EE392A [200259] Prof. Shubham Sahay

Week 2

Satisfiability Problem

References:

cs.sfu.ca[1], Udacity Theoritical CS[2], cliche_niche[3], ptolemy.berkeley.edu[4], Kamali[5] Subramanyan[5], SMT[6]

Purpose of the week:

Understand the boolean SAT problem. Solve a SAT example to get a gist of how solvers work and how to interact with them. Move on the analysing the SAT attack itself, and the variations that have come up recently.

Brief:

We saw about the SAT attack during IPPs, and how it uses a SAT solver to go through the input space, picks up a Differentiating Input Patterns (DIPs) that differentiates keys on the basis of oracle output and their outputs, and discards incorrect keys, reducing the search space depending on how effective were the DIPs that are found. The problem now lies in how to physically convert a locked netlist into SAT solver compatible form, and check it's performance for benchmarking circuits protection.

SAT Problem

The Boolean Satisfiability Problem, is a decision problem that infers whether a boolean formula can be true by finding the pattern of it's variables values which makes the formulae true.

More formally,

> Using Boolean logic, every propositional logic formula can be transformed into an equivalent conjunctive normal form (CNF), which can be exponentially longer.
>
> A literal is either a variable (called positive literal) or it's negation (called negative literal). A clause is a disjunction (union) of literals. A formula is in CNF form, if it is a conjunction of clauses. An example being the following:
>
> $$(x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge \neg x_1$$
>
> The SAT problem is to set the variables to a combination of {false, true} such that the formula becomes true.
>
> In the above problem, $x_1, x_2, x_3, x_4 = [0, 0, 0, 1]$ satisfies as one of the solutions.
>
> We'll also be assuming that the length of the formula is a function of polynomial in *n*, where *n* is the number of variables.

Week 2

By the definition of CNF, it directly corresponds to a PoS form of logic.

Note that SAT problem is NP-complete, i.e. any problem in NP can be polynomially reduced to this.

Any problem in NP can be solved in polynomial time, by a non-deterministic RAM.
This can be shown by thinking about how to convert any NP algorithm to a boolean formula.

For some SAT problems, it is useful to define *generalized conjuctive normal form* formula, as a conjunction of many *generalized clauses*, $R(l_1,...,l_n)$, for some Boolean function R and literals l_i.

SAT solvers using *DPLL* algorithm, *Semantic Tableau*, *Analytic Tableau* and some others. Most of them are available pre-implemented in Python.

CNF File description:

The CNF file format is an ASCII file format.

1. The file may begin with comment lines. The first character of each comment line must be a lower case letter "c". Comment lines typically occur in one section at the beginning of the file, but are allowed to appear throughout the file.
2. The comment lines are followed by the "problem" line. This begins with a lower case "p" followed by a space, followed by the problem type, which for CNF files is "cnf", followed by the number of variables followed by the number of clauses.
3. The remainder of the file contains lines defining the clauses, one by one.
4. A clause is defined by listing the index of each positive literal, and the negative index of each negative literal. Indices are 1-based, and for obvious reasons the index 0 is not allowed.
5. The definition of a clause may extend beyond a single line of text.
6. The definition of a clause is terminated by a final value of "0".
7. The file terminates after the last clause is defined.

Example of a CNF file, in DIMACS format:

```
c Mentioned in report
p cnf 4 3
1 -2 0
2 -3 4 0
-1 0
```

Upon providing this to the solver,

```
➜  playground ./solver < ./report.cnf
SAT
-1 -2 -3 4 0
```

CNF form is preferred because it's easy to detect conflicts and also easy to remember partial assignments that don't work out.

SAT Implementation Example

I'll use a classic problem that's convertible to CNF form as an example for understanding conversion to that form from a given problem.

Week 2

Statement:

Coloring a graph using *k*-colors. Given an undirected graph *G(V, E)* and a natural number *k*, is there an assignment color possible for $V \to \{1, 2, \cdots, k\}$ such that $\forall (u, v) \in E : \text{color}(u) \neq \text{color}(v)$?

Steps:

Generating CNF

We need to derieve a CNF formula *φ* such that *φ* is satisfiable iff G is *k*-colorable.

1. Variable encoding: Consider a simple encoding of the situation, by assigning one boolean variable to each node and color pair. Thus the variable $x_{i,j}, 1 \leq i \leq |V|, 1 \leq j \leq k$ is defined, and is *true* when *i*th node is assigned the *k*th color. This generates $k|V|$ variables.
2. Constraints encoding: Informally, we would like each node to be colored with some color, but a unique color, and need it's neighbours to be colored not with the same color. This can be broken down into 3 constraints
   a) At least one color: For v_i node, clause can be written as
   $$x_{i,1} \vee x_{i,2} \vee \cdots x_{i,k}$$
   b) At most one color (Optional constraint): For v_i node, if color *c* has been assigned, you can't assign it *d*, this being written as $x_{i,c} \to \neg x_{i,d}$, which can be re-written using the implies clause as $\neg x_{i,c} \vee \neg x_{i,d}$. Therefore for all vertices and for all pair of colors
   $$\bigwedge_{1 \leq c < k} \bigwedge_{c+1 \leq d \leq k} (\neg x_{i,c} \vee \neg x_{i,d})$$
   c) Different colors adjacent: For every edge $(v_i, v_j) \in E$, if v_i is colored with *c*, v_j must not be colored with c. This sounds like the above clause and can be written similary
   $$\bigwedge_{1 \leq c \leq k} (\neg x_{i,c} \vee \neg x_{j,c})$$
3. Putting constraints together
   {
   $i \in V, (i, j) \in E$:
   $$(x_{i,1} \vee x_{i,2} \vee \cdots x_{i,k}) \wedge (\bigwedge_{1 \leq c < k} \bigwedge_{c+1 \leq d \leq k} (\neg x_{i,c} \vee \neg x_{i,d})) \wedge (\bigwedge_{1 \leq c \leq k} (\neg x_{i,c} \vee \neg x_{j,c}))$$
   }
   Clearly, total number of clauses generated are $|V|(1 + k(k-1)/2) + cE$
4. Create a program that would generate .cnf file from this structure.
   This was done by encoding (*v*, *e*) as a variable.

```
    def index(v, k):
        global nV, K
        return v * (K) + k - K

    def unindex(idx):
        global nV, K
        idx += K
        k = idx % K
        v = int(idx / K)
        if k == 0:
            k = K
            v = v - 1
        return (v, k)
```

Such encoding functions helped in transforming the above formula directly to code.

Using SAT Solver over .cnf

This culminated with running the SAT solver based on *Conflict Driven Clause Learning*:

```
➜  playground time ./kcolor_cnf.py -K 6 -nV 80 -nE 600
Generated graph with 80 vertices, 600 edges, 6 colors and 7928 clauses
Saved as graph.cnf
SAT
Node coloring saved in ./output.txt
./kcolor_cnf.py -K 6 -nV 80 -nE 600  14.57s user 0.26s system 101% cpu 14.540
total
```

For smaller inputs like below's adjacency list

```
K: 3; nV: 5; nE: 6;
5: 1 2 3 4
4: 1 5
1: 4 5
3: 2 5
2: 3 5
```

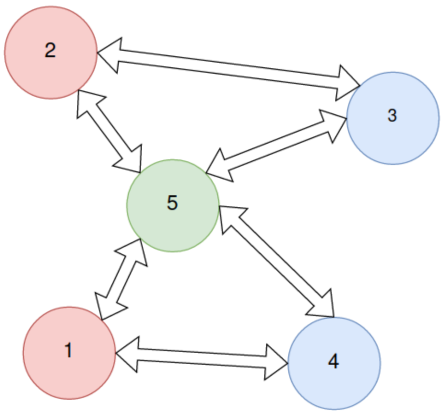


Figure 1: one of the possible solution for the given graph

We obtain colors as

```
v: 1 c: 2
v: 2 c: 2
v: 3 c: 1
v: 4 c: 1
v: 5 c: 3
```

Therefore once *.cnf* is created, the same method can be used to SAT, and decoding the states is equivalently performed.

Relating to SAT Attack

The original Satisfiability Attack was was first described in a paper titled "SAT-Based Reverse Engineering of Digital Circuits" by Brayton, et al. published in 2004. In this paper, the authors introduced the concept of using Satisfiability (SAT) solver to reverse engineer digital circuits, including logic-locked circuits, by finding a satisfying assignment that represents the secret key to unlock the circuit.

Week 2

The paper itself wasn't available over the internet, as far as I could search. I found another paper description, "Breaking Logic Locks with Boolean Satisfiability" by Xinyu Wang, et al. (2016) - which apparently provides a step-by-step guide to performing a SAT attack on logic-locked circuits and includes several case studies to demonstrate the effectiveness of the technique, but even that paper **isn't** available.

In Kamali et al., I found the following:

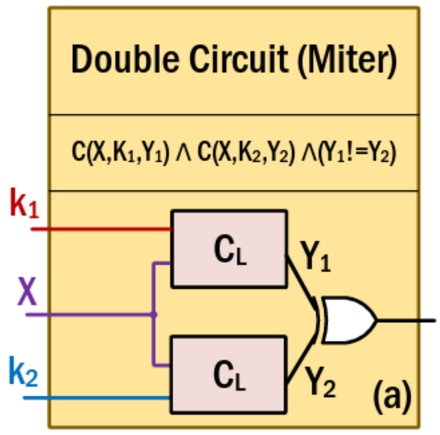


Figure 2: Miter ensuring different keys with different outputs

This block would output 1 when for the same input *X*, the locked circuit C_L gives different outputs, thus by definition making *X* a Differentiating Input Pattern. The key values (*k1* and *k2*) and the primary input pattern *X*, will be found by a SAT solver query.

This eval block checks by the oracle, giving out 1 when both keys give the same output, i.e. X_d doesn't behave as a DIP. (?I think the figure is wrong with the AND gate being replaced by XOR, but otherwise is confirmed later?)

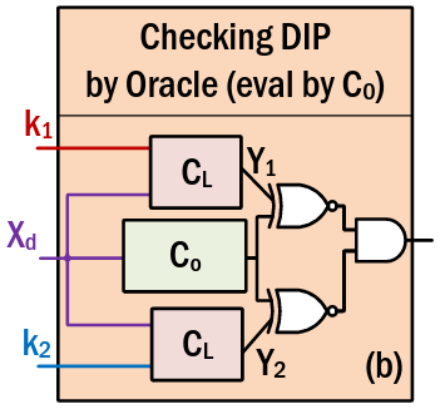


Figure 3: eval block including oracle C_0

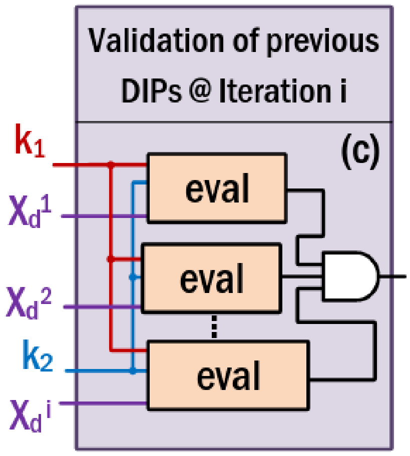


Figure 4: Validating DIPs at ith iteration for all previously found DIPs

All previously recognised DIPs are used to check whether the keys k_1 and k_2 are evaluated as 1 by each and every previously seen DIP or not. I'm not sure why we would want that to happen, because a DIP may differentiate 2 keys *k1* and *k2*, while might not differentiate *k2* and *k3* or *k4* and *k5*.

Using the above blocks and the CNF form, represented by *Learned Clauses*, the SAT circuit is created for *i*th iteration

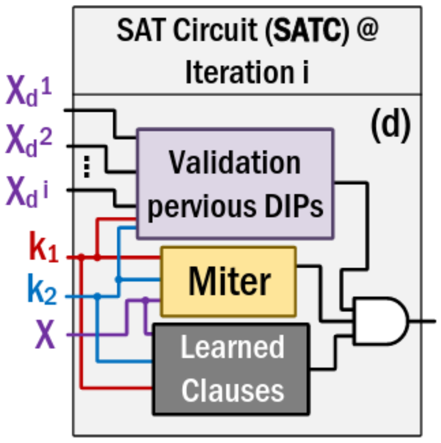
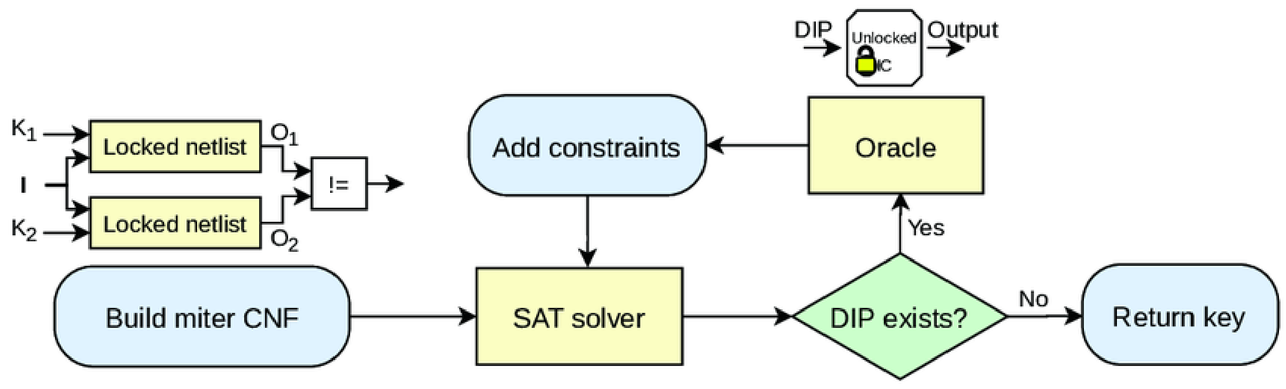


Figure 5: SATC at iteration i

Week 2



Algorithm

An encrypted combinational logic circuit can be represented by

$$\mathcal{C}(\vec{X}, \vec{K}, \vec{Y}) \subseteq \{0,1\}^{M+L+N}$$
$$\text{where } \vec{X} \in \{0,1\}^M; \vec{Y} \in \{0,1\}^N; \vec{K} \in \{0,1\}^L$$

such that $\vec{X}$ represents *M* primary inputs to the circuit, $\vec{K}$ represents *L* length key, $\vec{Y}$ represents *N* primary outputs of the circuit. Also,

$$\mathcal{C}_O(\vec{X}, \vec{Y}) \subseteq \{0,1\}^{M+N}$$

represents the oracle, which is a decoded and correctly behaving circuit. Note that if $\vec{K}$ is the correct key provided to $\mathcal{C}$, then $\mathcal{C} \equiv \mathcal{C}_O$. Note that the attacker doesn't have the relation $\mathcal{C}_O$, but can apply input patterns and observe outputs of the same. This is modelled by $eval : \{0,1\}^M \to \{0,1\}^N$.

The attacker goal now becomes obtaining K_C such that

$$\forall \vec{X} : \mathcal{C}(\vec{X}, \vec{K}_C, \vec{Y}) \iff eval(\vec{X}) = \vec{Y}$$

This is equivalent to solving the quantified boolean formula (a boolean formula with the formula is constraint of course, but also constraints in input/variables)

$$\exists K \ \ \forall X \ \ \mathcal{C}(\vec{X}, \vec{K}, \vec{Y}) \wedge \mathcal{C}_O(\vec{X}, \vec{Y})$$

(?I don't understand the result of boolean vectors getting and-ed here, maybe this is useful?)

In the above, the attacker can't construct a formula for $\mathcal{C}_O$, and has to use *eval* for a small number of inputs possibly.

For a given set of inputs $\{\vec{X}_1, \vec{X}_2, \cdots, \vec{X}_p\}$ and outputs $\{\vec{Y}_1, \vec{Y}_2, \cdots, \vec{Y}_p\}$, the adversary can find out a $\vec{K}$ that satisfies $\wedge_{i=1}^{p} \mathcal{C}(\vec{X}_i, \vec{K}, \vec{Y}_i)$. But upon a new ($i$+1)th observation, there's no guarantee that the $\vec{K}$ found before would satisfy $\wedge_{i=1}^{p+1} \mathcal{C}(\vec{X}_i, \vec{K}, \vec{Y}_i)$. The insight lies in considering keys as a part of *equivalent classes*, instead of individually.

The author explains to look for a member of the *equivalence class* of keys which produce the correct output for all input patterns seen. For this, iteratively rule out *equivalence classes* which produce the wrong output values for at least one DIP. Note that $\vec{X}_d$ behaves as a DIP iff

$$\mathcal{C}(\vec{X}^d, \vec{K}_1, \vec{Y}_1^d) \wedge \mathcal{C}(\vec{X}^d, \vec{K}_2, \vec{Y}_2^d) \wedge (\vec{Y}_1^d \neq \vec{Y}_2^d)$$

The second insight is if $\vec{X}_d$ is found, then the oracle can be used to reject either of $\vec{K}_1$ or $\vec{K}_2$ or both as not being of the equivalence class of the correct key.

The following was also defined in Kamali et al., orignally from Subramanyan et al.

```
1. function SAT_Attack(Circuit Cl, Circuit eval)
2.       i <= 0;
3.       F[0] <= Cl(x, k1, y1) ^ Cl(x, k2, y1);
4.       while SAT(F[i] ^ (y1 != y2)) do
5.             xd[i] <= sat_assignX (Fi ^ (y1 != y2));
6.             yd[i] <= eval(xd[i]);
7.             F[i+1] <= F[i] ^ Cl(xd[i], k1, yd[i]) ^ Cl(xd[i], k2, yd[i]);
8.             i <= i + 1;
9.       kc <= sat_assignK1 (F[i]);
```

Note that at line 3, we start from some initially known DIP. The method `sat_assign` is unclear. Also the idea behind doing *Cl*(x, k1, y1) ^ *Cl*(x, k2, y1) doesn't add up. Also how does eliminating a key, tell about it's equivalence class, and where are we iterating over the key space, it seems K_1 and K_2 remains the same throughout, though it doesn't.

Piecing all together:

Understood the power of changing perspectives. We go from a graph question to boolean formula, then used a commonly used and cutting-edge SAT solver to find the 2-SAT problem in polynomial time. We have to extend the idea of identifying apt variables, constraints and represeting them as boolean clauses in order to implement SAT attacks. I also tried analysing how the SAT attack works as described in the paper by Pramod Subramanyan. Need to continue this analysis more logistically.