goal of the verification problem is then to check if Sig_{in} = Sig_{out}, expressed in the primary inputs.

Theorem 1: *Given a combinational $GF(2^m)$ arithmetic circuit, composed of logic gates, described by algebraic expressions (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 for expressions in $\mathbb{Z}_{2^m}$ is provided in [4].

□

Theorem 2: *Let the number of logic elements (polynomials) in a $GF(2^m)$ multiplier be n. At each iterations, backward rewriting process generates n internal expressions, $F_0, F_1, ..., F_{n-1}$, such that every expression $F_i \in GF(2^m)$.*

Proof: Assuming that F_0=Sig_{out} and each $F_i \in GF(2^m)$, we prove that $F_{i+1} \in GF(2^m)$. Each variable in F_i represents output of some logic gate. During the rewriting process, this variable is substituted by a corresponding polynomial in Eq. (1). According to Theorem 1, resulting polynomial F_{i+1} correctly represents the function $F_{i+1} \in GF(2^m)$.

□

Theorems 1 and 2, together with the algebraic model in Eq. (1), provide the basis for polynomial reduction in backward rewriting in this work. This is described by Algorithm 1. Our method takes the gate-level netlist of a $GF(2^m)$ multiplier as input and first converts each logic gate into equations using Eq. (1). The output signature Sig_{out} is required to initialize the backward rewriting. The rewriting process starts with $F_0 = Sig_{out}$, and ends when all the variables in F_i are primary inputs. This is done by rewriting the polynomials representing logic elements in the netlist in a topological order [4]. Each iteration includes two steps: Step 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); Step 2) simplify the new expression by removing all the monomials (including constants) that evaluate to 0 in $GF(2)$ (line 3 and lines 7 - 10). The algorithm outputs the function of the design in $GF(2^m)$ after n iterations, where

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, and answer whether $Sig_{out}$==$Sig_{in}$
 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 the expression of $p_i$
 6:         $F_i \rightarrow F_{i+1}$
 7:         for each element/monomial $M$ in $F_{i+1}$ do
 8:             if the coefficient of $M\%2$==0
 9:             or $M$ is 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 $F_n$ =?$Sig_{in}$
```

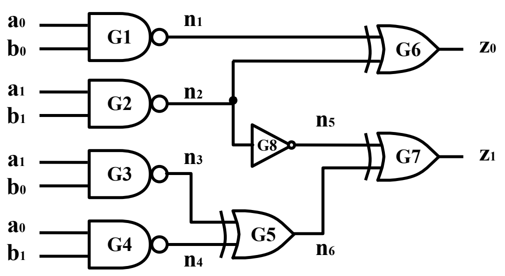

Fig. 2: The gate-level netlist of post-synthesized and mapped 2-bit multiplier over $GF(2^2)$. The irreducible polynomial $P(x) = x^2 + x + 1$.

n is the number of gates in the netlist. The final expression F_n can be used for functional verification, by checking if it matches the expected input signature (if provided).

Example 1 (Figure 2): We illustrate our method using a post-synthesized 2-bit multiplier in $GF(2^2)$, shown in Figure 2. 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, $F_0 = Sig_{out}$ is transformed into F_1 using polynomial of gate $G7_7$, z_1=n_5+n_6. This expression is simplified to $F_1 = z_0 + n_5x + n_6x$. Then, the polynomials F_{i+1} are successively derived from F_i and checked for a possible reduction. The first reduction happens when F_4 is transformed into F_5, where n_4 (at gate G_4) is replaced by $(1 + a_0b_0)$. After simplification, a monomial $2x$ is identified and removed from F_5 since 2%2=0. Similar reductions are applied during the transformations $F_6 \rightarrow F_7$ and $F_7 \rightarrow F_8$. Finally, the function of the design is extracted by Algorithm 1. A complete rewriting process is shown in Figure 3. We can see that $F_8 = Sig_{in}$, which indicates that the circuit indeed

Sig_{out}: F_0=z_0+xz_1	Eliminating terms
G7: F_1=z_0+x(n_5+n_6)	-
G6: F_2=n_1+n_2+x(n_5+n_6)	-
G5: F_3=n_1+n_2+x(n_3+n_4+n_5)	-
G8: F_4=n_1+n_2+x(n_3+n_4+n_2+1)	-
G4: F_5=n_1+n_2+x(n_2+n_3+a_0b_1)+2x	*2x*
G3: F_6=n_1+n_2+x(n_2+a_1b_0+a_0b_1+1)	-
G2: F_7=n_1+a_1b_1+1+x(a_1b_1+a_1b_0+a_0b_1)+2x	*2x*
G1: F_8=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)	-

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

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} is a polynomial ring). In other words, the reductions happen independently 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_5 and F_7 are extracted from output z_1 only. Similarly, in F_8, the reduction is from z_0.

Theorem 3: *Given a $GF(2^m)$ multiplier with $Sig_{out} = F_0 = z_0x^0 + z_1x^1 + ... + z_mx^m$; and $F_i=E_0x^0 + E_1x^1 + ... + E_mx^m$, where E_i is an algebraic expression in $GF(2)$ 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 $E_ix^{n_i}+E_kx^{n_k}$, where E_i and E_k are simplified in $GF(2)$. That is, $E_i = (e_i^1 + e_i^2 + ...)$, and $E_k = (e_k^1+e_k^2+...)$. After simplifying each of the two polynomials, there are no common monomials between $E_ix^{n_i}$ and $E_kx^{n_k}$. This is because for any element, $e_i^lx^{n_i} \neq e_k^jx^{n_k}$, for any pairs of (i,k) and (l,j). $\square$

IV. Implementation

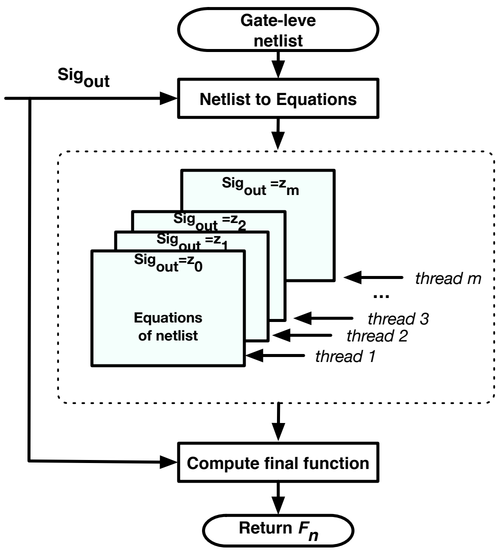

Fig. 4: Overview of parallel verification of GF multipliers.

This section describes the implementation of our parallel verification method for Galois field multipliers. The overview of the proposed technique is shown in Figure 4. Our approach takes the gate-level netlist as input, and outputs the extracted function of the design. It includes four steps:

1) Convert the gate-level netlist into algebraic equations. During this step, the gate-level netlist is translated into algebraic equations based on Eq.(1). The equations are levelized in topological order, to be rewritten by backward rewriting in the next step.
2) Split the output signature of $GF(2^m)$ multipliers into m polynomials with $Sig_{out\_i}=z_i$. These new *signatures* are represented by m equation files.
3) Split the function of m output bits into m separate functions, each to be processed by a separate thread using Algorithm 1. In contrast to work of [4], the internal expression of each output bit does not offer any polynomial reduction (*monomial cancellations*) for other bits.
4) 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.

Sig_{out0}=z_0	elim	Sig_{out1}=x·z_1	elim
G7: z_0	-	G7: x(n_5+n_6)	-
G6: n_1+n_2	-	G6: x(n_5+n_6)	-
G5: n_1+n_2	-	G5: x(n_3+n_4+n_5)	-
G8: n_1+n_2	-	G8: x(n_3+n_4+n_2)+x	-
G4: n_1+n_2	-	G4: x(n_2+n_3+a_0a_1)+2x	2x
G3: n_1+n_2	-	G3: x(n_2+a_1b_0+a_0b_1)+x	-
G2: n_1+a_1b_1+1	-	G2: x(a_1b_1+a_1b_0+a_0b_1)+2x	2x
G1: a_0b_0+a_1b_1+2	2	G1: x(a_1b_1+a_1b_0+a_0b_1)	-
Sig_{in}=a_0b_0+a_1b_1+x(a_1b_1+a_1b_0+a_0b_1)			

Fig. 5: Parallel extraction of a 2-bit GF multiplier shown in Figure 2.

Example 2 (Figure 5): We illustrate our parallel extraction method using the 2-bit multiplier in $GF(2^2)$ in Figure 2. 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 as Sig_{out0}+$Sig_{out1}x$ resulting in the polynomial Sig_{in}. In Figure 3, we can see that elimination happens three times (F_5, F_7, and F_8). According to Theorem 3, we know that the elimination happens within each element in GF(2^n). In Figure 5, one elimination in Sig_{out0} and two eliminations in Sig_{out1} have been done independently, as shown earlier (Figure 3).

V. Results

The verification technique described in this paper was implemented in C++. It performs backward rewriting with variable substitution and polynomial reductions in Galois field, using the approach discussed in Sections III and IV. The program was tested on a number of combinational gate-level $GF(2^m)$ multipliers taken from [1], including Montgomery multipliers [13] and Mastrovito multipliers [14]. The bit-width of the multipliers varies from 32 to 571 bits. The experiments of verifying Galois field multipliers using SAT, SMT, ABC [15] and Singular [16] have been presented in [1] and [5]. It shows that the rewriting technique performs significantly better than other techniques. Hence, in this work, we only compare our approach to those of [1] and [5]. Specifically, we compare our approach to the tool described in [5] on the same benchmark set. Our experiments were conducted on a PC with Intel(R) Xeon CPU E5-2420 v2 2.20 GHz x12 with 32 GB memory. As described in the next section, our technique is able to verify Galois field multipliers in multiple threads (up to 30 using our platform). In each thread, Algorithm 1 is applied on a single output bit. The number of threads is given as input to the tool.

A. Evaluation of Our Approach

The experimental results of our approach and comparison against [5] 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 [15] without any optimization. The largest circuit includes more than 1.6 million gates. This is also the number of polynomial equations and the number of rewriting iterations (see Section 3). The results generated by the tool, presented in [5] are shown in columns 3 and 4. We performed four different series of experiments, with a different number of threads, T=5, 10, 20, and 30. The runtime results are shown in columns 6 to 8 and memory usage in column 9. The timeout limit (TO) was set to 12 hours and

Mastrovito		[5]		This work				
Op size	# equations	Runtime (sec)	Mem (MB)	Runtime (sec)				Mem*
				T=5	*T=10*	*T=20*	*T=30*	*T=1**
32	5,482	0.83	3	1.90	1.54	0.95	1.09	10 MB
48	12,228	8.39	13	5.73	3.36	2.83	2.27	21 MB
64	21,814	28.90	21	11.08	7.88	6.87	6.74	37 MB
96	51,412	195.2	45	38.14	26.69	20.19	22.66	84 MB
128	93,996	924.3	91	91.67	62.68	54.99	56.76	152 MB
163	153,245	3546	161	192.6	137.5	120.7	113.1	248 MB
233	167,803	4933	168	294.1	212.7	180.1	170.6	270 MB
283	399,688	30358	380	890.7	606.5	549.7	529.8	642 MB
571	1628,170	*TO*	-	7980	5038	*MO*	*MO*	2.6 GB

TABLE I: Results of verifying Mastrovito multipliers using our parallel approach. T is the number of threads. TO=Time out of 12 hours. MO=Memory out of 32 GB.
(**T=1* shows the maximum memory usage of each thread.)

Montgomery		[5]		This work				
Op size	# equations	Runtime (sec)	Mem (MB)	Runtime (sec)				Mem*
				T=5	*T=10*	*T=20*	*T=30*	*T=1**
32	4,352	1.98	3	3.49	2.16	1.31	2.08	8 MB
48	9,602	14.19	13	17.71	10.67	9.16	6.01	16 MB
64	16.898	63.48	21	44.86	30.57	28.3	27.22	27 MB
96	37,634	554.6	45	234.3	157.8	133.1	142.3	59 MB
128	66,562	1924	68	208.9	121.3	115.8	110.4	95 MB
163	107,582	12063	101	1615.7	1172.3	1094.9	1008.1	161 MB
233	219,022	*TO*	168	722.3	564.8	457.7	479.8	301 MB
283	322,622	*TO*	380	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 12 hours. MO=Memory out of 32 GB.
(**T=1* shows the maximum memory usage of each thread.)

memory limit (MO) to 32 GB. The experimental results show that our approach provides on average 26.2x, 37.8x, 42.7x, and 44.3x 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 [5] fails after 12 hours.

Note that 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. These experiments are different than those in [5]. 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 the internal structure or clear boundaries of the blocks inside the design. The results are shown in Table II. For 32- to 163-bit Montgomery multipliers, our approach provides on average a 9.2x, 15.9x, 16.6x, and 17.4x speedup, for T = 5, 10, 20, and 30, respectively. Notice that [5] cannot verify the flattened Montgomery multipliers larger than 233 bits in 12 hours.

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 computation complexity depends not only on the bit-width of the multipliers, but also on the irreducible polynomial $P(x)$.

To analyze this dependency, we studied the effects of $P(x)$ on 4-bit multiplications implemented using different irreducible polynomials. The results are reported in Figure 6). We can see that when $P(x)_1$=$x^4 + x^3 + 1$, 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 $P(x)_2$=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 $P(x)_1$ and requires **6** XOR operations using $P(x)_2$. In other words, the gate-level implementation of the multiplier implemented using $P(x)_1$ has more gates compared to $P(x)_2$. In conclusion, we can see that irreducible polynomial $P(x)$ has significant impact on both design cost and the verification cost of the GF(2^m) multipliers.

$$
\begin{array}{ccccccc}
 & & & a_3 & a_2 & a_1 & a_0 \\
 & & & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
\hline
s_6 & s_5 & s_4 & s_3 & s_2 & s_1 & s_0
\end{array}
$$

$$
\begin{array}{c}
P(x)=x^4+x^3+1 \\
\begin{array}{c|c|c|c}
s_3 & s_2 & s_1 & s_0 \\
s_4 & 0 & 0 & s_4 \\
s_5 & 0 & s_5 & s_5 \\
s_6 & s_6 & s_6 & s_6 \\
\hline
z_3 & z_2 & z_1 & z_0
\end{array}
\end{array}
\qquad
\begin{array}{c}
P(x)=x^4+x+1 \\
\begin{array}{c|c|c|c}
s_3 & s_2 & s_1 & s_0 \\
0 & 0 & s_4 & s_4 \\
0 & s_5 & s_5 & 0 \\
s_6 & s_6 & 0 & 0 \\
\hline
z_3 & z_2 & z_1 & z_0
\end{array}
\end{array}
$$

Fig. 6: Analysis of the computation complexity of Galois field multipliers with different irreducible polynomials using two 4-bit GF multiplications, which are implemented using $x^4 + x^3 + 1$ and $x^4 + x + 1$.

B. Runtime and Memory Tradeoff

In this section, we discuss the tradeoff of runtime and memory usage of our approach. The plots in Figure 7 show how the average runtime and memory usage change with different number of threads. The vertical axis on the left is CPU runtime (in seconds), and on the right is memory usage (MB). Horizontal axis represents the number of threads T, ranging from 5 to 30. The runtime is significantly improved for T between 5 and 15. However there is not much speedup when T is greater than 20, most likely due to the memory

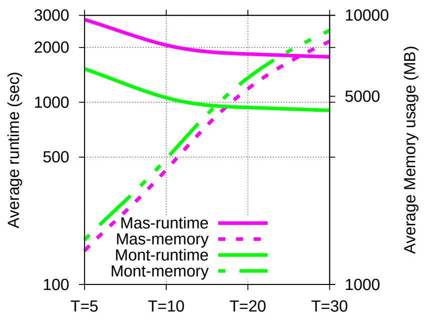

Fig. 7: Runtime and memory usage of our parallel verification approach as a function of number of threads T.

management synchronization overhead between the threads. Based on the experiments of Mastrovito multipliers (Table I), our approach is limited by the memory usage when the size of multiplier and T are large. In our work, $T = 20$ seems to be the best choice. Obviously, T varies on different platform depending on the number of cores, and the memory.

C. Verification of Synthesized GF Multipliers

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 efficiency of the rewriting technique relies on the amount of cancellations between the different terms of the polynomial, and the cancellations strongly depend on the order in which signals are rewritten. A good ordering of signals is difficult to be achieved in highly bit-optimized circuits.

In order 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 [15]. We repeatedly used the commands *resyn2* and *dch*[2] until ABC could not reduce the number of levels or the number of nodes any more. 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 observe that the Galois field multipliers are much easier to be verified after optimization. For example, the verification of a 283-bit Montgomery multiplier takes 15,300 seconds when T =20. After optimization, the runtime was just 169.2 seconds, which is 90x faster than verifying the original implementation. The memory usage is also reduced from 488 MB to 194 MB. In summary, in contrast to [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 this bit and does not require logic from other bits to be simplified (see Theorem 3). Hence, the complexity of function extraction is naturally reduced if logic cone is minimized.

VI. Conclusion

In this paper, we present an algebraic functional verification technique of gate-level $GF(2^m)$ multipliers, in which verification is performed in bit-parallel fashion. The method is based on extracting a unique polynomial in Galois field of each output bit independently. We demonstrate that this method is able to verify an n-bit GF multiplier in n threads, while applying on pre- and post-synthesized

Op size	*Mastrovito*		*Montgomery*	
	Runtime	Mem	Runtime	Mem
64	4.25 s	21 MB	15.3 s	38 MB
96	10.9 s	44 MB	40.5 s	54 MB
128	28.9 s	77 MB	27.1 s	78 MB
163	62.3 s	123 MB	205.2 s	153 MB
233	134.8 s	201 MB	141.4 s	199 MB
283	168.4 s	198 MB	169.2 s	194 MB
409	775.6 s	635 MB	750.6 s	597 MB

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

Mastrovito and *Montgomery* multipliers up to 571 bits. The results show that our parallel approach gives average 44× and 17× speedup compared to the best existing algorithm. In addition, we analyze the effects of irreducible polynomial and synthesis on verification of GF(2^m) multipliers.

Acknowledgment: This work was supported by an award from National Science Foundation, No. CCF-1319496 and No. 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] 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.

[3] 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.

[4] 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.

[5] T. Pruss, P. Kalla, and F. Enescu, “Equivalence Verification of Large Galois Field Arithmetic Circuits using Word-Level Abstraction via Gröbner Bases,” in *Proc. Design Automation Conference (DAC)*, 2014, pp. 1–6.

[6] C. Yu, W. Brown, and M. J. Ciesielski, “Verification of arithmetic datapath designs using word-level approach - A case study,” in *2015 IEEE International Symposium on Circuits and Systems, ISCAS 2015, Lisbon, Portugal, May 24-27, 2015*, 2015, pp. 1862–1865.

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

[8] R. E. Bryant and Y.-A. Chen, “Verification of Arithmetic Functions with Binary Moment Diagrams,” in *Design Automation Conference*, 1995, pp. 535–541.

[9] 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.

[10] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski, “Formal verification of arithmetic circuits using function extraction,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD).*, vol. PP, no. 99, pp. 1–12, March 2016.

[11] O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G.-M. Greuel, “An Algebraic Approach for Proving Data Correctness in Arithmetic Data Paths,” *CAV*, pp. 473–486, July 2008.

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

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

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

[15] A. Mishchenko *et al.*, “Abc: A system for sequential synthesis and verification,” *URL http://www. eecs. berkeley. edu/~ alanmi/abc*, 2007.

[16] 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.

[2]”dch” is the most efficient bit-optimization function in ABC.