# A Stochastic Computational Approach for Accurate and Efficient Reliability Evaluation

## A Python Implementation

Jake Humphrey

Department of Electronic and Electrical Engineering
Imperial College London
jbh111@ic.ac.uk

November 19, 2014

# Reliability of Circuits

Gates in a logic circuit are, alas, not perfect. They are susceptible to error, of which there are three main types:

- **Stuck-At-One Error:** The output of the gate goes high, regardless of the expected output.
- **Stuck-At-Zero Error:** The output of the gate goes low, regardless of the expected output.
- **Von Neumann Error:** The output of the gate becomes the inverse of the expected output.

# Masking Effects

However, there is a chance that errors in one gate will not propagate all the way to an output. This could be due to one of the following *masking effects*

- **Electrical Masking:** The error does not have a large enough effect on the amplitude of the logic signal to be detected at an input.
- **Temporal Masking:** The error is input to a latch but occurs at some point in time outside of the latch's detection window.
- **Logical Masking:** The error does not pass through a multi-input logic gate because the value of the other input(s) fix(es) the output of the gate.

# Reliability Analysis Principles

As it happens, Logical Masking is the most prominent masking type in logical circuits. It is therefore useful to be able to analyse circuits on their ability to logically mask errors.

If we define the *probability* of a signal as the proportion of time that it is logically True, then the basic idea is as follows:

- Construct a faulty representation of the circuit, which takes into account probabilities of each gate failing.
- Derive the probabilities of the output signals.
- Then the *reliability* of an output signal is the probability that it takes the same value in both the ideal and faulty circuits.

# Reliability Analysis Probabilistic Gate Models

However, existing algorithms are inefficient!

For example, *Probabilistic Gate Models* (PGMs) attempt to analytically derive the output probabilities as functions of the input probabilities and gate error probabilities.

The problem occurs when the inputs to a gate are not statistically independent, such as is the case when there are *reconvergent fanouts*. That is, when two or more inputs to a gate originated from a single signal.

The PGM equations do not account for statistically dependent signals, and the solution involves splitting the circuit into two sub-circuits. This approximately doubles the cost of the algorithm for each reconvergent fanout.

# Reliability Analysis Stochastic Logic

The use of *Stochastic Logic* can avoid these issues. With this approach, the input probabilities are used to generate input bitstreams, which are then propagated through the circuit. The output probabilities can then be accurately calculated from the output bitstreams.

# Reliability Analysis Stochastic Logic with Bernoulli Sequences

Existing Stochastic Logic algorithms use the input probabilities to generate *Bernoulli Sequences* of the form:

$$[X_0, X_1 \ldots X_{n-1}]; X_i \sim B(p)$$

for each input, where $p$ is the input probability.

However, this approach incurs a large computational overhead, as $n$ random numbers must be generated for each input. This can be significant for large circuits, because $n$ must be large to obtain accurate results.

# Reliability Analysis Stochastic Logic with Non-Bernoulli Sequences

To reduce the random number generation overhead, *Non-Bernoulli Sequences* can be used.

These sequences are generated deterministically with the expected number of 1s, and then randomly permuted.

This means only one random number generation is required per input bitstream.

# Reliability Analysis Algorithm

The information on the previous slides gives rise to an algorithm for Reliability Analysis using Stochastic Logic with Non-Bernoulli Sequences.

I have written a Python implementation of this algorithm, which I will describe over the following slides.

# Reliability Analysis Algorithm Pseudocode

**Input:** Logic circuit to be tested
**Output:** Reliabilities for each output

*faulty_circuit* $\leftarrow$ *circuit*
**for** gate in circuit.gates **do**
    *faulty_circuit.gates.add*(*error_gate*)
    *faulty_circuit.inputs.add*(*error_signal*)
**end for**
**for** input in faulty_circuit.inputs **do**
    *input.stream* $\leftarrow$ [*False*] $\times$ *SEQLEN*
    *input.stream*[0 : *input.prob* $\times$ *SEQLEN*] $\leftarrow$ *True*
    *shuffle*(*input.stream*)
**end for**

# Reliability Analysis Algorithm Pseudocode

**for** output in circuit.outputs **do**
    $correct \leftarrow []$
    **for** i from 0 to SEQLEN-1 **do**
        $sigin \leftarrow circuit.inputs[i]$
        $trueval \leftarrow propagate(sigin, circuit, output)$
        $faultval \leftarrow propagate(sigin, faulty\_circuit, output)$
        $correct.append(trueval == faultval)$
    **end for**
    $output.reliability \leftarrow correct.count(True)/SEQLEN$
**end for**
**return** [output.reliability for output in circuit.outputs]

# Reliability Analysis Algorithm Complexity Analysis

The most costly section is the final double loop!

We have to propagate a signal through each circuit once for each output, and SEQLEN times.

The propagation could be at worst O(number of gates in the circuit), if the signal has to propagate through each gate.

If we let $o$ be the number of outputs of the circuit, $n$ be the length of the Non-Bernoulli Sequences, and $g$ be the number of gates in the circuit, we have that the complexity of the algorithm is $O(ogn)$

# Reliability Analysis Algorithm Further Work

This algorithm still has room for improvement. If `propagate` is called once per output for the same input vector, it has to redundantly recalculate many intermediate values.

An improved `propagate` would calculate all the output values in one go, thus reducing the double loop to a single one over the length of the input sequences.

This would reduce the algorithm complexity to $O(gn)$

# Reliability Analysis Algorithm Runtime Analysis

| Input Files | c17.v | c432.v |
|---|---|---|
| Inputs | 5 | 36 |
| Outputs | 2 | 7 |
| Gates | 6 | 160 |
| Input Sequence Length | runtime /s | runtime /s |
| 1 | 0.00023 | 1.34 |
| 10 | 0.00067 | 12.2 |
| 100 | 0.00485 | 122 |
| 1 000 | 0.0473 | 1250 |
| 10 000 | 0.458 | No Data |
| 100 000 | 4.66 | No Data |

# Reliability Analysis Algorithm Runtime Analysis