Hardware Security
Final Project: Logic Locking

109062323 吳聲宏　**109062233** 蘇裕恆　**109062319** 楊智明

Folder architecture

There will be three folders with structure below

```
[random, fault, strong]
├── src
│   └── node.cpp # basic information for node
│   └── main.cpp # main file, connect all the files
│   └── encryption.cpp # the implementation for the encryption
│   └── node.h
│   └── encryption.h
├── Makefile
├── run_benchmark.sh # can test the quality of the locked circuit
```

Makefile

A few variables were created to simplify the makefile.

- **CXX:** the default is g++, and we are happy with that
- **CXXFLAGS:** flags for compiling, such as the version of C++ standard
- **LDLIBS:** the libraries for linking
- **SOURCES:** the files we want to compile
- **OBJECTS:** the names we want to call our object files
- **EXECUTABLE:** the name of the executable
- **INCLUDES:** the header files

First, the third part (`%.o`) is executed, where each `.cpp` is compiled to object files. We omitted the recipe because we are happy with the default one. After we have all the object files, the whole executable is made, so we can use that to encrypt the test benches. `clean` is for deleting every object file.

```
CXXFLAGS=-std=c++11 -O2
LDLIBS=-lm
```

```
SOURCES=./src/node.cpp ./src/encryption.cpp ./src/main.cpp
OBJECTS=$(SOURCES:.cpp=.o)
EXECUTABLE=logic_lock
INCLUDES=./src/*.h

all: $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
	$(CXX) $(OBJECTS) $(LDLIBS) -o $@

%.o: %.cpp ${INCLUDES}

clean:
	rm -rf $(OBJECTS) $(EXECUTABLE)
```

Compile

In the directory that contains `Makefile`, simply type `make`. Optionally, type `make -j 3` for a potentially faster compilation.

Profiling

`run_benchmark.sh`

We can run the profiling under the following condition:

[random, fault, strong]

```
├── src
│   └── node.cpp # basic information for node
│   └── main.cpp # main file, connect all the files
│   └── encryption.cpp # the implementation for the encryption
│   └── node.h
│   └── encryption.h
├── Makefile
├── run_benchmark.sh # can test the quality of the locked circuit
├── benchmark
│   └── c17.bench
│   └── c432.bench
    └── c3540.bench
```

With the following command:

```
source /home/HardwareSecurity112/tools/HS_Final_Project/env.cshrc
bash run_benchmark.sh
```

```
#!/bin/bash
```

```
echo "running profiling"
make clean
make -j 3
# Define the target benchmarks
targets=(
    "./benchmark/c17.bench"
    "./benchmark/c432.bench"
    "./benchmark/c3540.bench"
)

# Function to extract information from sld output
parse_sld_output() {
    local output="$1"

    # Extract keys and gates
    keys=$(echo "$output" | grep -oP 'keys=\d+' | cut -d '=' -f 2)
    gates=$(echo "$output" | grep -oP 'gates=\d+' | cut -d '=' -f 2)

    # Extract key
    key=$(echo "$output" | grep -oP 'key=\K.*')

    # Extract cube_count
    cube_count=$(echo "$output" | grep -oP 'cube_count=\d+' | cut -d '=' -f
2)

    # Extract cpu_time
    cpu_time=$(echo "$output" | grep -oP 'cpu_time=\d+\.\d+' | cut -d '='
-f 2)

    # Output the extracted information
    echo "Keys: $keys"
    echo "Gates: $gates"
    echo "Key: $key"
    echo "Cube Count: $cube_count"
    echo "CPU Time: $cpu_time"
}

# Loop through each target and execute the required commands
for target in "${targets[@]}"; do
    # Generate the test.bench filename based on the target
    test_bench="test.bench"
    # Execute the logic_lock command
    ./logic_lock "$target" "$test_bench" > /dev/null

    # Capture the output of the sld command
    sld_output=$(sld "$test_bench" "$target")

    # echo $sld_output
    # Parse and extract information from the sld output
    parse_sld_output "$sld_output"

    # Run the lcmp command with the parsed key
    lcmp_output=$(lcmp "$target" "$test_bench" key="$key")

    # Check if the output contains "equivalent"
    if echo "$lcmp_output" | grep -q "equivalent"; then
        echo -e "\e[32mOutput is equivalent\e[0m"
    else
        # Print error message in red
        echo -e "\e[31mError: Output is not equivalent\e[0m"
    fi

    # Calculate profiling result
    alpha=1 # not sure of the alpha & beta
    beta=100000
    profiling_result=$(echo "scale=10; $alpha * $cube_count / ($gates *
$keys) + $beta * $cpu_time / ($gates * $keys)" | bc -l)

    # Output the profiling result
    echo -e "\e[35mProfiling Result:$profiling_result\n \e[0m"

done
```

Visualization

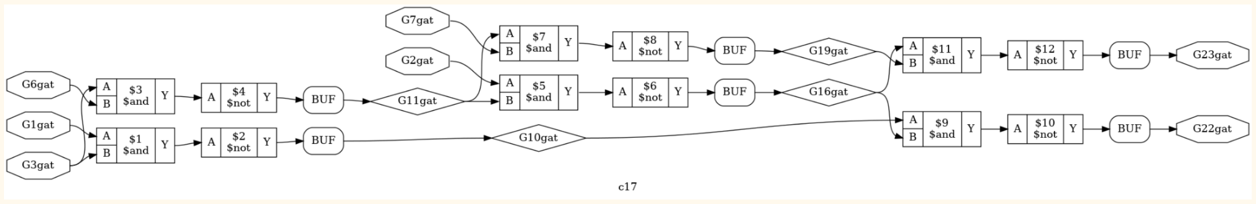
We used Yosys [3] as the visualization tool. On Ubuntu, we can install (an old version of) Yosys via `apt-get` [4]. Then, we use the `show` command [5] in Yosys to generate a Graphviz `.dot` [2] file. Since Yosys does not accept `.bench` files as input, we also wrote a very simple script to convert `.bench` files into Verilog files (not shown here.)

```
$ sudo apt-get install yosys
$ yosys
yosys> read_verilog c17.v
yosys> show -viewer none -prefix c17
yosys> exit
$ dot -Txdot c17.dot | dot -Tpng > c17.png
```

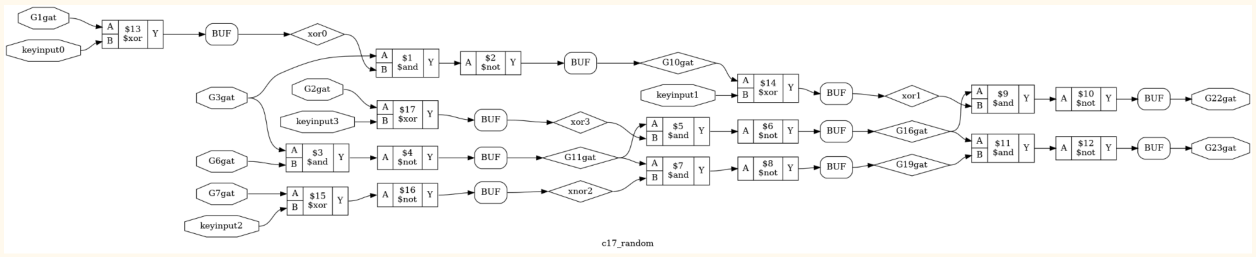
Note that the option `-viewer none` is only strictly correct since some newer versions of Yosys (e.g. Yosys 0.42 [3].) In older versions of Yosys (e.g. Yosys 0.9 [4],) this results in the shell complaining that the command `none` does not exist, but that is not a big deal.

The following figures show the original `c17.bench` and the same circuit locked with three different logic locking methods. We are able to visualize `c432.bench` and `c3540.bench` as well, but they are too huge to be shown in the report.

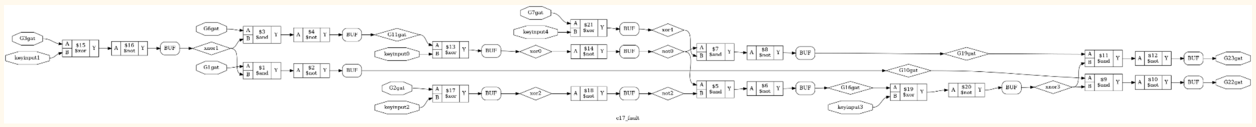
A strange thing about Yosys is that it breaks the NAND gates into an AND gate and an inverter. It is possible to edit the generated `.dot` files before converting them into graphical representations, though.



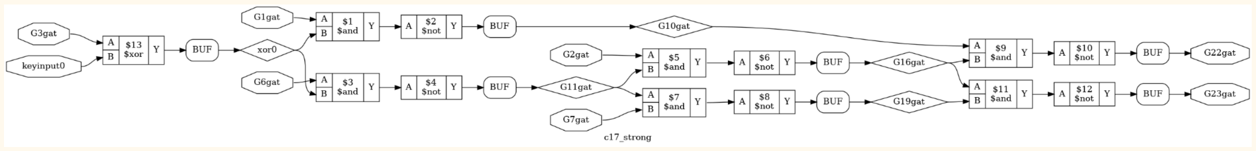
The original `c17.bench` visualized using Yosys 0.9.



The locked `c17.bench` using random locking.



The locked `c17.bench` using fault analysis-based locking.



The locked `c17.bench` using strong logic locking.

Implementation

Random Logic Locking

As the name implies, random logic locking with a *k*-bit key simply selects *k* random wires, and either an XOR gate or an XNOR gate is inserted for each wire.

According to EPIC, wires on the critical paths should be avoided. Since all we have are gate-level netlists, for simplicity, we define the delay of every primitive gate to be 1 time unit, and there is no additional delay for fan-in and fan-out. Afterwards, we compute the arrival time (AT) and the required arrival time (RAT) of each wire. If for a wire, RAT - AT = 0, then the wire is on a critical path; otherwise, a key gate can be inserted on the wire.

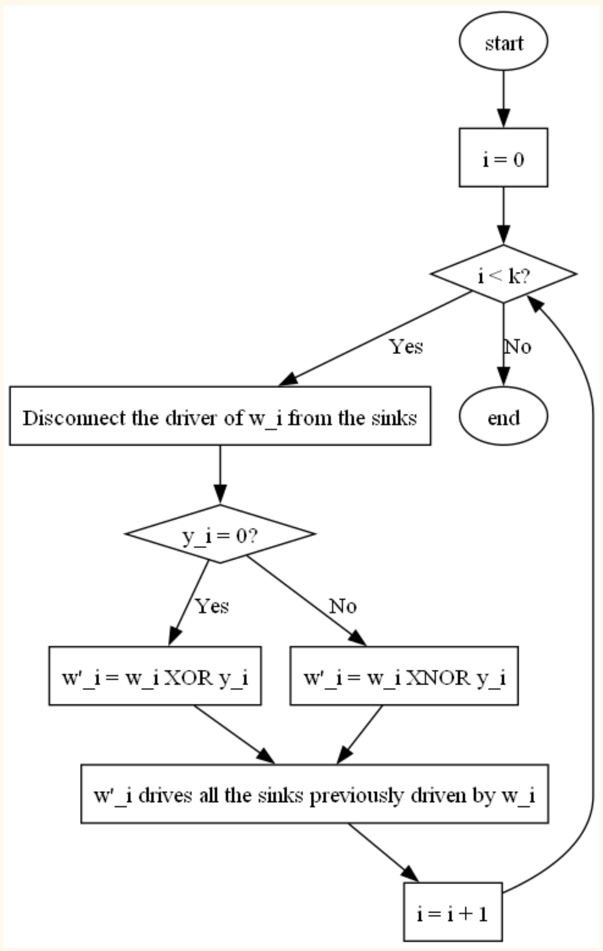
Pseudocode

```
Algorithm 1 Critical path analysis
Input: Netlist, the set of wires W, CellLibrary (especially gateDelay)
Output: For each wire in W, the arrival time (AT) and the required arrival
time (RAT)
globalMaxAT ← 0
for each wire w in W, in any topological order
    if w is primary input
        then AT(w) ← 0
    otherwise
        localMaxAT ← 0
        for each fan-in wire v
            localMaxAT ← max(localMaxAT, AT(v))
        AT(w) ← localMaxAT + gateDelay(d_w)  // d_w is the driver of w
        if w is primary output
            then globalMaxAT ← max(globalMaxAT, AT(w))
for each wire w in W, in any reverse topological order
    RAT(w) ← globalMaxAT
    if w is not primary output
        for each fan-out wire v
```

```
            RAT(w) ← min(RAT(w), RAT(v) - gateDelay(d_v))
```

```
Algorithm 2 Wire selection
Input: Netlist (especially the set of wires W,) AT, RAT, k (number of key
bits)
Output: k wires W' = {w_i} to be locked
for each wire w in W, in a random order
    if RAT(w) - AT(w) > 0
        then add w into W'
        if |W'| = k then return W'
```

Flow Chart



start
i = 0
i < k?
Yes
No
end
Disconnect the driver of w_i from the sinks
y_i = 0?
Yes
No
w'_i = w_i XOR y_i
w'_i = w_i XNOR y_i
w'_i drives all the sinks previously driven by w_i
i = i + 1

How to insert key gates to selected wires

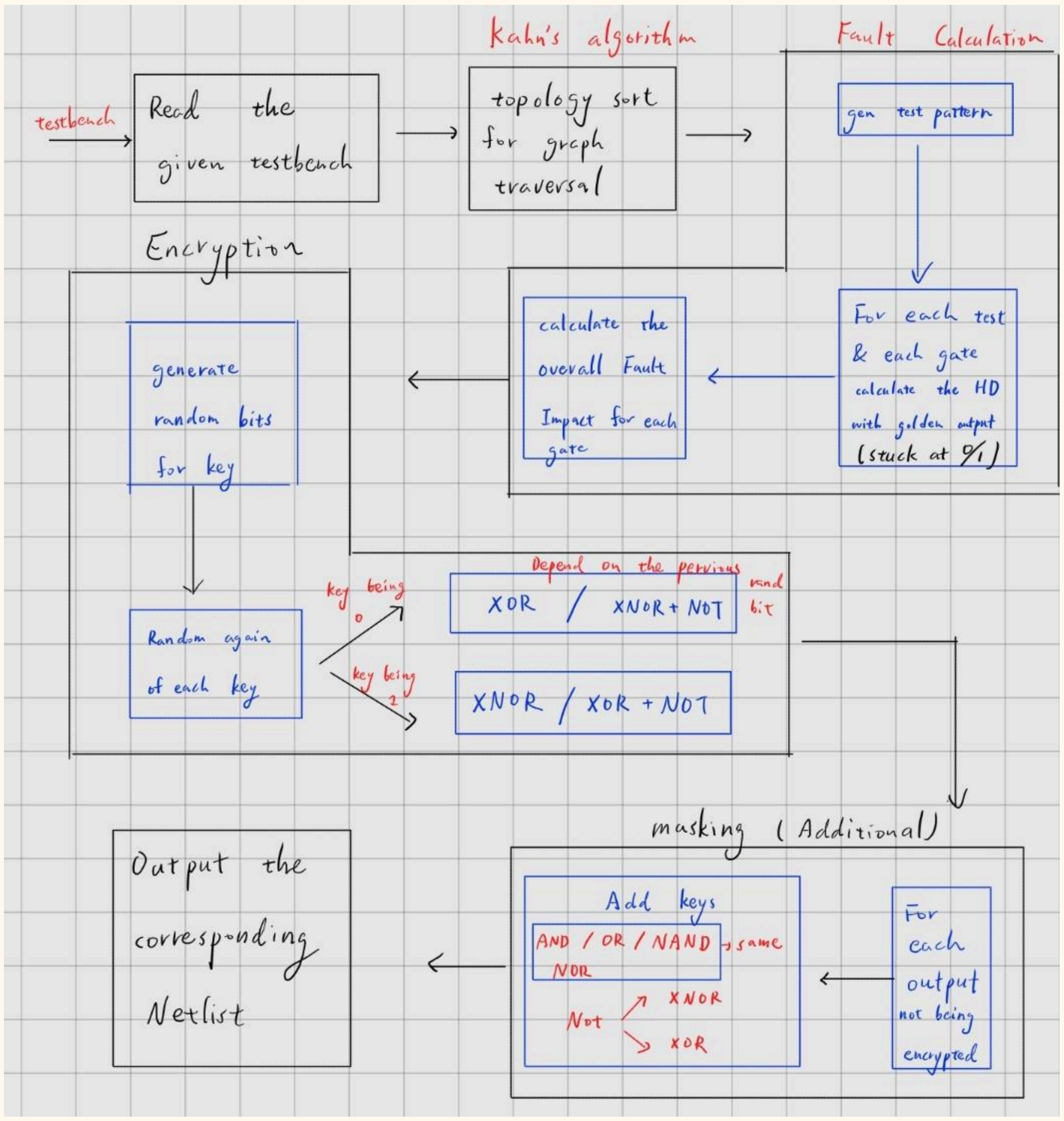
Fault Analysis-Based Logic Locking

In Fault Analysis-Based Logic Locking, the primary goal is to enhance the security of integrated circuits by identifying and locking the nodes that have the highest fault impact. This means focusing on the nodes that, when subjected to faults, are most likely to propagate errors to other parts of the circuit, thereby affecting the overall functionality.

Pseudocode

```
Input: Netlist, KeySize, EncryptionKey
Output: Encrypted netlist
Set the total number (n) of samples to calculate the FaultImpact;
Compute FaultImpact;
Function FaultImpact:
    For i <- n , we have a golden output:
        For each gate in the netlist:
            Fixed that gate to 0:
                Calculate the HammingDistance and add the corresponding
                NoP0 , NoO0 to the node’s information
            Fixed that gate to 1:
                Calculate the HammingDistance and add the corresponding
                NoP1 , NoO1 to the node’s information
    node->fault_impace =  NoP0 * NoO0 + NoP1 * NoP1

Select the gate with top k highest FaultImpact;
// Modification Phase
Generate R; // random number for key with size k
foreach bit = 0 <- k do
            if(R[bit] == 0){
                  If the key-bit is '0', then the key-gate structure can be
either 'XOR- gate' or ' XNOR- gate + inverter '.
            # select one of them randomly
            else(R[bit] == 1){
                  If the key-bit is '1', then the key-gate structure can be
either 'XNOR-gate' or ' XOR- gate + inverter '.
            # select one of them randomly
            }
}
```

Flow Chart


testbench
Read the given testbench
Kahn's algorithm
topology sort for graph traversal
Fault Calculation
gen test pattern
For each test & each gate calculate the HD with golden output (stuck at 0/1)
calculate the overall Fault Impact for each gate
Encryption
generate random bits for key
Random again of each key
key being 0
key being 1
Depend on the pervious rand bit
XOR / XNOR + NOT
XNOR / XOR + NOT
masking (Additional)
For each output not being encrypted
Add keys
AND / OR / NAND / NOR → same
Not
XNOR
XOR
Output the corresponding Netlist

Strong Logic Locking

In this part, since the paper algorithm contains a NP-complete problem, we decided to make some improvements. Although there are three versions, we only turned in the final version, since it had the best performance.

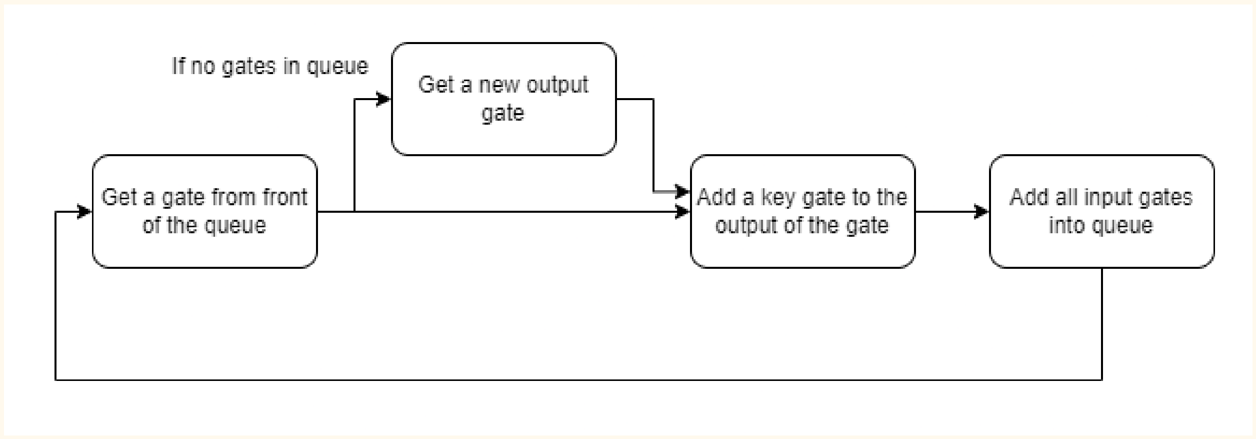
Naive Approach

In order for a key gate to be pairwise secure, the gate needs to be unable to be muted by other gates. We initially concluded that the closer a key gate is to the endpoint, the less likely it is to be muted. After that, to increase the probability of being pairwise secure, we added key gates to inputs of the original gate, since there was only one gate between input and output of a gate. So the main process is to find an output, and recursively add a key gate to each input.

Pseudocode

```
encoded_gate = 0
i = 0
check_queue //for storing to be encoded gates
while encoded_gate < keys{
      //if there are no more gates in queue, then get another output
      if(check_queue == 0) {
            check_queue.add(output[i++])
      }
      add a key-gate to the output of check_queue.front()
      for every input a in check_queue.front() {
            check_queue.add(a)
      }
      check_queue.pop()
      encoded_gate++
}
```

Flow Chart


If no gates in queue
Get a new output gate
Get a gate from front of the queue
Add a key gate to the output of the gate
Add all input gates into queue

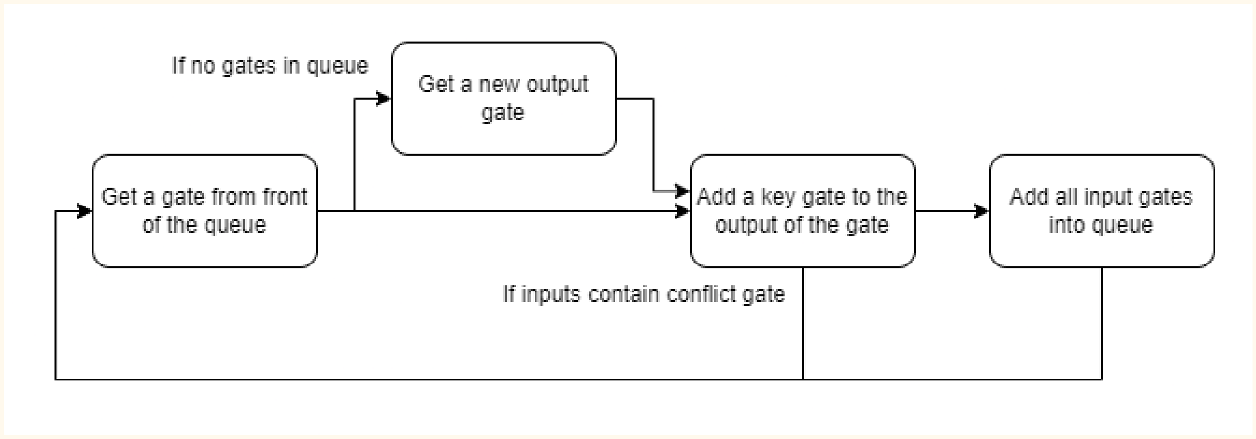
Easy Comparison

Based on the naive algorithm, we added a simple condition to check the possibility of being muted. Since we assumed the output and input of a gate are mostly what we were working with, we only checked each gate whether the inputs contained gates that would likely to mute out the gate. For example, if an and gate had another and gate as input, the original and gate would be more likely to output 0, since the input and gate would also be more likely to output 0. Other examples such as or gate, nor gate to or, nand. And gate, nand gate to and, nor. The initialization referred to the paper TA gave us. It tried to find the gate with most outputs, and used it as the start of the encryption.

Pseudocode

```
encoded_gate = 0
i = 0
check_queue //for storing to be encoded gates
while encoded_gate < keys{
      //if there are no more gates in queue, then get another output
      if(check_queue == 0) {
            check_queue.add(output[i++])
      }
      add a key-gate to the output of check_queue.front()
      //conflicts such as and->and,nor   or->or,nand ...
      if current gate has no conflict input gates {
            for every input a in check_queue.front() {
                  check_queue.add(a)
            }
      }
      check_queue.pop()
      encoded_gate++
}
```

Flow Chart


If no gates in queue
Get a new output gate
Get a gate from front of the queue
Add a key gate to the output of the gate
Add all input gates into queue
If inputs contain conflict gate

Pseudo Brute Force

In this method, we really implemented the method from the paper. In initialization, instead of finding outputs as a starting point, the paper tried to find the start by searching the gate that had not been encrypted and with max convergence. For each loop, we compared the output of two circuits, one regular, another we added a not gate at the end of a gate. If the outputs were the same, it meant if a key gate were installed there, it would be mutable. However, we simplified a few steps.

First, since we did not know how to check whether the two gates had a dominant relation, we skipped that process. We only used the brute force method to check whether two gates were pairwise secure or not.

Second, we only changed the inputs that did not connect to the two gates we were currently checking. We first found the inputs that would lead to the both gates, then we picked the inputs which were not found to do testing.

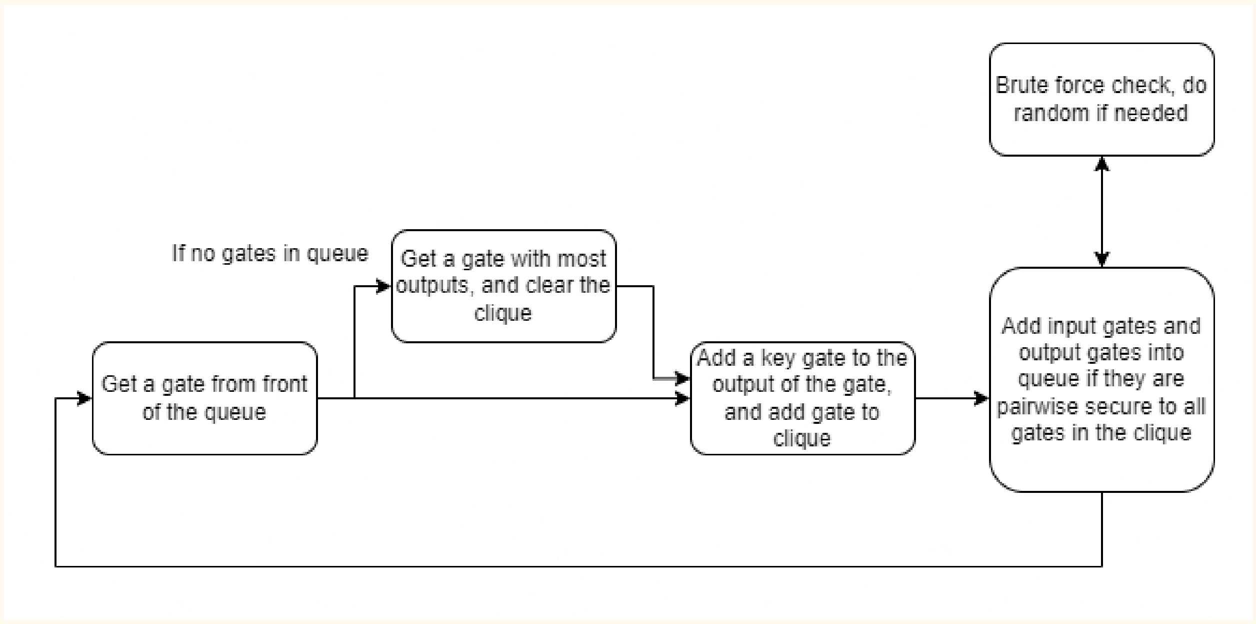
Last, if the inputs were too many, which meant brute force would take too long, we used random instead, and we set a limit to it. If the chosen inputs were small enough, brute force would be applied. Otherwise, a certain amount of loops, with random inputs with random values would be used to test the circuit.

Pseudocode

```
encoded_gate = 0
clique = {}
check_queue //for storing to be encoded gates
```

```
while encoded_gate < keys{
      if(check_queue == 0) {
            //if there are no more gates in queue, then get another gate
            check_queue.add(init())
            clique.clear() // the clique in algorithm 3 in the paper
      }
      add a key-gate to the output of check_queue.front()
      clique.add(check_queue.front())
      encoded_gate++
      for every input a in check_queue.front() {
            if a is checked: continue
            if check_pairwise(clique, a) == 1: check_queue.add(a)
      }
      for every input a in check_queue.front() {
            if a is checked: continue
            if check_pairwise(clique, a) == 1: check_queue.add(a)
      }
      check_queue.pop()
}

gate init() {
      cur_max = 0
      gate = NULL
      for every gate a {
            if a.output size > cur_max && not added key gate {
                  cur_max = output_size
                  gate = a
            }
      }
      return gate
}

bool check_pairwise(vector clique, gate a) {
      A_value = 10 // this is the threshold of brute force or random
      for every b in clique {
            //inputs that only reach a but not b are fine
            unused_inputs = {the inputs that dont go to a and b}
            if unused_inputs.size > a_value { //we set it as 10 in the code
                  for 0 ~ 2^a_value {
                        pick a_value of inputs randomly from unused_inputs
                        assign random values to the picked inputs
                        ori_output = outputs of the circuit
                        flip gate a //and to nand, or to nor, xor to xnor
                        a_output = outputs of the circuit
                        undo gate a

                        flip gate b
                        b_output = outputs of the circuit
                        undo gate b
                        if (ori_output == a_output != b_output
                          || ori_output == b_output != a_output)
                              return 0
                  }
            }
            else {
                  for every combination of inputs for unused_inputs {
                        ori_output = outputs of the circuit

                        //flipping and to nand, or to nor, xor to xnor
                        flip gate a
                        a_output = outputs of the circuit
                        undo gate a

                        flip gate b
                        b_output = outputs of the circuit
                        undo gate b
                        if (ori_output == a_output != b_output or
ori_output == b_output != a_output) return 0
                  }
            }
      }
      return 1
}
```

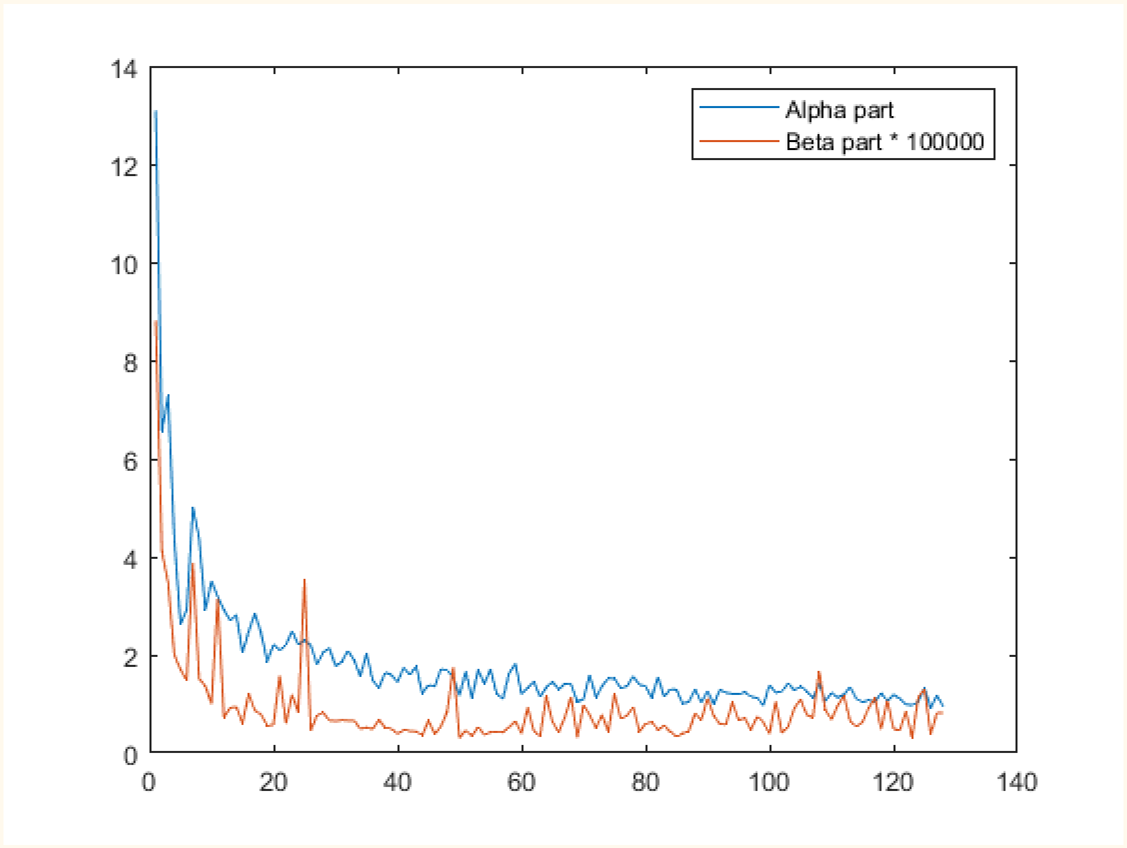
Flow Chart


Brute force check, do random if needed
If no gates in queue
Get a gate with most outputs, and clear the clique
Get a gate from front of the queue
Add a key gate to the output of the gate, and add gate to clique
Add input gates and output gates into queue if they are pairwise secure to all gates in the clique

Security Analysis Through SAT Attack and Equivalence

Random

We ran our random logic locking using the provided c3540.bench, with the length of the key ranging from 1 to 128. The figure below shows the quality of the locked circuit as a line chart, where only one locked circuit is analyzed for each key length. The alpha part is $\frac{cube\_count}{gates \times keys}$, and the beta part is $\frac{cpu\_time}{gates \times keys}$ (scaled 100000 times in the chart.)



From the chart shown above, we can infer that if we use the provided formula to calculate the quality of our randomly locked circuit, then the quality is rather stabilized when the length of the key is between 64 and 128 (except for random variations.) However, the quality is *much* greater if the length of the key is very low. We separated the alpha part and the beta part in this chart, and both metrics show the same trend. As a result, in the following security analyses, we fix α to 1 and β to 100000.

Fault Analysis:

The result below is run under c3540.bench

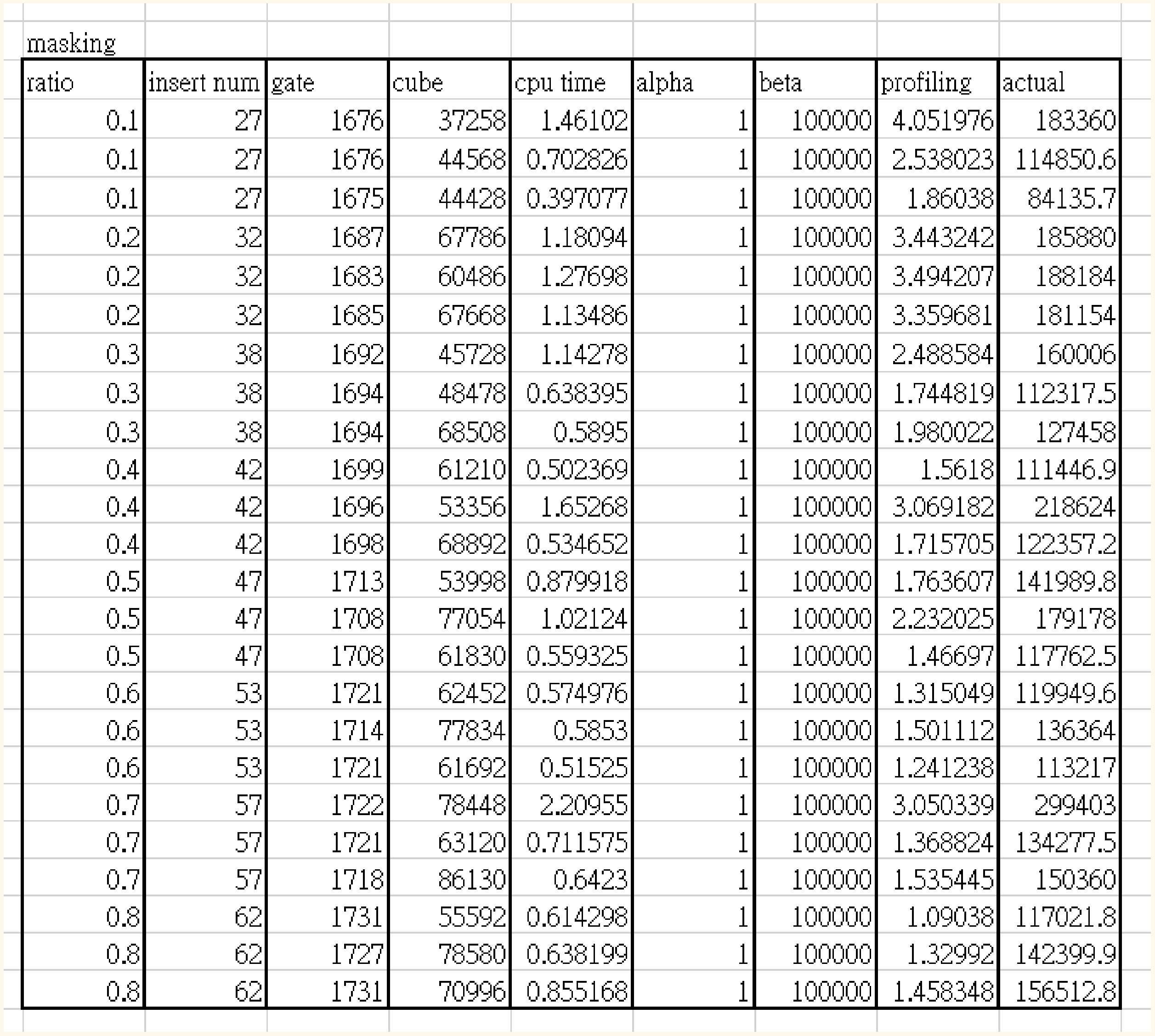
With masking after key insertion:

So let's analyze the following performance under that condition of total number insert + masking in the end.

Firstly, the table below presents the profiling results obtained from three experiments using the `run_benchmark.sh` script. Please note that these results are under the condition that masking is utilized.

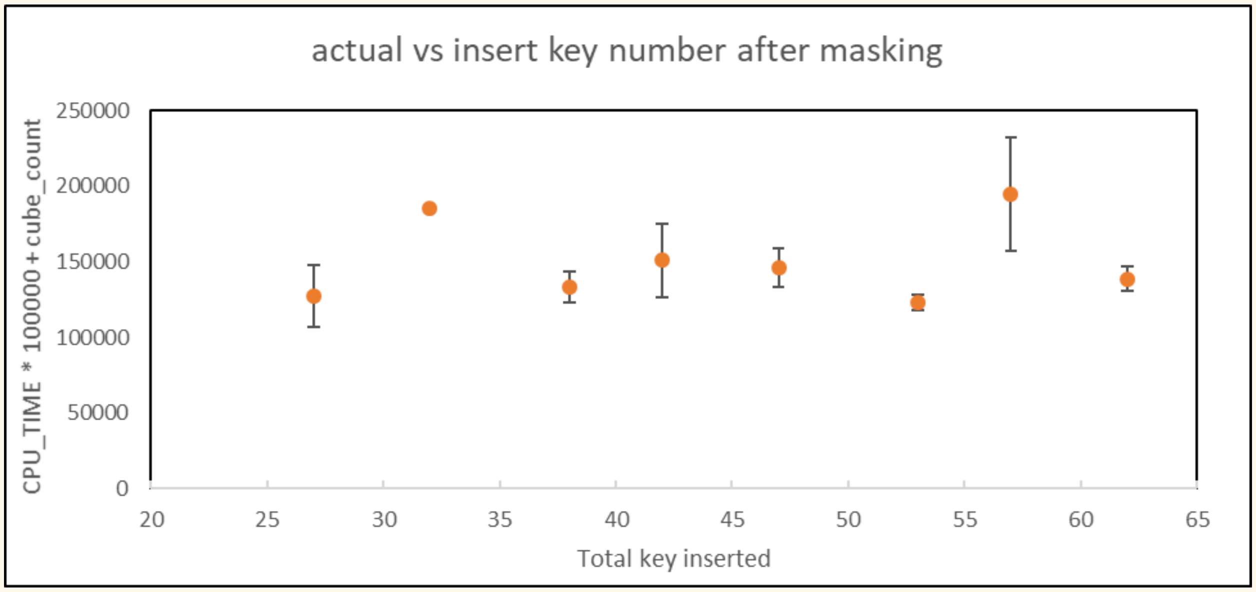
Here, the `ratio` keywords indicates the ratio between key inserted and the input size.

The `actual` keywords indicate the formula: cube * alpha + cpu_time * beta. In this experiment, the values of alpha and beta are set to 1 and 100,000, respectively. This ratio is derived from the numbers for cube_count and CPU_Time.



masking

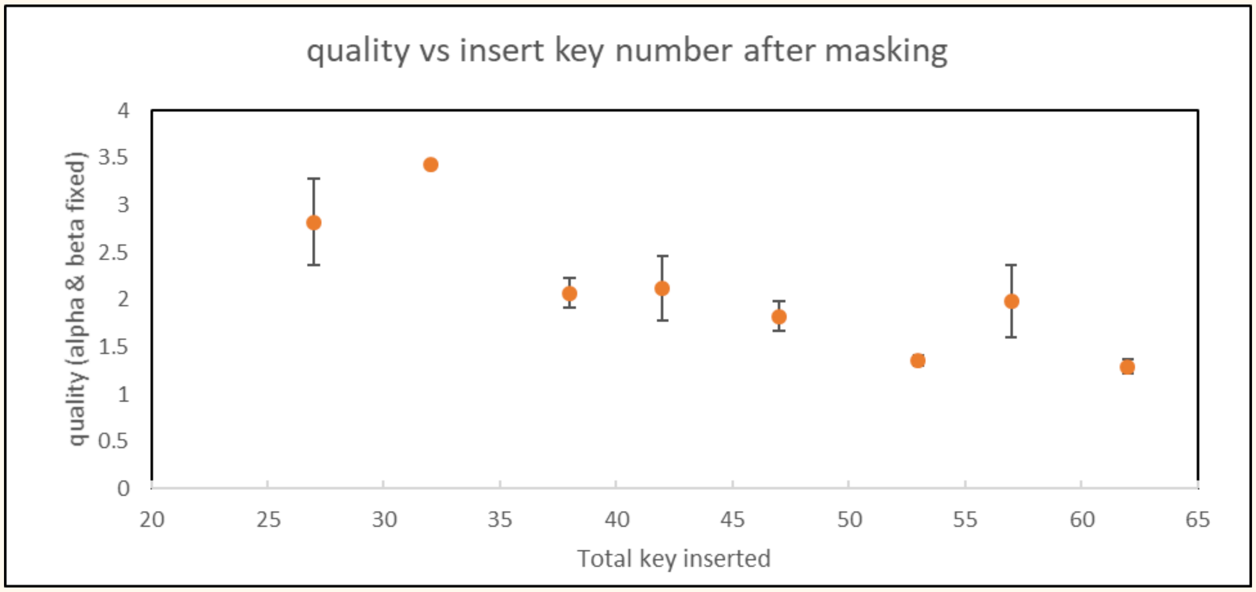
ratio	insert num	gate	cube	cpu time	alpha	beta	profiling	actual
0.1	27	1676	37258	1.46102	1	100000	4.051976	183360
0.1	27	1676	44568	0.702826	1	100000	2.538023	114850.6
0.1	27	1675	44428	0.397077	1	100000	1.86038	84135.7
0.2	32	1687	67786	1.18094	1	100000	3.443242	185880
0.2	32	1683	60486	1.27698	1	100000	3.494207	188184
0.2	32	1685	67668	1.13486	1	100000	3.359681	181154
0.3	38	1692	45728	1.14278	1	100000	2.488584	160006
0.3	38	1694	48478	0.638395	1	100000	1.744819	112317.5
0.3	38	1694	68508	0.5895	1	100000	1.980022	127458
0.4	42	1699	61210	0.502369	1	100000	1.5618	111446.9
0.4	42	1696	53356	1.65268	1	100000	3.069182	218624
0.4	42	1698	68892	0.534652	1	100000	1.715705	122357.2
0.5	47	1713	53998	0.879918	1	100000	1.763607	141989.8
0.5	47	1708	77054	1.02124	1	100000	2.232025	179178
0.5	47	1708	61830	0.559325	1	100000	1.46697	117762.5
0.6	53	1721	62452	0.574976	1	100000	1.315049	119949.6
0.6	53	1714	77834	0.5853	1	100000	1.501112	136364
0.6	53	1721	61692	0.51525	1	100000	1.241238	113217
0.7	57	1722	78448	2.20955	1	100000	3.050339	299403
0.7	57	1721	63120	0.711575	1	100000	1.368824	134277.5
0.7	57	1718	86130	0.6423	1	100000	1.535445	150360
0.8	62	1731	55592	0.614298	1	100000	1.09038	117021.8
0.8	62	1727	78580	0.638199	1	100000	1.32992	142399.9
0.8	62	1731	70996	0.855168	1	100000	1.458348	156512.8



As we can see, from the chart: α * `cube_count` + β * `cpu_time`

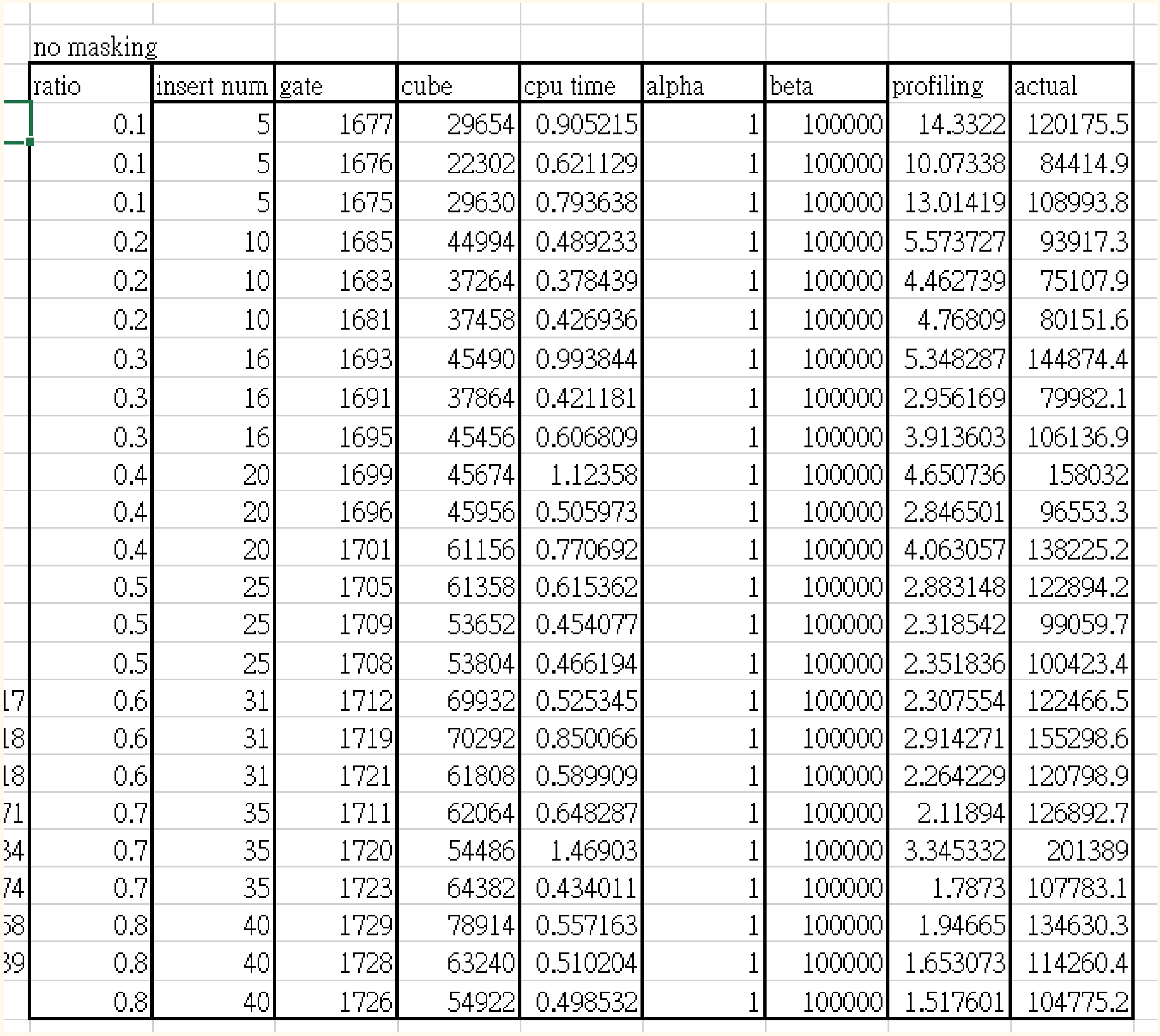
$$Quality = \alpha \times \frac{cube\_count}{gates \times keys} + \beta \times \frac{cpu\_time}{gates \times keys}$$

The uncertainty (big standard error) for the profiling makes us unable to find the best number of inserted keys.



Since gates * keys will increase, but cube_count and cpu_time will not keep up at the same pace, we know that the quality will drop.

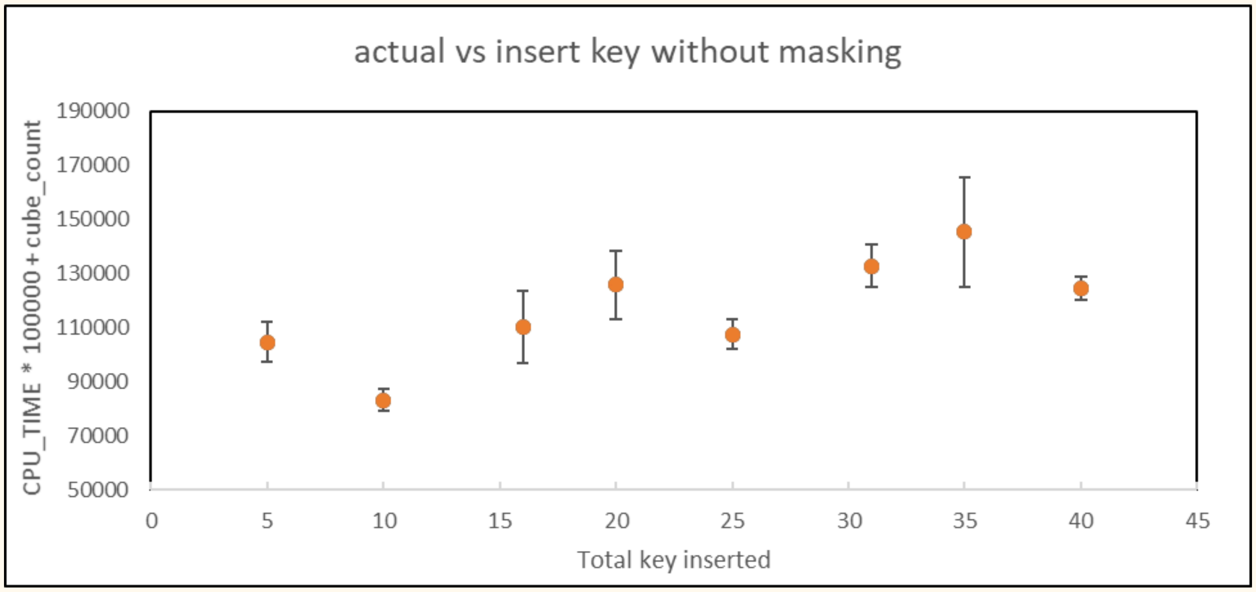
Without masking after key insertion:

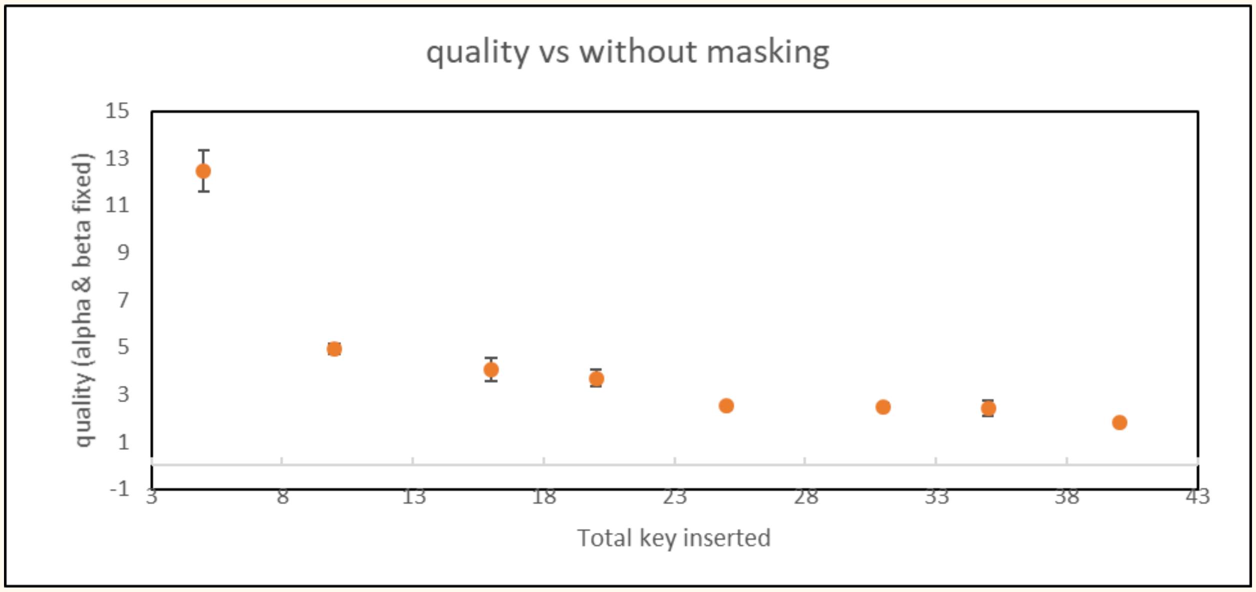


no masking

	ratio	insert num	gate	cube	cpu time	alpha	beta	profiling	actual
	0.1	5	1677	29654	0.905215	1	100000	14.3322	120175.5
	0.1	5	1676	22302	0.621129	1	100000	10.07338	84414.9
	0.1	5	1675	29630	0.793638	1	100000	13.01419	108993.8
	0.2	10	1685	44994	0.489233	1	100000	5.573727	93917.3
	0.2	10	1683	37264	0.378439	1	100000	4.462739	75107.9
	0.2	10	1681	37458	0.426936	1	100000	4.76809	80151.6
	0.3	16	1693	45490	0.993844	1	100000	5.348287	144874.4
	0.3	16	1691	37864	0.421181	1	100000	2.956169	79982.1
	0.3	16	1695	45456	0.606809	1	100000	3.913603	106136.9
	0.4	20	1699	45674	1.12358	1	100000	4.650736	158032
	0.4	20	1696	45956	0.505973	1	100000	2.846501	96553.3
	0.4	20	1701	61156	0.770692	1	100000	4.063057	138225.2
	0.5	25	1705	61358	0.615362	1	100000	2.883148	122894.2
	0.5	25	1709	53652	0.454077	1	100000	2.318542	99059.7
	0.5	25	1708	53804	0.466194	1	100000	2.351836	100423.4
17	0.6	31	1712	69932	0.525345	1	100000	2.307554	122466.5
18	0.6	31	1719	70292	0.850066	1	100000	2.914271	155298.6
18	0.6	31	1721	61808	0.589909	1	100000	2.264229	120798.9
71	0.7	35	1711	62064	0.648287	1	100000	2.11894	126892.7
34	0.7	35	1720	54486	1.46903	1	100000	3.345332	201389
74	0.7	35	1723	64382	0.434011	1	100000	1.7873	107783.1
58	0.8	40	1729	78914	0.557163	1	100000	1.94665	134630.3
39	0.8	40	1728	63240	0.510204	1	100000	1.653073	114260.4
	0.8	40	1726	54922	0.498532	1	100000	1.517601	104775.2

We will provide the same chart with the above condition.

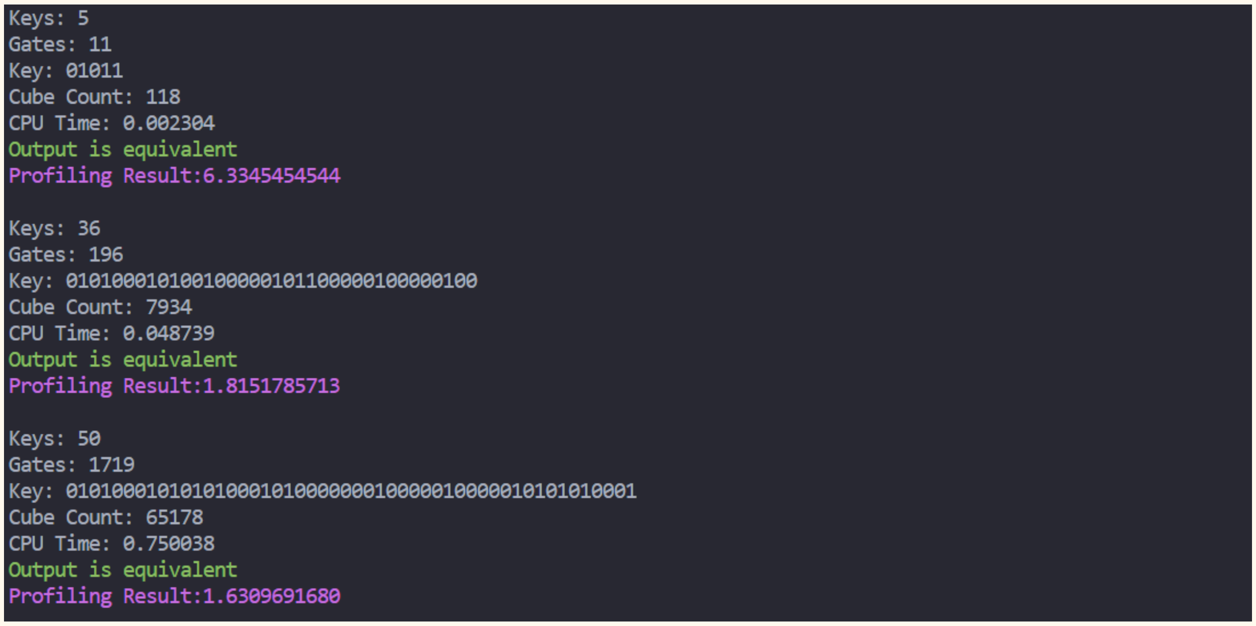




It is evident that we are still unable to determine the optimal key number, as demonstrated by masking. The quality trend remains nearly identical to that observed with masking.

Strong Logic Locking

Using the `run_benchmark.sh` on the pseudo brute force algorithm, and input size as key size, we got the following test result. First one is c17, second is c432, and the last one is c3540. Alpha is 1, beta is 100000.



```
Keys: 5
Gates: 11
Key: 01011
Cube Count: 118
CPU Time: 0.002304
Output is equivalent
Profiling Result:6.3345454544

Keys: 36
Gates: 196
Key: 010100010100100000101100000100000100
Cube Count: 7934
CPU Time: 0.048739
Output is equivalent
Profiling Result:1.8151785713

Keys: 50
Gates: 1719
Key: 01010001010101000101000000010000010000010101010001
Cube Count: 65178
CPU Time: 0.750038
Output is equivalent
Profiling Result:1.6309691680
```

Hardness & Suggestions

The initial phase of the project proved to be the most challenging, as we needed to develop a template for various locking methods. Fortunately, we discovered an excellent template on GitHub [1] that seamlessly converts benchmarks into code. This template enabled us to implement encryption directly through the code.

The most difficult part of logic locking based on EPIC is that the locking mechanism proposed in EPIC is not purely random; it also mentioned that wires should be selected to avoid critical paths. To conduct critical path analysis, we had to define the delays of a logic circuit when all we have are gate-level netlists. Other than that, random logic locking is relatively trivial.

The most challenging aspect of fault-based logic locking is the unclear description in the paper. I had to search through GitHub to gather additional information, which finally helped me understand parts of the paper's methodology.

The hardest part in strong logic locking is the fact that the algorithm contains a NP-complete problem. We thought very hard on how to minimize the calculation. In the end we came up with brute forcing until a certain threshold.

Reference

[1] Code Template (https://github.com/Ohpaipai/Homework/tree/main)

[2] Graphviz (https://graphviz.org/)

[3] Yosys (https://github.com/YosysHQ/yosys)

[4] Yosys on Ubuntu (https://packages.ubuntu.com/jammy/yosys)

[5] show - Yosys (https://yosyshq.readthedocs.io/projects/yosys/en/latest/cmd/show.html)

[6] GitHub (https://github.com/j30393/logic-locking/tree/main)