State Register Identification for Circuit Reverse Engineering

Aidan Wong | a.wong71720@gmail.com
DesCyPhy Lab
Alhambra High School | Class of 2023
USC Viterbi Department of Electrical Engineering | SHINE 2022

USC Viterbi School of Engineering

SHINE Summer High School Intensive in Next-Generation Engineering

INTRODUCTION

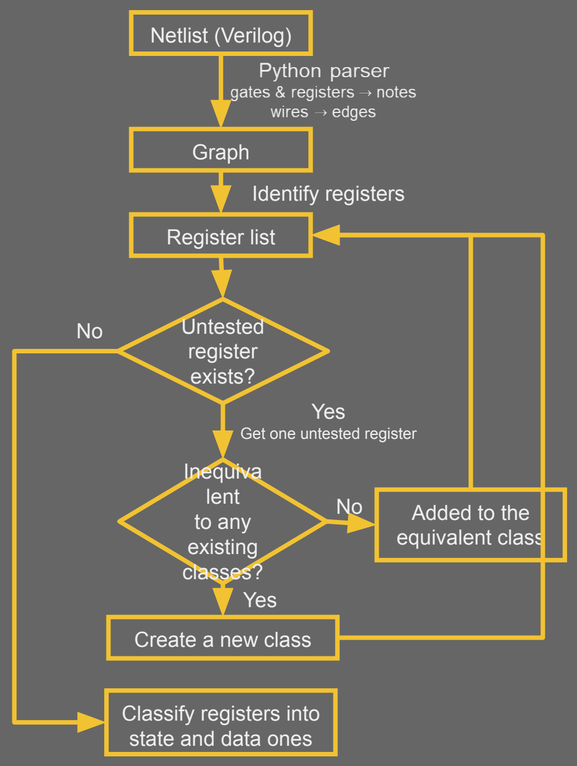
Modern integrated circuits pass through many hands from initial design to release into the supply chain, and typically contain third party intellectual property (IP).

Often, a design will use not just individual cells but large pregenerated IP cores for which the engineer may not have source code. The only access the engineer has to the third party circuitry is the netlist after the design has been compiled and synthesized. Netlists may also be recovered from fabricated chips returned from the foundry.

In either of these cases, malicious code may have been inserted into the design, either via extra logic in a third party IP core, or changes to the design by a malicious foundry. Verification that the design functions only as intended is difficult. The DesCyPhy Lab is working towards securing next generation microelectronic chips.

METHODS

There are several existing state register identification algorithms for reverse engineering of finite state machines. We implement and investigate one algorithm called RELIC-FUN, and analyze its performance by comparison with other algorithms. As opposed to other methods such as topological matching (RELIC) or identification using graph neural networks (ReIGNN), RELIC-FUN utilizes functional matching.



IMPLEMENTATION

Two algorithms derived from the pseudocode of RELIC-FUN were implemented to obtain the desired results.

1) The first algorithm explains how we classify registers.
   a) Finding *k*-slice: A slice of this form is a feasible cut of the output Y, and we call a conforming slice of k inputs *k*-feasible.
   b) Sorting signals & classes
      i) All registers in the target list are our target signals;
      ii) For each target signal, we identify one *k*-feasible slice of it, where *k* is specified by users;
      iii) Functional equality between two *k*-slices are decided by the second algorithm, and signals with functionally equivalent *k*-slices will be classified into the same class;
      iv) Signals in the classes with larger size are identified as data registers, while others are identified as state registers.

2) The second algorithm defines functional equality.
   To understand whether two signals are functionally equivalent or not, we should test them under all possible situations. Therefore, we propose three termination criteria.
   a) Different input lengths: If the input length of two slices are different, they are inequivalent;
   b) Different input permutations: If the inputs of two *k*-slices are just in different permutations, they are equivalent;
   c) Simulation-based equivalence: If the inputs of two *k*-slices are different, but the simulation results are the same for all possible input patterns, they are equivalent.

3-feasible slice

Algorithm 1 Compute equivalence classes

```
function relic-fun(signals, target − size)
    classes ← []
    for s ∈ signals do
        sl ← slice(s, target − size)
        found ← false
        for cl ∈ classes do
            t ← cl.first
            if ttequal(sl, t) and not found then
                add(cl, sl)
                found ← true
            end if
        end for
        if not found then
            newset ← set(sl)
            add(classes, newset)
        end if
    end for
    sort(classes, λx : x.size)
```

Pseudocode of the first algorithm

Algorithm 2 Test functional equality

```
function ttequals(A, B)
    if |A.inputs| ≠ |B.inputs| then
        return false
    end if
    for p ∈ permutations(|A.inputs|) do
        match ← true
        for i ∈ combinations({0, 1}, |A.inputs|) do
            if A(i) ≠ B(p(i)) then
                match ← false
            end if
        end for
        if match then
            return true
        end if
    end for
    return false
```

Pseudocode of the second algorithm

```
## test functional equality with all existing classes
for c in classes:
    ## select one signal from each class
    reference = classes[c][0]
    ## test the functional equality between them
    equiv = test_functional_equality(G, reference, \
        kslice[reference], kslice_inputs[reference], \
        item, kslice[item], kslice_inputs[item])
    ## add to the same class if functionally equivalent
    if equiv:
        classes[c].append(item)
        class_found = True
        break
if class_found == True:
    logging.info("find equivalent class")
    continue
## otherwise creating a new class for the signal
else:
    logging.info("create new class")
    classes[classes_idx] = []
    classes[classes_idx].append(item)
    classes_idx += 1
    continue
```

```
def test_functional_equality(G, ref_node, ref_slice, ref_inputs, target_node, target_slice, target_inputs):
    ref_sim_result = {}
    ## if the length of inputs are different -> unequal
    if len(ref_inputs) != len(target_inputs): return False
    ## if the input lists are differnt permutations of the same input group -> equal
    elif compare_lists(ref_inputs, target_inputs): return True
    ## test equality based on simulation results
    else:
        for perm in permutations(list(range(0,len(ref_inputs)))):
            target_sim_result = {}
            equiv = True
            for v in range(2**len(ref_inputs)):
                if v not in ref_sim_result:
                    input_pattern = list("{:03b}".format(v)) ## FIXME: number of bits is fixed here
                    input_pattern.reverse()
                    ref_sim_result[v] = simulate(G, ref_node, ref_slice, ref_inputs, input_pattern)
                rv = reorder(v, list(perm))
                if rv not in target_sim_result:
                    input_pattern = list("{:03b}".format(rv))
                    input_pattern.reverse()
                    target_sim_result[rv] = simulate(G, target_node, target_slice, target_inputs, input_pattern)
                if ref_sim_result[v] != target_sim_result[rv]:
                    equiv = False
                    break
            if equiv: return True
        return False
```

Our implementation of the two algorithms using Python

METRICS

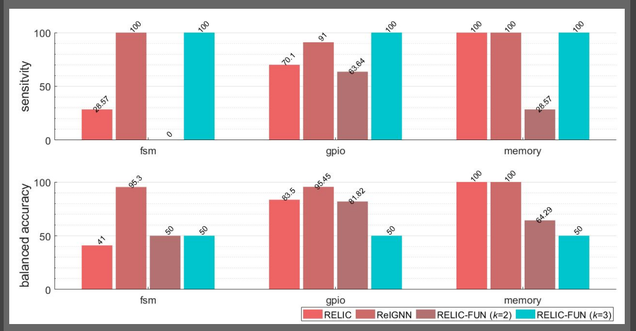
We use two metrics, sensitivity and balanced accuracy, defined as follows, to evaluate the performance of RELIC-FUN, and compare it wither other two algorithms, RELIC and ReIGNN.

Sensitivity: a measure of how well a test can identify true positives $\left(\frac{\text{\# correctly identified state registers}}{\text{\# true state registers}}\right)$

Specificity: a measure of how well a test can identify true negatives $\left(\frac{\text{\# correctly identified data registers}}{\text{\# true data registers}}\right)$

Balanced accuracy: the mean of sensitivity and specificity

RESULTS & ANALYSIS



To evaluate the performance of RELIC-FUN, we compare its predictions with the true labels of the nodes. Here *k* is one parameter specified by the users, and it defines how large slice of each signal we will use to test their functional equality. Two *k* values, 2 and 3, are used in our experiments to show how the value of *k* impacts the results. The sensitivity of RELIC-FUN (*k*=3) is 100% in all three cases, while RELIC and ReIGNN cannot achieve 100% for the gpio circuit. When reverse engineering the finite state machine (FSM) of a design, achieving 100% sensitivity is important because any missing state registers can lead to an incomplete FSM.

NEXT STEPS

According to the experimental results, RELIC-FUN is conservative to tell two registers are functionally equivalent, and that leads to a bad performance of balanced accuracy. Also, RELIC-FUN's performance is sensitive to the parameter *k*. To overcome these challenges, we could

a) Randomly select multiple possible k-slices for each target signal instead of one to avoid incomplete equality testing;
b) Automatic the process of choosing the value of *k*.

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation and gratitude to the following people, all of whom have made this great research experience possible…

to Professor Nuzzo who accepted me into the program and allowed me to experience electrical engineering research
to Dr. Katie Mills who runs a wonderful program, allowing many to meet each other and experience much
to the friends I made here at USC who made me laugh and enjoy every moment of the internship
to Ph.D. student Kaixin Yang, my mentor, who answered every question I had and gave me a lot of insight into the process
and to my family who have supported and helped me along the way in my journey to becoming an engineer.