Polynomial Word-Level Verification of Arithmetic Circuits

Mohammed Barhoush
Institute of Computer Science,
University of Bremen
Bremen, Germany
barhoush@uni-bremen.de

Alireza Mahzoon
Institute of Computer Science,
University of Bremen
Bremen, Germany
mahzoon@uni-bremen.de

Rolf Drechsler
Institute of Computer Science,
University of Bremen
Bremen, Germany
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
drechsle@uni-bremen.de

ABSTRACT

Verifying the functional correctness of a circuit is often the most time-consuming part of the design process. Recently, world-level formal verification methods, e.g., *Binary Moment Diagram* (BMD) and *Symbolic Computer Algebra* (SCA) have reported very good results for proving the correctness of arithmetic circuits. However, these techniques still frequently fail due to memory or time requirements. The unknown complexity bounds of these techniques make it impossible to predict before invoking the verification tool whether it will successfully terminate or run for an indefinite amount of time.

In this paper, we formally prove that for integer arithmetic circuits, the entire verification process requires at most linear space and quadratic time with respect to the size of the circuit function. This is shown for the two main word-level verification methods: backward construction using BMD and backward substitution using SCA. We support the architectures which are used in the implementation of integer polynomial operations, e.g., $X^3 - XY^2 + XY$. Finally, we show in practice that the required space and run times of the word-level methods match the predicted results in theory when it comes to the verification of different arithmetic circuits, including exponentiation circuits with different power values (X^P : $2 \leqslant P \leqslant 7$) and more complicated circuits (e.g., $X^2 + XY + X$).

CCS CONCEPTS

• **Hardware → Functional verification**;

KEYWORDS

formal verification, verification complexity, symbolic computer algebra, binary moment diagrams

ACM Reference Format:
Mohammed Barhoush, Alireza Mahzoon, and Rolf Drechsler. 2021. Polynomial Word-Level Verification of Arithmetic Circuits. In *19th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE '21), November 20- -22, 2021, Beijing, China.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3487212.3487333

MEMOCODE '21, November 20–22, 2021, Beijing, China

ACM ISBN 978-1-4503-9127-6 / 21/11…$15.00
https://doi.org/10.1145/3487212.3487333

1 INTRODUCTION

Arithmetic circuits are involved in a wide range of computation-intensive applications, (e.g. cryptography and signal processing) as well as in upcoming AI architectures (e.g. for machine learning). These circuits usually contain many logic gates, and their architectural complexity is high due to the multitude of multiplication and addition/subtraction units. As a result, they are very error-prone. It is, therefore, necessary after construction to verify that the circuit performs its intended function.

Several formal verification methods have been proposed to ensure the correctness of circuits: (a) bit-level verification methods (e.g., *Binary Decision Diagrams* (BDDs) [8] and *Boolean satisfiability* (SAT) [5, 13]) are very fast in the verification of adders but they quickly run out of memory when it comes to the verification of multipliers, (b) *Theorem Proving* [18] and *Term Rewriting* [15, 26] techniques can verify both multipliers and adders; however, they are not fully automated, and they usually require the update of the rewrite rules, and (c) word-level verification methods (BMDs [2, 4], and SCA [12, 16, 21–24, 27, 28]) have recently reported very good results for both multipliers and adders; as a result, they are powerful tools for checking the correctness of integer arithmetic circuits.

The word-level verification of arithmetic circuits consists of four phases: (1) representing the output of the arithmetic circuit as a word-level polynomial, (2) capturing the logical gates (or building blocks) of the circuit also as a set of polynomials, (3) step-wise substitution of the gate polynomials in output polynomials based on the reversed topological order of the circuit, (4) checking the obtained polynomial at the input to see whether it matches the word-level specification of the circuit. Both SCA-based and BMD-based methods use the aforementioned verification flow. The only difference is in the way that they represent polynomials. In SCA-based verification, the expanded polynomials are directly used during the verification process. However, in BMD-based verification, these polynomials are represented in the form of more compact diagrams (e.g., BMD and *BMD).

While formal methods including word-level techniques are theoretically robust and effective, in practice they sometimes fail due to time or memory constraints. These problems arise due to our lack of understanding regarding the complexity of different verification approaches. In particular, despite the significant progress of word-level verification, the research on their time and space complexity is very limited and thus their performance is unpredictable.

In this paper, we analyze the space and time complexity of verifying integer arithmetic circuits using two word-level verification

techniques, i.e. SCA and BMD. We prove that the arithmetic circuits can be verified in linear space and quadratic time with respect to the size of the circuit function. For instance, consider a multiplier on n-bit inputs X and Y. This circuit computes the function $X \cdot Y = \sum_{0 \leq i,j \leq n-1} 2^{i+j} x_i y_j$ which has size $O(n^2)$. Our results entail that the space and time required to verify the circuit are bounded by $O(n^2)$ and $O(n^4)$, respectively. Note that it is well known that the BMD and polynomial representations of a multiplier have quadratic size, however we find the bounds on the entire construction process. Finally, we use an extensive number of integer arithmetic benchmarks, including exponentiation circuits and more complicated arithmetic circuits, to compare the theoretical calculations with the experimental results. Thus, we show the correctness of the obtained verification complexities in practice.

The rest of the paper is structured as follows: Section 3 reviews the required preliminaries. SCA and BMD verification approaches are introduced and a more detailed description is given of the circuits covered in this paper. Then, in Section 4, the similarities of the BMD and the SCA approaches are highlighted. In Section 5, the verification of an arithmetic circuit is shown to take linear space and quadratic time with respect to the size of the circuit function. The BMD and polynomial representations of the circuit are built in a backward manner and the complexity of each step of the process is taken into account. Experimental results are given in Section 6, which support the theoretical bounds. Finally, Section 7 concludes the paper.

2 RELATED WORK

Despite the rapid progress of formal verification methods, the research on their complexities is very limited. PolyAdd [6] for the first time proved that the complete formal verification process of some adder architectures (i.e., ripple carry adder, conditional sum adder, and carry look-ahead adder) can be carried out polynomially using BDDs. The author showed that the underlying BDDs remain polynomial during the whole construction process. It was ensured by proving upper bounds on the BDD sizes for each internal signal. The authors of [19] extended PolyAdd by extracting the precise complexity bounds for the conditional sum adder. The complexity bounds for the prefix adders and the proof of polynomial formal verification were presented in [20]. Moreover, some research works have been recently done on polynomial BDD construction of totally symmetric functions [9], polynomial formal verification of tree-like circuits [7], and polynomial formal verification of *Arithmetic Logic Units* (ALUs) [10]. Overall, these works only focus on polynomial BDD-based verification of different architectures.

To the best of our knowledge, [17] is the only work on the complexity of word-level formal verification. The authors analyzed *BMD-based verification applied to the class of Wallace-tree like multipliers. They formally proved polynomial upper bounds on runtime and space requirements with respect to the input word sizes. They showed that the whole verification process is bounded by $O(n^2)$ with respect to space and $O(n^4)$ with respect to time, where n in the number of input bits. The proof in this work is only limited to Wallace-tree like multipliers, and it does not support other types of multipliers as well as other types of arithmetic circuits.

3 PRELIMINARIES

This section first provides an overview of SCA-based and BMD-based verification and then reviews the general structure of an arithmetic circuit.

3.1 Symbolic Computer Algebra

We briefly summarize some terminologies:

- A *Monomial* is a power product of variables:
$$M = x_1^{\alpha_1} x_2^{\alpha_2} \dots x_n^{\alpha_n}, \tag{1}$$
where $\alpha_i \in \mathbb{N}$.
- A *Polynomial* is a finite sum of monomials:
$$P = c_1 M_1 + \dots + c_j M_j, \tag{2}$$
where c_i is an integer coefficient, i.e., $c_i \in \mathbb{Z}$.
- The ordering of monomials in a polynomial is determined by the ordering of variables and their powers in each monomial [3], e.g., under the variable ordering $x > y$, the monomials xy^2, y^4, y have the following ordering:
$$xy^2 > y^4 > y. \tag{3}$$

The goal of SCA-based verification is to formally prove that the gate-level netlist and the word-level function of the circuit are equivalent. The word-level function of a circuit is determined by two polynomials: *Output Signature* (*OS*) and *Input Signature* (*IS*). Output signature is a polynomial that depicts the word-level representation of the circuit's output, e.g., for the *Half-Adder* (HA) of Fig. 1 the output signature equals $2c + s$. Input signature is a polynomial that represents the word-level function of the circuit based on the inputs, e.g., for the HA of Fig. 1 the input signature equals $a + b$ which is the addition of two 1-bit numbers.

Before verification, the word-level relation between the inputs and outputs of gates should be extracted. Assuming, z is the output and a and b are inputs of a gate, the word-level function of the four main gates are as follows:

$$\begin{aligned} &z = \neg a \Rightarrow z = 1 - a, && z = a \vee b \Rightarrow z = a + b - ab, \\ &z = a \wedge b \Rightarrow z = ab, && z = a \oplus b \Rightarrow z = a + b - 2ab. \end{aligned} \tag{4}$$

As a result, in Fig. 1, the gates' polynomials can be represented as follows:

$$\begin{aligned} s &= a + b - 2ab, \\ c &= ab. \end{aligned} \tag{5}$$

In order to prove the correctness of the circuit, first, the gates are ordered based on the reverse topological order of the circuit, i.e., from *Primary Outputs* (PO) to *Primary Inputs* (PI). Then, starting from the output signature *OS*, the variable representing a gate's output is substituted by the gate's polynomial in *OS* to get a new polynomial. This process continues until we reach the PI and finish substituting all gates. If the final polynomial in inputs equals input signature *IS*, the circuit is correct. Otherwise, the circuit is buggy. This process is called *backward substitution*.

The process of backward substitution for the HA in Fig. 1 is as follows:

$$2c + s \xrightarrow{AND} 2ab + s \xrightarrow{XOR} \cancel{2ab} + a + b - \cancel{2ab} = a + b, \tag{6}$$

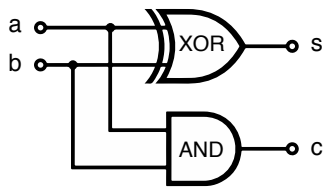

Figure 1: Half-Adder

since the final polynomial is equal to the input signature of a HA; thus, the circuit is correct. The polynomials which are obtained during the verification process, e.g., $2ab + s$, are referred to as *intermediate polynomials*.

The complexity of SCA-based verification is determined by the time and space required for each substitution step. This in turn largely depends on the size of the intermediate polynomials. For instance, to substitute the variable c with ab in $2c+s$ the polynomial is searched for the variable c, then the coefficient of c is multiplied with ab, and then the result is added to the rest of the polynomial.

3.2 Binary Moment Diagrams

Binary Decision Diagrams (BDDs) are built using the *Boole-Shannon decomposition*:

$$f = (1 - x) \cdot f_{\bar{x}} + x \cdot f_x, \tag{7}$$

where f_x is f evaluated at $x = 1$, and $f_{\bar{x}}$ is f evaluated at $x = 0$. By rearranging the terms, we get the *moment decomposition*:

$$f = f_{\bar{x}} + x \cdot f_{\dot{x}}, \tag{8}$$

where $f_{\dot{x}} := f_x - f_{\bar{x}}$ is called the *linear moment* of f with respect to x.

The BMD of f is a directed, rooted, acyclic graph which results from the recursive moment decomposition of f with respect to each of its variables. Every non-terminal node is labeled with a variable x, and has two successors representing the two parts of the decomposition of f with respect to x. The edge pointing to the successor $f_{\dot{x}}$ is called a *high-edge* while the edge pointing to $f_{\bar{x}}$ is called the *low-edge*. The terminal vertices have no successors and are labeled with integer values. The BMD structure is ordered, meaning that variables always appear once and in the same order throughout any path from the root to a terminal vertex. Furthermore, it is reduced, meaning there are no isomorphic sub-graphs or unnecessary nodes. A node is said to be unnecessary if the high-edge points to the terminal node labeled 0 which would mean that the function does not depend on the variable.

For instance the BMD of

$$F = W_0 + \sum_{i=1}^{n} W_i X_i \tag{9}$$

is shown in Fig. 2. Notice the BMD of linear functions contains the same number of non-terminal nodes and variables.

In general, BMDs are more efficient than BDDs in representing arithmetic functions. This is why BMDs are more widely used in the verification of multipliers and other polynomial circuits.

An extension of these structures is the *BMD which additionally makes use of edge-weights [4]. On each edge, an integer can be assigned which allows us in many cases to further reduce the graph into a smaller representation. While this is useful in function representation, using edge weights generally only increases the verification complexity. This is because the *BMD needs to be unfolded into its BMD form to perform operations like addition and multiplication which are required in verification.

BMDs can be used to verify if a circuit meets its specification. This is done by constructing the BMD of the circuit and comparing it to that of the specification. There are several ways to perform this construction; however, in this paper, we use a method called *backward construction* [14]. First, the BMD of the output signature is built. Notice that the variables in this BMD are the outputs of gates (or building blocks) in the circuit. The next step is to substitute each of these variables with their corresponding gate inputs. A new graph is constructed using the BMD of the previous step based on the following substitution formula:

$$F|_{x \leftarrow G} = F_{low(x)} + G \cdot F_{high(x)}, \tag{10}$$

where x is a node we want to substitute with the BMD G and $F_{low(x)}$ ($F_{high(x)}$) represents the BMD subgraph extending from the low-edge (high-edge) of node x. Hence, every step involves multiplication and addition of BMDs. After completing all the substitutions, the result is a BMD on the input variables representing the function of the circuit. An equivalence test with the BMD representing the specification function verifies whether the circuit is correct. Constructing the BMD of a specification function and running the equivalence test are both easy steps so the complexity of this verification method lies in the construction of the BMD of the circuit.

3.3 Integer Arithmetic Circuits

An integer arithmetic circuit computes a polynomial on its integer inputs. For the rest of the paper, we study a circuit C with n-bit binary input $X = x_{n-1}x_{n-2}...x_0$ which computes a polynomial of degree d on its inputs. A polynomial is said to have degree d if one of the terms has degree d while the rest of the terms have degree at most d. As an example, a circuit that computes polynomial $X^3 + X$ is an integer arithmetic circuit of degree 3. All the results easily generalize to circuits with more than one input.

Arithmetic circuits are usually implemented in two different ways (Chapter 2 in [25]):

(1) The partial products for each term are computed. Then, they are reduced to get the final results.
(2) Independent multiplication, exponentiation, addition and subtraction units are used in different levels to construct the arithmetic circuit.

As an example, the arithmetic circuit that computes $X^2 + XY$ can be implemented by generating partial products for X^2 and XY and subsequently reducing the partial products to get the final results. Alternatively, X^2 and XY can be implemented using an exponentiation and a multiplication unit, respectively. Then, the outputs are added using an addition unit.

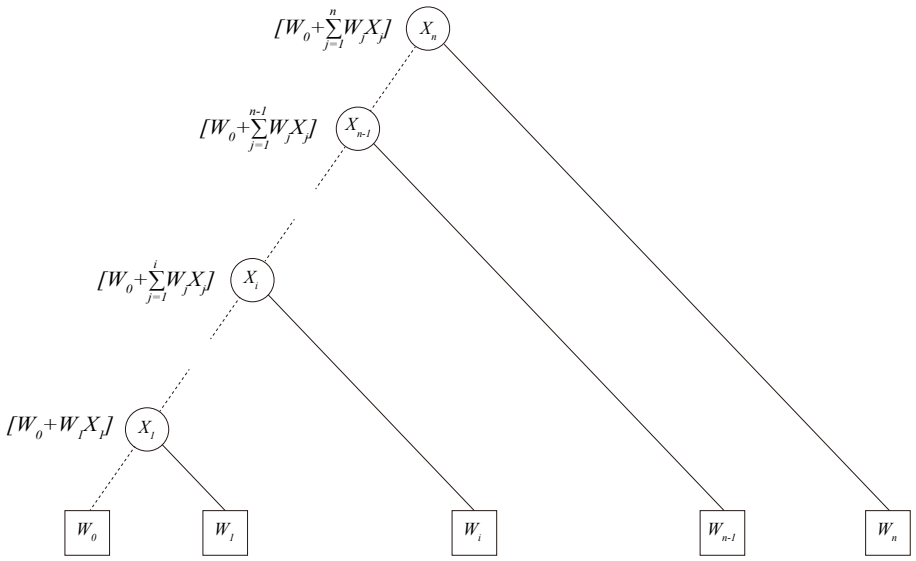

Figure 2: BMD of *F*

Arithmetic units built using the first approach as well as multipliers and exponentiation circuits in the second approach are made up of three stages:

(1) **Partial Product Generator (PPG):** This part generates all the partial products in the polynomial using $O(n^d)$ AND gates and sometimes NOT gates.
(2) **Partial Product Accumulator (PPA):** This part adds the partial products in the same bit position together until two bits are left in every position. There are several algorithms to do this such as array, Wallace tree, and balanced delay tree. All of these architectures use $O(n^d)$ adder cells.
(3) **Final Sum Adder (FSA):** At the end of the previous stage, two integers are obtained which need to be added. This is done in this stage using $O(n)$ adder cells.

Fig. 3 shows the three-stage structure of a squaring unit on a 3-bit integer $X = X_2X_1X_0$.

The adders and subtractors covered in this paper are the standard ripple-carry adders and ripple-borrow subtractors (Chapter 2 in [25]). These are built from series of full-adder and full-subtractor cells, respectively. A full-adder cell is similar to a half-adder but takes into account a carry bit from the previous lower bits while a full-subtractor computes the subtraction of two bits taking into account a borrow from the lower bits. The results in this paper cover all the circuits described in this section, i.e. circuits built from arithmetic units, ripple-carry adders, and ripple-borrow subtractors.

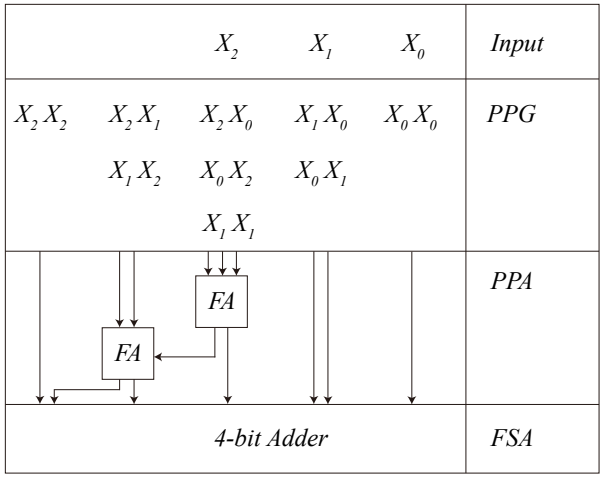

Figure 3: 3-bit Squaring Circuit

4 RELATION BETWEEN BMD AND SCA

In this section, the lemmas which are necessary to highlight the similarity of the BMD and SCA verification approaches will be proved. They allow us to analyze both approaches simultaneously.

Lemma 1. *Let $P(\mathcal{X})$ be a polynomial on $\mathcal{X} = \{x_0, ..., x_{n-1}\}$ and let t and m denote the number of terms and the number of multiplications in P, respectively. Then, the size of the BMD of P is bounded by $m + t$.*

Proof. It is clear that the term $cx_0^{\epsilon_0}x_1^{\epsilon_1}...x_{n-1}^{\epsilon_{n-1}}$, where $\epsilon_i \in \{0, 1\}$, belongs to P if and only if the path $\epsilon_0, \epsilon_1, ..., \epsilon_{n-1}$ in the BMD of P ends in the terminal node c.

Therefore the term $x_0^{\epsilon_0}x_1^{\epsilon_1}...x_{n-1}^{\epsilon_{n-1}}$ contributes at most $\epsilon_0 + \epsilon_1 + ... + \epsilon_{n-1}$ to the size of the BMD. This is the same as the number of multiplications in the term minus 1. □

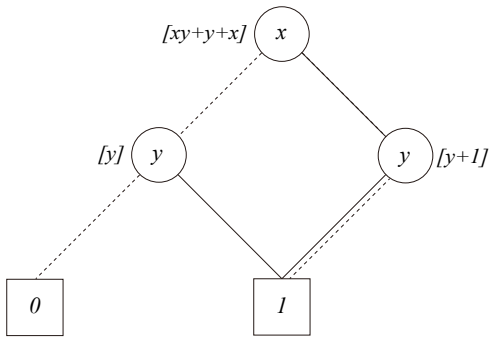

Figure 4: BMD of p

This is relevant since the size of a polynomial, i.e. its memory requirement, is equal to $m + t$.

For instance, the function $p(x, y) = xy + x + y$ has size 4 since it has 3 terms and 1 multiplication. By Lemma 1, the BMD of p should be bounded by 4. As seen in Fig. 4, the BMD of p has 3 non-terminal nodes so the bound holds.

For the rest of the paper let $\mathcal{X} = \{x_0, ..., x_{n-1}\}$ and let $P_d(\mathcal{X})$ denote a polynomial of degree d on $\mathcal{X}$.

Corollary 1. *The size of the BMD representation of $P_d(\mathcal{X})$ is bounded by $O(n^d)$.*

Another similarity between BMDs and their polynomials is the complexity of performing operations as stated in Lemma 2 [11].

Lemma 2. *The space and time required to add two polynomials (or BMDs) A and B is $O(|A| + |B|)$ and $O(|A| \cdot |B|)$, respectively. Similarly, the space and time required for multiplication is $O(|A| \cdot |B|)$.*

We say a variable x is linear in a polynomial P if all the terms in P that include x are linear.

Lemma 3. *Let P be a polynomial and let $|P|$ denote its size. Substituting a variable x that is linear in P with a function F of constant size takes $O(|P|)$ time and space. Furthermore, this holds whether the substitution is performed on the polynomial or the BMD.*

Proof. The lemma is clear when the substitution is performed on a polynomial. Only the case when it is performed on the BMD needs to be shown. Let $\bar{P}$ and $\bar{F}$ denote the BMDs of P and F, respectively. By Lemma 1, the size of the BMD of P is bounded by $|P|$. Since x is linear in P, to perform the substitution we need to carry out the operation $\bar{P}_{low(x_i)} + \bar{F} \cdot \bar{P}_{high(x_i)}$ where $\bar{P}_{high(x_i)}$ ($\bar{P}_{low(x_i)}$) is the subgraph extending from the high-edge (low-edge) of node x_i. This is done in two steps. First we multiply $\bar{P}_{high(x_i)}$ with $\bar{F}$. Since $\bar{P}_{high(x_i)}$ is a constant, this takes constant time and space. Next we add the BMD in the first step to $\bar{P}_{low(x_i)}$ which takes $O(|\bar{P}_{low(x_i)}|) = O(|P|)$ time and space. □

Lemma 1 shows that the size of a BMD is smaller than the size of its polynomial representation. While Lemma 3 shows that a substitution step requires the same space and time whether it is performed on a BMD or the corresponding polynomial (which is done in the SCA-based verification). In the rest of the paper, the focus is on bounding the complexity of the SCA approach which, by the previous lemmas, also bounds the BMD approach.

5 BOUNDING THE VERIFICATION COMPLEXITY

The circuit we analyze computes a polynomial of order d on n-bit input variable X and is built from arithmetic units, adders and subtractors as described in Section 3.3. We order the variables in the circuit so that the the outputs of a gate are consecutive and are higher than the inputs. Essentially, this means that the ordering is based on the gates.

To understand the verification complexity of the whole circuit we first study a single arithmetic unit U of order d_U with input X_U and output Y_U.

We will analyze the complexity of substituting the variables Y_U with the variables X_U. Note that any output or input of a gate in the circuit is considered a variable. For simplicity, the proof assumes:

(1) The initial polynomial is of the form $A + B \cdot Y_U$ where A and B are polynomials independent of the variables in U.
(2) The variables in U are ordered consecutively so that the focus remains on the the substitutions in U and we do not have to worry about other variables.

After completing the proof it will become clear that the results hold even if these assumptions are not met.

Note that the input and output of an adder or arithmetic unit are of the same order. For instance, in Fig. 3 the output is a $(2n + 1)$-bit integer representing the square of a n-bit integer. Since the number of units are constant with respect to n, the input and output of any unit in the circuit is $O(n)$. Hence, X_U and Y_U are both in $O(n)$.

Lemma 4. *The size of an intermediate polynomial obtained during the backward substitution of U is bounded by $O(|A| + |B| \cdot n^{d_U})$.*

Proof. The unit U is made up of three stages: PPG, PPA, and FSA. The method of backward substitution (or backward construction) begins with the end of the circuit. The purpose of the PPA and FSA stages is to reduce the sum of $O(n^{d_U})$ partial products bits into the $O(n)$ bit output using adder cells. The sum of the outputs of an adder cells is equal to the sum of its inputs. The word-level relation between inputs and outputs of three adder cells (i.e. HA, FA, and 7:3 counter (CN)) is as follows:

$$\begin{aligned}
&HA(in: X, Y \quad out: C, S) \quad \Rightarrow \quad 2C + S = X + Y \\
&FA(in: X, Y, Z \quad out: C, S) \quad \Rightarrow \quad 2C + S = X + Y + Z \\
&CN(in: X, Y, Z, W, Q, I, J \quad out: C_2, C_1, S) \quad \Rightarrow \\
&4C_2 + 2C_1 + S = X + Y + Z + W + Q + I + J
\end{aligned} \tag{11}$$

This can be generalized to other adder cells and means that all the intermediate polynomials obtained in these stages are linear in terms of the variables in U.

An adder cell has a constant number of outputs and inputs so each substitution introduces a constant number of new variables. Since all the intermediate polynomials are linear, each substitution grows the size by $O(B)$. There are $O(n^{d_U})$ adder cells so the size of the polynomials in the FSA and PPA stages is bounded by $O(|A|+|B|\cdot n^{d_U})$.

After substituting all the adder cells, we reach the PPG stage. In this stage, product cells such as AND gates calculate the partial products from the input X_U. Notice that the substitution of a product cell can only increase the size of an intermediate polynomial. Hence all of the polynomials in this stage must be smaller than the polynomial in terms of the inputs of U. Since U performs an operation of order d on $O(n)$ bits, this polynomial has size $O(|A|+|B|\cdot n^{d_U})$. □

Lemma 5. *The backward substitution of the FSA and PPA stages in U requires $O(|A| + |B| \cdot n^{d_U})$ space and $O((|A| + |B| \cdot n^{d_U}) \cdot |B| \cdot n^{d_U})$ time.*

Proof. Recall that during the substitutions of the FSA and PPA, the intermediate polynomials remain linear with respect to the variables in U. This means that the substitution of any cell output involves multiplication with B only. The result then needs to be added to the rest of the terms.

By Lemma 2, the multiplication step takes $O(|B|)$ space and time while the addition step takes $O(|A| + |B| \cdot n^{d_U})$ space and $O((|A| + |B| \cdot n^{d_U}) \cdot |B|)$ time. This is just the time for a single substitution. As there are $O(n^{d_U})$ substitutions that need to be performed, the total time is bounded by $O((|A| + |B| \cdot n^{d_U}) \cdot |B| \cdot n^{d_U})$. □

The same bound is obtained for the PPG stage:

Lemma 6. *The backward substitution of the PPG stage in U requires $O(|A| + |B| \cdot n^{d_U})$ space and $O((|A| + |B| \cdot n^{d_U}) \cdot |B| \cdot n^{d_U})$ time.*

Proof. Let $\mathcal{P} = \{p_i\}_{i \in K}$ be the set of variables representing outputs of gates in the PPG stage. Assume the variable p_i needs to be substituted with its inputs and let C_{p_i} be the coefficient of p_i in terms of the variables in $\mathcal{P}$. This substitution involves multiplication with $B \cdot C_{p_i}$ and then addition with the rest of the terms.

The multiplication step takes $O(|B| \cdot |C_{p_i}|)$ space and time while the addition step takes $O(|A| + |B| \cdot (|C_{p_i}| + n^{d_U}))$ space and $O((|A| + |B| \cdot n^{d_U}) \cdot |B| \cdot |C_{p_i}|)$ time. This is just the time for a single substitution. The total time is bounded by $O((|A| + |B| \cdot n^{d_U}) \cdot |B| \cdot (\sum_{i \in K} |C_{p_i}|))$.

It is not difficult to see that the substitution of p_i increases the size of the polynomial by a quantity in the range $[\frac{|C_{p_i}|}{d}, d \cdot |C_{p_i}|]$. This means that $\sum_{i \in K} |C_{p_i}| \leq O(n^{d_U})$. As a result, the maximum space required is bounded by $O(|A| + |B| \cdot n^{d_U})$ and the total time required is bounded by $O((|A| + |B| \cdot n^{d_U}) \cdot |B| \cdot n^{d_U})$. □

Note that throughout the proof we assume that the circuit does not make any unnecessary computations. For instance, a product cell computing $x_i \cdot x_i$ is unnecessary since this is just x_i. Of course without this assumption there is no limit to the size and complexity of a circuit.

Arithmetic circuits may also contain adder or subtractor units so the backward substitution of these units needs to be analyzed. As mentioned, all units have inputs and outputs of size $O(n)$. A ripple-carry adder calculates the sum of two $O(n)$-bit integers using $O(n)$ adder cells. If Y_U represented the output of an adder unit, then with the same arguments used to bound the verification complexity of the FSA and PPA in arithmetic units, it can be shown that the backward substitution of an adder requires $O(|A| + |B| \cdot n)$ space and $O((|A| + |B| \cdot n) \cdot n)$ time. A ripple-borrow subtractor is the same as a ripple-carry adder, except the full-adders are replaced with full-subtractors, and hence requires the same resources.

Notice that space required to substitute the variables of any unit is bounded by the size of the polynomial after all the gates are substituted. This idea applies to all units including unit outputs in B. After all the substitutions, the final polynomial of B must be of order at most $d - d_U$, otherwise the order of the specification would be larger than d. So the size of B is bounded by $O(n^{d-d_U})$. Similarly the size of A must be smaller than the size of the input signature which is in $O(n^d)$. Plugging these values into the bounds of Lemma 5 and Lemma 6 gives us the following result.

Lemma 7. *Substituting the gates of any unit is bounded by $O(n^d)$ with respect to space and $O(n^{2d})$ with respect to time.*

Note that the circuit computes a polynomial of order d on n variables. Hence, the circuit function has size $O(n^d)$.

Since the number of arithmetic units is constant with respect to n, this gives us a bound on the whole backward substitution of the circuit.

Theorem 1. *For integer arithmetic circuits, the entire verification process using BMDs or SCA requires linear space and quadratic time with respect to the size of the circuit function.*

Throughout the proof, a certain variable ordering was assumed. In particular, it is not necessary that a state of the form $A + B \cdot Y_U$ is obtained during the substitution process and there might be some variables outside of U ordered in between the variables of U. However, this does not change the results obtained. For instance, even if B changes as a result of substitutions performed outside of U, the size of B is still bounded by $O(n^{d-d_U})$ which is all that is required. Each substitution still needs the same time and space and there are the same number of substitutions performed regardless of the variable ordering.

Note that the final result of the verification process is the circuit function which entails a linear lower bound on the space required for verification. This means the upper bound on space obtained in Theorem 1 is tight which can be clearly seen in the experiments.

6 EXPERIMENTS

We have implemented the SCA-based verifier in C++. The benchmarks for the exponentiation circuits and more complicated arithmetic circuits are generated using abc [1] blast command (`%blast`). All experiments are performed on an Intel(R) Core(TM) i7-8565U with 1.80 GHz and 24 GByte of main memory.

We experimented extensively in order to check the correctness of the final bounds obtained in Section 5. First, exponentiation circuits X^P were tested on a range of exponents $2 \leq P \leq 7$ and input bit lengths between 0 and 64. For each bit length n and each exponent P, a n-bit exponentiation circuit performing the operation X^P was constructed with the same structure as a single arithmetic unit described in Section 3.3. The circuit was then verified using backward substitution and the run time (seconds) and space (maximum polynomial size) used by the process were plotted against the circuit function size, i.e., n^P. The results show that there is a linear relationship between space and function size (see Fig. 5) and a quadratic relationship between run-time and function size (see

Fig. 6). Therefore, the experiments indicate that the bounds hold for circuits with high powers and scale as predicted by the theory.

The exponentiation circuits are made up of only one arithmetic unit. Therefore to test out more general cases two circuits were considered: $X^2 + X \cdot Y + X$ and $X^3 + X \cdot Y$. These circuits were constructed using a combination of multipliers, exponentiation units, and adders which led to a more complex design with multiple units. Again, there is a linear relationship between space and function size (see Fig. 5) and a quadratic relationship between run-time and function size (see Fig. 6).

7 CONCLUSION

This work analyzed the complexity of two word-level verification approaches on arithmetic circuits: backward construction with BMDs and backward substitution with SCA. The similarity of the two approaches was studied and it was deduced that the BMD approach is at least as efficient as the SCA approach. Based on this, the complexity of both processes was simultaneously bounded polynomially, i.e., the run-time was shown to increase quadratically with the circuit function size, while the space required was shown to increase linearly. The theoretical bounds and experimental results indicate that both approaches are effective in verifying arithmetic circuits with multiple units and variables.

This work is, however, limited in its scope. Arithmetic circuits which utilize optimized structures such as carry look-ahead adder or Booth's multiplier for faster addition or multiplication are not considered. Nevertheless, the ideas herein shall be extended to these circuits in future work.

ACKNOWLEDGMENTS

Parts of this work have been supported by DFG within the Reinhart Koselleck Project *PolyVer: Polynomial Verification of Electronic Circuits* (DR 287/36-1).

REFERENCES

[1] 2018. ABC: A System for Sequential Synthesis and Verification. available at https://people.eecs.berkeley.edu/~alanmi/abc/.
[2] Randal E. Bryant. 1995. Binary decision diagrams and beyond: enabling technologies for formal verification. In *International Conference on Computer-Aided Design*. 236–243.
[3] David A. Cox, John Little, and Donal O'Shea. 1997. *Ideals Varieties and Algorithms*. Springer.
[4] David John Dempster and Michael George Stuart. 2001. *Verification Methodology Manual - Techniques for Verifying HDL Designs*. Teamwork International.
[5] Stefan Disch and Christoph Scholl. 2007. Combinational Equivalence Checking Using Incremental SAT Solving, Output Ordering, and Resets. In *ASP Design Automation Conf.* 938–943.
[6] Rolf Drechsler. 2021. PolyAdd: Polynomial Formal Verification of Adder Circuits. In *IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*. 99–104.
[7] Rolf Drechsler. 2021. Polynomial Circuit Verification using BDDs. arXiv:2104.03024.
[8] Rolf Drechsler and Bernd Becker. 1998. *Binary Decision Diagrams – Theory and Implementation*. Kluwer Academic Publishers.
[9] Rolf Drechsler and Caroline Dominik. 2021. Edge Verification: Ensuring Correctness under Resource Constraints. In *Symposium on Integrated Circuits and System Design*.
[10] Rolf Drechsler, Alireza Mahzoon, and Lennart Weingarten. 2021. Polynomial Formal Verification of Arithmetic Circuits. In *International Conference on Computational Intelligence and Data Engineering*.
[11] Reinhard Enders. 1995. Note on the complexity of binary moment diagram representations. In *Int'l Workshop on Applications of the Reed-Muller Expansion in Circuit Design*. 191–197.
[12] Farimah Farahmandi and Bijan Alizadeh. 2015. Gröbner basis based formal verification of large arithmetic circuits using Gaussian elimination and cone-based polynomial extraction. *Microprocessors and Microsystems* 39, 2 (2015), 83–96.
[13] Evguenii I. Goldberg, Mukul R. Prasad, and Robert K. Brayton. 2001. Using SAT for combinational equivalence checking. In *Design, Automation and Test in Europe*. 114–121.
[14] Kiyoharu Hamaguchi, Akihito Morita, and Shuzo Yajima. 1995. Efficient Construction of Binary Moment Diagrams for Verifying Arithmetic Circuits. In *International Conference on Computer-Aided Design*. 78–82.
[15] Deepak Kapur and Mahadevan Subramaniam. 1998. Mechanical Verification of Adder Circuits using Rewrite Rule Laboratory. *Formal Methods in System Design* 13, 2 (1998), 127–158.
[16] Daniela Kaufmann, Armin Biere, and Manuel Kauers. 2019. Verifying Large Multipliers by Combining SAT and Computer Algebra. In *Int'l Conf. on Formal Methods in CAD*. 28–36.
[17] Martin Keim, Rolf Drechsler, Bernd Becker, Michael Martin, and Paul Molitor. 2003. Polynomial Formal Verification of Multipliers. *Formal Meth. in Sys. Des.* 22, 1 (2003), 39–58.
[18] Robert P. Kurshan and Leslie Lamport. 1993. Verification of a multiplier: 64 bits and beyond. In *Computer Aided Verification*. 166–179.
[19] Alireza Mahzoon and Rolf Drechsler. 2021. Late Breaking Results: Polynomial Formal Verification of Fast Adders. In *Design Automation Conf.*
[20] Alireza Mahzoon and Rolf Drechsler. 2021. Polynomial Formal Verification of Prefix Adders. In *Asian Test Symp.*
[21] Alireza Mahzoon, Daniel Große, and Rolf Drechsler. 2018. PolyCleaner: Clean your Polynomials before Backward Rewriting to Verify Million-gate Multipliers. In *International Conference on Computer-Aided Design*. 129:1–129:8.
[22] Alireza Mahzoon, Daniel Große, and Rolf Drechsler. 2019. RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers. In *Design Automation Conf.* 185:1–185:6.
[23] Alireza Mahzoon, Daniel Große, and Rolf Drechsler. 2021. REVSCA-2.0: SCA-based Formal Verification of Non-trivial Multipliers using Reverse Engineering and Local Vanishing Removal. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021).
[24] Alireza Mahzoon, Daniel Große, Christoph Scholl, and Rolf Drechsler. 2020. Towards Formal Verification of Optimized and Industrial Multipliers. In *Design, Automation and Test in Europe*. 544–549.
[25] Lloris Ruiz, Castillo Morales, Parrilla Roure, and García Ríos. 2014. *Algebraic Circuits*. Springer.
[26] Shobha Vasudevan, Vinod Viswanath, Robert W. Sumners, and Jacob A. Abraham. 2007. Automatic Verification of Arithmetic Circuits in RTL Using Stepwise Refinement of Term Rewriting Systems. *IEEE Trans. on Comp.* 56, 10 (2007), 1401–1414.
[27] Cunxi Yu, Walter Brown, Duo Liu, Andre Rossi, and Maciej Ciesielski. 2016. Formal verification of arithmetic circuits by function extraction. *IEEE Transactions on Computer Aided Design of Circuits and Systems* 35, 12 (2016), 2131–2142.
[28] Cunxi Yu, Maciej Ciesielski, and Alan Mishchenko. 2017. Fast Algebraic Rewriting Based on And-Inverter Graphs. *IEEE Transactions on Computer Aided Design of Circuits and Systems* 37, 9 (2017), 1907–1911.

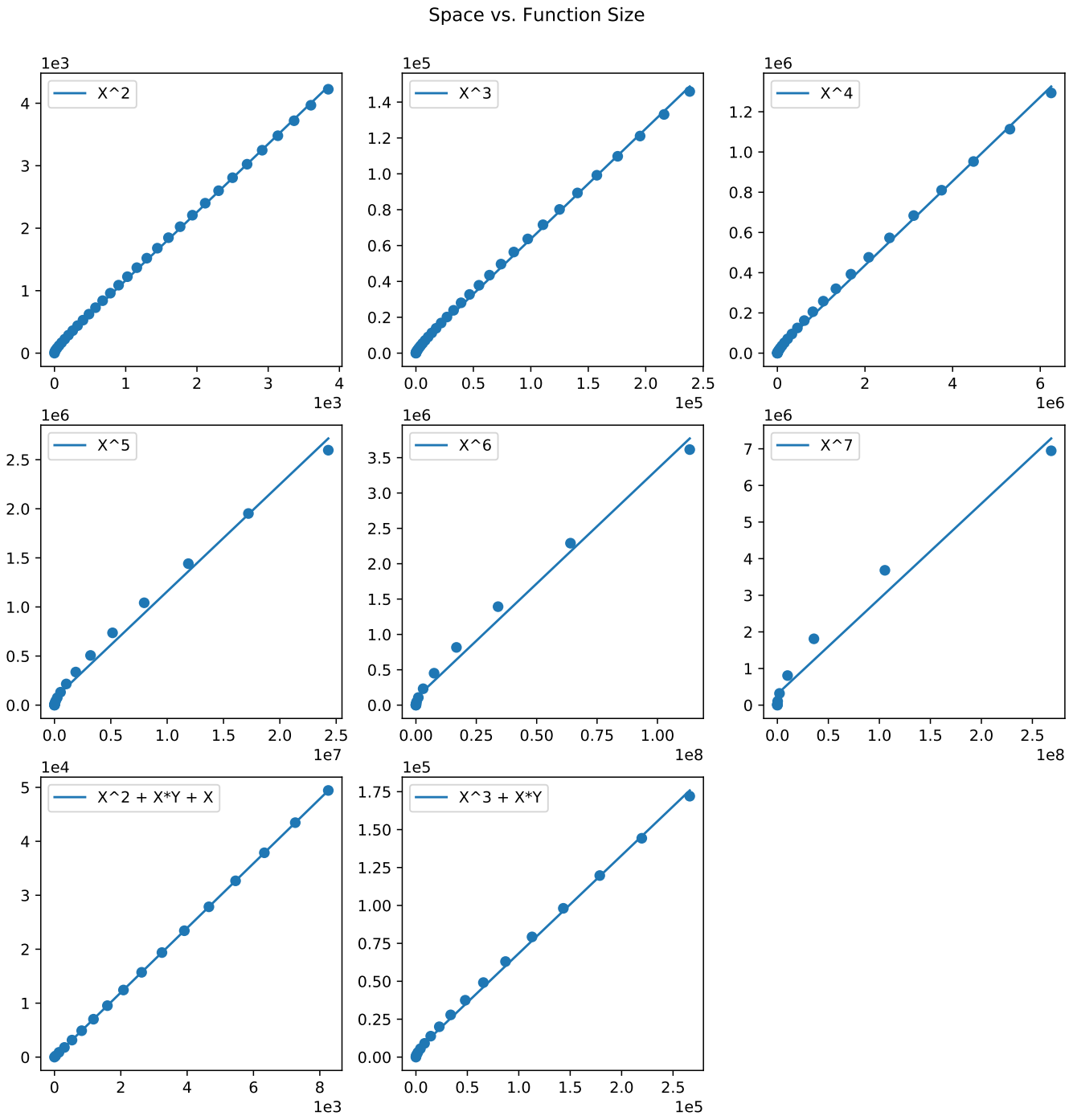

Figure 5: These graphs illustrate the space requirements for verifying a variety of circuits with different input bit lengths. The maximum space used in the verification process is plotted against the size of the circuit function. The points are fitted with linear polynomials (all R^2 values are larger than 0.99).

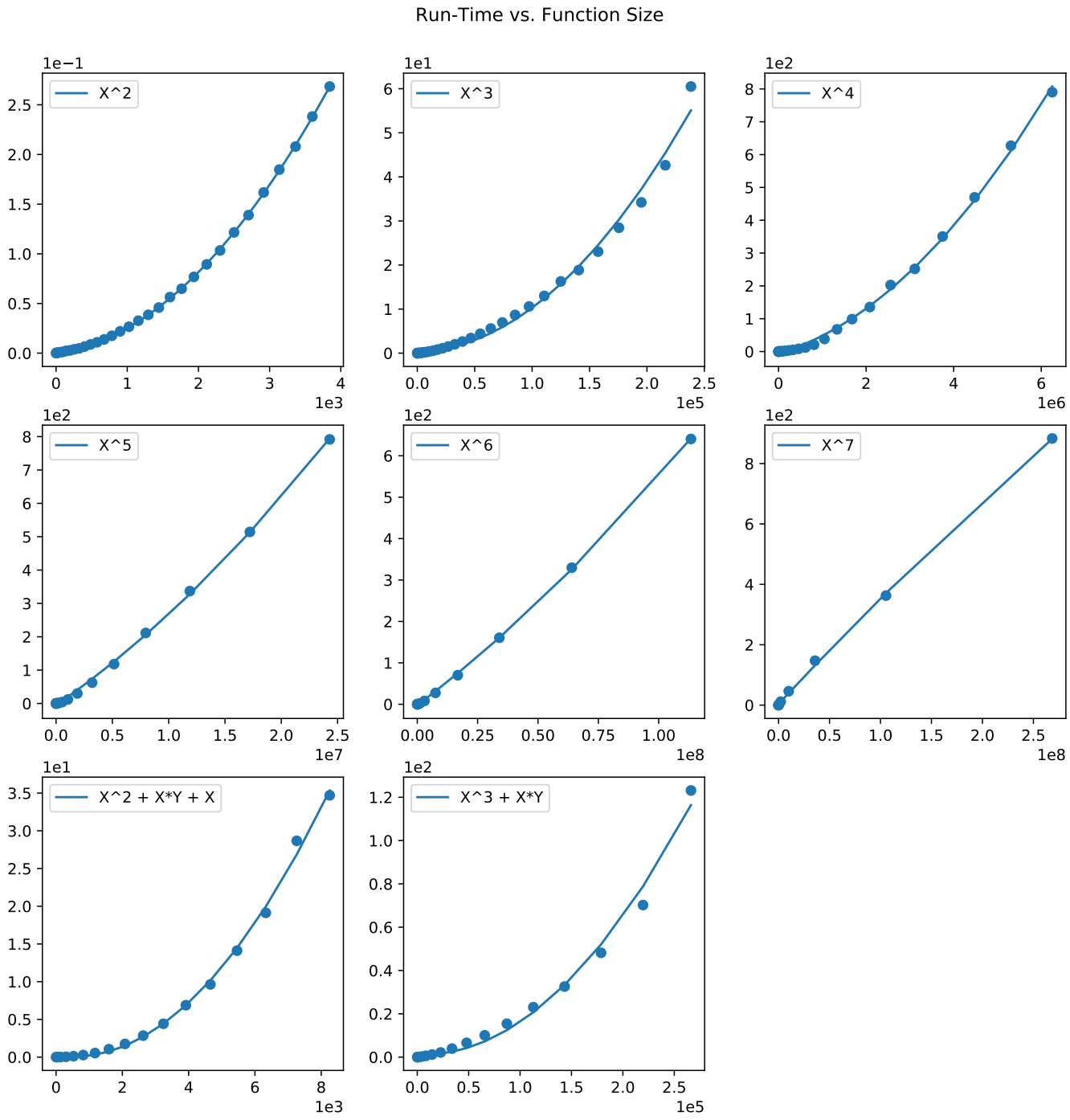

Figure 6: These graphs illustrate the time requirements for verifying a variety of circuits with different input bit lengths. The run-time in seconds used in the verification process is plotted against the size of the circuit function. The points are fitted with quadratic polynomials (all R^2 values are larger than 0.98).