# CS 675A: Homework #2

Sujit Kumar Muduli
Supriya Suresh

Posted: 26 January 2020
Due: 6 Feb 2020, 11:59 PM

This homework will explore the use of SAT solvers for automated reasoning in computer systems. The git repo containing the skeleton code is available at: `https://git.cse.iitk.ac.in/spramod/cs675a-2020-hw2`. Start by cloning this repository.

## 1 Circuit Equivalence (20 points)

In this problem, we will pick up where we left off in HW1 and implement equivalence checking of boolean expressions using a SAT solver.

### Problem

- Skim through the class definitions and their member functions in the following files; these files will be useful in implementing your solution.. (You should not modify either of these files.)

  1. `Solver.scala`
     - `class Solver`: This is a wrapper class over the Sat4j SAT solver. Look at the example in `testSolver` in `Main.scala`. This example shows how you would use the Solver class. In particular, note how a model (an assignment to the variables in the SAT instance) is extracted from the Solver.
  2. `SatAdapter.scala`
     - `class PositiveLiteral`: represents a positive literal.
     - `class NegativeLiteral`: represents a negative literal.
     - `class Literal`: Both of the classes above are subclasses of the abstract class Literal. The method `toInt` in literal returns a positive integer for positive literals and negative integer for negated literals. The unary `~` operator returns a `NegativeLiteral` if given a `PositiveLiteral` and vice versa.
     - `class Clause`: each clause is a list of literals. The `Solver` class provides function to add Clause instances to the sat solver.

- You need to implement a converter from boolean expressions (represented by the `Expr` class and its subclasses) to conjunctive normal form (CNF) using the Tseitin transformation. Create a separate class/object in a new file `CNFConverter.scala` in the `src` directory for this conversion.

- Then complete the implementation of `checkEquivUsingSat` in `Evaluation.scala`. Use your implementation in `CNFConverter.scala` to generate CNF and add the resulting clauses to a SAT solver object. If the expressions are not equivalent, `checkEquivUsingSat` should also returns an input value for which the two circuits produce different outputs.

There are some tests in `Main.scala` that you can use to ensure your code works as expected. Your assignment will be graded on a larger and more complex set of circuits, so be sure to supplement these tests with your own.

## 2 Differencing Firewalls

In this problem, you will use a SAT solver to find the precise set of packets on which the functionality of two firewalls differs. For the purposes of this problem, a firewall is a device that sits in between the Internet and a private computer network. The firewall has a set of programmable "rules." Each incoming packet from the Internet is examined based on these rules and is either let through (accepted), or rejected (denied).

| CIDR Notation $A.B.C.D/n$ | Decision | Notes |
|---|---|---|
| 172.27.17.1/24 | ACCEPT | matches when $A = 172 \wedge B = 27 \wedge C = 17$ |
| 128.112.1.1/16 | ACCEPT | matches when $A = 128 \wedge B = 112$ |
| 18.0.0.1/8 | ACCEPT | matches when $A = 18$ |
| else | DENY | matches everything else |

Table 1: A example firewall. (See `https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing` for more information about CIDR notation.)

For our simplified scenario, we will model a firewall where each rule only examines a packet's source IP address. The firewall accept a packet if and only if the source IP matches one of the rules. All other packets are dropped. Further this problem will model source addresses as IPv4 addresses. These are 32 bit values and specified as $A.B.C.D$ where $A$, $B$, $C$ and $D$ are integers between 0 and 255 and refer to the first (most significant), second, third and fourth (least significant) byte of the IP address respectively. Firewall rules are specified in Classless Inter-Domain Routing (CIDR) notation. CIDR notation $A.B.C.D/n$ can be read as matching the $n$ most significant bits of the 32-bit integer specified by $A.B.C.D$. An example firewall is shown in CIDR notation in Table 1.

### Problem

Given a firewall $F$, let $accept_F$ be the set of IP addresses accepted (let through) by the firewall. The set $accept_F$ can be specified by its indicator function: a function that examines the source IP address specified by the tuple $(A, B, C, D)$ and returns true if the IP address is included in the set (accepted by the firewall) and false otherwise. For example, for the firewall shown in Table 1, $accept_F(A, B, C, D) \doteq (A = 172 \wedge B = 27 \wedge C = 17) \vee (A = 128 \wedge B = 112) \vee (A = 18)$.

Given two firewalls $F_1$ and $F_2$, define the *difference* of these firewalls, denoted by $F_1 - F_2$, as the firewall which accepts the set of packets defined by $accept_{F_1} \wedge \neg accept_{F_2}$. In other words,

$F_1 - F_2$ is precisely the firewall that accepts only the packets accepted by $F_1$ and rejected by $F_2$. Note this definition is equivalent to viewing the firewall difference as the set difference operator over the sets of IP addresses accepted by the two firewalls.

- You will be given two firewalls $F_1$ and $F_2$ specified in the format shown in Table 1. Your task is to use a SAT solver to compute the difference of these firewalls.
- You need to implement the `firewallDifference` function in `Firewall.scala`. This takes two firewalls $F_1$ and $F_2$ as input and returns a set $d = F_1 - F_2$ containing generalized IP addresses as the difference in the CIDR format.

Sample test cases have been provided in `test1()` function in the class `Firewall`. You can (and probably should) create new test cases of your own. Your implementation will be evaluated on a separate set of test cases.

**Hints:** You will do this by encoding $accept_{F_1} \wedge \neg accept_{F_2}$ as a SAT formula and enumerating all satisfying solutions to this formula. Use the technique of "blocking" each solution after you've enumerated it. Further, as explained in class, enumerating each satisfying solution individually is not tractable, so when the SAT solver returns a single solution, you will need to generalize or "expand" this solution to cover a set of solutions.

## 3 Trustworthy or not?

Suppose you work for the Indian Space Research Organization (ISRO) and need to have a integrated circuit (IC aka chip) developed that will go into the next satellite launch. Unfortunately, India does not have a state-of-the-art IC fabrication facility. So, while ISRO can design the chip, simulate it and test it, the actual manufacturing will have to be done in China. ISRO may be concerned that the design that they send to China may not be the one that is actually fabricated. In fact, some research has shown that changing just a few gates out of several million gates is enough to introduce serious security vulnerabilities into an IC.

Unfortunately, these modifications are nearly impossible to detect because the IC will work correctly for almost all inputs. It will only be a few secret and carefully chosen inputs for which the vulnerability will be triggered. In theory one could just test the output of the fabricated circuit for all possible inputs because there are only a finite number of inputs and compare this output with the expected output from the design for each input. If there are any discrepancies, we know for sure that there has been some hanky-panky in the fabrication facility. In practice, this does not work for circuits with more than $\approx 50$ inputs.

Your job is to use SAT solvers to detect such modifications by carefully choosing test inputs that will expose them.

### Problem

- Look at the file `Circuit.scala`.

  Your job is to complete the function `checkEquivalenceOfCircuits` which takes in the actual circuit that you wanted to manufacture - `actualCircuit` (also sometimes called the "golden" circuit) and the fabricated circuit you got from China – `givenCircuitEval`.

- The latter, `givenCircuitEval`, is not given as an `Expr` but instead as a Scala function that returns the result of the fabricated circuit's output given an assignment to the input variables of the chip in the form of a map - `Map[Variable, Boolean]`.

  This models the fact that once you have a manufactured circuit, you can't really see the gates that comprise it. Instead you can provide it inputs and see what the outputs are.

- A naïve implementation of this function without the use of SAT solvers has been presented to you. It creates all possible inputs to the actual circuit that you wanted to manufacture and compares the output of this so-called "golden" circuit and the fabricated circuit's output. If at any point, you find that the golden circuit's output and the fabricated circuit's outputs don't match, you know that the manufactured circuit is not the circuit you sent for fabrication.

- Now, you have a friend in the fabrication facility in China who gave you some essential information regarding the fabricated chip C. He said that it is possible that **either C is equivalent to the actual circuit you asked for** or ***exactly one* of the many AND gates in the actual circuit has been modified to an OR gate**, but he doesn't know which one.

- It is not possible to check the circuits for all possible input variable assignments as there may be thousands inputs to the chip. Hence, the naïve implementation will not work. With the information given by your friend, you know all possible circuits that you can get from the fabrication facility in China, by modifying each AND gate into an OR gate one at a time. Can you use *this* fact to reduce the search space?

- The function `checkEquivalenceOfCircuits` should return `false` if the circuit you wanted and the circuit you received from China aren't equivalent and `true` if they are. A sample test has been provided in `CircuitTest.scala` to help you understand better. You can add more tests here.

  The actual testcases will be much larger than the one given.

  Submit only the file `Circuit.scala`.