Polymorphic Circuit Generation Using Random Boolean Logic Expansion

Jeffrey T. McDonald
jtmcdonald@southalabama.edu
University of South Alabama
Mobile, Alabama

Trinity L. Stroud
tls1627@jagmail.southalabama.edu
University of South Alabama
Mobile, Alabama

Todd R. Andel
tandel@southalabama.edu
University of South Alabama
Mobile, Alabama

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 paper, we consider the nature of obfuscating algorithms that perform iterative, step-wise 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 sub-circuit 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 sub-circuit), how can we produce a semantically equivalent (polymorphic) version of the sub-circuit 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.

In this paper, 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 some adjustment in order to reach potential circuit variants.

SAC '20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3374031

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**; *Software security engineering*;

KEYWORDS

software protection, indistinguishability obfuscation, random circuits, Boolean logic, polymorphic generation

ACM Reference Format:
Jeffrey T. McDonald, Trinity L. Stroud, and Todd R. Andel. 2020. Polymorphic Circuit Generation Using Random Boolean Logic Expansion. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3341105.3374031

1 INTRODUCTION

Intellectual property (IP) is currently embedded in both software and hardware that are used in almost every area of society today. As companies can typically have billions of dollars invested in such IP [1], the theft of IP has become a major concern for tech companies and countries around the world [17]. Malicious reverse engineering, copying, cloning, and tampering form the major technical threats that adversaries can use when they are legitimate end-users and have full powers to control the execution environment of the software [4, 5]. The primary technical means to achieve protection against such man-at-the-end (MATE) attacks has been to increase the cost of malicious reverse engineering by introducing programmatic-level protections through transformation techniques generally referred to as *obfuscation* [3]. In this paper, we focus in particular on the means by which transformation takes place and consider how such transformations might defend against adversarial reverse engineering and, ultimately, IP theft.

We consider programs that come in the form of Boolean logic gates and primarily consider transformation algorithms that make incremental changes by taking small subsets of the program and replacing them with more complex versions that are semantically equivalent. We extend the work of McDonald et al. [13, 14] that first posed this approach called iterative sub-circuit selection and replacement (ISR). ISR works by selecting a subset of gates (a sub-circuit) from a combinational logic circuit program and replacing

that selection with a semantically equivalent version. This process is repeated over and over again (iteratively) until some desired level of overhead or security is reached. McDonald et al. sought to use a uniform, random replacement sub-circuit by pre-generating static libraries of circuit families based on their input, output, and gate size. From these families, a truly random circuit can be chosen out of the set of all possible functionally equivalent versions which have been pre-enumerated and stored. As a negative, such static libraries grow in factorial storage size and time to generate [13]: thus, replacement libraries are limited to small input and small gate size families. As a result, only small gate selections (2-5 gates) with small input sizes (2-5 bits) are possible for ISR.

In [13], McDonald and Kim describe three abstractions targeted by malicious circuit reverse engineers: topology, signals, and components. ISR targets the efficient recovery of these abstractions from gate-level netlist programs. McDonald and Kim also identify two trade-off spaces in the design of an obfuscator: maximizing randomness in the obfuscation algorithm and maximizing the security of the obfuscated variants produced by an obfuscator. On one hand, the algorithm should be publicly known, consistent with Kerckhoff's principle [20] that security of a cryptosystem should only rest on keeping its key secret and not its algorithm (thus avoiding the practice of security by obscurity). On the other hand, to achieve a measurable degree of hiding program abstractions, deterministic algorithms can be used produce targeted results (boundary blurring, component encryption, etc.) [13, 15], but have the risk that an adversary can devise targeted deobfuscation algorithms to undo transformations. By maximizing random (non-deterministic) choices of the obfuscator (much like in a normal crypto-system), a public algorithm has less chance of producing predictable structural patterns that can be leveraged by an adversarial deobfuscator.

McDonald and Kim [13] point out that the greater the selection space of potential replacement circuits reached by an obfuscator, the less of an advantage an adversary has to create pattern and rule-based approaches that would reverse or undo transformations. Simonaire [21] modelled an ISR algorithm as a term rewriting system (TRS) to illustrate how larger selection and replacement sizes in an ISR algorithm would cause a TRS to not coalesce, essentially defeating a deobfuscator based on term rewriting. The ideal ISR algorithm should allow an arbitrary sub-circuit selection (of any gate or input size) and an arbitrary replacement sub-circuit of any gate size (above that of the original selected sub-circuit). If circuits used for replacement are fully enumerable, adversaries are more likely to use this knowledge to target circuit features that are produced as artifacts in circuit variants. McDonald et al. [16] demonstrated that a pattern-based circuit reducer could reduce ISR circuit variants created from 2-gate selection and 3-gate replacement from 60 to 80%. However, 4-gate replacements were reduced only between 20 to 30%. Thus, we are motivated to find methods for generating semantically equivalent, larger sub-circuits as replacements in ISR, while maximizing the random selection space and minimizing either storage or generation time of replacement families.

This paper introduces an efficient algorithm with linear run-time and no storage requirement for generating sub-circuit replacements that allows selected sub-circuits to have arbitrary sizes for inputs, outputs, and gate size. The technique relies on representing sub-circuits in Boolean logic form and then iteratively applying Boolean logic laws in reverse. Whereas Boolean logic laws have traditionally been used for purposes of reducing or simplifying Boolean equations, we use them to expand or increase the Boolean expression. We call this technique random Boolean logic expansion (RBLE) and consider it as the means by which replacement circuits can be generated in ISR algorithms. We provide initial results of empirical experiments for using RBLE in ISR scenarios. The remainder of the paper provides motivating scenarios to frame the context of the work (Section 2), general background related to RBLE concepts (Section 3), and a description of the approach (Section 4). Section 5 describes our experimental framework for initial characterization of the approach and section 6 provides analysis of the results. Section 7 provides our conclusions and future work.

2 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. [18] 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 discovery of numerous flaws in the cryptographic design. Obfuscation of gate-level netlist 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 interconnectivity of building block components used to create more complex circuits. In prior work, the effectiveness of component recovery algorithms against ISR-based transformations has been evaluated based on tractable limits of small selection/replacement sizes [15, 19], 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) [8], 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 [23] 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 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 [10] was one of the first implementations of a two-party protocol: it included a high level procedural language that allows translating the secure function into a one-pass (combinational) Boolean circuit. Since then, multiple implementations

of software-to-circuit compilers for MPC construction based on C and C++ have appeared [8].

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 has moved the defining aspect of hardware programs into a more fluid form, where reprogramamble hardware is becoming the norm. Almost 20 years ago, Wirth [22] 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 represents one of the earliest examples of this hardware/software marriage and is now an IEEE standard.

The growing use of software-to-hardware programming environments for both MPC and systems design provides context for how circuit transformation algorithms can 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 [11, 12] because software constructs are fundamentally transformed into circuit representations, but are realized as software statements. In this approach, software constructs such as *if* statements or math expressions are represented in code as circuit abstractions. Such abstractions are not currently handled by traditional static and dynamic software analyzers, beyond recovery of the hardware netlist itself. Such a dichotomy would hinder traditional software reverse engineering until appropriate tools integrate analysis for both software and circuit constructs. Thus, RBLE has the potential to translate directly into software protection schemes, though this is not the focus of this paper.

3 BACKGROUND & RELATED WORK

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.

3.1 Boolean Logic Laws and Expressions

A Boolean function is a function with a domain of values $\{0, 1\}$ and of a finite number of variables of value $\{0, 1\}$. Boolean logic laws express identities on the set $B = \{0, 1\}$ with binary operations conjunction ($\vee$, OR, $+$) and disjunction ($\wedge$, AND, $*$), and unary operation negation ($\overline{x}$, NOT, x'). Such laws include annihilation, identity, commutativity, associativity, idempotence, absorption, distribution, complementation, involution, and De Morgan's [6].

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

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

with our notation for the above expression being:

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

Boolean laws are normally applied repeatedly to reduce Boolean expressions to their simplest form, although the process is complex because pattern matching is required and a correct ordering of applied laws is required to achieve complete simplification [7].

#	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

Table 1: Boolean expression reductions

3.2 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 [9] and visually in schematic form. Figure 1 illustrates a small 5 input, 2 output, 6 gate combinational circuit 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 $f_i : B^n \rightarrow \{0, 1\}$, where $i = 1..m$ [7].

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 [2, 7]. Figure 1 illustrates the corresponding Boolean expression for the circuit structure.

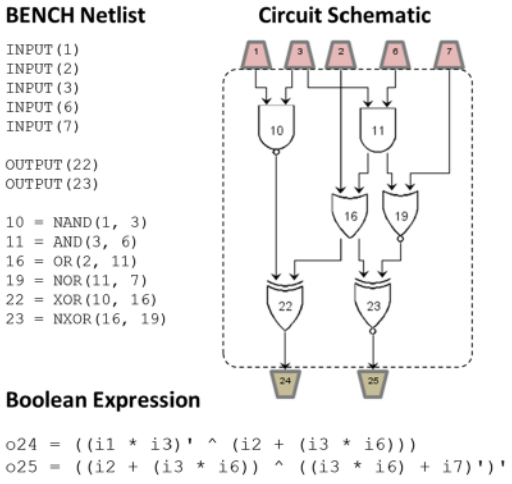

Figure 1: Equivalent circuit representations

Logic circuits are typically grouped in families based on their input, output, and gate sizes. We use the notation δ_{X-Y} to define the set of all circuits the same input size X and output size Y. We use the notation δ_{X-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 $\Omega = \{AND, OR, NOT\}$, $\Omega = \{NAND\}$, $\Omega = \{NOR\}$, or $\Omega = \{AND, OR, XOR, NAND, NOR, NXOR\}$. The fan-in of a gate is the number of unique inputs fed to the gate. For purposes of generating legal circuits within a family, we use the following rules to govern circuit properties related to their structure:

(1) Symmetry: GATE(X1,X2) equivalent to GATE(X2,X1)?
(2) Redundancy: Gates with same fan-in and gate type?
(3) Constant Signals: Allow constants 0 or 1 as nodes?
(4) Degeneracy: Gate inputs originate from same source gate?
(5) Fan-in: Allow gate fan-in greater than 2?
(6) Basis: What are allowable gate types Ω?
(7) Size: Do sets contain gates with exact size or less?
(8) Outputs: Allow gates with non-zero fan-out 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. We generate circuit families statically that minimize redundancy, disallow degenerate conditions and constant signals, use exact size, and allow only binary gates: these properties are typical for standard building blocks in larger combinational circuits. Figure 2 illustrates the legal family of δ_{2-1-1} circuits with this rule set given basis $\Omega = \{AND, OR, XOR, NAND, NOR, NXOR\}$, which consists of only the six basic logic gates themselves.

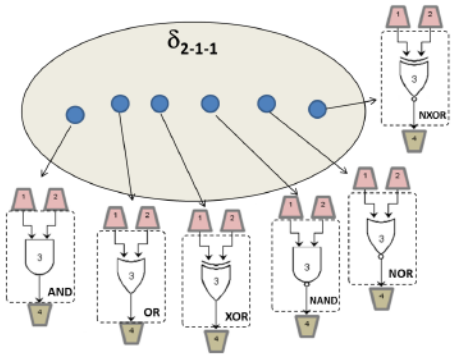

Figure 2: δ_{2-1-1} circuit family

4 BOOLEAN REDUCTION AND 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 Boolean logic laws. 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 actually change the structure of the circuit. For example, associativity or distributivity laws or laws which change the number of variables or values represented in part of the overall Boolean expression result in polymorphic variation. Laws such as commutativity would not, and therefore we can remove laws #3 and #4 from Table 1, leaving us with an optimized and reordered list of Boolean logic laws seen in Table 2. 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 3 possible expansions, the Boolean value 1 has 3 possible expansions, and any Boolean variable A has 10 possible expansions.

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', $(A + B)'$, $(A * B)'$, $(A \hat{} B)$, $(A \hat{} B)'$, $A * (B + C)$, $A + (B * C)$, $(A * B) * C$, and $(A + B) + C$. In Table 2, 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 rule 22 and 23, $(A \hat{} B)$ represents a 2-input *XOR* gate and $(A \hat{} B)'$ 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 $(A \hat{} A)$ or $(A * A')$, where A is any arbitrary variable that is already present in the expression.

#	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

Table 2: Boolean expression expansions

For multiple output circuits, each output is represented as its own Boolean logic expression.

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 expansion. 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 on the basis of either the number of expansions performed or on 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: $BE \rightarrow BE_1, BE_2, BE_3, ..., BE_n$. 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: $BE \rightarrow BE_1, BE_2, BE_3, ..., BE_{MAXEXPANSIONS}$, 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 gate size $size(C') = n$.

(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: $BE \rightarrow BE_1, BE_2, ..., BE_{MAXEXPANSIONS}$, 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 $size(C') >= n$.

Of the three policies, *STRICTSIZE* and *TARGETSIZE* are non-deterministic 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. 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 that can be applied to the current Boolean expression and returns a random choice from the set. Table 3 provides an example of applying a *FIXED* policy on a Boolean expression where 5 expansions are applied to the expression $o1 = (i0 * i1)$. Figure 3 illustrates circuit realization of the corresponding Boolean expressions created through expansion in Table 3.

5 EXPERIMENTAL EVALUATION

To provide an initial evaluation of the efficacy of RBLE in comparison to pre-generated static libraries, we ran three types of experiments that generated distributions of replacement circuits using both approaches. The goal of the experiments were 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

Algorithm 1: Random Boolean Logic Expansion (RBLE)

```
   input  :C,P,n
   output:C', where ∀x : C(x) = C'(x)
1  BE ← convert(C); done ← false;
2  fixed ← 0; attempts ← 0; numexp ← 0;
3  while not done do
4      expansions ← profile(BE);
5      expansion ← select(expansions);
6      BE ← apply(BE, expansion);
7      Ĉ ← realize(BE);
8      if P == FIXED then
9          fixed ← fixed + 1;
10         if fixed = n then
11             C' ← Ĉ; done ← true ;
12         end
13     else
14         if (P == STRICTSIZE and size(Ĉ) = n) then
15             C' ← Ĉ; done ← true ;
16         else if (P == TARGETSIZE and size(Ĉ) >= n) then
17             C' ← Ĉ; done ← true ;
18         else
19             numexp ← numexp + 1;
20             if numexp > MAXEXPANSIONS then
21                 BE ← convert(C); z ← 0;
22                 attempts ← attempts + 1;
23                 if attempts > MAXATTEMPTS then
24                     C' ← null; done ← true;
25                 end
26             end
27         end
28     end
29 end
30 return C';
```

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**

Table 3: Example Boolean expansion sequence

equivalent polymorphic variants (referred to as CIRCLIB [16]). 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 δ_{2-1-1} circuit family, which are seen in Figure 2. Figure 4 provides a visual reference to our experimental framework, which

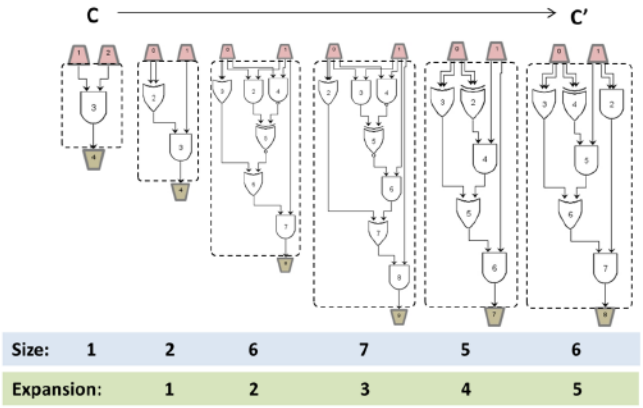

Figure 3: Example Expansion Circuit Realization

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 [15, 16]. All experiments were performed on an HP ZBook 17 G2 laptop with an Intel i7-4710MQ 2.50 GhZ CPU and 32GB installed RAM.

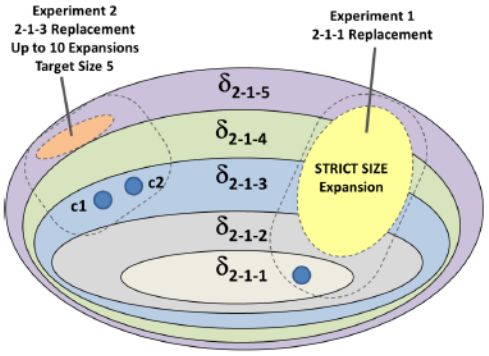

Figure 4: Empirical evaluation framework

5.1 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 6 basic gate types in the δ_{2-1-1} family, we performed two sets of generations that created a sum total of 188,000 circuits:

(1) 1000 variants from CIRCLIB and 1000 variants from RBLE, totalling 6,000 circuits for each gate type and method, of target size 2, 3, 4, and 5. The only exception was that the δ_{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) 10000 variants from CIRCLIB and 10000 variants from RBLE, totalling 60,000 circuits for each gate type and method, of target size 2, 3, and 4. The only exception was that the δ_{2-1-2} 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.

5.2 Experiment 2: Fixed Expansion/Targeted Replacement

We evaluate RBLE under a *FIXED* expansion policy, using six pairs of circuits chosen from the δ_{2-1-3} family, where each pair of δ_{2-1-3} circuits (C_1,C_2) are semantically equivalent to one of the basic gate circuits in the δ_{2-1-1} family (*AND*,*OR*,*XOR*,*NAND*,*NOR*,*NXOR*). For each circuit in each circuit pair (C_1,C_2), 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 CIRLIB, each circuit in the pair (C_1,C_2) 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 (C_1,C_2) 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.

5.3 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 CIRLIB 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.

6 RESULTS

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) δ_{2-1-1} family. Given standard circuit creation options for CIRCLIB (described in section 3.3), each original gate only has 4 possible variants in the δ_{2-1-2} family. The replacement circuits as seen in Figure 5 show

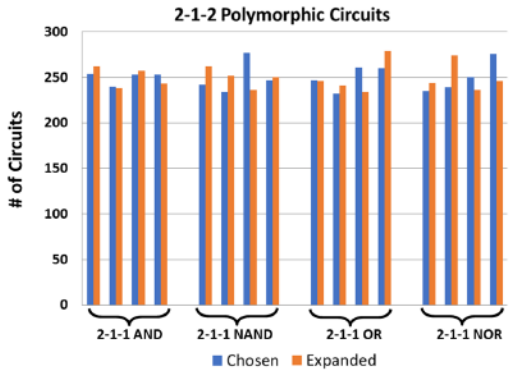

Figure 5: Number circuits for δ_{2-1-2} Strict Replacement-1K

that both RBLE and CIRCLIB create roughly equal distributions for all 4 circuits, for all 4 gate types.

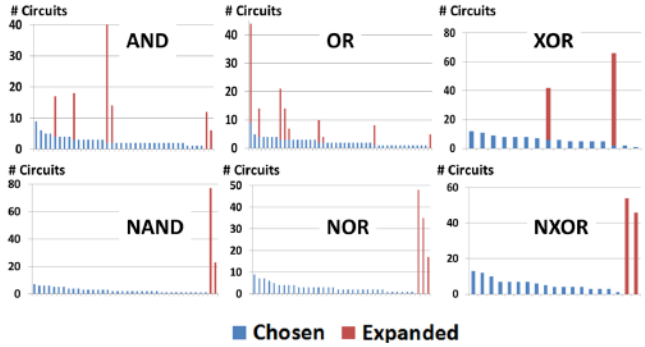

Figure 6: Number circuits for δ_{2-1-3} Strict Replacement-1K

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 δ_{2-1-1}. 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 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 δ_{2-1-1}. 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

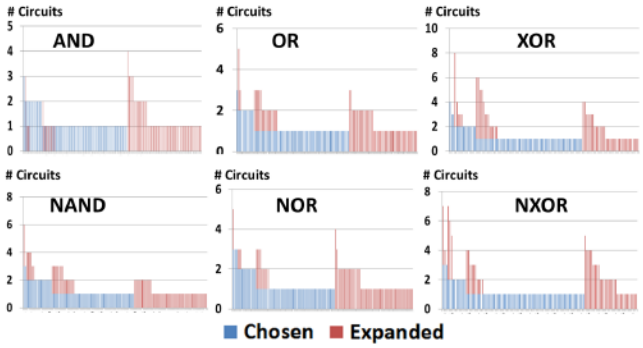

Figure 7: Number circuits for δ_{2-1-4} Strict Replacement-1K

reach circuits that are not part of the CIRLIB 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.

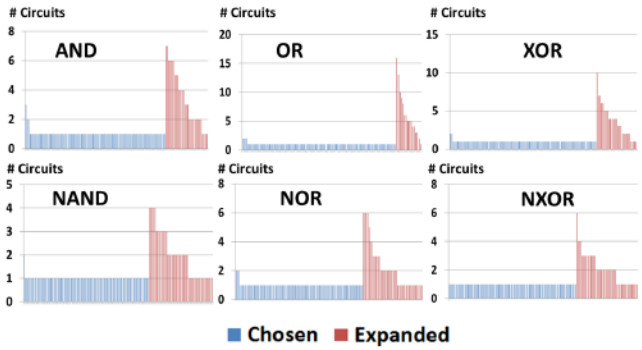

Figure 8: Number circuits for δ_{2-1-5} Strict Replacement-1K

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 δ_{2-1-1}. 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 1000 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 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 δ_{2-1-1}. The chart summarizes the number of circuits (each bar representing an identical matching circuit between the RBLE and CIRCLIB methods) for all gate types, ordering variants in descending order by the frequency with which they are 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

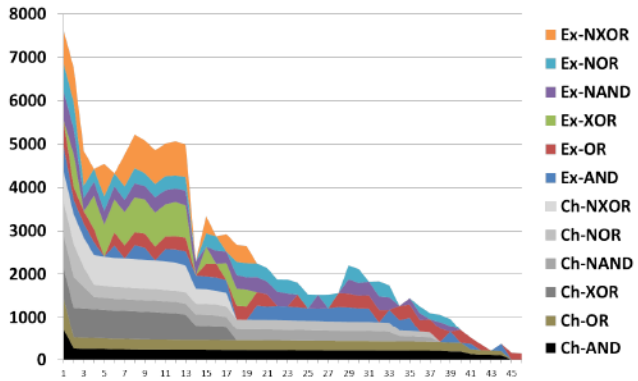

Figure 9: Number circuits for δ_{2-1-3} Strict Replacement-10K

fact that there are overlaps of structurally equivalent circuits in CIRCLIB, so that certain circuits have a higher probability of being chosen. We observe also 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.

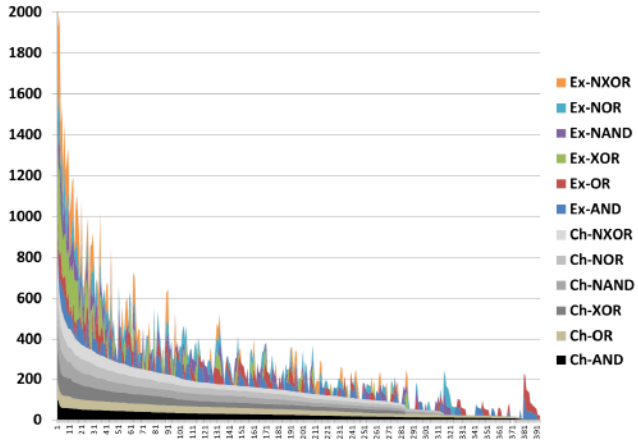

Figure 10: Cumulative circuits δ_{2-1-4} Strict Replacement-10K

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 δ_{2-1-1}. 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 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

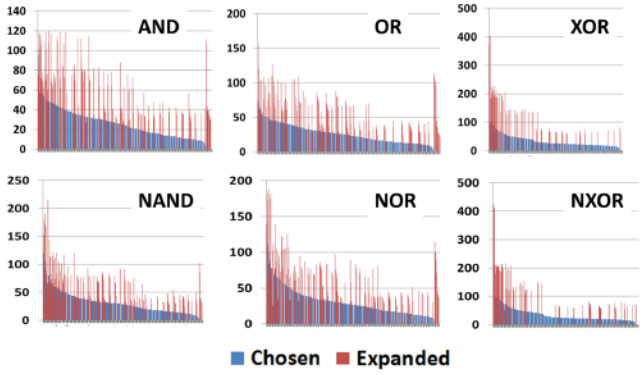

Figure 11: Number circuits δ_{2-1-4} Strict Replacement - 10K

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 to RBLE. 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 δ_{2-1-2} family, they are nearly 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 1000 for each of the original δ_{2-1-1} 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 sets of Boolean laws.

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 δ_{2-1-5} family. We can observe in Figure 12 the distribution of CIRCLIB replacements, where the number of actual CIRLIB 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 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

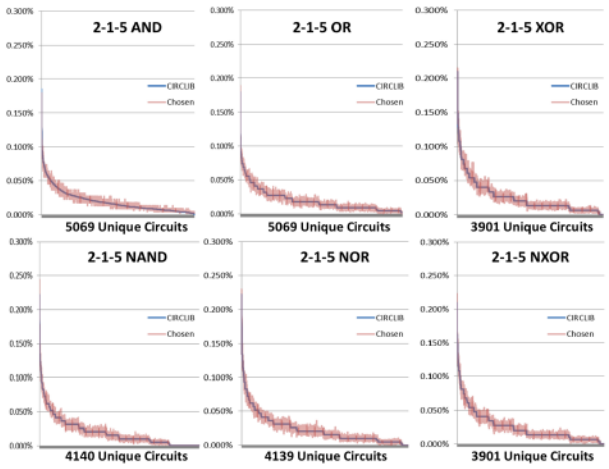

Figure 12: Distributions for δ_{2-1-5} Chosen Replacement

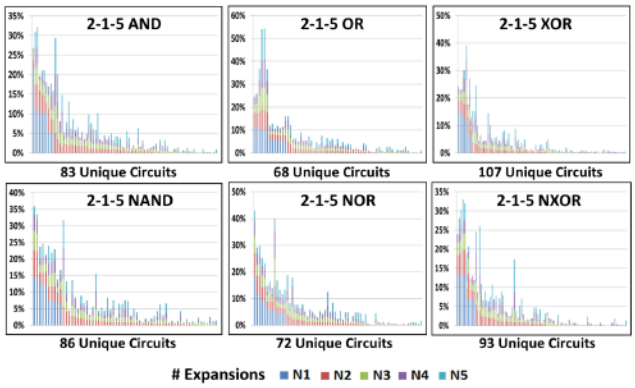

Figure 13: Distributions for δ_{2-1-5} Expanded Replacement

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.

Experiment 2 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 δ_{2-1-3} circuits with each pair being semantically equivalent to

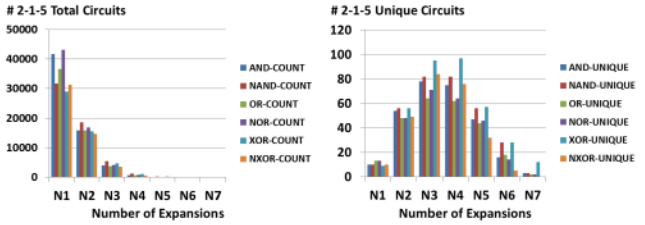

Figure 14: δ_{2-1-5} expanded replacement summary

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 δ_{2-1-X} no greater than size 7 [16].

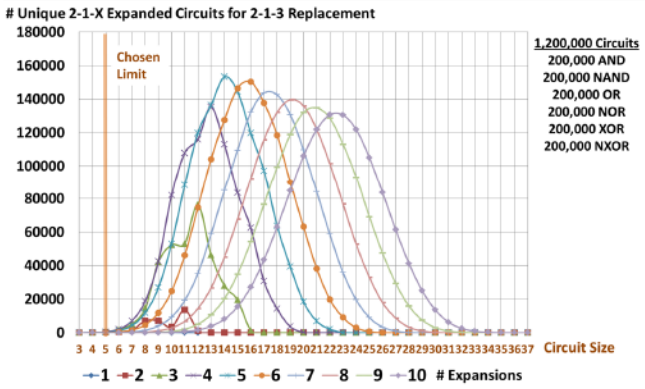

Figure 15: Unique expanded replacements (200,000 per type)

7 CONCLUSIONS AND FUTURE WORK

In this paper, we introduce 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 δ_{2-1-1} and δ_{2-1-3} 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 has ability to generate a large number of unique variants uniformly when various expansion sizes are used. Based on these initial results, our future work will 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 size, as the signals are typically considered to be provided outside the circuit.

8 ACKNOWLEDGEMENTS

This work is partially supported by National Science Foundation awards 1811560 and 1811578 in the NSF 17-576 Secure and Trustworthy Cyberspace (SaTC) program.

REFERENCES

[1] 2016. BSA Global Software Survey: Seizing Opportunity Through License Compliance. https://globalstudy.bsa.org/2016/. Accessed: 2019-08-12.
[2] R. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (Aug 1986), 677–691. doi: 10.1109/TC.1986.1676819.
[3] C. Collberg. 2018. Code Obfuscation: Why is This Still a Thing?. In *Proc of the 8th ACM Conference on Data Application Security and Privacy (CODASPY '18)*. ACM, New York, NY, USA, 173–174. doi: 10.1145/3176258.3176342.
[4] C. Collberg, J. Davidson, R. Giacobazzi, Y. Gu, A. Herzberg, and F. Wang. 2011. Toward Digital Asset Protection. *IEEE Intelligent Systems* 26, 6 (Nov. 2011), 8–13. doi: 10.1109/MIS.2011.106.
[5] C. Collberg and J. Nagra. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection* (1st ed.). Addison-Wesley.
[6] Y. Crama and P. Hammer. 2011. *Boolean Functions: Theory, Algorithms, and Applications.* doi: 10.1017/CBO9780511852008.
[7] G. De Micheli. 1994. *Synthesis and Optimization of Digital Circuits* (1st ed.). McGraw-Hill Higher Education.
[8] D. Evans, V. Kolesnikov, and M. Rosulek. 2018. A Pragmatic Introduction to Secure Multi-Party Computation. *Foundations and Trends in Privacy and Security* 2, 2-3 (2018), 70–246. doi: 10.1561/3300000019.
[9] M. 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. doi: 10.1109/54.785838.
[10] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. 2004. Fairplay—a Secure Two-party Computation System. In *Proc. of the 13th Conference on USENIX Security Symposium (SSYM'04)*. USENIX Association, Berkeley, CA, USA.
[11] R. Manikyam. 2019. *Program Protection Using Software Based Hardware Abstraction.* Ph.D. Dissertation. University of South Alabama.
[12] R. Manikyam, J. McDonald, T. Andel, and M. Yampolskiy. 2019. Poster: Analyzing Program Protection Using Software-Based Hardware Abstraction. In *3rd Workshop for Women in Hardware and Systems Security (WISE)*.
[13] J. McDonald and Y. Kim. 2012. Examining Tradeoffs for Hardware-Based Intellectual Property Protection. In *Proc. of the 7th Intrnl Conference on Information Warfare and Security 2012 (ICIW 2012)*. 192–202.
[14] J. McDonald, Y. Kim, and M. Grimaila. 2009. Protecting Reprogrammable Hardware with Polymorphic Circuit Variation. In *Proceedings of the 2nd Cyberspace Research Workshop, June 2009, Shreveport, Louisiana, USA.*
[15] J. McDonald, Y. Kim, D. Koranek, and J. Parham. 2012. Evaluating component hiding techniques in circuit topologies. *IEEE International Conference on Communications*, 1138–1143. doi: 10.1109/ICC.2012.6364542.
[16] 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. doi: 10.1145/1774088.1774508.
[17] 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/. Accessed: 2019-08-12.
[18] K. Nohl, D. Evans, Starbug, and H. Plötz. 2008. Reverse-engineering a Cryptographic RFID Tag. In *Proceedings of the 17th Conference on Security Symposium (SS'08)*. USENIX Association, Berkeley, CA, USA, 185–193. doi: 10.1109/54.785838.
[19] J. Parham, J. McDonald, M. Grimaila, and Y. Kim. 2010. A Java based Component Identification Tool for Measuring the Strength of Circuit Protections. In *Proc. of the 6th CSIIRW 2010, Oak Ridge, TN, USA, April 21-23, 2010.* 1.
[20] F. Petitcolas. 2011. *Kerckhoffs' Principle.* Springer US, Boston, MA, 675–675. doi: 10.1007/978-1-4419-5906-5_487.
[21] E. Simonaire. 2008. *Sub-Circuit Selection and Replacement Algorithms Modeled as Term Rewriting Systems.* Master's thesis. AF Inst of Technology, WPAFB, OH.
[22] N. Wirth. 1998. Hardware compilation: translating programs into circuits. *Computer* 31, 6 (June 1998), 25–31. doi: 10.1109/2.683004.
[23] A. Yao. 1986. How to Generate and Exchange Secrets. In *Proc. of the 27th Annual Symposium on Foundations of Computer Science (SFCS '86)*. 162–167. doi: 10.1109/SFCS.1986.25.