Component Hiding Using Identification and Boundary Blurring Techniques

THESIS

James D. Parham Jr., First Lieutenant, USAF

AFIT/GE/ENG/10-22

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GE/ENG/10-22

Component Hiding Using Identification and Boundary Blurring Techniques

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering

James D. Parham Jr., B.S.E.E.

First Lieutenant, USAF

March 2010

AFIT/GE/ENG/10-22

Component Hiding Using Identification and Boundary Blurring Techniques

James D. Parham Jr., B.S.E.E.
First Lieutenant, USAF

Approved:

LtCol J. Todd McDonald, PhD (Chairman)	9 MAR 10 date
Dr. Yong C. Kim (Member)	9 MAR 2010 date
Dr. Michael R. Grimaila (Member)	9 MAR 2010 date

Abstract

Protecting software from adversarial attacks is extremely important for DoD technologies. When systems are compromised, the possibility exists for recovery costing millions of dollars and countless labor hours. Circuits implemented on embedded systems utilizing FPGA technology are the result of downloading software for instantiating circuits with specific functions or components. We consider the problem of component hiding a form of software protection. Component identification is a well studied problem. However, we use component identification as a metric for driving the cost of reverse engineering to an unreasonable level.

We contribute to protection of software and circuitry by implementing a Java based component identification tool. With this tool, we can characterize time required for carrying out adversarial attacks on unaltered boolean circuitry. To counter component identification methods we utilize boundary blurring techniques which are either semantic preserving or semantic changing in order to prevent component identification methods. Furthermore, we will show these techniques can drive adversarial cost to unreasonable levels preventing compromise of critical systems.

Acknowledgements

First and foremost, I owe a large debt of gratitude to my wife for her encouragement and support over the last 16 months. I especially want to thank you for enduring the long hours away from our family. You have been a magnificent part of my academic achievements and I could not have done any of this without you.

I also owe a large debt of gratitude to my thesis advisor, Lt Col Todd McDonald, for his patience, availability and enthusiasm. I have become interested in topics and made decisions that I would never have otherwise. To the remainder of my committee, Dr. Yong Kim and Dr. Michael Grimaila, thank you for your suggestions and advice. You helped me overcome obstacles and get back on track more times than you know. Thank You.

James D. Parham Jr.

Table of Contents

Page

List of Figures

List of Tables

List of Symbols

Symbol		Page
P	Program or Program Circuit	7
Ω	Circuit Basis	8
G	Graph	8

List of Abbreviations

Abbreviation		Page
RFID	Radio Frequency Identification	2
CLA	Carry Look Ahead	6
IC	Integrated Circuit	6
FPGA	Field Programmable Gate Array	7
DAG	Directed Acyclic Graph	8
CORGI	Circuit Obfuscation via Randomization of Graphs Iteratively	10
ISCAS	International Symposium on Circuits and Systems	17
SEC	Single Error Correcting	18
ALU	Arithmetic Logic Unit	18
DED	Double Error Detecting	18
COS	Circuit Obfuscation System	22
CIS	Component Identification System	22
CUT	Component Under Test	25
SOP	Sum Of Products	40
PET	Program Encryption Toolkit	111

Component Hiding Using Identification and Boundary Blurring Techniques

I. Introduction

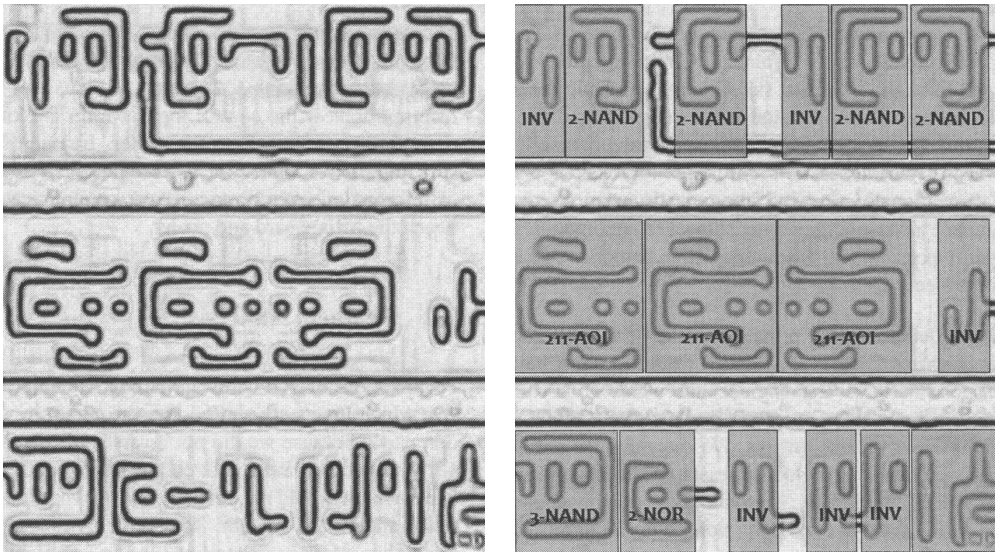
Today's military systems are complex systems composed of critical hardware and software components. The DoD must take steps ensuring these technologies are not easily compromised by an adversary's attempt to recover the function, construction or other embedded information contained within. Top down design results in the abstraction of large circuits, or functions, into smaller subcircuits referred to as components. If an adversary identifies these components, it may lead them to the overall circuit function [4]. Hiding components in a manner which drives up, to an unreasonable level, the time and effort required for component identification is one method possible for protecting critical U.S. military systems.

1.1 Problem Definition

The task of reverse engineering a system or product is not always completed with malicious intent. When documentation used for sustaining large and complex military systems becomes lost or damaged, reverse engineering becomes an acceptable means for discovering the composition of the older system. However, this is not always a reverse engineers' intent. In 2008, the Fiscal Year 2009 budget requested $183.8 billion in modernization to meet future threats. This total included both procurement, as well as research and development [11]. When an adversary reduces this cost by reverse engineering a system, implementing protective measures becomes extremely important.

Component identification is one method used by reverse engineers for discovering key elements of a circuit. Nohl et al. used this technique revealing the cryptographic

cypher in the Mifare Classic Radio Frequency Identification (RFID) tag. They physically exposed the transistors in each layer of its silicon chip and grouped them into gates performing logical functions such as the AND or XOR function. With computer tools, identification of circuit gates was accomplished [9].



(a) Before automated template detection. (b) After automated template detection.

Figure 1.1: The results of exposing each layer of the silicon chip for identifying gate level structure [9].

Because knowing gate level structure alone is not feasible for discovering a circuit's function, they began focusing their attention on grouping gates into identifiable cryptographic components. Knowing the circuitry contains a 48-bit register and a number of XOR gates made locating the area of the chip containing the cryptographic components possible [9]. Next, discovery of gate connections was accomplished using additional computer tools. Now, from the map of logic gates and connections between them almost all needed information for discovering the cryptographic algorithm is known [9].

The Dutch government invested about $2 billion in the use of this RFID technology for public transit system ticketing [13]. Discovering the cryptographic keys makes it possible for attackers to access the transit system at no cost. Implementa-

tion of hardware and software protection is extremely important for preventing these types of attacks on U.S. critical technologies.

1.2 Goals and Hypothesis

The Mifare RFID example demonstrates how use of computer tools is important to the component identification process. The student team identified logic gates and connections between them from the silicon chip's discovered transistors using various computer tools. This provided a complete picture of the internal circuitry, or white box construction. Having knowledge of what components were necessary for the cryptographic cipher allowed them to isolate the physical location of the cryptographic circuitry. What happens when a reverse engineer has knowledge of a circuit's gate level structure, but has no idea what its function is? How would an engineer go about identifying internal components that compose a circuit? The first research goal is implementing a component identification tool for identifying known components of a larger circuit. We will use this tool for performing an analysis on the effectiveness of current component hiding techniques and gain insight in to the difficulty an adversary may have when attempting to discover circuit components in an unknown circuit. Can this tool or similar tools be defeated by performing circuit transformations? What can be done to hide components in plain sight? The second research goal is determining a suitable transformation defeating this and possibly similar identification tools. To answer these questions we use various combinational logic benchmark circuits. With these tools and techniques in place we can increase circuit and software protection levels preventing adversaries from gaining access to U.S. critical technologies.

1.3 Organization

The remainder of this thesis is organized as follows. Chapter II discusses current methods and techniques used for discovering circuit components and ultimately an unknown circuit function. Chapter III covers the methodology and analysis used for this research. Chapter IV details the results of the identification tool implementation

and component hiding algorithms. Chapter V summarizes the contributions of this research and future works for improving the results. Appendix A provides a logic gate legend for graphs used throughout this thesis. Appendix B provides additional details of the module library and custom benchmark circuits used for the component identification tool. Appendix C contains identification time analysis figures. Appendix D details creation of the custom benchmark circuits. Appendix E shows UML diagrams of Java classes created during this research and Appendix F provides a table of circuits contained in the current module library.

II. Literature Review

This chapter summarizes background material relevant to our research. Readers familiar with reverse engineering and obfuscation principles may omit this chapter from their reading. We organize this chapter in the following manner. Section 2.1 covers topics in reverse engineering. Section 2.2 provides a brief introduction to obfuscation principles. Section 2.3 covers boolean circuits representing software programs and additional information on our use of boolean circuits. Section 2.4 introduces CORGI, a circuit obfuscation engine, and techniques used within. Section 2.5 covers techniques of component identification and Section 2.6 covers the ISCAS-85 benchmark circuits.

2.1 Reverse Engineering

Reverse engineering is a systematic approach to gaining a basic understanding of hardware or software systems and its structure when you have nothing more than the system itself [12]. There are different reasons for reverse engineering, of which one is creating the necessary documentation needed to aid in maintenance, strengthen enhancement, or support replacement [1]. Determining how a product functions so knowledge of the original systems purpose is gained is a second reason. In some cases a device may contain partial cryptographic keys or algorithms for decoding and decrypting information. Protecting these algorithms and keys prevents illegal access to these pieces of information [2]. Reverse engineers perform different types of analysis depending on the knowledge of their circuit of interest, either black box or white box.

2.1.1 Black Box Analysis. Black box analysis is a reverse engineering technique where nothing is known about the system of interest except the number of circuit inputs and outputs, referred to as I/O space. In the case of logic circuits, identifying the overall function requires enumerating all input combinations and measuring the circuit output. This is an unavoidable step when dealing with low level logic circuits consisting of truly random logic [4]. This is possible for small input size. For example, circuit c17 from Figure 2.6 has only five inputs. To enumerate all

possible combinations would require only $2^5 = 32$ different combinations. However, for large circuit inputs this is a intractable task. For example, a simple 64-bit adder with a carry-in pin has a total of 129 input pins and 65 output pins. If a reverse engineer, with no prior knowledge of the circuit applies the inputs, it would take 2^{129} attempts or 2^{99} seconds, roughly 2×10^{22} years, using a state-of-the-art one gigahertz tester costing over $1 million [8].

2.1.2 White Box Analysis. When a reverse engineer knows the "construction", or internal structure, of a system the engineer performs white box analysis. In software, a program's construction is the collection of language statements that define its topography [6]. With a circuit, this is in the form of a net list or other descriptive file or the actual circuit schematic. Having this information available, the reverse engineer has a combination of techniques they can employ. The two techniques this research focuses on are the identification of library and repeated circuit modules or components. Common components such as multiplexors, decoders, adders, and CLA generators are found in IC manufacturer's databooks or cell libraries and in textbooks [4]. Using these databooks identifying components of a circuit can lead the engineer to discover the circuit's function. Repeated modules can also give away information about a circuit. The c6288 16-bit multiplier is made up of 240 adder components. Discovering a circuit's functions becomes more efficient for a reverse engineer when regularity of the circuit structure is identified and exploited.

Hiding components in both a random and deterministic manner may force an adversary to perform black box analysis which is impossible in instances discussed in Section 2.1.1. Other effects of these techniques are introducing additional inputs, outputs or both to a circuit changing its black box structure. Increasing the input space will increase the time necessary to enumerate all input combinations therefore, increase an adversary's time for discovering a circuit's function.

2.2 Obfuscation

One defense against reverse engineering is obfuscation, a process that renders software unintelligible but still functional. We accomplish this by transforming a program into an equivalent variant harder to reverse engineer. Collberg states, given a set of obfuscating transformation $T = \{T_1, \ldots, T_n\}$ and a program P you can find a new program $P' = T_i(P)$ such that [2]:

- P' is sematic preserving.
- Obscurity of P' is maximized, i.e., the time to reverse engineer the program increases.
- Resilience of P' is maximized, i.e., it will be difficult to create a tool to undo the transformation or extremely time consuming.
- Stealth is maximized, i.e., the statistical properties of P' or similar to those of the original program P.
- The time/space cost to create P' is minimized.

We apply obfuscation, using various techniques which we discuss later, to our circuits of interest. Generally, we apply obfuscation transformation techniques in a way in which the circuit semantics are left unchanged, white box obfuscation. However, if we allow transformations to change the black box structure of the circuit and use a recovery function to maintain the intended output, other possibilities exist [6]. Black box transformations are an area of focus for this research effort.

2.3 Boolean Circuits Representing Programs

Today since more and more programs are distributed in easily decompilable formats, rather than native binary code, it is increasingly important for us to consider ways for protecting our programs. An example of a decompilable program is the Java class file format [2]. Other programs we consider are those executed on hardware processors such as Field Programmable Gate Arrays (FPGA). Embedded systems

using FPGAs are able to realize circuits consisting of many different components. Some of these include logic gates, controllers and arithmetic logic [8]. When studying techniques for protecting software systems we need a method of expressing these programs in a modifiable form. We accomplish this through expressing programmatic logic as combinatorial boolean circuits [10]. Since FPGAs realize boolean circuits expressed by software programs, the terms software protection and circuit protection are used interchangeably for the remainder of this thesis. Circuit protection is an easier task than software protection. However, it still remains an extremely difficult task.

2.3.1 Circuit Definition. A circuit is defined by the number of inputs, outputs, intermediate gates and circuit basis. The circuit basis, Ω, is a set of gates a circuit may be composed of. For example, the set $\Omega = \{$AND, OR, NOR, NAND, XOR, NXOR$\}$, having a basis size $|\Omega| = 6$, is a complete six gate basis [6]. The circuits utilized in this research use the basis set $\Omega = \{$AND, OR, NOR, NAND, XOR, NXOR, NOT, BUFFER$\}$ with a size $|\Omega| = 8$. A 5-2-6 circuit is a circuit with five inputs, two outputs and six intermediate gates. It is important to note intermediate gates include output gates but not inputs in their total. Appendix A provides a gate legend for referencing when viewing circuit graphs displayed in this thesis.

2.3.2 Modeling a Boolean Circuit. A combinational logic circuit modeled as a graph is easily analyzed. Since a combinational circuit has no loops and signal flows from input to output, representing a circuit by Directed Acyclic Graph (DAG) is best. We now detail basic theory behind this modeling approach.

2.3.2.1 Basic Graph Theory. A graph G is a triple consisting of a vertex set $V(G)$, and an edge set $E(G)$ and a relation that represents each edge with its endpoints [14]. A vertex $v \in V(G)$ represents a gate or an input of the circuit and an edge $e \in E(G)$ represents a wire of the circuit. For an edge or wire to be

directed means flow can occur in only one direction. This is shown by a function assigning each edge an ordered pair. The first vertex in the ordered pair is the tail and the second is the head. It is said that an edge goes from its tail to its head. This terminology is taken from the arrows used to draw these kinds of graphs [14]. An edge can also be indicated by listing the two vertices the edge connects for example, the term uv simply means the edge between vertex u and vertex v. Arrows are omitted from the circuit graphs, in this thesis, since all modeled circuits are combinational logic circuits.

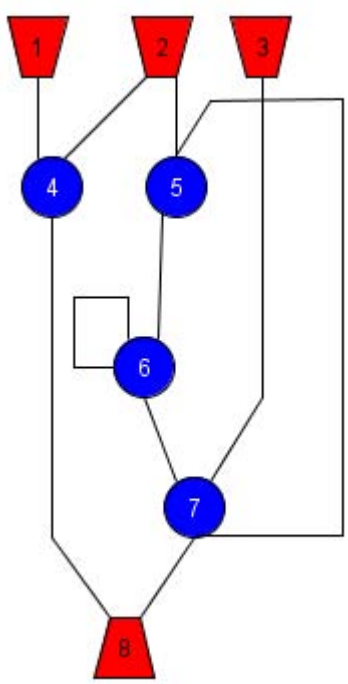


Figure 2.1: An example graph used to illustrate basic graph theory principles.

A path is a list of adjacent vertices that occur in order. Tracing the circuit flow from one vertex to another would reveal a path between two vertices. A loop in a graph is a single edge whose endpoints are the same. In the case of a logic gate, a loop existing defines a wire from the output of a gate to the input of the same gate. A cycle is a path beginning and ending at the same vertex and is different from a loop since more than one edge defines a path.

The graph in Figure 2.1 demonstrates basic graph theory principles. A path p exists between vertex two and vertex eight and is defined by the list {2,5,6,7,8}. This is valid in the combinational circuit graph. Vertex six has a loop and there also exists a cycle defined by {5,6,7,5}. For a combinational circuit both loops and cycles are illegal states. For this reason a DAG is used for modeling the combinational logic circuit.

Definition - *The **neighborhood** of a subgraph G, $N(G)$, consists of all vertices v adjacent to the $v \in G$, but not contained in G [14].*

Definition - *The **outdegree** of a vertex v, is the number of edges with the tail v [14].*

Definition - *The **indegree** of a vertex v, is the number of edges with the head v [14].*

2.4 Circuit Obfuscation via Randomization of Graphs Iteratively - CORGI

CORGI is a white box circuit obfuscation tool which takes a circuit and produces a semantically equivalent variant, meaning the circuit still has the same number of inputs and outputs and performs the same logic function but has a different white box structure [6]. In most cases CORGI produces a variant with an increased number of gates. CORGI accomplishes this by selecting circuit gates based on various selection strategies. These strategies depend on circuit division into levels. For CORGI, levels begin at the output with level zero and increase in value towards the input. A predecessor of a gate at level n will belong to level $n+1$. Table 2.1 shows CORGI gate selection strategies. Once a set of gates is selected, the subcircuit is defined (induced) by the set of selected gates, as well as all connections ("wires") leading into or out of those gates [6].

Input size, output size, and gate size defines a subcircuit selected by the selection algorithm. A boolean six-tuple of circuit generation options is necessary for the library generation algorithm. Each of these options, shown in table 2.2, influences the number of circuits produced for the selection library. After the selection library is generated, a random choice is made and the subcircuit is replaced with a semantically equivalent

Table 2.1: CORGI selection strategies [10].

Selection Algorithm	Description
RandomSingleGate	Selects a single gate at random
RandomTwoGates	Selects two gates at random
RandomLevelTwoGates	Selects a hierarchial level at random, and limits replacement to two gates selected at random from that level ($\pm$ 1 level)
FixedLevelTwoGates	Same as RandomLevelTwoGates except the hierarchial level is specified
LargetLevelTwoGates	Same as FixedLevelTwoGates except the hierarchial level is the one containing the most gates
OutputLevelTwoGates	Same as FixedLevelTwoGates except the hierarchial level is 0

circuit. This process is repeated for a predetermined number of iterations producing a final circuit variant equivalent to the original circuit.

Table 2.2: Subcircuit replacement options utilized by CORGI [6].

Option	Description
SymmetricGates	Should a gate with input (X_1, X_2) be considered equivalent to a gate with input (X_2, X_1)
RedundantGates	Can a circuit have two such that the truth table for each gate, based on all circuit inputs, be the same
AllowConstants	Should a circuit have immediate access to the constants true and false
DoubleInputs	Should the inputs to a gate be allowed to originate in the same place
ExactCount	Will the circuits being generated have an upper bound on its size or will the circuit be exactly the selected size
SimpleOutput	When this is set to true, circuit outputs must all be at the lowest level sinks in the circuit graph

2.4.1 Circuit Normalization. Redundant logic pathways are introduced by the variant producing algorithms [7]. Removal of redundant signals and producing a minimal sized semantically equivalent circuit is the purpose of the normalization process. [7] developed a set of circuit normalization algorithms used during our ex-

perimentation process. These algorithms aid in final circuit variant production. Kim implemented the following reduction patterns:

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

In the case of an obfuscated circuit, normalization may be one of the first steps an adversary takes to reduce the problem of reversing a circuit. By including this as part of techniques introduced in this research, minimized circuit variants are produced and analyzed allowing for a higher confidence in the inability of reverse engineering circuit variants.

2.5 Identifying Circuit Components

Circuit white box analysis can lead to a circuit's overall function. As discussed in section 2.1.2, determining components of a circuit can expedite the process of understanding a circuit's overall function. Analyzing a circuit for candidate subcircuits and then comparing each candidate against a library of known modules is the basis of our component identification tool.

The problem of enumerating a circuit for candidate subcircuits is enormous. After modeling the circuit of interest as a DAG, all candidate subgraphs are enumerated. In a fully connected graph there is a possibility of $n!$ subgraphs, where n is the number of vertices in the graph. A completely connected combinational circuit is

highly unlikely, but from this number it is possible to see an intractable number of possible subcircuits may exist in a circuit. White et al. developed a candidate subcircuit enumeration algorithm that provides all the interesting gate clusters within a target circuit [16]. This algorithm functions on three basic rules:

1. **Unique Enumeration** - No subgraph is created more than once.
2. **Fully Specified Vertices** - All vertices in a subgraph must be fully specified or the subgraph is discarded.
3. **Contained Vertices** - All vertices in a subgraph must be contained or the subgraph is discarded.

These rules are discussed in greater detail in Sections 2.5.1 and 2.5.2.

2.5.1 Unique Enumeration. Using an effective and efficient technique of unique enumeration when identifying components is necessary because the size of possible components in a circuit is extremely large. During component identification, the first step is ensuring only one creation path for each component is utilized [15]. The order vertices are added to a subgraph is its creation path. This path only allows the addition of neighboring vertices with an index less than the subgraph index, the index of the first vertex in the path. These vertices compose the subgraph frontier.

Definition - *The* ***frontier*** *of a subgraph G, $F(G)$, consists of all vertices v such that $v \in N(G)$ and the index of v is less than the index of the first vertex in the creation path* [15].

The frontier contains expansion vertices. These vertices compose the reachable frontier and are a subset of the frontier. In the trivial case of a single vertex the frontier and reachable frontier are the same. Once the expansion of a graph begins the two are no longer equivalent.

Definition - *The* ***reachable frontier*** *of a subgraph G, $F^R(G_n)$, consists of all vertices v that may be added to G such that $v \in F(G_n)$ and either ($v \notin F(G_{n-1})$) or ($v \in F^R(G_{n-1})$ and the index of $v <$ the index last added to the creation path)* [15].

Combining these together ensures a circuit graph is enumerated in such a way every creation path is unique. Adding any vertex to a subgraph in its reachable frontier satisfies rule one of the subcircuit enumeration algorithm.

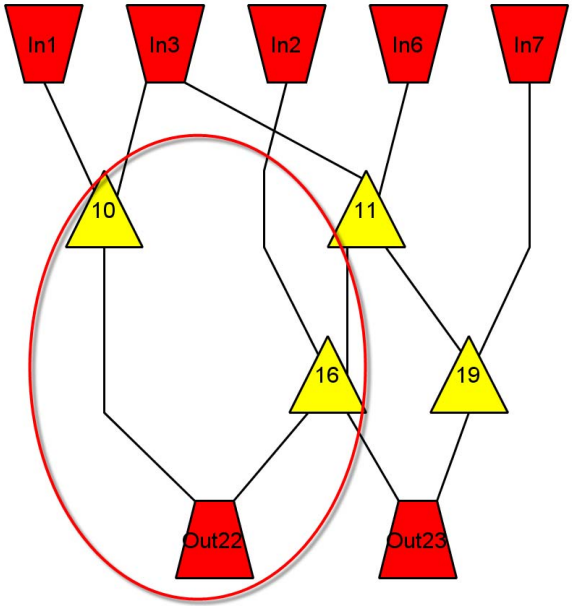


Figure 2.2: Example of frontier and reachable frontier in the c17 circuit graph.

In Figure 2.2 the subgraph, H, with creation path {Out22,16,10} is circled in red. The $N(H) = \{Out23, 11, In2, In3, In1\}$. Because the highest index of the subgraph is Out22, the vertex Out23 is excluded from the frontier of the subgraph. Vertex 11 is excluded from the reachable frontier because its index is greater than the last vertex in the creation path. The remaining vertices, {In1,In2,In3} are in the reachable frontier.

2.5.2 Focused Enumeration. Even though a subgraph, H, is unique it is not always an acceptable candidate subcircuit. An acceptable subcircuit meets two conditions. Each $v \in H$ is fully specified and contained. A fully specified vertex has either all of its predecessors or none of its predecessors in the subgraph. A contained vertex is fully specified and joined within the subgraph by either all of its successors or none of its successors. Only when these two conditions are met is a subgraph considered a candidate subcircuit [15].

Definition - *In a subgraph H of a graph G, a vertex $v \in V(H)$ is a* ***fully specified vertex*** *if* $(\forall u | u \in V(G) \wedge uv \in E(G) \rightarrow u \in V(H)) \vee (\forall u | u \in V(G) \wedge uv \in E(G) \rightarrow u \notin V(H))$ [15].

Definition - *In a subgraph H of a graph G, a vertex $v \in V(H)$ is a* ***contained vertex*** *if v is fully specified and* $((\forall u | u \in V(G) \wedge uv \in E(G) \rightarrow u \in V(H)) \vee (\forall u | u \in V(G) \wedge uv \in E(G) \rightarrow u \notin V(H)))$ [15].

Rule two and rule three ensure these conditions are met. Rule two ensures every $v \in V(H)$ is fully specified. When a vertex is added to a subgraph it is checked to ensure this condition is satisfied. If it is not, and if the vertex required to satisfy this condition is in $F^R(H)$, it is immediately added. If its not possible to add a vertex, the subgraph is discarded as a candidate. Rule three ensures the circuit is contained. Just as in rule two, as a vertex is added it is checked for containment. If any vertices are needed and they are in $F^R(H)$, they are added immediately. Rule two and three occur in a recursive manner until either a subgraph is created or discarded.

2.5.3 Series Versus Parallel Components. When the I/O space of a circuit is within the realm of an adversary, there is a distinct difference in the difficulty of hiding components in series versus components in parallel. Using black box techniques, namely I/O enumeration, an adversary isolates the outputs that are driven by particular inputs. This "leaks" information about the number of possible components in a circuit and also helps create the gate clusters for further analysis. Discovering series components is more difficult because I/O behavior does not reveal this relationship. For example if the circuit in Figure 2.3 is a 6-4-25 circuit, an adversary, in just 64 steps, will know at least two components compose the circuit. The dependency of component D becomes known and also components A, B, and C are considered a single component. A permutation circuit is one method for preventing discovery of circuit information.

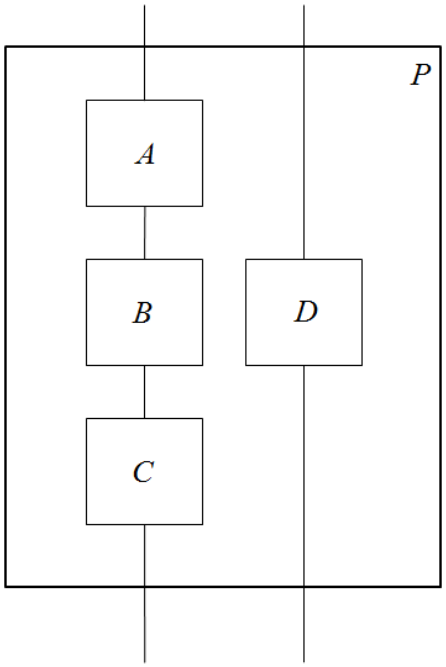


Figure 2.3: The four basic component cases in a circuit.

2.5.4 Permutation Circuit. The permutation circuit creates a one-to-one and onto relationship between the input and output of the circuit. This type of circuit can encrypt circuit output containing parallel components in such a way that adversarial black box analysis does not give away information regarding circuit component structure. [5] shows parallel components sharing a circuit boundary are easily identified using black box analysis. Figure 2.4 shows two parallel components in a

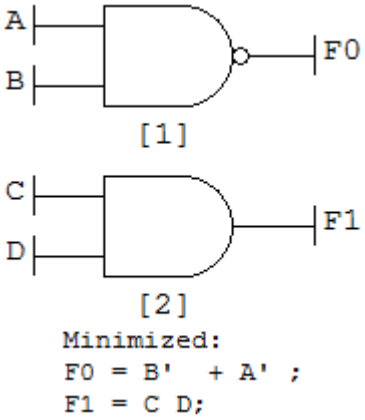


Figure 2.4: Two parallel circuit components, an AND gate and NAND gate.

circuit. The minimized circuit logic equations reveal this circuit is composed of two individual components and shows the dependent inputs of each circuit output. Figure

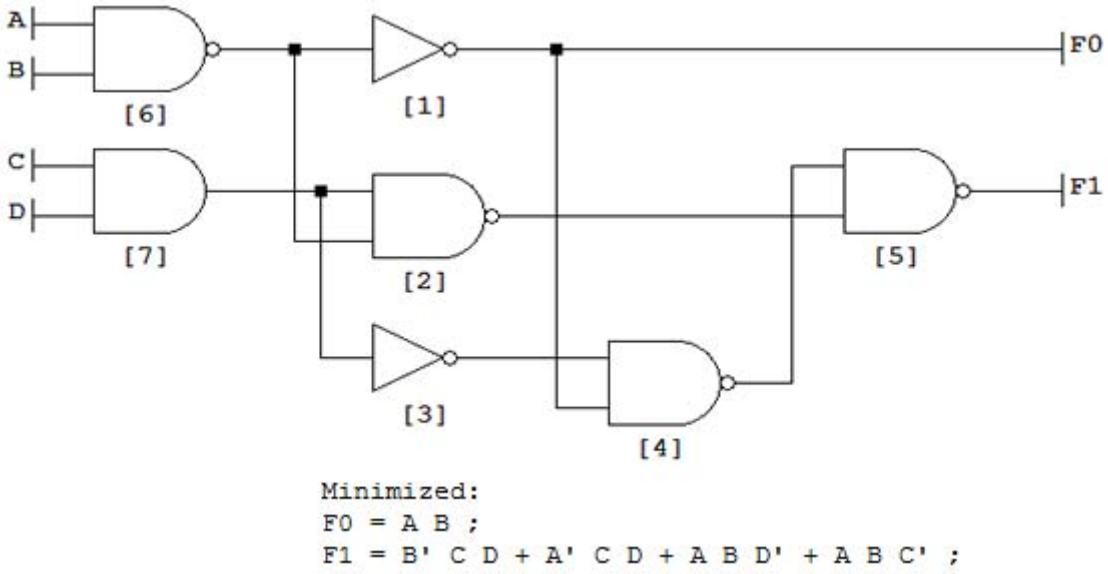


Figure 2.5: Two parallel circuit components with 2-bit permutation circuit connected to output.

2.5 shows results after attaching a two bit permutation circuit. Since F_1 depends on all four circuit inputs, the two components are not revealed by black box analysis. When using permutation circuits the original circuit function is changed. However, recovery is easy with a decryption circuit reversing the permutation.

2.6 ISCAS-85 Benchmarks

The ISCAS-85 benchmark circuits are industrial circuit designs. When originally introduced, only the netlist for these designs was made available both for confidentiality reasons and to allow them to be viewed as random logic circuits [4]. Hansen et al. used these circuits to research high-level generation techniques and found the ISCAS-85 circuits had well defined high level structures based on common logic circuit building blocks. Their reverse engineering techniques allowed them to recover the functionality of the benchmark circuits and create high-level models [4]. These models are now available to other researchers and are part of the foundation for our

variant producing algorithms. Table 2.3 lists ten benchmark circuits and their basic description.

Table 2.3: Circuits contained in the ISCAS-85 benchmark suite [4].

Circuit	Function	Inputs	Outputs	Gates
c432	27-channel interrupt controller	36	7	160
c499	32-bit SEC circuit	41	32	202
c880	8-bit ALU	60	26	383
c1355	32-bit SEC Circuit	41	32	546
c1908	16-bit SEC/DED circuit	33	25	880
c2670	12-bit ALU and controller	233	140	1,193
c3540	8-bit ALU	50	22	1,669
c5315	9-bit ALU	178	123	2,307
c6288	16-bit multiplier	32	32	2,406
c7552	32-but adder/comparator	207	108	3,512

2.6.1 BENCH File. For our research we primarily use the BENCH file for textually representing boolean circuits. The c17 circuit is a simple 5-2-6 test circuit. Figure 2.6(a) shows the BENCH file representation of c17.

The BENCH file provides an overview of the circuit. It shows the number of inputs, outputs, inverters, and total number of gates contained in the circuit. A synopsis of the gate types follows the total number of gates. Next, inputs and outputs are listed with their corresponding gate identification number. Finally, all gates in the circuit are listed along with their input gate identification numbers. Circuit objects are constructed in CORGI while parsing the BENCH file. For example, NAND(10) has input from INPUT(1) and INPUT(3) while OUTPUT(22) is a NAND gate receiving input from NAND(10) and NAND(16).

2.6.2 c6288 - An ISCAS-85 Benchmark Circuit of Particular Interest. The c6288 benchmark, 16-bit multiplier, represents a much larger gate-level circuit. Figure 2.7 shows how 2160 gates form 240 full and half adder cells, each with nine gates, is arranged in a 15x16 matrix [3]. This circuit is of interest because of the number of repeated modules, as discussed in section 2.1.2. Figure 2.8 shows the topology of

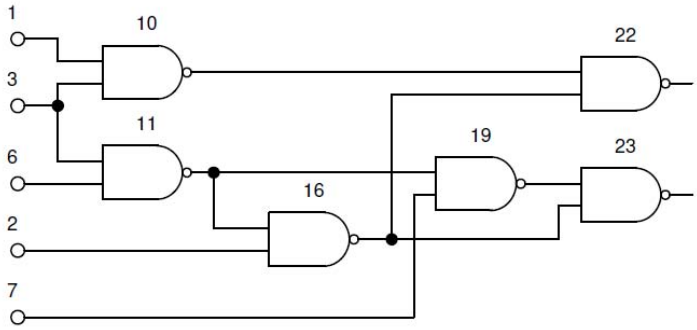
```
# 5 inputs
# 2 outputs
# 0 inverters
# 6 gates(6 NANDs)

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

OUTPUT(22)
OUTPUT(23)

10 = NAND(1, 3)
11 = NAND(3, 6)
19 = NAND(11, 7)
16 = NAND(2, 11)
22 = NAND(10, 16)
23 = NAND(16, 19)
```

(a) c17 BENCH File



(b) c17 Circuit

Figure 2.6: BENCH file and circuit schematic for the c17 benchmark circuit.

each adder in the circuit. The top 15 half adders take on a configuration where the two NOR gates labeled V are NOT gates and there is no carry in. The bottom half adder takes on a configuration where the two NOR gates labeled W are NOT gates and there is no carry out. These make up a total of 16 half adders in the circuit. The remainder of the circuit is composed of 224 full adder components.

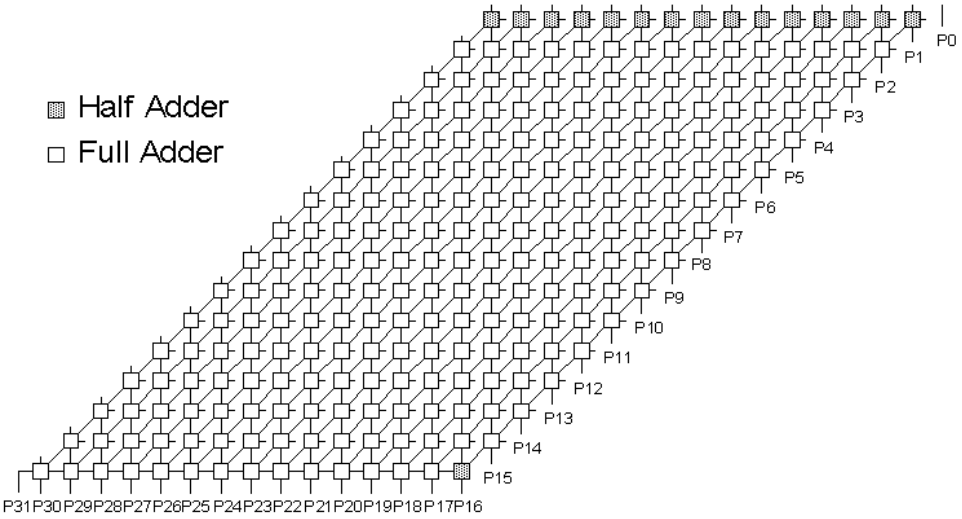


Figure 2.7: Matrix arrangement of the full and half adders of the c6288 16-bit multiplier [3].

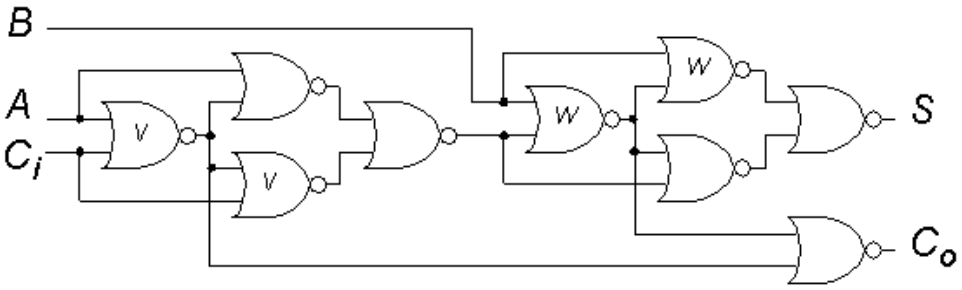


Figure 2.8: Topology of c6288 adders [3].

III. Defeating Component Identification Using Boundary Blurring

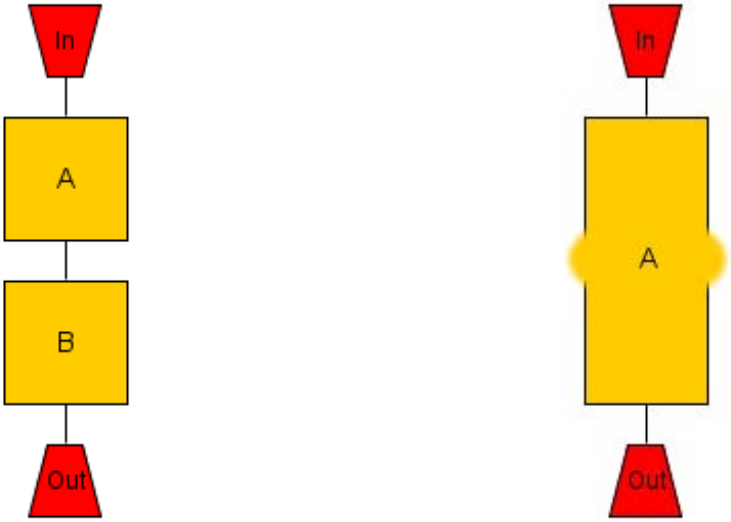
This research focuses on hiding circuit components increasing the cost of reverse engineering of critical technologies. Component identification algorithms exist making component boundaries identifiable [15]. Driving reverse engineering costs to unreasonable levels may result from effectively hiding component boundaries in larger circuits. Boundary blurring algorithms using both deterministic and random measures obscuring component boundaries and forcing reverse engineers to rely on other component identification methods is our research focus.

3.1 Problem Definition

When producing a circuit variant for reverse engineering protection, function discovery time for an obfuscated circuit is one measure of success. Since component identification enhances this process, measuring identification time answers several questions about what level of obfuscation provides a reasonable amount of protection. A second measure of success is the number of components identified within a circuit. Determining the effectiveness of implemented obfuscation techniques and to what extent each must be applied is possible when the number of circuit components is known. White box variation will not hide all circuit information. However, it will increase the cost of reversing a circuit by increasing complexity and reversing time. Past research in software protection focuses on random circuit variants [6, 10], where we turn our focus to more deterministic approaches.

Implementing a component identification tool for characterizing current obfuscation techniques is our first research goal. When circuits have large I/O space a reverse engineer will seek alternate means for discovering unknown circuit function. Understanding the construction of a circuit, or its white box structure, is one such method. Component identification gives an adversary insight into the construction of an unknown circuit. When internal components are known understanding overall circuit function becomes much easier. Boundary blurring is a deterministic technique

which makes component boundaries less distinct causing identification algorithm failure. Figure 3.1 illustrates the basic boundary blurring concept. Realizing boundary blurring algorithms for the purpose of understanding deterministic white box variants is our second research goal. We target component boundaries for technique application and measure component hiding effectiveness during this research.



(a) Two distinct components.

(b) After blurring is applied, distinct boundaries no longer discoverable.

Figure 3.1: The basic idea behind boundary blurring concepts.

Our component identification tool is implemented in Java for use with the Circuit Obfuscation System (COS). White's subcircuit algorithm enumerates candidate components which are compared to components in our known library. To verify identification time is a function of gate size, circuits of varying size are provided to the component identification tool and components with varying size are compared for equivalence while direct time measurement, in milliseconds, is made.

Two systems are utilized to achieve these research goals. The first system, the COS, is the base system we add boundary blurring algorithms to. We use these additional algorithms for increasing COS performance. The second system is the Component Identification System (CIS). We use this system to characterize the diffi-

culty of reverse engineering circuits and also measure obfuscation performance of the COS. Our implementation of White's algorithm is the foundation of the CIS.

3.1.1 Circuit Components. A subcircuit is a component when combined with all other subcircuits a higher level function is performed. The ISCAS-85 benchmark circuit c6288, is a 16 bit multiplier containing 240 individual components. The components implement either half adder or full adder functionality. Components also may conceal the functions of circuits. Using a permutation component, discussed in Section 2.5.4, may encrypt and hide the output of two or more components in a circuit. Unfortunately, if the component boundaries are not concealed, identification is relatively straight forward.

3.1.2 Component Boundaries. There are nine possible component boundaries in large circuits. Figure 3.2 shows each possible case in a circuit P.

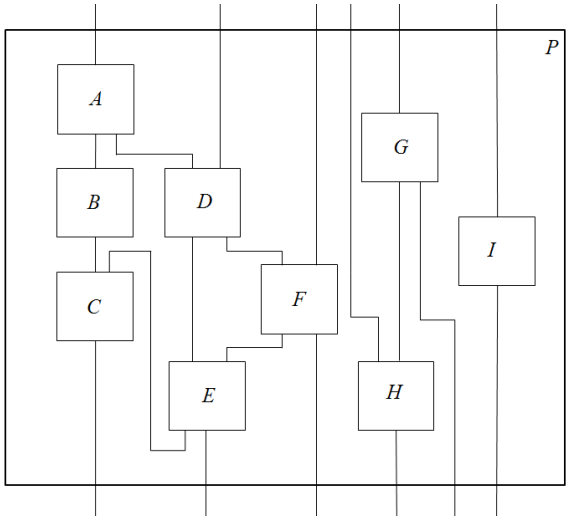


Figure 3.2: Boundaries considered for boundary blurring techniques.

- Component **A** - Input shared with circuit boundary and output shared with other circuit components.
- Component **B** - Input and output shared with circuit components.

- Component **C** - Input shared with circuit components and output shared with circuit boundary.
- Component **D** - Input shared with circuit boundary and component; output shared with circuit components.
- Component **E** - Input shared with circuit components and output shared with circuit component and circuit boundary.
- Component **F** - Input and output shared with circuit boundary and component.
- Component **G** - Input shared with circuit boundary and output shared with circuit boundary and another component.
- Component **H** - Input shared with circuit boundary and component; output shared with circuit boundary.
- Component **I** - Input and output shared with circuit boundary only.

We implement two techniques for hiding component boundaries, the Multilevel Blur and the Don't Care Blur. These are accomplished using different blurring selection strategies maximizing blurring within the circuit. The Multilevel Blur changes the signature in a portion of the circuit. We recover the original signature at a specified blur level. Allowing recovery to occur outside of the original circuit requires additional circuitry. The Don't Care Blur adds additional component inputs. When using component identification tools an adversary could identify invalid components. For example a three input two output full adder would become a four input two output full adder. We discuss these techniques in greater detail in Section 4.1.

3.2 System Boundaries

Each system used in this research has a specific purpose. The COS produces circuit variants aiding in protecting critical technologies. The CIS identifies circuit components, develops a measure of difficulty for reverse engineering circuits and measures the COS obfuscation performance.

3.2.1 Circuit Obfuscation System. The COS is composed of six primary components. First, it contains the circuit model which represents the boolean logic circuits it manipulates. Second, the system utilizes selection strategies facilitating obfuscation methods. A circuit library component, the Component Under Test (CUT), selects a replacement for the chosen subcircuit. The next two components of the system are importers and exporters. These components read in or write out user selected file types containing descriptions of manipulated circuits. Finally, the last component of the system is the personal computer which executes Java code. Figure 3.3 provides and overview of the COS.

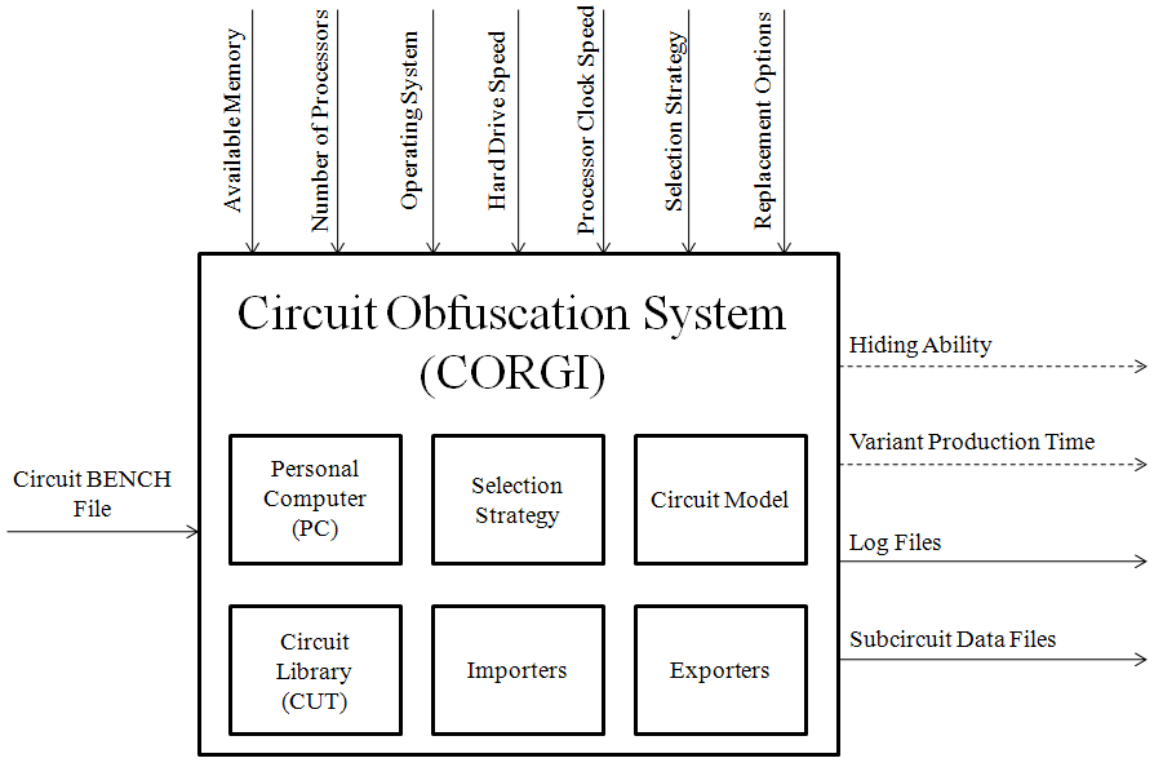


Figure 3.3: Circuit Obfuscation System.

System workload consists of the combinational logic file chosen for obfuscation. System parameters are composed of circuit library generation options controlling the replacement circuit selected for insertion into the circuit. Table 3.1 shows current selection strategies.

Table 3.1: CORGI Selection Strategies [10]

Selection Algorithm	**Description**
RandomSingleGate	Selects a single gate at random
RandomTwoGates	Selects two gates at random
RandomLevelTwoGates	Selects a hierarchial level at random, and limits replacement to two gates selected at random from that level (± 1 level)
FixedLevelTwoGates	Same as RandomLevelTwoGates except the hierarchial level is specified
LargetLevelTwoGates	Same as FixedLevelTwoGates except the hierarchial level is the one containing the most gates
OutputLevelTwoGates	Same as FixedLevelTwoGates except the hierarchial level is 0

The COS has four system outputs, two metric and two response. System metrics include component hiding ability after application of obfuscation techniques and the circuit variant production time. We measure hiding ability with the CIS, discussed in Section 3.2.2, and circuit variant production in minutes. System responses include user selected logs and circuit description files. The COS has multiple log and description files however, for the purposes of this research we categorize them as a single system output.

3.2.2 Component Identification System. The CIS is a tool designed for making reverse engineering of logic circuits an easier task and contains a personal computer, the COS and our implementation of White's subcircuit enumeration algorithm. The personal computer has no specific requirement other than the ability of running Java-based programs. The COS passes the necessary circuit and component objects to the CIS. The subcircuit enumeration algorithm provides candidate subcircuits for comparison against a known component library. Any identified candidate is written in BENCH and graphml file formats. Figure 3.4 provides an overview of the CIS.

We use ten combinational logic circuits, the ISCAS-85 benchmark suite, as the workload. System parameters include processor clock speed and subcircuit output

limit. We limit gate count of identified candidates using a subcircuit size limit. This gate count includes both intermediate gates and input gates. For example, a full adder in c6288 has nine intermediate gates and three input gates making it a 12 gate subcircuit. Larger candidate components slow identification time.

The CIS has six outputs, three metrics and three responses. System metrics include candidate identification time, identification accuracy, and algorithm execution time. We label the time, in milliseconds, from the beginning of component expansion to the time candidate gate size is satisfied as candidate identification time. We also save candidate circuits in BENCH and graphml formats. These files compose our system responses. Total number of correctly identified circuit components measures identification accuracy. For example, if all 240 adders (224 full adders and 16 half adders) of the 16-bit multiplier are identified, the subcircuit algorithm is considered 100% accurate. We define algorithm execution time as the time, in minutes, from the start to finish of circuit searching. The log file containing data for each system metric is the final system response.

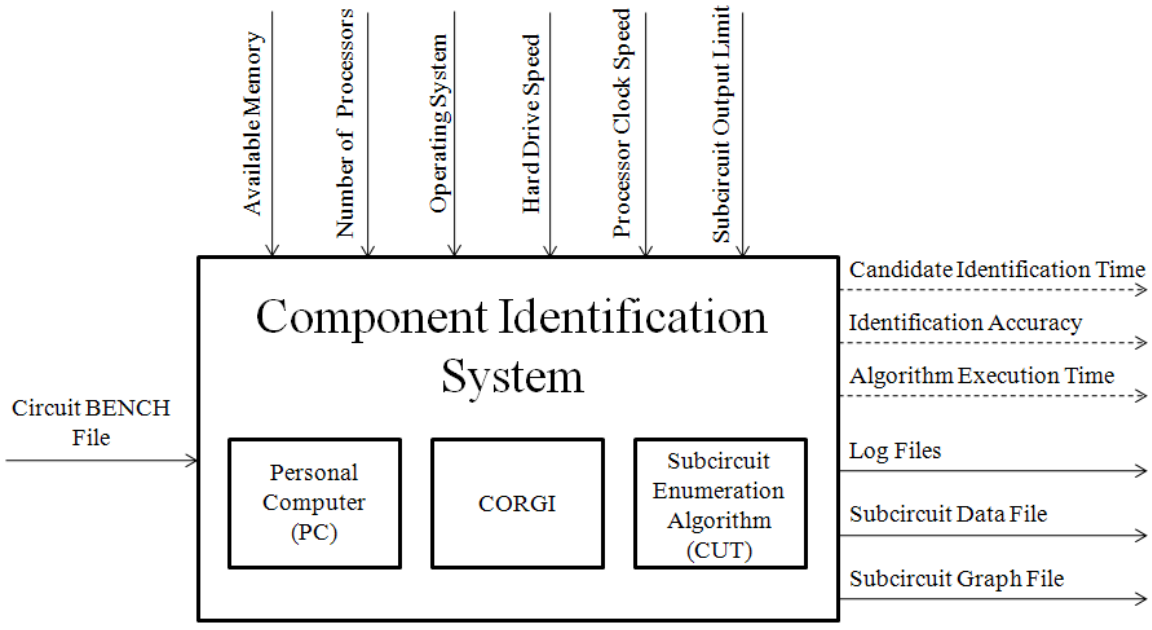


Figure 3.4: Component Identification System

3.3 Component Obfuscation System Services

Creating circuit variants is the COS primary service. When the system is provided a BENCH file and variant options chosen, it produces semantically equivalent circuits with modified white box structure. We measure the effectiveness of this service using the CIS. The COS produces variants in an iterative manner and applies further obfuscation with each iteration. We explore the number of iterations required before components of a known circuit are no longer identifiable.

When components are identified their boundaries (component inputs and outputs) become apparent. We add two blurring techniques to the COS, Multilevel Blur and Don't Care Blur. Multilevel Blurring provides a downward blur in a circuit while Don't Care Blurring provides a similar downward blur, but also allows connections between an additional circuit gate and any other gate not creating a circuit loop. These two techniques, along with different blurring selection strategies, create circuit variants defeating the CIS forcing the use of different circuit function discovery techniques by reverse engineers.

3.4 Component Identification System Services

The CIS identifies components in combinational logic circuits. Successful identification results in system responses of exported BENCH and graphml files, enabling visual identification of components.

3.5 Workload

The COS workload consists of circuits described in BENCH file format. We characterize CIS identification time using the ISCAS-85 benchmark circuits. The primary parameter effecting subcircuit enumeration execution time is the number of component gates. Because each logic gate is expanded into candidate subcircuits, longer execution times result from larger circuits.

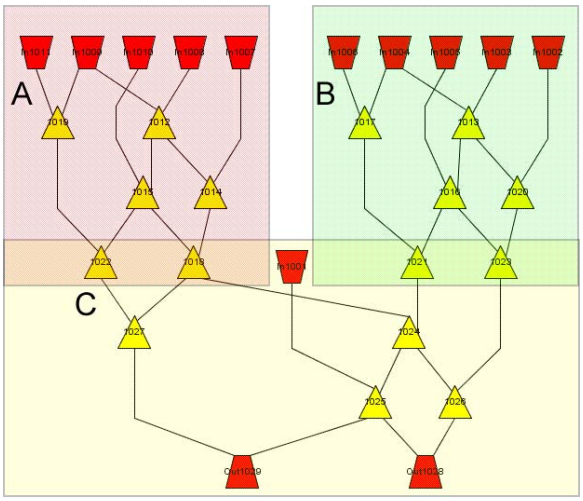


Figure 3.5: Circuit composed of 3 c17 circuits. It is possible to see shared boundaries.

The CIS is supplied circuit variants as its workload. A search is conducted for components identified before boundary blurring techniques are applied. The COS' selection and replacement strategies and boundary blurring techniques will increase circuit gate counts therefore, we expect an increase in search time for candidate identification. Completely effective blurring techniques prevent component identification.

3.6 Performance Metrics

The amount of production time and number of iterations required for circuit variants are the performance metrics for the COS. Time is measured in minutes beginning when the COS is started and ending when the circuit variant is produced. We use the time measurement to verify the performance increase the boundary blurring techniques provide. A boundary blurring iteration is a single attempt to apply either the Multilevel or Don't Care blur.

CIS performance metrics include time necessary for identifying subcircuits, accuracy of identification and overall search execution time. We measure component identification time in milliseconds. This time starts when component expansion begins and ends when component gate count is reached. The size based component identification time is useful in two ways. First, it gives an indication of how large a

circuit variant must be to slow down a reverse engineer using similar tools and secondly, it measures the subcircuit identifier limits. Execution time is a similar metric. We measure execution time in minutes starting when component identification begins and ending when program execution stops. Knowing total execution time provides an estimate of time required for reverse engineering even larger circuits.

Identification accuracy measures whether the algorithm can identify known components in benchmark circuits. The 16-bit multiplier discussed in Section 2.6.2, has 240 individual adders. When all candidates are enumerated, we use additional heuristics for equivalence checking. If these are not identified the algorithm is considered inaccurate.

3.7 System Parameters

Three main parameters affect COS performance. With increased computer clock speed, circuit variant production time is decreased. Selection strategy and replacement circuit options also affect COS run time. These strategies and options are listed in Table 2.1 and 2.2 respectively.

Two primary parameters affect CIS performance. First, is CPU clock speed. Since subcircuit enumeration is a recursive expansion of subgraphs, increased processor speeds improve candidate identification. The subcircuit output limit affects system performance. Because candidate expansion is recursive, larger subcircuit limits increase subcircuit identification time. We identify the maximum practical subcircuit output limit during our analysis.

Other system parameters affecting performance of both systems include size of available memory, number of PC processors, type of operating system, and hard drive speed. We do not analyze these parameters during this research.

3.8 Factors

Factors are a subset of system parameters varied during performance measurements. Each value a factor takes on is a level. We use specific factors and levels for each system used in this research for obtaining our experimental goal. CIS experimental goal, is determining time for discovering components in a circuit by a reverse engineer. COS experimental goal, is determining its effectiveness in hiding components in a circuit.

3.8.1 Characterizing Component Identification System Candidate Identification Time. The size limit of the output component is the primary parameter affecting performance. This limit controls how deep the identification algorithm's recursive expansion goes. Once a candidate component is identified, it is saved and the next gate is expanded based on specific enumeration rules. The circuit gate count also effects the performance of the system. When large circuits are enumerated, just as with the size limit of the output circuit, the recursive expansion goes much deeper. Combining a large circuit and large component size produces expected run times of hours and possibly days.

We expect an increase in identification as the output limit is increased, so the levels and factors are chosen carefully. We expect quick execution with subcircuit output limits below 25 gates. Once an output size of 25 is reached, especially in large circuits, the time may increase to an unreasonable level. For this reason, subcircuit size levels are three through 25, inclusive, and then steps of 15 until an output size of 100 is reached. Workload factors are the ISCAS-85 benchmark circuits listed in Table 2.3. These circuits represent a broad range of circuit sizes. Table 3.2 shows the factors and levels of our analysis.

Practical component identification using our CIS, requires searching for components in descending size order. We use output size limit levels to estimate time required to search a circuit.

Table 3.2: Factors and levels for characterizing component identification time of the Component Identification System

Factor	Level	
Circuit Size Subcircuit Size	Subcircuit Size	Circuit Size
	{3, 4,. . . , 25, 40, 55, 70, 85, 100}	160
		202
		383
		546
		880
		1193
		1669
		2307
		2406
		3512

3.8.2 Characterizing the Baseline Circuit Obfuscation System. Parameters effecting the COS most are replacement options, selection strategies and number of obfuscation iterations. There are six different replacement options, discussed in Chapter II, utilized by the COS. These options form a six tuple providing 64 different replacement options. From empirical studies, FFFTTT produces circuits most like traditional VLSI circuit designs [6]. We use a random selection strategy where two gates is replaced with three and two gates is replaced with four for our subcircuit replacement. We determine how many iterations are performed by the initial circuit size. Table 3.3 outlines factors and levels for characterizing COS performance before the using boundary blurring techniques.

3.9 Evaluation Technique

We use direct measurement for evaluating the CIS. Measuring the identification time of each candidate component at each size level determines system performance. This is repeated for each workload circuit. We record subcircuit identification time in milliseconds and complete execution time in minutes in the experiment log files. Since components of each benchmark circuit is known, we determine identification accuracy by comparing the candidate subcircuits to known subcircuits. This automated

Table 3.3: Factors and Levels of the Circuit Obfuscation System Experiments

Factor	Level		
Selection Strategy	Circuit	Iterations	Gates Replaced
RandomTwoGate	c6288	1000	3
			4
		1500	3
		2000	3
		2500	3
		3000	3
	c10448	25	3
			4
		50	3
			4
		75	3
			4
		100	3
		125	3
	c3315555	250	3
			4
		375	3
			4
		500	3
		625	3
		750	3
	c70281304	500	3
			4
		750	3
		1000	3
		1250	3
		1500	3

process compares circuit semantics. We ensure the number of inputs and outputs are equivalent as well as circuit function.

We analyze the COS using direct measurement and determine performance of component hiding using the CIS. Circuit variants produced with each iteration level are searched for known components. We expect variants with higher iteration levels to have fewer components identified. After these measurements are made we apply boundary blurring techniques and repeat the identification process.

3.10 Experimental Design

Full factorial experiments are used. We first characterize CIS performance by enumerating candidate components in each ISCAS-85 circuit. We show output size levels for these experiments in Table 3.2. Next, component hiding experiments are ran on variants of c6288 and three benchmark circuits of interest using factors and levels in Table 3.3. This gives a baseline of effectiveness for current obfuscation techniques. Finally, we use the CIS to measure hiding effectiveness of our boundary blurring techniques. Factors and levels for blurring experiments are shown in Table 3.4.

Table 3.4: Factors and levels for boundary blurring experiments.

Factor	Level	
Selection Strategy	Circuit	Iterations
IdentifiedBoundary	c6288	480
	c10448	8
	c3315555	36
	c70281304	77
MaxFanOut	c6288	463
	c10448	4
	c3315555	1
	c70281304	3

3.11 Summary

Implementing boundary blurring techniques to frustrate or prevent reverse engineering is our primary research goal. Our second research goal is implementing

a component identification tool for measuring boundary blurring effectiveness. We perform these measurements using methods outlined in this chapter. Circuits used include the ISCAS-85 benchmark circuits and custom circuit types, one composed of parallel and series components and another with very large I/O space. We will show boundary blurring is an effective measure against reverse engineering.

IV. Results

Hiding circuit components is a critical element of software and circuit protection. If an adversary can not identify circuit components they are forced to perform other methods of analysis. Circuit boundaries become evident when component identification tools are used. Performing blurring algorithms on circuits in both random and deterministic manners causing component identification algorithms to fail is a counterneasure.

In this chapter, two boundary blurring algorithms are detailed. We explain and present results for experiments measuring the current effectiveness of the Circuit Obfuscation System (COS) and performance of the Component Identification System (CIS), both outlined in Chapter III. This chapter is divided into the following sections. Section 4.1 details each of the boundary blurring algorithms, Section 4.2 describes our implementation of a component identification tool, Section 4.3 details custom benchmark circuits created for this research, Section 4.4 provides provides results of component identification tool performance, Section 4.5 discusses the current effectiveness of the COS against component identification, Section 4.6 describes experiments and results of the Multilevel Blur, Section 4.7 describes experiments and results of the Don't Care Blur and Section 4.9 is a detailed analysis of candidate component identification time.

4.1 Boundary Blurring Algorithms

We added two boundary blurring techniques, briefly mentioned in Section 3.3, along with blurring selection strategies to the variant producing algorithms already implemented in the COS. We accomplished this using selection strategies both random and deterministic in nature. Each strategy selects the replacement gate, replacement gate type and blur level.

Definition - *The multilevel blur* ***replacement gate*** *is the circuit gate having its gate type changed during a single blur iteration. In the don't care blur, the replacement*

gate type is not changed. However, it will have a newly introduced gate connected to its output.

Definition - *The **replacement type** is the modified gate type of the replacement gate when multilevel blurring is executed. In a don't care blur, this is the gate type of the new gate connected to the replacement gate's output.*

Definition - *The **blur level** is the number of levels closer to the output of the circuit at what point the modified signals in the circuit are recovered. Circuit outputs are level zero.*

Definition - *A **recovery gate** is a gate at which modified signals in the circuit are recovered. There may be one or many recovery gates for a single blurring iteration.*

4.1.1 Blurring Selection Strategies. We utilized three primary blurring selection strategies for each blurring technique. These are a random strategy, max fan-out strategy and identified boundary strategy. Each primary strategy has variations on which elements are random or deterministic. We outline these below.

- SelectionMaximumFanOut - Selects all gates in a circuit that have maximum out degree.
- SelectionMaximumFanOutReplaceType - Selects all gates in a circuit that have maximum out degree and randomly selects the replacement gate type.
- SelectionMaximumFanOutReplaceTypeLevel - Selects all gates in a circuit that have maximum out degree, randomly selects the replacement gate type and randomly chooses the blur level.
- SelectionRandomReplacement - Randomly chooses the replacement gate in a circuit.
- SelectionRandomReplacementReplaceType - Randomly chooses the replacement gate and the replacement gate type.

- SelectionRandomReplacementReplaceTypeLevel - Randomly chooses the replacement gate, replacement gate type and the blur level.
- SelectionIdentifiedBoundaryReplaceType - Randomly selects the replacement type for the blur.
- SelectionIdentifiedBoundaryReplaceTypeLevel - Randomly selects the replacement type and blur level.
- SelectionIdentifiedBoundaryRndPredecessorReplaceType - Randomly selects a single predecessor gate of an identified boundary gate and randomly selects the replacement type.

When gates with maximum fan-out include circuit input gates, the next smaller fan-out is used. Circuit inputs are not used by our blurring algorithms.

4.1.2 Multilevel Boundary Blur. Initially, we implemented multilevel blurring for component hiding. This blur is applied at a specific depth, or level, in the circuit. Applying multilevel blurring requires selecting or specifying a replacement gate. The replacement gate type is modified during the blurring process causing a signal change in all gates succeeding it. The signals are recovered so the circuit remains semantically equivalent. Recovery gates are the point where signals are recovered and are identified by obtaining replacement gate successors down to the specified blur level.

Figure 4.1 is a circuit with three blur levels labeled. We show the replacement gate labeled G and the gates of each level below it labeled with their respective level. When the gate type of G is modified, any gate in which a path exists between gate G and itself is effected. We execute the following steps to recover the signals:

1. Determine the intermediate and recovery gates. An iterative loop is performed a number of times equivalent to the blur level being performed. On the first iteration the successors of the replacement gate are obtained and on each suc-

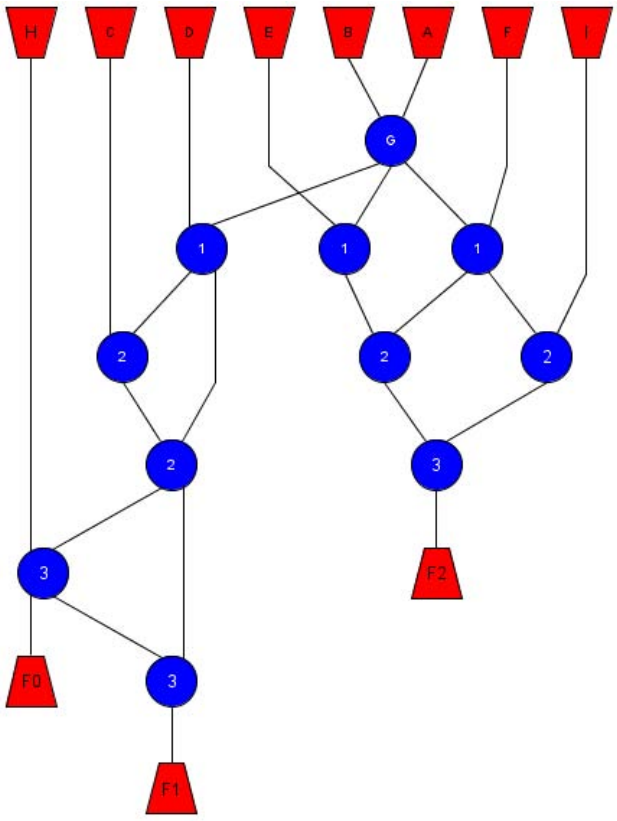


Figure 4.1: A level three blur circuit.

cessive iteration the successors of those gates are obtained. The gates obtained during the last iteration are labeled the recovery gates.

2. Determine the inputs needed to evaluate the original function, F_0, and modified function, F_1. Each input of the replacement gate is added to the input list as input A and input B therefore, a replacement gate must have an in-degree of two. The remaining inputs are added by iterating over the intermediate and recovery gates and adding, to the input list, any predecessor that is not in the list of intermediate or recovery gates. The size of the input list equals n.
3. Create a working circuit to evaluate F_0 and F_1.
4. Generate an input bit list. This list is generated based upon the number of inputs to the working circuit. There are 2^n input values for a given circuit.
5. Evaluate F_0 of the recovery gate.
6. Change the replacement gate type to its new type where $\Omega = \{AND, NAND, OR, NOR, XOR, XNOR\}$.
7. Evaluate F_1 of the recovery gate with the modified replacement gate.
8. Determine the bits in F_0 that must be recovered. This is accomplished by iterating through the F_0 string. The index of each bit with value 1 is recorded.
9. Create a term list for performing a Quine-McCluskey sum of products (SOP) reduction. This term list is produced by taking the input string and F_1 value for each index in step 8 and concatenating them together.
10. Perform the SOP reduction to create a reduced term list.
11. Disconnect the recovery gate from its successors and store them in a list of gates to reconnect.
12. From the reduced term list find inputs that are inverted and add necessary gates to the original circuit. Store the inverted literals.
13. Create necessary AND and OR gate connections in the original circuit based upon results from SOP reduction.

14. Reconnect the SOP output to the appropriate reconnection gates from the reconnect gate list.

Figures 4.2, 4.3 and 4.4 show an example of simple level one blur on a 3-1-3 circuit where the output gate is a buffer. Gate four is the replacement gate, whose type is changed from an AND gate to an OR gate and gate five is the recovery gate. Index seven is the only term in F_0 that must be recovered, as seen in Table 4.2. Because only one term is recovered, no SOP reduction is required. The resulting term used to recover the modified signal is {1 1 1 0} expressed by the logic function $ABCF_1'$.

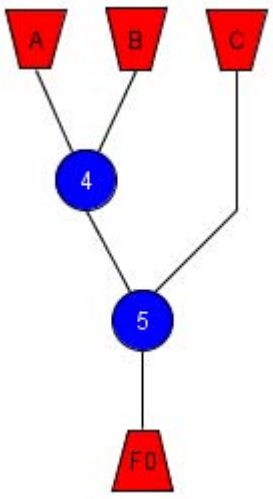


Figure 4.2: A simple circuit in which a level one blur is applied and no SOP reduction is required. Gate four, the replacement gate, is modified from an AND gate to an OR gate, and gate five is the recovery gate.

The second circuit, shown in Figure 4.5, requires a SOP reduction. In this case, Table 4.2 shows there are three terms for reduction, {0 1 1 0, 1 0 1 1, 1 1 1 1}. Using Quine-McCluskey reduction results in reduced terms {0 1 1 0, 1 X 1 1}. We interpret the X in the second term as a don't care for input B. These reduced terms produce the logic function $A'BCF_1' + ACF_1$.

During a multilevel blur, working circuits with a large number of inputs may result. This makes it possible for a high number of reduction terms. For this reason a

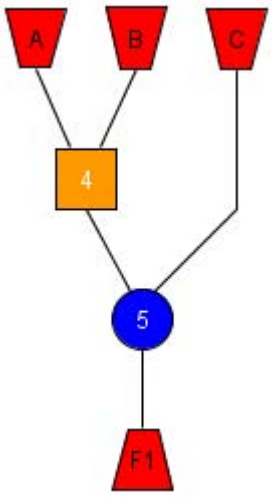


Figure 4.3: Circuit of Figure 4.2 after the replacement gate type is changed.

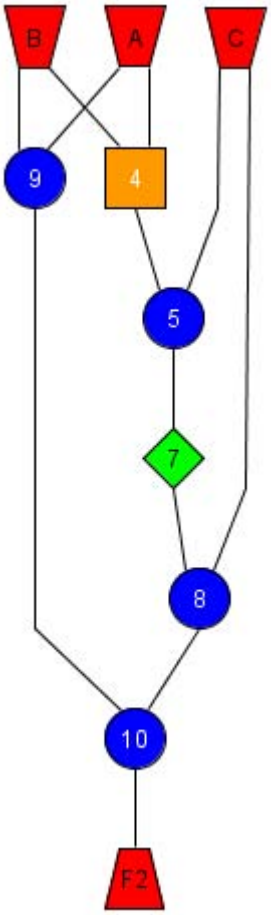


Figure 4.4: The final circuit after the replacement gate is modified and the original signal, F_0, is recovered.

Table 4.1: Truth table for 3-1-3 circuits in Figures 4.2, 4.3 and 4.4. The original signal, F_0, is modified to F_1 when the replacement gate type is changed and is recovered with SOP reduction as seen in function F_2. Function F_2 is equivalent to F_0.

A	B	C	F_0	F_1	F_2
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	1	0	1

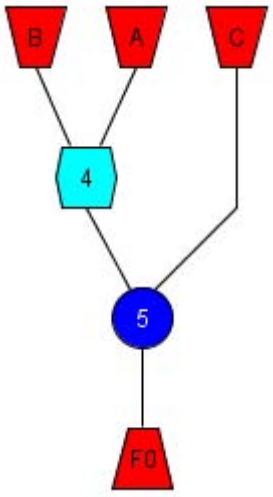


Figure 4.5: A simple circuit in which a level one blur is applied and a SOP reduction is required. Gate four, the replacement gate, is modified from an OR gate to an AND gate, and gate five is the recovery gate.

Table 4.2: Truth table for the circuit shown in Figure 4.7. Function F_2 is equivalent to F_1.

A	B	C	F_0	F_1	F_2
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	1	0	1
1	0	0	0	0	0
1	0	1	1	0	1
1	1	0	0	0	0
1	1	1	1	1	1

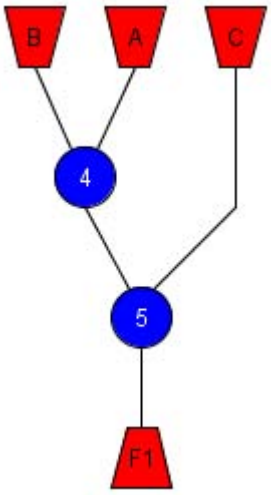


Figure 4.6: Circuit of Figure 4.5 after the replacement gate type is changed.

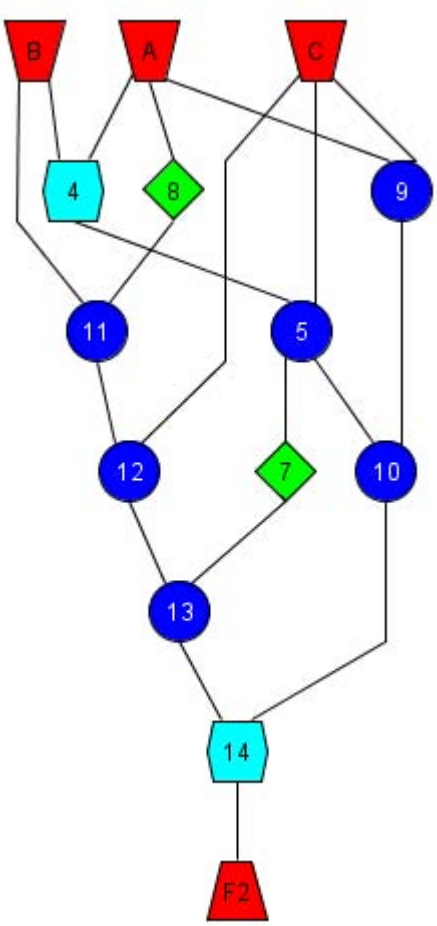


Figure 4.7: The final circuit after the replacement gate is modified and the original signal, F_0, is recovered.

limit of eight inputs is placed on the working circuit. When this input size is exceeded, no blurring takes place for the current replacement gate.

4.1.3 Don't Care Boundary Blur. The Don't Care Blur introduces an additional input to a component or circuit changing its black box structure. This input does not affect the component output. However, it will cause CIS failure. Primary cause of failure is a newly introduced input. A random connection between the new gate and another circuit gate forms a connection between two components in a circuit. When the additional input is added at the circuit boundary the algorithm identifies a candidate component with a larger input size. An example of this is a full adder with four inputs. This make no sense for a full adder component. The don't care blur replacement gate has a new gate added directly to its output. The new gate's remaining input allows creation of the new component input. The following steps are executed during a Don't Care Blur:

1. Select the replacement gate.
2. Disconnect and store the replacement gate's successors in a reconnect gate list.
3. Select a recovery gate to connect to the output of the replacement gate where $\Omega = \{AND, NAND, OR, NOR, XOR, XNOR\}$
4. Create a working circuit to evaluate F_0 and F_1.
5. Store the inputs of the replacement gate as input A and Input B.
6. Generate a list of input strings for a two bit input.
7. Evaluate F_0, the output of the replacement gate, and expand F_0 to 8 terms. See Table 4.3 for expansion of F_0.
8. Connect the recovery gate to the replacement gate output of the working circuit and generate a new input list for a three bit input.
9. Evaluate F_1, the output of the recovery gate.

10. Determine the bits in F_0 that must be recovered. This is accomplished by iterating through the F_0 string. The index of each bit with value one is recorded.
11. Create a term list for performing a Quine-McCluskey SOP reduction. This term list is produced by taking the input string and F_1 value for each index in step the and concatenating them together.
12. Perform the SOP reduction to create a reduced term list.
13. From the reduced term list find inputs that are inverted and add necessary gates to the original circuit. Store the inverted literals.
14. Create necessary AND and OR gate connections in the original circuit based upon results from SOP reduction.
15. Reconnect the SOP output to the appropriate reconnection gates in the reconnect gate list.
16. Randomly select a circuit gate and connect it to the additional input. The selection must not create a circuit cycle.

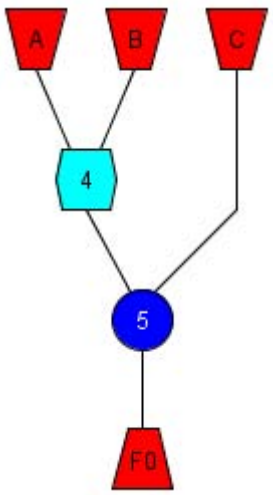


Figure 4.8: A simple circuit before a don't care blur is performed. Gate four is the replacement gate.

Table 4.3: Function F_0 expanded to accommodate the additional input.

A	B	F_0
0	0	0
0	0	0
0	1	1
0	1	1
1	0	1
1	0	1
1	1	1
1	1	1

Table 4.4: Function F_0 and F_1. F_0 must be recovered from the output of F_1.

A	B	D	F_0	F_1
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
1	1	0	1	0
1	1	1	1	1

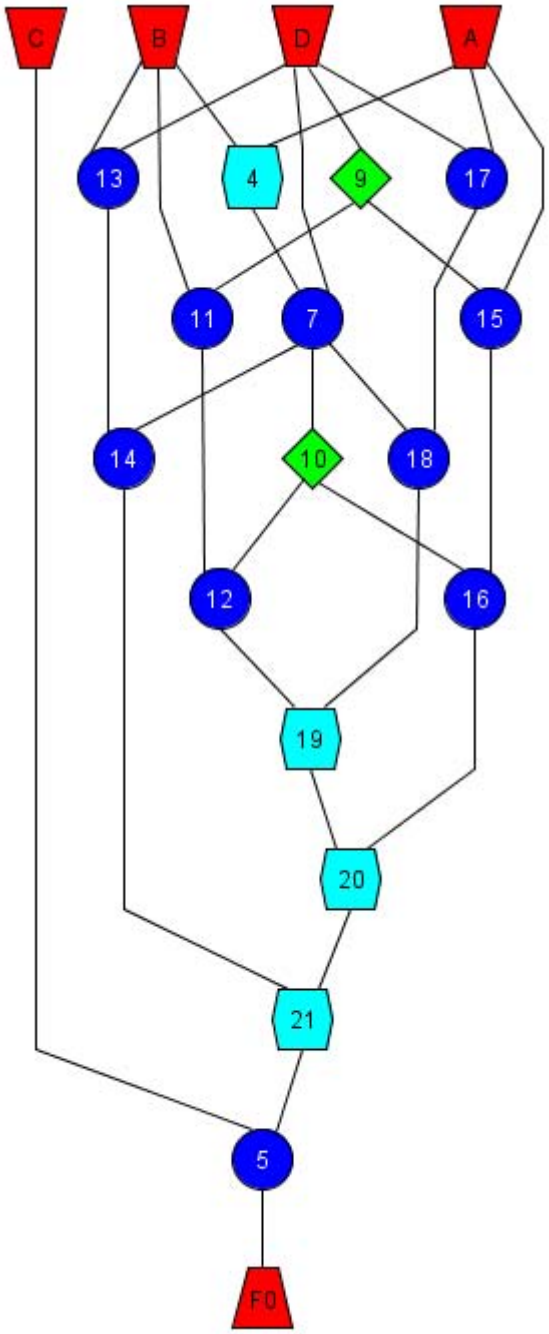


Figure 4.9: A modified circuit resulting from a don't care blur. Gate four is the replacement gate.

Recovery terms for F_0 are indices two through seven resulting in terms {0 1 0 0, 0 1 1 1, 1 0 0 0, 1 0 1 1, 1 1 0 0, 1 1 1 1}. After performing SOP reduction, the reduced terms are {X 1 0 0, X 1 1 1, 1 X 0 0, 1 X 1 1} which produce logic equation $BD'F_1' + BDF_1 + AD'F_1' + ADF_1$. Figure 4.9 shows the modified circuit after a Don't Care blur is performed with additional input D. Since the additional input becomes a don't care input, we connect it to any gate not creating a circuit cycle. The connection point is selected randomly during the blur process.

4.1.4 Additional Inputs and Outputs. Blurring techniques create additional component inputs or outputs. When the multilevel blur is performed and both the replacement and recovery gate are internal to the circuit, no additional inputs or outputs result. However, when the recovery occurs outside the originating component, additional outputs are added in the component containing the replacement gate and additional inputs are added to the component containing the recovery gate. This results from connections crossing component boundaries. The Don't Care blur only adds an additional input to the component containing the replacement gate. These changes affect the component's black box structure.

4.2 Implementing a Component Identification Tool

Reverse engineering a circuit using component identification techniques requires the use of specialized computer tools. In this research, we explore the time an adversary may take identifying components of an unknown circuit. We complete this task by implementing a JAVA based component identification tool. The heart of the identification tool is the CIS. However, the CIS only provides candidate subcircuits of the larger circuit of interest. Since circuit composition is unknown, enumerating candidate components of different sizes accomplishes the component search. Larger components may contain smaller components so searching is conducted in decreasing size order.

It is necessary to check candidates for equivalence to known components. We accomplish this by performing truth table analysis comparing the candidate component and a known library component. This presents a problem for components having a large I/O space. To overcome this, we realized a custom benchmark library containing components with a maximum input and maximum output size of six.

Identification is a multi pass process. A single pass consists of identifying all candidates in a circuit for each size in a size array or range. We perform equivalence checks between candidate components and library components having the same I/O space. When a match is found or all modules with the same I/O space are compared, we terminate equivalence checking. We export BENCH and graphml files for each matching component and after each pass is completed, gates from identified components are removed reducing the size of the circuit for subsequent passes. We repeat this process until no components are identified during a single pass.

4.2.1 Equivalence Checking. Truth table analysis mentioned in Section 4.2 is dependent on input and output order. Because of this dependency, we compare all possible input and output combinations when checking for equivalence using truth table methods. Given n inputs and m outputs, the total number of I/O permutations necessary to confirm a candidate is equivalent is $n!m!$. A 6-4-x circuit requires 17,280 comparisons when a candidate is not equivalent. On the other hand, a 3-2-x circuit, matching the I/O space of the full adder, takes only 12. Equivalence checking for the candidate component terminates immediately when a positive match is made.

4.2.2 Circuit Reduction. After each enumeration pass, all identified components are removed from the circuit under investigation. The possibility of shared boundaries between two or more components prevents the removal of all subcircuit gates. We remove all inputs without predecessor gates, all outputs without successor gates and the remaining intermediate gates. Identifying all circuit components is not always possible without identified component reduction.

4.2.3 Component Library. A component library is necessary for comparing candidate subcircuits discovered during subcircuit enumeration. The library is a directory containing subdirectories labeled with the I/O space of the circuit. For example, a directory labeled 4-3 contains module BENCH files of known four input three output circuits. Because truth table analysis is used for equivalence checking, the composition of a library circuit is not important. Constructing a BENCH file with proper I/O space and output functions and placing it in the proper directory enters a component into the library. Currently the module library contains 28 known components shown in Appendix F.

4.3 Custom Benchmark Circuits

The c17 and c6288 benchmark circuits were instrumental in developing the identification tool of Section 4.2, but the remaining ISCAS-85 benchmark circuits presented certain obstacles, which we will discuss in Section 4.4. To overcome these difficulties, we created a custom component benchmark set. These circuits are shown in Table 4.5. Input and output size of the custom benchmark circuits are chosen to minimize I/O space and for producing larger circuits with known composition. Each custom component performs a random function generated during creation. Each circuit is created in the following way:

1. Choose the input and output size of the circuit.
2. Using RandomCircuitOutput.java, generate a random output for the circuit.
3. Synthesize the circuit using Logic Friday. This ensures a minimized implementation.
4. Using yEd Graph Editor, the Logic Friday synthesis is transcribed to graphml format.
5. ConvertGraphMLToBenchAndShapedGraphML.java converts transcribed graphml files to bench files and updated graphml formats.

Table 4.5: Custom bench files created for this research and component identification tool development.

Circuit Name	Inputs	Outputs	Gates	Component Size
c2215	2	2	13	15
c237	2	3	5	7
c249	2	4	7	9
c3211	3	2	8	11
c3418	3	4	15	18
c3516	3	5	13	16
c3622	3	6	19	22
c4222	4	2	18	22
c4327	4	3	23	27
c4440	4	4	36	40
c5241	5	2	36	41
c5355	5	3	50	55
c5479	5	4	74	79
c6276	6	2	70	76
c63103	6	3	97	103
c64145	6	4	139	145

Appendix D shows the process of creating a simple three input two output circuit. It is worth noting, Logic Friday will synthesize an inverter as a NAND gate with a single input connected to a logic one or either a NOR gate with a single input connected to a logic zero. When this occurs, inverter gates are placed in the transcribed graphml file. Custom components are shown in Appendix B.

4.4 Component Identification Tool Performance

Our component identification tool does not identify all ISCAS-85 benchmark circuit components. These circuits were our target for tool testing. However, since we are unable to identify all components, custom benchmark circuits were created. The cause of these failures result from dependent components, those with shared inputs or gates, in the circuit under investigation. Success was obtained with circuit c17, c6288 and the remaining custom circuits.

4.4.1 Problems Identified in Circuit c432. Circuit c432 identified weaknesses of our component identification tool. Splitting shared inputs into single inputs and disabling rule three of candidate enumeration allows visual identification of module one. However, the shared inverter gates between module two and module three prevents identification of remaining components. Success with c6288 and failures with modules one, two and three in c432 prompted further investigation of independent circuit components. Figure 2.7 shows how c6288 is composed of independent adder components.

Circuit c3633, shown in Figure 4.10, is a simple circuit composed of three components demonstrating components with shared inputs. When provided as the circuit under investigation only component c3418 is identified. Figure 4.11 shows both remaining components after reduction occurs. Input gates 1005, 1006 and 1007 are inputs to removed component c3418. Subcircuit enumeration rule three and the shared input, In1001, cause identification failure.

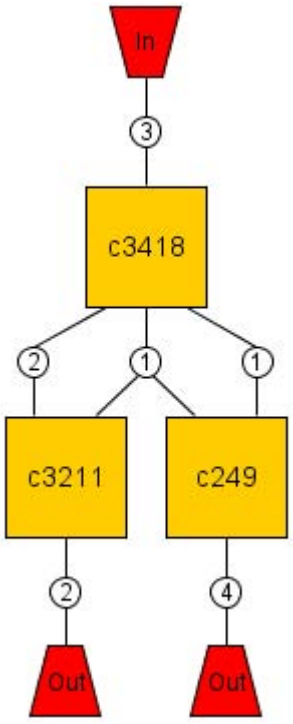


Figure 4.10: Circuit c3633 high level diagram showing three component composition and sharing of input between component c3211 and c249.

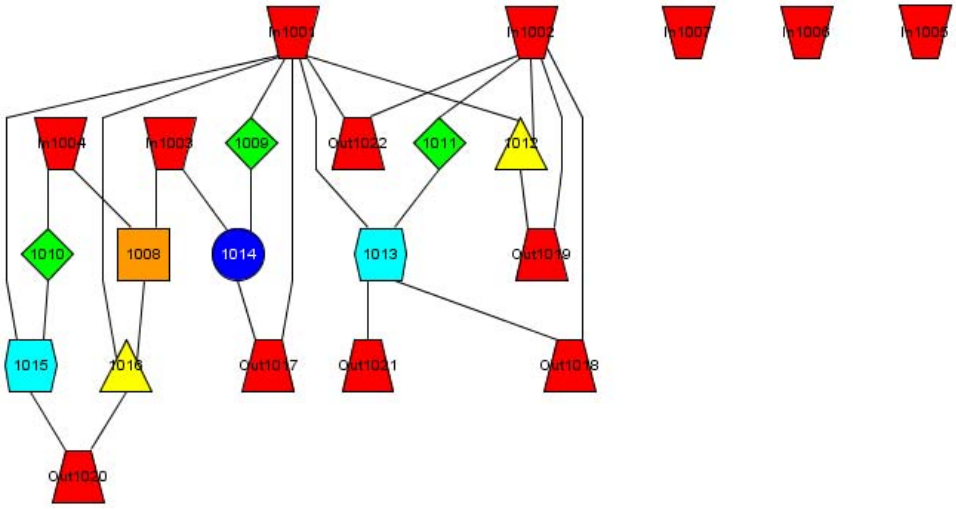


Figure 4.11: Circuit graph of remaining two components in c3633 after component identification.

4.4.2 Initial Component Identification. Before using the custom benchmark components to construct larger circuits, we input each to the identification tool as a circuit under investigation. This ensured our tool will identify each benchmark circuit. We conducted a search for components equivalent in size of circuit under investigation. The identification tool identifies all custom benchmark circuits and c17. Average execution time after five identification experiments is shown in Table 4.6.

4.4.3 Initial Benchmark Circuit Testing. Circuit c6288 is the only ISCAS-85 benchmark where all components are identified. We constructed larger benchmark circuits for further exploring the identification tool. We used Circuits c182449 and c212352 for initial testing. These circuits represent circuits composed of multiple independent components. Circuit c182449 has $2^{18} = 262144$ input combinations while c212352 has $2^{21} = 2.097 \times 10^6$ input combinations. Circuit c3315555 is a 12 component circuit containing all 9 of the component boundary types identified in Figure 3.2. The circuit has $2^{33} = 8.59 \times 10^9$ input combinations. The final circuit created during this research is c70281374. This circuit has 70 inputs, 28 outputs and contains 26 components. The total number of input combinations is $2^{70} = 1.18X10^{21}$ and would

Table 4.6: Mean and median identification time for individual component identification using component identification tool.

Circuit	Mean Execution Time (ms)	Median Execution Time (ms)
c2215	192	99
c237	171.8	77
c249	186.2	78
c3211	190.4	76
c3418	225.6	130
c3516	301	163
c3622	591.6	510
c4221	106.8	106
c4327	212.8	207
c4440	778.4	633
c5241	645.4	641
c5355	1769.6	1765
c5479	2043.6	2042
c6276	8431.8	8439
c63103	35865.8	35939
c64145	137316.8	137319
c17	280.2	92

take 3.74×10^3 years to enumerate using state-of-the-art testers. An adversary is forced to use white box analysis on such circuits. High level diagrams of c182449, c212352, c3315555 and c70281374 are shown in Figure 4.13, 4.14, 4.15 and 4.16 and gate level diagrams are contained in Appendix B. We targeted each component size in initial identification tests using a size array, to decrease search times. Execution time for searches in unknown circuits or circuits with obfuscation take longer. The results of the initial component identification are shown in table 4.7 and experiment design is shown in Figure 4.12.

4.5 Circuit Obfuscation System Performance

We determine the ability for the COS to hide circuit components by using the identification tool on variants of the original circuit. We conducted our search using an appropriate search range and results of variants outlined in Table 3.3. All circuit

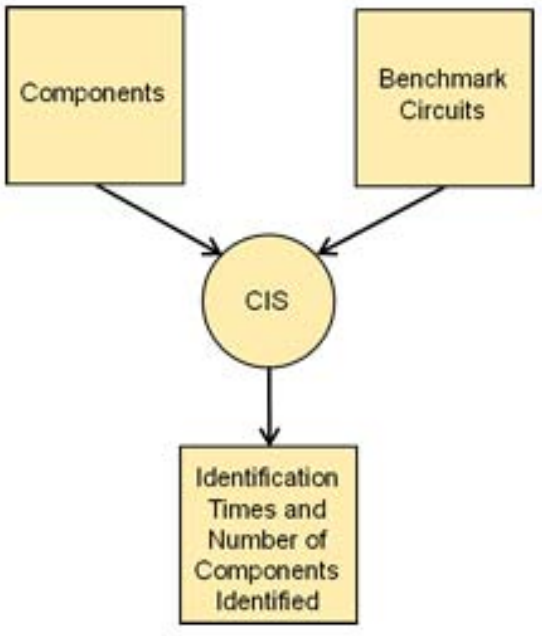


Figure 4.12: CIS experiment design. We provide individual components and benchmark circuits to the CIS recording identification time and number of components identified.

Table 4.7: Mean initial identification times for custom benchmark circuits.

Circuit	Components	Search Set	Passes	Mean Identification Time (min)
c6288	240	{12,11}	1	1.167
c182449	6	{145,103,76, 55,27}	2	11.97
c212352	7	{76,55,21,7}	3	2.28
c3315555	12	{145,103,76, 41,27,18,11,9}	3	16.72
c70281374	26	{145,103,76, 41,27,18,11,9}	4	40.58

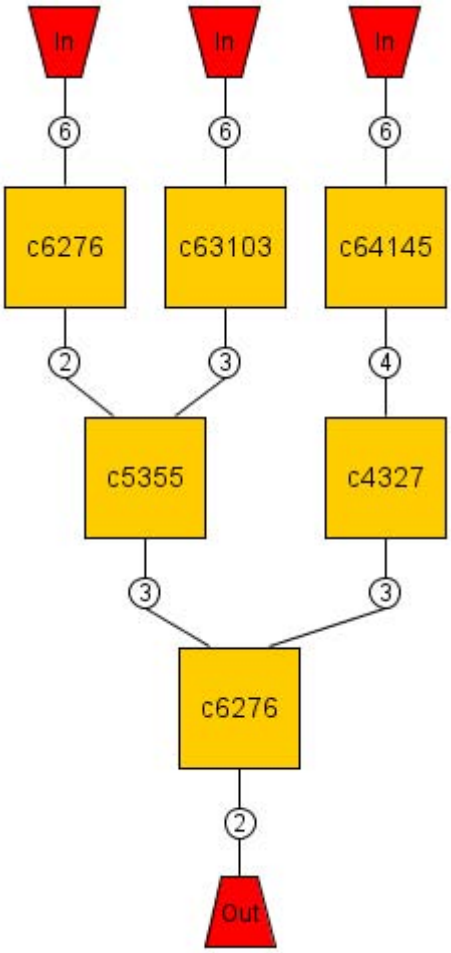


Figure 4.13: Circuit c182449 high level diagram.

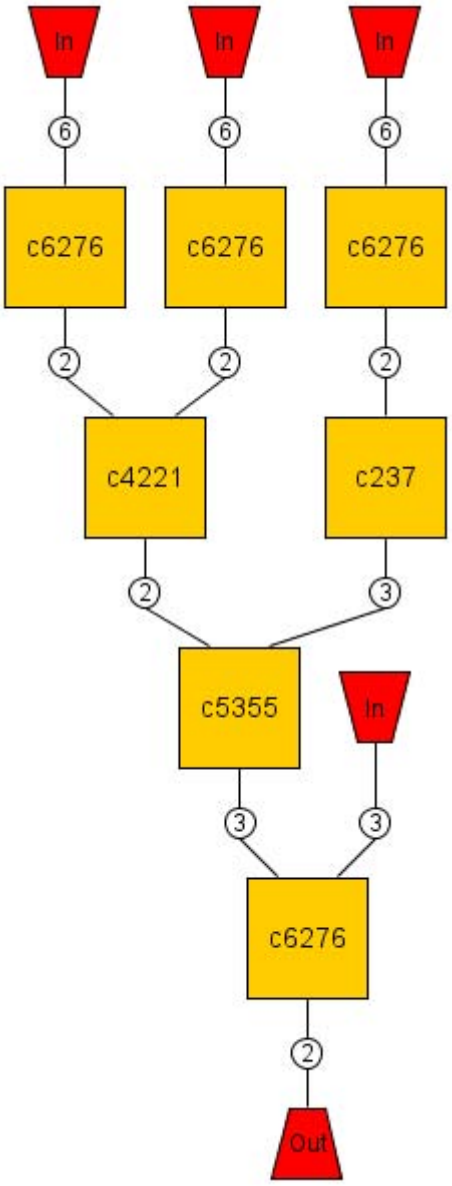


Figure 4.14: Circuit c212352 high level diagram.

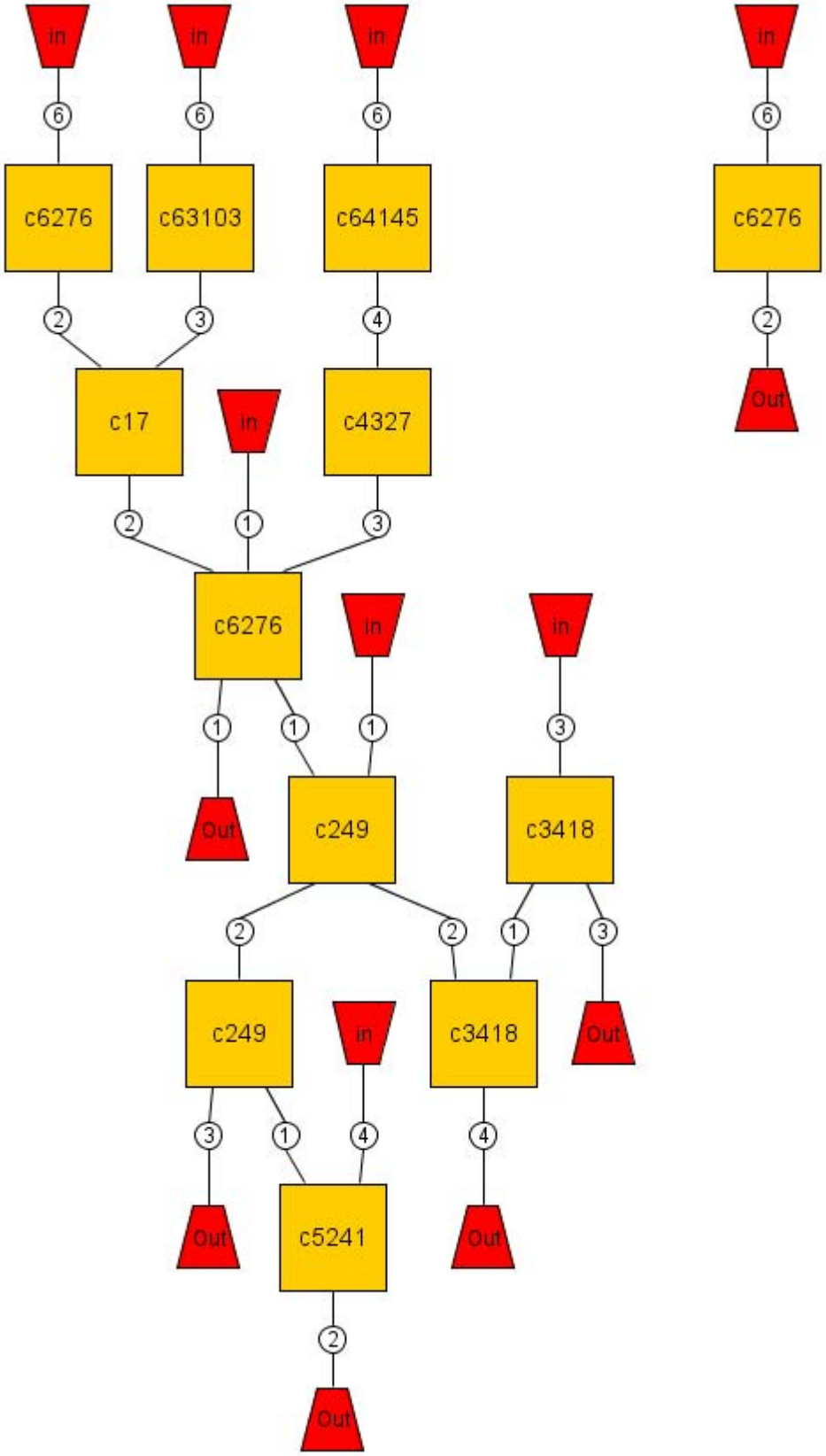


Figure 4.15: Circuit c3315555 high level diagram.

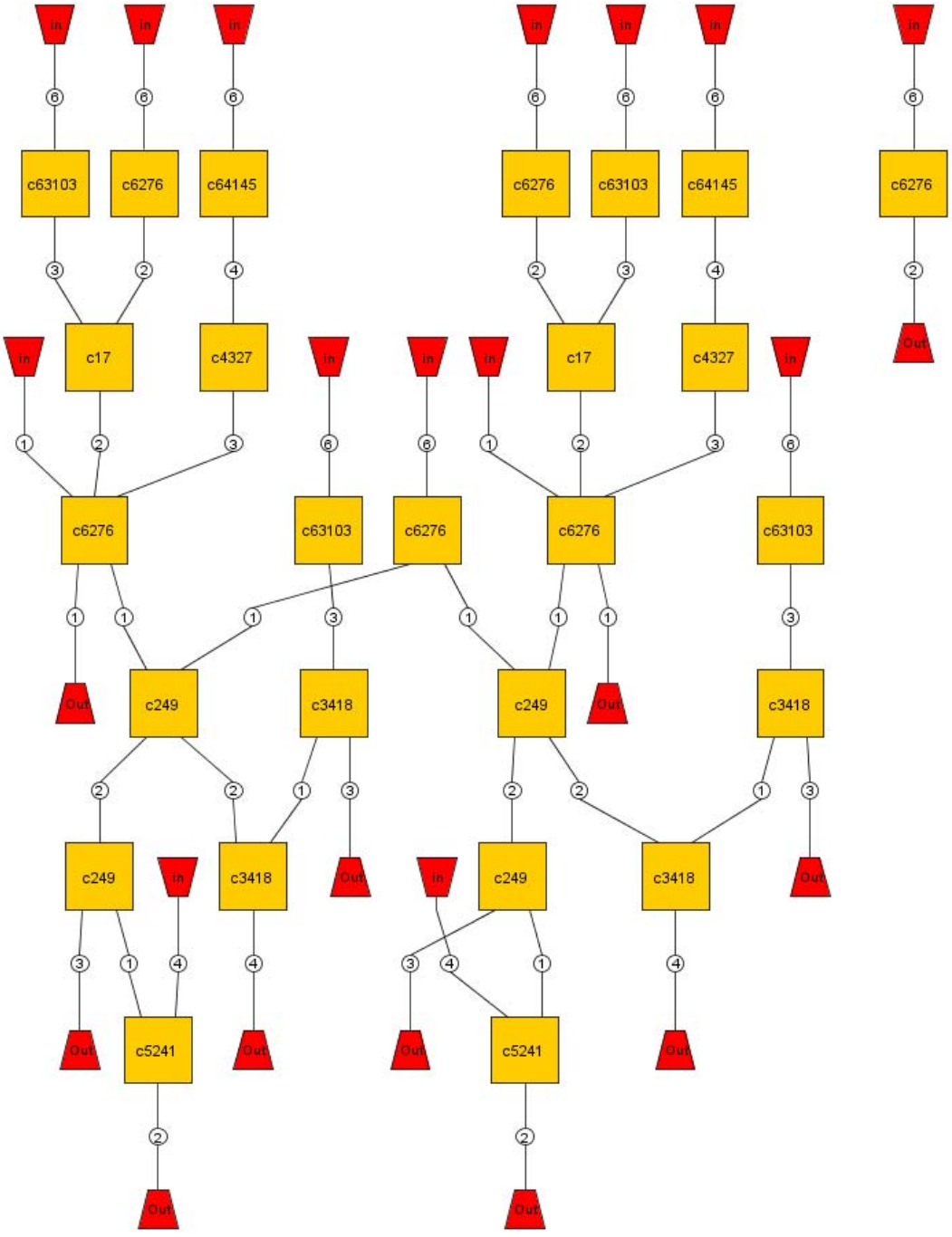


Figure 4.16: Circuit c70281374 high level diagram.

variants created are normalized before providing them to the component identification tool. Figure 4.17 details COS experimental design.

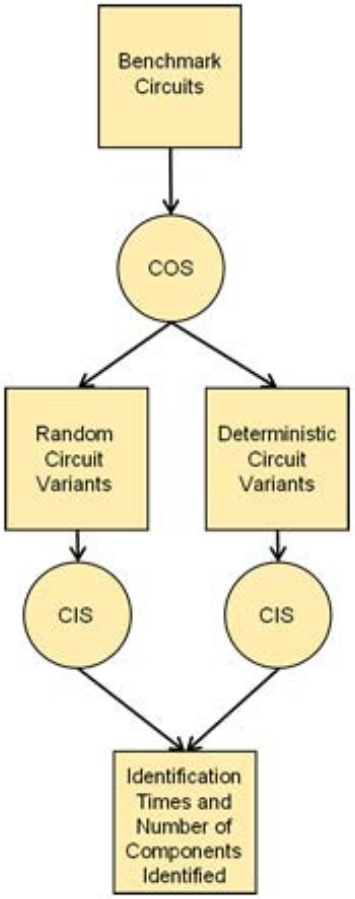


Figure 4.17: COS experiment design. We record identification time and number of components identified for random and deterministic variants.

4.5.1 c6288 Three Gate Replacement. Originally, c6288 contains 2448 gates and is composed of 240 adder components. After normalization, 3000 iterations produced the largest circuit with an increase of 22 gates and also most effective hiding with 19, or 7.9%, of components hidden. Increasing the number of iterations may increase hiding, but empirical evidence shows normalization after three gate replacement results in an almost unchanges circuit. Table 4.8 shows size of circuit variants and number of components identified by the component identification tool.

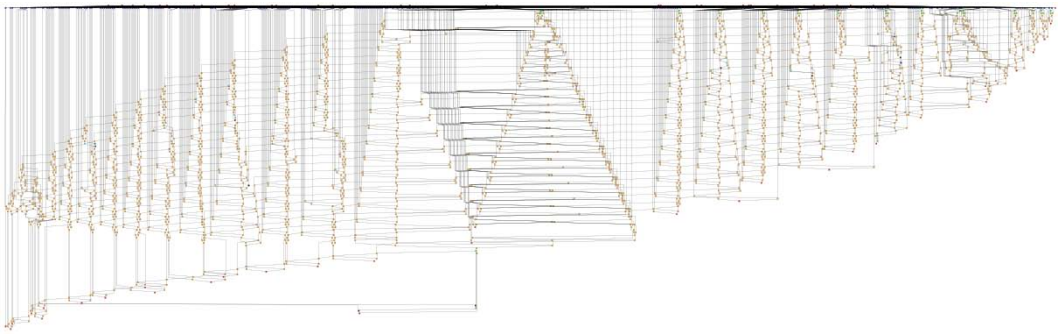


Figure 4.18: c6288 variant with most effective hiding.

Table 4.8: Size of c6288 variants and number of components identified.

Iterations	Replacement Size	New Size	Comp. Identified	Search Time (min)
1000	3	2458	229	1.21
1500	3	2458	230	1.31
2000	3	2456	233	1.25
2500	3	2467	226	1.39
3000	3	2470	221	1.39

4.5.2 c10448 Three Gate Replacement. Originally, c10448 contains 58 gates and is composed of three components. After normalization, 125 iterations produced the largest circuit with an increase of 30 gates. However, after 75 iterations 100% component hiding was achieved. Table 4.9 shows size of circuit variants and number of components identified by the component identification tool.

Table 4.9: Size of c10448 variants and number of components identified.

Iterations	Replacement Size	New Size	Comp. Identified	Search Time (min)
25	3	71	2	1.72
50	3	76	1	3.66
75	3	84	0	2.32
100	3	87	0	2.55
125	3	88	0	3.23

4.5.3 c3315555 Three Gate Replacement. Originally, c3315555 contains 588 gates and is composed of 12 components. This circuit is significant to this research because it contains components covering all nine component boundary cases. Table 4.10 shows size of circuit variants and number of components identified by the com-

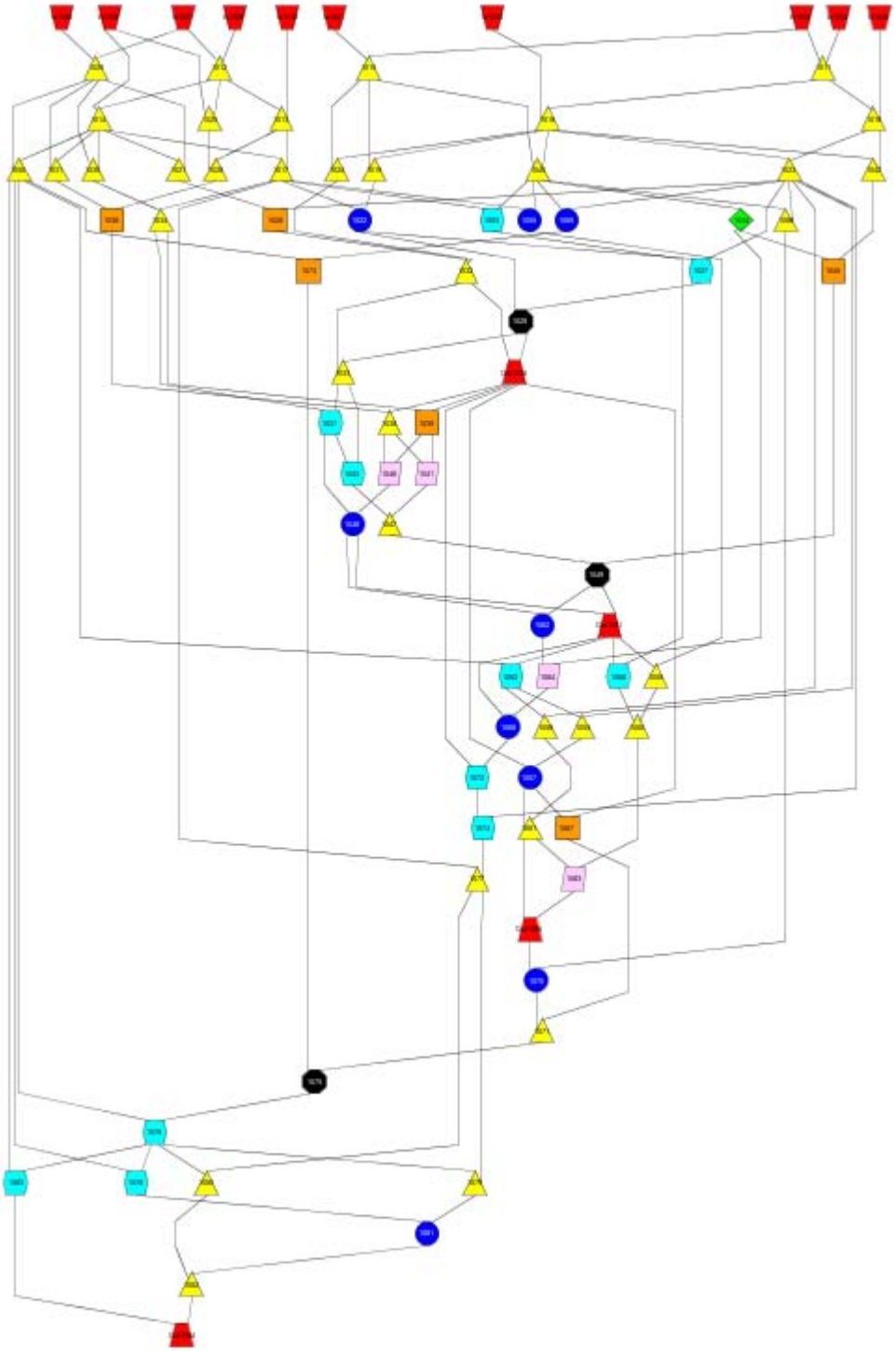


Figure 4.19: c10448 variant with most effective hiding.

ponent identification tool. In each case, the c6276 circuit covering component case I is identified. After normalization 625 iterations produced a circuit equivalent to

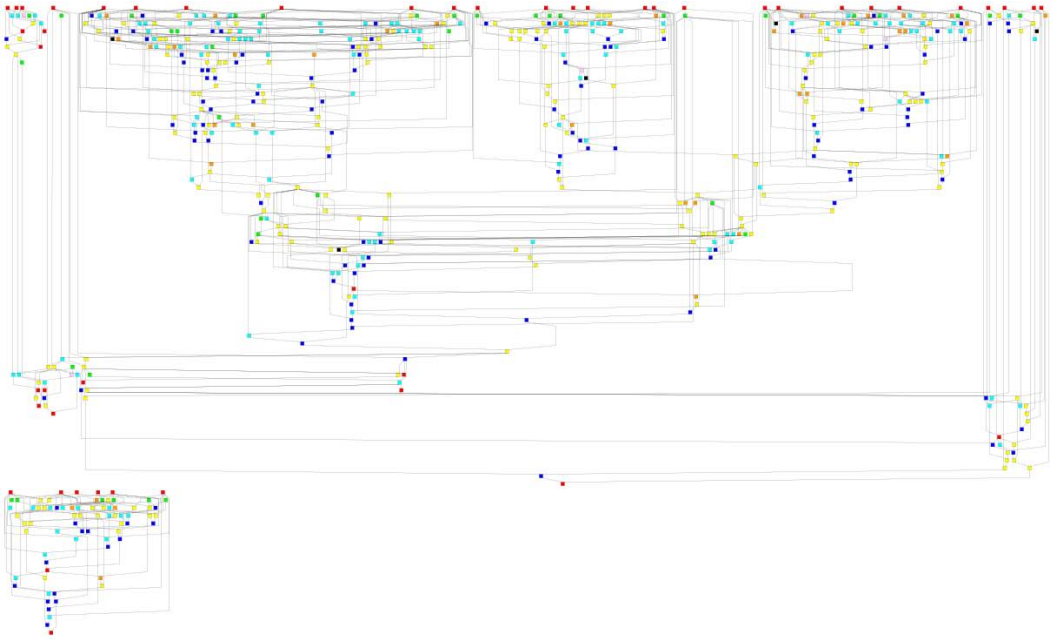


Figure 4.20: c3315555 variant with most effective hiding.

Table 4.10: Size of c3315555 variants and number of components identified.

Iterations	Replacement Size	New Size	Comp. Identified	Search Time (min)
250	3	576	4	639.18
375	3	581	3	1026.67
500	3	579	1	1601.63
625	3	588	4	2085.51
750	3	582	5	1257.49

the size of the original. All other iterations resulted in reduced circuit sizes from normalization removing output buffers.

4.5.4 c70281374 Three Gate Replacement. Originally, c70281374 contains 1374 gates and is composed of 26 components. This circuit contains all nine component boundary cases and has an input size preventing enumeration of all possible input combinations forcing an adversary to perform white box analysis. Table 4.11 shows the results of COS performance tests. The c6276 circuit covering component

case I is identified in all but the 1250 iteration variant. A circuit merge occurred resulting in no identification of the case I component.

Table 4.11: Size of c70281374 variants and number of components identified.

Iterations	Replacement Size	New Size	Comp. Identified	Search Time (min)
500	3	1336	7	2056.35
750	3	1337	7	2021.88
1000	3	1340	12	2672
1250	3	1346	6	2049.53
1500	3	1351	6	2317.89

4.5.5 c6288 Four Gate Replacement. After three obfuscation cycles, the circuit variant contains 5789 gates an increase of 136% from the original circuit. We conducted component identification with a size search range of 11 to 50 gates and only four components were identified. Execution time for a single pass of the component identification tool was 2080 minutes.

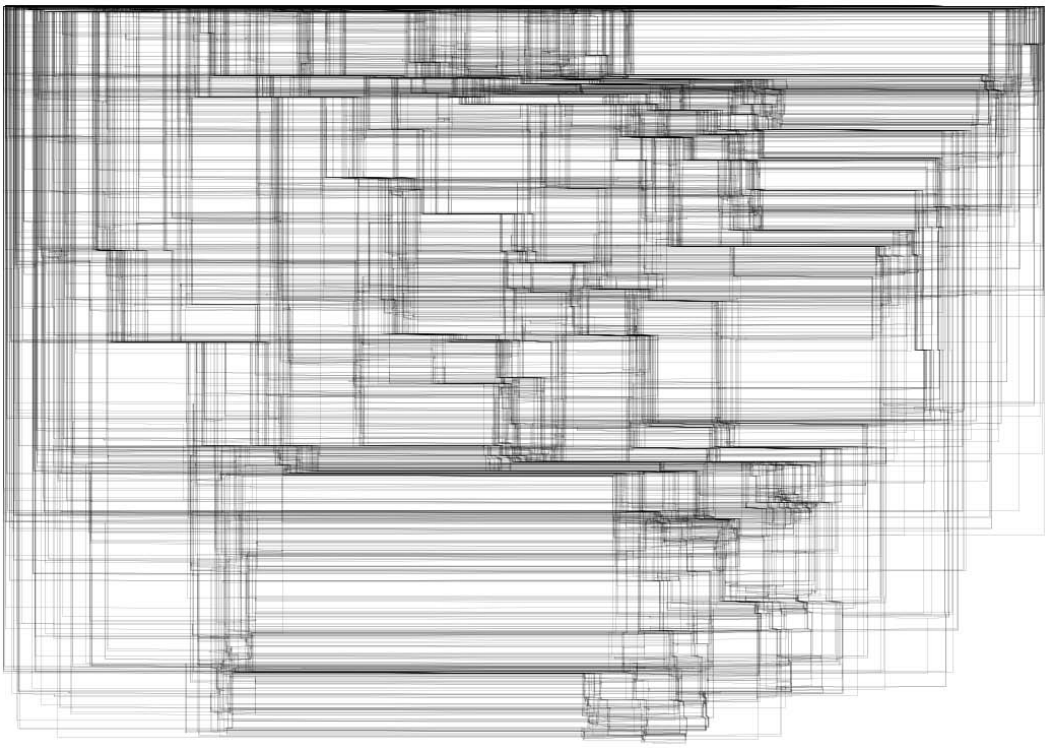


Figure 4.21: c6288 four gate replacement variant.

4.5.6 c10448 Four Gate Replacement. After two obfuscation cycles for each iteration size, the largest variant was produced with 75 selection and replacements. The circuit increased from 48 to 204 gates, a 325% increase. We conducted component identification searches with a search range of 11 to 100 gates. Table 4.12 shows COS performance test results.

Table 4.12: Size of c10448 variants and number of components identified.

Iterations	Replacement Size	New Size	Comp. Identified	Search Time (min)
25	4	151	0	20.21
50	4	150	0	23.21
75	4	204	0	48.49

4.5.7 c3315555 Four Gate Replacement. After two obfuscation cycles for each iteration size, the largest variant was produced with 375 selections and replacements. The circuit increased from 555 to 1442 gates, a 160% increase. We attempted component identification searches with a search range of nine to 175 gates. We terminated component identification after 78 hours of execution due to excessive search time. When execution was terminated, the search for components in the 250 iteration variant had decreased to component sizes of 151 gates and in the 375 iteration circuit component search size decreased to 168 gates. Table 4.13 shows results of COS performance tests.

Table 4.13: Size of c3315555 variants and number of components identified.

Iterations	Replacement Size	New Size	Comp. Identified	Termination Time (min)
250	4	1132	0	4714
375	4	1442	0	4710

4.5.8 c70281374 Four Gate Replacement. After two obfuscation cycles, the produced circuit variant contains 2496 gates an increase of 91% from the original circuit. We conducted component identification with a size search range of 11 to 200 gates. We terminated component identification after 78 hours of execution due to

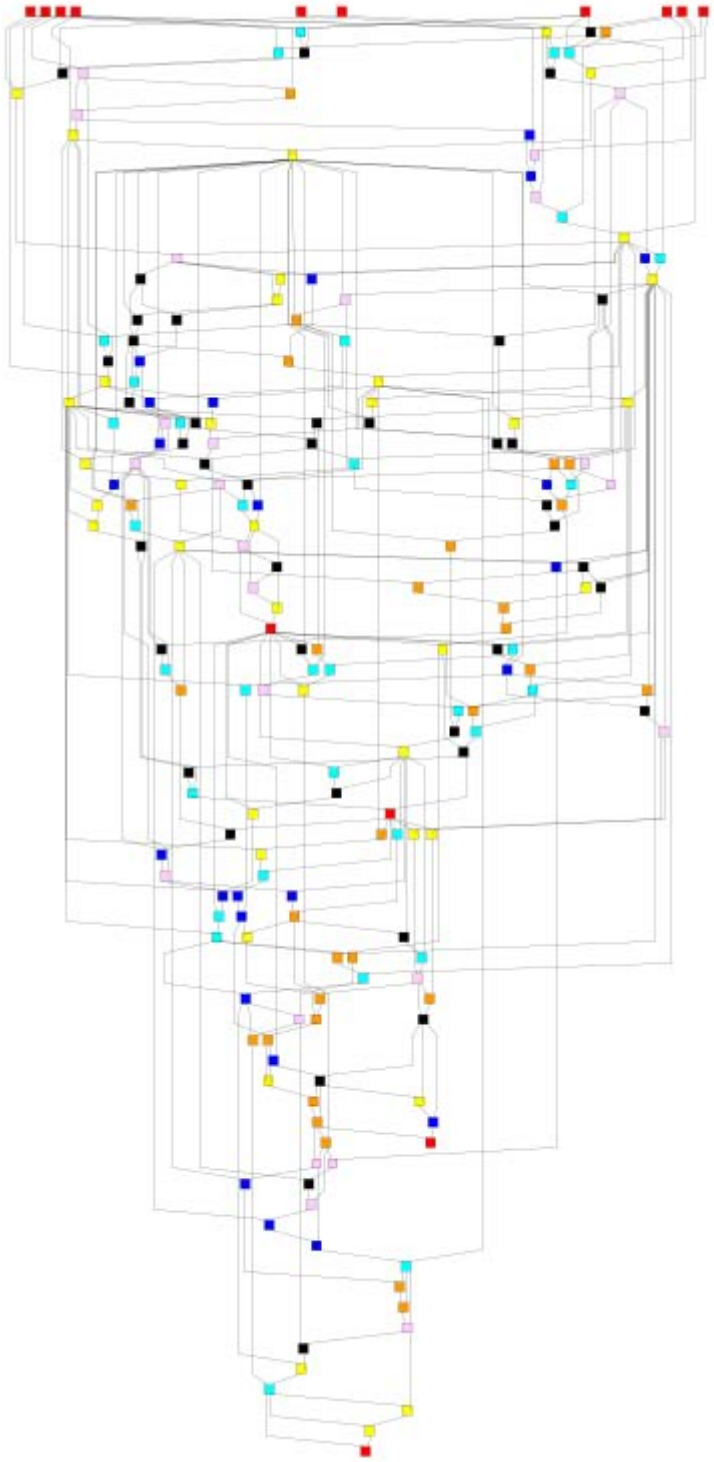


Figure 4.22: c10448 variant with most effective hiding.

Figure 4.23: c3315555 variant produced from 375 obfuscation iterations.

excessive search time. When execution was terminated component search size had decreased to components composed of 198 gates.

4.6 Multilevel Blur Experiments

We performed Multilevel blur experiments using selection strategies outlined in Table 3.4. For all experiments the initial blur level is level three and reduces when a blur attempt fails. Blurring on a specific replacement gate is terminated once a successful blur is achieved or once failed attempts occur on levels one, two and three. The number of blurs attempted is compared with the number of iterations performed with the COS.

4.6.1 c6288 Blurring Results. Applying Multilevel Blurring using the identified boundary blurring experiment has no effect on c6288. Circuit topology prevents any blurring using this strategy. With max fan-out strategy a total of 463 successful multilevel blurs occur. In our experiment an OR gate was chosen as the random replacement gate type, the circuit increased to 11382 gates and 100% component hid-

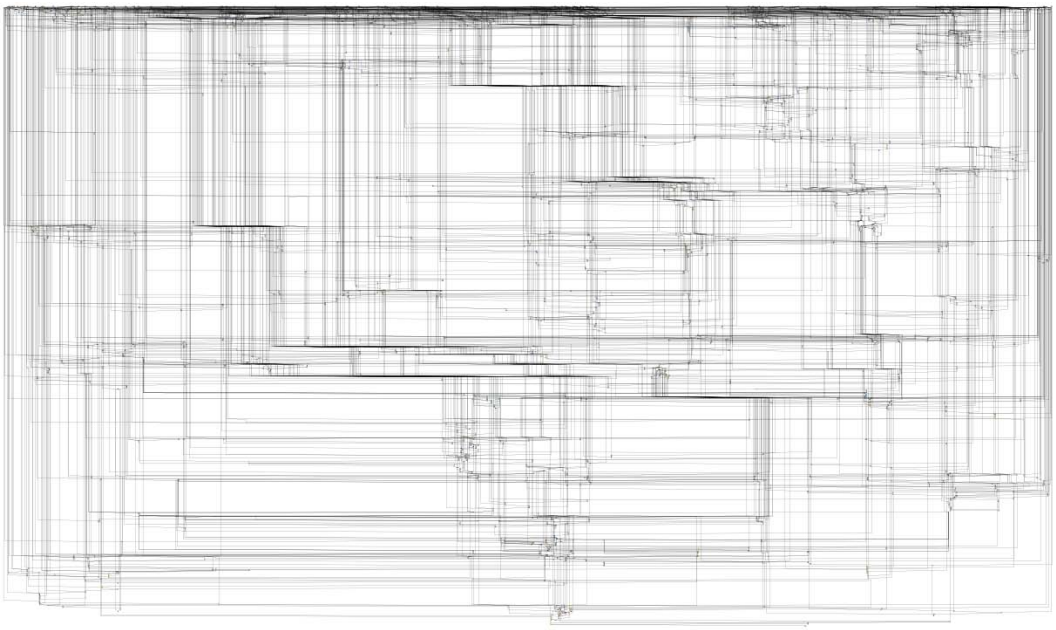


Figure 4.24: c70281374 variant produced from 500 obfuscation iterations.

ing is achieved. Figure 4.25 shows the resulting gate level diagram of the identified boundary experiment and Figure 4.26 shows the circuit variant after the max fan-out strategy is executed.

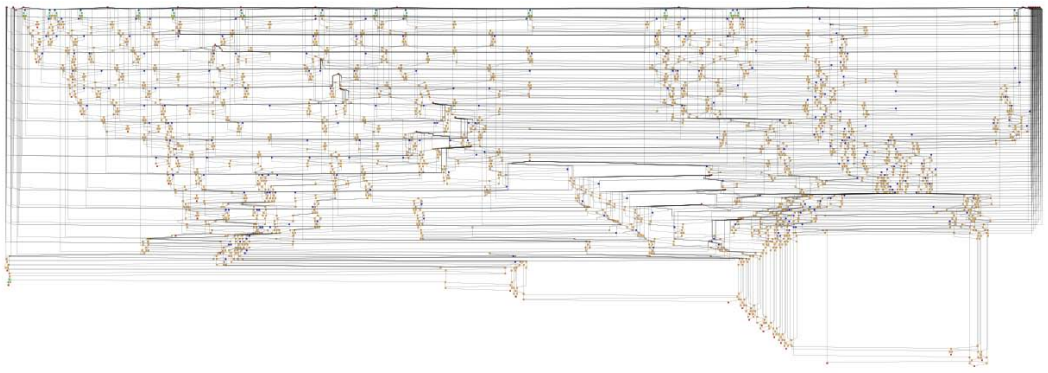


Figure 4.25: c6288 after apllication of identified boundary multilevel blurring. No changes are made to the circuit.

4.6.2 c10448 Blurring Results. Performing multilevel blurring on identified boundaries results in eight identified boundaries and four successful blurs. Of the four blurs, three are level one and the remaining blur is level two producing a circuit

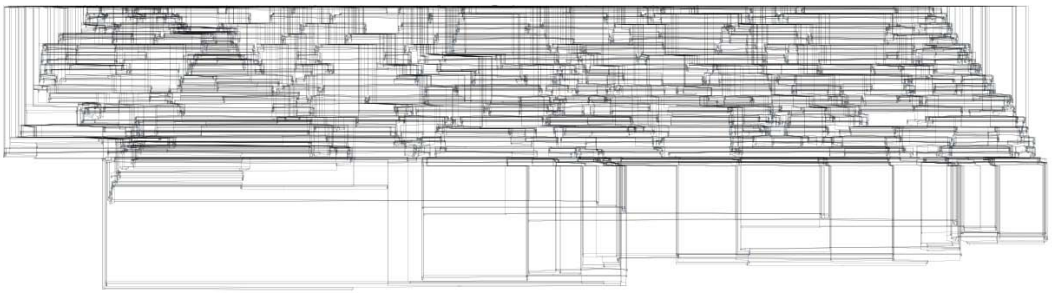


Figure 4.26: c6288 after application of max fan-out multilevel blurring.

with 337 gates or 480% larger. Figure 4.27 shows the circuit produced. Searching the circuit variant for components results in no identified components.

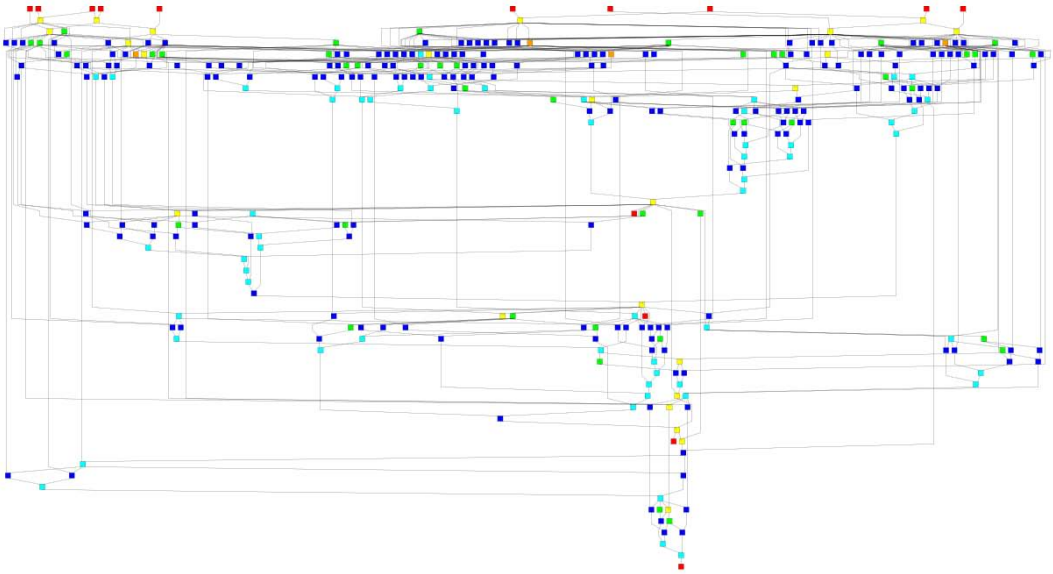


Figure 4.27: c10448 after application of identified boundary multilevel blurring.

4.6.3 c3315555 Blurring Results. Initially the circuit under investigation contains 588 gates. After we apply multilevel blurring the circuit variant contains 1187 gates, an increase of 101.9% increase. From the 36 identified boundary gates, 11 were successfully blurred. Figure 4.28 shows the circuit before blurring is applied and Figure 4.29 shows the results of the 11 successes. In these circuit representations, it is possible to see multilevel blurring does not merge disconnected circuits. When

the variant is provided to the component identification tool, zero components are identified.

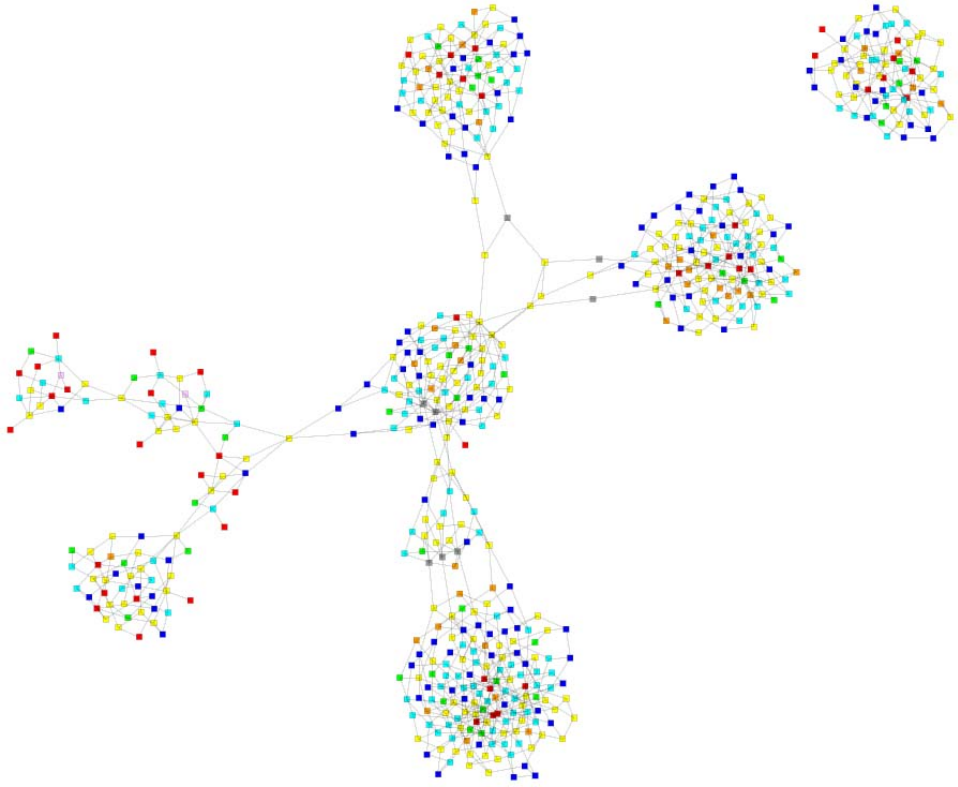


Figure 4.28: Circuit c3315555 before application of boundary blurring.

Max fan-out results in four gates identified for blurring. Multilevel blurring is successful for all four gates and proved successful in defeating the component identification tool. When the circuit variant is provided to the component identification tool zero components were identified.

4.6.4 c70281374 Blurring Results. There are 78 identified boundaries in c70181374. When applying multilevel blurring, 17 boundary gates are successfully blurred. This increased the circuit's size from 1374 to 2401 gates, a 74.7% increase. Attempting to identify components in the circuit variant was not successful. We terminated component identification due to extremely slow progress. At least one component should have been identified since multilevel blurring did not merge the

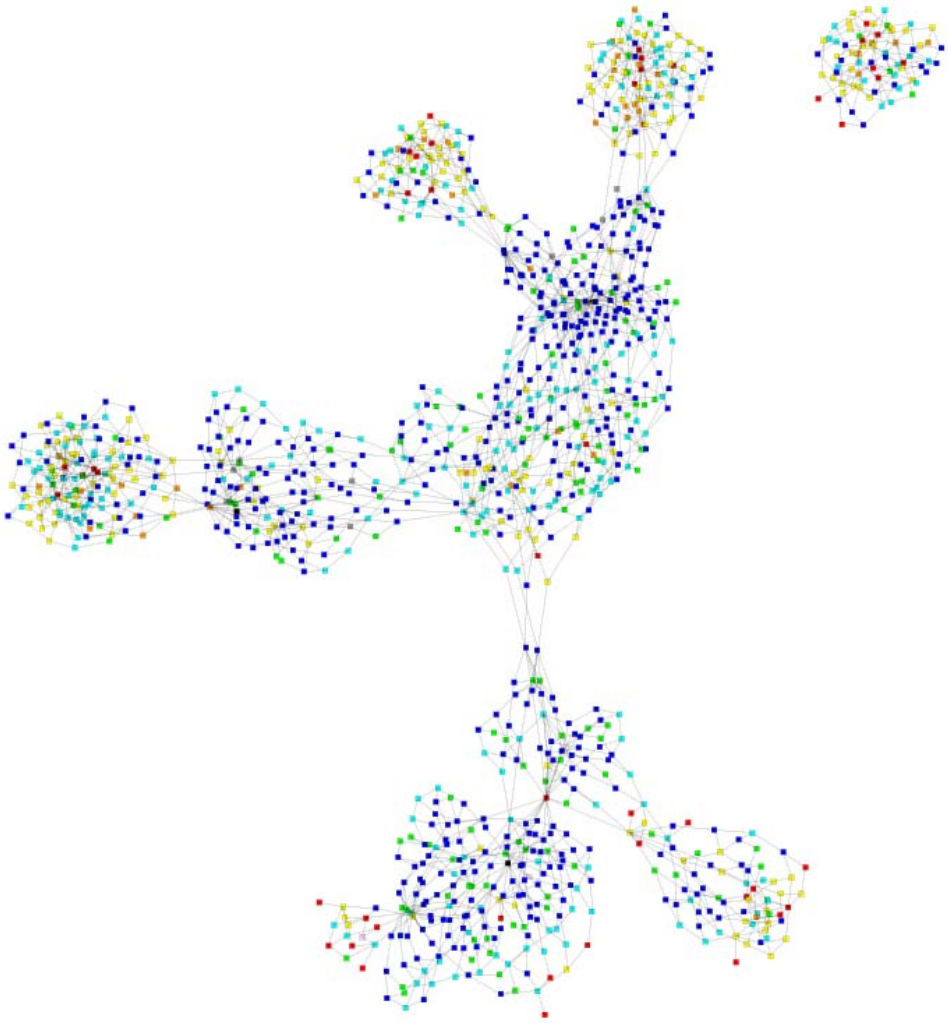


Figure 4.29: Circuit c3315555 after application of multilevel boundary blurring.

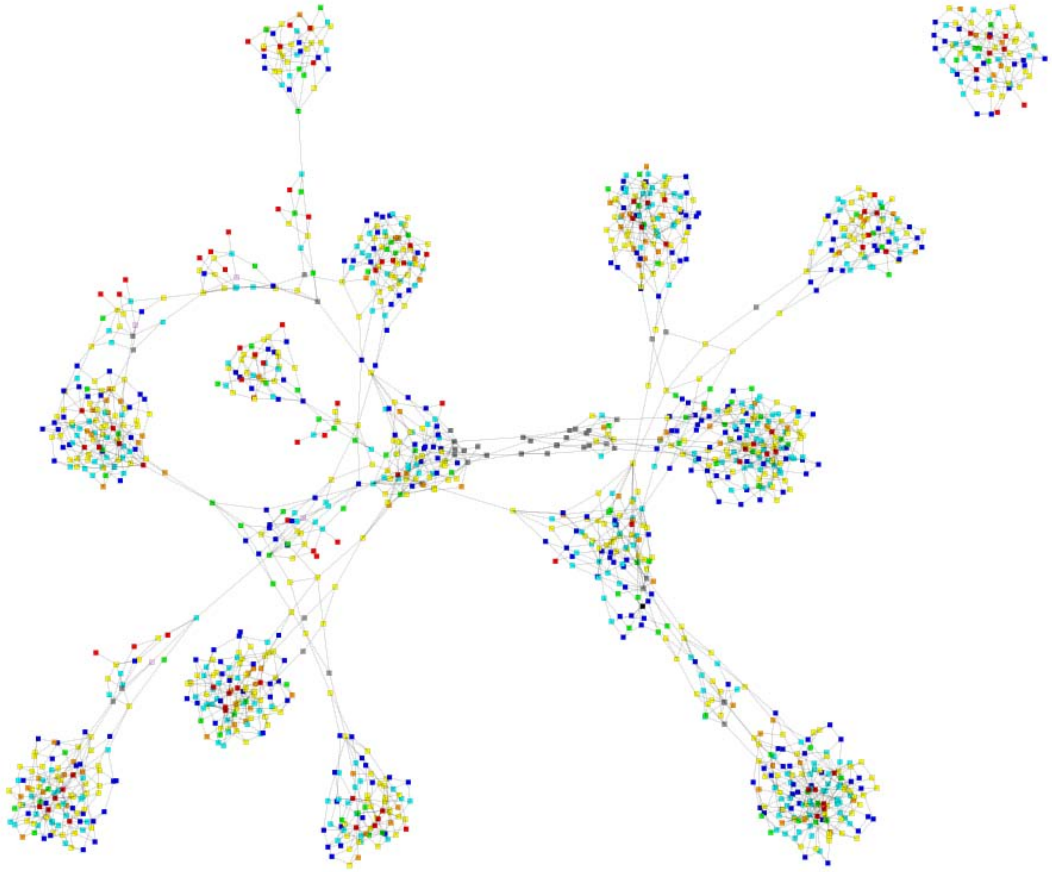


Figure 4.30: Organic view of c70281374.

case I component with the rest of the circuit. Figures 4.30 and 4.31 show c70281304 before and after blurring.

Max fan-out resulted in three identified gates and three successful blurs. The circuit variant size increased only 3.5% with 17 identified components.

4.7 Don't Care Blur Experiments

We performed Don't Care blur experiments using the identified boundary blur experiments. Boundaries of identified components are recorded and a Don't Care blur is performed on each boundary gate. The replacement gate is selected randomly as discussed in Section 4.1.3. The number of blurs successfully attempted is compared with the number of iterations performed with the COS.

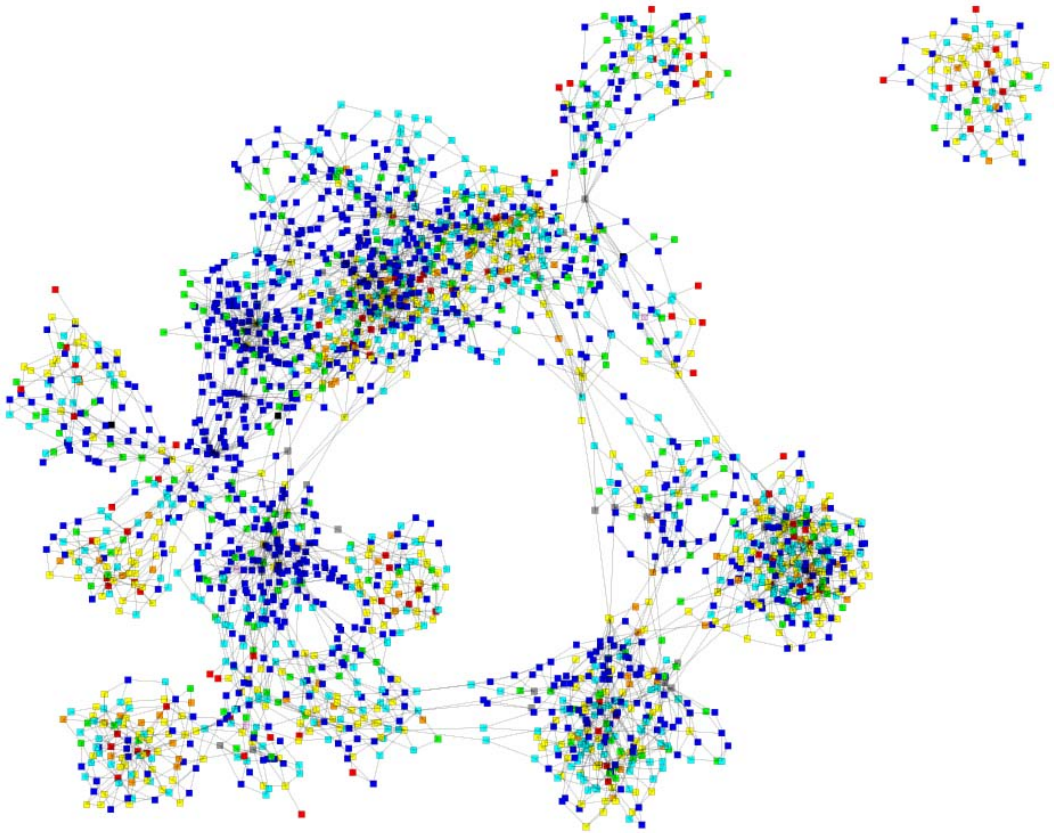


Figure 4.31: c70281374 after identified boundary multilevel blurring.

4.7.1 c6288 Blurring Results. There are 480 identified boundary gates in c6288. This number results from two output gates on each of 240 components. All 480 gates were successfully blurred using Don't Care blurring, increasing the circuit's gate size from 2448 to 7052 gates, a 188% increase. Performing component identification on the blurred circuit results in 100% component hiding. Organic circuit graphs are shown in Figure 4.32 and 4.33.

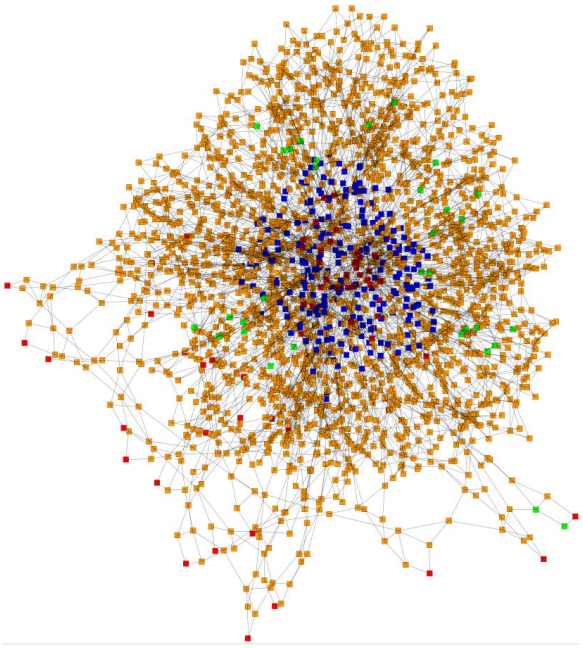


Figure 4.32: Organic layout of c6288 before application of boundary blurring.

4.7.2 c10448 Blurring Results. There are eight identified boundary gates in c10448. Each of the c17 circuits contribute two boundary gates and four gates from the four bit permutation circuit. Blurring was only successful on each c17 output gate increasing the circuit in size from 58 to 121 gates, a 108% increase. Component hiding, in this case, was not 100% effective. The 4-bit permutation circuit was still identifiable. However, the two c17 circuits were not. Random connection of the additional don't care input created a merge between the two c17 circuits. Figure 4.34 shows the results after boundary blurring is applied and Figure 4.35 shows the remaining circuit after identification is ran on the blurred circuit.

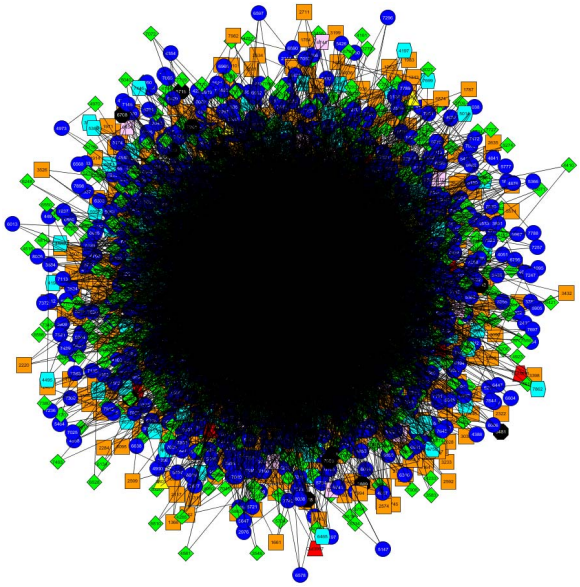


Figure 4.33: Organic layout of c6288 after application of don't care boundary blurring.

4.7.3 c3315555 Blurring Results. There are 36 identified boundary gates in c3315555, where each boundary gate is a component output gate. When applying blurring to each boundary gate, 19 of 36 are blurred successfully increasing circuit size by 28.5% from 588 to 822 gates. Providing the blurred circuit to the component identification tool, as the circuit under investigation, resulted in 100% component hiding. This circuit highlighted additional properties of the Don't Care blur. In Figure 4.36, it is possible to see ten distinct circuit components. Component number ten is the case I component which causes the circuit graph to be seen as two disconnected graphs. Figure 4.37 shows the circuit graph after blurring is applied. The internal components are not as easily identified and the circuit graph has been merged into a single circuit.

4.7.4 c70281374 Blurring Results. There are 78 identified boundaries in c70281374 circuit. When applying Don't Care blurring, 40 boundary gates were successfully blurred. This increased the circuit's size from 1374 to 1859 gates, a 35.3% increase. As with multilevel blurring, we terminated component identification due to slow progress. However, the case I component is merged with the remaining

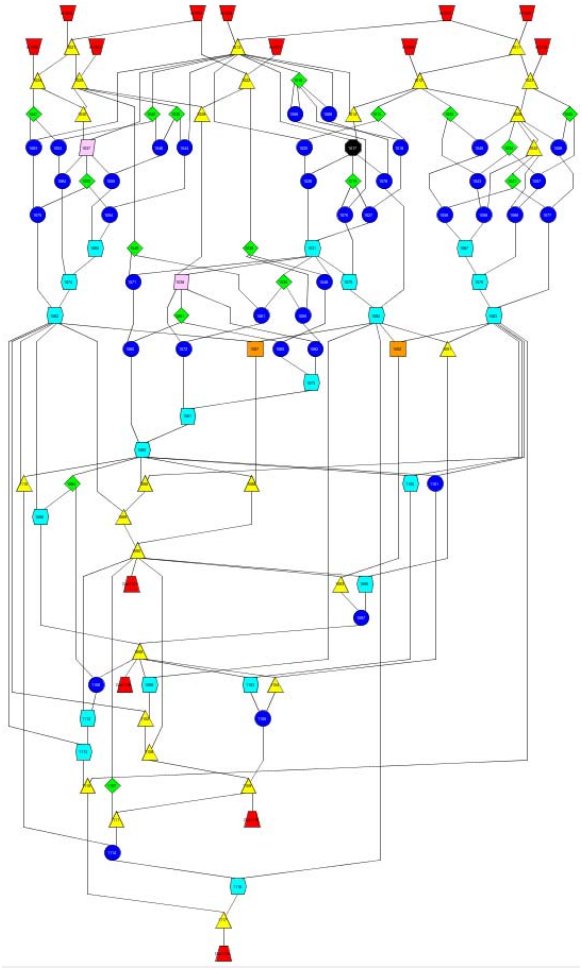


Figure 4.34: Circuit c10448 after application of identified boundary blur.

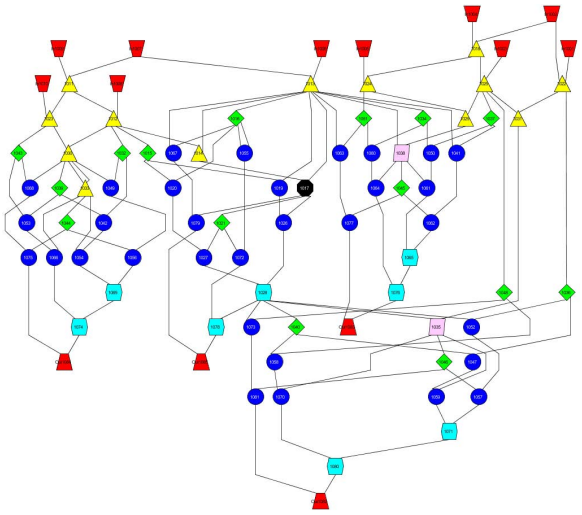


Figure 4.35: Remaining c17 circuits merged as a result of the Don't Care blur.

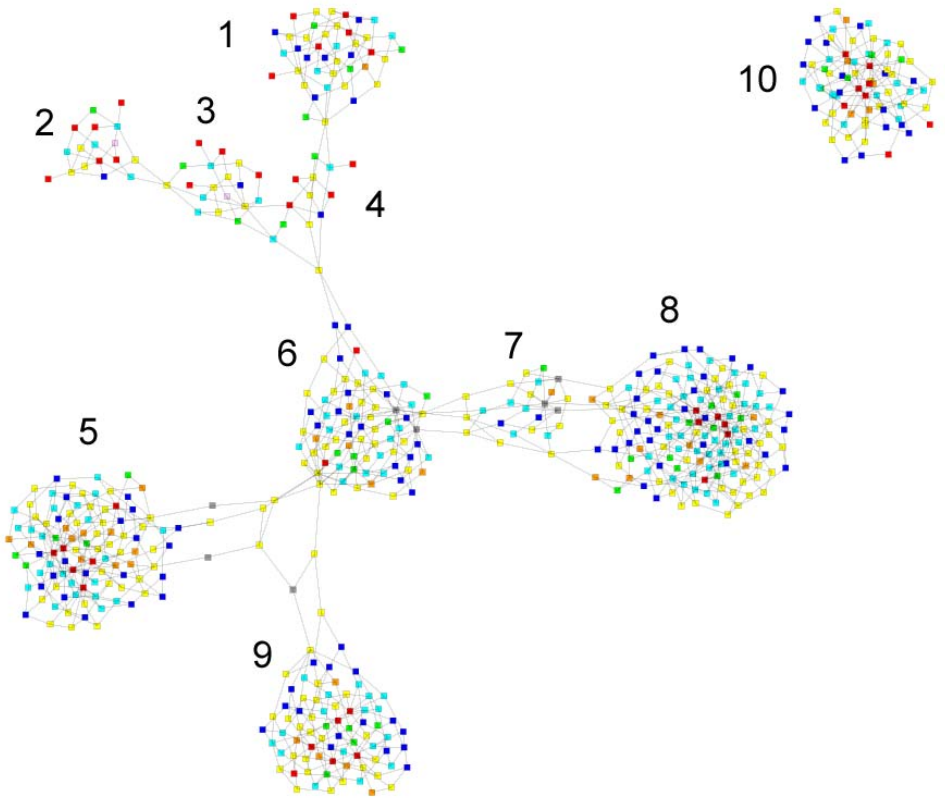


Figure 4.36: Circuit c3315555 before boundary blurring is applied. Individual components are relatively easy to identify using organic graph representation.

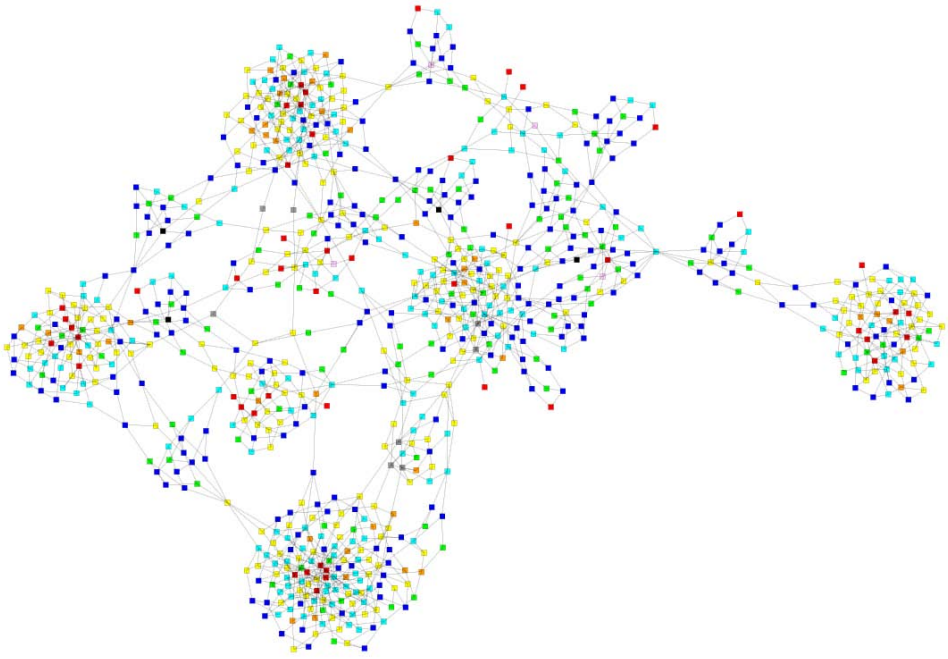


Figure 4.37: Circuit c3315555 after application of don't care boundary blurring.

circuit as a result of the random connections of don't care inputs. Figure 4.38 shows blurring results.

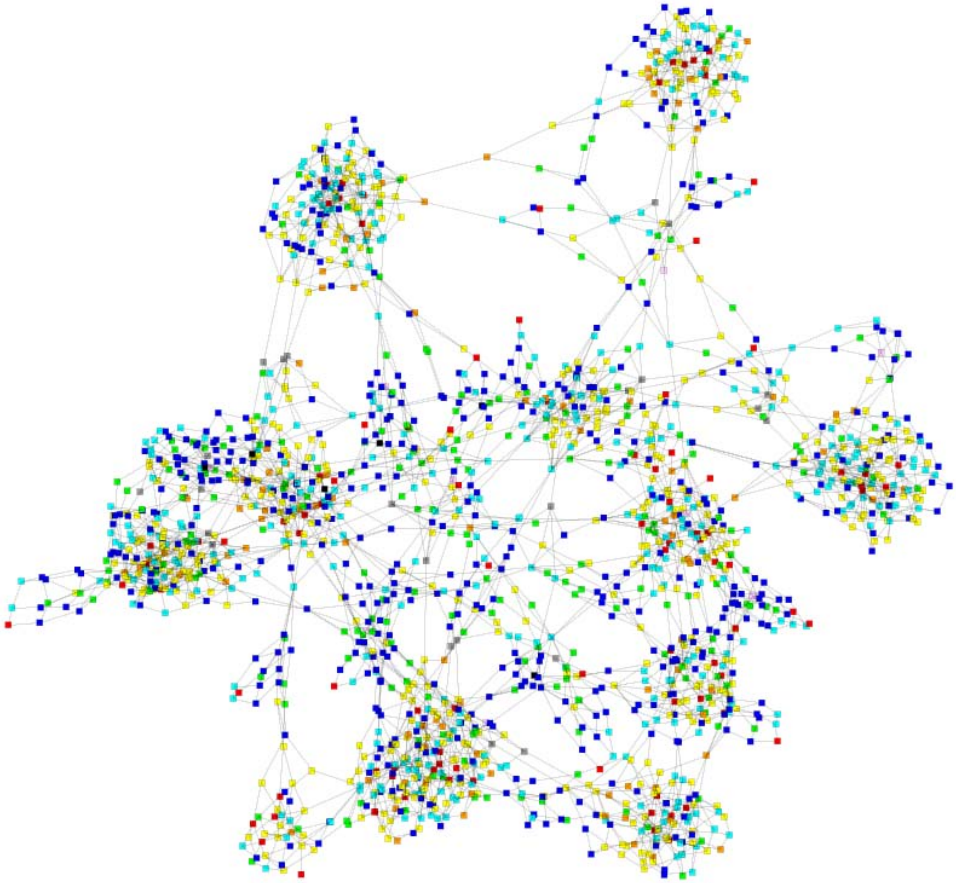


Figure 4.38: Circuit c70281374 after application of don't dare boundary blurring.

4.8 Variant Production Time vs. Components Identified

Producing circuit variants with current selection and replacement strategies is a very time consuming task. The average time to create a circuit variant using a random two gate selection and three gate replacement was 120.5 minutes and the median was 23 minutes. Performing a random two gate selection and four gate replacement averaged 440.43 minutes with a median of 243 minutes. Using blurring strategies for variant production of our benchmark circuits takes less than 50.58 minutes of which 40.58 minutes is time required to perform boundary identification. With knowledge

Table 4.14: Average times, in minutes, to produce circuit variants.

Circuit	Sel. 2 Rep. 3	Sel. 2 Rep. 4	Multilevel	Don't Care
c6288	391.6	243	6.5	4
c10448	< 2	75.3	< 1	< 1
c3315555	10.6	708	< 1	< 1
c70281374	76.8	1198	< 1	< 1

Table 4.15: Number of components identified in circuit variants.

Circuit	Sel. 2 Rep. 3	Sel. 2 Rep. 4	Multilevel	Don't Care
c6288	227.8	4	0	0
c10448	.6	0	0	1
c3315555	3.4	0	1	0
c70281374	7.6	0	0	0

of the circuit boundaries blurring requires ten minutes or less. Table 4.14 provides detailed performance times and Table 4.15 shows the number of components identified by the component identification tool.

4.9 Analysis of Candidate Component Identification Time

To characterize the time required for component identification, candidate subcircuits of the ten ISCAS-85 circuits were enumerated. The candidate sizes are listed in Table 3.4. During the identification process, we logged the identification time of each candidate. As expected, when circuit size increased, identification time also increased. Figure 4.39 shows the average enumeration time after five rounds of candidate subcircuit identification.

To understand candidate subcircuit identification time, a histogram for all identification times was created. $log_{10}(time)$ was plotted for easier visualization. Figure 4.40 is a histogram of identification times. A total of 465500 measurements were logged during candidate enumeration. The data shows expansion time for each circuit gate and not all expansions produce a candidate circuit. If at any point during the recursive expansion of the identification algorithm rule two or rule three can not be satisfied the subgraph is discarded. However, the measurement is logged. The

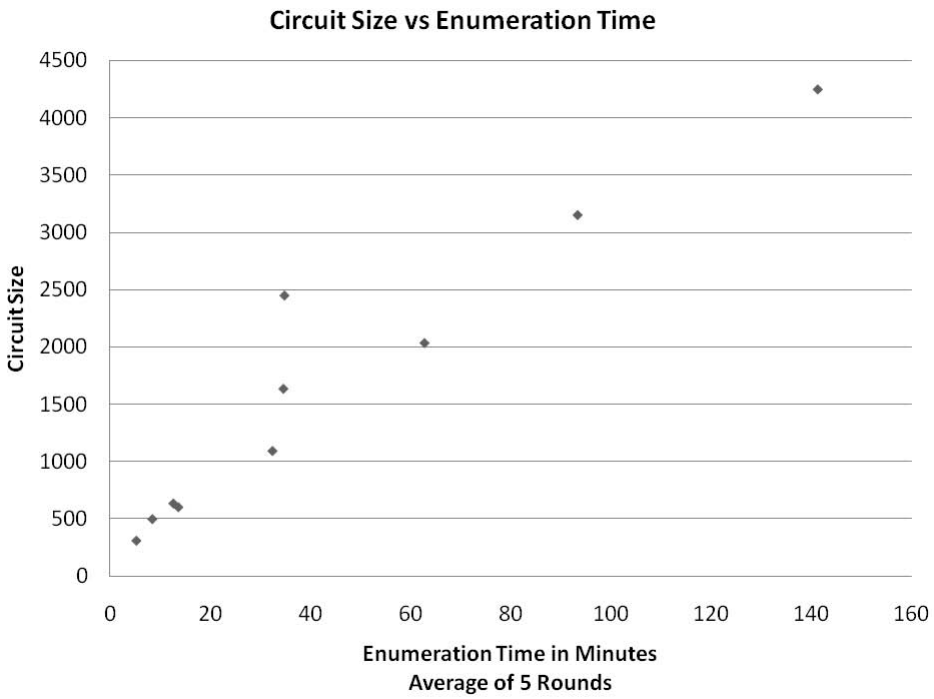


Figure 4.39: The average time required to enumerate all levels of candidate subcircuits in minutes. The graph shows circuit size versus time to enumerate all candidates.

median identification time is 2.312ms and the mean is 61.594ms. These time are only valid for characterization levels. We provide individual histograms for each circuit in Appendix C.

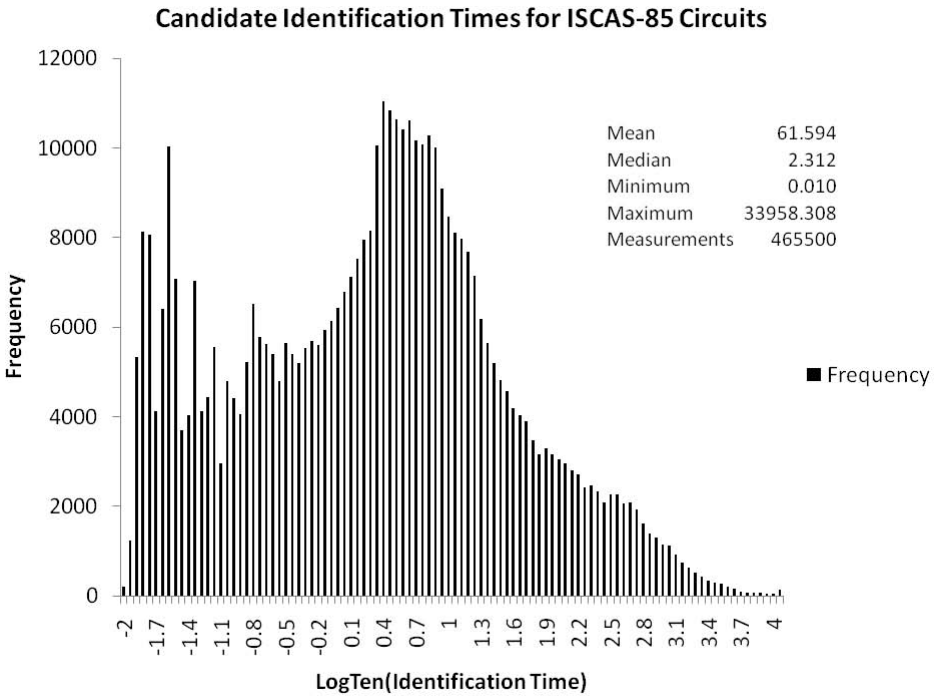


Figure 4.40: Histogram of candidate identification time for all ISCAS-85 circuits combined.

V. Conclusions

This chapter discusses research contributions and future works for improving tool implementations.

5.1 Goals and Hypothesis

Because component identification provides a faster way of understanding a circuit, it is an important part of reverse engineering. However, component identification requires knowledge of a circuit's structure. The component identification tool along with a known component library will successfully identify components in larger circuits. This tool is most successful when circuits are composed of independent circuit components. Implementing additional identification heuristics will improve component identification in other circuits. Boundary blurring algorithms are an effective means of defeating the foundational enumeration algorithm of the identification tool and it is assumed these types of methods will defeat other identification tools.

5.2 Contributions

5.2.1 Multilevel Blurring. Multilevel blurring is a subcircuit replacement algorithm that defeats component identification. It is important since it shows the possibility exists for changing signals within a circuit and recovering them either internal or external to the original circuit boundary. If the assumption that other component identification tools use similar enumeration techniques is made, this blurring strategy is useful in increasing the time an adversary would spend discovering a circuit's unknown functionality.

5.2.2 Don't Care Blurring. Don't Care blurring is a second blurring algorithm which defeats component identification. This algorithm introduces additional inputs to a circuit component or entire circuit depending on its implementation. Just with multilevel blurring, if we assume other tools identify components using similar algorithms, this method of blurring will delay an adversary in determining a circuit's intended purpose.

5.2.3 Component Identification Tool. Component identification is possible on circuits constructed of independent circuit components. Candidate components are enumerated from circuits of interest and compared to a library of known components. Functions of circuits with large I/O space, which must be reversed by white box methods, are discovered and it is possible to estimate the time and effort necessary to complete these tasks. We use this tool for measuring effectiveness of circuit protection and with additional development this tool will become a significant Program Encryption Group tool.

5.2.4 BENCH File Creation Tool. Creating circuits by hand for the Circuit Obfuscation System (COS) is a tedious process. Initial circuit creation can take hours and is prone to error. With this tool, a simple graph is constructed using proper gate shapes and analyzed by the tool. The result is BENCH and shaped graphml files of the newly constructed circuit.

5.2.5 Shaped Graphml Graphs. Graphml files exported with shapes and colors representing specific gate types improves visual analysis of circuit graphs. When analyzing boundary blurring algorithms it is possible to visualize where clusters of new gates are added in circuit variants.

5.3 Future Works

5.3.1 Validate Implementation of Subcircuit Enumeration. The subcircuit enumeration algorithm used by the component identification tool needs validation. Collaborating with others using this algorithm or having an independent expert analyze the implementation is necessary. Applying subcircuit enumeration unmodified to a circuit being reversed engineered is not feasible however, this does not preclude the use of the techniques in practical application [15]. Through our component identification tool implementation discovery of useful algorithms were made. Performing operations such as input splitting or renumbering gate indices may increase the robustness

of the tool. These algorithms were partially implemented during tool development and require further application research.

5.3.2 Component Equivalence Checking. Candidate component equivalence is determined using truth table analysis. This is feasible only for circuits with small I/O space. Truth table analysis fails when trying to compare module one of c432 which is an 18 input component. Using other equivalence methods such as reduced order binary decision diagrams will make it possible to analyze components with larger I/O space. Improved equivalence checking is necessary for improving the identification tool.

5.3.3 Working with Module Library in Memory. When candidates match input and output size of a known library module, the known component is imported for analysis. Each module is stored in BENCH file format in a directory labeled by the I/O space of the circuit. In a circuit such as c6288, 240 circuit imports occur. Working with library modules in memory, by importing them at the start of component identification, would improve the performance of the identification tool and reduce system I/O operations.

5.3.4 Error Checking in BENCH File Tool. Time required to produce circuit BENCH files from scratch is significantly reduced using this tool. However, complete error checking is not implemented. A user may design a circuit and make gate connection errors. The BENCH file and graph files are produced, but execution errors occur when the COS or CIS attempts to utilize them. Error checking must be explored and implemented to make this a more robust tool.

Appendix A. Gate Legend

This appendix outlines shapes and colors used for identifying specific gates types in circuit graphs used throughout this thesis.

Table A.1: Gate symbols used in graphs.

Gate Type	Symbol
INPUT	
OUTPUT	
AND	
NAND	
OR	
NOR	
NOT	
XOR	
XNOR	
BUFFER	

Appendix B. Custom Circuit Gate Level Diagrams

This appendix shows 16 custom components created for developing our component identification tool and testing boundary blurring algorithms. These circuits were created using the process outlined in Section 4.3. We also include custom circuits composed from our component modules.

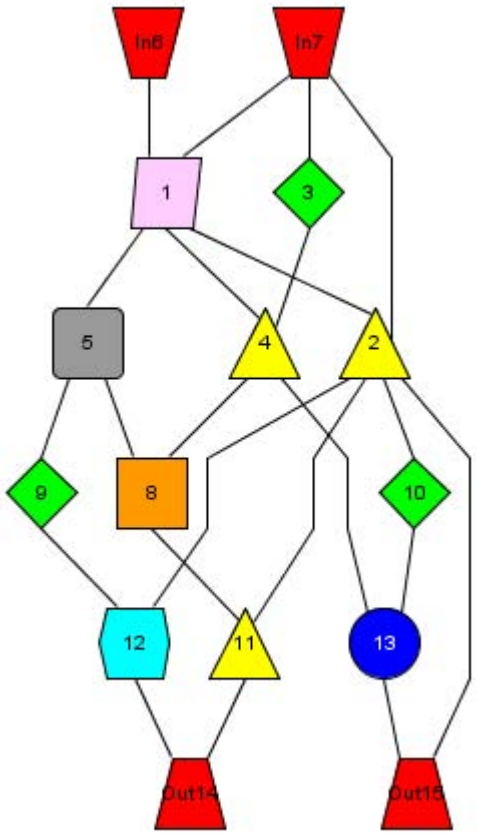


Figure B.1: Benchmark Circuit c2215

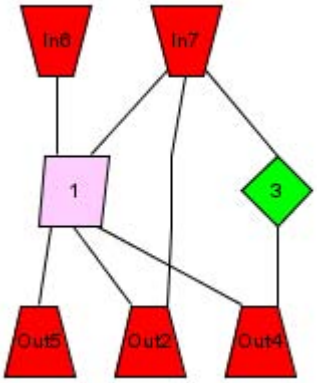


Figure B.2: Benchmark Circuit c237

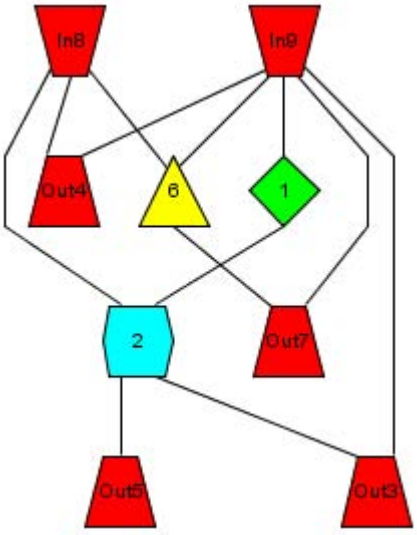


Figure B.3: Benchmark Circuit c249

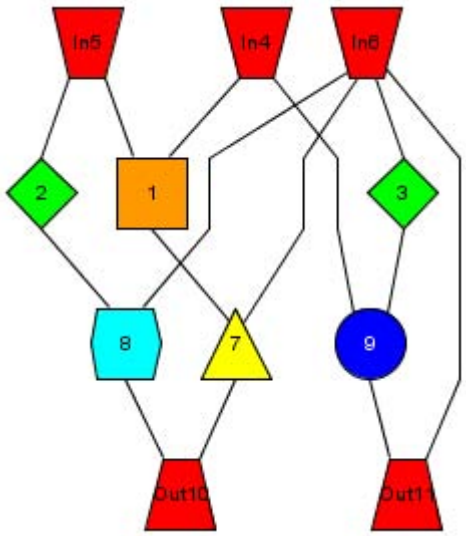


Figure B.4: Benchmark Circuit c3211

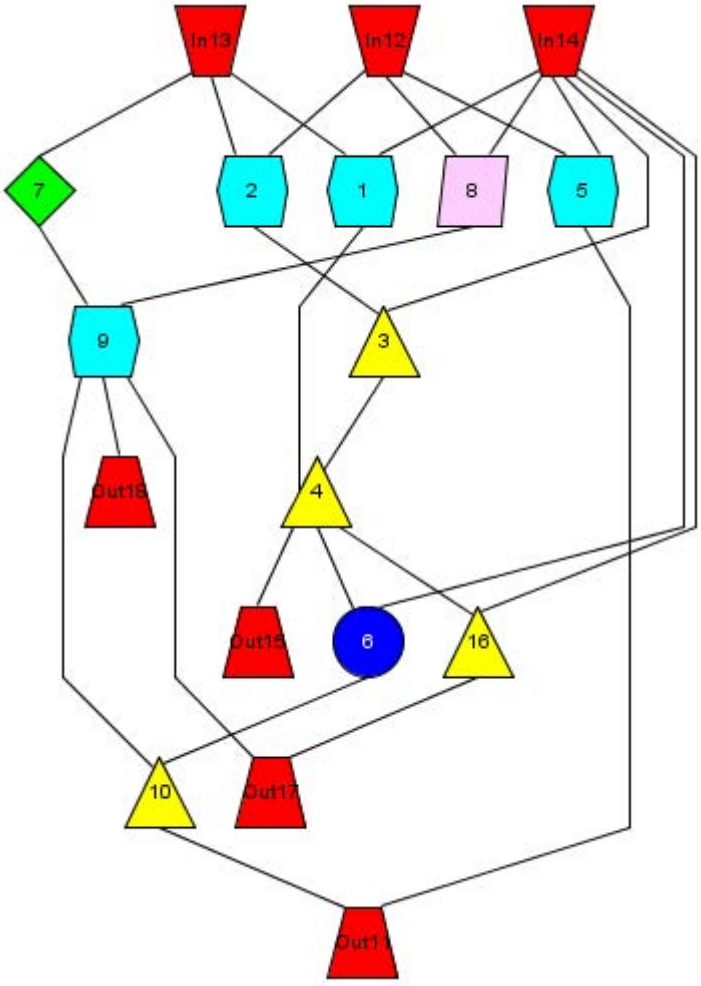


Figure B.5: Benchmark Circuit c3418

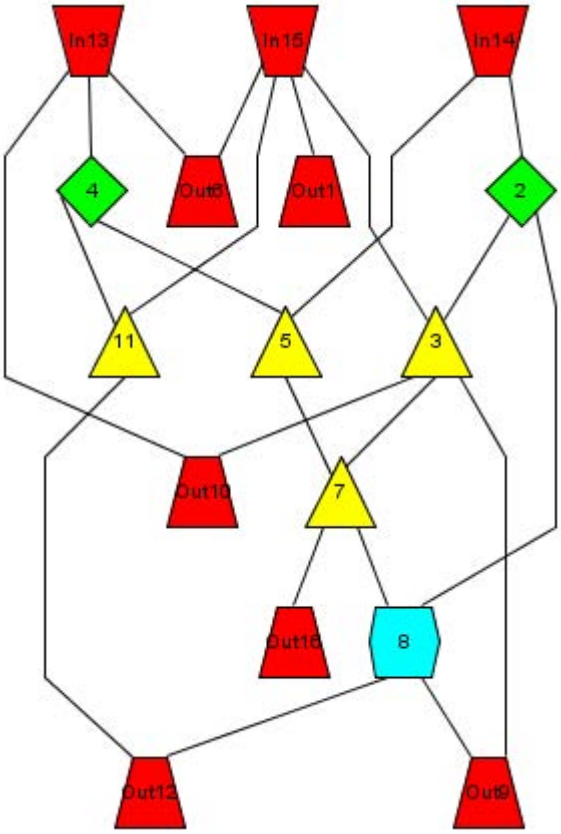


Figure B.6: Benchmark Circuit c3516

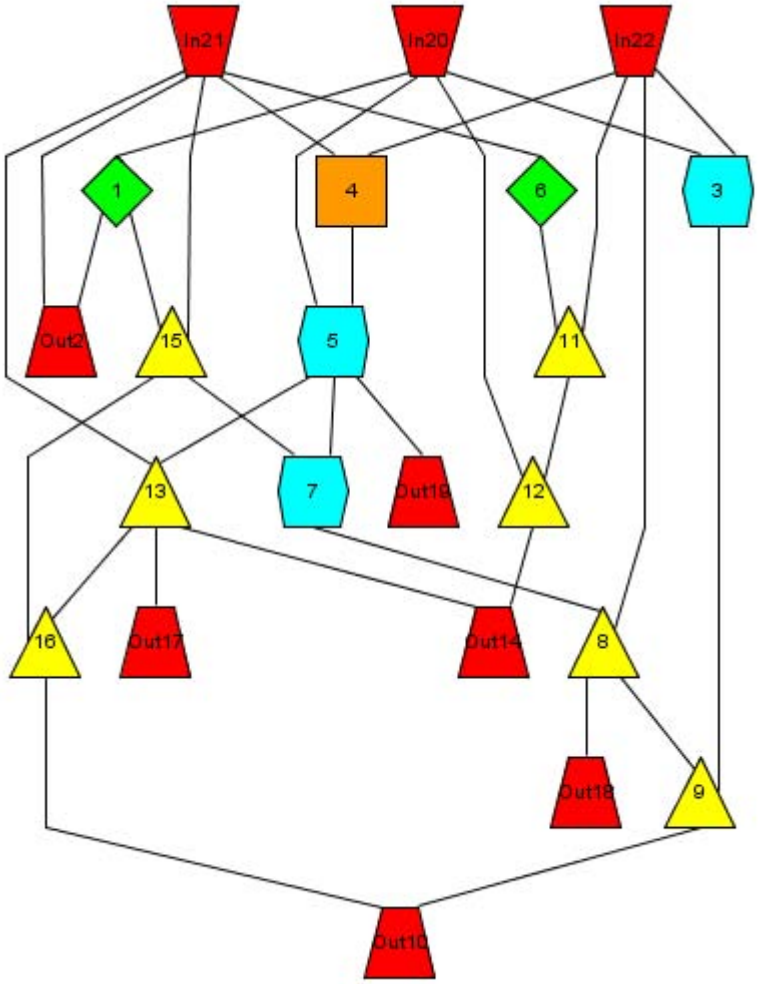


Figure B.7: Benchmark Circuit c3622

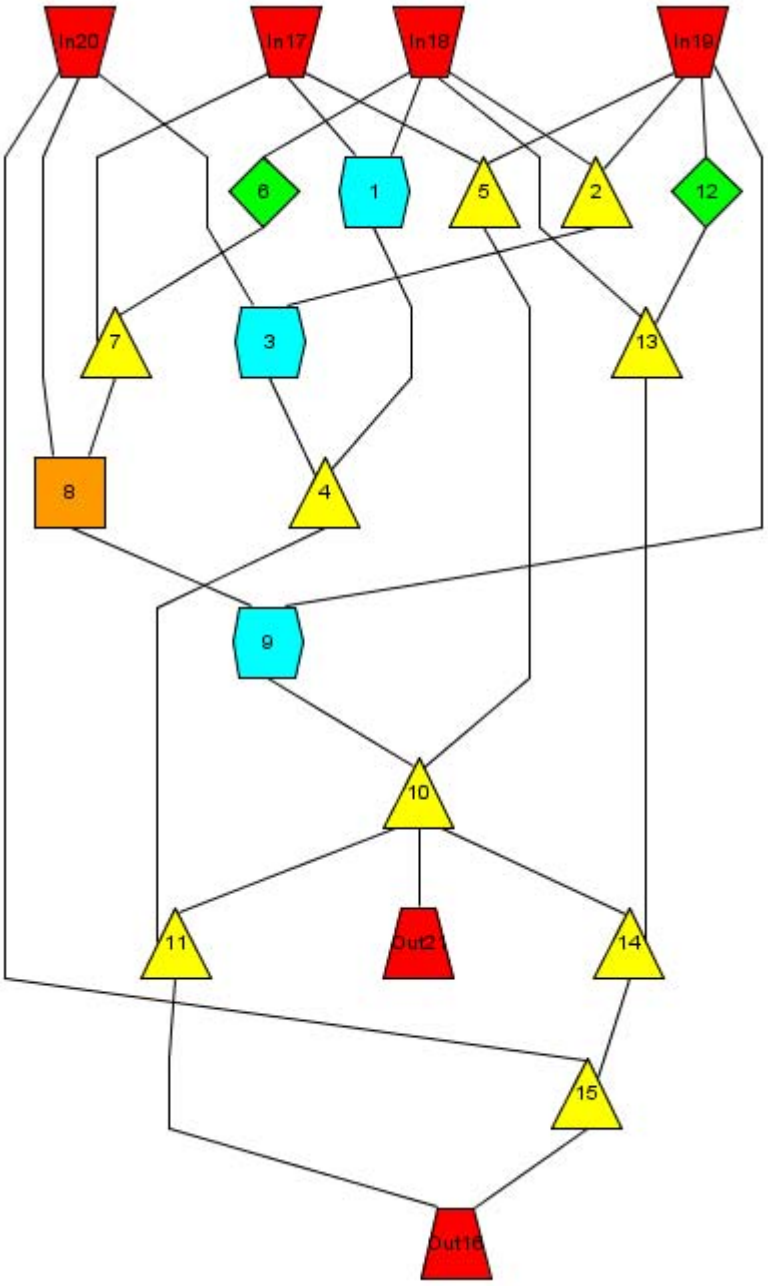


Figure B.8: Benchmark Circuit c4221

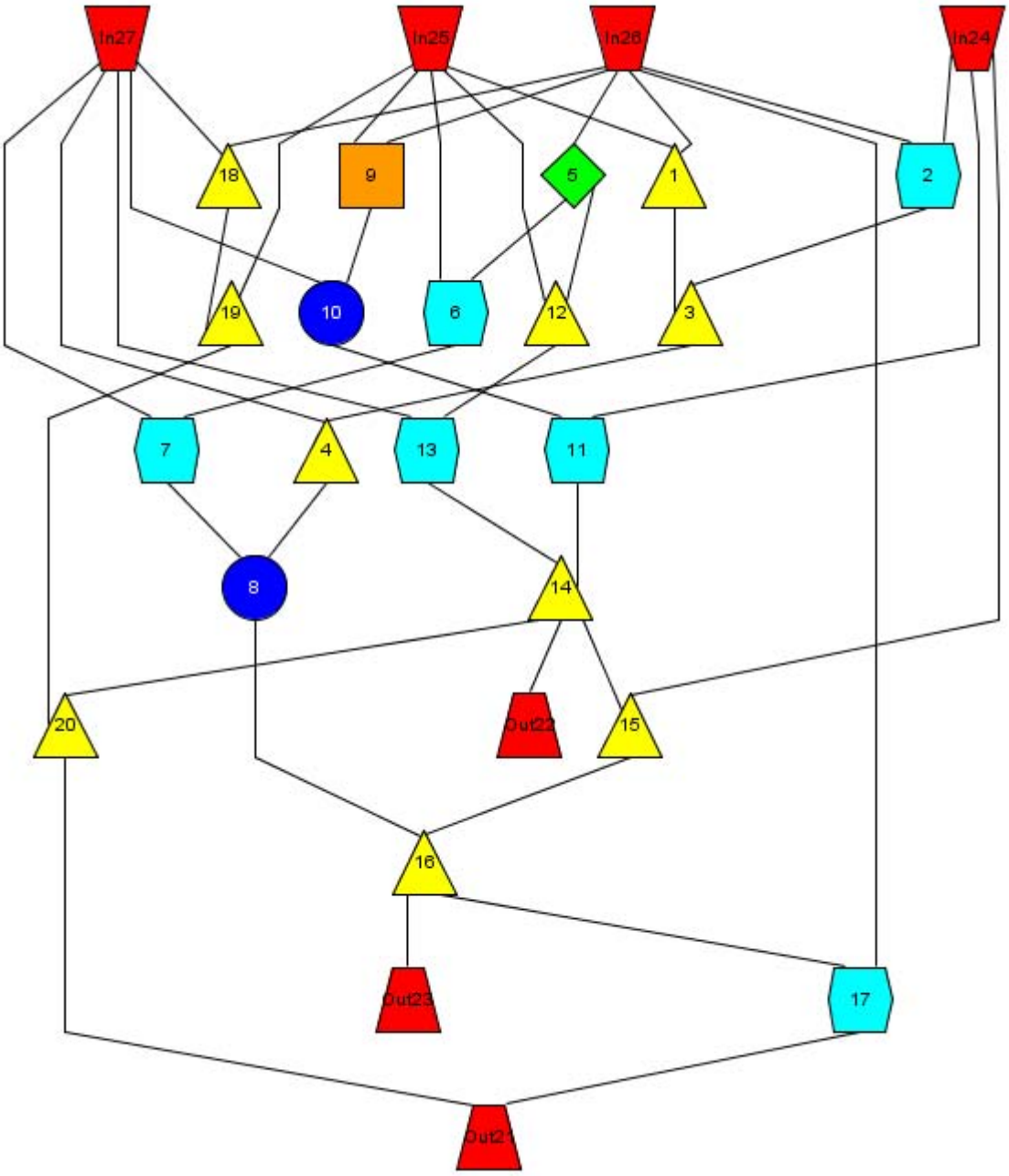


Figure B.9: Benchmark Circuit c4327

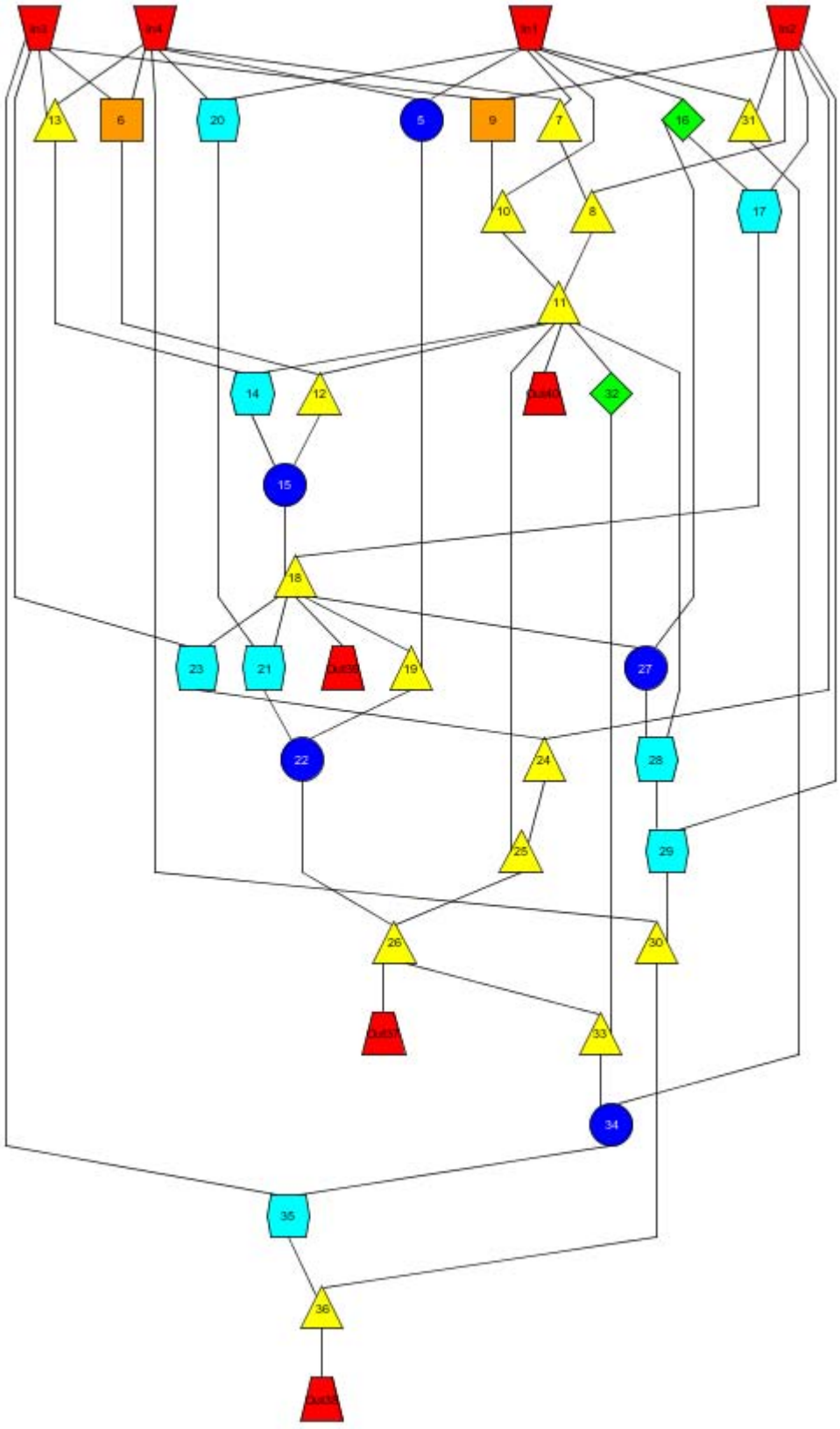


Figure B.10: Benchmark Circuit c4440

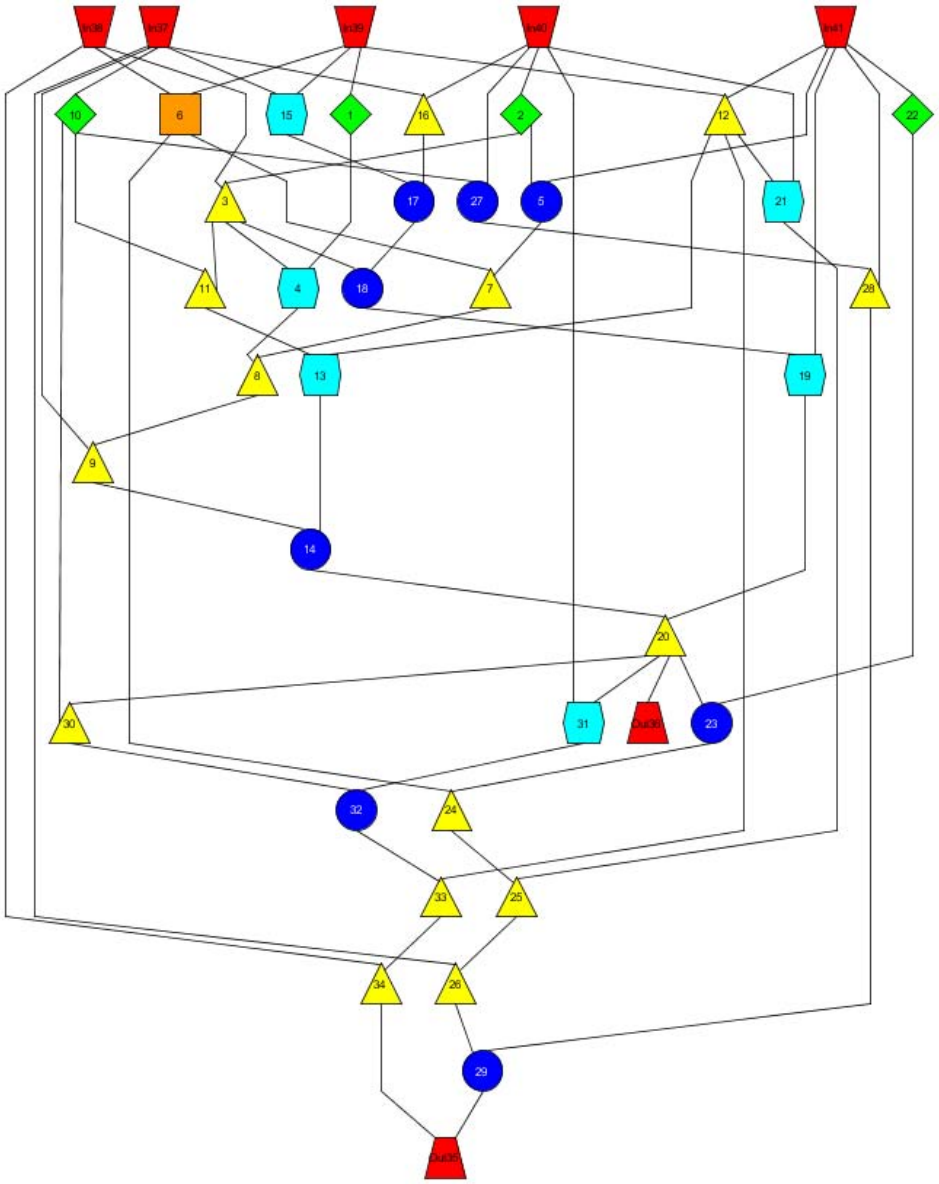


Figure B.11: Benchmark Circuit c5241

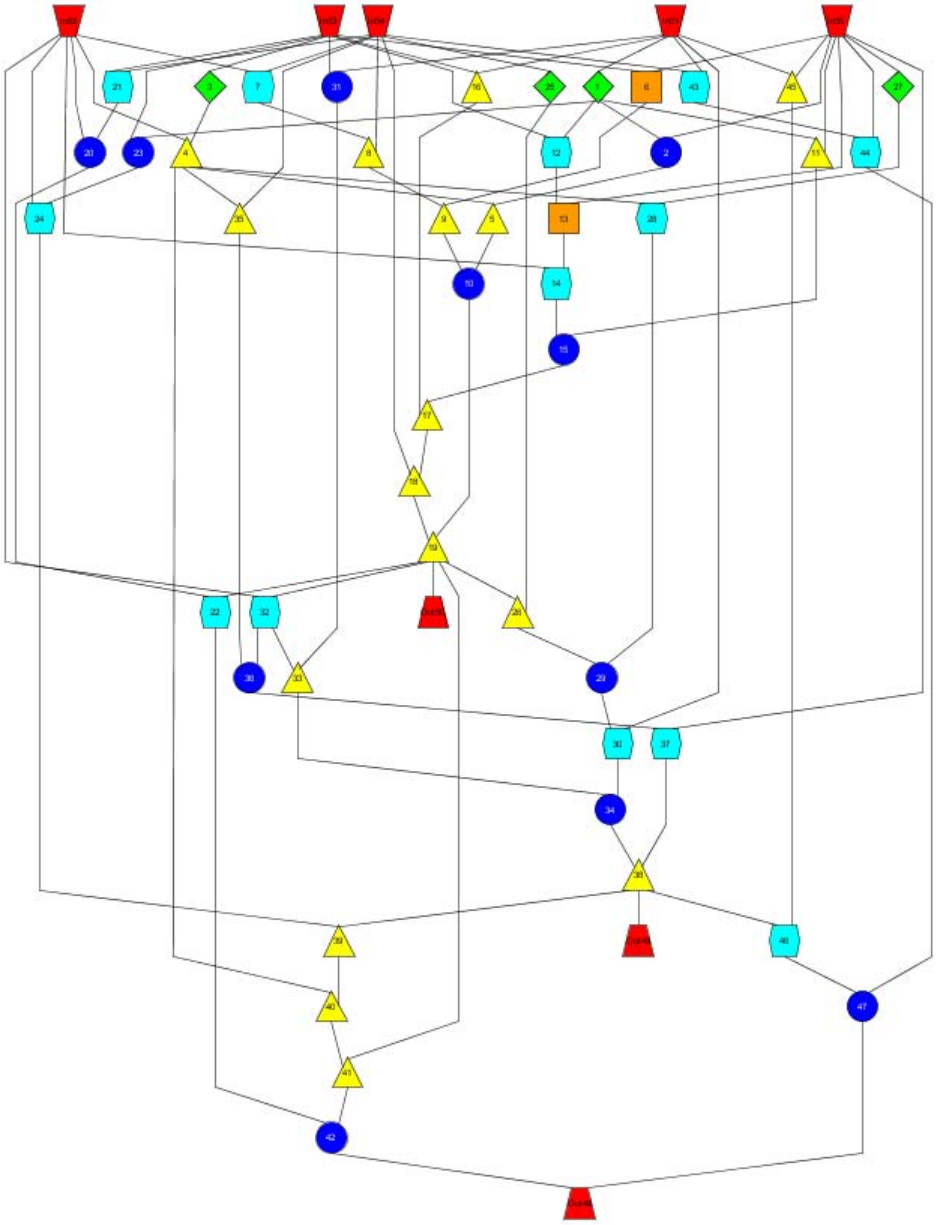


Figure B.12: Benchmark Circuit c5355

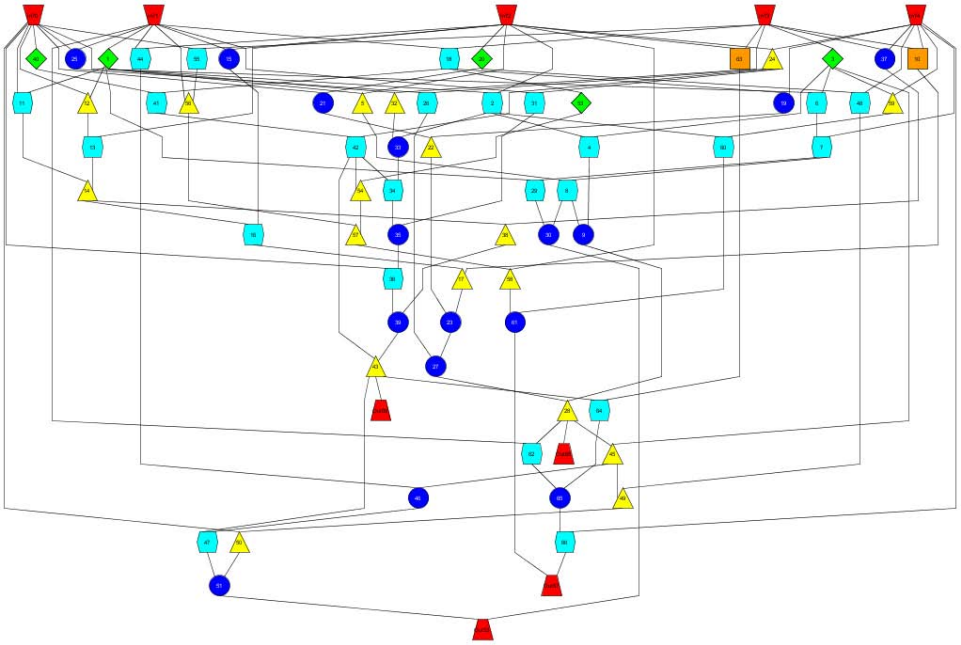


Figure B.13: Benchmark Circuit c5479

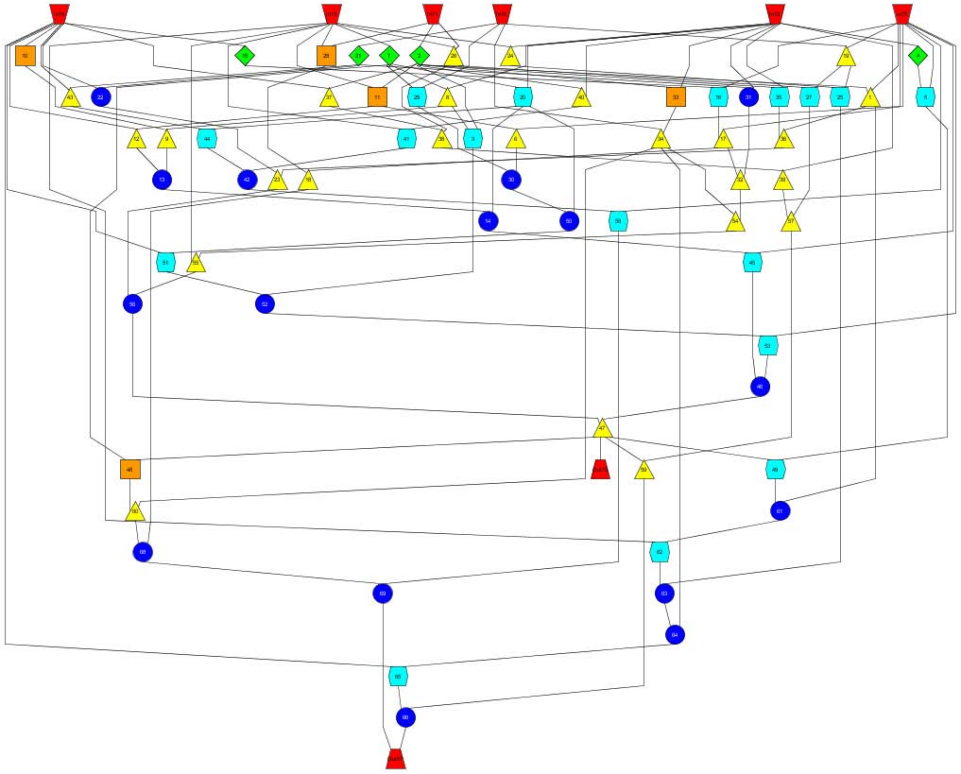


Figure B.14: Benchmark Circuit c6276

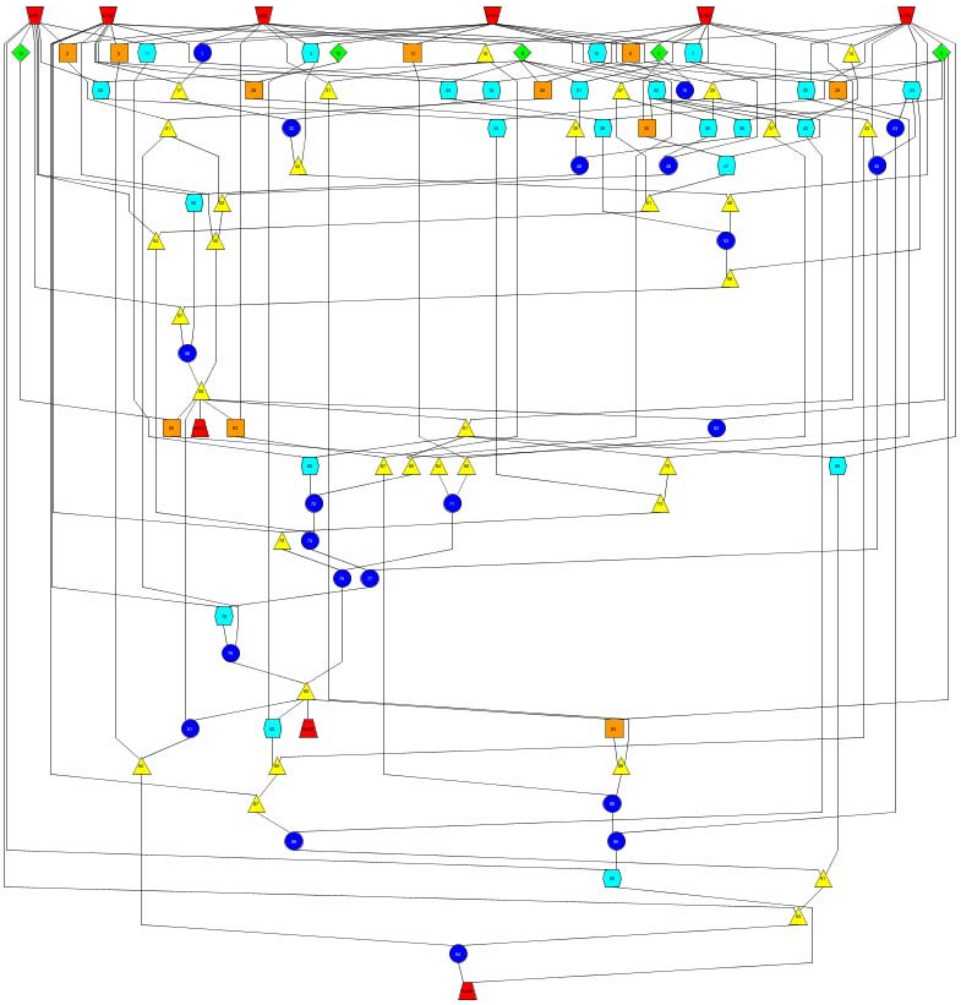


Figure B.15: Benchmark Circuit c63103

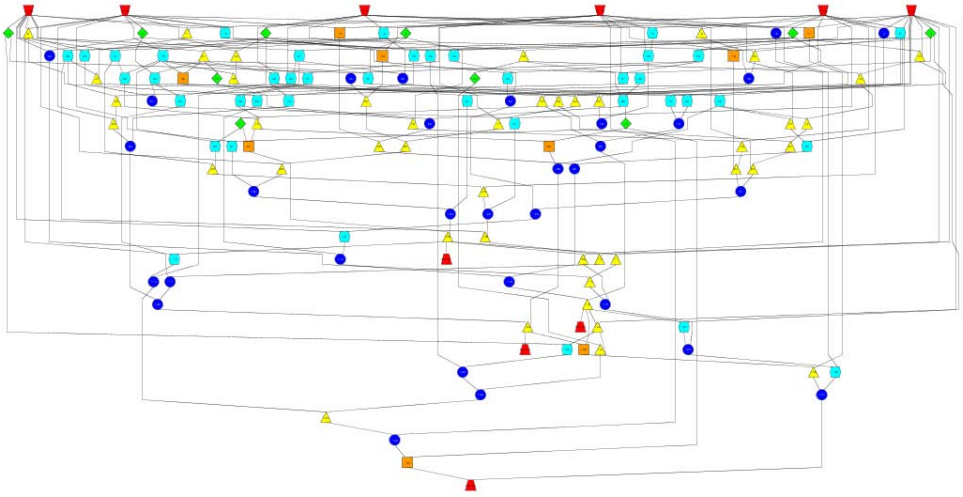


Figure B.16: Benchmark Circuit c64145

Figure B.17: Benchmark Circuit c182449

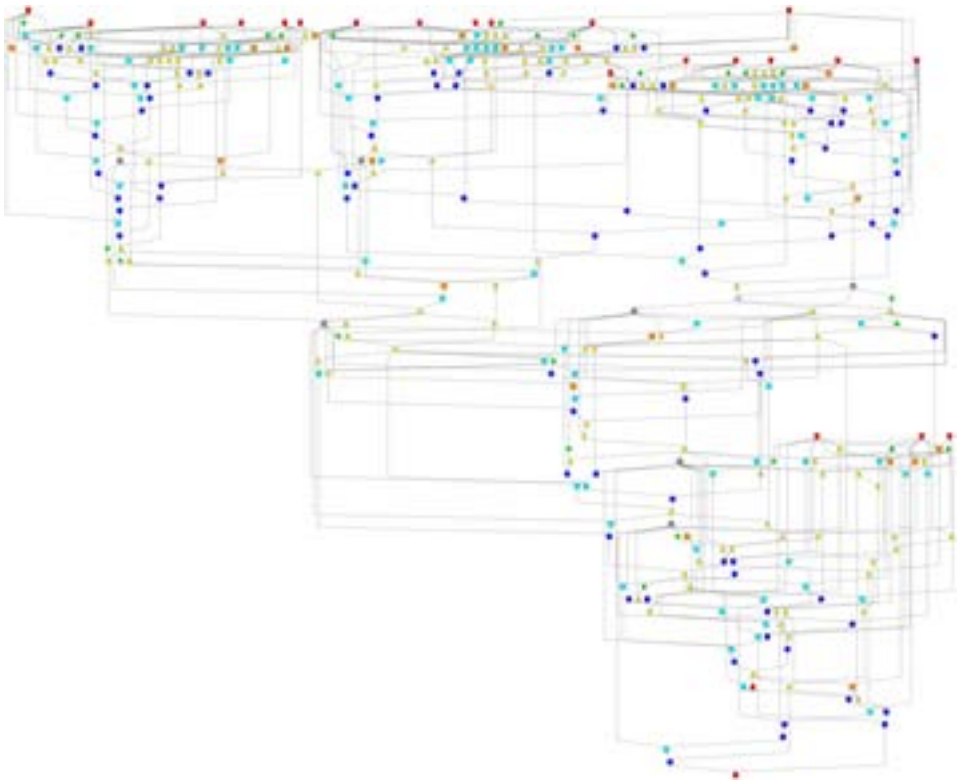


Figure B.18: Benchmark Circuit c212235

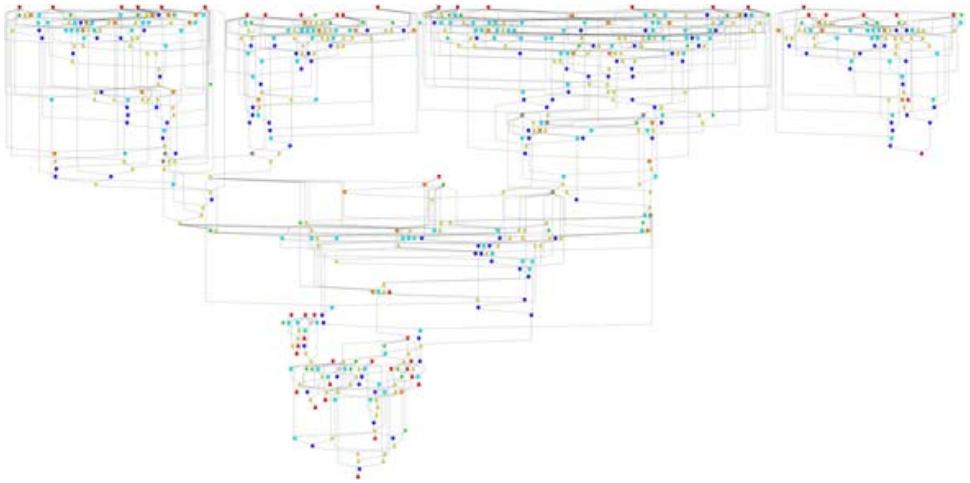


Figure B.19: Benchmark Circuit c3315555

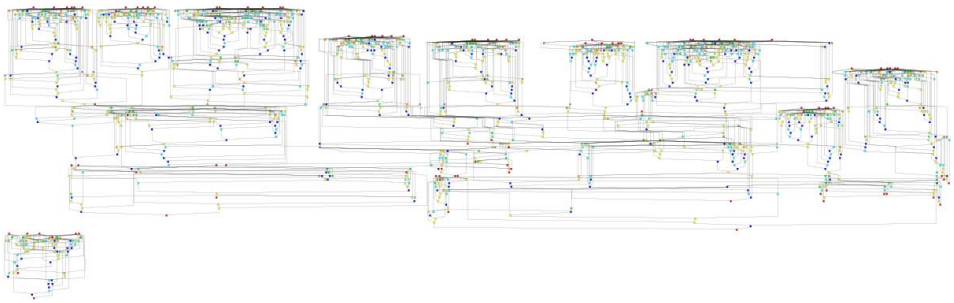


Figure B.20: Benchmark Circuit c70281374

Appendix C. Histograms of Candidate Component Identification Time

This appendix shows histograms of identification times for each ISCAS-85 circuit. The histograms include expansion time for every vertex while attempting to identify components of sizes specified in Table 3.2.

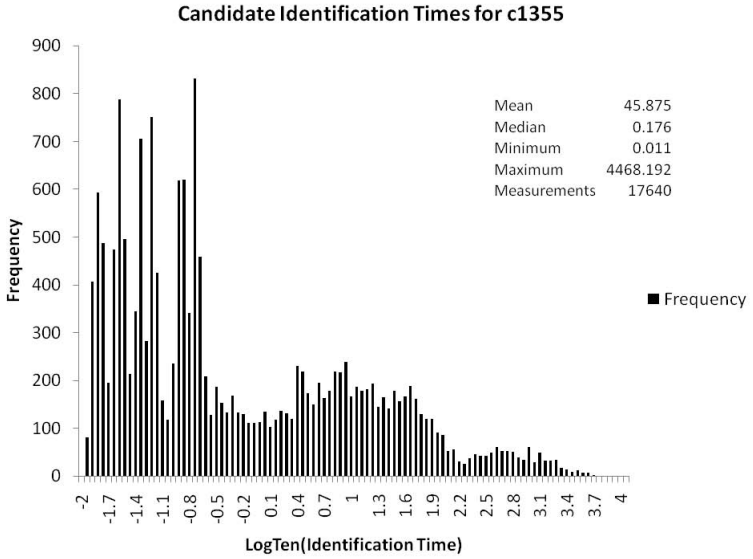


Figure C.1: Candidate identification times for circuit c1355.

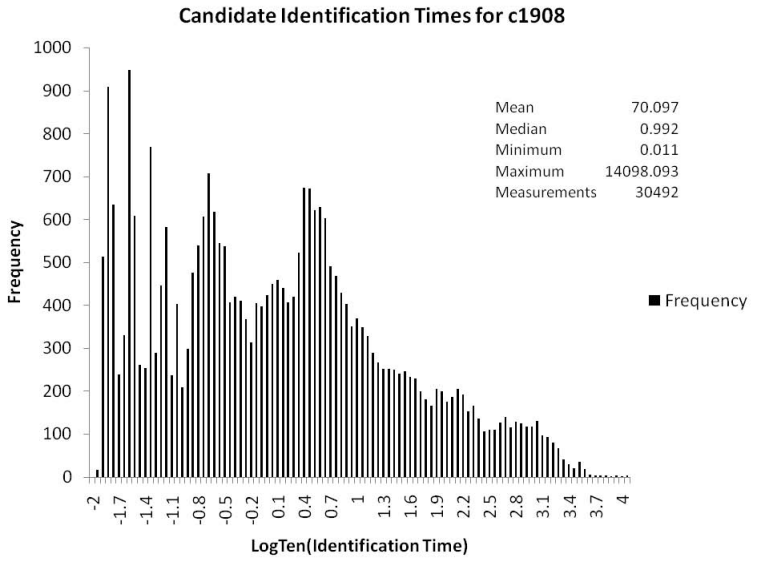


Figure C.2: Candidate identification times for circuit c1908.

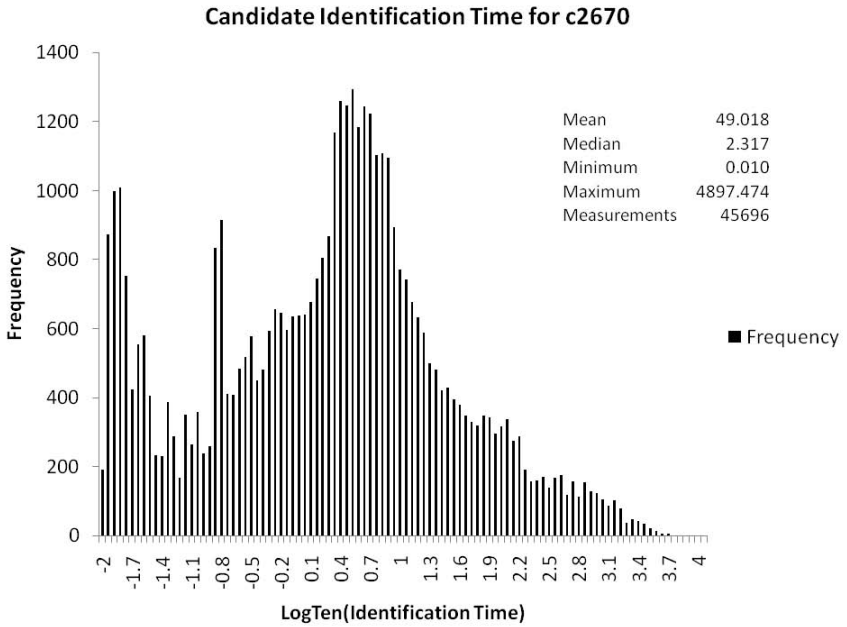


Figure C.3: Candidate identification times for circuit c2670.

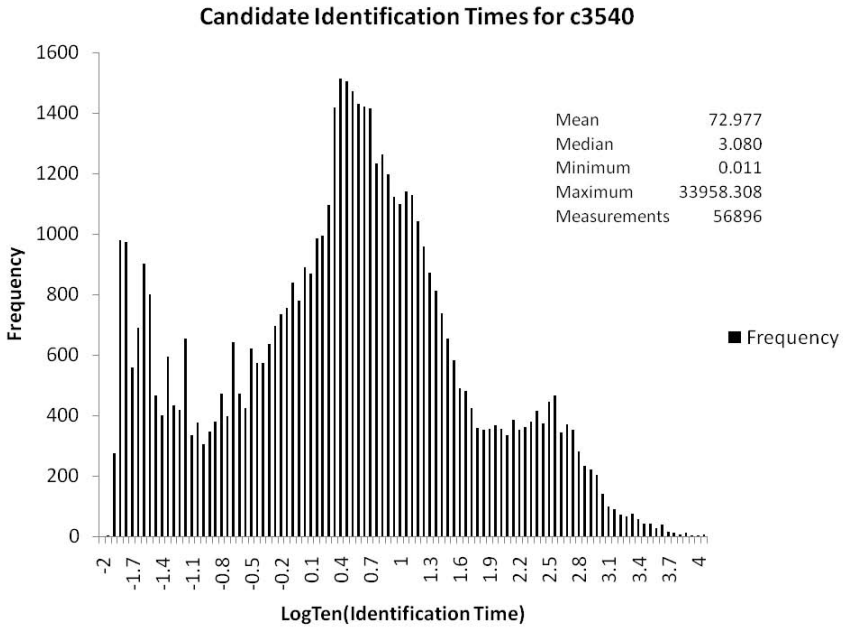


Figure C.4: Candidate identification times for circuit c3540.

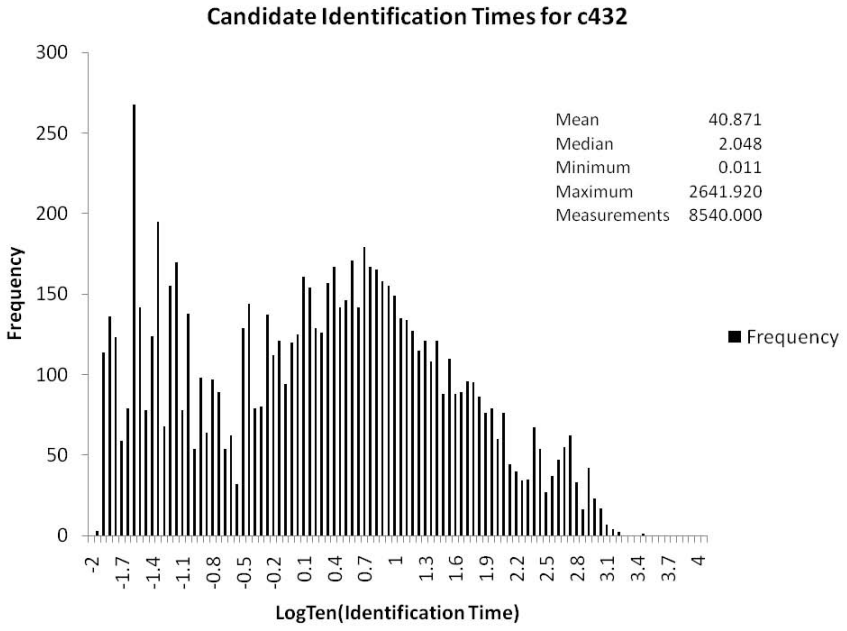


Figure C.5: Candidate identification times for circuit c432.

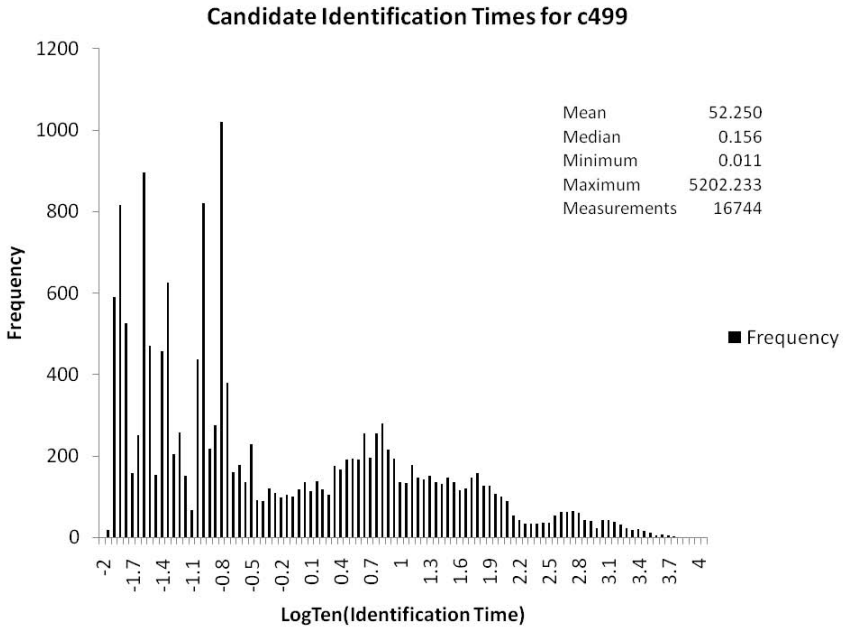


Figure C.6: Candidate identification times for circuit c499.

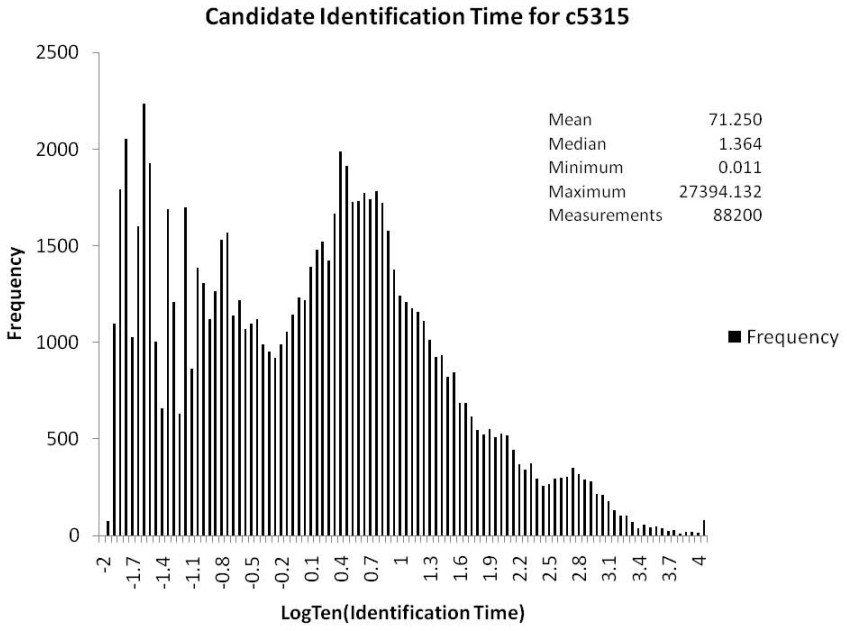


Figure C.7: Candidate identification times for circuit c5315.

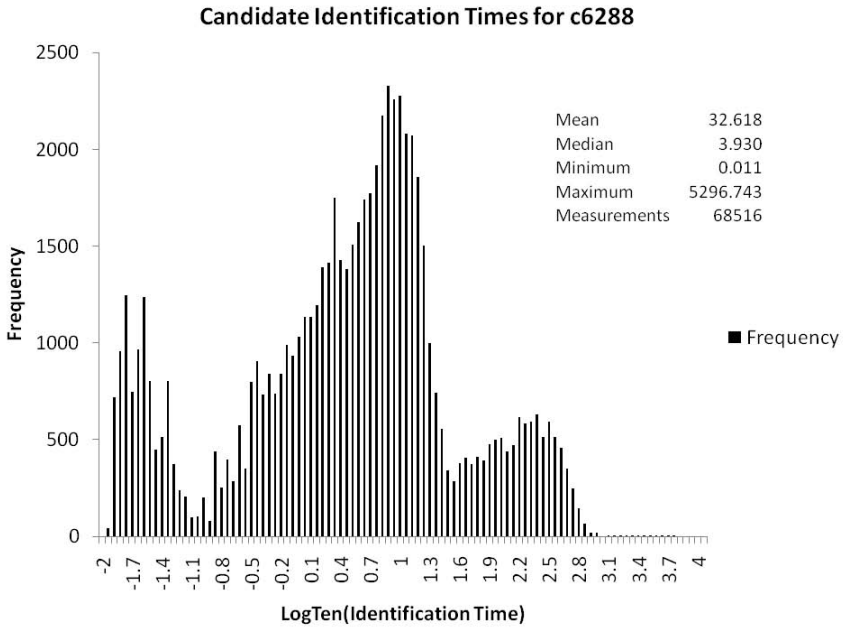


Figure C.8: Candidate identification times for circuit c6288.

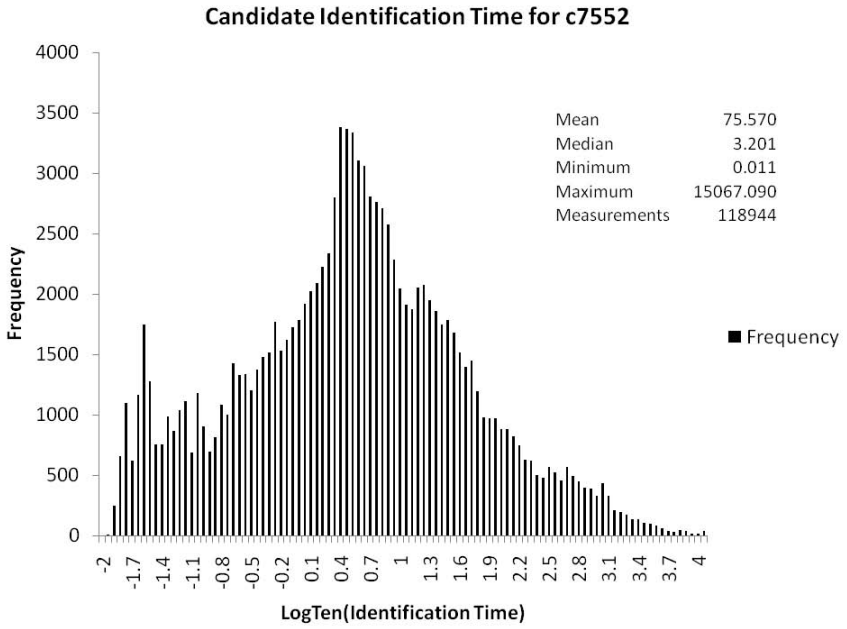


Figure C.9: Candidate identification times for circuit c7552.

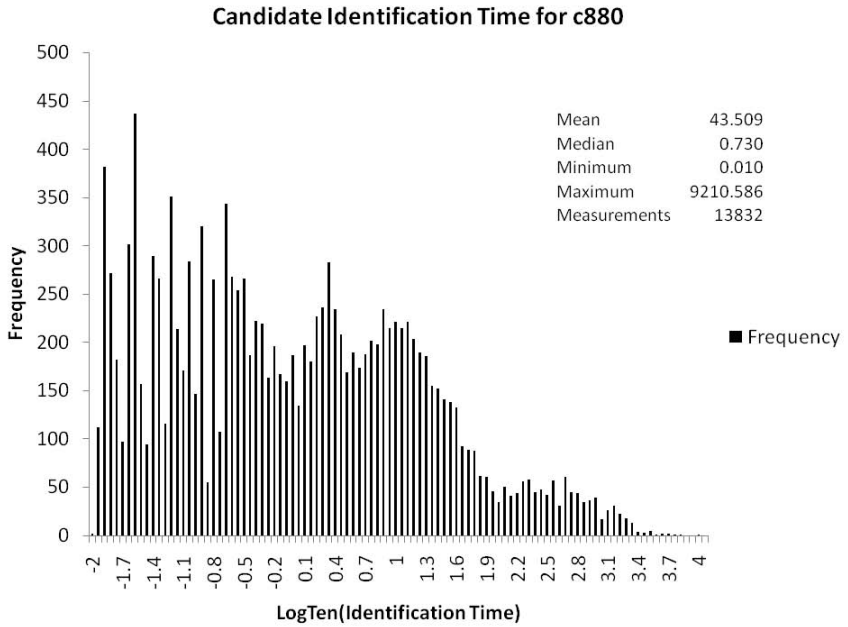


Figure C.10: Candidate identification times for circuit c880.

This appendix shows results of the generation process for a three input four output custom bench file. We outline the process in Section 4.3.

```
0 -- 11
1 -- 11
2 -- 00
3 -- 11
4 -- 10
5 -- 00
6 -- 11
7 -- 10
```

Figure D.1: Output of RandomCircuitOutput.java with three inputs and four outputs specified.

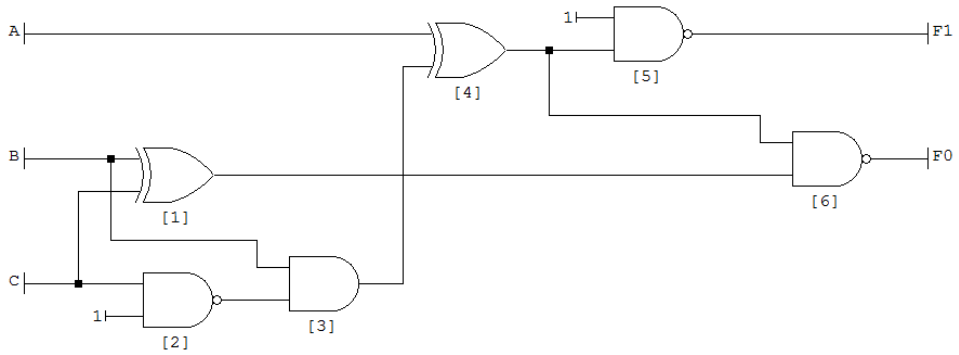


Figure D.2: Synthesized circuit after entering the results of the random circuit output into Logic Friday.

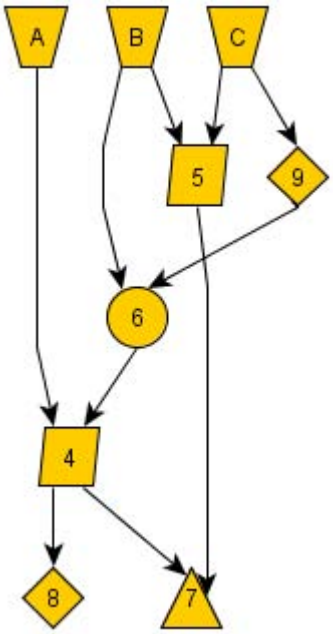


Figure D.3: Results of Logic Friday transcribed into yEd Graph Editor.

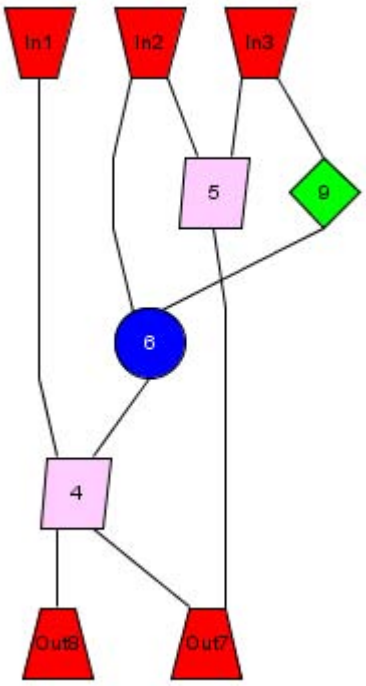


Figure D.4: Updated graph after execution of ConvertGraphMLToBenchAndShapedGraphML.java.

```
# C:\Users\Parham\Desktop\A3-2Example\Bench\3-2transcribed.bench.txt
#
# 3 Inputs
# 2 Outputs
# 2 Inverters
# 4 Intermediate gates( 2 ANDs + 2 ORs  )

INPUT(1)
INPUT(2)
INPUT(3)

OUTPUT(7)
OUTPUT(8)

4 = XOR(1,6)
5 = XOR(2,3)
6 = AND(2,9)
7 = NAND(4,5)
8 = NOT(4)
9 = NOT(3)

# Bench file generated from yEd .graphml file.
```

Figure D.5: Bench file produced from ConvertGraphMLToBenchAndShapedGraphML.java execution.

Appendix E. UML Diagrams

This appendix contains UML diagrams for software we developed during our research. Class files were added to the Program Encryption Toolkit (PET) for purposes of exploring component identification and hiding. Empty classes in larger diagrams are detailed individually later in this appendix. Complete PET dependencies are not shown.

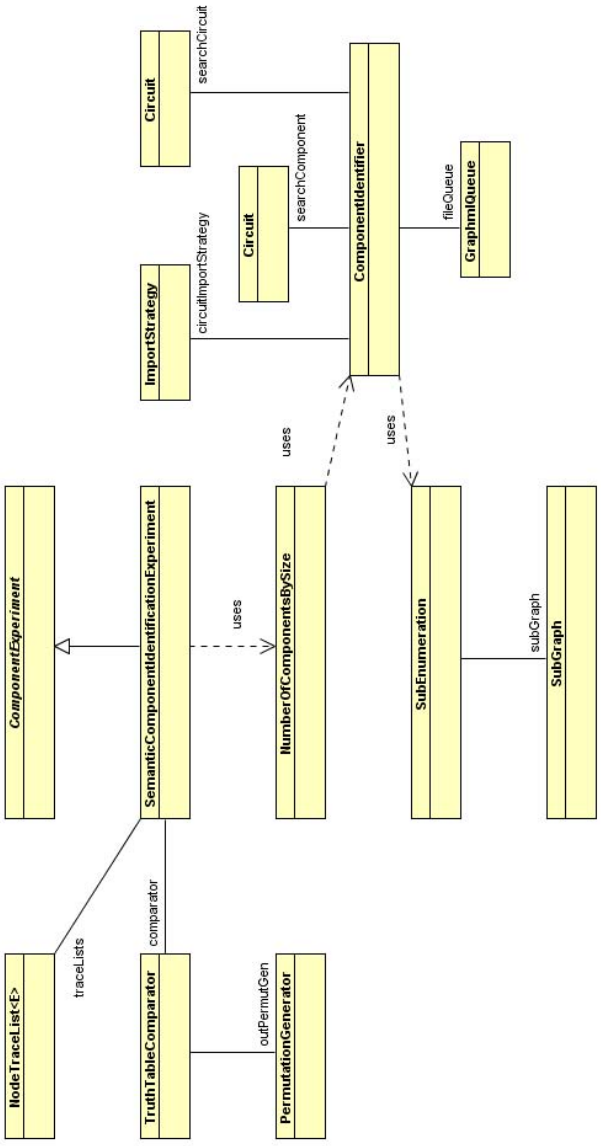


Figure E.1: Component Identification Tool class diagram.

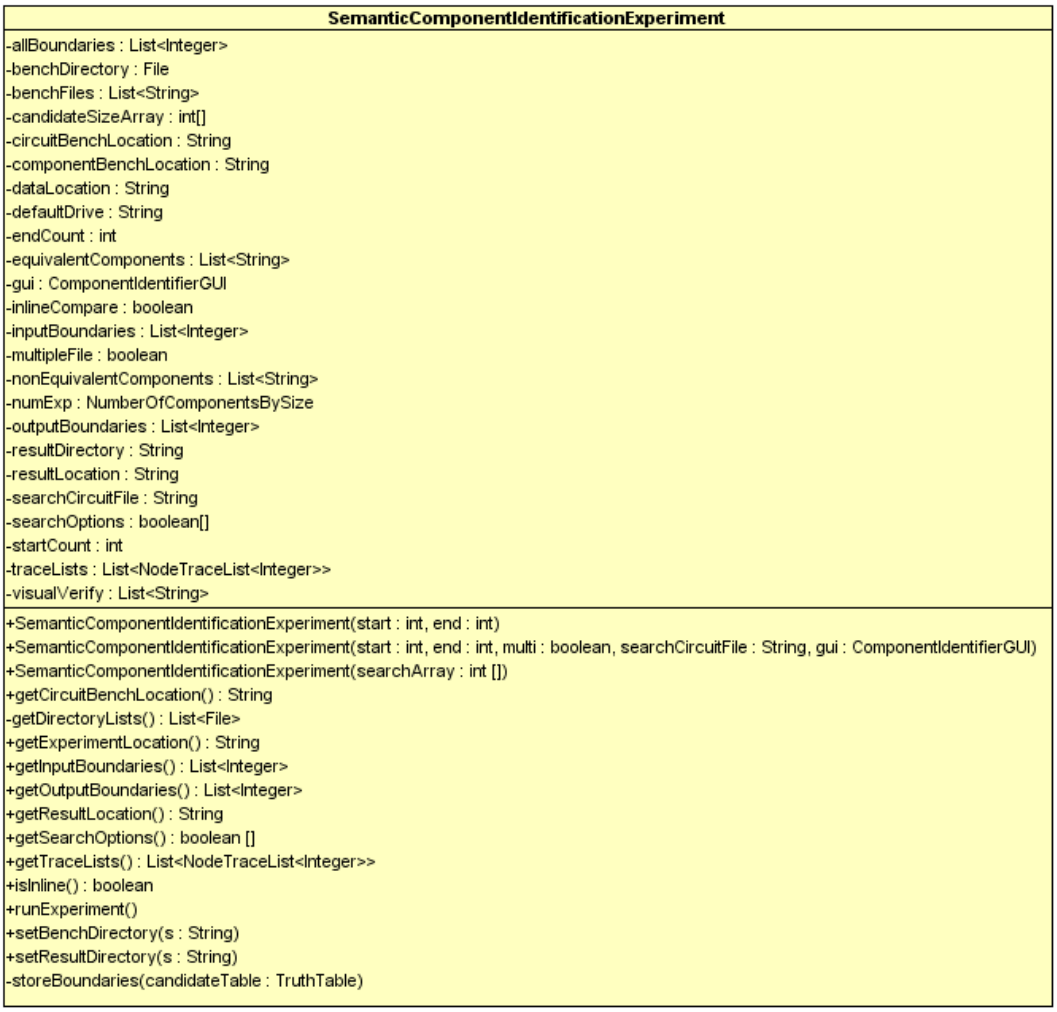


Figure E.2: SemanticComponentIdentifierExperiment class diagram.

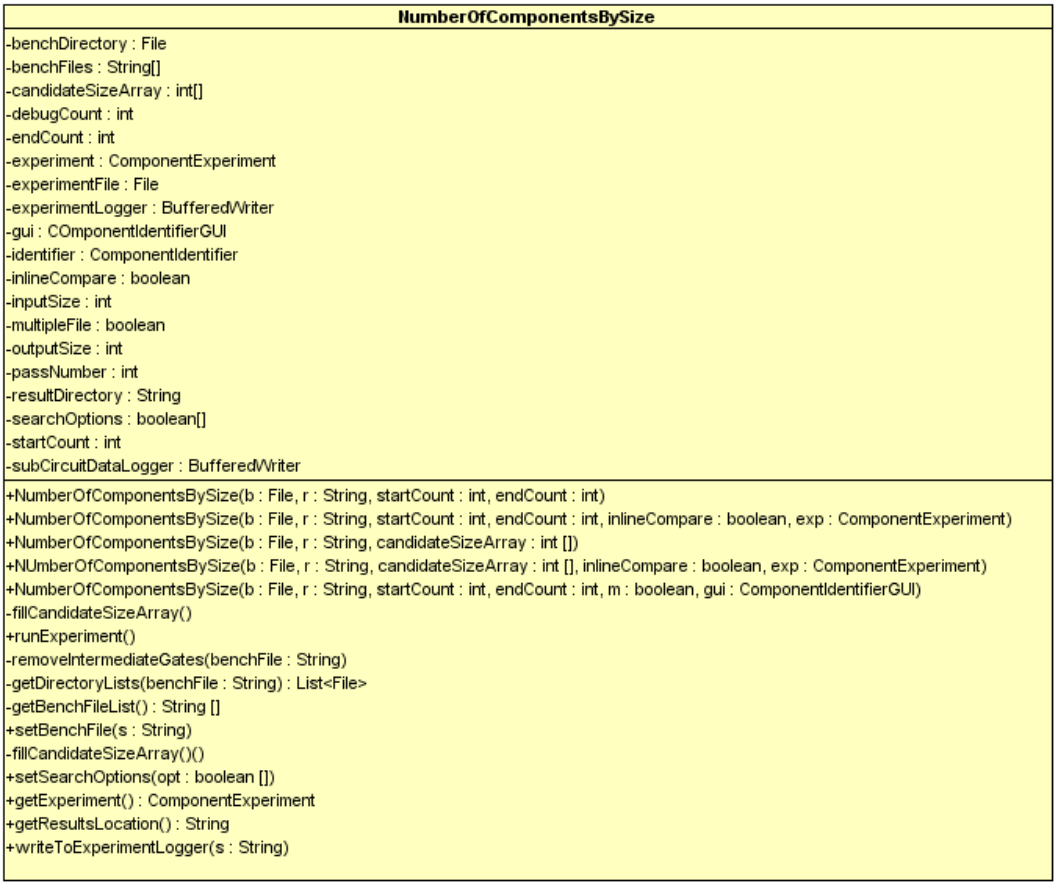


Figure E.3: NumberOfComponentsBySize class diagram.

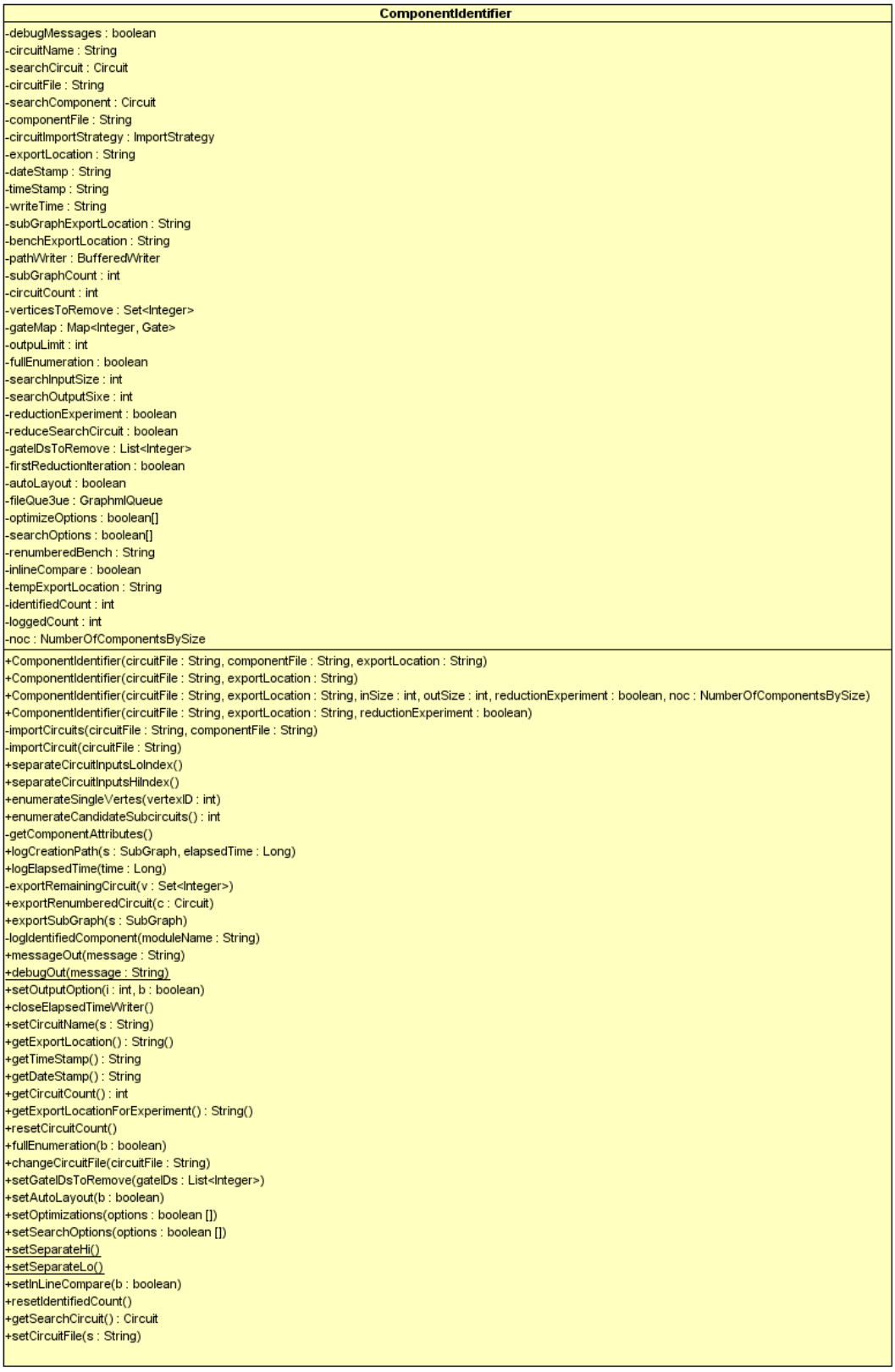


Figure E.4: ComponentIdentifier class diagram.

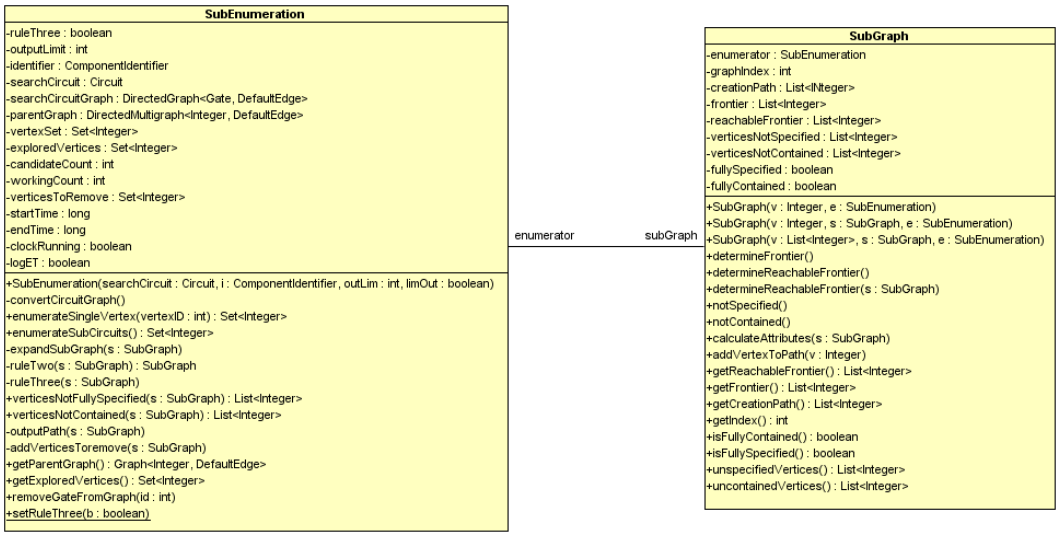


Figure E.5: SubEnumeration class diagram.

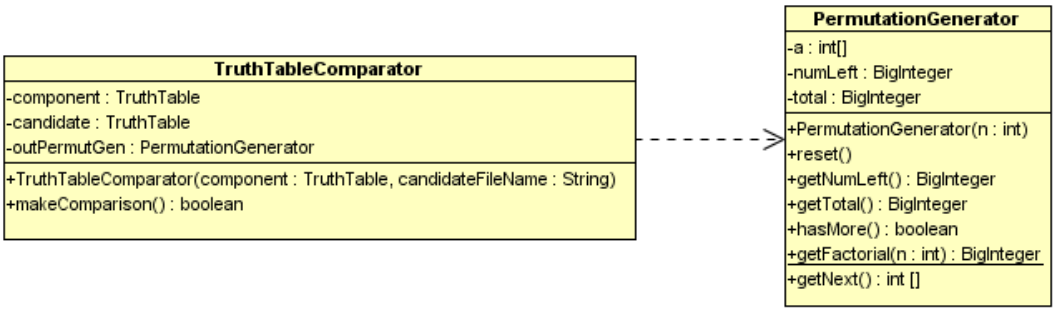


Figure E.6: TruthTableComparator class diagram.

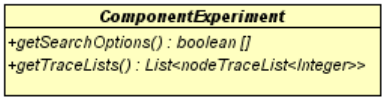


Figure E.7: ComponentExperiment class diagram.

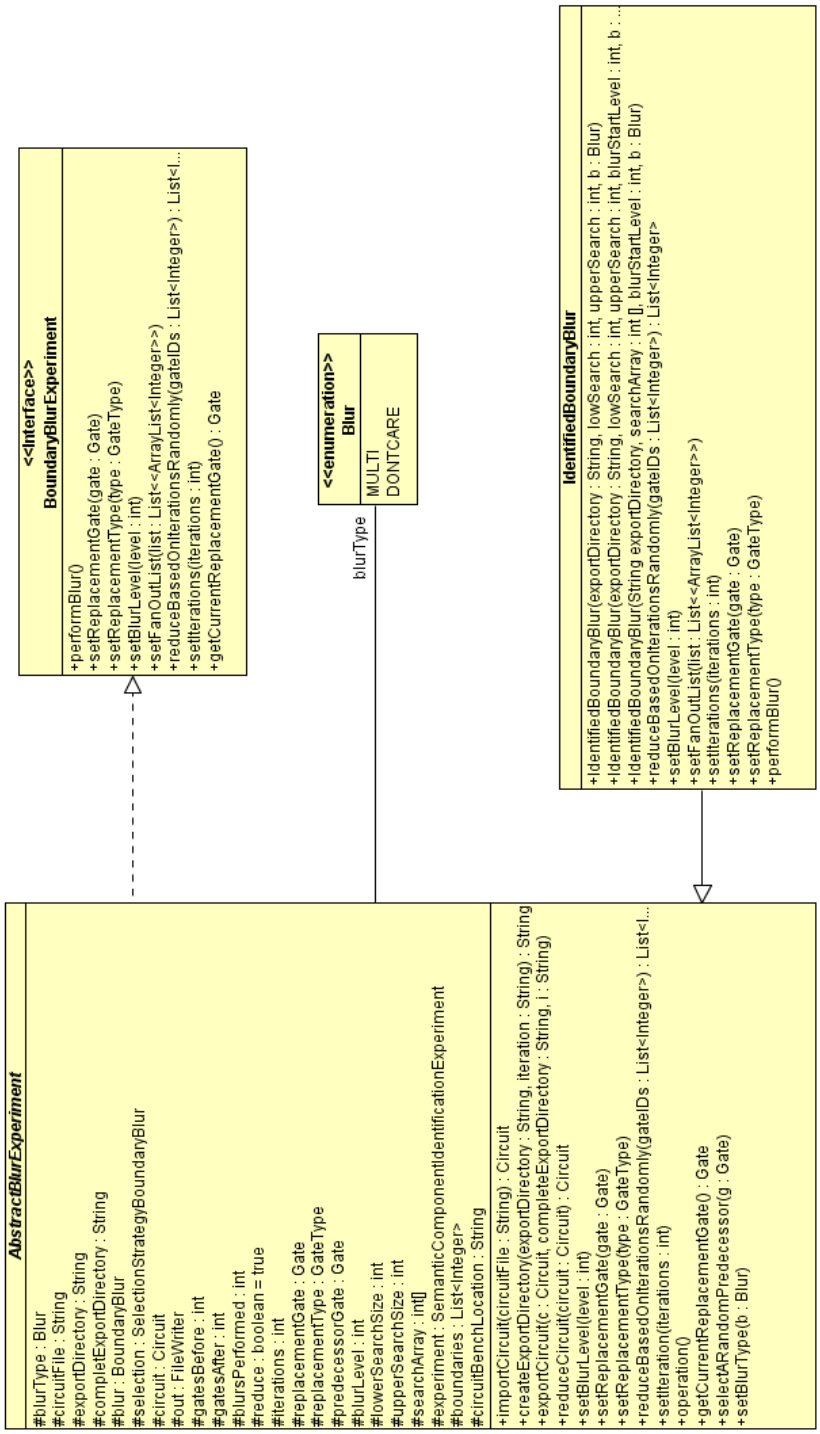


Figure E.8: IdentifiedBoundaryBlur class diagram.

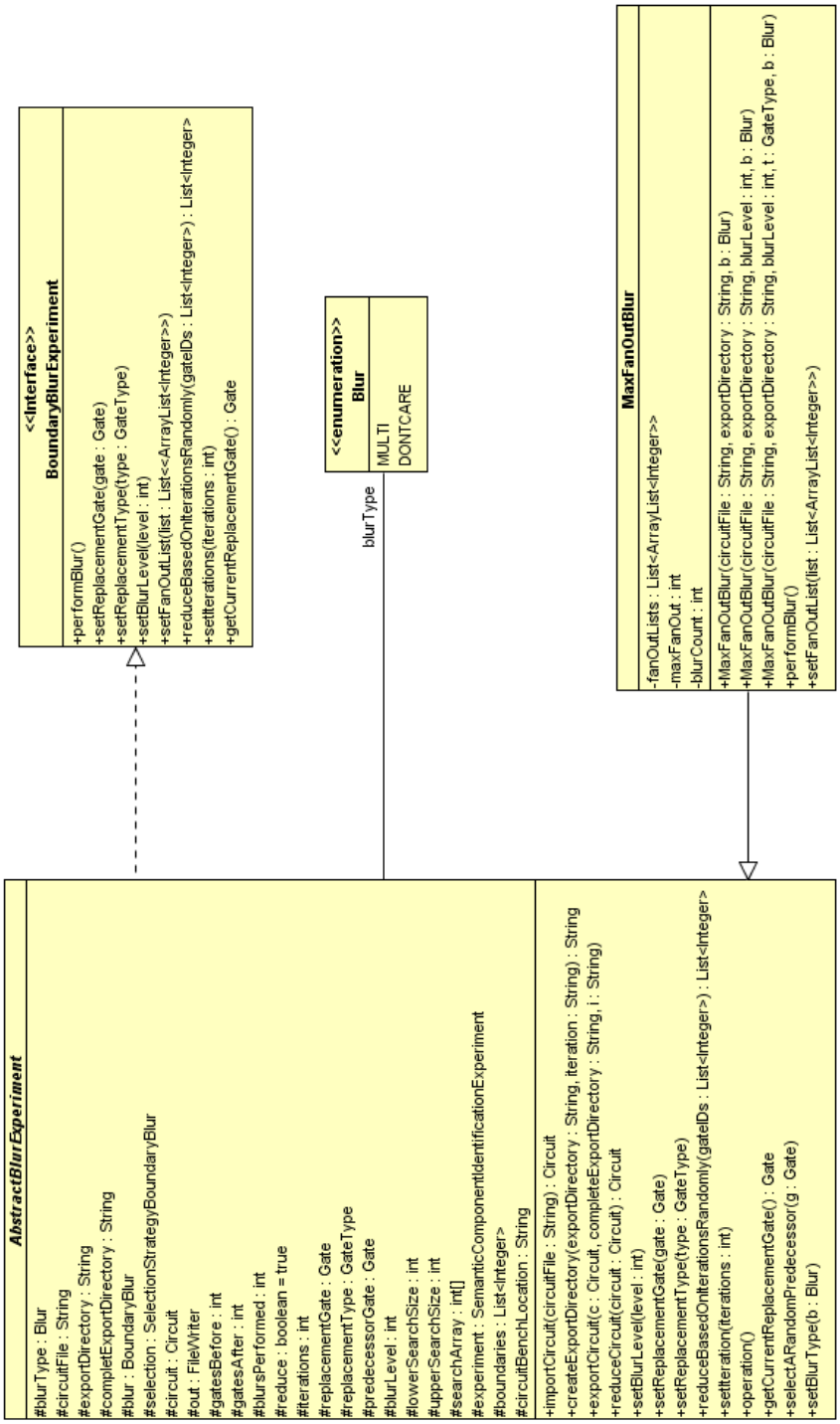


Figure E.9: MaxFanOutBlur class diagram.

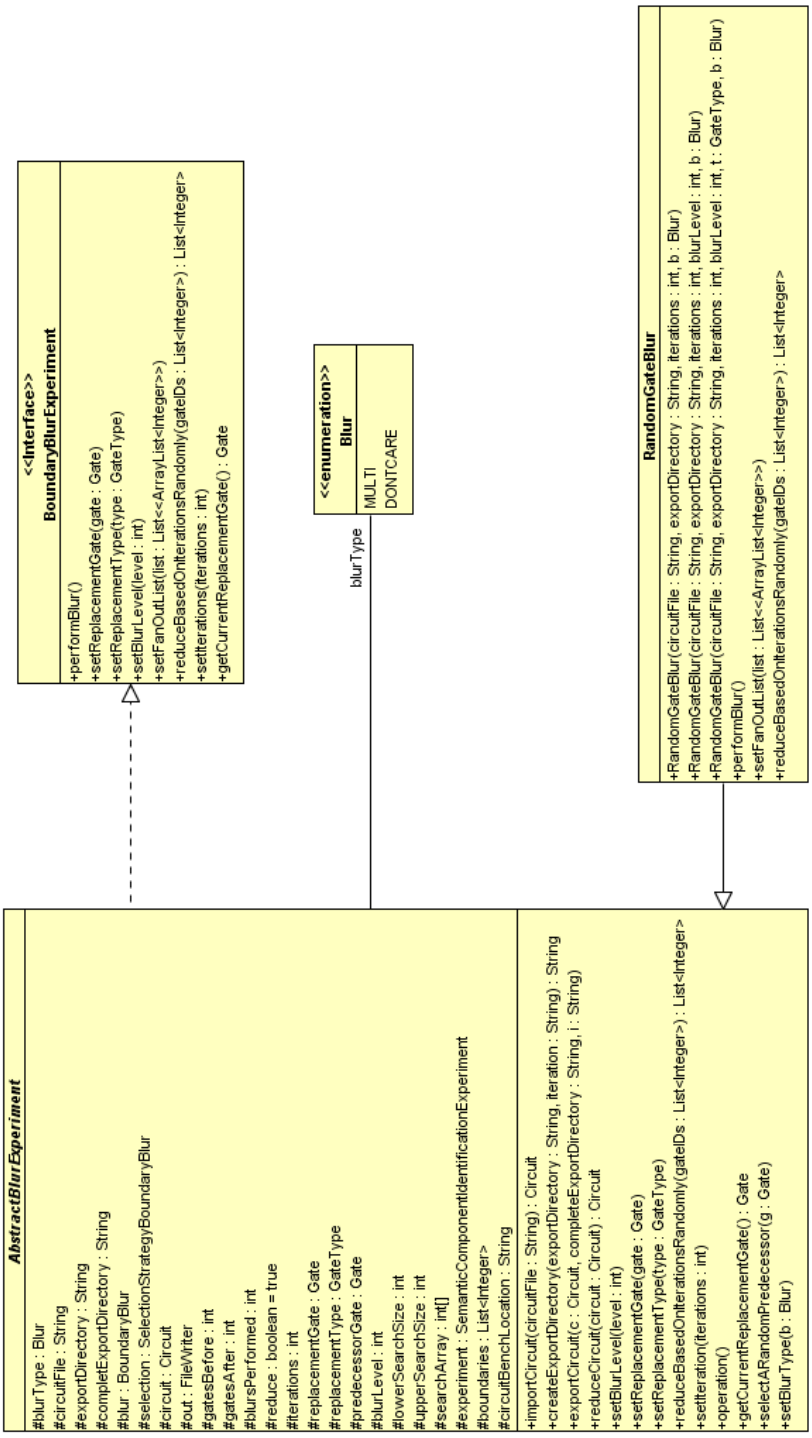


Figure E.10: RandomGateBlur class diagram.

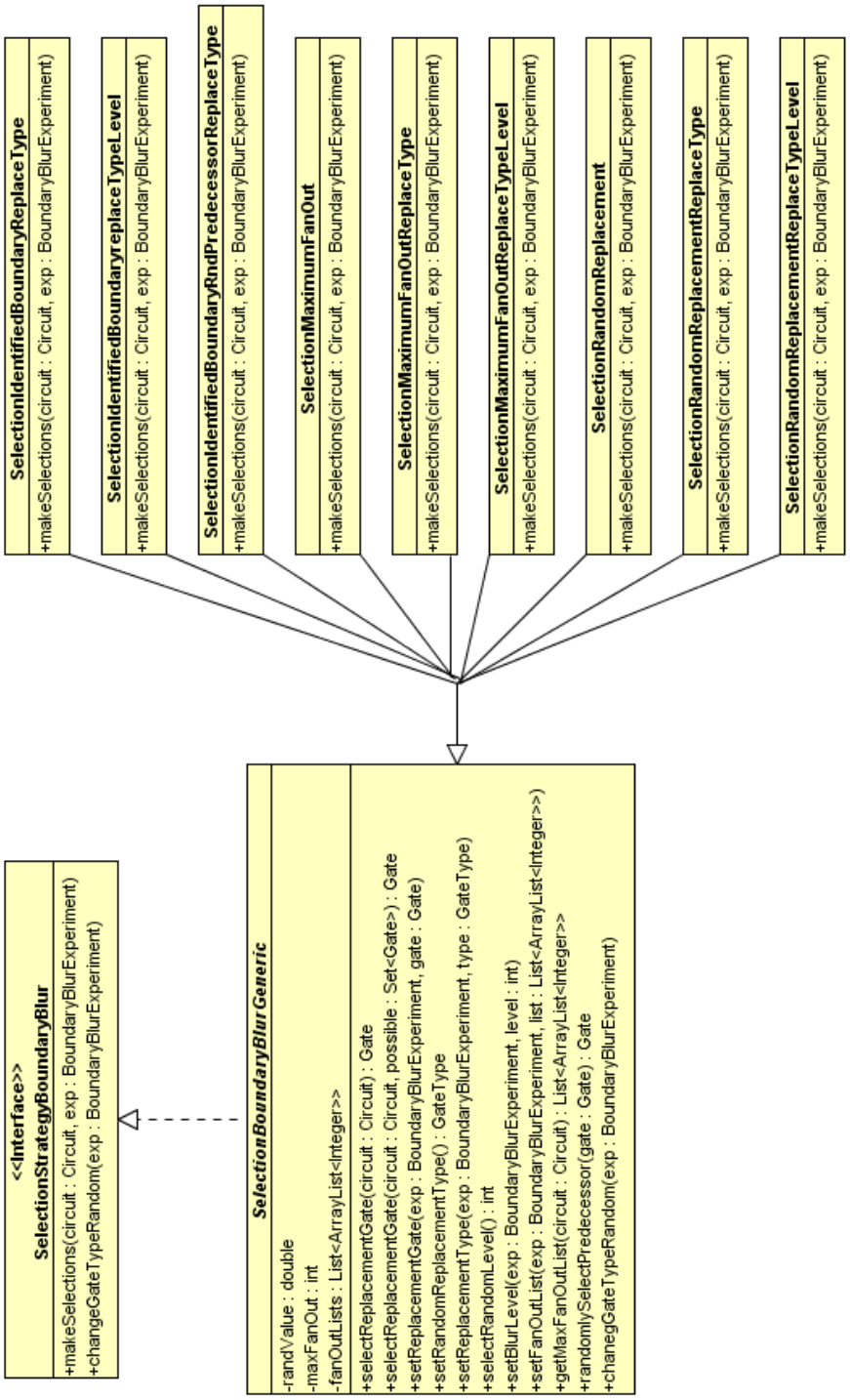


Figure E.11: Boundary selection strategies class diagram.

Appendix F. Identification Tool Module Library Circuits

This appendix lists BENCH files included in the known component library which also includes the custom bench files listed in Appendix B

Table F.1: Circuits contained in the module library and their I/O space.

Module File	Inputs	Outputs
BUFFER	1	1
NOT	1	1
AND	2	1
NAND	2	1
NOR	2	1
OR	2	1
XNOR	2	1
XOR	2	1
c2215	2	2
halfAdder	2	2
c237	2	3
c249	2	4
c3211	3	2
fullAdder	3	2
c3418	3	4
c3516	3	5
c3622	3	6
vPattern	4	1
c4211	4	2
c4327	4	3
c4440/4BitPermA	4	4
c17	5	2
c5241	5	2
c5355	5	3
c5479	5	4
c6276	6	2
c63103	6	3
c64145	6	4

Bibliography

1. Chikofsky, Elliot J. and James H. Cross II. "Reverse Engineering and Design Recovery: A Taxonomy." *IEEE Software*, 7(1):13 –, 1990. ISSN 07407459.

2. Collberg, C.S. and C. Thomborson. "Watermarking, tamper-proofing, and obfuscation - tools for software protection". *Software Engineering, IEEE Transactions on*, 28(8):735–746, Aug 2002. ISSN 0098-5589.

3. Hansen, Mark C., Hakan Yalcin, and John P. Hayes. "ISCAS-85 C6288 16x16 Multiplier". World Wide Web. Available at http://www.eecs.umich.edu/ jhayes/iscas/c6288.html.

4. Hansen, Mark C., Hakan Yalcin, and John P. Hayes. "Unveiling the ISCAS-85 benchmarks: a case study in reverse engineering". *IEEE Design and Test of Computers*, 16(3):72 – 80, 1999. ISSN 07407475. URL `http://dx.doi.org/10.1109/54.785838`. Carry look ahead;Error correcting circuits;Register transfer;.

5. J. Todd McDonald, Yong C. Kim Michael R. Grimaila, Eric D. Trias. "Using Logic-Based Reduction For Adversarial Component Recovery". 2009.

6. Jeffrey T. McDonald, Kenneth Norman, Yong C. Kim. "Introducing CORGI: A Framework for Whitebox Circuit Obfuscation". 2008. Unpublished, provided as reference material for CSCE693 Spring 2009.

7. Kim, Hanseok. *Removing Redundant Logic Pathways in Polymorphic Circuits*. Master's thesis, Graduate School of Engineering, Air Force Institute of Technology (AETC), Wright-Patterson AFB OH, March 2009. AFIT/GCS/ENG/09-03.

8. Kim, Yong C. and Lt. Col. J. Todd McDonald. "Considering Software Protection for Embedded Systems". *Crosstalk The Journal of Defense Software Engineering*, 22(6):4–8, 2009.

9. Nohl, Karsten, David Evans, Starbug Starbug, and Henryk Plötz. "Reverse-engineering a cryptographic RFID tag". *SS'08: Proceedings of the 17th conference on Security symposium*, 185–193. USENIX Association, Berkeley, CA, USA, 2008.

10. Norman, Kenneth E. *Algorithms for White-box Obfuscation Using Randomized Subcircuit Selection and Replacement*. Master's thesis, Graduate School of Engineering, Air Force Institute of Technology (AETC), Wright-Patterson AFB OH, March 2008. AFIT/GCS/ENG/08-17.

11. Office of the Under Secretary of Defense (Comptroller). "FY 2009 Budget Request". World Wide Web, 2008. URL `http://comptroller.defense.gov/defbudget/fy2009/2009_Budget_Rollout_Release.pdf`.

12. Rugaber, Spencer. “The use of domain knowledge in program understanding”. *Ann. Softw. Eng.*, 9(1-4):143–192, 2000. ISSN 1022-7091.

13. Tanenbaum, Andrew. “News Summary of Broken Dutch Transit Card”. World Wide Web, 2008. URL `http://www.cs.vu.nl/ ast/ov-chip-card/`.

14. West, Douglas B. *Introduction to Graph Theory*. Prentice-Hall, Inc., Upper Saddle River, NJ, second edition, 2001.

15. White, Jennifer L. *Candidate Subcircuit Enumeration for Module Identification In Digital Circuits*. Ph.D. dissertation, Department of Computer Science and Engineering, Michigan State University, 2000.

16. White, Jennifer L., Anthony S. Wojcik, Moon-Jung Chung, and Travis E. Doom. “Candidate subcircuits for functional module identification in logic circuits”. 34 – 38. Chicago, IL, USA, 2000. ISSN 10661395. Functional module identification;.

Vita

Lieutenant James D. Parham Jr. graduated from Port St. Joe High School in Port St. Joe, Florida. He entered undergraduate studies at the University of West Florida in Pensacola, Florida and obtained his Bachelor of Science degree in Electrical Engineering in 2006. He was commissioned through Reserve Officer Training Corps in 2006.

Lieutenant Parham entered the US Air Force in 1993. He was first assigned to the 3450th Technical Training Squadron Lowry AFB, Colorado where he received apprentice training as an F-111 avionics technician. He was then assigned to 27th Component Repair Squadron, Cannon AFB, New Mexico where he performed maintenance on the ALQ-99 Tactical Jamming Subsystem of the EF-111A aircraft. He earned the Cannon AFB Maintenance Professional of the Year award in 1995. In 1998, as an avionics journeyman, he was transferred to 16th Electronic Warfare Squadron, Eglin AFB, Florida where he began support of mission data development for the F-15 Tactical Electronic Warfare System (TEWS). By 2001, he obtained craftsman status. In 2001, he was assigned to 18th Maintenance Squadron, Kadena AB, Japan. There he was TEWS Intermediate Support Station team leader until 2002 at which time, he became an 18th Maintenance Group F-15 Avionics Quality Assurance Professional. In 2003 he was accepted into the Airman Education and Commissioning Program and was assigned to University of West Florida. In 2006 he earned his Bachelor of Science in Electrical Engineering from University of Florida. After commissioning he was assigned to 752d Combat Sustainment Group as a Global Positioning System Engineer in the Joint Service System Management Office. In 2008 he was selected to attend the Air Force Institute of Technology, Wright-Patterson AFB, Ohio. Upon graduation he will be assigned to the Directed Energy Directorate, Kirtland AFB, New Mexico.

Permanent address: 2950 Hobson Way
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
25–03–2010	Master's Thesis	Sept 2008 — Mar 2010

4. TITLE AND SUBTITLE

Component Hiding Using
Identification and
Boundary Blurring Techniques

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

James D. Parham, 1st Lt, USAF

5d. PROJECT NUMBER

10-299

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GE/ENG/10-22

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Dr. Robert L. Herklotz
Air Force Office of Scientific Research, AFMC
801 North Randolph Street, Rm 732
Arlington VA 22203-1977
703-696-9544 (DSN: 426) robert.herklotz@afosr.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

AFOSR/NL

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approval for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT Protecting software from adversarial attacks is extremely important for DoD technologies. When systems are compromised, the possibility exists for recovery costing millions of dollars and countless labor hours. Circuits implemented on embedded systems utilizing FPGA technology are the result of downloading software for instantiating circuits with specific functions or components. We consider the problem of component hiding a form of software protection. Component identification is a well studied problem. However, we use component identification as a metric for driving the cost of reverse engineering to an unreasonable level. We contribute to protection of software and circuitry by implementing a Java based component identification tool. With this tool, we can characterize time required for carrying out adversarial attacks on unaltered boolean circuitry. To counter component identification methods we utilize boundary blurring techniques which are either semantic preserving or semantic changing in order to prevent component identification methods. Furthermore, we will show these techniques can drive adversarial cost to unreasonable levels preventing compromise of critical systems.

15. SUBJECT TERMS

software protection, component hiding, reverse engineering, component identification

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			LtCol Jeffrey T. McDonald
U	U	U	UU	142	19b. TELEPHONE NUMBER *(include area code)* (937) 255–3636, ext 4639; jmcdonal afit.edu