Computer Algebraic Approach to Verification and Debugging of Galois Field Multipliers

Tiankai Su[1], Atif Yasin[1], Cunxi Yu[1,2], Maciej Ciesielski[1]
ECE Department, University of Massachusetts, Amherst, USA[1] LSI, EPFL, CH[2]
tiankaisu@umass.edu, ayasin@umass.edu, cunxi.yu@epfl.ch, ciesiel@umass.edu

***Abstract*—The paper presents a novel method to verify and debug gate-level arithmetic circuits implemented in Galois Field arithmetic. The method is based on forward reduction of the specification polynomials of the circuit in $GF(2^m)$ using $GF(2)$ models of its logic gates. We define a forward variable order "$FO >$" and the rules of forward reduction that enable verification, bug detection, and automatic bug correction in the circuit. By analyzing the remainder generated by forward reduction, the method can determine whether the circuit is buggy, and finds the location and the type of the bug. The experiments performed on Mastrovito and Montgomery multipliers show that our debugging method is independent of the location of the bug(s) and the debugging time is comparable to the time needed to verify the bug-free circuit.**

***Keywords*— Galois Field, Arithmetic circuits, Formal verification, Computer Algebra, Logic Debugging.**

I. Introduction

Galois Field (GF) arithmetic has numerous applications in digital communication, cryptography and security engineering, and formal verification of such circuits is of prime importance. A large body of work has been published on formal verification of arithmetic circuits, both integer arithmetic [1][2][3][4] and GF arithmetic circuits [5][6]. The most successful verification techniques for verifying arithmetic circuits are based on computer algebra [7]. In this approach, the verification problem is solved by proving that the implementation satisfies the specification, where the specification and the implementation are represented by polynomial rings, $F[x]$. A typical approach is based on polynomial reduction using *Gröbner Basis*, which transforms the verification problem to *membership testing* of the specification polynomial in the ideals [1][5]. An alternative approach to verify gate-level arithmetic circuits uses algebraic rewriting of the polynomial at the primary outputs (POs) into a polynomial at the primary inputs (PIs). The method, termed *backward rewriting*, is performed in a *reverse topological order* [4][6] and is characterized by a good runtime performance due to a large number of monomial cancellations during rewriting [6][8].

However, very little work has been done on *debugging* of arithmetic circuits. One such work reasons about the bugs in integer circuits by analyzing the remainder polynomial generated by backward rewriting [9][10]. However, it is limited by an uncontrollably large size of the remainder generated in case of a buggy circuit. Another approach combines backward rewriting with *forward rewriting*, carried from PIs to POs, and compares the two results to identify and correct the bugs [11]. This method also suffers from the polynomial size explosion in a buggy circuit and the complexity of forward rewriting.

In this paper we propose a novel approach to *debugging* of GF arithmetic circuits based on *forward rewriting*. This technique does not suffer from the polynomial size explosion encountered by other methods and allows one to identify and automatically correct the bugs in GF circuits. We limit our attention to bugs that are caused by using a wrong logic gate (e..g, using an AND gate instead of an XOR), referred to as *gate replacement*, which is the most common source of errors. Specifically, we make the following novel contributions:

- Use *forward* variable order "$FO >$" which enables performing verification and debugging at the same time.
- The method does not suffer from memory explosion problem.
- It can handle multiple dependent bugs.
- The performance of bug detection is not related to the location of the bug; the time it takes to verify a buggy circuit is about the same as verifying a bug-free circuit.

II. Background

A. Galois Fields

Galois field (GF) is a number system with a finite number of elements and basic arithmetic operations of addition, multiplication, and division. Of particular interest in hardware design for cryptography systems are *binary extension fields*, denoted GF(2^m) (or $\mathbb{F}_{2^m}$), which are finite fields with 2^m elements represented by polynomial rings $F[x]$. The addition of finite field elements is the addition of polynomials, with coefficients computed in GF(2). Multiplication of field elements in GF(2^m) is performed modulo *irreducible polynomial* $P(x)$ of degree m with coefficients in GF(2). Extension fields are used in many cryptography applications, such as *Advanced Encryption Standard* (AES) and *Elliptic-curve cryptography* (ECC).

B. Computer Algebra Approach

Algebraic approach used in this work relies on polynomial representation of the circuit specification and its logic gate components. *Input Signature* is the polynomial in terms of the PI variables that represents the expected function of the circuit, denoted Sig_{in}. For example, input signature for an n-bit binary adder is $Sig_{in} = \sum_{i=0}^{n-1} 2^i(a_i + b_i)$. *Output Signature* is a binary encoded polynomial expressed in the PO variables, denoted by Sig_{out}. Output signature of an unsigned arithmetic circuit with m output bits $\{z_i\}$ is $Sig_{out} = \sum_{i=0}^{m-1} 2^i z_i$. *Backward rewriting* is the process that transforms the output signature Sig_{out} into a unique input signature Sig_{in} using algebraic (polynomial) models of the logic gates of the circuit [4][6]. This work models Boolean operators using algebraic models of $GF(2)$. For example, the pseudo-Boolean model of XOR in integer arithmetic, XOR(a, b)=$a + b - 2ab$, is reduced in GF(2) to $(a + b + 2ab)$ mod 2 = $(a + b)$ mod 2. The following algebraic models are used to describe basic logic gates in $GF(2^m)$ [1][6]:

$$\begin{aligned} &\neg a = 1 + a \bmod 2 \\ &a \wedge b = a \cdot b \bmod 2 \\ &a \vee b = a + b + a \cdot b \bmod 2 \\ &a \oplus b = a + b \bmod 2 \end{aligned} \tag{1}$$

To address the debugging problem of gate-level GF circuits we define two orders by which monomials are listed in each polynomial:
Definition 1: *$BO >$ order: A backward, reverse-topological order, such that every output signal of a gate is always greater than its input signal.* The set of polynomials representing a logic cone ordered according to $BO >$ is called a BO base.
Definition 2: *$FO >$ order: A forward, topological order, such that every output signal of a gate is always less than its input signal.* The set of polynomials ordered according to $FO >$ is called an $\underline{FO \text{ base}}$.

For example, the polynomials for an OR gate z in these orders are:

$BO >$ order: $z + a + b + a \cdot b$,

$FO >$ order: $a + b + a \cdot b + z$.

The $BO >$ order has been used successfully in verification works of [5][8][6], but, as demonstrated in this paper, it is ineffective in debugging of GF circuits. The approach described in this paper is based on an $FO >$ order.

C. GF Multiplier Principles

Galois Field multiplication is performed modulo an irreducible polynomial $P(x)$, a polynomial that cannot be factored into nontrivial polynomials over the given field [12][13]. The inputs and outputs of GF(2^k) multiplication are k-bit binary numbers. An example of a 4-bit GF(2^4) multiplication, with irreducible polynomial $P(x)$=x^4+x^3+1, is shown in Figure 1.

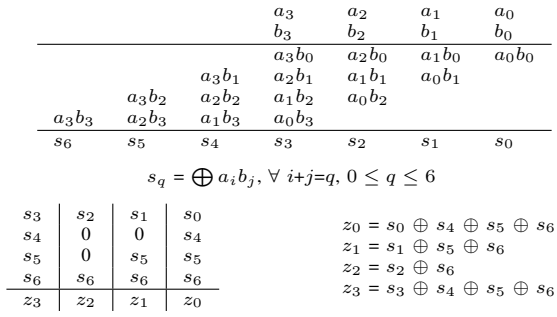

Fig. 1: Multiplication in GF(2^4): Z mod $P(x)$ = $A \cdot B$ mod $P(x)$, where $P(x)$=x^4+x^3+1.

The GF multiplication is performed in a straightforward way by: 1) generating and adding the partial products; and 2) reducing the result over GF(2^m) with $P(x)$. The partial products are generated in the same way as in the integer multiplication, using AND operations. However the sum of the partial products (denoted s_q in Figure 1) is obtained using a series of XORs, since additions in finite field are implemented as XOR operations. This multiplication structure is called Mastrovito multiplier [14]. Note that in GF arithmetic there is no carry propagation between the columns of the result bits. Hence, each bit can be computed separately as a linear (XOR) sum of the product terms in the respetive column. An alternative method for performing fast modular multiplication is the Montgomery multiplication [15]. It works by transforming two integer inputs, A, B, into Montgomery forms, $AR\ mod\ N$ and $BR\ mod\ N$, for some constant R, and computing the product $ABR\ mod\ N$. The multiplication result $A \cdot B$ is then obtained by transforming the result from the Montgomery form, as $A \cdot B = A \cdot B \cdot R^{-1} mod\ P(x)$. Since the Montgomery form generator is a GF multiplication with a constant input R, each of the four components can be considered as a simplified Mastrovito multiplication [16].

III. Bug Identification

This section describes an algorithm that utilizes the property of GF multipliers to identify bug(s) in the circuit. It consists of two major parts: 1) Remainder Generation and 2) Bug Analysis.

Algorithm I describes forward rewriting; it imposes the $FO >$ order on the polynomials and reduces the specification by the FO base. Each polynomial representing a logic gate in the FO base has the form: $\{head\}, z_i$. The $\{head\}$ is the set of head monomials representing the gate inputs; the tail monomial z_i is the output of the gate. The degree of the monomial is the sum of the degrees of all its variables. For example, polynomial of an XOR gate is $a + b + z_i$, where the head set is $\{a, b\}$, each of degree one, and the tail is z_i.

Algorithm 1 Remainder Generation via Forward Rewriting

Input: Gate-level netlist of the GF circuit
Input: Expected $InputSignature_i$ of each output bit
Output: Residual polynomials $\{Remainder_i\}$
Output: Non-zero elements of FO_i base

1: $\mathcal{PO}$=$\{z_0, z_1, ..., z_{n-1}\}$: primary output bits of the GF circuit
2: **for** $i \leftarrow 0$ **to** $n-1$ **do**
3: Extract cone PO_i from the gate-level netlist and generate the FO_i base
4: $Spec_i \leftarrow InputSignatrue_i - z_i$
5: **while** $Spec_i \neq 0$ **do**
6: $Success_{in} \leftarrow 0$
7: **for** each polynomial P_j in FO_i base **do**
8: **for** each monomial M_k in P_i **do**
9: **if** M_k divides $Spec_i$ **and** $Deg(M_k) == Deg(Spec_i)$ **and** M_k is not the last monomial in P_j **then**
10: Reduce $Spec_i$ by P_j; Set P_j in FO_i to 0; $Success_{in} \leftarrow 1$
11: **end if**
12: **end for**
13: **end for**
14: **if** $Success_{in} == 0$ **then**
15: Move the leading term of $Spec_i$ into $Remainder_i$
16: **end if**
17: **end while**
18: **end for**
19: **return** $\{Remainder_i$, non-zero FO_i base$\}$

The specification of the circuit is defined as $Sig_{in} - Sig_{out}$, the difference between the correct (expected) world-level input signature Sig_{in} and the output signature Sig_{out} of the binary encoded outputs. Since in a GF circuit the output bits are independent from each other, we can define $Spec_i$, the bit-level logic of output z_i, as $Sig_{in}(i) - z_i$. The input signature $Sig_{in}(i)$ is known, computed using the irreducible polynomial $P(x)$. The verification goal is then to reduce each specification $Spec_i$ by its FO_i base. If the result is 0, we conclude that the logic cone associated with bit i is bug-free. Otherwise, a non-zero $Remainder_i$ will indicate the presence of a bug (or bugs) in that cone.

The first step of Algorithm I is to extract all the cones from the circuit and derive the corresponding $Spec_i$ (lines 3-6). Recall that the FO_i base is the set of polynomials with an $FO >$ order describing the input-output relationship for cone i. In the process of creating logic cones for each output, the common logic is duplicated, effectively making each cone $fanout\text{-}free$[1]. This means that every intermediate signal will appear only once in the head monomials of FO_i base.

The next step is to reduce each $Spec_i$ by its corresponding FO_i base. In lines 8-10, the Algorithm scans the polynomials in the FO_i base to check if the leading term lt_i of $Spec_i$ is divisible by *any* of the $head$ monomials with the same degree as lt_i. This reduction is different than in the polynomial reduction based on the $BO >$ order, where only a *leading* monomial is used in the division. Allowing the use of *any* of the head monomials is essential in identifying the bugs.

As an example, consider the reduction of $Spec_i = a + b + R$, where R represents the remaining set of monomials, by polynomial $f = a + b + z$. The result of such a reduction should be $Spec_i = z + R$. However, if the monomial a is missing in $Spec_i$ due to a bug (this is the case when the gate with output a is false), i.e., when $Spec_i = b + R$, the head monomial b in f will be used to divide $Spec_i$. The result of such a reduction will be $Spec_i = a+z+R$. The monomial a in the reduced $Spec_i$ will never be eliminated by other polynomials in FO_i base, since there would be no other gate with signal a as input. When examining the content of the final remainder, it will be clear that the gate with output a is the source of the bug. This is exactly what will help us locate the bug in the Bug Analysis step.

[1]This is true for most structures such as Mastrovito multipliers, but may not be true for more complex ones, such as Montgomery; we currently limit our attention to those structures that can be made *reconvergent fanout free*.

In summary, once there's a polynomial P_i in FO_i base contains a head monomial that divides $Spec_i$, it will be used to reduce $Spec_i$ and then be set to 0 (avoid endless loop). In summary, if there is a polynomial P_i in FO_i base that contains a head monomial that divides $Spec_i$, it will be used to reduce $Spec_i$. The polynomial P_i will then be removed from the base (each base polynomial can only be used once during the reduction). However, if the leading term of $Spec_i$ is not divisible by any of the head monomials in the FO_i base, it will be moved into $Remainder_i$. This process will be repeated until $Spec_i$ becomes empty (lines 13-18); or until it cannot be reduced anymore, in which case the content of the Remainder will be used to identify the bug.

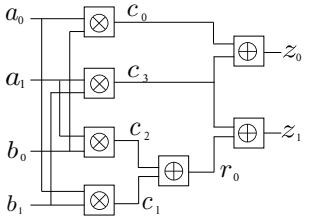

Fig. 2: A two-bit Mastrovito GF multiplier.

Example 1: Consider a bug-free two-bit Mastrovito multiplier in Figure 2. The circuit can be separated into two cones: z_0 and z_1. The FO_1 base of cone z_1 includes polynomials: $p_1 = a_0b_1 + c_1$; $p_2 = a_1b_0 + c_2$; $p_3 = a_1b_1 + c_3$; $p_4 = c_1 + c_2 + r_0$ and $p_5 = c_3 + r_0 + z_1$, each written in $FO >$ order. According to the definition, $Spec_1$ is equal to the input signature of z_1 minus z_1, the output signature. The expected input signature of the circuit is $F = (a_0 + Xa_1) \cdot (b_0 + Xb_1) = a_0b_0 + X(a_0b_1 + a_1b_0) + X^2a_1b_1$. After $GF(2)$ reduction with the irreducible polynomial $P(X) = X^2 + X + 1$, we obtain $F = X(a_0b_1 + a_1b_0 + a_1b_1) + (a_0b_0 + a_1b_1)$. Hence, for output z_1, associated with X, we have $Spec_1 = (a_0b_1 + a_1b_0 + a_1b_1) + z_1 (mod\ 2)$. Similar expression can be derived for $Spec_0$ of output z_0.

The next step is to reduce the specification $Spec_1$ by the polynomial base of FO_1. This process is shown in Figure 3. We can see that $Spec_1$ is eventually reduced to 0. That is, $Remainder_1 = 0$ and all polynomials in FO_1 have been used, which indicates that cone z_1 is bug-free. We proceed similarly with bit z_0 to determine that its logic cone is also bug-free.

Example 2: Let us now consider the case when there is a bug in the-bit multiplier in Figure 2. Let the bug be caused by replacing the XOR gate r_0 with an AND gate in cone z_1. That is, polynomial $p_4 = c_1 + c_2 + r_0$ is replaced by $p_4 = c_1c_2 + r_0$ in the FO_1 base, while $Spec_1$ remains the same.

The process of reducing $Spec_1$ by the new FO_1 base is shown in Figure 3. Note that in the 4th iteration of the reduction, the polynomial $c_1 + c_2$ in $Spec_1$ is not divisible by any of the head monomials in FO_1, so it will be moved into $Remainder_1$. The monomial r_0 in the 5th iteration also be moved to $Remainder_1$.

As a result, we obtain a non-zero $Remainder_1 = c_1 + c_2 + r_0$, and a non-zero FO_1 base: $p_4 = c_1c_2 + r_0$. This is a clear manifestation of a bug; namely, the XOR gate $(c_1 + c_2 + r_0)$ has been replaced by an AND gate $(c_1c_2 + r_0)$, while during the reduction, polynomial $c_1 + c_2$ should have been canceled by r_0. In a similar fashion we can identify other types of errors caused by gate replacement.

Table I shows some common cases of erroneous gate-replacement in GF circuit for basic 2-input logic gates: AND, OR, and XOR. Similar relations can be derived for other gates as needed. In the table, signals a and b represent the inputs, and z is the output of the false gate. By analyzing the $Remainder_i$ and the non-zero FO_i base generated by the algorithm, we can readily determine the type of the error and locate the bug. This is possible because the remainder

Polynomials in FO_1 base of a bug-free cone	$Spec_1$=a_0b_1+a_1b_0+a_1b_1+z_1
p_1=a_0b_1+c_1	$Spec_1/p_1$=c_1+a_1b_0+a_1b_1+z_1
p_2=a_1b_0+c_2	$Spec_1/p_2$=c_1+c_2+a_1b_1+z_1
p_3=a_1b_1+c_3	$Spec_1/p_3$=c_1+c_2+c_3+z_1
p_4=c_1+c_2+r_0	$Spec_1/p_4$=2c_2+c_3+r_0+z_1 mod 2
p_5=c_3+r_0+z_1	$Spec_1/p_5$=2r_0+2z_1 = 0 mod 2
Polynomials in FO_1 base of a buggy cone	$Spec_1$=a_0b_1+a_1b_0+a_1b_1+z_1
p_1=a_0b_1+c_1	$Spec_1/p_1$=c_1+a_1b_0+a_1b_1+z_1
p_2=a_1b_0+c_2	$Spec_1/p_2$=c_1+c_2+a_1b_1+z_1
p_3=a_1b_1+c_3	$Spec_1/p_3$=c_1+c_2+c_3+z_1
p_4=c_1c_2+r_0	$Spec_1/FO_1$=c_1+c_2+c_3+z_1
p_5=c_3+r_0+z_1	$Spec_1/p_5$=c_1+c_2+r_0+2z_1 mod 2

Fig. 3: Generating $Remainder$ with Forward Rewriting of bug-free and buggy logic cone of output bit z_1. $Remainder = 0$ for bug-free cone, and $Remainder = c_1 + c_2 + r_0$ for a buggy cone.

TABLE I: Bug Analysis

Bug Type	Correct gate	False gate	Remainder	Non-zero base
1	XOR	AND	$a + b + z$	$a \cdot b + z$
2	XOR	OR	$a \cdot b$	-
3	AND	XOR	$a \cdot b + z$	$a + b + z$
4	AND	OR	$a + b$	-
5	OR	XOR	$a \cdot b$	-
6	OR	AND	$a + b$	-

contains the names of the input and output signals of the false gate. Notice that the remainder for type 2 and 5 in Table I are the same $(a \cdot b)$; and the same is true for the remainders of type 4 and 6 $(a+b)$. However, since the location of the false gate is known by its inputs, we can trace it in the circuit netlist to determine which case it is.

The bug of type $2, 4, 5, 6$ are all associated with an OR gate, either as a correct or a false gate. Since the polynomial of the OR gate is $f = a + b + ab + z$, the $Spec$ polynomial can be reduced by f, regardless whether the inputs are in the sum form $(a + b)$ or in the product form $(a \cdot b)$. This means that the signature "wave" propagating through the circuit during forward rewriting will always hit the bug, so it will only leave the residual polynomial with their inputs $(a + b$ or $a \cdot b)$ in the remainder. As a result, they do not have any *non-zero base* and the *Remainder* contains only the input variables of the false gate, making it easy to identify the bug ($c.f.$ discussion in the next section). On the other hand, for bug types 1 and 3, both the input signals (a, b) and the output signal (z) of the false gate appear in the *Remainder*. Furthermore, a *non-zero base* indicates that the polynomial propagating through the circuit during forward rewriting cannot "go through" the bug. As we can see in the next section, the multiple bugs, when they appear together, will affect each other.

At this point the reader should fully appreciate the difference between forward rewriting (using an $FO >$ order) and backward rewriting (using a $BO >$ order). The backward rewriting for a bug-free circuit will also produce a zero remainder, as in the forward case, indicating that the circuit implements the correct function. However, in the buggy circuit the remainder will be different than in the forward case. By construction, such a remainder will contain only by the *primary inputs* (PI), but not the input and output signals of the *false gate*. Hence it does not provide sufficient information about the type of the bug and its location. For example, assuming the same bug as in Example 2 (i.e., XOR gate r_0 replaced with an AND gate), the backward rewriting would produce $Remainder_1 = a_0b_0a_1b_1 + a_1b_1$, with no indication as to the source of the bug. Furthermore, a single bug can make the backward remainder very large, making the analysis of the source of the bug difficult and the debugging process very hard. In contrast, in forward reduction, the remainder contains the input signature of the faulty gate and the location of the bug does not affect the size of the remainder.

IV. Multiple Bugs Analysis

The debugging method using forward reduction described in the previous section can be readily extended to multiple bugs. In general, arithmetic bugs can be divided into *independent* and *dependent* bugs. Independent bugs are those that will not affect each other; typically they appear in different cones. Since each cone is verified separately, each independent bug can be treated as a single bug in its own cone.

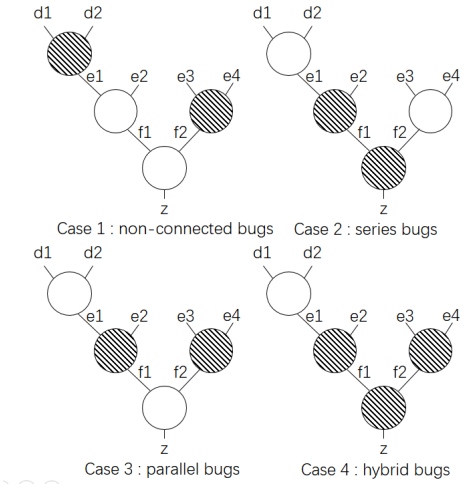

Fig. 4: Different cases for dependent bugs.

Dependent bugs can be classified into four cases, shown in Figure 4. The blank circle in the figure represents the correct gate. The shaded circle represents a *false gate*, i.e., the gate that was erroneously replaced by another gate. It can be of any type discussed in Table I, and even extended to other gates. Assume that in a correct implementation the gate e_1 is an AND gate and the remaining gates are XOR. With this, the specification of the cone with output z is $Spec = d_1d_2 + e_2 + e_3 + e_4 + z$.

The remainder in the case of multiple bugs depends not only on the cases shown in Figure 4, but also on the type of the bugs, listed in Table I (c.f. Section III). Recall that intermediate input variables appear exactly once in each fanout-free cone. For this reason, the remainder for the circuit with bugs of type 2, 4, 5, 6 of Table I (i.e., those that have only false inputs left in the remainder) will be composed of disjoint sets of complete input pairs. As a result, the bugs of these types will not affect each other, regardless how they are connected (c.f. Case 1, 2, 3, 4 in Figure 4). One just needs to identify the complete input pairs in the remainder to detect the bugs. On the other hand, as mentioned in Section III, for bug types 1 and 3 of Table I, both the input signals and the output signal of the false gate will appear in the *Remainder*. For this reason, when multiple bugs appear together, they will affect each other, as in the case 2, 3, and 4 in Figure 4,

Example 3: Consider Case 1 in Figure 4, where the bugs at e_1 and f_2 are not connected directly. Assume that the AND gate e_1 is replaced by an XOR, and the XOR gate f_2 by an AND gate. The remainder computed by Algorithm I, $Rem = d_1d_2 + e_1 + e_3 + e_4 + f_2$ can be partitioned into two disjoint groups, based on the input and output variables, such that each group forms a complete input-output pair: $(d_1d_2 + e_1)$ and $(e_3 + e_4 + f_2)$. In this case, it is always possible to achieve such a partition, since the two bugs are not connected directly. Specifically, the analysis of the non-zero base tells us that d_1, d_2, e_3, e_4 are the input signals (the head monomials of the respective base polynomials), and e_1, f_2 are output signal (the tail monomials). Therefore, both bugs can be detected, one for e_1 gate and the other for f_2 gate.

Analysis of the other cases in Figure 4, where the false signals interact with each other, is more complex, as illustrated by the following example.

Example 4: Consider Case 2 in Figure 4, when the XOR gates f_1 and z are replaced by the AND gates. Here we cannot find a complete input-output pair in $Remainder = e_1 + e_2 + f_2 + z$, because signal f_1 is not only the false output of gate f_1 but also the false input of gate z. In this case we need to search for incomplete pairs in order to properly identify the bugs. For Cases 3 and 4, with the same $Remainder = e_1 + e_2 + e_3 + e_4 + z$, the input-output pairs cannot be found. The result depends on how z can be expressed as a combination of f_1, f_2 that partitions the remainder into groups. The detailed discussion of these cases is beyond the scope of this paper.

V. Results and Conclusions

The debugging technique described in this paper was implemented in Python and interfaced with the computer algebra tool, Singular [17], to affect the polynomial reduction. The experiments were performed on an Intel® Core™ CPU i5-3470 @ 3.20 GHz × 4 with 15.6 GB memory, using Mastrovito multipliers up to 256 bits as benchmarks [14]. The more complex and challenging Montgomery multipliers [16] have only been partially tested and are not reported here.

TABLE II: Results of Mastrovito multipliers with single bug per cone.

Largest cone Operand size	# polys	Runtime for bug-free cone (sec)	Avg. runtime for buggy cone (sec)	Max runtime for buggy cone (sec)
z_5Mas16	167	0.36	0.36	0.37
z_{21}Mas64	539	6.2	6.3	6.3
z_{63}Mas128	1167	19.3	19.4	19.5
z_{10}Mas256	2033	46.3	46.5	46.6

Table II shows the verification results for a single bug inserted randomly in the circuit and illustrates the fact that the location of the bug does not affect the verification performance. To ensure that the location of the bug is the only variable factor in this experiment, for each circuit we extracted the largest output logic cone and inserted a single bug in it. The experiment was repeated for each cone 20 times, each time randomly changing the location of the bug, so they can be anywhere in the circuit and of any type shown in Table I. The average time of the experiments was computed and the worst case runtime it took to locate the bug recorded. As we can see in the Table, the longest and average times are similar, which means that the time to locate and correct the bug does not depend on its location in the circuit. Furthermore, the time to verify the bug-free circuit is almost the same as the debugging of a single bug. In other approaches, the difference between these two times can be significant [1] [18].

TABLE III: Results of Mastrovito multipliers with multiple bugs.

Operand size	Runtime for bug-free circuit (sec)	Result of [19]	Number of bugs	Avg. runtime for multiple bugs circuit (sec)
8	0.33	0.09	16	0.37
16	0.84	0.42	24	0.92
32	3.89	0.83	32	4.23
64	30.39	28.90	40	31.30
128	283.72	924.3	48	286.94
163	667.38	3,546.0	56	676.07
256	2,111.43	6,728.0	64	2,135.92

Table III shows the debugging results for Mastrovito multipliers with multiple bugs. It gives the time it takes to verify the bug-free circuit; makes comparison with [19]; shows the number of bugs and

the time to debug multiple bugs. The data of Table III, in conjunction with that in Table II, shows that the runtime for verifying the entire circuit is much less than the runtime of verifying a single cone multiplied by the number of cones. This is because the verification of each cone is performed in parallel since the cones are independent from each other.

Column 3 of Table III compares the performance of our mehtod with that of [19] that used backward reduction with the $BO >$ order and Groebner basis. As we can see, our results are superior for circuit sizes above 64 bits. This suggests that $FO >$ order can be a better choice for GF verification. The major advantage of the $FO >$ order, however, is the performance of debugging, as shown in columns 4 and 5 of Table III. We randomly inserted 8 bugs (dependent or independent) in each cone. The runetime difference between the bug-free circuit and a buggy circuit is negligible. Even in the largest case of 256-bit Mastrovito multiplier, with 64 bugs inserted, the runtime difference is insignificant.

In summary, our verification scheme based on forward reduction algorithm offers an effective method for identifying and removing bugs in GF circuits. Unlike in other methods, its performance does not depend on the location of the bug, and the time to locate the bug is comparable to verifying a bug-free circuit. Future work will extend the method to handle other GF circuits and more advanced multipliers, such as Montgomery. We will also consider other types of bugs, such as miswiring, missing gates, using wrong polarity, and others.

Acknowledgment: This work has been funded by the National Science Foundation, grant CCF-1617708.

REFERENCES

[1] J. Lv, P. Kalla, and F. Enescu, "Efficient Groebner 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 Groebner 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, "Efficient Symbolic Computation for Word-level Abstraction from Combinational Circuits for Verification Over Finite Fields," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. PP, no. 99, p. 1, November 2015.

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

[7] D. Cox, J. Little, and D. O'Shea, *Ideals, Varieties, and Algorithms*. Springer, 1997.

[8] 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.

[9] F. Farahmandi and P. Mishra, "Automated Test Generation for Debugging Arithmetic Circuits," in *Proceedings of the conference on Design, automation and test in Europe (DATE)*. EDA Consortium, 2016.

[10] Farimah Farahmandi and Prabhat Mishra, "Automated debugging of arithmetic circuits using incremental Groebner basis reduction," in *ICCD 2017, Boston, MA, USA. (to appear)*, 2017.

[11] S. Ghandali, C. Yu, D. Liu, B. Walter, and M. Ciesielski, "Logic Debugging of Arithmetic Circuits," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2015, pp. 113–118.

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

[13] T. Nagell, *Introduction to Number Theory*. Almqvist & Wiksell Stockholm, 1951.

[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] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

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

[17] 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.

[18] T. Su, C. Yu, A. Yasin, and M. Ciesielski, "Formal verification of truncated multipliers using algebraic approach and re-synthesis," in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2017, pp. 415–420.

[19] 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.