given v_q is scheduled before all others

$$TF(v_1, l) = SF(v_i, l) + PF(v_m, l)$$
$$+ PF(v_n, l) + SF(v_j, l)$$
$$+ SF(v_k, l).$$

⇓
we don't consider already scheduled nodes
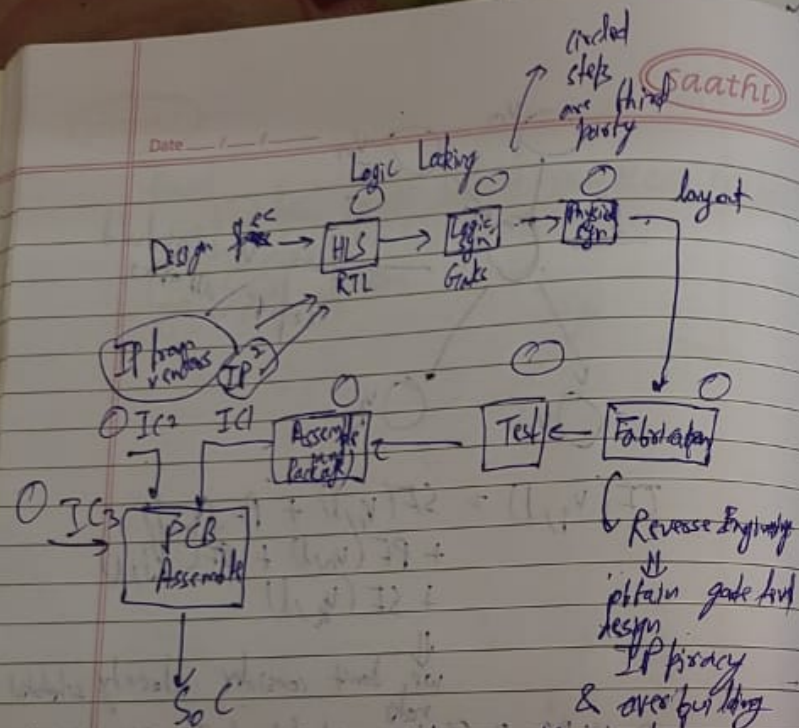
FD-MLRC is $O(n^2)$ as calculation of TF takes $O(n)$ time

FD-MR-LC (g, λ) od mm a

while (all op are scheduled)
{
     compute the time frame (mobility
     compute operation & type probability
     compute self force, pred/succ force
     Schedule the opn v_i with least total force in its time frame
}

$\longrightarrow O(n^3)$

Logic Locking

circled steps are third party

$$\text{Design spec} \rightarrow \boxed{\text{HLS}} \rightarrow \boxed{\text{Logic syn}} \rightarrow \boxed{\text{Physical syn}} \rightarrow \text{layout}$$
$$\text{RTL} \qquad \text{Gate}$$

(IP from vendors) → IP

○ IC2 IC1 → $\boxed{\text{Assemble packet}}$ ← $\boxed{\text{Test}}$ ← $\boxed{\text{Fabrication}}$

○ IC3 → $\boxed{\text{PCB Assemble}}$

↓
SoC

C Reverse Engineering
↓
obtain gate level design
IP Piracy
& overbuilding

Hardware Trojan → Malicious Modification of design

To prevent reverse engineering the soln is logic locking

Logic
Locking

X-Nor

d

Xor

$k_2 = 0$
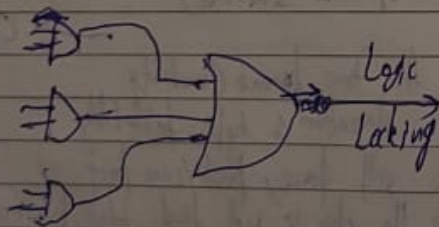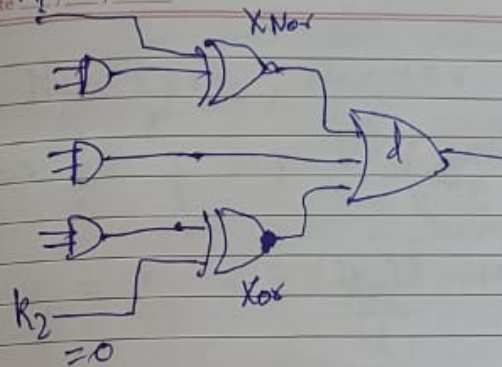
This circuit is equivalent to earlier only if $k_1 = 1$, $k_2 = 0$
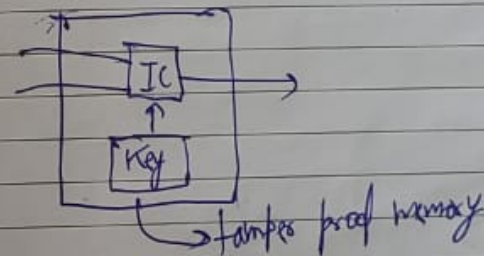
→ tamper proof memory
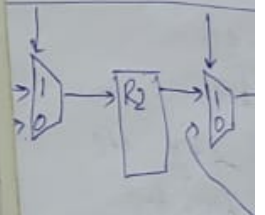
M L - RC:

Objective is to increase concurrency to decrease latency

List: MLRC (G, u)

$l = 1$

while (any operation remains to be scheduled) {

for each resource type R {

U_{lk} = candidate available in l,

T_{lk} = running operation in l

Calculate the total force of $v_i \in U_{lk}$

Select $S_k \subseteq U_{lk}$ s.t $|T_{lk}| + |S_k| \leq a_k$

and TF minimized

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
(oracle) key

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
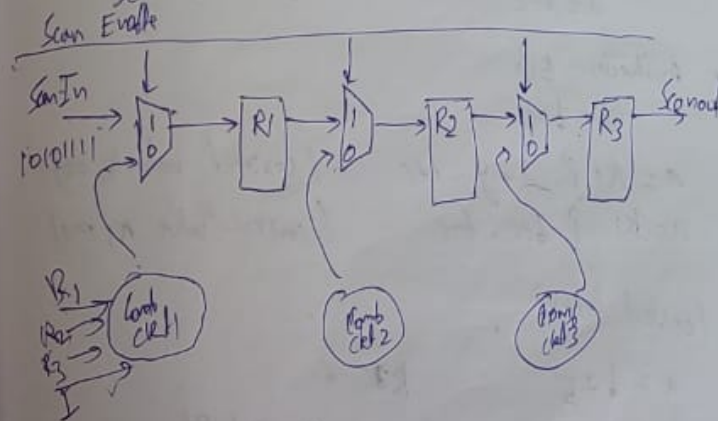
n-reg in scan chain

- Run n cycle with Scan Enable = 1 (This stores the input pattern in registers)

- one clk, run the design ScanEnable = 0 (we know the expected output for each register)

- Run n clk with SE = 1 (we get the output pattern from register)

C-level faking

- ~~Abilities~~ Conditional Exp

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

⇓

$k_2 = 0$		else $k_2 = 1$

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

- Arithmetic Exp

$$a = b+c$$

$a = k_1 ? 2*y : b+c \qquad (\text{correct value } k_1 = 0)$

$a = k_1 ? b+c : b-c \qquad (\text{correct value } k_1 = 1)$

- Constant faking

$a = b + 5 \qquad\qquad k = 6$

$$
\begin{array}{ccc}
5 & & 0 1 0 1 \\
\oplus & = & 0 1 1 0 \\
6 & & \\
\hline
& & 0 0 1 1
\end{array}
$$

$a = b + 3 \oplus k$

Logic locking (H LOCK)
- Behavioral level ⟶ C-level ⟶ 2 levels at which locking
                    ⟶ during HLS   can be done
                                       at behavioral level

- RTL ⟶ (TAO) ASSUME
          HOST
          Dishonest Oracle

- Gate level
- 3 types of locking in general { = constant   • highly config for
                                  = operation   • if key not
                                  = condition   know operation
                                                     lost
                                                   • semantic info

H-LOCK   TAO   ASSURE/HOST

   C   C   C
   | | |
   ↓   ↓   ↓
[Locking]   [TAO HLS+Lock]   [HLS]
   | | |
   ↓   ↓   ↓
Locked C-code   Locked   RTL
   | RTL |
   ↓   ↓
[HLS]   [Locking]
   | |
   ↓   ↓
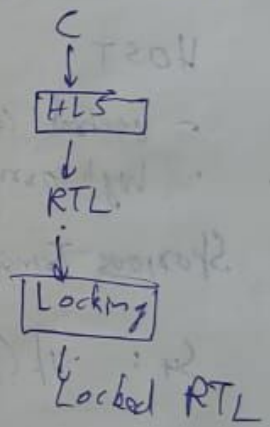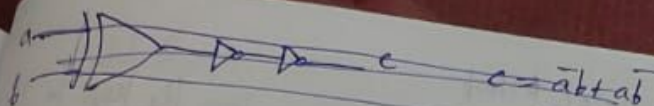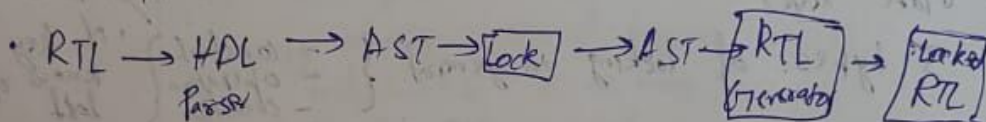Locked RTL   Locked RTL
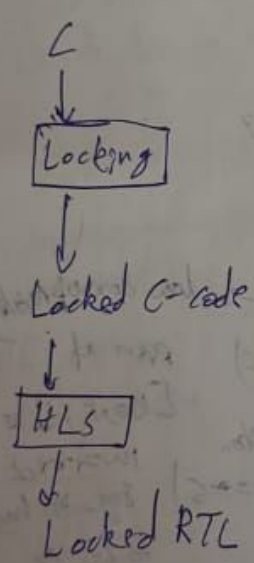
• Advantage of Locking at higher level ⟹ we have semantic
                                          info that can be used

Comparison:

HLock   TAO
• Easy to implement   - harder to implement
- HLS independent   • less overhead
  (not dependent on
  software)

$$c = \bar{a}b + ab$$
$$\bar{a}b + a\bar{b}$$
$$(a + b)(\bar{a} + \bar{b})$$

The downside of RLL is that RLL circuit need to output even for incorr... the state-...

- TAO can avoid overhead since it avoid recomputing the operations in case of a fake 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+a : b−c) causing overhead

ASSURE / HOST

- RTL $\longrightarrow$ HDL $\longrightarrow$ AST $\longrightarrow$ [Lock] $\longrightarrow$ AST $\longrightarrow$ [RTL Generator] $\longrightarrow$ [Locked RTL]
             Parser

ASSURE $\begin{cases} -\text{Constant} \\ -\text{Operation} \\ -\text{another} \end{cases}$

HOST

- control corruptability $\longrightarrow$ low corruptability
- high corruptability $\longrightarrow$ corrupt control

Spurious Transition in controller

$\longrightarrow$ low corruptability

S_4 : if ([k_0, k_1] = 2b'00 & reg_∂_1 == 10)   even if
        next state = S8   Spurious transition   [k_0, k_1] is
    else if ([k_0, k_1] = 2b'01 & reg_∂_2 == 5)   incorrect
         next state = S9   reg_∂_1 has
    else   to be 10
        next state = S5 <original transition for spurious transition

Gate level logic locking
Combinational Part

→ Pre SAT Era - RLL, FLL, SLL
→ SAT Attack
→ Post. SAT — SarLock, Anti-SAT, Cyde-SAT, SFLL
          Provably Secure Dishonest Oracle, Delay
          Lock, Clock locking

⇓ Locks combinational load
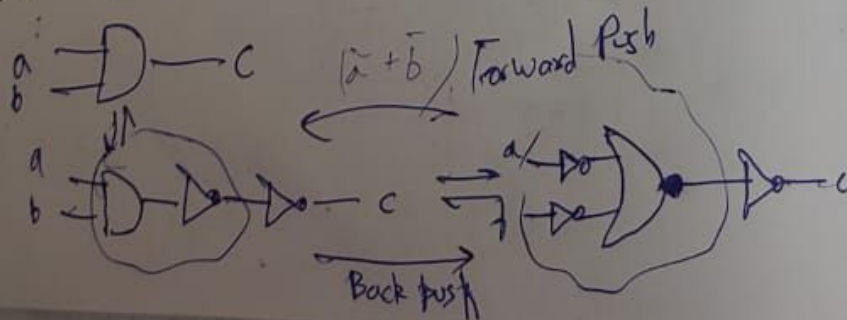
Sequential Part

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

Random Logic Locking.
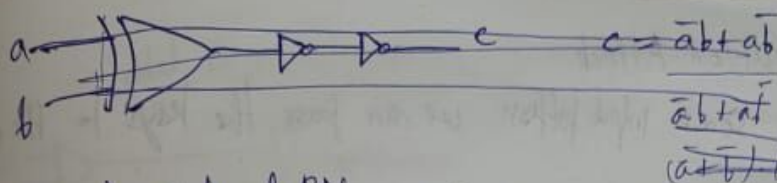XOR/XNOR based key insertion at random locations
⇓
Key easily identified for XOR $= 0$ and for XNOR $= 1$

• Soln — Bubble Pushing

$(\bar{a} + \bar{b})$ Forward Push

Back push

Combinational Logic Locking

$$c = \bar{a}b + a\bar{b}$$
$$\overline{\bar{a}b + a\bar{b}}$$
$$(a+b)(\bar{a}+b+1)$$

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 $\begin{cases} \text{Stuck at zero} \\ \text{stuck at one} \end{cases}$ FI$_G$ (Fault impact of G)
$$= (P_0 \times O_0) + (P_1 \times O_1)$$

↓

VLSI Testing for physical flaws in the circuit

~~VLSI Testing of~~ Logic Locking

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

Fault excitation
Fault prop
Multiple fault

P random input patterns
Q random key values
L Locked Netlist
O$_L$ M-bit output

Tamper
proof
memory

P=

Mask

Flip

if I = K invert
if I ≠ K keep

avoid overhead since it avoid recompiling the executions
of a false key as f an simply borrow atba

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

SLL
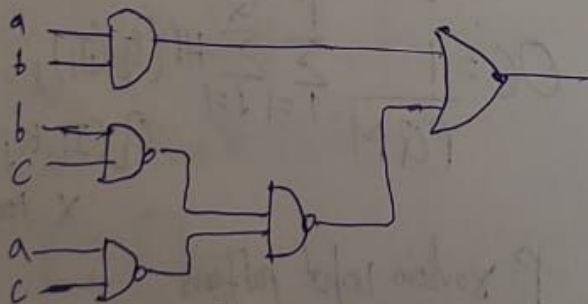Inserts keys to maximize interdependence by increasing
clique size

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

Integer Domain $\Rightarrow$ $x+y > 10$
$\downarrow$
SMT Solver

Combinational Logic Locking

Locked Design
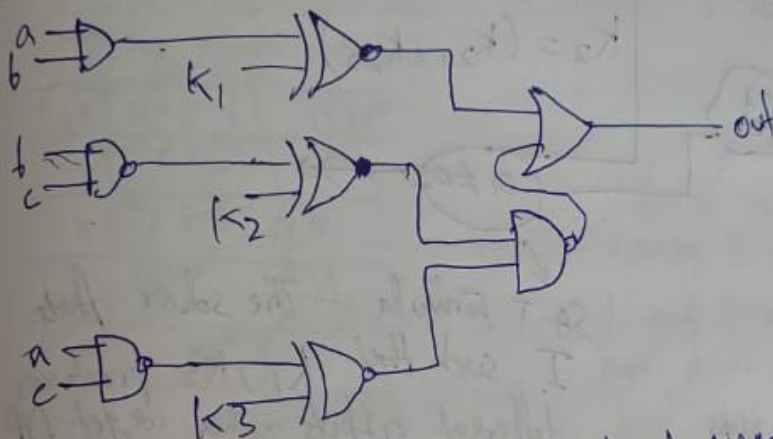
For design we can solve this formula

$$out = OR(XNOR(AND(a,b), K_1), NAND(XOR(NAND(b,c), K_2), XNOR(NAND(a,c), K_3)))$$

if (c)
$\quad a = x + y$ $\Longleftrightarrow$ $a = (ite, c, x+y, x-y)$
else
$\quad a = x - y$ $\qquad\qquad$ if-else

But what to do for C code

Not good
does n't narrow
down keys

	a	b	c	Actual o/p	Locked design	Possible key
	0	1	1	1	1	00/01
						10/11
	0	0	0	0	0	00
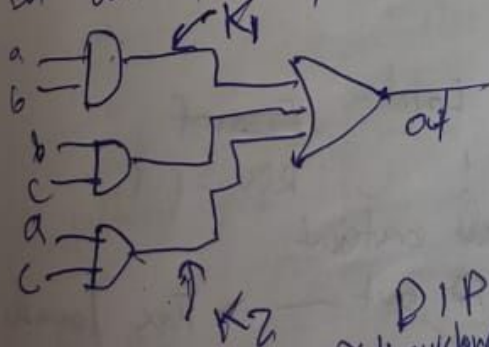
DIP
Distinguishing Input
Pattern → identifies atleast one wrong key

→ if $I = K$ invert the output
→ if $I \neq K$ keep output as it

an avoid overhead since it avoid recompiling the operations
of a false key as it can simply borrow attack
circuit on the other hand

→ Input vector (a, b, c)

$$K_1 = (K_{11}, K_{12})$$
$$K_2 = (K_{21}, K_{22})$$

copy 1 of ckt — K_1
copy 2 of ckt — K_2
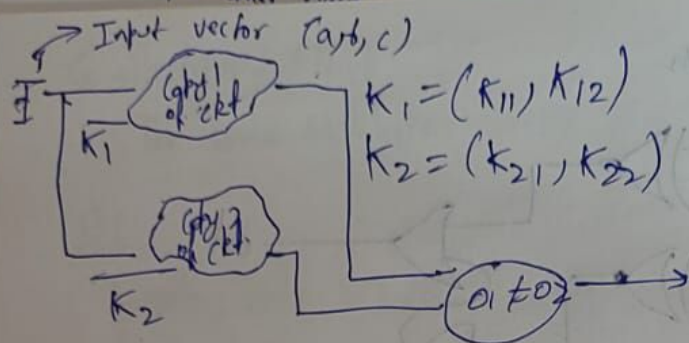
$O_1 \neq O_2$

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

SAT Steps

Find a DIP → DIP → Get correct o/p from Oracle → Eliminate keys inconsistent with I/O pairs

Add constraint to SAT formula corresponding to wrong key
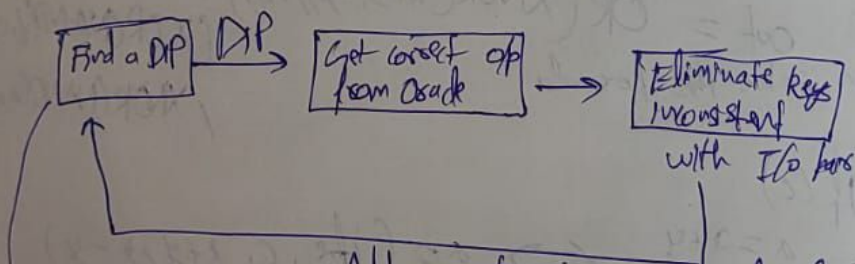
once NO DIP

Run solver for formula $O_1 = O_2$
to get one of the possible correct key

a b c	Actual Label	~~Expected~~
0 0 0	0 1	key = 1 1
0 0 0		

So add constraint

$1 = $ ~~out~~ out ⟶ This formula
$a = 0, b = 0, c = 0$ eliminates other wrong keys for (0,0,0) as well

Combinational Logic Locking

Time taken by SAT Attack

$$T = \sum_{i=1}^{\lambda} t_i$$
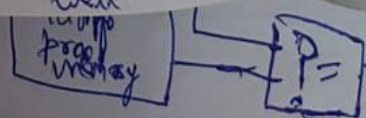
$\lambda = $ # Iterations = # DIP

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

Either increase λ 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 λ.
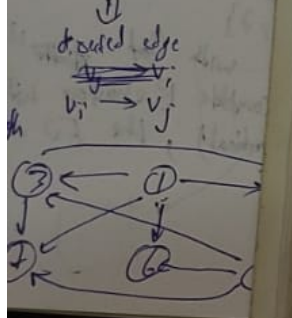We do this by modifying circuit

$$\Downarrow$$

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
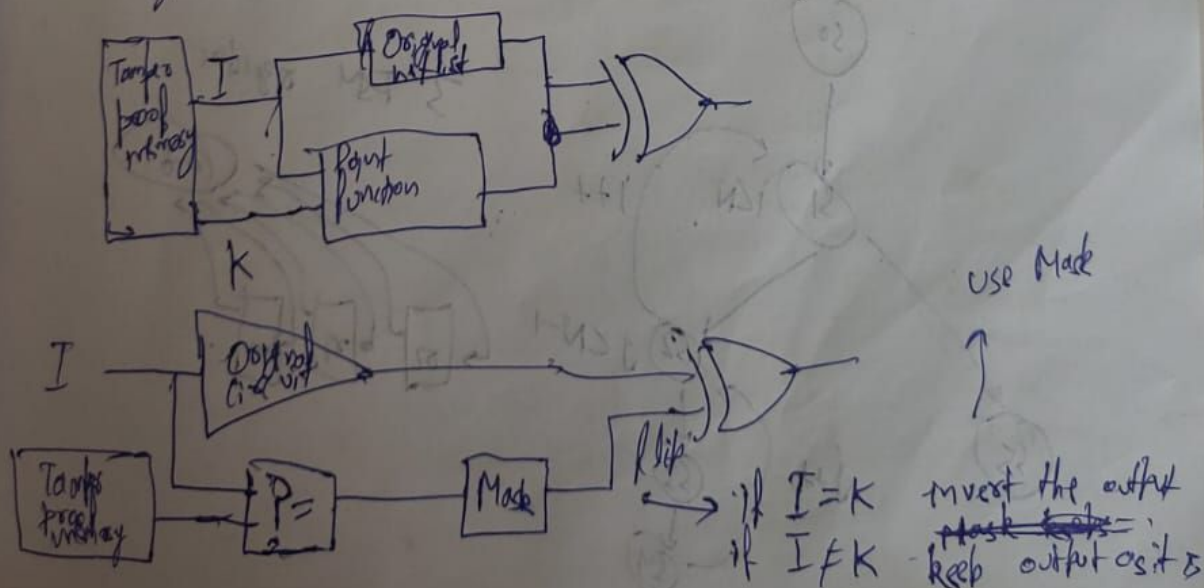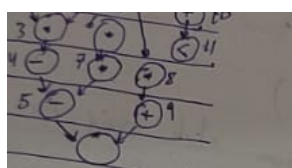
use Mask

→ if I = K invert the output
→ if I ≠ K keep output as it is

ANTISAT (In ANTISAT ready ... ... changed)
AND Tree
Cycle SAT → Prevents SAT from working in even / iteration

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

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 " "

$\frac{3}{8}$ FSM registers
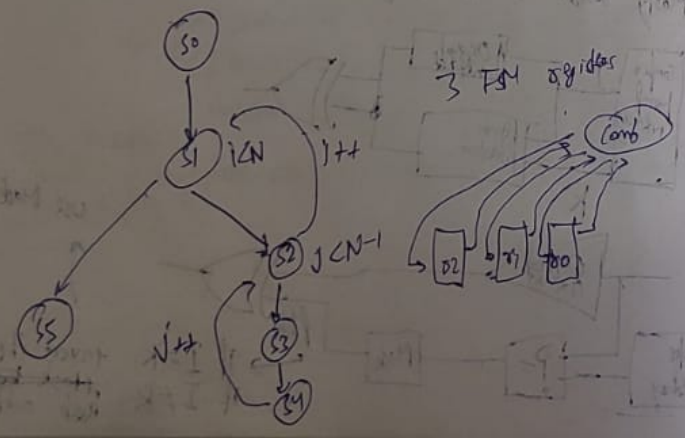
For logic locking attack
we have oracle, locked netlist and their scan chain

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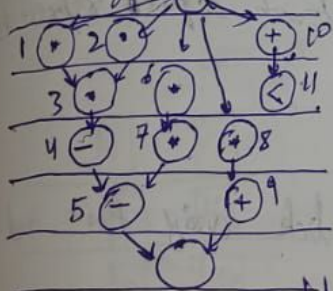each input/output thus we can't apply SAT attack to a
netlist

- FSM locking → ~~Not yet to~~ Unbroken

  - Identify controller FSM in design
    Add a authentication block
    Add a dummy FSM

    The ~~co~~ start state of original FSM reachable only for
    a correct sequence of inputs which act as keys otherwise
    it moves to dummy start state

- Lock Scan Chain of netlist

- Lock Scan Chain of Oracle

Resource Allocation & Binding

Two problems to solve

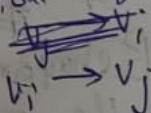↳ FU Allocation and Binding
↳ Reg Allocation and Binding

Sequ graph

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

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

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

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

Compatibility Graph (For each type of graph

$$G = (V, E), \quad V = \{ v_i / v_i \text{ is a node in sequence graph}, i = 1, ... n \}$$

$$E = (v_i, v_j) \text{ if schedule of } v_i \text{ and } v_j \text{ are not overlapping}$$

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

⟱   ⟱

directed edge   directed edge

$v_i \rightarrow v_j$ $v_j \rightarrow v_i$
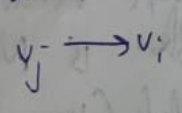
Compatibility graph for Mult

Compatibility Graph
$w(G) \Rightarrow M$
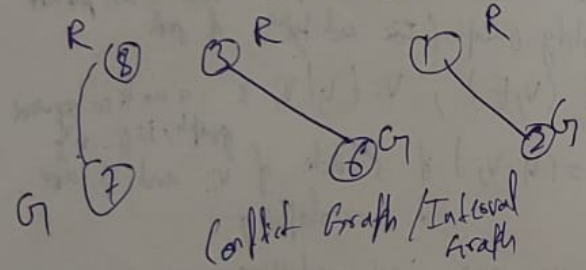$K(G) \Rightarrow$
Perfect G
$(G) = \chi$
$(G) =$

Clique cover problem

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

$$\# FU = k(G(V,E))$$

Identify the maximum clique every time and remove corresponding edges

Conflict Graph / Interval Graph

We apply graph colouring here to find how many resources are required

$FU \; R = \{1, 3, 8\}$
$FU \; G = \{2, 6, 7\}$

For generic behaviour with loop, if-else, for all) the problem is NP-complete. Behaviour with only single basic block (non-hierarchical), the FU alloc and is polynomial time solvable.
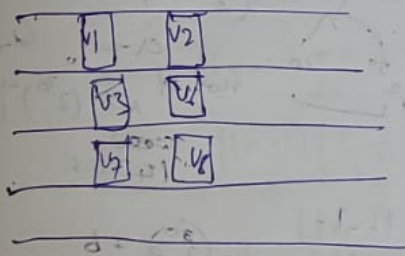
Left Edge Algorithm

Conflict Graph, Interval Graph

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

$\Downarrow$

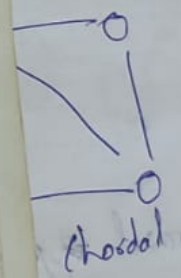Can be mapped to intervals

$k(G) \rightarrow$ clique cover
minimum
to cover

Perfect Graph
$(G) = x(G)$
$(G) = k(G)$

de having mo

(hosdol

erfect Group

se osisiden

$\Rightarrow$