CS517: Theory of Computation
Project: Using SAT solver

Gayathri A Garimella, Akshith Gunasekaran

June 11, 2019

Introduction and Motivation

By design, the mining process of a Bitcoin protocol is computationally hard. We would like to translate the difficulty of mining to a SAT formula, which is a well-known NP-complete problem). Over the years, many SAT solvers and their optimizations have evolved. We will employ one such solver to find a satisfying assignment to our CNF-boolean formula and interpret this assignment as a solution to the mining process.

In this project, we consider a simplified version of the Proof-of-Work(PoW) protocol of the Bitcoin consensus. As suggested by the name, PoW ensures that no participant can acquire certain privileges in the protocol without investing a *large* amount of computational effort. The huge computational cost discourages adversarial behavior. In Bitcoin, this computational effort is associated with searching for a special value called the *nonce* that is appended to a known *block header* value such that the hash value of *block header + nonce* lies below a certain *target* value. The lower the value of a target the harder it is to find a satisfactory nonce value and vice-versa. Notice that in spirit this is similar to conducting a pre-image attack on a hash function. In non-technical terms, by the collision-resistant property of hash functions, our best attempt at finding a satisfactory nonce value is by brute-force.

Approach

In this section, we provide a high-level discussion of the reduction of the mining problem to a SAT problem. We credit the work of [3], for providing the guiding principles and ideas needed for the reduction and subsequent implementation.

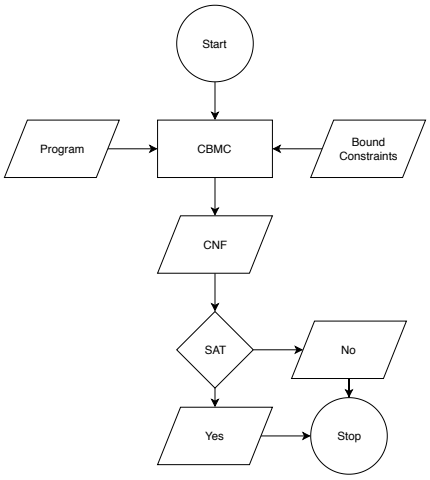


Figure 1: Illustrates the control flow of the program.

Converting Bitcoin Mining to SAT

Section outline: As illustrated by the figure, we start by introducing a program verification tool called the CBMC. We describe how we can use it to convert our mining program into a SAT formula. This is followed by a discussion of our mining program with a focus on the hash function-SHA256 that we use for the process. We delve into the specific rules of the CBMC conversion process that are relevant to the mining program. Lastly we discuss how we employ the SAT solver and how we interpret the results of such a SAT solver as a result to the mining process.

a. C-Bounded Model Checker

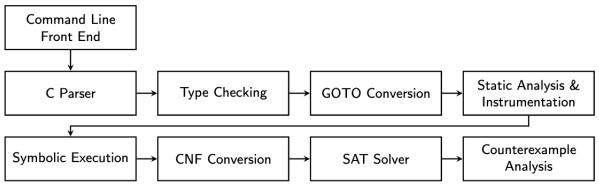


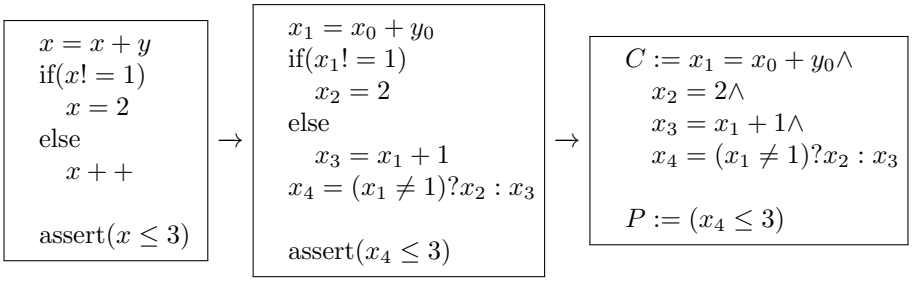
Figure 2: CBMC architecture

A bound model checker reduces the questions about program control flow to constraints that can be solved using SAT solvers. We use C-Bounded Model Checker(CBMC) [2] that converts a C-program into a SAT formula. As an example, consider a multi-threaded application where we want the favorable

property of avoiding deadlock. It attempts to find a counterexample/execution path that violates the deadlock. More generally, given an invariant property of the program the CBMC attempts to find an execution path that leads to a violation of the invariant. The CBMC performs the steps explained below.

Front end. The command-line front-end takes in user parameters and uses an off-the-shelf C preprocessor (gcc -E) and builds a parse tree for the input program. *Type checking* is also performed and a table containing all the symbols are populated. If any inconsistencies are found, CBMC aborts.

Intermediate Representation. In this stage, all non-linear control flows such as branching statements, loops and jumps are converted to *guarded goto* statements (GOTO program). After this step CBMC adds a new initialization function that assigns values to all global variables and then calls the main entry function. Static analysis is done to resolve all possible pointer references. Assertions are added to prevent memory leaks.



Middle end. CBMS performs symbolic execution by unwinding loops and translating the GOTO statements to *static single assignment* (SSA) form. By the end of this process a program is converted to equations with different variable names to avoid multiple assignments and within guarded statements. These guarded statements determine if a certain assignment is done during actual execution.

Back end. The equations from the middle end are translated to a CNF formula. The SAT solver finds a path violating at least one assertion and translates the sequence of assignments that lead to the violation into counter-examples.

```
for (n = 0; n < 8; n++)
{
    *hash = initial_message_digest[n];
    hash++;
}
// becomes
n = 0;
while(n<8){
    *hash = initial_message_digest[n];
    hash++;
    n++;
}
// becomes
```

```
n = 0;
if(n<7){
    *hash = initial_message_digest[n];
    hash++;
    while(n<8){
        *hash = initial_message_digest[n];
        hash++;
    }
    n++;
}
// so on ... and finally
n = 0;
if(n<1){
    *hash = initial_message_digest[n];
    hash++;
    n++;
    if(n<2){
        .
        .
        assert(n==8)
    }
    n++;
}
```

Listing 1: Loop Unwinding

We adapt this approach to solve for the nonce in the mining problem. We give the model checker the hashing algorithm and other inputs that are required and leave the nonce as a free variable, with assumptions around its range and assertions around its target. Doing so, when the model checker finds a nonce which fails the assertion i.e. finds a nonce less than a certain target it returns the nonce as a candidate which violated the assertion. Thereby solving the PoW puzzle. The tool is called a bounded model checker due to a bound on the number of loops within the program. Lastly, we point out that we bound the range of the nonce values to accelerate the search because without it we would require significantly high hash rate. Current hash rate of the bitcoin network is 55,000,000 tera hashes per second. [1]

b. Mining program

Bitcoin mining, involves a the following steps. 1. Transactions that are broadcasted by the peers are collected in a mempool, a buffer of transactions waiting to be processed. 2. The miner chooses a set of transactions from the mempool, optimized to miximize the miners profit. 3. The selected set of transaction are grouped in a merkle tree. The merkle merkle root is computed. 4. The merkle root along with the protocol version number, previous block's hash, current time, network target and the nonce are hashed to compute the hash of the current block. These values make up the block header. Of all the values in the block header the nonce is the only random value. The miner needs to find a nonce such that the chosen nonce when hashed (twice) along with the other header values results in a hash that has to satisfy the target constraint.

Modelling all the steps above is not necessary. It is sufficient to only model the last step: finding a nonce given other header values that also satisfies the constraints imposed by the target value. We limit out implementation to this smaller scope of the mining process.

```
for nonce in nonce_range:
    if (hash(block+nonce) < target):
        return nonce
```

Hash function: SHA256 As indicated by the pseduo-code above, an important part of the implementation is the hash function. We will identify various kinds of operations involved in the SHA256 algorithm and indicate how the CBMC algorithms converts each of them into a SAT formula. The entire implementation can be found in the appendix. As described in the previous section on CBMC we apply the principles to various parts of the code.

Loops: C Bounded model checker as indicated by the name requires all the loops to be bounded. Each loop in the code is unwound as a series of *if* conditionals.

```
for (n = 0; n < 8; n++)
{
    *hash = initial_message_digest[n];
    hash++;
}
// becomes
n = 0;
while(n<8){
    *hash = initial_message_digest[n];
    hash++;
    n++;
}
// becomes
n = 0;
if(n<7){
    *hash = initial_message_digest[n];
    hash++;
    while(n<8){
        *hash = initial_message_digest[n];
        hash++;
    }
    n++;
}
// so on ... and finally
n = 0;
if(n<1){
    *hash = initial_message_digest[n];
    hash++;
    n++;
    if(n<2){
        .
        .
        assert(n==8)
    }
    n++;
```

```
}
```

Listing 2: Loop Unwinding

c. Role of SAT Solver

After the model checker converts any given program with all its free variable, constraints and assertions are converted to a SAT and the model checkers uses a sat solver to check for possible violations of the assertions made. If it finds any, it returns the values for which the assertion failed. Below, we look at certain code snippets and explain them in the context of the SAT solver.

a We initialize the nonce as a non-deterministic variable in the program. This eliminates the need for a loop over all possible range values. We ensure that the nonce variable is the *only* free variable in the program. Thus, we can check if the invariant is violated by trying different nonce values. The range of allowed nonce values determines the running time of the SAT solver. **Takeaway:** By writing the mining step in terms of a non-deterministic value we translate the complexity of finding a nonce value to the SAT solver. If we wrote it as a while loop over a deterministic variable then the mining complexity would remain within the C program.

```
1
2 // non-deterministically select nonce
3 unsigned int *u_nonce = nondet_uint();
4 __CPROVER_assume(*u_nonce > nonce - 10 && *u_nonce < nonce + 10);
```

Listing 3: Assumptions for nonce

b The $CPROVER-assume$ step allows us to make assumptions about the output of the function which is the hash value. In particular, we ask that the first few blocks of hash have the value 0 to match our required target value. This concept is hard to internalize when viewed as a linear program. However, notice that this program is converted into a static SAT formula where a satisfying assignment implies a satisfactory nonce value. Thus, these assumptions about the output are static constraints that direct us to our specified nonce value.

```
1 __CPROVER_assume(
2     (unsigned char)(hash[7] & 0xff) == 0x00 &&
3     (unsigned char)((hash[7] >> 8) & 0xff) == 0x00 &&
4     (unsigned char)((hash[7] >> 16) & 0xff) == 0x00
5 );
```

Listing 4: Assumptions for hash

c Based on the nonce value, we can determine the index of the first non-zero block in a satisfactory target value. In the example below, we check if the 4th byte(after right shift of 24 bits) is non-zero to determine if the nonce matches the target. The job of the CBMC is to find a way to negate the assertion. That is, to find a way to make the flag = 0. This would imply a nonce whose fourth byte is 0 which meets our requirements of a valid nonce.

```
int flag = 0;
if ((unsigned char)((hash[7] >> 24) & 0xff) != 0x00)
    flag = 1;
assert(flag == 1);
```

Listing 5: Assertions

Implementation

Overview

Our tools consists of the following:

- Mining algorithm in C. 212 lines of code. It computes SHA256 hash of the given block.
- A json blob to input block header data.
- A python script. 80 lince of code. It takes an instance of the block and outputs a mining algorithm with constraints specific to that instance.
- A Makefile that abstracts all the running instructions.

Creating an instance for CBMC

The CMBC takes only one instance of the problem and finds a nonce for that single instance. But if we weer to use to tool to calculate the nonce for any random block, we would need the CMBC to solve more than just one instance. To achieve this we use a python script that takes a json blob containing the header values and a C file which contains the template of the mining algorithm and outputs a C file for the given instance of the block header values. Thus making is a generic solution for more than just one instance of the block header.

```
{
    "version": "01000000",
    "prev": "81
    cd02ab7e569e8bcd9317e2fe99f2de44d49ab2b8851ba4a308000000000000"
    ,
    "merkle": "
    e320b6c2fffc8d750423db8b1eb942ae710e951ed797f7affc8892b0f1fc122b
    ",
    "time": "c7f5d74d",
    "target": "f2b9441a"
}
```

Listing 6: A block header

Optimization

Nonce is a 4 byte value. Allowing the SAT solver to search through the entire integer space for a satisfying nonce value takes significantly longer time. So we limit it to a smaller range, letting us verify a proof-of-concept of the proposed idea. We show some experimental performance evaluation of the tools in the next section.

Evaluation

CBMC uses the SMT2 solver by default. Using this solver we evaluate the time it takes for the SAT solver to converge on a valid nonce.

Nonce Range	Time
10	8s
100	3hrs
1000	did not terminate

CBMC also lets us switch out the SAT solver backend for a different SAT solver. We evaluate the performance with all the available backends for a fixed nonce range of 10.

SAT Solver	Time
SMT2	8s
Boolector	12s
CVC4	13s
Z3	18s

In our evaluations we use the SAT solvers with the default options and do not add any custom parameters for tuning. These values can vary by tuning the SAT solver parameters.

Conclusion

We have been able to successfully demonstrate the reduction of the Bitcoin mining process to a SAT problem which can be solved using SAT solver, albeit with better performance.

References

[1] Bitcoin hash rate. https://www.blockchain.com/en/charts/hash-rate.

[2] CBMC – c bounded model checker. http://www.kroening.com/papers/tacas2014-cbmc.pdf.

[3] SAT solving – an alternative to bitcoin mining. https://jheusser.github.io/2013/02/03/satcoin.html.

Appendix

```
// SHA256
// ref: https://github.com/okdshin/PicoSHA2

unsigned int initial_message_digest[8] = {
    0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
    0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19};

unsigned int add_constant[64] = {
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
    0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
    0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
    0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
    0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
    0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
    0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
    0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
    0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2};

// init sha256
void sha_inithash(unsigned int *hash)
{
    int n;

    for (n = 0; n < 8; n++)
    {
        *hash = initial_message_digest[n];
        hash++;
    }
}

// after init, use this to process the sha256 of the chunk
void sha_processchunk(unsigned int *hash, unsigned int *chunk)
{
    unsigned int w[64], s0, s1;
    unsigned int a, b, c, d, e, f, g, h;
    unsigned int t1, t2, maj, ch, S0, S1;
    int n;

    // Read in chunk. When these 32bit words were read, they should
     have been taken as big endian.
    for (n = 0; n < 16; n++)
        w[n] = *(chunk + n);

    // Extend the sixteen 32-bit words into sixty-four 32-bit words
    :
    for (n = 16; n < 64; n++)
    {
        s0 = (w[n - 15] >> 7 | w[n - 15] << (32 - 7)) ^ (w[n - 15]
```

```
    >> 18 | w[n - 15] << (32 - 18)) ^ (w[n - 15] >> 3);
54          s1 = (w[n - 2] >> 17 | w[n - 2] << (32 - 17)) ^ (w[n - 2]
    >> 19 | w[n - 2] << (32 - 19)) ^ (w[n - 2] >> 10);
55          w[n] = w[n - 16] + s0 + w[n - 7] + s1;
56      }
57
58      // Initialize hash value for this chunk:
59      a = *(hash + 0);
60      b = *(hash + 1);
61      c = *(hash + 2);
62      d = *(hash + 3);
63      e = *(hash + 4);
64      f = *(hash + 5);
65      g = *(hash + 6);
66      h = *(hash + 7);
67
68      // Main loop:
69      for (n = 0; n < 64; n++)
70      {
71          S0 = (a >> 2 | a << (32 - 2)) ^ (a >> 13 | a << (32 - 13))
    ^ (a >> 22 | a << (32 - 22));
72          maj = (a & b) ^ (a & c) ^ (b & c);
73          t2 = S0 + maj;
74          S1 = (e >> 6 | e << (32 - 6)) ^ (e >> 11 | e << (32 - 11))
    ^ (e >> 25 | e << (32 - 25));
75          ch = (e & f) ^ ((~e) & g);
76          t1 = h + S1 + ch + add_constant[n] + w[n];
77
78          h = g;
79          g = f;
80          f = e;
81          e = d + t1;
82          d = c;
83          c = b;
84          b = a;
85          a = t1 + t2;
86      }
87
88      // Add this chunk's hash to result so far:
89      *(hash + 0) += a;
90      *(hash + 1) += b;
91      *(hash + 2) += c;
92      *(hash + 3) += d;
93      *(hash + 4) += e;
94      *(hash + 5) += f;
95      *(hash + 6) += g;
96      *(hash + 7) += h;
97  }
```

Listing 7: SHA256