Reverse Engineering Integrated Circuits Using Finite State Machine Analysis

Jessica Smith[†], Kiri Oler[*], Carl Miller[*], David Manz[*]
[*]*Pacific Northwest National Laboratory*
Email: first.last@pnnl.gov
[†]*Washington State University*
Email: jess.smith@wsu.edu

Abstract

Due to the lack of a secure supply chain, it is not possible to fully trust the integrity of electronic devices. Current methods of verifying integrated circuits are either destructive or non-specific. Here we expand upon prior work, in which we proposed a novel method of reverse engineering the finite state machines that integrated circuits are built upon in a non-destructive and highly specific manner. In this paper, we present a methodology for reverse engineering integrated circuits, including a mathematical verification of a scalable algorithm used to generate minimal finite state machine representations of integrated circuits.

1. Introduction

The integrity of our computing hardware is of critical concern in industries such as energy generation and distribution, aviation, and health care. Many of the ICs that control our desktop computers, servers, SCADA systems, and a range of other devices are designed in the U.S. but put into silicon overseas. [1] This creates a large gap in our control of the supply chain which puts all systems that use this hardware at risk for modification or injection attacks. Today, many organizations spend enormous amounts of money verifying the integrity of a given piece of hardware; they are then locked in to that hardware for decades afterwards, resulting in obsolete hardware and software running critical systems.

Currently, there is no way of verifying the integrity of the entire supply chain, from design to use, to ensure the level of integrity needed. In dividing this work into smaller, more feasible pieces, we have chosen to focus on examining the end product–the integrated circuit (IC). Many modern ICs are built upon finite state machines (FSMs). In this research, we have developed a method for rediscovering the FSM that an IC is built upon using a nondestructive and intelligent brute force method. Prior work has focused on destructive reverse engineering methods that use images of the transistor levels to determine function [2], and nondestructive characterization techniques like power usage, timing delays, current leakage, and electromagnetic imaging which can be used to certify an IC against a known-good IC. [3]–[5].

There are a few destructive existing methods for determining if an IC deviates from the original design; these are expensive and time consumptive but extremely accurate. Alternatively, there are a variety of non-destructive imaging methods for determining if an unknown IC is different from an 'assumed-good' benchmark IC. However, these methods often only work for large or active differences, and are based on the assumption that the benchmark chip has not been corrupted. What is needed, and what we will detail in the following sections, is an algorithm to enable a fast, non-destructive method of reverse engineering ICs to ensure their veracity. We must assume the worst case scenario in which we have no prior knowledge, no design documents, no labeling, or an out-of-production IC.

The mathematical theory behind our approach is presented in two parts. First, we construct a tree representing the IC, and then we determine the underlying state machine based on said tree. The evaluation tree is constructed by evaluating every possible input stream on the IC. Each evaluation tree is unique for each FSM, and any two FSMs that share the same evaluation tree are equivalent. It should be noted that evaluation trees are normal, as defined in [6], meaning the ordering on the nodes is preserved, allowing for the concept of descendant nodes and subtrees, which will be necessary as we proceed. Through basic pattern matching we can reduce the nodes and subtrees to work backwards towards the original state machine. This work will verify that both operations yield a state machine equivalent to the implemented machine. In addition, to mathematical verification, we tested our approach using a combined hardware/software implementation (subject to a future publication).

We begin by discussing the foundations of this approach, previous work, and related material. We then present our contribution to the problem by providing the mathematical foundation and the specific solution based upon finite state machines and tree exploration.

2. Prior Work

In theory, a black box is a device which takes inputs (signals applied to pins or ports) and then responds (via pins, ports, or other methods). The actual process which transforms the input into output is unknown. A black box test or analysis attempts to discover this functionality without damaging the system under test. There are a variety of

URI: http://hdl.handle.net/10125/41507
ISBN: 978-0-9981331-0-2

analysis methods, both destructive and nondestructive, that explore either the physical or logical makeup of the IC.

Physical Reverse Engineering The basic destructive method for exploring an IC is relatively simple in concept, but difficult to execute with the required precision. This method physically removes micrometer thin layers of the IC, paring it down slowly, and taking pictures with an electron microscope at each layer. [7] It is then possible to, using those images, reconstruct the physical layout of the chip and work upwards through the layers to recreate the transistor mapping. With the right tools and techniques, this is a very accurate and exact method. However, it is also destructive to the extreme - after stripping the IC down, there is nothing left but the plastic base and I/O pins. With some knowledge, it is possible to perform invasive probing of the IC, rather than a full destructive reverse engineering, to perform a more limited quality control check on the IC. In invasive probing, a very fine probe is inserted into the IC, after removal of the outer plastic or ceramic casing. This is usually only performed on exposed faces (not moving into deeper layers of the IC), due to the extreme difficulty of drilling into the IC without damaging any of the surrounding components. [8]

IC Characterizations Invasive probing and destructive reverse engineering damage the IC under test, often beyond any further usage. Other methods, such as structural, optical, or electrical characterizations [9], can be used to gain a less destructive but also a less detailed understanding of the IC. While these analysis methods cannot provide in-depth description of functionality that probing and reverse engineering can, they do provide a method for quickly checking for deviations from a known-good IC. These methods are more often used by manufacturers on samples from bulk IC purchases. The primary method of nondestructive fault detection in an IC uses electromagnetic, heat, or other physical outputs from the IC to determine a known-good baseline and then compare to this baseline for detection of alterations. It has been shown that in older ICs, each state switch of a transistor outputs a small amount of photons that can be detected. For example, in an IC designed to perform an AES encryption, the initial AES key can be recovered through this type of characterization. [10] Most other imaging methods are not this precise; they can detect unusual variations in the IC structure to 1 or 2 millimeters, which is not enormously helpful in determining what internally, at the transistor level, has been altered from the original design.

Logical Exploration Physical reverse engineering methods can be destructive or non-conclusive, and are based solely on the physical. Logical methods seek to reverse engineer the logical underpinnings of the IC, the state machines. There are three primary papers which we will review here; the first is Edward E. Moore's foundational theoretical paper [11], the second is a doctoral dissertation (and corresponding, supporting conference papers) by Michael Brutscheck [12]–[16] and the third was the preliminary work for this paper.

In 1956, Edward F. Moore described a mathematical foundation for logically reverse engineering finite state machines. In his Gedanken-Experiments [11], Moore details a situation in which a researcher needs to discover the functionality of a device without physically dismantling it; his example was drawn from experiences in the world wars, and assumed that the device would have an anti-tamper seal that would, upon breaking, destroy the device. In Moore's paper, an experiment is performed on the state machine by giving it a specific input sequence and recording the output sequence. Moore posits that, given two fully reachable state machines, S and T, each with a table of input sequences and the corresponding outputs which are produced upon the entry of the input into the system, these two state machines are isomorphic if the input and output table obtained from querying state machine S can be obtained by substituting new names for the input/output table obtained from state machine T, and vice versa. Isomorphic machines will always have the same behavior; they are indistinguishable from one another by any experiment, sequential or branching. State q_j is indistinguishable from state q_i (both of state machine S) if and only if every experiment performed beginning with state q_j has the same output as the same experiment performed on q_i. Two states are distinguishable only if they are not indistinguishable. It is then also possible to state that state q_i of machine S is distinguishable from state q_j of machine T if there exists an experiment that, if the same input is placed on both machines starting with their respective initial states, the output is not identical.

Moore's method, while logically sound, applied only to abstract or conceptual state machines. Real FSMs embedded in ICs pose additional complications, such as non-fully reachable state machines and more complex state machines, as well as the physical limitations inherent in interacting with a real IC. A non-destructive method of characterization of the FSM an IC implements was proposed by Brutscheck, et al. [14], which shows potential in implementing Moore's algorithms. The Brustcheck method follows several stages. The first of these is the pin determination based on electrostatic discharge theory, as described in [9], [17], and applied in both [15], [18], followed by a determination of the IC type: combinatorial, sequential linear, or sequential nonlinear. The FSM is then divided into Mealy or Moore automata. These determinations are assumed in this work; the Brutscheck method fulfills the requirements to determine the state machine competently and does not require further advancement.

In 2013, this author published a method of reverse engineering a state machine, improving upon prior work by allowing for on-terminating state machines and creating an implementation which provided much quicker results. [19] This was further explored in Smith's PhD dissertation. [20] The algorithm was originally implemented on a single core, single thread environment as a proof of concept. Results to this work were promising, and this work seeks to further that research.

Note: for brevity reasons, the software and hardware design & implementation discussion is omitted from this paper. Preliminary information has been covered in previous work and improvement details will be the subject of future work.

3. Prerequisites

Below we define the structures and notation relevant to the proposed method, culminating with the concept of tree equality for the purpose of manipulating those trees representing the FSMs.

3.1. Assumptions

The work presented here relies upon a few critical assumptions:

- The IC state machine must be a Moore FSM (i.e. the output depends only on the machine's state), and more specifically, not a Mealy FSM (i.e. the output depends on both current state and the input). It is possible to translate a Moore to a Mealy, or vice versa but this algorithm was based on Mooore's work
- An isolated state machine.
  The FSM must be in isolation and separated from any outside source which may affect the states or state transitions. In practice, this means that the FSM cannot be connected to any sort of non-volatile memory. Also, it cannot be allowed to take any input outside of that which is provided via the algorithmic testing apparatus.
- Scalability is possible.
  Though we have a theoretic basis for trees with any number of children/input pins, the larger this number, the greater the impact to the efficiency of our algorithm, which is a topic to be explored more thoroughly in future work.
- The single origin point is always accessible.
  There must be a single point of origin that can be accessed through a reset-style input. Physically this may be embodied in the power-off/power-on reset or a designated reset pin. To explore the tree properly, it is necessary that the exploration always begin at the same point.

3.2. Tree Framework

In order to better understand the unknown functionality and processes of a given FSM, the behavior of that FSM will be modeled using a tree structure. More specifically, an FSM is represented as a tree T with a set of nodes N, where each node has, at most, $c = 2^x$ children, where $x \in \mathbb{N}^+$, implying the FSM being modeled has x input pins. A node without any children is referred to as a leaf. For algorithmic purposes, nodes are labeled numerically (top to bottom, left to right), beginning with 0 for the root node and reading left to right down each level of the tree. Likewise, edges originating from the same parent node are grouped together and labeled numerically from left to right. See Figure 1 for an illustration of the node and edge labeling conventions.

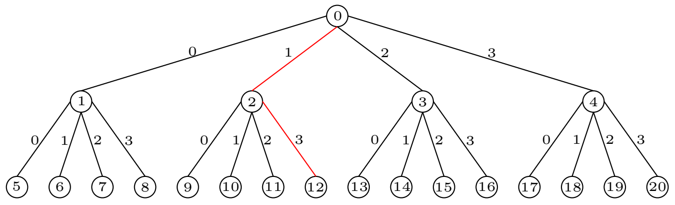

Figure 1. Example of a tree with nodes and edges labeled and an example path highlighted in red.

Each node in the tree represents a state in the FSM, with the child nodes representing the states that can be transitioned into from the given node or state. The root node of the tree is the FSM's initial state. A tree with no nodes or children is referred to as the empty tree and denoted **0**. As with the formal definition of a tree, each tree is a set N with c functions defined as follows:

$$\xrightarrow{S_i}: N \rightarrow N \cup \mathbf{0}$$

For example, if node n has children $s_1, s_2, \ldots, s_c$, then $n \xrightarrow{S_i} s_i$ for $1 \leq i \leq c$ or $S_i(t) = s_i$. Additionally, conventions dictate that a tree T is referred to by its root node, meaning notationally $S_1(T)$ refers to the leftmost child of node T, which is, in turn, the root node of the leftmost subtree of T. Likewise, the subtree labels proceed in ascending order from left to right. Therefore, S_cT refers to the rightmost child of T and rightmost subtree of T. We write $S_i(T) \leq T$ to denote that $S_i(T)$ is a subtree of T.

The behavior of the FSM is then mapped as paths through its representative tree. A path $P = p_0, p_1, ..., p_{d-1}$ where d is the length of the path and each p_j, where $0 \leq j \leq d-1$, indicates the label of the edge to select when moving from the current node in the sequence, e.g. a 0 indicates a move to the node's leftmost child, while a 1 indicates a move to the node's second leftmost child and so on. Paths pass through a sequence of nodes, the labels of which can be referenced as follows $labels = l_0, l_1, ..., l_d$, such that $l_k \xrightarrow{S_i} l_{k+1}$ for $1 \leq k \leq d-1$ and $1 \leq k \leq c$. Further, a node n is classified as a descendant of node l if and only if there exists a path from l to n. In the context of FSMs, the input to state l_1 led to a transition to state l_2 and so on and so forth through the state machine and on down the tree.

For a given path, p, or series of inputs to the FSM, the label l of the final node reached is given by the following:

$$l = \sum_{i=0}^{d}(2^{ix} + c(i)) \tag{3.1}$$

where

$$c(i) = (2x-1)\sum_{j=0}^{i-1}(c(j)) + p_i \tag{3.2}$$

Conversely, if we are given a label, l, and need to solve for the path, p of length d from the root to the node with the given label, i.e. the series of inputs to the FSM, is given by the following:

$$p_i = (l_{i+1} - 1) \text{ mod } 2x \tag{3.3}$$

Which means we start with the given label l_d and work our way backwards up the tree to the root node, by first solving for p_{d-1} as follows:

$$p_{d-1} = (l_d - 1) \text{ mod } 2x \tag{3.4}$$

where l_d is the given label. We can use the given label to solve for the label of its parent node using the following equation:

$$l_{i+1} = \frac{l_{i+1+1} - \frac{(2x)^{i+1+1}-1}{2x-1}}{2x} + \frac{(2x)^{i+1} - 1}{2x - 1} \tag{3.5}$$

which in turn, requires knowing the length of the path d, given by:

$$d = \left\lceil \frac{log(2lx - l + 1)}{log(2x)} - 1 \right\rceil \tag{3.6}$$

Equations 3.6 and 3.5 are derived from the fact that the number of nodes in a tree is $\frac{(2x)^{d+1}-1}{2x-1}$.

3.3. Equality

We now examine what it means for two trees to be equal. For Moore's FSMs [11], equality of nodes is established if the output is the same for both nodes. This is an example of an equivalence relation (denoted as $=_N$), illustrating that equivalence relations are valid on evaluation trees. Let $=_N$ be an equivalence relation on nodes of a tree. Two trees are equal $T_1 = T_2$ if and only if T_1 and T_2 are both **0** or all of the following are true: $T_1 =_N T_2$, and $S_i(T_1) = S_i(T_2)$ for all i such that $1 \leq i \leq c$.

Theorem 3.1. *$=$ is an equivalence relation.*

Proof. The proof is by structural induction.
Base Case: $\mathbf{0} = \mathbf{0}$. This is trivially true, since there is only one empty tree.
Let T_1, T_2, and T_3 be trees in which all of their subtrees are equal.

Reflexive
: We know $T_1 =_N T_1$ by reflexivity on $=_N$. Since all the subtrees of the given trees are equal, we also know $S_i(T_1) = S_i(T_1)$. Therefore, $T_1 = T_1$.

Symmetric
: Let $T_1 = T_2$. We know $T_2 =_N T_1$ by symmetry on $=_N$, since $=_N$ is an equivalence relation. Furthermore, $S_i(T_2) = S_i(T_1)$, since all subtrees of the given trees are equal. Thus $T_2 = T_1$.

Transitive
: Let $T_1 = T_2$ and $T_2 = T_3$. We know $T_1 =_N T_2$ and $T_2 =_N T_3$, thus $T_1 =_N T_3$ by transitivity on $=_N$, since $=_N$ is an equivalence relation. By definition $S_i(T_1) = S_i(T_2)$ and $S_i(T_2) = S_i(T_3)$. Finally, since all subtrees of the given trees are equal, we know $S_i(T_1) = S_i(T_3)$. Therefore, $T_1 = T_3$.

Thus by structural induction $=$ is an equivalence relation on trees.

□

Corollary 3.2. *If $A = B$ and if P_A is a path through A and P_B is the same path through B, then $P_A = P_B$.*

Proof. A path through a tree is also a tree. Consequently, the corollary follows directly. □

Thus far in this discussion of trees, the tree has not been limited to the finite or acyclic variations. Consequently, this definition of equality will not work for any practical computation due to real world limitations on time and space resources. Therefore, we will now present a limited version where equivalence between trees is determined out to a given depth, d. Let $=_d$ be a relation on trees, where $d \in \mathbb{N}$. Then $T_1 =_0 T_2$ if and only if T_1 and T_2 are both **0** or $T_1 =_N T_2$. Additionally, $T_1 =_d T_2$ if and only if T_1 and T_2 are both **0** or all of the following are true: $T_1 =_N T_2$, $S_i(T_1) =_{d-1} S_i(T_2)$ for all i such that $1 \leq i \leq c$.

Theorem 3.3. *$=_d$ is an equivalence relation on trees.*

Proof. Since $=_d$ is a limited version of $=$, the proof is almost identical to that of Theorem 3.1 and has thus been omitted. □

Corollary 3.4. *If P_A is a path through A and P_B is the same path through B, and $A =_d B$ and $|P_A| < d$ then $P_A = P_B$.*

By applying the above limitation, we now have an effective means of comparing trees.

4. Solution

Now that the necessary structures for representing a FSM as an evaluation tree and a means for comparing two FSMs based on their respective evaluation trees have been established, we are ready to present a procedure for streamlining the creation of evaluation trees to make them more practical.

4.1. Folding

It is simple to show that if a state machine has a loop, then the subtrees generated from those states are identical. The basic premise of the process presented here is to go in the other direction. That is, we find identical subtrees and replace the redundancies with a loop back to a single copy of the subtree, as shown in the example in Figure 2.

Theorem 4.1. *If two FSMs M and N generate the same evaluation tree T, then M is equivalent to N.*

Proof. Let M and N be two FSMs that both generate T. Two machines are equal if and only if for every input word

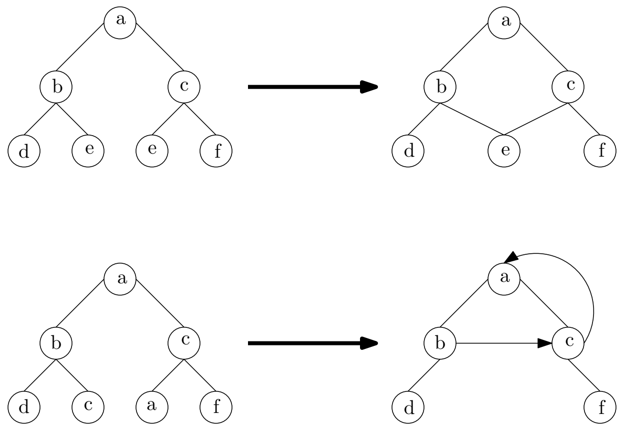

Figure 2. Two simple examples of the folding procedure.

w, $M(w) = N(w)$. Let w be an arbitrary word. $M(w)$ can be evaluated by following the transition function for M. Now $M(w)$ can be related to a path in T, where a transition from one state to another is modeled as the move from a parent node to one of its children. If $w_k = 0$, then continue with $S_1(T)$; if $w_k = 1$, then continue with $S_2(T)$ and so on. This path will produce $M(w)$. Since it is possible to do the same thing for N because N also generates T, $M(w) = N(w)$. Finally, since w was arbitrary $N = M$. $\square$

Theorem 4.2. *Given FSM M and its corresponding evaluation tree T, if $A \leq T$ and $B \leq T$ and $A = B$, then there is an equivalent machine M' where A and B represent the same state in M'.*

Proof. Assume that $A \leq T$ and $B \leq T$ and $A = B$. If A and B are the same state then we are done. Now construct a machine M' where every transition to state B is replaced by state A. Let T' be the evaluation tree for M'. Now either $B \leq S_i(T)$ for some i such that $1 \leq i \leq c$ or B is T, in which case $B = T$. If $B = T$ then $A = T'$, so $T' = A = B = T$ and thus $T' = T$. Otherwise, $B \leq S_i(T)$. Again there are two possibilities: $B \leq S_j(S_i(T))$ (where $1 \leq j \leq c$), or $B = S_i(T)$. Continue this process until we find a subtree D of T where $B = D$. Let D' be the same descendant in T', then $D = B = A = D'$. Now Ds parent is equal to D's parent. $S_i(p(D)) =_N S_i(p(D'))$, and $D = D'$. By induction, every ancestor of D is equal to every ancestor of D'. Since T is an ancestor of D and T' is the same ancestor of D', then $T = T'$. Therefore, since the two evaluation trees are equal $M = M'$. $\square$

4.2. Algorithm

Using Theorem 4.2 presented above, it is now possible to formulate an algorithm. The premise of the algorithm is as follows: a tree or subtree can be replaced with an equivalent tree while allowing the underlying FSM to remain unchanged. It is then possible to eliminate entire branches of the tree by looping back to a node that has previously been explored. The implementation shown in Algorithm 1 employs a basic breadth first search on a tree, meanwhile, Algorithm 2 replaces every occurrence of a in the tree with b.

Algorithm 1 Fold

```
procedure FOLD(Tree $T$)
    Q $q$
    $seen \leftarrow \emptyset$
    $q \leftarrow T$
    while $q \neq \emptyset$ do
        $t \leftarrow q$
        if $\exists x \in seen : t = x$ then
            switch$(t, x, seen)$
        else
            $seen \leftarrow t$
            $q \leftarrow l(t)$
            $q \leftarrow r(t)$
        end if
    end while
end procedure
```

Algorithm 2 Switch

```
procedure SWITCH(Tree $a$, Tree $b$, Set⟨Tree⟩ $seen$)
    for $x \in T$ do
        if $l(x)$ is $a$ then
            $l(x) \leftarrow b$
        end if
        if $r(x)$ is $a$ then
            $r(x) \leftarrow b$
        end if
    end for
end procedure
```

As a consequence of Theorem 4.2, we have the following corollary.

Corollary 4.3. *At each step of the algorithm, T is an equivalent state machine.*

Proof. The only modification made to T is the switch procedure, which only replaces one equivalent subtree with another, thus the corollary follows. $\square$

Theorem 4.4. *The Fold algorithm halts.*

Proof. T has been generated by a FSM M, so by definition T is finite. Therefore, the longest path through T without seeing an equivalent state is $|M|$. Algorithm 1 is then guaranteed to cut every branch after length $|M| + 1$. Thus the output tree is bounded by $|T| < c^{|M|+1}$. Since this implementation utilizes a breadth first search, it is impossible to travel down an infinitely long branch. Therefore, the algorithm must halt when the tree is fully folded, or after $c^{|M|+1}$ steps. $\square$

4.3. Finiteness

As already noted, we cannot use the normal definition of $=$ on trees because the algorithm would then be potentially infinite. Unfortunately, if we attempt to use $=_d$ rather than $=$, the theorems and resulting algorithm are no longer true. However, with a little care in selecting an appropriate value for d, we can show that the theorems are almost always true and still useful.

First, it should be noted that for trees T_1 and T_2, intuition dictates that the larger d is, the more accurate our results will be, since we will be considering a greater portion of the tree. That is:

$$\lim_{d\to\infty} T_1 =_d T_2 \equiv T_1 = T_2.$$

Next, given that the diameter of a graph is defined to be the length of the longest geodesic, or shortest path, among all node pairs [21], let $\mathrm{Diam}(M)$ denote the diameter of the evaluation tree corresponding to a given FSM, M.

Theorem 4.5. *Theorem 4.2 remains true if the* $\mathrm{Diam}(M) \leq d$.

Proof. Assume that $A =_d B$, but that A and B do not represent the same state in the machine. Then $\exists e > d : A \neq_e B$. Let e be the smallest such number where this is true. This implies that there are paths P_A and P_B in A and B respectively that are the same path, but are not equivalent and $|P_A| = |P_B| = e$. Since e is the shortest such path, no loops in the state machine have been made. Therefore, there are at least e states in M. Furthermore, there is a path P_A that is a minimum path between two states with at least e states. Thus, by definition, $\mathrm{Diam}(M) \geq e > d$. □

Theorem 4.5 is actually much stronger than one would expect. While there are example state machines of size $d+2$ where the folding technique will fail, these examples are a relatively small set of possible state machines. In general, using a depth of d for $=_d$ will distinguish between different state machines of size up to c^{d-1}.

4.4. Comparison Depth

An additional component of this methodology that must be addressed is at what point it is possible to assert that two states A and B are equivalent? How deep in the tree structure that one must compare? This point will hereafter be referred to as comparison depth. Within the context of evaluation trees, the comparison depth can be thought of as the depth to which A and B must be evaluated and found equivalent before being deemed equivalent states overall. The acceptable comparison depth for a given FSM is determined by multiple factors based on the system in question. For example, a more critical system will require a higher comparison depth. To formalize this concept, let the depth, $d = cl$, represent the necessary comparison depth. Then A and B are considered equivalent states if $A =_{cl} B$. As a consequence, the FSM's evaluation tree must be evaluated to a minimum of depth $cl+1$. Figure 3 shows a fold procedure performed on equivalent states D and E with a comparison depth of 2.

5. Optimization

Now that we have established a valid, finite algorithm, we turn our attention toward optimizing its speed. The primary inhibitor faced by the algorithm is the potential combinatorial explosion. Consider a FSM with 70 states. Constructing a binary evaluation tree that is 70 levels deep requires 2^{70} operations, or 1.18×10^{21} operations. Assuming one trillion operations per second, which is a high estimate, that many operations would require roughly 3500 years to calculate. Clearly, this is prohibitively expensive. Furthermore, the vast majority of the evaluation tree isn't needed, owing to the likelihood that many of the high level states, those in the first ten tiers or so of the tree, will be matching states. By iteratively exploring the tree to the comparison depth, comparing and reducing the tree, and then exploring only those un-reduced nodes, it is now possible to perform an intelligent brute force exploration. This method will reduce the amount of the tree that must be explored, drastically reducing the time required. For example, to re-evaluate the 70-state tree described above with a comparison depth of five, the algorithm can first explore seven levels deep in the tree. If the initial tree has one line of matching states, say the rightmost (a waiting state, where a given state will remain in that state for all inputs except one specific input), we can already eliminate approximately 1/8 to 1/4 of the tree.

Through this intelligent brute force method, we are able to iteratively explore and reduce the larger, possibly infinite tree. To explain this in greater detail, we present Figure 4 and an example using this graphical representation. Our initial exploration of the tree should be to a depth equal to the comparison depth plus at least one. A depth of the comparison depth plus one will allow us to detect if the origin node is identical to any of its children, or if those first level children are identical to each other.

After this initial exploration, step 1 in Figure 4, and initial reduction, step 2, of the primary tree, we then consider any of the leaf nodes (defined to be leaf nodes by the comparison depth in use) that have not been removed due to higher-level loops and reductions. Each of the unreduced leaf nodes become the origin of a secondary tree. All of the secondary trees are explored to the comparison depth plus one depth (step 3) and reduced individually (step 4). After each secondary level tree has been explored, we then group all the secondary level trees, or siblings, and combine them with their parent primary tree. This larger tree, consisting of the entire currently-explored node space, is then reduced as a whole (step 5).

We then consider any leaf nodes not yet removed or looped back into the tree. These unexplored leaves become the origin nodes of the tertiary trees and are explored to the comparison depth plus one depth again, as shown in step 6. We reduce each of the tertiary tree individually (step 7) and then join them with only their siblings and secondary-level

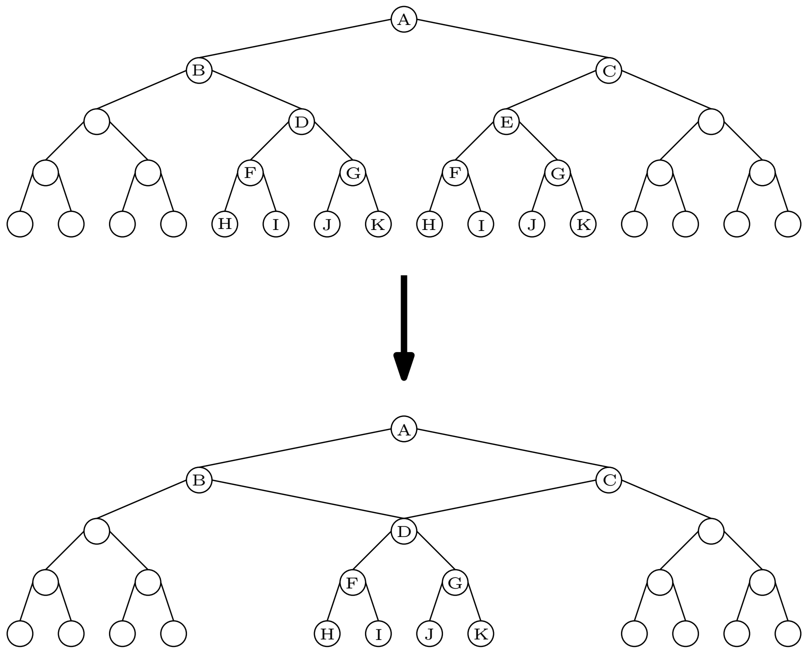

Figure 3. An example of a fold procedure performed with a comparison depth of 2.

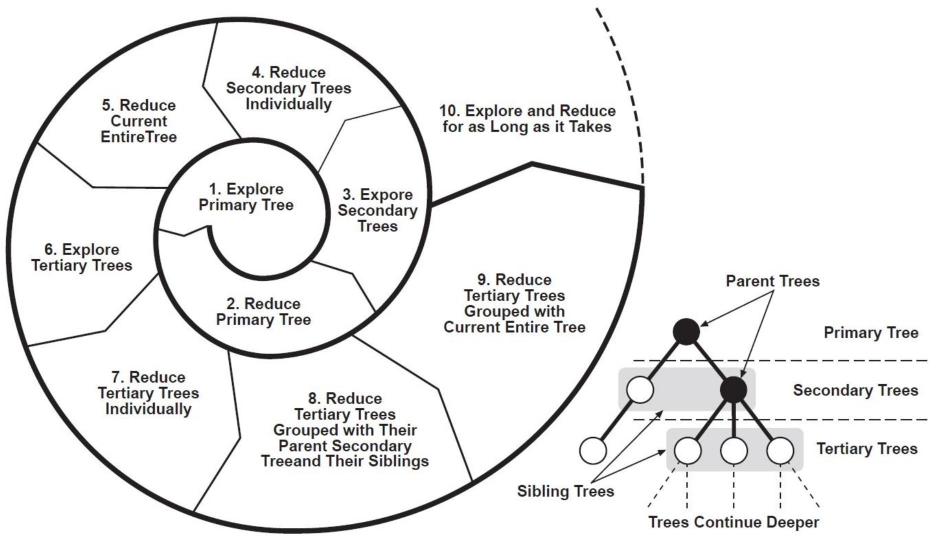
1. Explore Primary Tree
2. Reduce Primary Tree
3. Expore Secondary Trees
4. Reduce Secondary Trees Individually
5. Reduce Current EntireTree
6. Explore Tertiary Trees
7. Reduce Tertiary Trees Individually
8. Reduce Tertiary Trees Grouped with Their Parent Secondary Treeand Their Siblings
9. Reduce Tertiary Trees Grouped with Current Entire Tree
10. Explore and Reduce for as Long as it Takes
Parent Trees
Primary Tree
Secondary Trees
Tertiary Trees
Sibling Trees
Trees Continue Deeper

Figure 4. A graphical representation of the iterative folding process

parent (step 8). This can be seen in the smaller image in Figure 4. After reducing all of the tertiary trees with their respective siblings and parents, we then reduce the entire tree, including the primary, secondary, and tertiary trees, (step 9).

This process repeats, expanding with each level of the tree. If at any point all of the leaf nodes are removed or looped back into the tree, the exploration of new nodes ends and we simply perform the reduction steps.

Therefore, since we don't need the entire tree, we need an algorithm that will work iteratively on subtrees. Let fold$_d$ be the fold procedure while using $T_1 =_d T_2$ as our test for tree equality, rather than $T_1 = T_2$.

Theorem 5.1. *If T is an evaluation tree for M and T_k is T evaluated out to depth k, then $fold_d(T_k) =_{k-d} fold_d(T)$*

Before presenting the proof for Theorem 5.1, let us pause to clarify the notation used. T_k is a tree evaluated k levels deep. The result is a $k-d$ tree with the leaf nodes below depth $k-d$ yet to be evaluted, which means that we are unable to test equality. Therefore, folding a finite tree will produce the same results for the first $k-d$ levels.

Proof. Since the fold procedure uses a breadth first search, the evaluation of fold$_d(T_k)$ and fold$_d(T)$ are identical for the first $k-d$ levels. Therefore, the resulting trees are identical. □

With Theorem 5.1, it is possible to formulate an improved, iterated algorithm outlined in psuedocode in Algorithm 3.

Algorithm 3 Iterative Fold

```
procedure FOLD-ITER(Tree $T$, $\mathbb{N}k$)
    while $T$ has empty descendants do
        Generate $k$ new levels of $T$
        fold($T$)
    end while
end procedure
```

By Theorem 5.1, at each iteration i a new tree with height ik where $T_{ik} =_{ik-d} T$ is created. A FSM is generated when T has no empty descendants remaining, which is to say leaf nodes that do not point back into the tree, and therefore, still have branches to be explored.

Finally, there is one last optimization to add. This algorithm can be parallelized by processing each subtree separately. This requires a theorem slightly more general than the last one. Theorem 5.1 states that running fold$_d(T_k)$ is equivalent to running fold$_d(T)$ for up to $k-d$ levels. Next we want to show that this is true for running fold$_d$ on any subtree of T.

Theorem 5.2. *Let T be an evaluation tree for M, where $D < T$, and D_k is D evaluated out to depth k. Then $fold_d(D_k) =_{k-d} fold_d(D)$. Furthermore, T is still equivalent.*

Proof. The first part follows immediately from Theorem 5.1. For the second part, let D be the ith child of T. That is, there is a path from T to D of i nodes. Now let T_{D_k} be the tree resulting from folding D_k, and let T_D be the tree resulting from folding D. Every subtree in T_{D_k} and T_D that do not include D or D_k are clearly equivalent, so the only subtrees left are the ancestors of D. Let P_D be D's parent and P_{D_k} be D_k's parent, then $P_D \xrightarrow{S_i} D$. Clearly $P_{D_k} =_N P_D$, and $S_j(P_{D_k}) =_{k-d} S_j(P_D)$ for $j \neq i$, and by the last theorem $D_k =_{k-d} D$. Therefore $P_{D_k} =_{k-d+1} P_D$. By induction, all of the ancestors of D are equivalent, therefore $T_{D_k} =_{k-d+i} T_D$. □

With Theorem 5.2 it is possible to process each tree separately and the final tree will still be equivalent.

The ability to divide the tree into smaller subtrees is very conducive to an implementation technique which further speeds the computation along; with this easy division, we can now pass off the subtrees to a distributed computing environment. By spreading the load of the computation over multiple cores and multiple servers, we have been able to explore trees with more than 50 states in less than a minute [19].

6. Future Work

In following work, we plan to explore the following paths:

- Detailing the required comparison depth.
  Currently, the confidence number is just a number with no weight of real-world application behind it. One user may define a comparison depth of 20 as sufficient for a critical system, while another user may require a comparison depth of 100 for the same system. Future work will attempt to provide some baseline numbers to be used for this purpose.
- Methods for implementing this algorithm that increase speed.
  The translation from mathematical notation to a programming language is not always efficient. We are exploring methods for implementing the theorems presented here that result in reduced time or computing cycle usage. We are also considering customized hardware to increase the speed of processing.
- Functional testing on simulated FSMs.
  Preliminary usage testing has been performed on this system, but further testing and abuse is required to ensure the complete veracity of our implementation.
- Methods for broadly assigning meaning to parts of the FSM.
  The theorems detailed here bring us from a tree to a state machine. However, a state machine is still fairly unreadable to a human without the additional information about what internal inputs, outputs, and state transitions relate external impacts.

Other work that may be of interest that builds upon this work includes:

- Remove the assumptions about the lack of memory in the IC.
  One of the initial assumptions for this work was– "An isolated state machine ... the FSM cannot be connected to any sort of non-volatile memory." The reason for this assumption was that the inclusion of memory creates a possible number of inputs as large as the bits in the memory; this increased the complexity of the processing enormously. However, through use of distributed or cloud computing, an exploration of this may become feasible.
- Breadth-first exploration.
  We have been working from the assumption that the tree will be parsed in a breadth-first manner. In the general case, this has been proven to be the most efficient. However, there may be edge cases in which another tree exploration method may be desired or needed.

7. Conclusion

The above method demonstrates that given an isolated state machine with a reset capability, we can model the machine using a tree framework which allows for machine to machine comparisons and the comparison of states within the machine to find an optimal representation. Through this method it is possible to take a state machine-based IC and, using only the standard input and output pins, re-discover the original FSM. Consequently, we can determine if the in-silicon FSM matches the designed FSM, or rediscover the functionality of an unknown IC. Both capabilities provide a non-destructive means of validation for security purposes.

References

[1] S. Pope, "Trusted integrated circuit strategy," in *IEEE Transactions on Components and Packaging Technologies*, vol. 31, Mar. 2008, p. 230233.

[2] E. Chikofsky and I. Cross, J.H., "Reverse engineering and design recovery: a taxonomy," *Software, IEEE*, vol. 7, no. 1, pp. 13–17, Jan 1990.

[3] S. Wei and M. Potkonjak, "The undetectable and unprovable hardware trojan horse," in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, May 2013, pp. 1–2.

[4] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems, CHES 01*, 2001, p. 251261.

[5] M. Yamashita, K. Kawase, C. Otani, T. Kiwa, and M. Tonouchi, "Imaging of large-scale integrated circuits using laser terahertz emission microscopy," *Optics Express*, vol. 13, January 2005.

[6] R. Diestel, *Graph Theory*. Springer-Verlag, 2010.

[7] G. Masalskis and R. Navickas, *Reverse engineering of CMOS integrated circuits*. Electronics and Electrical Engineering, 2008.

[8] T. Nagatsuma, M. Shinagawa, M. Yaita, and K. Takeya, "Electro-optic probing technology for ultrahigh-speed IC diagnosis," in *Instrumentation and Measurement Technology Conference*, May 1994, p. 14761483vol.

[9] M. Razeghi, *Semiconductor Characterization Techniques*, 3rd ed. New York, NY: Springer US, 2009.

[10] J. Ferrigno and M. Hlavac, *When AES blinks: introducing optical side channel*. Information Security, IET, 2(3):9498, Sep. 2008.

[11] E. F. Moore, *Gedanken-experiments on sequential machines*. Automata studies, 1956, vol. 34.

[12] M. Brutscheck, M. Franke, A. Schwartzbacher, and S. Becker, "Investigation and implementation of test vectors for efficient ic analysis," in *Signals and Systems Conference*. Becker, 2008.

[13] M. Brutscheck, M. Franke, S. Becker, and A. Schwartzbacher, "Structural division procedure for efficient IC analysis," in *International Solid State Circuits Conference*, Jun. 2008, p. 1823.

[14] ——, "Optimisation and implementation of a nonlinear identification procedure of unknown ics," in *International Solid State Circuits Conference*. Becker, 2009.

[15] M. Brutscheck, B. Schmidt, M. Franke, A. Schwartzbacher, and S. Becker, "Identification of deterministic sequential finite state machines in unknown cmos ics," in *International Solid State Circuits Conference,*. Becker, 2010.

[16] M. Brutscheck, "Systematic analysis of unknown integrated circuits," Ph.D. dissertation, Dublin Institute of Technology, 2009.

[17] S. Dabral and T. Maloney, *Basic ESD and I/O Design*. Wiley Interscience, 1998.

[18] W. Greason and K. Chum, "Experimental determination of ESD latent phenomena in CMOS integrated circuits," *IEEE Transactions on Industry Applications*, vol. 28, no. 4, pp. 755–760, Jul 1992.

[19] J. Smith, "Non-destructive state machine reverse engineering," in *International Symposium on Resilient Control System*, 2013.

[20] ——, "A Non-destructive Analysis Method for Integrated Circuit-based Finite State Machines," Ph.D. dissertation, Washington State University, 2016.

[21] F. Harary, *Graph Theory*. Addison-Wesley, 1969.

8. Acknowledgments

This work is supported by the Pacific Northwest National Laboratory's Supply Chain Integration For Integrity (SCI-FI) project, funded by the Department of Energy, Office of Electricity Delivery and Energy Reliability, Research and Development Division, RC-CEDS-2012-02.
PNNL-SA-100838