Using Logic-Based Reduction For Adversarial Component Recovery

J. Todd McDonald, Eric D. Trias, Yong C. Kim, and Michael R. Grimaila
Air Force Institute of Technology
Wright-Patterson Air Force Base, Ohio 45433
001-937-255-3636
[jmcdonal, etrias, ykim, mgrimail]@afit.edu

ABSTRACT
A current means to protect intellectual property embedded in both circuits and software involves creating a functionally equivalent variant with subjective qualities related to difficulty of reverse engineering. In this paper, we consider the problem of protection in a smaller, generalized class of programs based on Boolean logic primitives. We consider Boolean logic reduction as one means to quantify hardness of undoing structural transformations designed to impede reverse engineering. We detail our experiences in using both commercial synthesis tools and organic red-team tools that simplify transformations using known basic logic patterns. Using simple component recovery on candidate circuits, we show how specific variation methods impact adversarial analysis and posit relationships between specific transformations and corresponding difficulty of reversal.

Categories and Subject Descriptors
K.6.5 [**Management of Computing and Information Systems**]: Security and Protection – *invasive software, physical security, hacking.*

General Terms
Security, Theory, Experimentation, Measurement, Design.

Keywords
Hardware security, intellectual property protection, reverse engineering, circuit obfuscation, anti-tamper applications

1. INTRODUCTION
Measuring the difficulty of reverse engineering or precisely defining (any) concrete program characteristic hidden by an obfuscating transformation remains a hard problem for researchers [3,10]. On one hand, creators of intellectual property would like to delay, frustrate, or completely prevent recovery of their original design components when adversaries perform analysis on realized versions of their programs (e.g., physical ASIC chips or executable machine code). On the other hand, forensics experts would like some notional measure of hardness or an upper bound on time when considering the difficulty of reverse engineering malicious viruses. In either case, researchers and practitioners both would prefer objective measures of hardness or mathematically definable means of hiding over subjective assessments that rely on skill or familiarity of a reverse engineer.

From a theoretical perspective, no general transformation algorithm can fully hide *all* information in a circuit variant when compared to only the information that can be obtained by just running input/output pairs of the original circuit [1, 2]. The notion of a *virtual black box* that approaches perfectly secure hardware protection which shields all internal analysis of a circuit structure remains impossible to produce. However, there remains the question of whether *any* form of variation provides *any* useful form of protection.

We consider in this paper the context of white-box polymorphic structural variation and the degree in which we can measure, characterize, and compare effectiveness of these kinds of transformations. In particular, we consider applications where white-box polymorphism may prove useful. By considering straight-line programs based on simple logic grammars (*AND*, *OR*, *NOT*, and their basic derivatives), we focus more precisely on what actually defines obfuscation using a polymorphic circuit generator that creates functionally equivalent versions of circuits. We also report on the effectiveness of component-hiding as a by-product of random transformations used by the generator and interpret their effectiveness by using pattern matching logic reduction heuristics compared to a standard commercial logic synthesis tool.

To organize the paper, we first discuss in Section 2 the relationships between circuit structural variation and protection properties. Section 3 provides a description of pattern-matching heuristics which reduce circuit variants and our results with using these techniques on small merged circuits. Section 4 gives our results with analyzing several worst-case circuits using simple pass/fail tests from our custom logic minimization algorithm and a commercial synthesis tool.

2. CIRCUIT VARIATION / PROTECTION
In general, obfuscation seeks to find suitable variants of an original program or circuit that accomplish the same function as the original. Practitioners typically define suitability of the variant by whether or not the variant demonstrates *some* security property of interest. In most cases, we link the security property to driving up the cost of reverse engineering or demonstrating that heuristic recovery of certain program information is equivalent to the work

SAC'10, March 22-26, 2010, Sierre, Switzerland.

of solving a known hard problem [3]. We seek to identify the point at which a given variant may be considered "useful" for a practical obfuscation purpose.

2.1 Experimental Language

For our purposes, we consider the language of logic circuits which consist of imperative Boolean functional statements. Combinatorial circuits are by definition finite, acyclic, and represented by directed acyclic graphs (DAGs). Because combinational logic has no loops, we can express the black-box behavior of a circuit readily by enumeration of all inputs, subsequent evaluation and propagation of signals on all intermediate gates, and recording of the corresponding output. We characterize black-box circuit behavior as a Boolean function, $f:\{0,1\}^n \rightarrow \{0,1\}^m$, where n is the input size (or input length) and m is the output size (or output length) in bits. We use a traditional BENCH grammar to define our experimental circuits [4,7]. Based on this grammar, we characterize a circuit in terms of four parts: number of inputs, number of outputs, intermediate gate size, and the circuit's basis set $\Omega \subseteq \{AND, OR, XOR, NAND, NOR, XNOR\}$. A circuit over Ω is a DAG having either nodes mapping to functions in Ω (referred to as *gates*) or having nodes with in-degree 0 being termed *inputs*. We also distinguish one (or more) intermediate nodes as *outputs*.

2.2 Preserving vs. Changing Semantics

Given a circuit, we define two major categories of transformations: semantic preserving and semantic changing. We consider all structural or syntactic changes (i.e., renumbering gate IDs) to a circuit as being a white-box change. Obfuscation, by its classic theoretical definition, considers only changes that preserve or keep the original function of a circuit (*semantic preserving*) [1,2]. In [5], the authors describe the value of white-box transforms where the function is changed, but the output of the function for such constructions must be *recovered* or *refined*. For recovery, the output of the circuit variant requires decryption similar to normal data ciphers that use keys. For refinement, the output of the circuit is hidden in plain sight, but intermixed with real circuit output.

Figure 1 illustrates the theoretical function for all semantic preserving obfuscation algorithms; all such algorithms select equivalent circuits from a given functional family set. For example, given a circuit family set denoted by $\delta_{X\text{-}Y\text{-}S\text{-}\Omega}$, where X is input length, Y is output length, S is intermediate gate size, and Ω is basis, we let δ_C represent a subset of this family containing all circuits that compute the same function as circuit C. Standard semantic-preserving obfuscation algorithms take circuit C and return a variant C'_A from the subset δ_C, such that $O(C) = C'_A$. The characterization of the algorithm O and the distribution of circuits it produces remains an open problem, but the random program model of [5] and the best possible obfuscation definition of [1] both suggest that the strongest algorithm is one which returns candidates from the set δ_C on an equiprobable or randomly selected basis. If the environment exists where the functionality of the circuit may be changed (i.e., the context in which a circuit is used may be adapted), then the possibility exists for transformations that change the black-box behavior of a circuit in predetermined ways. Figure 1 illustrates another form of obfuscating algorithm which takes circuit C and returns circuit C'_B such that $O(C) = C'_B$, but $C'_B \in \delta_{CB}$, which is another (different) functional family subset of $\delta_{X\text{-}Y\text{-}S\text{-}\Omega}$.

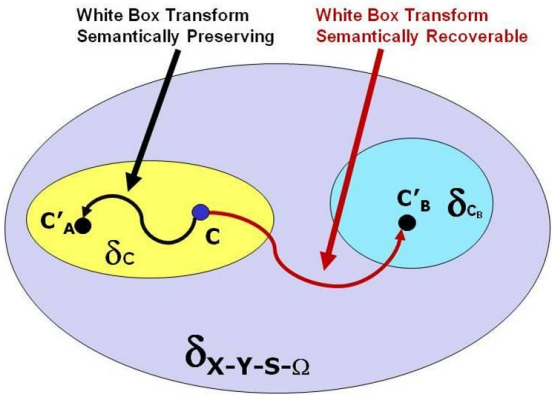


Figure 1. White-box transformations that preserve and change black-box semantics.

As negative and positive theoretic results indicate, changing I/O behavior may be the only hope of proving anything concrete about obfuscation. We motivate our interest in white-box semantic preserving changes from 1) our desire to characterize fitness of circuits within the set δ_{CB} so that we might evaluate whether black-box transformations are seamlessly integrated into the structure of a circuit variant such as C'_B, and 2) our desire to characterize some positive benefit of providing a circuit variant C'_A (versus the original) when the environment is not conducive to black-box changes to the original circuit. In this paper, we focus on the latter issue and discuss a practical model for measuring the effect of white-box changes to circuit properties of interest.

2.3 Variation Methodology

For this study, we use a polymorphic circuit generator that employs iterative sequences of small sub-circuit selections and replacement. Each individual selection and replacement represents a variation on a small scale: the generator replaces each sub-circuit selected with a functionally equivalent replacement sub-circuit. We consider specifically how to characterize and measure whether aspects of circuit obfuscation (a hiding property of interest) manifests as an artifact of the variation process. Our experimental setup analyzes the nature of the structural and logical changes produced by our variation algorithm. Based on the logic and component properties of the selection and replacement operations themselves, we characterize specific trends that occur within a circuit that undergoes transformation.

What percentage of the transformations introduced via a sub-circuit selection and replacement algorithm can be *undone* via optimization or logic minimization? If we could answer this question, we may have better insight into measuring the degree of actual obfuscation taking place. In this particular paper, we look at the effect of sub-circuit transformations themselves and formulate an approach for answering the measurability question, which we now detail in the remainder of the paper.

Figure 2 provides a summary view of the variation algorithm under our study: the algorithm takes a version of a circuit, performs some selection strategy that chooses gates (constituting a sub-circuit C_{sub}), replaces that sub-circuit with an equivalent version (C_{rep}), and then repeats the process. We define such an obfuscation experiment as a 5-tuple: (C,n,λ,τ,ψ) where C is an original circuit, n is the number of iterations (sequence of

selection/replacement), λ is a set of selection algorithms with cardinality n where $s_i \in \lambda$ indicates the selection algorithm performed during iteration i, τ is a set of selection algorithms with cardinality n where $r_i \in \tau$ indicates the replacement algorithm performed during iteration i, and ψ is a set of gates which are given selection priority during the incremental execution of the experiment.

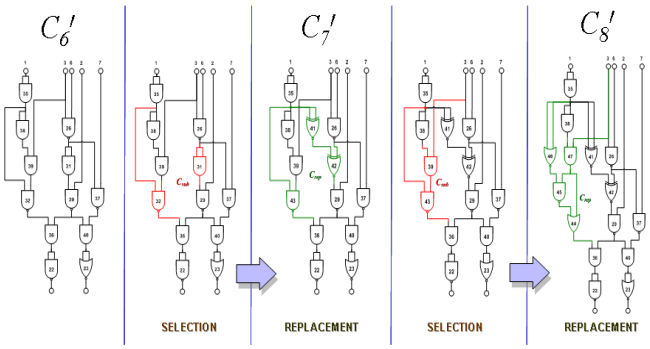


Figure 2. Polymorphic circuit generation via select/replace.

Experiments using this generational approach essentially create a random walk from a starting C to some final circuit C' within a function family δ_C, as seen in Figure 1. There exists four different decisions that can be made for each selection and replacement which define how an experimental walk progresses. These decisions define the selection set λ and replacement set τ for a given experiment and are described as follows:

- *Random selection:* Select a sub-circuit $C_{sub} \subset C$ based on a purely random basis.
- *Random replacement:* Select a replacement circuit $C_{rep} \in \delta C_{rep}$ on a purely random basis. Here, δC_{rep} represents the set of all functionally equivalent replacements for C_{rep}.
- *Smart selection:* Only select sub-circuits C_{sub} which have a particular property or select sub-circuits based on partially random/partially deterministic basis.
- *Smart replacement:* Similar to smart selection, only select replacement circuits C_{rep} from a library which have a particular property. In this case, the replacement may be generated or deterministically chosen based on specific properties of the selection

2.4 Protective Properties

The goals of a reverse engineer operating with malicious intent vary greatly. In terms of copying or reversing intellectual property for the purpose of adaptation, resell, and reuse, the essential goal of reverse engineering centers on understanding the design intent of the original circuit. In many cases, the same reverse engineering goals for a circuit matches with those of generalized software. Many practitioners define reverse engineering as analyzing a system to identify its partitions and their relationships with each other–for the explicit purpose of creating an identical and more easily understandable version at a higher level of abstraction [6].

We can describe the levels of digital circuit abstraction in the follow manner, based on increasing level of detail: 1) architectural (behavioral), 2) register transfer language (RTL), 3) gate level, 4) transistor level, and 5) physical layout. The BENCH grammar essentially represents a gate level view of a circuit description. In our experimental framework, we examine gate-level variants with identical functionality to an original gate-level circuit description. We define the goals of a reverse engineer along the lines of abstraction and consider the following protective categories:

1) **Topology Recovery:** reproducing the original gate level structure of the circuit
2) **Signal Recovery:** reproducing some or all of the internal signal transitions of intermediate gates within the original circuit (the resulting truth table columns that result when input signals are propagated through a circuit structure to produce output signals)
3) **Component Recovery:** reproducing the architectural or component level relationships of the original circuit (this is in essence the well-studied problem of *module identification* [8, 11]). In [11], for example, module identification consists of two parts: first, enumerating potential sub-circuits, and second, matching known component functions to those sub-circuits. We refer to this entire process of successful enumeration and matching as **component identification** or **component recovery**.
4) **Control Recovery:** reproducing some or all of the critical signals that act as control functions of the original circuit.
5) **Functional Recovery:** identifying the function of the original circuit, either by reduced logic synthesis or canonical truth table derivation.

In [7], Hansen *et al.* list similar reverse engineering techniques and goals based on deriving the following: (known/standard) library modules or program components, repeated modules, expected global structures, computed functions, control functions, bus structures, and common names. The underlying goal of such analysis assumes that logic gates grouped into higher level modules makes reverse engineering easier. Of interest, we note that Hansen's definition of black box relates to circuit understanding on a large scale: when all other derivation methods fail, we must consider a circuit (or sub-circuit) a black box if the function is unknown or if the circuit consists of truly random logic.

We note that size *alone* of the variant may or may not have any bearing on the ability of any adversary to accomplish goals related to these protective categories. We may observe that, in general, larger intermediate gate size may result in longer analysis times or may consume larger resources (memory and storage), but no definitive link exists between protection and size per se. Our interest concerns whether definable properties of protection emerge at certain sizes or whether they emerge based on the method by which we create a circuit variant using the experimental choices described in Section 2.3.

2.5 Component Recovery

Since component recovery plays a major role in reverse engineering study and research, we focus on this adversarial goal as it relates to analyzing circuit variants. By narrowing our focus, we setup a test environment that makes failure (to hide components) easy to identify. Even though measuring *absolute* success requires further work, we report here the results of experimental studies along these partial lines.

Given a circuit C, its gate set G, its input set I, and an integer $k > 1$, where k is the number of components, a set M of components $\{c_1,\ldots, c_k\}$ partitions G and I into k disjoint sets of inputs and/or gates. Figure 3 illustrates the four base cases by which a component might partition a circuit in relationship to other components or gates. Consider a circuit whose component set consists of four disjoint gate sets, $M = \{A,B,C,D\}$. Component A receives input from the circuit boundary and produces output received by another component; component B receives input from

a component and produces output received by another component; component *C* receives input from another component and produces output at the circuit boundary; and component *D* receives input from the circuit boundary and produces output at the circuit boundary.

In terms of component hiding that occurs as an artifact of white-box structural variation, component *D* in Figure 3 represents at least one instance of a "Kobayashi maru[1]" or no-win scenario for an obfuscating algorithm. Because semantic preserving algorithms produce variants that must maintain the function of the original circuit, gates within component *D* have the greatest probability of recovery when compared to scenarios *A*, *B*, and *C*. This is because no function preserving obfuscator can alter the I/O relationships of component *D*, thereby giving analysis algorithms an advantage to distinguish gates within *D* from the rest of the circuit. It should also follow intuitively that components that fit the pattern of component *B* have a greater chance of hiding because they are fully contained within the circuit boundary.

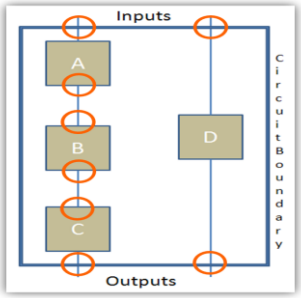
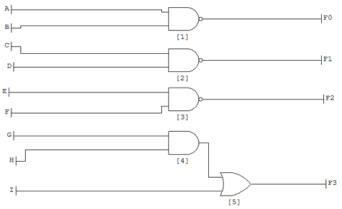


Figure 3. Component configuration base cases.

We can think of this example also from a gate level view. Consider a five gate sub-circuit with 9 inputs, 4 outputs, and five intermediate gates as in Figure 4. In this example, the logical representation of the circuit in its most reduced form (accomplished via Espresso heuristic minimization) shows that output F0 is driven by only inputs A and B. Likewise outputs F1, F2, and F3 have clear independence of relationships between their inputs terms and the output terms of the circuit. In other words, input A drives only output F0 and no other. This represents the component D scenario of Figure 3, if we consider gate 1 to be the component.



Reduced Logic:
F0 = B' + A' ;
F1 = D' + C' ;
F2 = F' + E' ;
F3 = G H + I;

Figure 4. Gate-level component example.

We define the problem of component hiding in a no-win scenario (one where the component touches both I/O boundaries of the circuit) as being the identification of the independent relationships that exist between input and output terms. If our goal is to obscure or hide the fact that 4 original components existed in the circuit of Figure 4, then our job is futile if *all* we do is change the white-box structure of the component internally. This is because any 9-input/4-output circuit is within the realm of canonical minimization, regardless of how many gates comprise its internal structure.

This also illustrates clearly the impossibility of obfuscation in certain cases–namely, those cases where logic analysis or truth table analysis would reveal the property being hidden or obscured. In the case of Figure 4, logic analysis of *any* gate size variant (even one with say 10,000,000,000,000 gates) would clearly reveal the component relationships between *F0/A/B*, *F1/C/D*, *F2/E/F*, and *F3/G/H/I* because full truth table enumeration remains tractable. Logically speaking, we can also consider the goal of component hiding (for a worst case scenario such as this) as the ability to produce a variant where outputs of one component are driven or have dependency on inputs not part of its original component structure. We refer to this property as *term interleaving*, *induced redundancy*, or *structural overlapping*. If we take output F0 of Figure 4 as an example, we can perform Boolean operations to accomplish interleaving in the following manner: $F0 = B'(C+C')+A'(H+H') = B'C + B'C' + A'H + A'H'$. This version defines output *F0* with some redundant terms (from inputs *C* and *H*). Of course given the variant that contains $F0 = B'C + B'C' + A'H + A'H'$ structurally, it is easy to reduce the logic of the terms and remove the overlap, just based on truth table synthesis alone.

We are therefore motivated by instances where an adversary does not have the ability to synthesize such components based on truth table logic or instances where the adversary must rely (only) on heuristic minimization techniques. We can achieve such scenarios when the input/output size combined with the internal gate size of the circuit makes such logic analysis and reduction intractable in the average case. In other words, the only guarantee that an adversary recovers components when they are originally configured in such "no-win" scenarios (D in Figure 3) is when the circuit is conducive to perfect minimization to begin with (i.e., the semantics of the truth table would clearly reveal component independence or boundaries). Heuristics (such as ESPRESSO) may be able to reduce a circuit, but run the risk of not revealing independence of input/output terms that were originally independent. Our study centers on analyzing the effect of logic minimization in cases where full circuit synthesis is not possible or remains intractable in the canonical case. If abstract this now to the circuit level, we can derive a class of circuits that define hard cases for component hiding. Such circuits are created by merging or composing multiple independent circuits together into one common circuit, where only the input/output boundary of the circuit is shared.

Figure 5 illustrates such a circuit *C* that has component set $M = \{c_1,c_2\}$ and each component matches the worst case component scenario *D* of Figure 3. *C'* illustrates a variant of circuit *C* where *apparent* merging of components has occurred via some form of induced logical redundancy. If circuit *C* were of a form that prevents canonical truth table minimization and synthesis (e.g., input size > 60), we reduce an adversary to heuristic logic minimization or white-box topology analysis as the means to perform component recovery. This adversarial limitation defines the context and goals of our study, which we expound next.

[1] http://en.wikipedia.org/wiki/Kobayashi_Maru: A test in the fictional Star Trek movie universe for Star Fleet cadets where the computer is allowed to cheat so that it always wins

3. REMOVING REDUNDANT LOGIC

Figure 6 illustrates functionally equivalent variants of a single circuit. We will use these variants to discuss the notion of logic based pattern matching and circuit reduction. The circuit in Figure 6 (seen in graphic representation as *a* and *e*) is a 15 input, 6 output circuit that contains three merged, independent components (each with 5 inputs and 2 outputs respectively). Figure 6 also illustrates three semantically equivalent circuit variants (*b*, *c*, and *d*) of *a* produced using the variation process described in Section 2.3. Variant *b* illustrates that some transformation processes will keep the independence of the original components, whereas variants *c* and *d* represent the outcome of experiments where *apparent* merging has occurred.

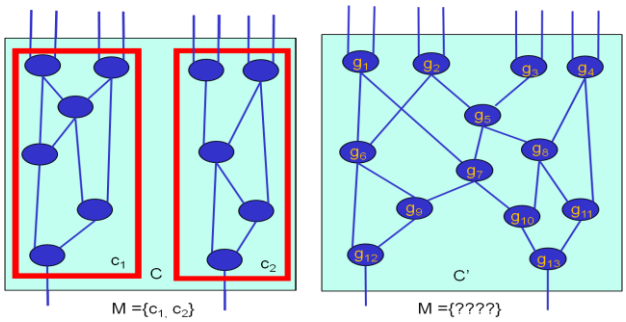


Figure 5. Merged independent circuits: which original component (c_1 or c_2) does any gate g_x belong?

Circuits *e*-*h* in Figure 6 represent the outcome of applying our pattern-based logic reduction techniques to a particular variant. Circuit *e* is the reduced form *a*, which is identical to *a* and cannot be reduced any further. Circuit *f* represents the reduced form of *b*, which still reveals the original component properties of *a*. Circuit *g* represents a reduced form of circuit *c*, which shows the apparent merging reduces to a form where component properties of circuit *a* are revealed. Circuit *h* represents a reduced form of circuit *d*. Circuit *h* represents the interesting case for our consideration: on heuristic minimization, the *apparent* merging of the three circuits remains inseparable.

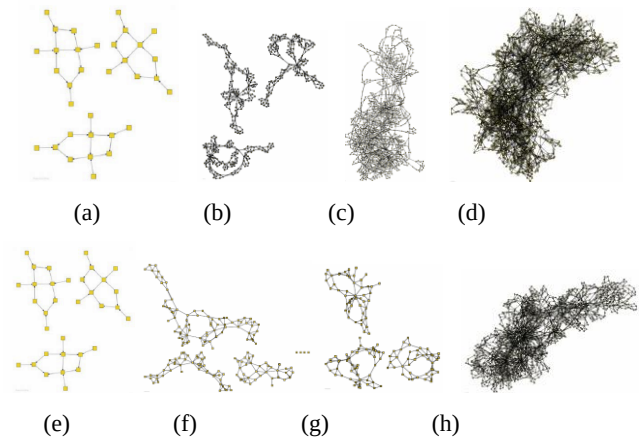


Figure 6. Variants of original multi-component circuit.

These circuits illustrate a rudimentary visual feature that is useful for considering the effects of both circuit variation and circuit reduction. When we examine the graphs of circuit variants, they demonstrate one of two properties in terms of their graph: 1) either the variant circuit graph contains multiple (but larger) sub-graphs, or 2) the variant circuit graph contains a single graph with merged nodes. As a very simple measure, we consider variants that do not exhibit a merged graph harder to analyze or reverse engineer than variants that do exhibit independent sub-graphs (especially if the reverse engineering goal is worst-case component recovery). If our proposed reduction sequences can take a variant with a merged single graph and reduce it to a circuit with independent sub-graphs, we may also conclude that no obfuscation took place (at least in terms of hiding the topology of worst-case components).

How much of this *induced* redundancy between independent components *cannot* be removed and how do we characterize the efficiency of the algorithms that would remove them, assuming no truth-table based synthesis could occur at the circuit level? In Section 3.1 we briefly discuss the techniques we use to minimize the circuit seen in Figure 6 and we give results related to size and levels as a measure of success. From a graph-based perspective, we can also apply a simple *failure* test: if component merging takes place as a result of our variation process, we may use graph analysis to determine whether such merging remains after we apply logic reduction.

3.1 Pattern Identification and Reduction

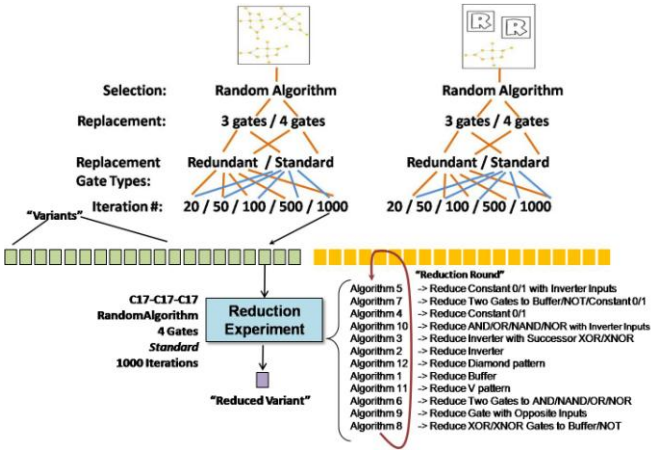
Cryptanalysts use knowledge of a cryptosystem under study to find weaknesses which might compromise the security of the overall encryption scheme. In our case, we assume and also verify empirically that a large majority of semantic-preserving sub-circuit replacements are variations of basic logic laws. For example, the absorption law for logic equivalence states that $p \equiv p \wedge (p \vee q)$. In gate structure form, we can represent the equivalent BENCH syntax as [INPUT p; INPUT q; OUTPUT 2; 1=OR(p,q); 2=AND(p,1)]. Redundant logic pathways such as this are common artifacts of iterative selection/replacement algorithms. Other classic examples of easily reducible logic structures may be seen in white-box circuit structures such as buffers, double inverters, constant 0/1 gates, and so forth. In [9], we detail the work of Kim in creating a logic-reducer based on these common patterns which represent a majority of the variation induced by small selection size strategies (1 or 2 gate selections) in iterative experiments. Functional identification also offers another possibility for reduction where an algorithm may try to find white-box structures that reduce to simple *AND*, *OR*, *NAND*, or *NOR* logic patterns. We use two such functional structures that come from diamond-based and V-based topology patterns. For brevity, we mention here only the major categories and descriptions of reduction rules as follows:

- Reduce Buffer
- Reduce Inverter with Successor XOR / XNOR
- Reduce Constant 0/1 with Inverter Inputs:
- Reduce 2 Gates to Buffer/NOT/Constant 0/1
- Reduce Gate with Opposite Inputs
- Reduce V Pattern
- Reduce Inverter
- Reduce AND/ OR/ NAND/ NOR with Inverter Inputs
- Reduce 2 Gates to AND/NAND/OR/NOR
- Reduce 2 XOR/XNOR to Buffer/NOT
- Reduce Constant 0/1
- Reduce Diamond Pattern

3.2 Reduction Experiments

In order to characterize the overall effect of reduction versus the different approaches with which to create a circuit variant, we examine two different versions of small, independently merged

3-component circuits (as illustrated in Figure 6). Figure 7 gives an overview of a reduction experiment, starting with a circuit variant *C*. We examine two 15 inputs / 6 output circuits (designated C17-C17-C17 and R17'-C17-R17") using 3 different experimental settings. For each experiment, we apply a specific selection, replacement, and generation approach for replacement gates using an increasing number of iterations. For notation purposes, *Standard* and *Redundant* options refer to settings for a specific experiment [9]. They are used to guide the way that replacement circuits are generated. We use *Standard* to indicate that replacement sub-circuits do not contain redundant gates themselves (i.e., buffers, redundant signals, constant ones, or constant zeros). *Redundant* indicates the experiment used replacement circuits that could possibly contain redundant signals and terms themselves.



procedure reductionExperiment(circuit C)
1: $C_0 \leftarrow C$
2: Generate reduction round RR using random permutation // 12 reduction algorithms
3: **for** $rounds = 1$ to MAXROUNDS **do** // MAXROUNDS = 10 reduction rounds
4: **for** $x = 1$ to 12 **do**
5: $C_x \leftarrow RR_x(C_{x-1})$
6: **end for**
7: **end for**
8: $C' \leftarrow C_{12}$
9: Generate log file

Figure 7. Reduction experiment and algorithm description.

Based on this configuration, our experimental set contained 40 circuit variants. For each of the 40 variants, we implement the *reductionExperiment* algorithm and apply a random ordering of the 12 algorithms, where we call one application of one of the 12 algorithms a "reduction round," as seen in Figure 7. Since there are 12! permutations of algorithmic ordering, we decided to use random ordering and assess which orderings produce best and worst case reduction. An experiment consists of one or more reduction rounds, applied over and over again, up to some *MAXROUNDS*. For our experiments we set 10 as the maximum number of rounds and note that we never observe further reductions past 8 rounds in any circuit variant.

3.4 Experimental Results

Our experimental design provides a starting point for objective comparison of variation/obfuscation algorithms and circuit variants. We report subset of results here from [9] of our iterative experiments based on percent of reduction in size as shown in Table 1. Based on this measure, we conclude that the smaller the reduction, the better the resulting obfuscation. Thus, *Standard*(4) produces the better obfuscated variant, based on normalized size. Table 2 gives our best-case reduction results and Table 3 gives our worst-case reduction results, based on the 40 variants under consideration, showing which experimental settings and circuit produced the greatest and least average reduction in size and number of levels (circuit depth). We conducted further analysis of the l000 iteration variants to determine why some of the remaining gates were not removed (all additional gates are by nature by-products of induced redundancy). In short, the apparent affect of overlapping replacements of newly introduced logic connections (which by definition are redundant), creates patterns which are not covered explicitly by our simple 1- and 2-gate patterns. We observe in Table 4, that of the 480 gates remaining in the *Redundant*(4) experimental variant, there are still 83 manually identifiable inverters and 21 manually identifiable constant generators. However, in the 1458 gates of the variant produced with *Standard*(4) sub-circuits, we identify only 18 inverters through manual inspection.

	Experiment Type (gate replacement size)			
Iterations	Standard(3)	Redundant(3)	Standard(4)*	Redundant(4)
20	44.74 %	42.50	**25.00**	55.93
50	58.90	68.49	**21.14**	55.74
100	60.80	78.74	**25.33**	62.88
500	75.80	80.18	**26.35**	75.62
1000	77.41	84.22	**31.56**	73.46

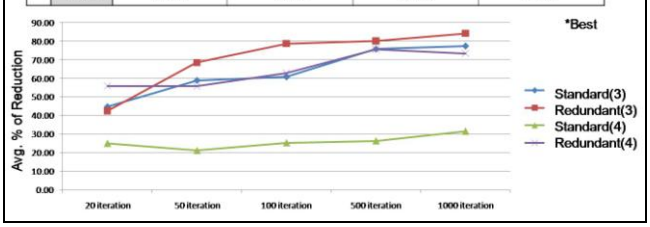


Table 1. Average Reduction in Gate Size (in %)

	Variant (Obfuscated)	Reduced (Avg)	Reduced (Best)	Reduced (Worst)
Gates	1096	173 (84.22%)	158 (85.58%)	185 (83.12%)
Levels	265	40 (84.91%)	35 (86.79%)	41 (84.53%)

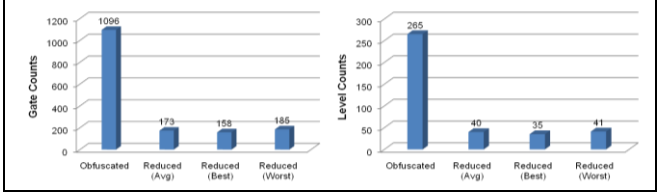


Table 2. Best Case Reduction Result: C17-C17-C17, *Redundant* (3), 1000 iteration

	Obfuscated	Reduced (Avg)	Reduced (Best)	Reduced (Worst)
Gates	2133	1483 (30.47%)	1474 (30.90%)	1495 (29.91%)
Levels	614	426 (30.62%)	425 (30.78%)	428 (30.29%)

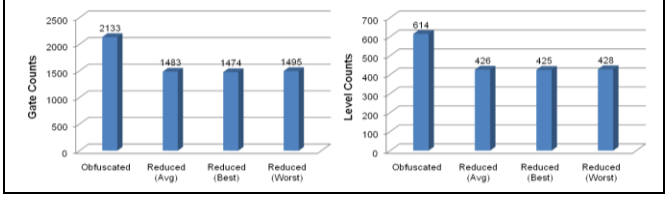


Table 3. Worst Case Reduction Result: R17'-C17-R17", *Standard* (4), 1000 iteration

Given our initial set of circuits and options, we summarize our findings in terms of algorithmic options in the generation process and their apparent effect on our heuristic based reduction. In general, larger gate size replacements (4 vs. 3 gates) affect reduction, which corresponds with the idea that larger replacement options would produce larger and larger numbers of new logic patterns to analyze and categorize. We also surmise (rightly), that circuits which have higher occurrences of redundant patterns (those variants produced under the *Redundant* replacement option) would lend themselves to better reduction rates when basic logic equivalence patterns are in view.

1000-Iterations		Experiment Type (gate replacement size)			
		Standard (3)	Redundant (3)	Standard (4)	Redundant (4)
Gates Remaining	Buffer	0	0	0	0
	Inverter	4	25	18	83
	Constant 0/1	0	19	0	21
	Total	192	129	1458	480

Table 4. Remaining Gate Counts

Finally, iteration count does indeed matter because merging (at least as a by-product of random selection and replacement), only emerges with larger iteration count, i.e., 1000 is better than 20 iterations. Lower iteration counts have higher probability of pattern detection as well, which corresponds to our intuition that small numbers of simple selection/replacements do not provide anything useful in terms of hiding or variation.

3.5 Observable Component Hiding

All 40 variants in our experiment derive from worst-case component hiding circuits that use merging of three originally independent sub-circuits. As Figure 6 illustrates, some variants (i.e., Fig. 6-b) produce no apparent merging while other variants do (i.e., Fig. 6- c and d). Of the variants that produce a unified circuit graph, our logic reduction technique (in some cases) sufficiently removes enough of the redundancies in some cases so that three distinct graphs emerge (i.e., Fig. 6-g). Table 5 summarizes simple graph-based observation of our 40 circuit variants, both before (Table 5-Variant columns) and after applying reduction (Table 5-Reduced columns), and whether or not we observe three distinct graphs (NO) or a single graph (YES).

		Experiment Type (gate replacement size)							
		Standard (3)		Redundant (3)		Standard (4)		Redundant (4)	
		Variant	Reduced	Variant	Reduced	Variant	Reduced	Variant	Reduced
Iterations	20	NO	NO	NO	NO	YES	YES	YES	NO
	50	NO	NO	YES	NO	YES	YES	YES	YES
	100	NO	NO	YES	NO	YES	YES	YES	YES
	500	NO	NO	YES	YES	YES	YES	YES	YES
	1000	NO	NO	YES	YES	YES	YES	YES	YES

Table 5. Report on Graph Variants Merging (YES=merged)

We consider instances where three distinct graphs are observed as outright failures in terms of component hiding. In cases where one graph remains, we also observe that iteration and generation options (*Redundant* or *Standard*) affect whether reduction allows identification of the original three circuit graphs. These results confirm our observations based on gate size and level as presented in Table 4–in particular, higher reduction rates on logic minimization tend to indicate less likelihood that component hiding emerges as a by-product of a random variation process.

4. CASE STUDY EXPERIMENT

Given these results, we note that all 40 circuits in question would *all* fail circuit level logic synthesis analysis because 15 input, 6 output circuits are well within the range of commercial tools and heuristic minimization algorithms to analyze. Consistent with our discussion in Section 2, truth table analysis would clearly reveal the input/output independence of the original three circuits that we merge together to produce our starting case circuits. Therefore, no white-box structural variation (alone) can hide component information for circuits with components that are independent of each other.

Though the results of our logic-based pattern matching experiments give some indication of whether one algorithm may produce better component hiding as an artifact, we desire to know whether component hiding artifacts would be present in more realistic, larger scale circuit examples. For this purpose, we create five artificially "worst-case" circuits that are themselves merged versions of smaller, independent circuits. In some cases, we utilize existing circuits from the ISCAS-85 benchmark set, and in other cases we create our own test cases.

4.1 Experimental Circuit Set

For our case study, we develop a set of five circuits with distinctive characteristics. All circuits have in common the fact that they are composed of independent circuits merged together and sharing common circuit I/O boundary, exhibiting our worst-case scenario in terms of component hiding. For two circuits, we compose the *c432*, *c499*, and *c880* ISCAS-85 benchmark circuits (seen as *c432-c499* and *c432-c880* in Table 6). The merging of *c432* and *c499*, for example, produces a 77-bit input and 39-bit output circuit. The "*ISCAS Merge*" circuit represents a composition of 8 independent circuits together, reaching an I/O space of 362 bits and 192 outputs. We also consider two hard-case scenario circuits that are simply inputs tied to outputs, or in other words, circuits with only buffers connecting each input with its output. Such a circuit duplicates every input on its output bits. We consider a 100-bit buffer circuit (*Buffer-100*) as well as a 500-bit version (*Buffer-500*) for analysis purposes.

The original versions of each circuit in the case study show independent disconnected graphs among their constituent components. They represent larger scale circuit versions which may force adversarial analysis tools (such as commercial reducers) to rely on other than best-case reduction or logic synthesis. Our interest in the case study analysis remains whether pattern-based logic reduction holds promise when compared to commercially available tools and whether either form of analysis breaks down with larger I/O space circuits.

4.2 Experiment Setup

For our synthesis analysis, we use Quartus II®, a software tool developed by the Altera Corporation® for the design and synthesis of circuit designs for FPGAs and other programmable devices. Using Quartus II®, we were able to compare pattern-based reduction with commercial optimization in several ways. In terms of producing variants for analysis, our case study comparison only considers a *simple* or *complex* version of the generation algorithm, as seen in Table 6. For the *simple* algorithm approach, we use only a 2-gate random selection and 3-gate fully random replacement. We run some number of iterations that range from

3000-10000. For the *complex* algorithm approach, we utilize a variety of selection algorithms that pick 1, 2, 3, or 4 gates and replace them with a random choice, up to 2 to 3 gate sizes larger. Since not every selection has a valid replacement for the given size requested, we also use goal-based measures to guarantee a minimum number of replacements. Our case study used anywhere from 3000-5000 guaranteed replacements in the complex approach.

For minimization and analysis of the variants, we use two approaches. First, we convert circuit variants into an appropriate VHDL form and use Quartus II® to analyze them. Second, we perform pattern-based reduction as described in Section 3. Using the synthesis tool, we employ the technology viewer mode to see how a circuit NETLIST is synthesized for an FPGA. Such analysis provides us insight as to whether the tool associates inputs with outputs and whether or not inputs from different components drive outputs in other components. The tool, for example, rendered as three independent sub-circuits (based on input and output correlation or pin groupings) and mapped to specific independent sets of logic units.

4.3 Case Study Results

To summarize the results of our study, we present in Table 6 the percentage reduction in size and levels, based on the original, of our test circuit variants (both simple and complex) based on our pattern-matching algorithm. We also list the graph-based analysis view of each original and their variants, based on how many distinct, independent components are visible.

	c432-c499			c432-c880			ISCAS Merge			Buffer-100			Buffer-500		
Variant Algorithm	O	S	C	O	S	C	O	S	C	O	S	C	O	S	C
Pattern Based Reduction Size/Levels	-	85%	21-29%	-	63%	22%	-	16-18%	9%	-	90%	28%	-	89%	26%
	-	89%	24-36%	-	72%	24%	-	70%	23%	-	93%	29%	-	92%	28%
Independent Components (pattern-based reduction)	2	2	1	2	2	1	8	1	1	100	59	15	500	253	109
Logic Cells (Quartus II)	133	155	165	173	184	185	1600	1685	nn	0	0	0	xx	xx	xx
Independent Components (as realized by Quartus II)	2	2	2	2	2	2	nn	nn	nn	100	100	100	xx	xx	xx
						O – original circuit S – Simple C - Complex				nn– not tested xx – too big based on I/O					

Table 6. Case Study Summary Results

Table 6 illustrates that, at least for simple-algorithm variants, our logic-based pattern matching reduction performs comparably to the Quartus II® tool. We note that the *ISCAS Merge* circuit proved the most difficult to analyze for either tool. In the case of 500-buffer circuit, the tool did not have the I/O pin capability that could handle the circuit, but our analyzer provided modest reduction in the simple-algorithm variant. The number of logic cells reported by the tool also represents around 6-8 gates of realized circuitry as each one implements a configurable 16-bit look up table. We note in summary that the pattern-based reduction meets our expectations considering results indicated by circuits under study in Section 3. Namely, variants produced by simple selection algorithms (2 gate selections) reduce at higher rates than variants created with more complex pattern capability.

5. CONCLUSIONS

In this paper, we report experimental results of using logic-based pattern matching algorithms as a red-team analysis tool for circuit variants intended for obfuscating or protecting intellectual property. We show the efficacy of pattern-based matching algorithms and their effect on circuit variants, providing a concrete approach to compare and understand different generation algorithms. Much future work remains to refine our logic-based pattern matching algorithm to be more competitive with currently available commercial synthesis tools. We note, in conclusion, that none of our experimental options for producing the circuit variants were deterministically geared to produce component hiding. We only consider random variation with some small degree of determinism in how the experimental generation proceeds. We demonstrate, at least empirically, that hiding properties such as component hiding, may be achievable as attributes of variation processes that exploit maximum randomness in the choices of the circuit generator. Future work will focus on additional aspects of hiding, especially when deterministic means are integrated into the generation process.

6. ACKNOWLEDGEMENTS

This material is based upon work supported in part by the U.S. Air Force Office of Scientific Research under grant number F1ATA09048G001. The views expressed in this article are those of the authors and do not reflect the official policy or position of the Unites States Air Force, Department of Defense, or the U.S. Government.

7. REFERENCES

[1] S. Goldwasser and G. Rothblum, "On best-possible obfuscation," *LNCS, Vol. 4392, TCC 2007*, Springer, (21-24 Feb 2007), 194–213.

[2] B. Barak, O. Goldreich, et al. "On the(im)possibility of obfuscating programs," *Elec. Coll. on Computational Complexity*, 8, 2001.

[3] C. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation: tools for software protection," *IEEE Trans. Softw. Eng.*, 28(8):735–746, 2002.

[4] F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits," *Proc. IEEE Intl Symp. Circuits and Systems*, IEEE Press. Piscataway, N.J (1985) 695–698.

[5] J. T. McDonald, Y. Kim, and A. Yasinsac. Software issues in digital forensics. *ACM Operating Systems Review*, 42(3), April 2008.

[6] E. Chikofsky and J. H. Cross II. "Reverse Engineering and Design Recovery: A Taxonomy," *Crosstalk*, January 1990.

[7] M. Hansen, H. Yalcin, and J. Hayes, "Unveiling the ISCAS-85 benchmarks: a case study in reverse engineering," *Design & Test of Computers*, IEEE, 16(3):72–80, 1999.

[8] C. Alpert and A. Kahng, "Recent directions in netlist partitioning:A survey," *Integration: The VLSI Journal*, 19:1-81, 1995.

[9] H. Kim, "Removing Redundant Logic Pathways in Polymorphic Circuits," *Master's Thesis*, A.F. Institute of Tech., March 2009.

[10] Y. Kim and J.T. McDonald, "Considering Software Protection for Embedded Systems." *CrossTalk: The Journal of Defense Software Engineers*, 4-8, Sept/Oct 2009.

[11] J. White, A. Wojcik, M. Chung and T. Doom, "Candidate subcircuits for Functional Module Identification in Logic Circuits." *GLSVLSI '00, Proceedings of the 10 th Great Lakes Symposium on VLSI*, ACM SIGDA, 34-38, 2000.