PNNL-25330

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

Reverse Engineering Integrated Circuits Using Finite State Machine Analysis

Kiri Oler
Carl Miller

March 2016

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by **Battelle** *Since 1965*

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor Battelle Memorial Institute, nor any of their employees, makes **any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights**. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or Battelle Memorial Institute. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information,
P.O. Box 62, Oak Ridge, TN 37831-0062;
ph: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov

Available to the public from the National Technical Information Service,
U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161
ph: (800) 553-6847
fax: (703) 605-6900
email: orders@ntis.fedworld.gov
online ordering: http://www.ntis.gov/ordering.htm

This document was printed on recycled paper.
(9/2003)

PNNL-25330

Reverse Engineering Integrated Circuits Using Finite State Machine Analysis

Kiri Oler
Carl Miller

March 2016

Prepared for
the U.S. Department of Energy
under Contract DE-AC05-76RL01830

Pacific Northwest National Laboratory
Richland, Washington 99352

Reverse Engineering Integrated Circuits Using Finite State Machine Analysis

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. 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 [1], and non-destructive characterization techniques like power usage, timing delays, current leakage, and EM imaging which can be used to certify an IC against a known-good IC. [7, 3, 9].

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 [2], 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.

We begin by discussing the motivation behind this work in greater detail. We then present our contribution to the problem by first providing the mathematical foundation and then outlining the software and hardware implementation used to test our theory.

2 Motivation

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. [6] 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.

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. How-

ever, 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.

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).
- 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

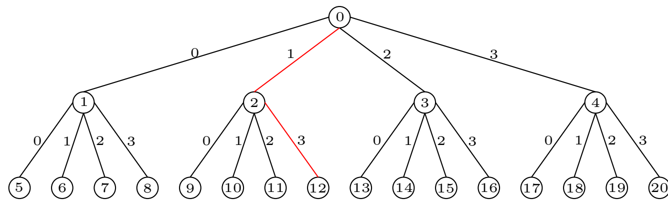


Figure 1: Example of a tree with nodes and edges labeled and an example path highlighted in red.

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. 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 $\mathbf{0}$. As with the formal definition of a tree, each tree is a set N with c functions defined as follows:

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

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 $P = p_0, p_1, \ldots, p_{d-1}$ 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 $labels = l_0, l_1, \ldots, l_d$, 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:

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

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:

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

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 [5], 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 $\mathbf{0}$ 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 $\mathbf{0}$ or $T_1 =_N T_2$. Additionally, $T_1 =_d T_2$ if and only if T_1 and T_2 are both $\mathbf{0}$ 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.

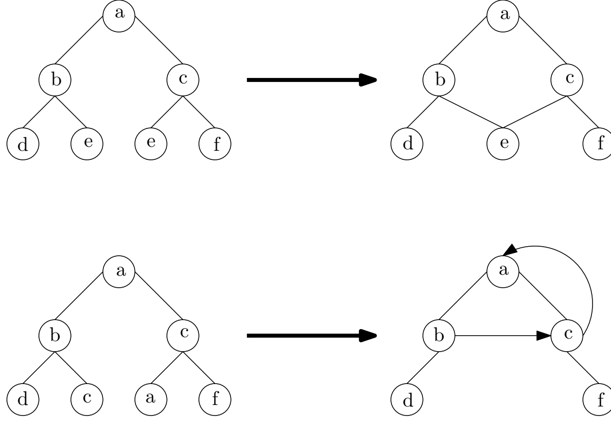


Figure 2: Two simple examples of the folding procedure.

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 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 $N = M$. ☐

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 $M = M'$. ☐

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 ← ∅
    q ← T
    while q ≠ ∅ do
        t ← q
        if ∃x ∈ seen : t = x then
            switch(t, x, seen)
        else
            seen ← t
            q ← l(t)
            q ← r(t)
        end if
    end while
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.*

Algorithm 2 Switch

```
procedure SWITCH(Tree a, Tree b, Set⟨Tree⟩ seen)
    for x ∈ T do
        if l(x) is a then
            l(x) ← b
        end if
        if r(x) is a then
            r(x) ← b
        end if
    end for
end procedure
```

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

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. □

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 [4], 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 $\mathrm{Diam}(M) \le d$.*

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, $\mathrm{Diam}(M) \ge e > d$. □

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 Confidence Levels

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? This point will hereafter be referred to as confidence level. Within the context of evaluation trees, the confidence level 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 confidence level 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 confidence level. To formalize this concept, let the depth, $d = cl$, represent the necessary confidence level. 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 confidence level 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 confidence level depth,

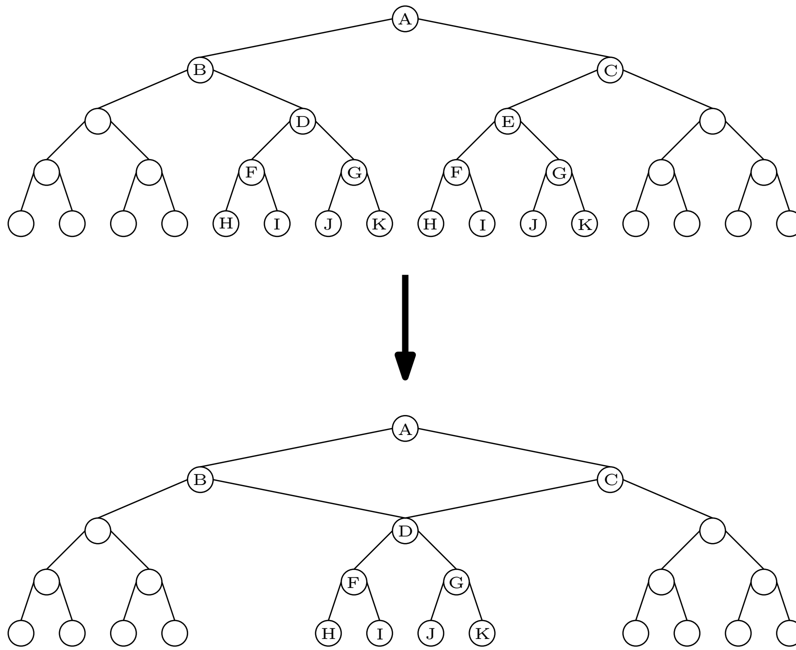


Figure 3: An example of a fold procedure performed with a confidence level of 2.

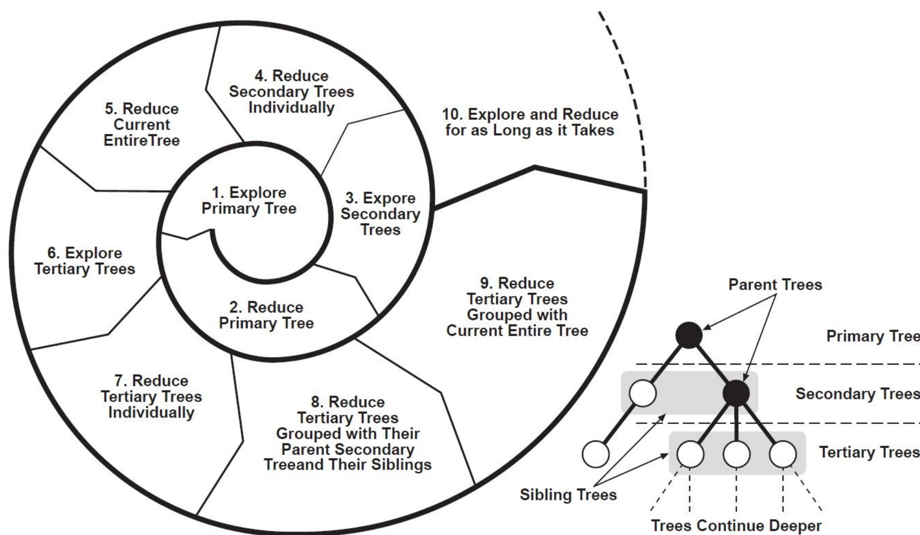


Figure 4: A graphical representation of the iterative folding process

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 confidence level 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 confidence level plus at least one. A depth of the confidence level 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 confidence level 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 confidence level 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 confidence level 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 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 $\text{fold}_d(T_k)$ and $\text{fold}_d(T)$ 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, ℕk)
    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 $\text{fold}_d(T_k)$ is equivalent to running $\text{fold}_d(T)$ 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 [8].

6 System Design

6.1 Overview

Given that our reverse engineering methodology has the goal of understanding a state machine-based IC, our implementation takes the following approach. The first step in analyzing an ICs functionality is to determine which pins are connected, i.e, which are input, output, clock and reset. The next step in determining the state machine upon which an IC is built is filling out a state tree of all possible input-output streams. For a single-input IC, each node in the tree has at most two children (i.e. one child for each possible value of the input pin); for a two-input-pin IC, each node in the tree has at most four children, and so on. From this tree, we can use reduction techniques and formal methods in our algorithm to reduce and prove identical states, bringing us to an approximation of the original state machine.

Figure 5 illustrates the components of the hardware evaluation system which consists of an OpenMpi distributed system comprised of a single Dell PowerEdge 2950 master server (referred to hereafter as "the Boss") and several PowerEdge slave servers (referred to hereafter as "the Minions") along with an FPGA-based hardware test harness (referred to hereafter as "the IC Harness"). The IC to be evaluated is placed into a socket in the IC Harness (which is connected to the Boss server via a USB cable) and when instructed by software running on the boss, performs pin profiling and state machine branch exploration of the IC state machine. To summarize at a high level how the hardware is used, pin profiling is the initial step, then using the input and output pin definitions obtained from the pin profiling process, multiple iterations of state machine branch exploration commands would be issued by the Boss to the IC Harness, gradually building up and pruning the state tree. As the Boss retrieves the branch exploration data (which represents the output states for the corresponding input stimuli being sent to the IC Harness), the data (usually subtree information) is parceled out to the Minion servers for processing with each Minion's results being amassed by the Boss into a single tree. This process of requesting the IC Harness assert a series of input stimuli and return the corresponding output states, using the output states to build up nodes in the state tree that is then pruned, folded and reduced continues until the IC has been explored to a pre-defined depth (the confidence level) at which point the final estimated state machine tree is complete.

6.2 Software Implementation

After establishing the mathematical foundation, we implemented the software side of our approach primarily in C++. Python scripts were used in order to test and validate proper operation of the hardware testing components and USB communications. Our initial implementation was limited to processing hard-coded trees with rigid input requirements and processing methods. Now the software incorporates multiple options for inputs and processing. Via flags passed on the command line, the user can set the maximum depth of the tree to be explored, the confidence level to be used in the folding algorithm, choose whether to use the parallelized or non-parallelized version of the code, specify the input file containing the required IC information to build the tree, and set options for what information should be included when writing the trees to output (e.g. node ID/address/state/depth), as well as the output directory and how often the trees should be written throughout the process.

The software has two options for constructing the tree representing the FSM in question. The first uses information provided by an IC definition file. This option was implemented for testing and experimentation purposes. Using a set defintion file where the behavior of the chip is controlled and predictable, we can verify that the output from the folding process is as expected. The definition file has two main components: general information about the IC (primarily the number of inputs expected for each state and the number of outputs given by each state) and the truth table for the IC. The truth table functions similarly to a conventional logic truth table in that it enumerates every possible combination of inputs and their respective outputs. From this the code can determine the possible transitions between states based on all possible inputs and construct a tree to process through the folding algorithm. This method of tree construction is referred to hereafter as FSM mode, since it operates from a known

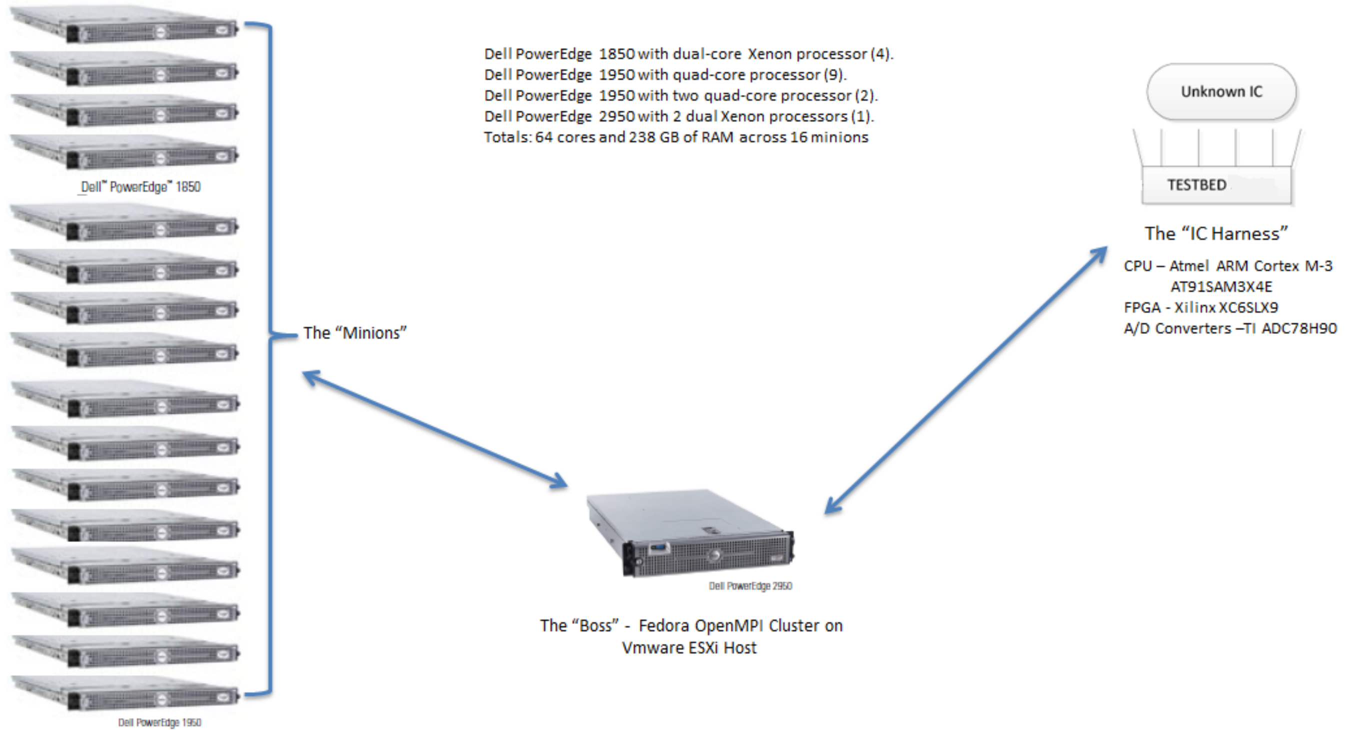


Figure 5: Diagram of the hardware layout used for testing the algorithm.

FSM rather than directly with an IC.

The second option is to communicate directly with the hardware being used to analyze the IC. In this scenario, the state transition information is gathered directly from the IC using a pin profiling process described in further detail in the next section. This method is envisioned to be a small scale version of the practical implementation of our approach, i.e. how we see our approach being implemented in the real world.

To visually follow the impact and/or progress of the folding algorithm the before and after versions of the tree representing the FSM, as well as intermediate states, if specified via command line arguments, are written using the graph description language DOT. Using small examples (relatively speaking) for testing purposes, we compared the unfolded (before) version of the tree to the folded (after) version of the tree in order to see how much redundancy was removed, and therefore, efficiency gained. Additionally, we compared the fully folded version of the tree to a known IC modeled as tree in order to verify the correctness of the algorithm. Figure 6 shows the before version of a tree modeling the FSM of an IC with a single input pin, and initially explored to a depth of 5. Figure 7 shows the graph output by the folding algorithm after processing the tree in Figure 6 using a confidence level of 3.

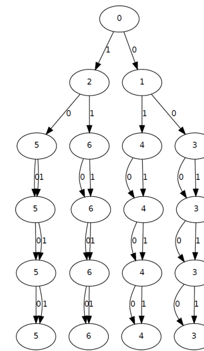


Figure 7: Graph output by the folding algorithm using a confidence level of 3.

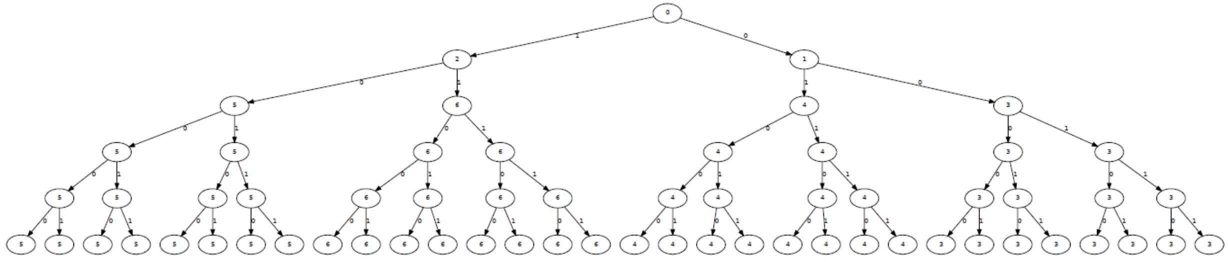


Figure 6: Tree modeling the FSM of single pin IC explored to a depth of 5 before any folding is done.

6.3 Hardware Implementation

Moving to the hardware side of the implementation, the functioning of an integrated circuit is explored using an ARM CPU and a FPGA testing unit (IC Harness), which characterizes pin functions via pin profiling, then communicates with the Boss running the software described in the previous section to create a FSM model using branch exploration. This exploration is done via a stimulus/response process, which will be described in the remainder of this section.

6.3.1 Pin Profiling

First, the IC Harness uses a pin profiling process to gather required information about the IC. It does so in four phases:

1. Take two voltage measures for each pin.
   For the first measure, the pin in question is grounded, then the other pins are pulled toward power through their pull-up resistors and the sum of their voltages constitutes the measure. Next, all pins except the one in question are grounded, while it is pulled toward the power, and its voltage is stored as the second measure.

2. Identify the power and ground pins.
   Using the measures from the previous phase we can determine which of the pins is the power pin and which is the ground pin. The power pin is the one yielding the lowest sum of voltages from the other pins while grounded. Meanwhile, the ground pin yields the lowest individual voltage while the other pins are grounded.

3. Determine which pins are input pins and which pins are output pins.
   The averages of the two measures calculated in the first phase (excluding the power and ground pins) are used to categorize the remaining pins as either input pins or output pins. If the voltage of the ground pin is less than the average, we can infer that the input pins are those whose summed voltage values are below the average and the output pins are those whose summed voltage values are above the average. Conversely, if the voltage of the ground pin is greater than the average, we can infer that the input pins are those whose summed voltage values are above the average and the output pins are those whose summed voltage values are below the average.

4. Find the clock and reset pins.
   We assume that the output starts at zero and iterate over all possible combinations of inputs and for each combination one pin at a time is toggled between 0 and 1. The clocking state refers to the combination of inputs that trigger a nonzero output. The clocking pin is the pin that was toggled on in combination with the particular permutation of inputs that yielded the nonzero output. The reset pin is the pin within the clocking state that when toggled on resets all outputs to zero.

6.3.2 Internal Logic Testing

With the knowledge of each pin's role within the IC (i.e. input/output/power/ground/reset/clocking), it is now possible to explore the IC's functionality and determine the FSM upon which it is based and eventually construct a tree modeling the FSM. The IC Harness communicates with the software side via the ARM CPU, which can both take commands and deliver response data. Initially, all input stimulus strings were generated by the software and transmitted to the testing unit; however, we discovered efficiency is greatly improved if the stimulus requests necessary for exploration of the IC are generated by the FPGA testing unit's ARM CPU in response to branch exploration (i.e. exploring a specific path through the tree) requests from the Boss.

A given branch exploration stimulus request is processed as follows: When the request is first received by the IC Harness, the CPU signals the FPGA to clear its input and output queues and waits until they are cleared. Next, using the most recent pin-profile output, and the starting node address provided by the request packet, the CPU calculates the unique series of input stimuli to put the IC into the state represented by the starting node address. Next, each possible combination of input settings (i.e, for an IC with 2 input pins, this combination is 00,01,10, 11) is processed as follows, the FPGA is instructed to reset the IC, execute the input stimulus string corresponding to the start node address, and then apply one of the input pin combinations and record the resulting output pin states as an integer value. The previous process generates the child nodes of the start node specified in the request packet. The FSM state of each child node is equal to the value of the output pin value. Repeated iterations of branch explorations combine to construct the FSM's representative tree, which will be discussed more thoroughly in the next section.

6.3.3 State Exploration

The exploration of the representative tree is done breadth-wise using a series of spurts. A spurt is a sequence of stimulus streams (or paths) starting from a parent node, one for each possible permutation of the inputs, with a reset in between. The stimulus generator takes a desired parent node address (as the parent node may not be the root node) and creates the input stimulus using Equation 3.3, which finds a path from the initial state to the given node label. For non-root starting nodes, the spurt will include the stimulus needed to get from the root to the desired parent node address, which will be inserted between the reset and the spurt stimulus. Figure 8 demonstrates step-by-step the exploration process, for a 2-input pin IC, depth of one branch exploration, by showing the tree diagram in the left column and the testing unit's internally generated stimulus and returned response in the right column. The "Request" column represents the input stimulus string sent fom the ARM CPU to the FPGA. The "Spurt" component represents one of the possible input pin state combinations, while the "Stimulus" component of this column represents the input stimulus string for the start node address. The "Response" column represents the information to be returned to the Boss server in response to a branch exploration request. The "Spurt" component represents the node address of a child node of the start node while the "Stimulus" component of this column represents the input pin state of the child. The output pin state is not reflected in the table but is returned with the child data in the response packet sent via USB to the Boss. This table represents a tree branch exploration using a depth of one, we refer to this case of depth of one as a "Burst" exploration. In future work we may extend support for branch explorations to greater depths.

6.3.4 Communication Protocol

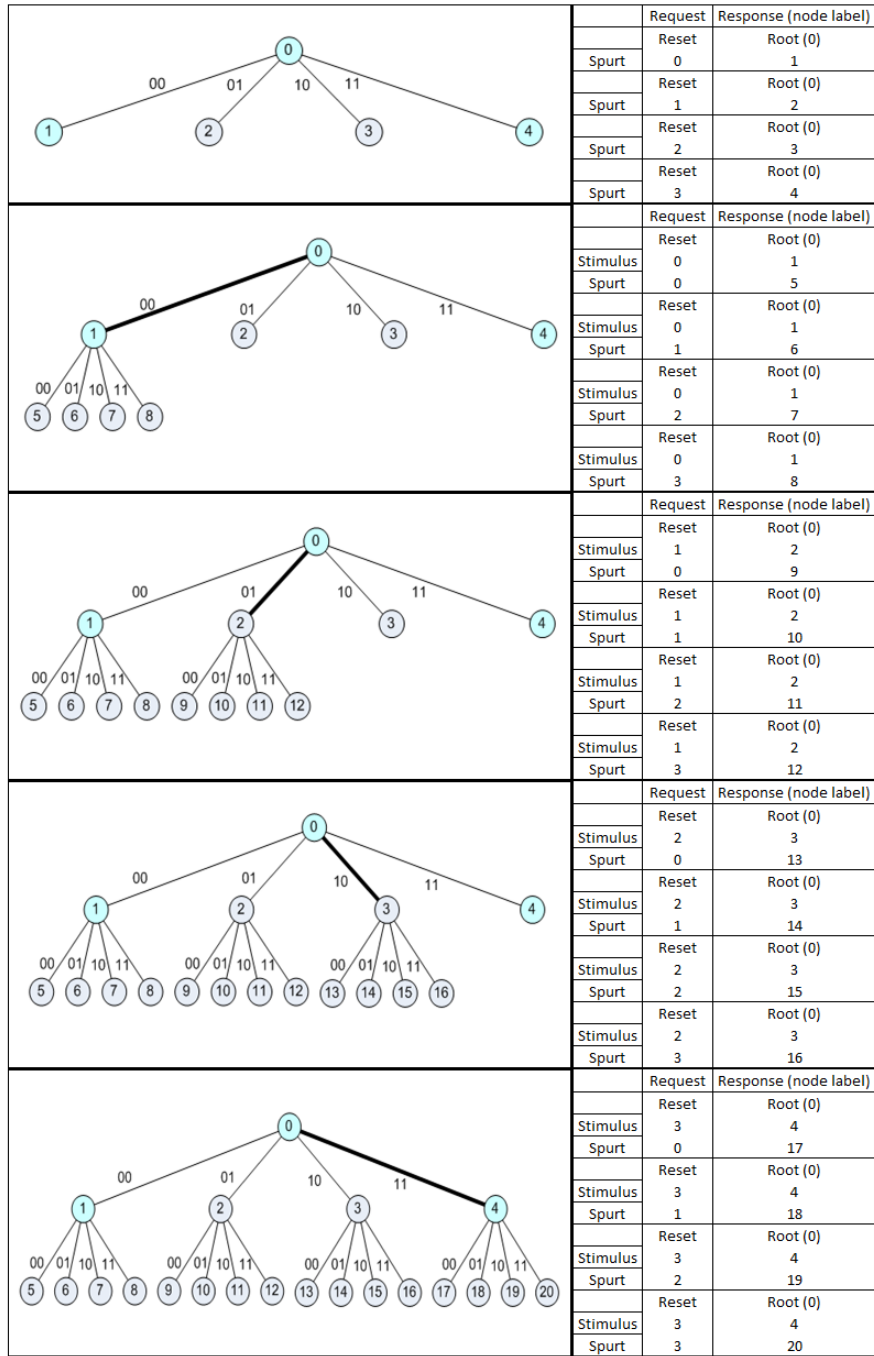
In order to send and receive the request described above as necessary to the exploration process, a specific protocol is required. Packets following the protocol have several required fields, outlined as follows:

- Command: This is the operation to be performed by the testing unit. Possible commands include: reset, obtain pin profile, conduct branch exploration, switch to FSM mode, and run a diagnostic.
- Status: Used by the testing unit to positively acknowledge requests.
- Parameter: This field is used differently based on the command. For branch exploration, the depth of the tree to explore to would be specified. While in FSM mode, this contains the table ID number for the FSM to be used. Otherwise, this field transmits any error codes.
- Flags: Used to indicate the existence of extended data meaning that the amount of response data exceeds what can be represented by the 32-bit Data Length field. The additional length (not including what is specified in the Data Length field) is provided by a 4-byte field which will immediately follow the header. If operating in FSM mode, this field is used to indicate the format of the input table.
- Data Length: Specifies the length of response data to be returned by a branch exploration command. When responding to a pin profile command, this field specifies how many pins the IC has.
- High Octet: Serves as either the address of the starting node or the current state of the FSM when executing exploration commands and contains the response data returned from the testing unit when executing pin profile and diagnostic requests.

7 Future Work

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

- Detailing the required confidence level.
  Currently, the confidence number is just a number with no weight of real-world application behind it. One user may define a confidence level of 20 as



	Request	Response (node label)
	Reset	Root (0)
Spurt	0	1
	Reset	Root (0)
Spurt	1	2
	Reset	Root (0)
Spurt	2	3
	Reset	Root (0)
Spurt	3	4

	Request	Response (node label)
	Reset	Root (0)
Stimulus	0	1
Spurt	0	5
	Reset	Root (0)
Stimulus	0	1
Spurt	1	6
	Reset	Root (0)
Stimulus	0	1
Spurt	2	7
	Reset	Root (0)
Stimulus	0	1
Spurt	3	8

	Request	Response (node label)
	Reset	Root (0)
Stimulus	1	2
Spurt	0	9
	Reset	Root (0)
Stimulus	1	2
Spurt	1	10
	Reset	Root (0)
Stimulus	1	2
Spurt	2	11
	Reset	Root (0)
Stimulus	1	2
Spurt	3	12

	Request	Response (node label)
	Reset	Root (0)
Stimulus	2	3
Spurt	0	13
	Reset	Root (0)
Stimulus	2	3
Spurt	1	14
	Reset	Root (0)
Stimulus	2	3
Spurt	2	15
	Reset	Root (0)
Stimulus	2	3
Spurt	3	16

	Request	Response (node label)
	Reset	Root (0)
Stimulus	3	4
Spurt	0	17
	Reset	Root (0)
Stimulus	3	4
Spurt	1	18
	Reset	Root (0)
Stimulus	3	4
Spurt	2	19
	Reset	Root (0)
Stimulus	3	4
Spurt	3	20

Figure 8: Diagram showing the successive exploration of a tree representing a 2-pin IC

sufficient for a critical system, while another user may require a confidence level 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.

8 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 accepted 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] Elliot J Chikofsky, James H Cross, et al., *Reverse engineering and design recovery: A taxonomy*, Software, IEEE **7** (1990), no. 1, 13–17.

[2] Reinhard Diestel, *Graph Theory*, Springer-Verlag, Heidelberg, 2010.

[3] Karine Gandolfi, Christophe Mourtel, and Francis Olivier, *Electromagnetic analysis: Concrete results*, Cryptographic Hardware and Embedded SystemsCHES 2001, Springer, 2001, pp. 251–261.

[4] Frank Harary, *Graph Theory*, Addison-Wesley, Reading, Massachusetts, 1969.

[5] Edward Moore, *Gedanken-experiments on Sequential Machines*, Automata Studies: Annals of Mathematics Studies (1956), no. 34.

[6] Sydney Pope, *Trusted integrated circuit strategy*, IEEE Transactions on Components and Packaging Technologies (2008), no. 31.

[7] Miodrag Potkonjak, Ani Nahapetian, Michael Nelson, and Tammara Massey, *Hardware Trojan horse detection using gate-level characterization*, Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE, IEEE, 2009, pp. 688–693.

[8] Jess Smith, *Non destructive state machine reverse engineering*, ISRCS, 2013.

[9] Xiaoqing Wen, Yoshiyuki Yamashita, Shohei Morishima, Seiji Kajihara, Laung-Terng Wang, Kewal K Saluja, and Kozo Kinoshita, *Low-capture-power test generation for scan-based at-speed testing*, Test Conference, 2005. Proceedings. ITC 2005. IEEE International, IEEE, 2005, pp. 10–pp.

9 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