Developing a Deterministic Polymorphic Circuit Generator Using Random Boolean Logic Expansion

Trinity Stroud
University of South Alabama

Developing a Deterministic Polymorphic Circuit Generator Using Random Boolean Logic Expansion

By

Trinity Stroud

A thesis submitted in partial fulfillment of the requirements of the Honors College at University of South Alabama and the Bachelor of Sciences in the Computer Science Department

University of South Alabama

Mobile

May 2021

Approved by:

[illegible]

Mentor: J. Todd McDonald

[illegible]

Committee Member: Todd R. Andel

[illegible]

Committee Member: Dimitrios Damopoulos

Kathy J. Cooke

Dean, Honors College

ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation under the Secure and Trusted Computing (SaTC) grant CNS-1811560 and 1811578. The project was a collaborative effort between the University of Nebraska at Omaha (UNO) and the University of South Alabama (USA).

ABSTRACT

Securing applications on untrusted platforms can involve protection against legitimate end-users who act in the role of malicious reverse engineers and hackers. Such adversaries have access to the full execution environment of programs, whether the program comes in the form of software or hardware. In this thesis, we consider the nature of obfuscating algorithms that perform iterative, stepwise transformation of programs into more complex forms that are intended to increase the complexity (time, resources) for malicious reverse engineers.

We consider simple Boolean logic programs as the domain of interest and examine a specific transformation technique known as Iterative Selection and Replacement (ISR), which represents a practical, syntactic approach for obfuscation. Specifically, we focus on improving the security of ISR by maximizing the flexibility and potential security of the replacement step of the algorithm which can be formulated in the following question: Given a selection of Boolean logic gates (i.e., a subcircuit), how can we produce a semantically equivalent (polymorphic) version of the subcircuit such that the distribution of potential replacements represents a random, uniform distribution from the set of all possible replacements?

This practical question is related to the theoretic study of indistinguishability obfuscation, where a transformer for a class of circuits guarantees that given any two semantically equivalent circuits from the class, the distribution of variants from their obfuscation are computationally indistinguishable. Ideally, polymorphic circuits that follow a random, uniform distribution provide stronger protection against malicious analyzers that target identification of distinct patterns as a basis for deobfuscation and simplification.

We introduce a novel approach for polymorphic circuit replacement called Random Boolean Logic Expansion (RBLE), which applies Boolean logic laws (of reduction) in reverse. We compare this approach against another proposed method of polymorphic replacement that relies on static circuit libraries. As a contribution, we show the strengths and weaknesses of each approach, examine initial results from empirical studies to estimate the uniformity of polymorphic distributions, and provide the argument for how such algorithms can be readily applied in software contexts. RBLE provides a unique method to generate polymorphic variants of arbitrary input, output, and gate size. We report initial findings for studying variants produced by this method and, from empirical evaluation, show that RBLE has promise for generating distributions of unique, uniform circuits when size is unconstrained, but for targeted size distributions, the approach requires adjustment for reaching potential circuit variant.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

IP	Intellectual Property
ISR	Iterative Selection and Replacement
LHS	Left-Hand Side
MPC	Multi-Party Computation
PET	Program Encryption Toolkit
RBLE	Random Boolean Logic Expansion
RHS	Right-Hand Side
RPM	Random Program Model
RSR	Random Selection and Replacement

LIST OF FIGURES

<u>LIST OF TABLES</u>

INTRODUCTION

Intellectual property (IP) is currently embedded in both software and hardware that are used in almost every area of society today. Companies can typically have billions of dollars invested in such IP. Theft of IP has therefore become a major concern for tech companies across the globe and, in particular, the United States. This research project concerns polymorphism and circuit protection for the purpose of approaching an efficient obfuscation algorithm which could serve as a defense against adversarial reverse engineering and, ultimately, IP theft.

This approach would involve transforming programs such that the amount of time and resources necessary for a malicious reverse engineer to recover IP in a circuit program becomes undesirable or, in the best case, complete infeasible [1]. A solution to this problem of reverse engineering could be applied in the areas of circuit and software protection to the effect that IP of any manner could be safeguarded to a degree against attacks meant to discern the function and form of said programs [2].

One particular type of obfuscating transformation, known as a selection/replacement algorithm [3], involves taking small parts of a circuit and replacing that small part with a functionally equivalent version, or variant, with some different structure. This process is repeated over and over again (iteratively) until some desired level of overhead or security is reached, and the program is deemed to be obfuscated.

In order to reach this solution, the specific question we look to answer is: Can we create a deterministic circuit generation algorithm that approaches a uniform random selection from the set of all circuits that implement a specific function? Such an algorithm could greatly improve the efficiency and capacity of selection/replacement algorithms over, for example, other approaches that involve the enumeration and selection of circuits from static libraries that must be stored on disk or randomly generated [4].

In order to answer this question, we plan to implement a Random Boolean Logic Expansion (RBLE) algorithm as a part of the existing Program Encryption Toolkit (PET) software. As combinational circuits are, in essence, no different from Boolean expressions [5], we will be using this algorithm while introducing random choices so that we can create polymorphic circuit variants from their manipulated Boolean expressions as part of a selection/replacement algorithm [6], thereby reducing the quantity of resources required to perform operations and introducing interesting potential avenues for program protection.

After a component of a circuit's Boolean expression is selected in the process of obfuscation, this RBLE algorithm would be employed in order to randomly determine from a list of Boolean logic laws one to apply to the subexpression that would alter its form while preserving its function. The resulting subexpression would take the place of the original selected from the circuit's expression so that a portion of the circuit has been obfuscated with no change made to its overall function. Each output function of the circuit would be represented with separate Boolean expressions, though the functions themselves may be closely connected [7] and this method applied to each. This process would be repeated, with different portions of the expression being transformed randomly, until such a time as the circuit's expression is considered to be appropriately obfuscated.

With a similar intent as McDonald, Kim, and Koranek's research to compare the efficiencies of obfuscation algorithms [3], to ensure that this RBLE algorithm truly approaches a uniform random selection from the set of all circuits that implement a specific function, we will create distributions of the functions of the circuit families produced by each of the following methods: the random and iterative application of Boolean logic laws to a circuit's expression to produce a variant, the enumeration and random selection of a circuit from a circuit family, and the generation of a circuit until a functionally equivalent variant is produced. Comparing these distributions will allow us to verify that our RBLE algorithm is truly as random as these other methods of random selection/replacement and random generation.

MOTIVATING CONTEXT

We provide two motivating scenarios that give context to this work: one primarily hardware-based and one primarily software-based. Nohl et al. [8] were some of the first researchers to illustrate the relative ease of physically reverse engineering hardware implementations of integrated circuit boards to recover gate-level programs (also known as netlists). In their work, they analyzed the Mifare Classic RFID cipher that was used as part of a public transit system card. Once these gate level constructs were recovered, design-level components could be reverse engineered to reveal the implementation of the cipher itself which led to the discovery of numerous flaws in the cryptographic design. Obfuscation of gate-level netlists programs through algorithms such as ISR would offer one potential countermeasure to design-level recovery of components: this specifically involves preventing recovery of the number, type, and inter-connectivity of building block components used to create more complex circuits. ISR has been utilized in prior work to evaluate the security of component recovery algorithms [9, 10, 11] and thus RBLE would provide new directions for such research.

Although our focus is primarily on circuit-to-circuit transformation, ISR with RBLE-based replacement has implications for software-to-software transformation. First, we note two different domains where software is converted into hardware representation. In the area of secure multi-party computation (MPC) [12], the predominant question of interest is how to securely compute a joint function on private inputs from distrusting participants while revealing nothing more than the result of the function. MPC schemes beginning with the seminal work of Yao on garbled circuits [13] have traditionally taken the joint function of interest and represented it in a standard circuit form consisting of AND, OR, and NOT gates. The cryptographic aspect of MPC protocols involves garbling the circuit in such a way that its evaluation by two or more parties results in the privacy of inputs as well as intermediate computations. MPC research has seen a rebirth of interest in recent years as well as a multitude of practical implementations that support translation of functions into circuits in Boolean, arithmetic, or formula form. Fairplay [14] was one of the first implementations of a two-party protocol: it comprised a high-level procedural language to support the expression of secure functions and the translation to one-pass Boolean circuits. Since then, multiple implementations of software-to-circuit compilers for MPC construction based on C and C++ have appeared such as OblivC [15, 16], ABY [17], EMP [18], and PICCO [19].

In the domain of systems design and synthesis, the distinction between hardware and software has become less clear for some time. The advent of field programmable gate arrays (FPGAs) has moved the defining aspect of hardware programs into a more fluid form, where reprogrammable hardware is becoming the norm. Almost 20 years ago, Wirth [20] pointed out the ease by which traditional software constructs (sequence and choice) are easily translatable to combinational logic while looping constructs can be handled with sequential logic forms. He was one of the first researchers to argue for a common language to express both software and hardware constructs. Since then, several realizations of this concept have made their way into commercial synthesis tools and systems design thought. SystemC [21] is probably the more well-known and earliest examples of this hardware/software marriage and is now an IEEE standard. Such programming environments support hardware description languages such as Verilog or VHDL and their translation to C++ [22].

The growing use of software-to-hardware programming environments for both MPC and systems design provides context for how circuit transformation algorithms might eventually be used for software protection against MATE attacks. In particular, software-based hardware abstractions pose an ideal method to frustrate traditional software analysis and reverse engineering techniques [23]. RBLE thus also has possibility to translate directly into software protection schemes based on hardware abstraction in the future, though this is not the focus of this paper.

The primary motivation for our research was whether an efficient (deterministic) circuit generation algorithm could be used to create polymorphic circuit variants. Ideally, this algorithm should produce distributions that approach a random, uniform selection from the set of all possible choices of semantically equivalent circuits. We compare our approach against a known static enumeration approach that, despite its ability to generalize to arbitrary circuit selections, does generate random, uniform distributions. To understand the generational approaches, we provide a review of basic definitions and concepts related to RBLE in the following sections.

BOOLEAN LOGIC

The definition used by Svensson to describe a Boolean algebra [24] is as follows:

"By a Boolean algebra we mean a lattice that is both distributive and complementary. We note a couple of immediate consequences of the definition.

- *Any Boolean algebra is a bounded lattice. This is because a Boolean algebra is complementary, and by definition a complementary lattice has to be bounded,*
- *The cancellation law holds in a Boolean algebra, since a Boolean algebra is a distributive lattice,*
- *Each element in a Boolean algebra has exactly one complement."*

The corresponding notation for a general Boolean algebra is then $(B,\ \wedge,\ \vee,\ ',\ 0_B,\ 1_B)$, where B is a set; $\wedge$ and $\vee$ are binary operators for intersection and union, respectively; $'$ is the unary operator for complementation; and 0_B and 1_B are the two possible values for any element of B. This is a more simplified form of the way in which we choose to represent Boolean expressions, or functions. As a combinational circuit is, in essence, equivalently representable as a Boolean expression by its function [5], we have chosen certain symbols and rules for describing this notation.

BOOLEAN LOGIC FUNCTIONS & LAWS

A Boolean function is a function with a domain of values $\{0, 1\}$ and of a finite number of variables of value $\{0, 1\}$. Theorem 1 [25] lists some of the properties by which a Boolean function is defined on the set $B = \{0, 1\}$ with binary operations of conjunction ($\vee$, OR, $+$) and disjunction

($\wedge$, AND, $*$), and a unary operation of negation (x, NOT, x'). These are the basis for the application of Boolean algebra laws.

Theorem 1. For all $x, y, z \epsilon B$, the following identities hold:

(1) Annulment: $x \vee 1 = 1$ *and* $x \wedge 0 = 0$*;*

(2) Identity: $x \vee 0 = x$ *and* $x \wedge 1 = x$*;*

(3) Commutativity: $x \vee y = y \vee x$ *and* $x \wedge y = y \wedge x$*;*

(4) Associativity: $(x \vee y) \vee z = x \vee (y \vee z)$ *and* $x \wedge (y \wedge z) = (x \wedge y) \wedge z$*;*

(5) Idempotence: $x \vee x = x$ *and* $x \wedge x = x$*;*

(6) Absorption: $x \vee (x \wedge y) = x$ *and* $x \wedge (x \vee y) = x$*;*

(7) Distributivity: $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$ *and* $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$*;*

(8) Complementation: $x \vee x = 1$ *and* $x \wedge x = 0$*;*

(9) Involution: $x = x$*;*

(10) De Morgan's Laws: $(x \vee y) = x \wedge y$ *and* $(x \wedge y) = x \vee y$*;*

The following table (Table 1) lists the combinational circuit gate types representable in Boolean expressions along with their symbols and truth tables, which describe for every combination of possible values the resulting value of a circuit gate:

Table 1: Logic Gate Types and Truth Tables

Name	Notation	Truth Table		
NOT	A'	A		A'
		0		1
		1		0
AND	$A * B$	A	B	$A * B$
		0	0	0
		0	1	0
		1	0	0
		1	1	1
NAND	$(A * B)'$	A	B	$(A * B)'$
		0	0	1
		0	1	1
		1	0	1
		1	1	0
OR	$A + B$	A	B	$A + B$
		0	0	0
		0	1	1
		1	0	1
		1	1	1
NOR	$(A + B)'$	A	B	$(A + B)'$
		0	0	1
		0	1	0
		1	0	0
		1	1	0
XOR	$A \wedge B$	A	B	$A \wedge B$
		0	0	0
		0	1	1
		1	0	1
		1	1	0
NXOR	$(A \wedge B)'$	A	B	$(A \wedge B)'$
		0	0	1
		0	1	0
		1	0	0
		1	1	1

BOOLEAN LOGIC EXPRESSIONS

A typical way to represent Boolean functions are with Boolean expressions, which are logical statements that, upon evaluation, have a value of either 0 or 1, false or true. For notation purposes, we express the three primary operators within Boolean expression as follows:

disjunction (∨, OR) with $+$; conjunction (∧, AND) with $*$; and negation, (x, NOT) with x'. There is an additional operator, XOR, represented in our notation as the traditional programmatic binary XOR (^), which is a derived operation based on disjunction, conjunction, and negation rules as follows:

$$x \text{ XOR } y = (x \wedge y) \vee (x \wedge y)$$

with our notation for the above expression being:

$$x \hat{} y = (x * y') + (x' * y).$$

Table 2 illustrates Boolean expressions derived from Boolean algebra laws where the left-hand side (LHS) is equivalent functionally to the right-hand side (RHS). These algebraic laws are normally applied repeatedly to reduce Boolean expressions to their simplest forms [26].

The following is an example of Boolean expression from the C2-1-2 circuit family, which consists of those circuits with two inputs (C**2**-1-2), one output (C2-**1**-2), and two gates (C2-1-**2**):

$$g1 \;=\; ((i1 \,\hat{}\, i0) \;+\; i0)'. \tag{1}$$

Here, $g1$ represents the value of the single output belonging to the circuit, and $i0$ and $i1$ represent the values of the two inputs to the circuit. Using the notations detailed in Table 1, we see that this expression describes a circuit consisting of a NOR gate of input $i0$ and the XOR of inputs $i0$ and $i1$. We can perform on this expression an operation known as reduction, which simplifies the form of a Boolean expression while preserving its overall function. This can be done by applying to it the Boolean logic laws included in Table 2.

Table 2: Boolean Expression Reductions

#	Original		Reduction	Law
1	A * A	=	A	Idempotence
2	A + A	=	A	Idempotence
3	A * B	=	B * A	Commutativity
4	A + B	=	B + A	Commutativity
5	A * (B * C)	=	(A * B) * C	Associativity
6	A + (B + C)	=	(A + B) + C	Associativity
7	A * (A + B)	=	A	Absorption
8	A + (A * B)	=	A	Absorption
9	A * (B + C)	=	(A * B) + (A * C)	Distributivity
10	A + (B * C)	=	(A + B) * (A + C)	Distributivity
11	A * 0	=	0	Annihilation
12	A + 0	=	A	Identity
13	A ^ 0	=	A	Identity
14	A * 1	=	A	Identity
15	A + 1	=	1	Annihilation
16	A ^ 1	=	A'	Negation
17	A * A'	=	0	Complementation
18	A + A'	=	1	Complementation
19	(A')'	=	A	Involution
20	(A + B)'	=	A' * B'	De Morgan's
21	(A * B)'	=	A' + B'	De Morgan's
22	(A + B) * (A' + B')	=	A ^ B	Derivation
23	(A' * B) + (A * B')	=	A ^ B	Derivation
24	(A + B)' + (A * B)	=	(A ^ B)'	Negation
25	A ^ A	=	0	Annihilation
26	(A ^ A)'	=	1	Annihilation
27	(A * B') + (A * B)	=	A	Annihilation
28	(A' * B') + (A' * B)	=	A'	Negation
29	(A + B) * (A + B')	=	A	Annihilation
30	(A' + B) * (A' + B')	=	A'	Negation

For each Boolean logic law detailed above, the LHS of the expression is functionally equivalent to the RHS of the expression. This fact allows us to substitute a portion of an expression that matches the LHS or RHS of a logic law with its corresponding RHS or LHS, respectively. To demonstrate this concept, the following steps detail the reduction of the Boolean expression (1) belonging to the circuit family C2-1-2 to the expression representing the C2-1-1 circuit of the same function:

$$g1 = ((i1 \wedge i0) + i0)'$$

Step 1: $(((i1' * i0) + (i1 * i0')) + i0)'$ (applied law #23)

Step 2: $((i0 + (i1' * i0)) + (i1 * i0'))'$ (applied law #6)

Step 3: $(i0 + (i1 * i0'))'$ (applied law #8)

Step 4: $((i0 + i1) * (i0 + i0'))'$ (applied law #10)

Step 5: $((i0 + i1) * 1)'$ (applied law #18)

Step 6: $(i0 + i1)'$ (applied law #14)

$$g1 = (i0 + i1)'$$

DIGITAL LOGIC CIRCUITS

Combinational circuits directly implement Boolean logic via a set of logic gates (called the basis set Ω) such as AND, OR, XOR, NOT, NAND, NOR, and NXOR. Structurally, they can be expressed in a number of ways including textually in netlist languages such as BENCH format [27] and visually in schematic form. Figure 1 illustrates a combinational circuit belonging to the family C5-2-6 in schematic form with corresponding BENCH netlist. Behaviorally, an n-input, m-output circuit combinational circuit can be seen as an array of Boolean functions $fi : Bn \rightarrow \{0, 1\}$, where $i = 1..m$ [26].

A Boolean expression can directly represent combinational logic netlists by assigning each circuit output a function, assigning circuit inputs as Boolean variables in the expression, and directly translating each circuit gate to its corresponding logic expression. Thus, combinational circuits are equivalently represented structurally as a Boolean expression [26, 28]. Figure 1 illustrates the corresponding Boolean expression for the circuit structure.

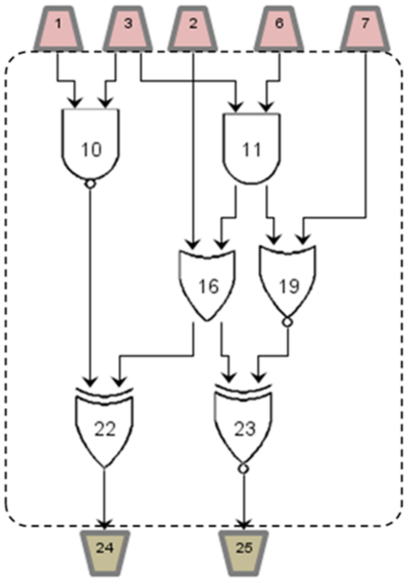
BENCH Netlist

```
INPUT(1)
INPUT(2)
INPUT(3)
INPUT(6)
INPUT(7)

OUTPUT(22)
OUTPUT(23)

10 = NAND(1, 3)
11 = AND(3, 6)
16 = OR(2, 11)
19 = NOR(11, 7)
22 = XOR(10, 16)
23 = NXOR(16, 19)
```

Circuit Schematic



Boolean Expression

```
o24 = ((i1 * i3)' ^ (i2 + (i3 * i6)))
o25 = ((i2 + (i3 * i6)) ^ ((i3 * i6) + i7)')'
```

Figure 1: Equivalent Circuit Representations

Logic circuits are typically grouped in families based on their input, output, and gate sizes. We use the notation δX-Y to define the set of all circuits the same input size X and output size Y. We use the notation δX-Y-S to represent families of circuits with gate size S. We assume circuits that are within a family are derived from a common basis set Ω, where typical basis sets may include Ω = {AND, OR, NOT}, Ω = {NAND}, Ω = {NOR}, or Ω = {AND, OR, XOR, NAND, NOR, NXOR}. The fan-in of a gate is the number of unique inputs fed to the gate. Legal circuits within a family are also governed by rules related to their structure:

(1) Symmetry: Should we consider a gate with inputs (X1, X2) as equivalent to a gate with inputs (X2, X1)?

(2) Redundant Gates: Should we allow gates that are identical to other gates (same fan-in and same gate type)?

(3) Constant Signals: Should we allow the circuit immediate access to the constants 0 or 1?

(4) Degeneracy: Should we allow both inputs to a gate to originate from the same source gate?

(5) Fan-in: Should we allow gates with multiple fan-in versus simple binary (2 fan-in) gates?

(6) Basis: What set of gates Ω can constitute the circuit structure?

(7) Size: Does the set contain all circuits up to a certain gate size bound or only circuits with an exact gate size?

(8) Outputs: For multiple output circuits, which gates should be allowed as outputs?

Given answers to these constraints, different circuit families can be produced, with more relaxed constraints producing larger numbers of circuits in the same identical family. As an example, families of legal circuits that minimize redundancy, disallow degenerate conditions and constant signals, use exact size, and allow only binary gates are typical for standard building blocks in larger combinational circuits. Figure 1 illustrates the legal family of $\delta 2-1-1$ circuits given basis Ω = {AND, OR, XOR, NAND, NOR, NXOR}, which consists of only the six basic logic gates themselves.

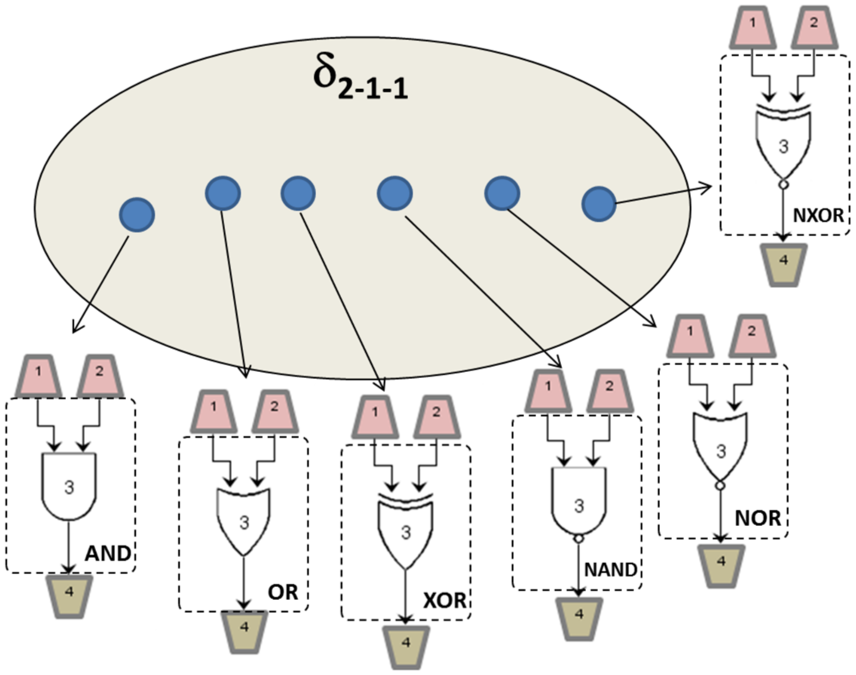


Figure 2: δ2−1−1 Circuit Family

RANDOM SELECTION & REPLACEMENT

An algorithm that could be applied to a Boolean expression such that polymorphic circuit variants can be produced could be a useful application of the theory towards hindering the efforts of adversarial attacks against IP. McDonald and Kim [1] perform such research to examine the effects of random and deterministic techniques of obfuscation on logic-level definitions.

The Random Selection and Replacement (RSR) algorithm they describe operates by selecting subcircuits of an original circuit and replacing those subcircuits with functionally equivalent variants chosen either from static libraries or produced through the repeated application of Boolean logic laws to the Boolean expression of a subcircuit. They found that there are tradeoffs to using either an obfuscation algorithm that implements random choice or one that targets specific hiding properties of a circuit.

If we can consider incorporating an algorithm to manipulate Boolean expressions as part of an existing RSR algorithm, potentially our implementation would lessen the number of resources required to perform operations while also demonstrating uniform random selection for a distribution of circuits. This algorithm would also hopefully circumvent the issue McDonald, Kim, and Koranek [4] noticed with typical RSR algorithms that required the enumeration of all possible variants for a selection: as the size of the selection increased, performing this task became intractable.

In addition to creating an algorithm that approaches a uniform random selection of circuits, it would be preferable that the variants produced by the algorithm also approach the effectiveness of the random selection of variants in terms of the information revealed by the intermediate gate transformations. The Random Program Model (RPM) used by McDonald, Kim, and Grimaila [29] to study this exact problem used white-box and black-box transformations for protecting the intent of a circuit such that there would be no correlation between the behavioral information leaked by the obfuscated circuit and the behavior of the original circuit and that there would be no more correlation between the structural topology of the obfuscated circuit and the original circuit than for any randomly selected circuit. Such an arrangement would be preferable, as an obfuscated variant that can be analyzed to derive the original circuit from leaked behavioral or structural information would make for poor protection against malicious reverse engineering.

BOOLEAN REDUCTION & EXPANSION

Decades of research have been devoted to finding efficient algorithms for reducing circuit logic functions to their smallest size, thereby minimizing power and layout space in realized physical circuits. All reductions can ultimately be related to the application of one or more laws as seen in Table 2. Random Boolean Logic Expansion (RBLE) works by applying these laws in reverse.

The list of Boolean logic laws used by the RBLE algorithm to perform expansion versus reduction can be reduced to include only those logic laws which change the structure of the circuit. For example, associativity or distributivity are laws which change the number of variables or values represented in part of the overall Boolean expression and result in polymorphic variation. Laws such as commutativity would not, and therefore we can remove laws #3 and #4 from Table 2, leaving us with an optimized and reordered list of Boolean logic laws seen in Table 3.

The laws have been rearranged such that their original expressions are ordered by lowest-to-highest form. This ordering allows us to easily recognize that, for example, the Boolean value 0 has three possible expansions, the Boolean value 1 has three possible expansions, and any Boolean variable A has ten possible expansions.

Table 3: Boolean Expression Expansions

#	Original		Expansion	Law	Relative Gates
1	0	=	A * 0	Annihilation	CONST0
2	0	=	A * A'	Complementation	CONST0
3	0	=	A ^ A	Annihilation	CONST0
4	1	=	A + 1	Annihilation	CONST1
5	1	=	A + A'	Complementation	CONST1
6	1	=	(A ^ A)'	Annihilation	CONST1
7	A	=	A * A	Idempotence	AND
8	A	=	A + A	Idempotence	OR
9	A	=	A * (A + B)	Absorption	AND,OR
10	A	=	A + (A * B)	Absorption	OR, AND
11	A	=	A + 0	Identity	OR, CONST0
12	A	=	A ^ 0	Identity	XOR, CONST0
13	A	=	A * 1	Identity	AND, CONST1
14	A	=	(A')'	Involution	NOT
15	A	=	(A * B') + (A * B)	Annihilation	AND,OR,NOT
16	A	=	(A + B) * (A + B')	Annihilation	AND,OR,NOT
17	A'	=	A ^ 1	Negation	XOR, CONST1
18	A'	=	(A' * B') + (A' * B)	Negation	AND,OR,NOT
19	A'	=	(A' + B) * (A' + B')	Negation	AND,OR,NOT
20	(A + B)'	=	A' * B'	De Morgan's	NOR
21	(A * B)'	=	A' + B'	De Morgan's	NAND
22	A ^ B	=	(A + B) * (A' + B')	Derivation	XOR
23	A ^ B	=	(A' * B) + (A * B')	Derivation	XOR
24	(A ^ B)'	=	(A + B)' + (A * B)	Negation	NXOR
25	A * (B + C)	=	(A * B) + (A * C)	Distributivity	AND,OR
26	A + (B * C)	=	(A + B) * (A + C)	Distributivity	OR,AND
27	(A * B) * C	=	A * (B * C)	Associativity	AND
28	(A + B) + C	=	A + (B + C)	Associativity	OR

To perform RBLE, we take a candidate circuit C and represent its circuit structure as a Boolean expression BE. The Boolean expression is then profiled to provide a potential set of logic expansions that may be applied, based on the presence of original expressions in BE seen in Table 2: 0, 1, A, A', $(A + B)'$, $(A * B)'$, $(A \wedge B)$, $(A \wedge B)'$, $A * (B + C)$, $A + (B * C)$, $(A * B) * C$, and $(A + B) + C$. In Table 3, the $(A + B)'$ expression in rule #20 represents a 2-input NOR gate, whereas $(A * B)'$ in rule #21 represents a 2-input NAND gate. In rules #22 and #23, $(A \wedge B)$ represents a 2-input XOR gate and $(A \wedge B)'$ represents a 2-input NXOR gate. In rules #17-19, A' represents the presence of a NOT gate that receives a signal from some part of the circuit netlist.

Thus, each original expression corresponds to a basic digital logic gate or input to a logic gate (some variable A) in the circuit netlist. For purposes of expansion, 0 and 1 represent constant 0 or 1 signals, which are kept in Boolean expression form until the circuit structure is realized in its final form. At that point, any 0 and 1 in the Boolean expression are replaced with a circuit netlist structure that generates the constant signal. So, for example, any 0 signal can be replaced with $(A \wedge A)$ or $(A * A')$, where A is any arbitrary variable that is already present in the expression. For multiple output circuits, each output is represented as its own Boolean logic expression.

Algorithm 1: Random Boolean Logic Expansion (RBLE)

```
input  : $C, P, n$
output : $C'$, where $\forall x : C(x) = C'(x)$
$BE \leftarrow convert(C)$; $done \leftarrow false$;
$fixed \leftarrow 0$; $attempts \leftarrow 0$; $numexp \leftarrow 0$;
while not done do
    $expansions \leftarrow profile(BE)$;
    $expansion \leftarrow select(expansions)$;
    $BE \leftarrow apply(BE, expansion)$;
    $\hat{C} \leftarrow realize(BE)$;
    if $P == FIXED$ then
        $fixed \leftarrow fixed + 1$;
        if $fixed = n$ then
            $C' \leftarrow \hat{C}$; $done \leftarrow true$ ;
        end
    else
        if ($P == STRICTSIZE$ and $size(\hat{C}) = n$) then
            $C' \leftarrow \hat{C}$; $done \leftarrow true$ ;
        else if ($P == TARGETSIZE$ and $size(\hat{C}) >= n$) then
            $C' \leftarrow \hat{C}$; $done \leftarrow true$ ;
        else
            $numexp \leftarrow numexp + 1$;
            if $numexp > MAXEXPANSIONS$ then
                $BE \leftarrow convert(C)$; $z \leftarrow 0$;
                $attempts \leftarrow attempts + 1$;
                if $attempts > MAXATTEMPTS$ then
                    $C' \leftarrow null$; $done \leftarrow true$;
                end
            end
        end
    end
end
return $C'$;
```

Algorithm 1 provides a summary of the RBLE approach. Given the profile of a Boolean expression BE and the set of its potential expansions, one is chosen pseudo-randomly and then applied to the expression. The new expression then becomes the input to the next round of expansions. Application of Boolean logic laws guarantees semantic equivalence of all intermediate Boolean expression forms. Each expansion thus produces a new Boolean expression, semantically equivalent to BE, based on the number of expansions that are applied, until some constraint is reached. We express constraints in the form of an input to the RBLE algorithm that we term expansion policy (P) with three possible values: $STRICTSIZE$, $TARGETSIZE$, and $FIXED$. The expansion policy value (n) is provided as input to the RBLE algorithm alongside the policy choice P. The condition for completion can be based on either the number of expansions performed or the size of the resulting polymorphic circuit.

Given a candidate circuit (C) with Boolean expression represented as (BE), policy choice (P), policy value (n), a number of maximum expansions ($MAXEXPANSIONS$), a number of maximum attempts ($MAXATTEMPTS$), RBLE will produce as output a polymorphic circuit variant C'. Expansion policies (P) and policy value (n) are defined as:

(1) $FIXED$: Apply a fixed number of expansions (n) to BE, which results in an ordered list of intermediate Boolean expression forms: $BE \rightarrow BE_1, BE_2, BE_3, \ldots, BE_n$. The final circuit C' is directly realized by gate level realization of the expression BE_n.

(2) $STRICTSIZE$: Apply expansions to BE until the corresponding gate size of C' is exactly equal to strict size n. This results in a potential sequence of intermediate Boolean expression forms: $BE \rightarrow BE_1, BE_2, BE_3, \ldots, BE_{\text{MAXEXPANSIONS}}$, where $MAXEXPANSIONS$ is some limit of expansions. Each intermediate Boolean expression

form BE_x is converted to its circuit netlist form C' and size of the circuit is computed. If the size$(C') = n$, the algorithm terminates and returns C'. If the limit $MAXEXPANSIONS$ is reached, the process is repeated with a fresh set of Boolean expansions starting with BE. The algorithm will terminate when a maximum number of attempts ($MAXATTEMPTS$) have been reached, which may result in failure to produce a polymorphic circuit C' with gatesize$(C') = n$.

(3) $TARGETSIZE$: Apply expansions to BE until the corresponding gate size of C' is greater than or equal to target size n. This results in a potential sequence of intermediate Boolean expression forms: $BE \rightarrow BE_1,\ BE_2,\ BE_3,\ \ldots,\ BE_{\mathrm{MAXEXPANSIONS}}$, where $MAXEXPANSIONS$ is some limit of expansions. Each intermediate Boolean expression form BE_x is converted to its circuit netlist form C' and size of the circuit is computed. If the size$(C') \geq n$, the algorithm terminates and returns C'. If the limit $MAXEXPANSIONS$ is reached, the process is repeated with a fresh set of Boolean expansions starting with BE. The algorithm will terminate when a maximum number of attempts ($MAXATTEMPTS$) have been reached, which may result in failure to produce a polymorphic circuit C' with size$(C') \geq n$.

Of the three policies, $STRICTSIZE$ and $TARGETSIZE$ are nondeterministic in the sense that they could fail to generate a polymorphic circuit variant with an exact or target gate size within pre-determined bounds, and thus they also have non-deterministic runtimes. However, the $FIXED$ expansion policy is deterministic and will always produce a variant in some predictable, linear amount of time.

In discussing Algorithm 1, an overview of the functions involved in computation is necessary to understand the execution of the program. The $profile$ function returns the set of all potential subexpressions within a Boolean expression which can have a Boolean expansion applied to it. The function $convert$ takes a circuit netlist and returns a Boolean expression consistent with the structure of the circuit. The function $realize$ takes a Boolean expression and returns a circuit netlist, where all constant 0 and 1 signals are converted into logic gate constructions. The function $apply$ takes as input a Boolean expression and a selected part of the expression that corresponds to a legal Boolean expansion rule then applies the expansion and returns a new Boolean expression. The function $select$ takes as input a set of legal Boolean expansions and returns a pseudo-random choice from the set.

Table 4 provides an example of applying a $FIXED$ policy on a Boolean expression where five expansions are applied to the expression $o1 = (i0 * i1)$. Figure 3 illustrates circuit realization of the corresponding Boolean expressions created through expansion in Table 4.

Table 4: Example Boolean Expansion Sequence

o1 = (i0 * i1)	**size(C)=1**
1: ((i0 + i0) * i1)	**rule 8, size(C')=2**
2: (((i0 + i0) + 0) * i1)	**rule 11, size(C')=6**
3: (((i0 + i0) + (i1 * 0)) * i1)	**rule 1, size(C')=7**
4: (((i0 + i0) + (i1 * (i0 ^ i0))) * i1)	**rule 3, size(C')=5**
5: (((i0 + i0) + (i1 * (i0 ^ i0))) * (i1 * i1))	**rule 7, size(C')=6**
o1 = (((i0 + i0) + (i1 * (i0 ^ i0))) * (i1 * i1))	**size(C')=6**

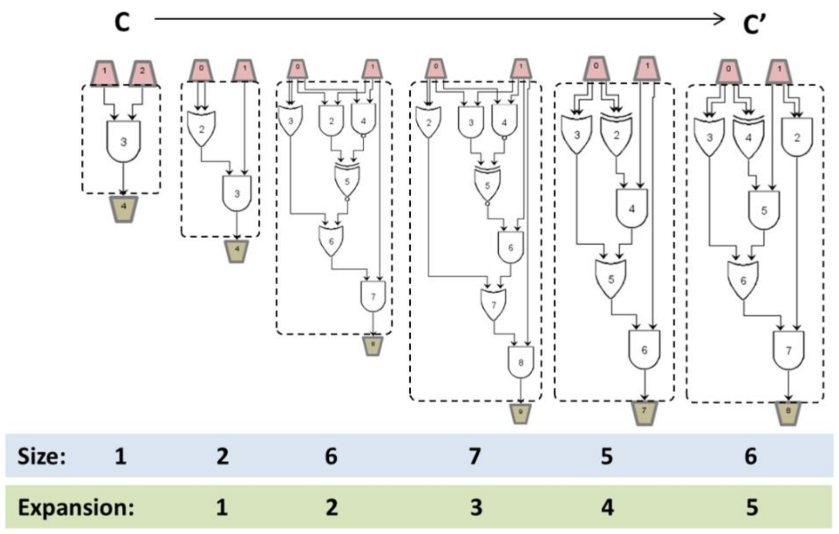


Figure 3: Example Expansion Circuit Realization

EXPERIMENTAL METHODS

To provide an initial evaluation of the efficacy of RBLE in comparison to pre-generated static libraries, we ran two types of experiments that generated distributions of replacement circuits using both approaches. The goal of the experiments was to understand the limits of RBLE in approaching a uniform distribution similar to what is possible with fully enumerating all possible circuit structures and storing them statically, thus being able to choose a replacement randomly from the set of all possible functionally equivalent polymorphic variants (referred to as CIRCLIB [11]).

We also wanted to evaluate the new possibility of creating polymorphic variants with sizes well beyond the current size limits of the static chosen-circuit approach. We exercised the algorithm on simple circuits to initially assess the characteristics of RBLE distributions. For this study, we only considered replacements for the six basic logic gates in the $\delta 2-1-1$ circuit family, which are seen in Figure 2.

Figure 4 provides a visual reference to our experimental framework, which we expand next. To generate circuit variants, we implemented RBLE in a Java-based code suite and utilized the open-source version of CIRCLIB created by McDonald et al. to generate static circuit libraries [10, 11]. All experiments were performed on an HP ZBook 17 G2 laptop with an Intel i7-4710MQ 2.50 GhZ CPU and 32GB installed RAM.

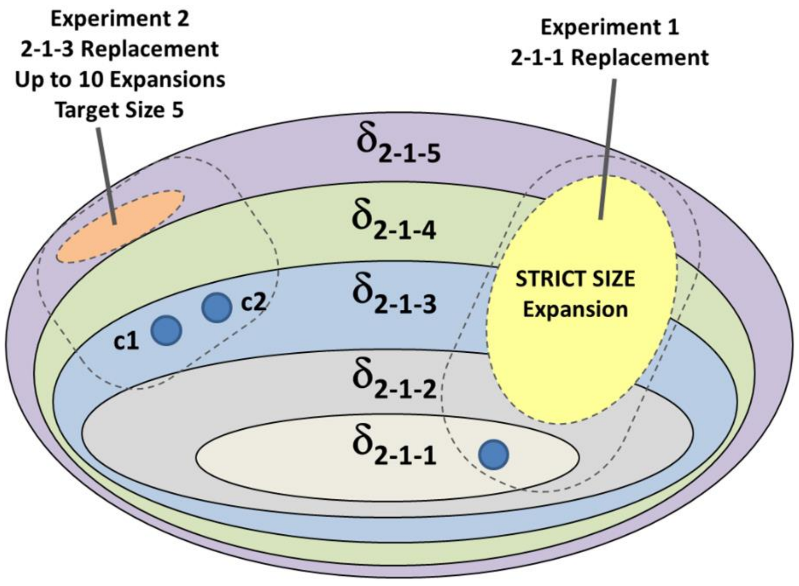

Experiment 2
2-1-3 Replacement
Up to 10 Expansions
Target Size 5
Experiment 1
2-1-1 Replacement
$\delta_{2\text{-}1\text{-}5}$
$\delta_{2\text{-}1\text{-}4}$
$\delta_{2\text{-}1\text{-}3}$
$\delta_{2\text{-}1\text{-}2}$
$\delta_{2\text{-}1\text{-}1}$
c1
c2
STRICT SIZE
Expansion

Figure 4: Empirical Evaluation Framework

EXPERIMENT 1: STRICT SIZE REPLACEMENT

We first evaluate RBLE under a $STRICTSIZE$ expansion policy, as this is the closest comparison to a chosen-circuit approach with CIRCLIB. For each of the six basic gate types in the $\delta 2{-}1{-}1$ family, we performed two sets of generations that created a total of 188,000 circuits:

(1) 1,000 variants from CIRCLIB and 1,000 variants from RBLE, totaling 6,000 circuits for each gate type and method, of target size 2, 3, 4, and 5. The only exception was that the $\delta 2{-}1{-}2$ family has no valid semantically equivalent XOR or NXOR circuits that have gate size 2, so only 4,000 circuits were created for this target size family. In total, 22,0000

circuits were generated for analysis. In results notation, we refer to this as the 1K distribution set.

(2) 10,000 variants from CIRCLIB and 10,000 variants from RBLE, totaling 60,000 circuits for each gate type and method, of target size 2, 3, and 4. The only exception was that the δ2−1−2 family has no valid semantically equivalent XOR or NXOR circuits that have gate size 2, so only 40,000 circuits were created for this target size family. In total, 160,0000 circuits were generated for analysis. In results notation, we refer to this as the 10K distribution set.

EXPERIMENT 2: FIXED EXPANSION/TARGETED REPLACEMENT

We evaluate RBLE under a $FIXED$ expansion policy, using six pairs of circuits chosen from the δ2−1−3 family, where each pair of δ2−1−3 circuits (C1, C2) are semantically equivalent to one of the basic gate circuits in the δ2−1−1 family (AND, OR, XOR, NAND, NOR, NXOR). For each circuit in each circuit pair (C1, C2), we create 100,000 variants chosen from CIRCLIB libraries with a target gate size of 5. For RBLE, we create 100,000 variants of each circuit with number of expansions n ranging from $n = 1..10$.

For CIRCLIB, each circuit in the pair (C1, C2) resulted in 100,000 variants of size 5, for a total of 200,000 per basic gate type, and 1,200,000 total circuits. For RBLE, each circuit in the pair (C1, C2) resulted in 1,000,000 variants given 10 possible expansion values, for a total of 2,000,000 per basic gate type, and 12,000,000 circuits total. As a result, a total of 13,200,000 circuits were generated for this experiment.

ANALYSIS

For analysis purposes, we refer to CIRCLIB variants as chosen replacement and RBLE variants as expanded replacements. We stored the results of the circuit distributions for each experiment type in BENCH netlist circuit files. Analysis was then performed on the BENCH files corresponding to each experiment type. We created a form of structural hash to uniquely identify the structure of each circuit netlist so that circuits with the same structure could be easily identified and grouped together. As part of the study, we learned that static CIRCLIB libraries contain structurally identical circuits that are semantically equivalent, even though CIRCLIB creates different netlist circuits for them in the static libraries. We explain the ramifications of this more in the Results section.

For Experiment 2, we also recorded sizes of the various circuits that were created based on different numbers of expansions being applied to the original circuit. We made special note of circuits that matched the target gate size (5) which the CIRCLIB algorithm used. As a result of using variable number of expansions with RBLE, an original circuit with 100,000 variants will only have some percentage that match target size 5, which we explore further in the Results section.

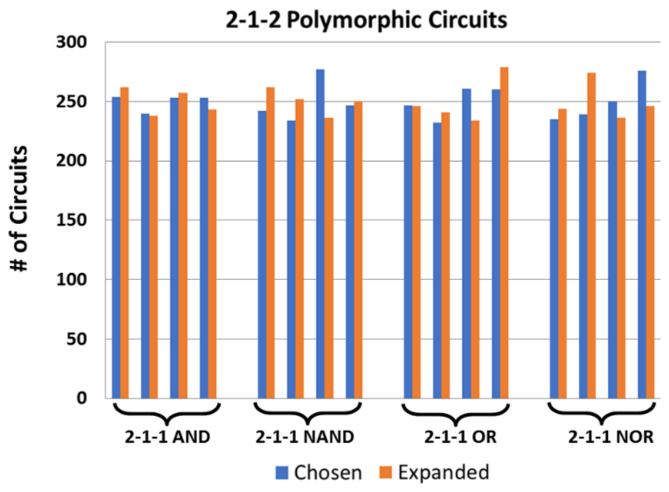


Figure 5: Number Circuits for δ2−1−2 Strict Replacement-1K

RESULTS & DISCUSSION

We report first the results of Experiment 1 distributions. Figure 5 shows the results of 1K distributions of the four circuit types which are possible for (gate size = 2) replacements of AND, OR, XOR, and NAND gates that are part of the (gate size = 1) δ2−1−1 family. Given standard circuit creation options for CIRCLIB, each original gate only has 4 possible variants in the δ2−1−2 family. The replacement circuits as seen in Figure 5 show that both RBLE and CIRCLIB create roughly equal distributions for all 4 circuits, for all 4 gate types.

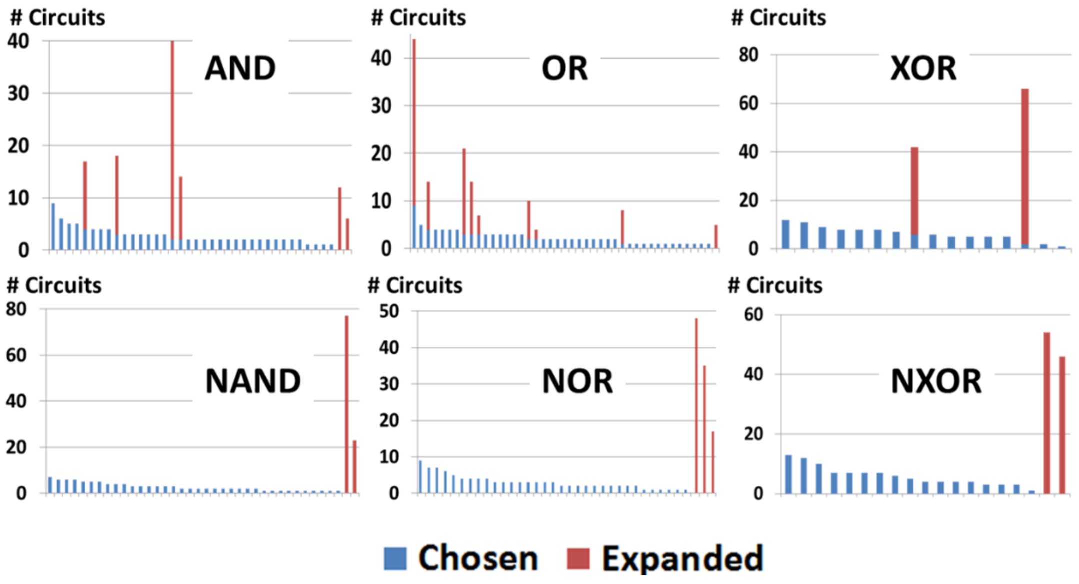


Figure 6: Number Circuits for δ2−1−3 Strict Replacement-1K

Figure 6 shows the results from Experiment 1 where 1,000 variants of size 3 were created for the original AND, OR, XOR, NAND, NOR, NXOR gates in δ2−1−1. The chart shows a combination of number of circuits for both methods, where circuits with the same structure are aligned. CIRCLIB variants follow a fairly uniform distribution for all 6 gate types, whereas RBLE replacements only represent a small number of the same circuits from the CIRCLIB potential set, with non-uniform distribution ranging from 3-10 circuits of size 3. The RBLE difference is due in part to the fact that only a small subset out of the 28 possible expansions may result in size 3 circuits.

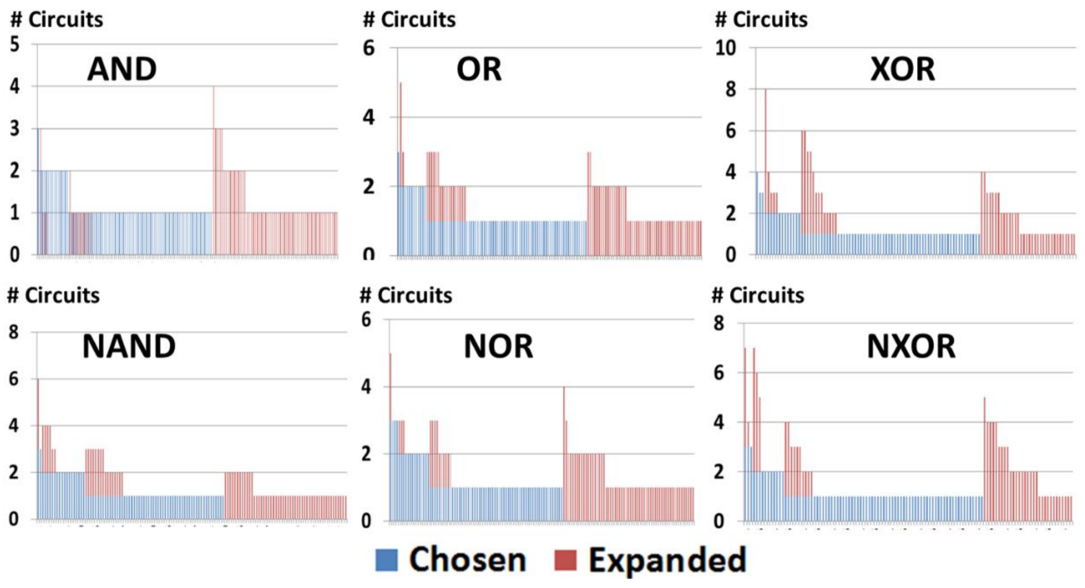


Figure 7: Number Circuits for δ2−1−4 Strict Replacement-1K

Figure 7 shows the results from Experiment 1 where 1,000 variants of size 4 were created for the original AND, OR, XOR, NAND, NOR, NXOR gates in δ2−1−1. The chart shows a combination of the number of circuits for both methods, where circuits with the same structure are aligned. CIRCLIB variants follow a fairly uniform distribution for all 6 gate types with 1-2 circuits being chosen from 70-80 possible variants. RBLE creates circuits that overlap between 5-10 of the same circuits that CIRCLIB produces (roughly 8% of the CIRCLIB sets). RBLE replacements of size 4 have a roughly uniform distribution ranging from 3-10 circuits from 50-60 possible variants. This size distribution reveals how RBLE construction can reach circuits that are not part of the CIRCLIB family because of creation rules: in particular, RBLE allows degenerate circuit conditions such as 2-input gates that have the same source. Most of the RBLE circuits are thus disjoint from the CIRCLIB variants.

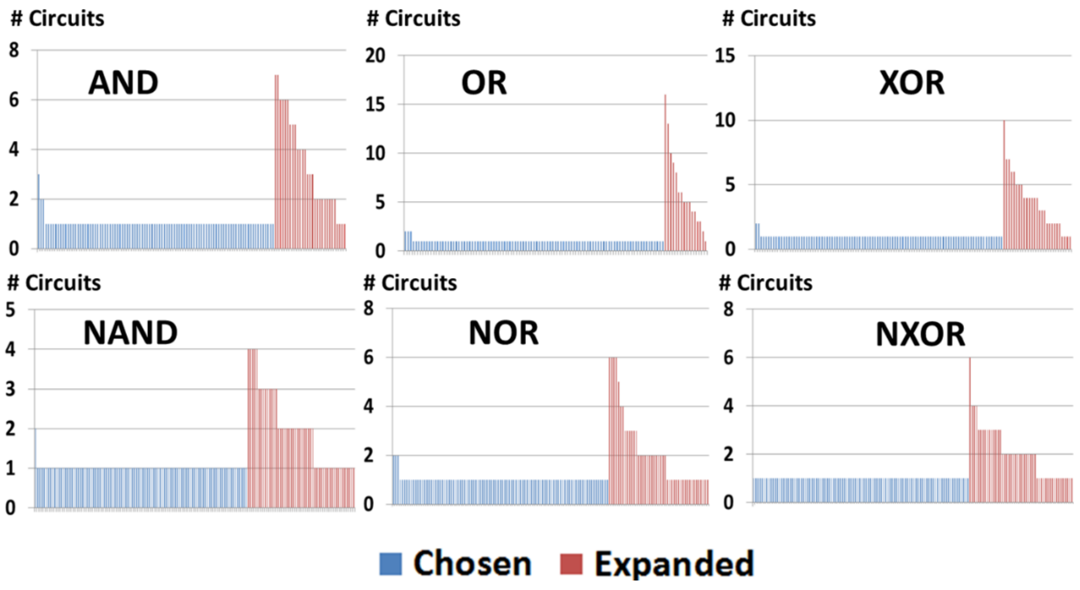


Figure 8: Number Circuits for δ2−1−5 Strict Replacement-1K

Figure 8 shows the results from Experiment 1 where 1,000 variants of size 5 were created for the original AND, OR, XOR, NAND, NOR, NXOR gates in δ2−1−1. The chart shows a combination of number of circuits for both methods, where circuits with the same structure are aligned. CIRCLIB variants again follow a fairly uniform distribution for all 6 gate types, whereas RBLE replacements have a similar distribution as with size 4 replacements. Given that only 1,000 variants were created for RBLE, the amount of variability is clearly less than what is possible with CIRCLIB variants, and certain circuit variants are created under RBLE with above average frequency. For size 5 replacements, the distributions show no overlap at all between the variants chosen by CIRCLIB and those expanded by RBLE.

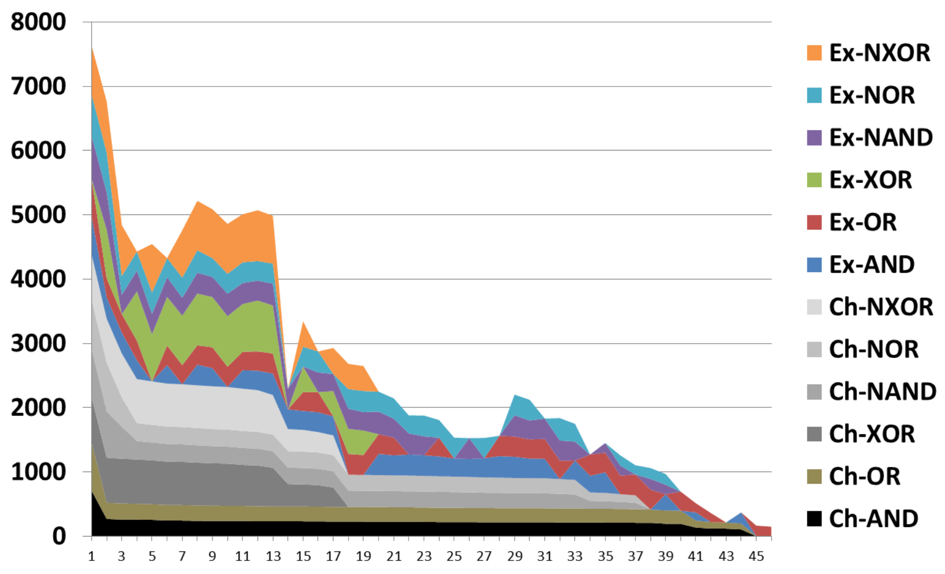


Figure 9: Number Circuits for δ2−1−3 Strict Replacement-10K

Figure 9 shows the results from Experiment 1 where 10,000 variants of size 3 were created for the original AND, OR, XOR, NAND, NOR, NXOR gates in δ2−1−1. The chart summarizes the number of circuits (each bar representing an identical matching circuit between the RBLE and CIRCLIB methods) for all gate types, ordered by the highest frequency that the variant is chosen by CIRCLIB. In a larger set of circuit replacements (10,000 attempts vs 1,000 attempts), it can be seen that CIRCLIB circuits do not follow a purely equal distribution. This is due to that fact that there are overlaps of structurally equivalent circuits in CIRCLIB, so that certain circuits have a higher probability of being chosen. We also observe that for size 3 replacements, allowing larger distributions (in this case 10,000 variants) shows that there are more overlaps with CIRCLIB variants that are chosen, depending on the gate type.

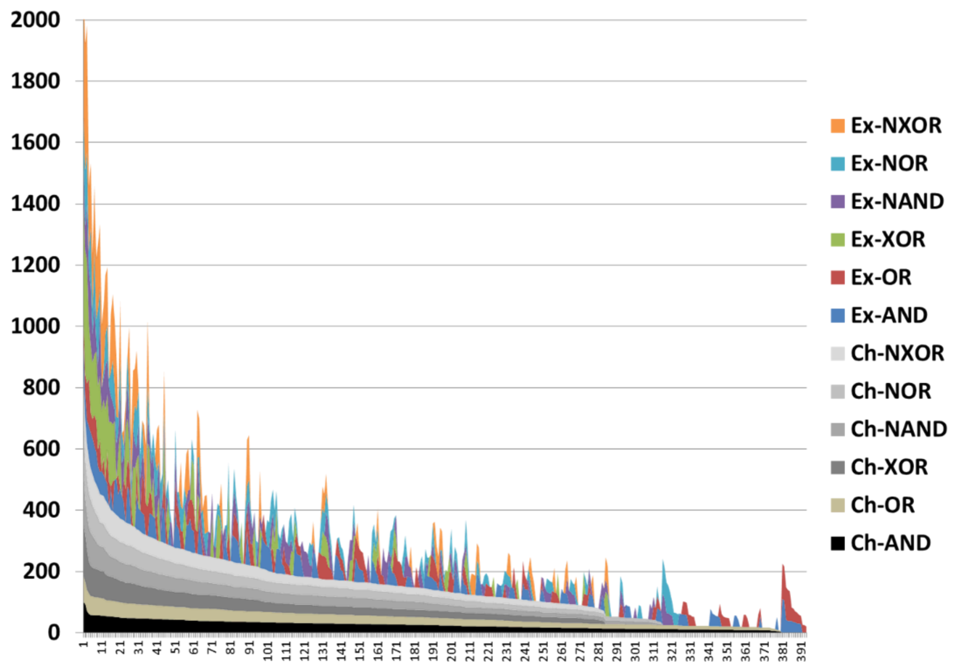


Figure 10: Cumulative Circuits δ2−1−4 Strict Replacement-10K

Figure 10 shows the results from Experiment 1 where 10,000 variants of size 4 were created for the original AND, OR, XOR, NAND, NOR, NXOR gates in δ2−1−1. The chart summarizes the number of circuits (each bar representing an identical matching circuit between the RBLE and CIRCLIB methods) for all gate types, ordered by the highest frequency that the variant is chosen by CIRCLIB. RBLE replacements do not follow the same distribution as CIRCLIB, but overall, both approaches select circuits from the same range of unique circuits for each gate type.

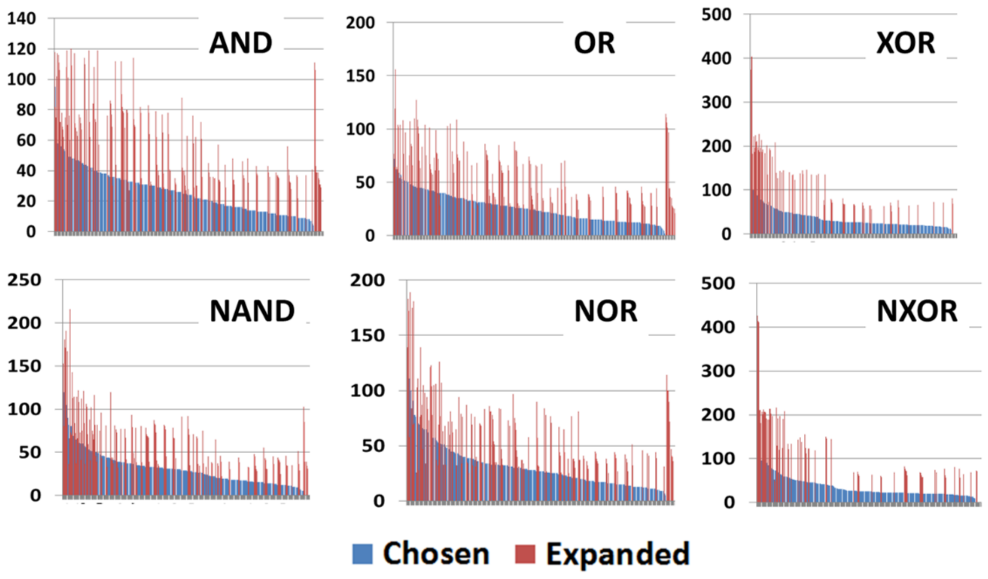


Figure 11: Number Circuits δ2−1−4 Strict Replacement - 10K

Figure 11 provides a more precise view of size 4 replacements created by the two methods for the 10K distribution set following strict expansion policy. In this view, the spiky nature of the RBLE replacements that are generated are compared against the same identical circuit that is chosen by the CIRCLIB algorithm. The chart is ordered based on highest frequency of CIRCLIB variants that were generated. As a strength, around 60% of unique CIRCLIB circuits are also created by expansion, but with RBLE producing a higher frequency of those variants in comparison. Strict size expansion policy, being non-deterministic, may result in failure to produce a variant: for these experiments, there were no maximum attempt failures.

EXPERIMENT 1 SUMMARY

To summarize analysis for Experiment 1, we observed that RBLE distributions are not completely uniform in comparison to CIRCLIB variants. Given replacements in the δ2−1−2 family, they are near identical. Beyond that, distributions vary considerably based on the number of variants generated (1K vs 10K). This is partially because the set of potential semantically equivalent replacements is above 1,000 for each of the original δ2−1−1 circuits for sizes 4 and 5. RBLE does generate a reasonable subset of potential CIRCLIB variants under strict expansion policy for all gate types, for target gate sizes 2 through 5. The spiky nature of the distributions and lack of ability to produce near matching distributions do point to weaknesses in the RBLE algorithm in regard to uniformity. We believe this is because only expansions were included in the RBLE algorithm (see Table 2). In order to reach a larger potential set of circuits, we believe that both reductions (see Table 1) and expansions should be used, as some circuits are not possible to create without both set of Boolean laws.

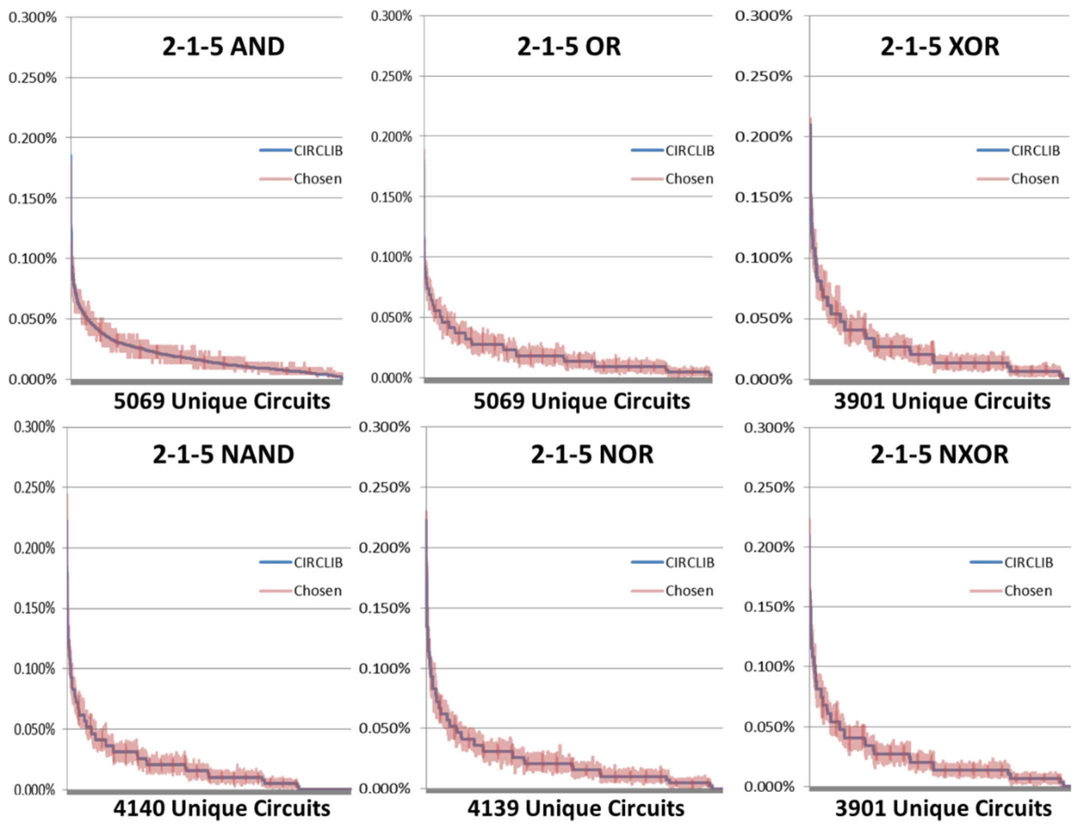


Figure 12: Distributions for δ2−1−5 Chosen Replacement

EXPERIMENT 2 SUMMARY

We report next the results of Experiment 2 distributions. We look first at the comparable set of circuits for all expansion possibilities that are created by RBLE and chosen with CIRCLIB that matched our target gate size of 5. Figure 12 and Figure 13 show the distribution results, per gate type, for replacements in the δ2−1−5 family. We can observe in Figure 12 the distribution of CIRCLIB replacements, where the number of actual CIRCLIB circuits of a given structure are compared against the number that are generated by the chooser algorithm. This shows that certain CIRCLIB circuits are over-represented, and thus the distribution among 200,000 variants of each gate type is not completely uniform.

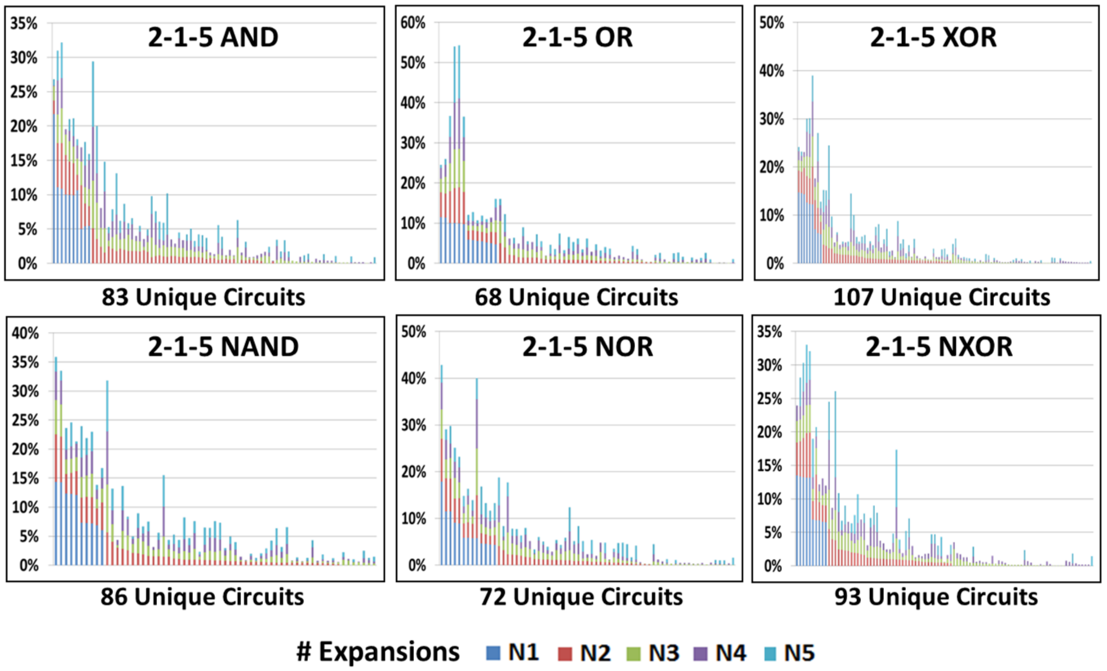


Figure 13: Distributions for δ2−1−5 Expanded Replacement

Figure 13 shows a summary of expanded circuits. We do not show expansions of 7 through 10 because they resulted in only a few or 0 circuits being produced that have gate size 5. This shows the closest comparison to CIRCLIB selection where the size is exact. The two figures also show the stark difference between potential unique circuits which can be reached by either approach. For each of the 200,000 variants chosen through CIRCLIB, all of the potential semantically equivalent versions for each gate type were reached: this includes 5069 AND variants, 4140 NAND variants, 5069 OR variants, 4139 NOR variants, 3901 XOR variants, and 3901 NXOR variants. Figure 14 provides a summary of the unique variants reached through expansion, where the gate size was 5. For example, with 3 expansions, RBLE produced 78 AND variants, 82 NAND variants, 64 OR variants, 71 NOR variants, 95 XOR variants, and 84 NXOR variants. The smallest number of unique variants was produced with 1 and 7 expansions.

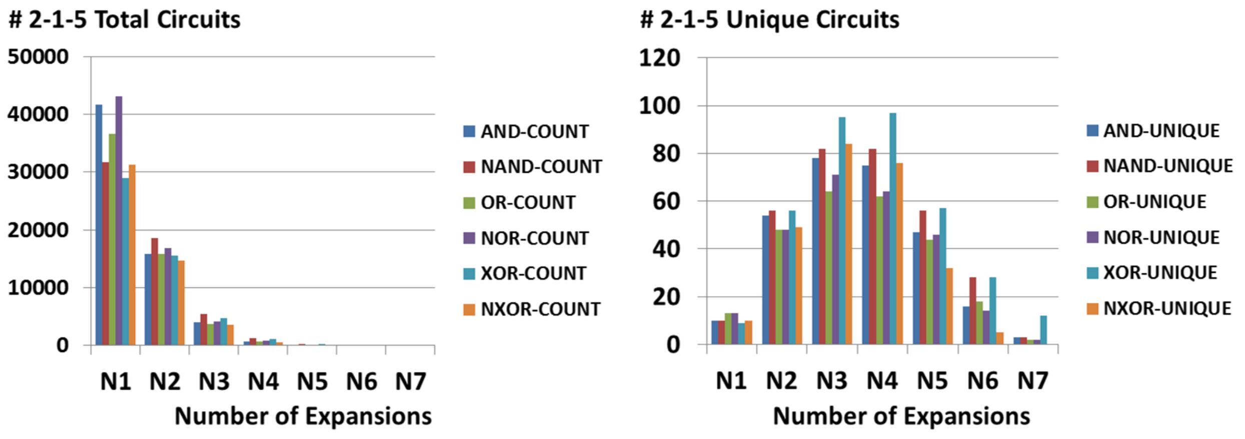


Figure 14: δ2−1−5 Expanded Replacement Summary

One of the primary benefits of RBLE is its ability to create variants of much larger size than what is feasible with the static CIRCLIB approach. Figure 15 illustrates the total distribution of circuits produced by RBLE for all gate types as part of Experiment 2, regardless of size. With 1 expansion, circuits between size 3 and 8 can be reached, whereas with 10 expansions, circuits between size 6 and 37 can be reached. The chart summarizes the distribution of the 1,200,000 circuits produced for 6 pairs of δ2−1−3 circuits with each pair being semantically equivalent to the original six basic gate types. This figure only shows unique circuits that are produced, ranging up to 153,303 for 5 expansions. Of the 12,000,000 circuits generated by RBLE, 8,253,348 circuits were unique, which speaks more to the uniform possibilities of RBLE when replacement size is not a limiting factor. The ability to reach larger circuit replacement possibilities opens up new potential for iterative sub-circuit selection and replacement as a result. A static approach, for example, would be limited by conventional disk file storage system constraints to libraries for δ2−1−X no greater than size 7 [11].

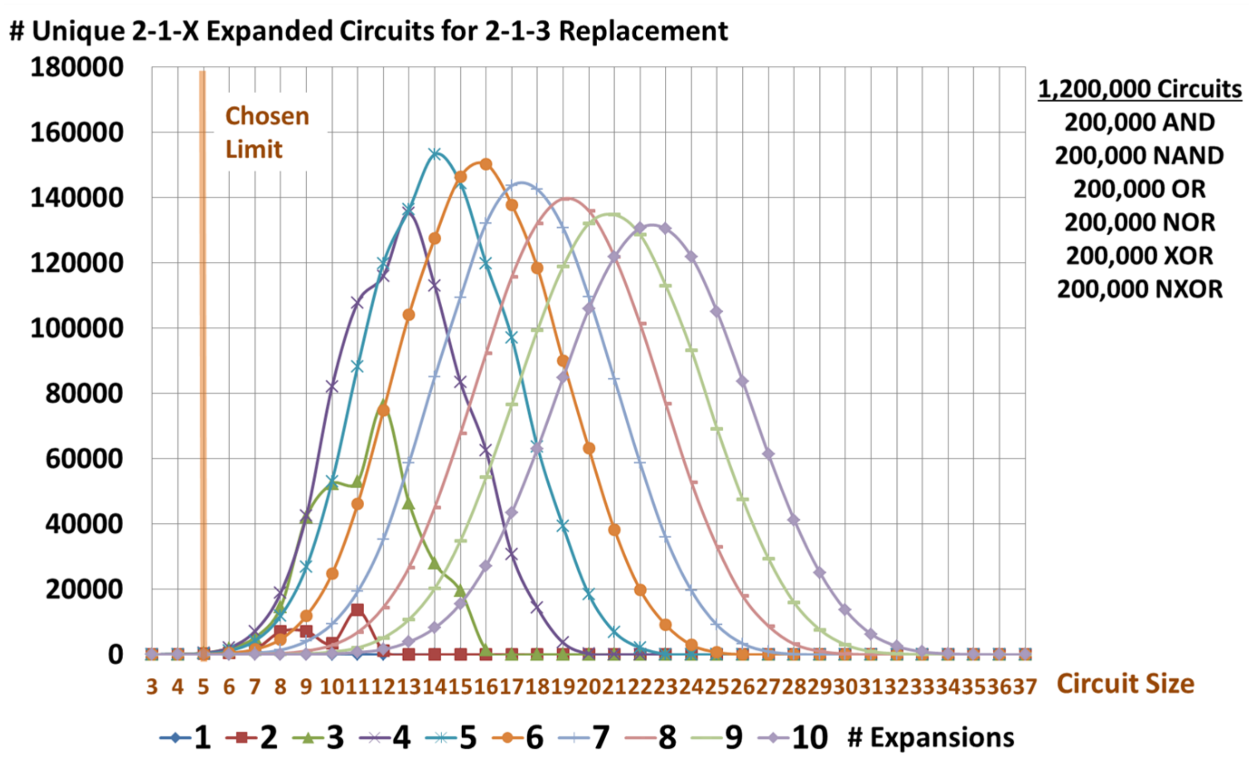


Figure 15: Unique Expanded Replacements (200,000 per type)

CONCLUSION

We introduced a novel method for generating polymorphic circuit variants based on inverse application of Boolean logic laws: Random Boolean Logic Expansion (RBLE). We generated and studied 13,360,000 circuit variants as semantically equivalent replacements for simple $\delta2{-}1{-}1$ and $\delta2{-}1{-}3$ circuits. Our initial empirical study shows that RBLE exhibited instances of uniformity when a specific sized circuit is required (strict size expansion policy) but can only reach a small percentage of comparable circuits from a static library selection when fixed expansions are used. However, when size is not a factor, RBLE can generate many unique variants uniformly when various expansion sizes are used.

Based on these initial results, future work should focus on addressing the inability of RBLE to reach certain circuits in a possible population of alternatives: we expect that the addition of reduction laws alongside expansion laws will address this problem. If we considered the presence of constant 0 and 1 signals as valid, this would also provide greater flexibility to reach circuits of certain sizes, as the signals are typically considered to be provided outside the circuit.

REFERENCES

[1] J. T. McDonald, Y. C. Kim, and D. Koranek, “Deterministic Circuit Variation for Anti-Tamper Applications,” *Proceedings of the Cyber Security and Information Intelligence Research Workshop (CSIIRW-2011)*, Oak Ridge, TN, Oct. 12-14, 2011.

[2] M. C. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering," *IEEE Design & Test of Computers*, vol. 16, no. 3, pp. 72-80, July-Sept. 1999, doi: 10.1109/54.785838.

[3] J. T. McDonald, Y. C. Kim, D. Koranek, and J. D. Parham, “Evaluating Component Hiding Techniques in Circuit Topologies,” *International Conference on Communications, Communication and Information Systems Security Symposium (ICC-CISS-2012)*, Ottawa, Canada, June 10-15, 2012.

[4] J. T. McDonald, Y. C. Kim, and M. R. Grimaila, “Protecting Reprogrammable Hardware with Polymorphic Circuit Variation,” *Proceedings of the 2nd Cyberspace Research Workshop*, Shreveport, LA, USA, June 2009.

[5] H. R. Anderson, “An Introduction to Binary Decision Diagrams,” *Efficient Algorithms and Programs*, Lecture, The IT University of Copenhagen, Copenhagen, Denmark, 1999.

[6] J. T. McDonald and Y. C. Kim, “Examining Tradeoffs for Hardware-Based Intellectual Property Protection,” *Proceedings of the 7th International Conference on Information Warfare (ICIW-2012)*, University of Washington, Seattle, USA, March 22-23, 2012.

[7] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677-691, Aug. 1986, doi: 10.1109/TC.1986.1676819.

[8] K. Nohl, D. Evans, S. Starbug, and H. Plötz. 2008. “Reverse-Engineering a Cryptographic RFID Tag.” *Proceedings of the 17th Conference on Security Symposium (SS’08)*. USENIX Association, Berkeley, CA, USA, 185–193. http://dl.acm.org/citation.cfm?id=1496711.1496724

[9] M. R. Grimaila, Y. C. Kim, J. D. Parham, and J. T. McDonald. 2010. “A Java based Component Identification Tool for Measuring the Strength of Circuit Protections.” *Proceedings of the 6th Cyber Security and Information Intelligence Research Workshop, CSIIRW 2010*, Oak Ridge, TN, USA, April 21-23, 2010. 1. https://doi.org/10.1145/1852666.1852668

[10] J. McDonald, Y. Kim, D. Koranek, and J. Parham. 2012. "Evaluating component hiding techniques in circuit topologies," *IEEE International Conference on Communications*, pp. 1138–1143. https://doi.org/10.1109/ICC.2012.6364542

[11] J. McDonald, E. Trias, Y. Kim, and M. Grimaila. 2010. "Using logic-based reduction for adversarial component recovery." *Proceedings of the ACM Symposium on Applied Computing*, 1993–2000. https://doi.org/10.1145/1774088.1774508

[12] D. Evans, V. Kolesnikov, and M. Rosulek. 2018. "A Pragmatic Introduction to Secure Multi-Party Computation." *Foundations and TrendsÂő in Privacy and Security* 2, 2-3 (2018), 70–246. https://doi.org/10.1561/3300000019

[13] A. C. Yao. 1986. "How to Generate and Exchange Secrets." *Proceedings of the 27th Annual Symposium on Foundations of Computer Science (SFCS '86)*. IEEE Computer Society, Washington, DC, USA, 162–167. https://doi.org/10.1109/SFCS.1986.25

[14] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. 2004. "Fairplay—a Secure Two-party Computation System." *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04)*. USENIX Association, Berkeley, CA, USA, 20–20. http://dl.acm.org/citation.cfm?id=1251375.1251395

[15] S. Zahur and D. Evans. 2015. "Obliv-C: A Language for Extensible Data- Oblivious Computation." Cryptology ePrint Archive, Report 2015/1153. https://eprint.iacr.org/2015/1153.

[16] S. Zahur, M. Rosulek, and D. Evans. 2015. "Two Halves Make a Whole." *Advances in Cryptology - EUROCRYPT 2015*, Elisabeth Oswald and Marc Fischlin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 220–250.

[17] D. Demmler, T. Schneider, and M. Zohner. 2015. *ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation*. In NDSS.

[18] X. Wang, S. Ranellucci, and J. Katz. 2017. "Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation." *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 21–37. https://doi.org/10.1145/3133956.3134053

[19] Y. Zhang, A. Steele, and M. Blanton. 2013. "PICCO: A General-purpose Compiler for Private Distributed Computation." *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security (CCS '13)*. ACM, New York, NY, USA, 813–826. https://doi.org/10.1145/2508859.2516752

[20] N. Wirth. 1998. "Hardware compilation: translating programs into circuits." Computer 31, 6 (June 1998), 25–31. https://doi.org/10.1109/2.683004

[21] D. C. Black, J. Donovan, B. Bunton, and A. Keist. 2009. *SystemC: From the Ground Up, Second Edition (2nd ed.)*. Springer Publishing Company, Incorporated.

[22] S. S. Abrar, M. Jenihhin, and J. Raik, "Extensible open-source framework for translating RTL VHDL IP cores to SystemC," *2013 IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)* (2013), pp. 112–115.

[23] R. Manikyam, J. T. McDonald, W. R. Mahoney, T. R. Andel, and S. H. Russ. 2016. “Comparing the Effectiveness of Commercial Obfuscators Against MATE Attacks.” *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering (SSPREW ’16)*. ACM, New York, NY, USA, Article 8, 11 pages. https://doi.org/10.1145/3015135.3015143

[24] P. Svensson, “Chapter 4: Boolean Algebras,” *Boolean Algebras*, Växjö, Sweden, Dec. 2009, pp. 21–31.

[25] Y. Crama and P. Hammer. 2011. *Boolean Functions: Theory, Algorithms, and Applications*. https://doi.org/10.1017/CBO9780511852008

[26] G. D. Micheli. 1994. *Synthesis and Optimization of Digital Circuits (1st ed.)*. McGraw-Hill Higher Education.

[27] M. C. Hansen, H. Yalcin, and J. P. Hayes. 1999. “Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering.” *IEEE Des. Test* 16, 3 (July 1999), 72–80. https://doi.org/10.1109/54.785838

[28] C. Collberg. 2018. “Code Obfuscation: Why is This Still a Thing?” *In Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy (CODASPY’18)*. ACM, New York, NY, USA, 173–174. https://doi.org/10.1145/3176258.3176342

[29] Yasinsac and J. T. McDonald, “Of Unicorns and Random Programs,” *Proceedings of the 3rd IASTED International Conference on Communications and Computer Networks (IASTED/CCN-2005)*, Marina del Rey, CA, USA, Oct. 24-26, 2005.

[30] H. R. Andersen and H. Hulgaard, "Boolean Expression Diagrams," *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science (LICS-1997),* Warsaw, Poland, 1997, pp. 88-98. doi: 10.1109/LICS.1997.614938.

[31] R. E. Bryant, "Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification," *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD-1995),* San Jose, CA, USA, 1995, pp. 236-243. doi: 10.1109/ICCAD.1995.480018.

[32] J. T. McDonald, E. D. Trias, Y. C. Kim, and M. R. Grimaila, “Using Logic-Based Reduction for Adversarial Component Recovery,” *Proceedings of the 25th ACM Symposium on Applied Computing,* Sierre, Switzerland, March 2010.

[33] H. Kim, “Optimizing Redundant Logic Pathways in Polymorphic Circuits,” Thesis, Department of Electrical and Computer Engineering, Air Force Institute of Technology, 2009.

[34] P. Svensson, “Chapter 5: Boolean Algebras and Electronic Circuits,” *Boolean Algebras*, Växjö, Sweden, Dec. 2009, pp. 32–40.

[35] M. Genesereth and E. Kao, “Chapter 2: Propositional Logic,” *Introduction to Logic*, Morgan & Claypool Publishers, 2012, pp. 13–22, ISSN.

[36] 2016. BSA Global Software Survey: Seizing Opportunity Through License Compliance. https://globalstudy.bsa.org/2016/.

[37] Bryant. 1986. “Graph-Based Algorithms for Boolean Function Manipulation.” *IEEE Trans. Comput. C-35*, 8 (Aug 1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

[38] C. Collberg, J. Davidson, R. Giacobazzi, Y. X. Gu, A. Herzberg, and F. Wang. 2011. “Toward Digital Asset Protection.” *IEEE Intelligent Systems* 26, 6 (Nov. 2011), 8–13. https://doi.org/10.1109/MIS.2011.106

[39] C. Collberg and J. Nagra. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection (1st ed.)*. Addison- Wesley Professional.

[40] J. McDonald, Y. Kim, and M. Grimaila. 2009. *Protecting Reprogrammable Hardware with Polymorphic Circuit Variation.*

[41] J. McDonald, Y. Kim, and D. Koranek. 2011. "Deterministic circuit variation for anti-tamper applications," *ACM International Conference Proceeding Series* (10 2011). https://doi.org/10.1145/2179298.2179376

[42] J. McDonald and Y. Kim. 2011. "Examining Tradeoffs for Hardware-Based Intellectual Property Protection."

[43] T. Miracco. 2016. “The Hidden Cost of Software Piracy In The Manufacturing Industry.” https://www.manufacturing.net/article/2016/02/ hidden-cost-software-piracy-manufacturing-industry/.