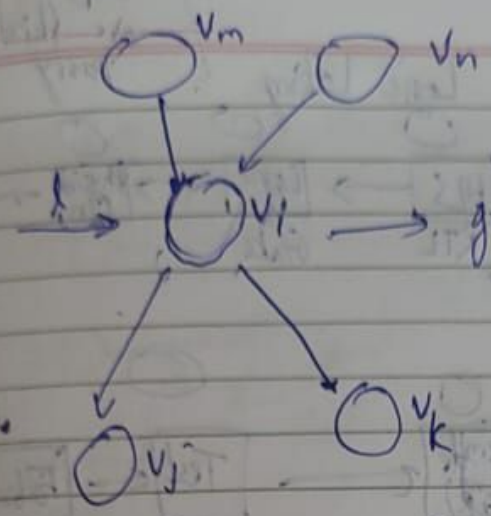


given v_i is scheduled before all others

$$TF(v_i, l) = SF(v_i, l) + PF(v_m, l) + PF(v_n, l) + SF(v_j, l) + SF(v_k, l)$$

⇓

we don't consider already scheduled nodes

FD-MLRC is $O(n^2)$ as calculation of TF takes $O(n)$ time

FD-MR-LC (a, λ) obj mm a

{

while (all ops are scheduled) → $O(n^3)$

{

compute the time frame (mobility)

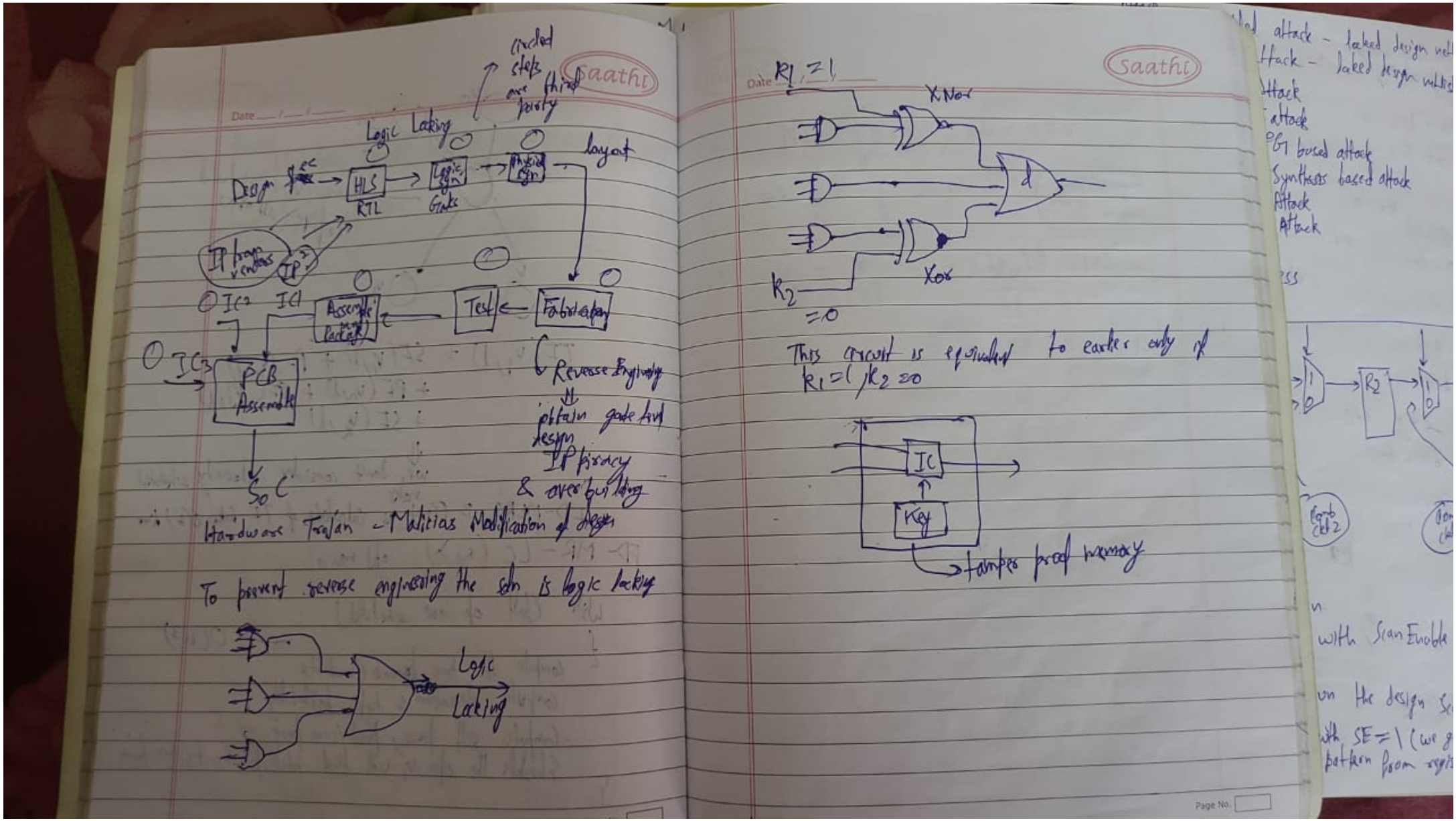
compute operation & type probability

Compute self force, pred/succ force

Schedule the op v_i with least total force in its time frame

}

Page No.


Saathi
Date
Logic Locking
circled steps are third party
Design spec
HLS
RTL
Logic Syn
Gate
Physical Syn
layout
IP from vendors
IP
IC2
IC1
Assembly testing
Test
Fabrication
IC3
PCB Assembly
SoC
Reverse Engineering
obtain gate level design
IP piracy & over building
Hardware Trojan - Malicious Modification of design
To prevent reverse engineering the soln is logic locking
Logic Locking
Saathi
Date
K1 = 1
XNor
Xor
K2
=0
This circuit is equivalent to earlier only if K1=1 & K2=0
IC
Key
tamper proof memory
attack - locked design
attack - locked design
attack
attack
PG based attack
Synthesis based attack
Attack
Attack
ss
R2
with Scan Enable
Page No.

ML-RC:

Objective is to increase concurrency to decrease latency

List MLRC (G, m)
{
  l = 1
  while (any operation remains to be scheduled) {
    for each resource type k {
      Uk = candidate available in l,
      Tlk = running operation in l -
      Calculate the total force of v_i ∈ Uk
      Select S_k ⊂ U_k s.t. |Tlk| + |S_k| ≤ q_k
      and TF maximized

Logic Locking

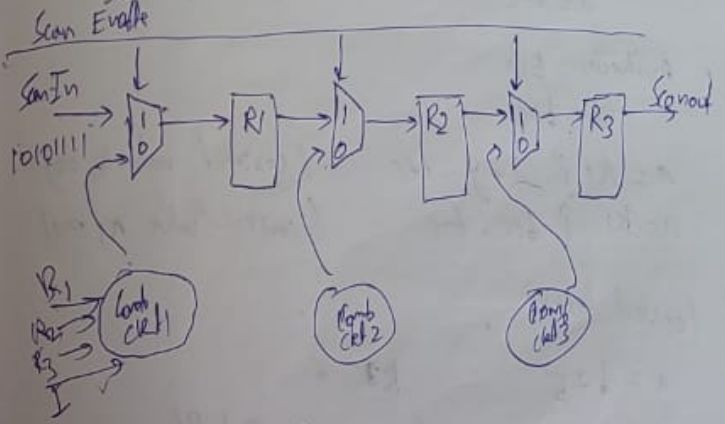
Different abstraction level
- C level
- RTL
- Gate Level

Threat Model

→ Extract a gate level netlist from the layout by reverse engineering (locked design)
→ Attacker has info about the design (white box)
→ Buy functional IP from market → can be used to verify the key (oracle)

Attack
→ oracle guided attack - locked design netlist + oracle
→ oracle less attack - locked design netlist
→ Functional Attack
→ SAT attack
→ ATPG based attack
→ Re-Synthesis based attack
→ Structural Attack
→ ML based Attack

Scan Chain Access



n-reg in scan chain
- Run n cycle with ScanEnable = 1 (This stores the input pattern in registers)
- one clk, run the design ScanEnable = 0 (we know the expected output for each registers)
- Run n clk with SE = 1 (we get the output pattern from registers

C-level hacking

- ~~Arithmetic~~ conditional Exp

```
if (c)
    a = b+c ;
else a = b-c ;
```

⇓

```
if (c ⊕ k2)        k2 = 0
    a = b+c
else
    a = b-c
```

or

```
if ((c ⊕ k2)       else k2 = 1
    a = b-c
else a = b+c
```

- Arithmetic Exp

```
a = b+c
a = k1 ? 2*y : b+c      (correct value k1 = 0)
a = k1 ? b+c : b-c      (correct value k1 = 1)
```

- Constant Hacking

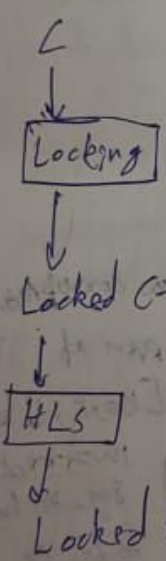
```
a = b+5        k2 = 6

5      =  0101
⊕ 6    =  0110
-------------------
       =  0011
```

a = b + 3 ⊕ k

Logic locking

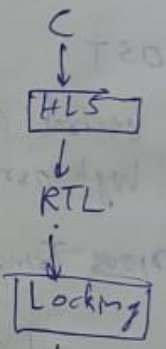
- Behavioral level → (HLOCK) → C-level / during HLS → 2 levels at which locking can be done at behavioral level
- RTL → (TAO) → ASSURE, HOST, Dishonest Oracle
- Gate level
- 3 types of locking in general { - constant - operation - condition } → highly classified
  - if key not known operation lost
  - semantic info

H-Lock



TAO

ASSURE/HOST

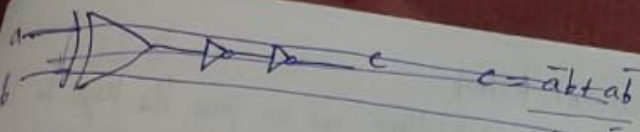


- Advantage of Locking at higher level => we have semantic info that can be used

Comparison:

HLock
- Easy to implement
- HLS independent (not dependent on software)

TAO
- hard to implement
- less overhead



$\bar{a}b + a\bar{b}$

~~$(a+b)(a+b)$~~

The drawback of RLL is that RLL circuit ...
correct output even for incorrect ...
the right ...

- TAO can avoid overhead since it avoid recomputing the operations in case of a false key as it can simply borrow output from existing circuit on the other hand in H-Lock extra computation needed to be present always (K1 ? b+c : b-c) causing overhead

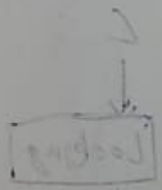
ASSURE / HOST

- RTL → HDL Parser → AST → Lock → AST → RTL Generator → Locked RTL

ASSURE { - constant - operation - branch

HOST:

- control corruptability → low corruptability
- high corruptability → corrupt control



Spurious Transition in controller.

→ low corruptability

S4 : if ([k0,k1] = 2'b00 & reg_01 == 10) next_state = S8 spurious transition

else if ([k0,k1] = 2'b01 & reg_02 == 05) next_state = S9

else next_state = S5 <original transition

even if [k0,k1] is incorrect reg_x has to be ... for spurious transition

Gate level logic locking

Combinational Part

- Pre SAT Exa - RLL, FLL, SLL
- SAT Attack
- Post SAT - SarLock, Anti-SAT, Cycle-SAT, SFLL, Provably Secure Dishonest Oracle Delay Lock, Clock locking

⇓ Locks combinational part

Sequential Part

- Locking the FSM Part Harpoon
- Lock the datapath regs Scanlock

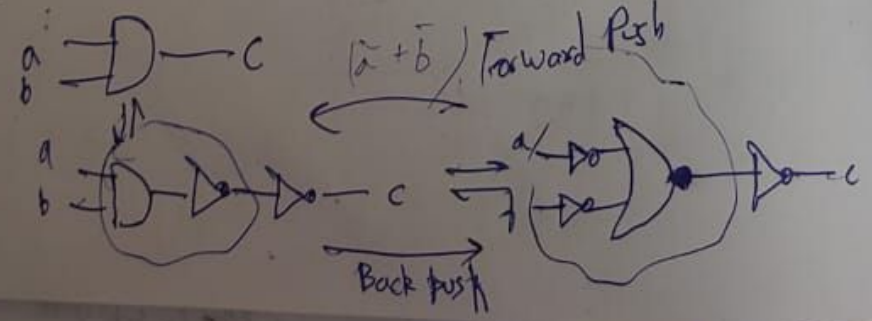
Random Logic Locking.

XOR/XNOR based key insertion at random location

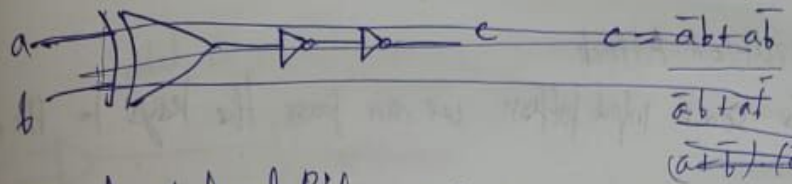
⇓

Key easily identified for XOR = 0 and for XNOR = 1

- Soln - Bubble Pushing



Combinational Logic Locking



The drawback of RLL is that RLL circuit may produce correct output even for incorrect key values for many of the inputs - Low Output Corruptibility.

Fault Analysis based Logic Locking (FLL)

Based on analysis we put XOR, XNOR instead of randomly to ensure that output corruptibility is high.

Physical Fault { Stuck at zero / stuck at one

FI_G (Fault impact of G) $= (P_0 \times O_0) + (P_1 * O_1)$

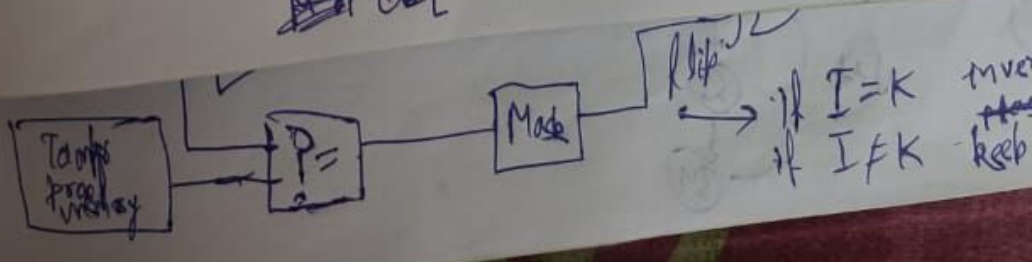
VLSI Testing for physical flaws in the circuit

~~VLSI Testing~~ / ~~Logic Locking~~

Sensitization
Fault prop
Multiple fault

$$OC = \frac{1}{PQM} \sum_{I=1}^{P} \sum_{j=1}^{Q} HD(O_E(I_i), O_L(I_i, K_j)) \times 100\%$$

P random input patterns
Q random key values
L Locked Netlist
M bit output



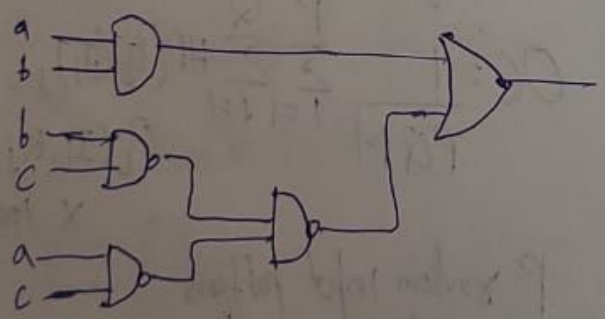
avoid overhead since if used recomputing the operations
of a false key as it can simply borrow other

Sensitization Attack
For some input pattern we can pass the keys to the output

SLL
Inserts keys to maximize interdependence by increasing clique size

SAT Attack
Satisfiability Problem – Given a formula f, is there an input for which it becomes TRUE
Boolean Formula : $(a \wedge b) \wedge (a' \wedge b')$ unsatisfiable
↓ SAT solver

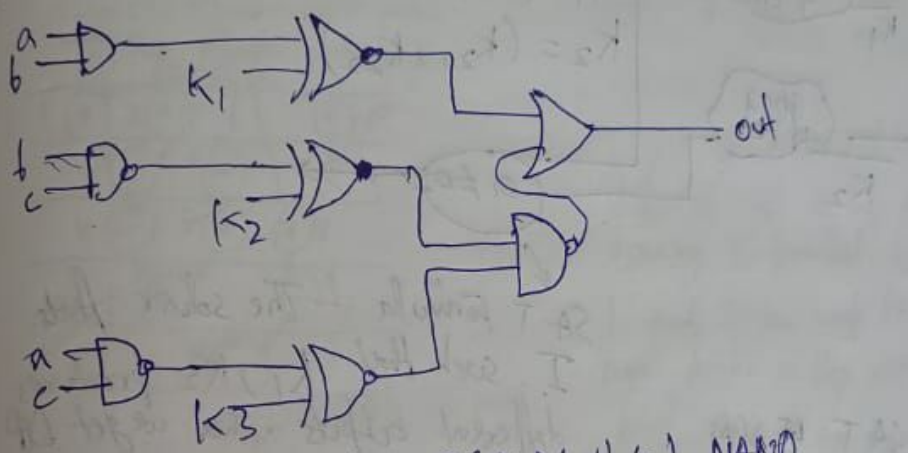
Integer Domain : $x + y > 10$
↓
SMT solver



Combinational Logic Locking

Truth table ... CAT Attack

Locked Design



For design we can solve this formula

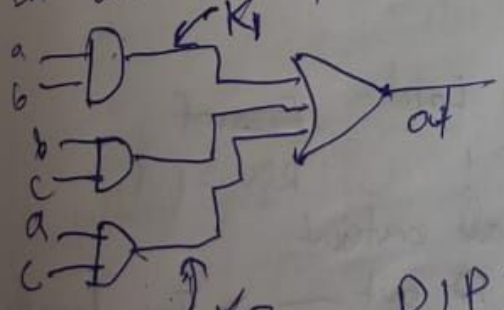
out = OR(XNOR(AND(a,b),K1), NAND(XOR(NAND(b,c),K2), XNOR(NAND(a,c),K3)))

if (c)
a = x+y
else
a = x-y

$\Leftrightarrow$ a = (ite, c, x+y, x-y)

if-else

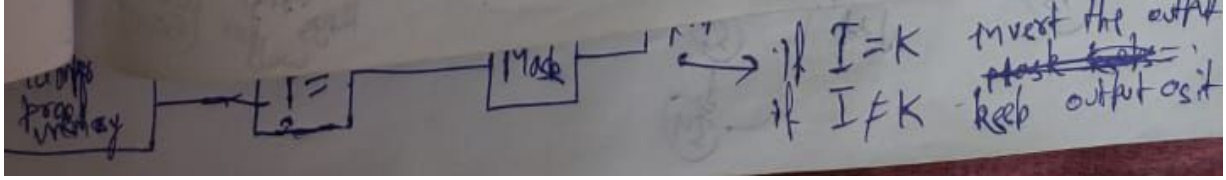
But what to do for C code



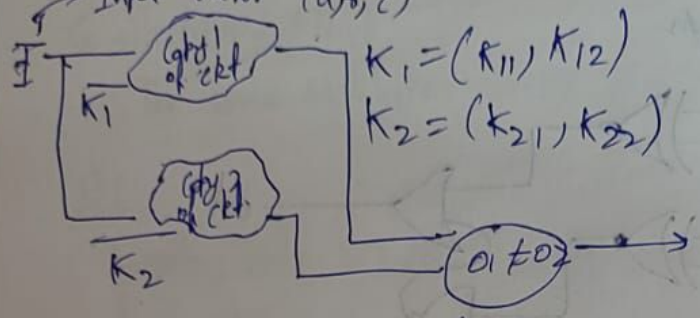
a	b	c	Actual op	Locked design	Possible key
0	1	1	1	1	00, 01, 10, 11
0	0	0	0	0	00

Not good does not narrow down keys

DIP
Distinguishing Input Pattern → identifies atleast one wrong key

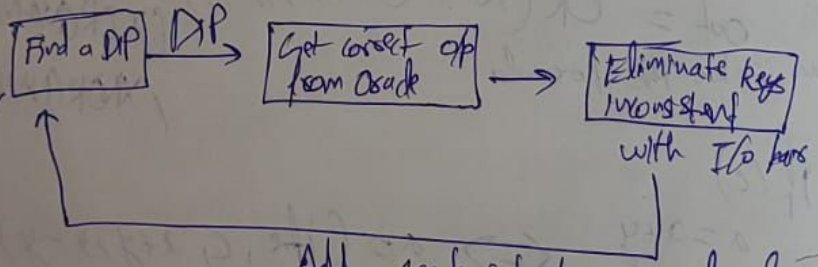


an avoid overhead since if used recomputing the functions of a false key as it can simply borrow output on the other hand



SAT Formula ... The solver finds I such that K_1, K_2 produces different outputs. Thus we get DIP

SAT steps



Add constraint to SAT formula corresponding to wrong key

once NO DIP
Run SAT solver for formula $O_1 = O_2$
to get one of the possible correct key

a	b	c	Actual	Locked	Encoded
0	0	0	0	1	key = 1 1

So add constraint

1 = out → This formula eliminates other keys for wrong (0,0,0)s as well

a=0, b=0, c=0

Combinational Logic Locking

Time taken by SAT Attack

$T = \sum_{i=1}^{n} t_i$

n = # iterations = # DIPs

t_i = time taken by SAT solver to get DIP$_i$

Either increase n or t_i to thwart SAT attack

Strong DIP – eliminates large number of incorrect keys

Weak DIP – " small " " "

If we ensure that we get only weak DIP we can increase n.

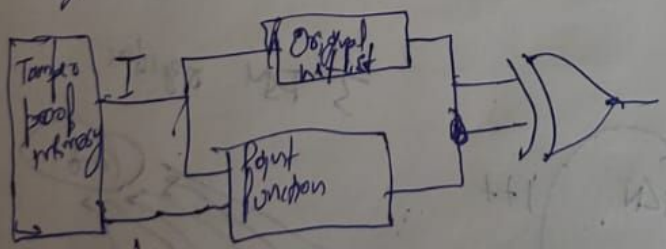
We do this by modifying circuit

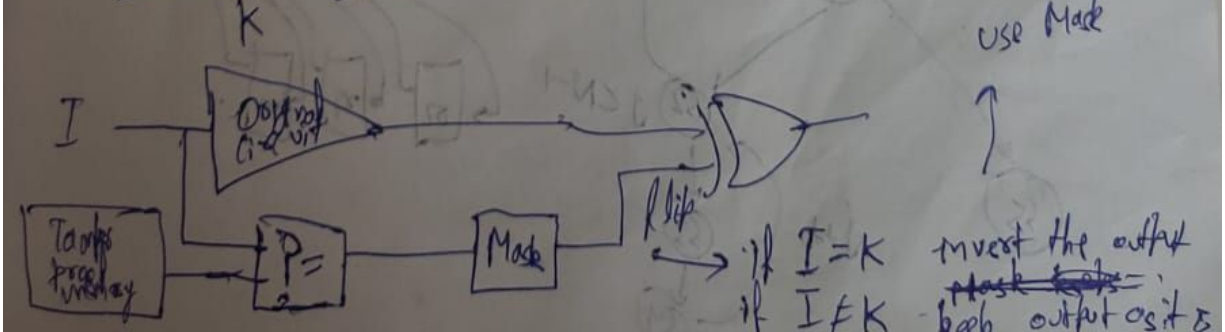
⇓

Each Input partitions keys into correct/incorrect based on corresponding output. If we ensure this partition is skewed for each input then number of keys eliminated is very less.

SARLOCK (SAT Attack Resistant Logic Locking)

Modify design so that it generates weak DIPs





Segm graph

Compatibility Graphs (For

$G = (V, E)$, $V =$

$E = (v_i, v_j)$ if sch

not co

v_i, v_j are

if $A_j \geq d_i + d_i$ o

directed edge

$v_i \rightarrow v_j$

ANTI SAT (In ANTISAT

AND Tree

Cycle SAT → Prevents SAT from working in even 1 iteration

The problem with SARLOCK, ANTISAT, AND TREE is low output corruptibility thus ~~~~ locked circuit gives almost correct output and susceptible to structural attacks because

~~~~

SARLOCK, ANTISAT is susceptible to structural attack since original circuit is left untouched

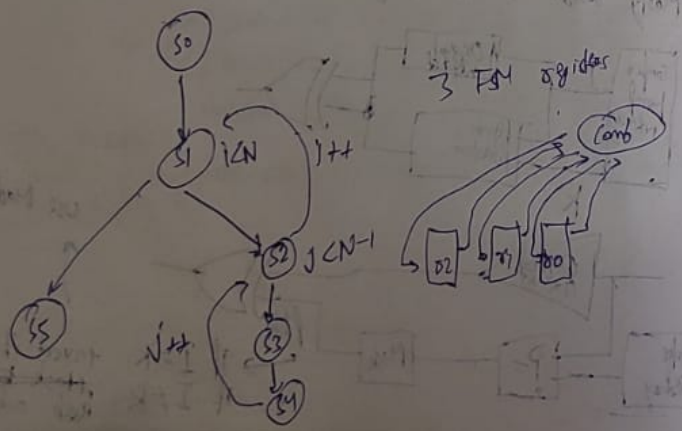
SFLL – HD (original circuit is also modified to create Functionality Stripped Circuit FSC)

Sequential Locking

Registers/FF

→ Datapath registers → usually connected in scan chain

→ FSM registers → not connected " " "



For logic locking attack we have oracle, locked netlist and their scan chain

If we lock scan chain we can't get internal values for each input output thus we can't apply SAT attack to a circuit

- FSM locking → ~~Not yet to~~ Unbroken
  - Identify control the FSM in design
  - Add a authentication block
  - Add a dummy FSM
  - The ~~set~~ start state of original FSM reachable only for a correct sequence of inputs which act as key otherwise it moves to dummy start state
- Lock Scan Chain of netlist
- Lock Scan Chain of Oracle
~~~~

Conflict Graph

$\chi(G)$ → Maximum nodes having same color

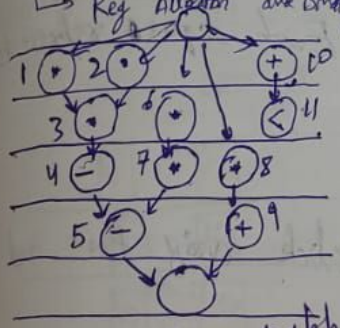
Resource Allocation & Binding

Two methods to solve

→ FU Allocation and Binding

→ Reg Allocation and Binding

(The part before this is only in another notebook and before that is in this notebook)



Seqn graph

1 and 2 must map to different FU as they are running in parallel.

1 and 3 can map to same FU as they ~~can run~~ don't run in parallel

Compatibility Graph (For each type of graph)

$G_T = (V, E)$, $V = \{v_i / v_i$ is a node in sequencing graph, $i = 1 \ldots n\}$

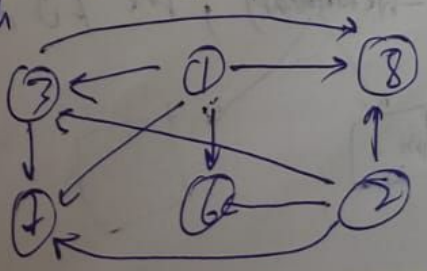
$E = (v_i, v_j)$ if schedule of v_i and v_j are not overlapping

v_i, v_j are not overlapping if $t_j \geq t_i + d_i$ or $t_i \geq t_j + d_j$

↓ directed edge $v_i \rightarrow v_j$

↓ directed edge $v_j \rightarrow v_i$

Compatibility graph for mult

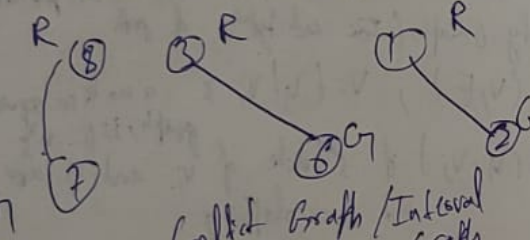


Clique cover problem

Identify min disjoint cliques that cover all the nodes of compatibility graph. Each clique represents one FU

$\#FU = K(G_1(V,E))$

Identify the maximum click every time and remove corresponding edges



Conflict Graph / Interval Graph

We apply graph coloring here to find how many colors are required

FU R = {1,3,5}

FU G = {2,6,7}

For generic behaviour with (loop ~~if~~ if-else, fn call), the problem is NP-complete. Behaviour with only single basic block (non-hierarchical), the FU allocation is polynomial time solvable.

Left Edge Algorithm

Conflict Graph, Interval Graph

Every Interval Graph is Conflict Graph but not the other way around.

⇓

can be mapped to intervals