BITCOIN MINING IN A SAT FRAMEWORK

Jonathan Heusser

@jonathanheusser

DISCLAIMER

JUST TO BE CLEAR..

This is research! Not saying ASICs suck

I am not a cryptographer, nor SAT solver guy

WTF

REALISED PHD RESEARCH CAN MINE BITCOINS

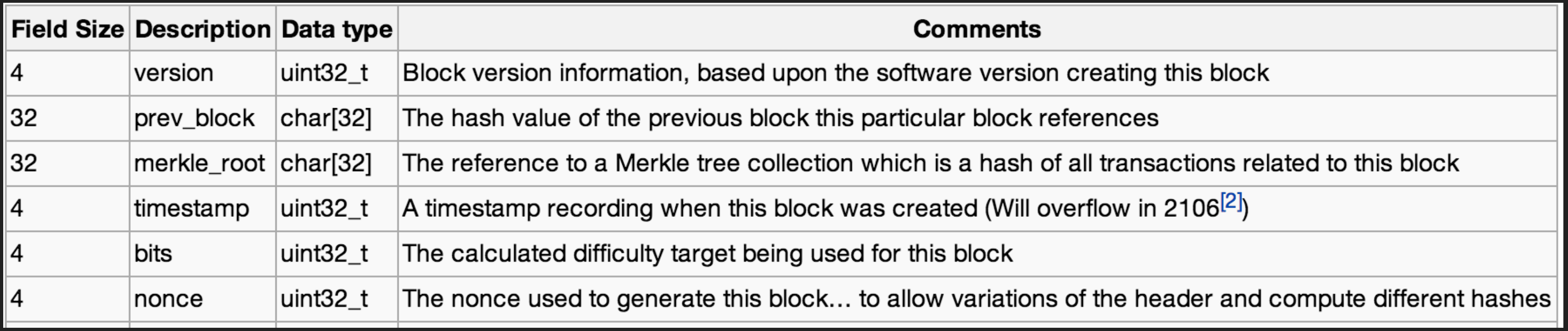
Phd in static analysis + information theory

Quantifying information leakage in programs

Same techniques can be used for mining without brute force!

REVIEW

BLOCK HEADER



Field Size	Description	Data type	Comments
4	version	uint32_t	Block version information, based upon the software version creating this block
32	prev_block	char[32]	The hash value of the previous block this particular block references
32	merkle_root	char[32]	The reference to a Merkle tree collection which is a hash of all transactions related to this block
4	timestamp	uint32_t	A timestamp recording when this block was created (Will overflow in 2106[2])
4	bits	uint32_t	The calculated difficulty target being used for this block
4	nonce	uint32_t	The nonce used to generate this block… to allow variations of the header and compute different hashes

MINING CORE

GETBLOCKTEMPLATE

```
template = getblocktemplate()
while extranonce < MAX:
   block_header = create(template, extranonce)

   while nonce < MAX:
      if f(block_header) < target:
         return 'Found valid block'
      nonce++
   extranonce++
```

nonce and extranonce pointers into block_header

MINING CORE

MINERS FOCUS ON BRUTEFORCE

```
template = getblocktemplate()
while extranonce < MAX:
   block_header = create(template, extranonce)

   while nonce < MAX:
      if sha2(sha2(block_header)) < target: // f(x
         return 'Found valid block'
      nonce++
   extranonce++
```

f is considered a blackbox, not part of algorithm

brute force, because no method or logic involved

no connection between f and nonce

PROPERTY

AVALANCHE EFFECT

Good hash: 1 bit flipped in input, a lot of bits touched in output

Observing the output of a hash function tells you nothing about input

Output uniformly distributed no matter what input distribution

---

If that was not the case: search possible by playing with nonce

Make f run as fast as possible!

Pools, GPU, FPGA, ASIC

IN THIS TALK

Connect f and nonce!

Using tools from program verification: model checker and SAT solver

IN THIS TALK

Connect f and nonce! No brute force

Using tools from program verification: model checker and SAT solver

---

Build declarative specification for mining

Model specification using model checking

Solve for nonce using SAT solver

DECLARATIVE SPECIFICATION (VS IMPERATIVE ALGO)

```
nonce = * // don't care the actual value! Any valu
hash = sha2(sha2(block_header))
assume(hash < target) // constraint
```

Specification for set of valid mining solutions

Here, f and nonce connected through assumption and global constraint

How to encode and solve?

MODEL CHECKING

FORMAL VERIFICATION USING MODEL CHECKING

Extremely successful in practice but not well known (Turing Award)

CPU designs, avionics, medical apps only safe due to verification

---

Given system, check exhaustively properties of that system

Provide counter example to violation of property

Example property: absence of dead locks, floating point errors, etc

VERIFICATION OF PROGRAMS IS HARD

State explosion: trivial program has infeasible number of states

Abstraction or restriction of power necessary

Bounded model checker is only a bug hunting tool. Bounding loops

VERIFICATION OF PROGRAMS IS HARD

State explosion: trivial program has infeasible number of states

Abstraction or restriction of power necessary

Bounded model checker is only a bug hunting tool. Bounding loops

---

CBMC bounded model checker translates C to logic and hunts for bugs

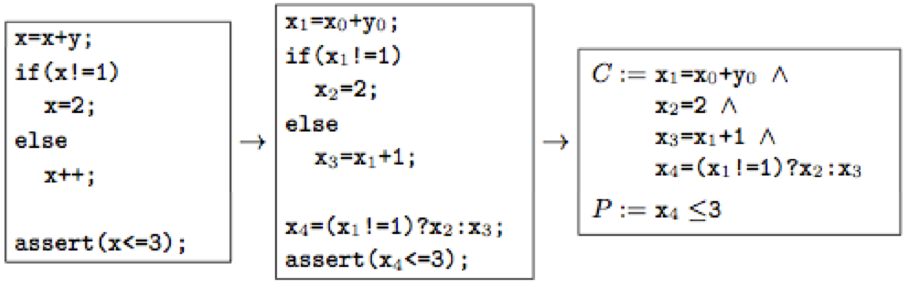
bug means specification violation

C TO PROPOSITIONAL LOGIC

Bitvector variables, unrolled loops, SSA form, ...

Semantics mostly preserved

Program is one global constraint



```
x=x+y;
if(x!=1)
   x=2;
else
   x++;


assert(x<=3);
```

→

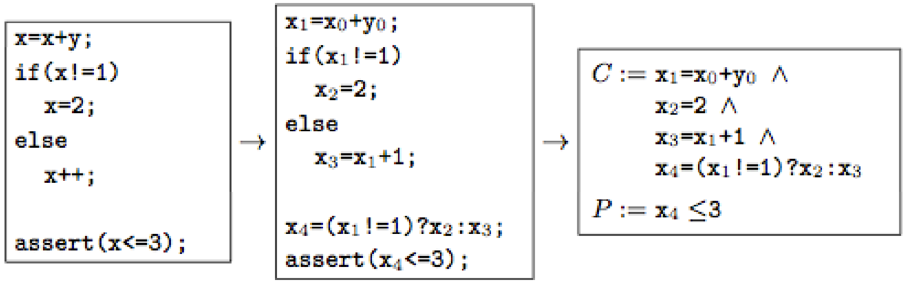
```
x_1=x_0+y_0;
if(x_1!=1)
   x_2=2;
else
   x_3=x_1+1;


x_4=(x_1!=1)?x_2:x_3;
assert(x_4<=3);
```

→

$$\begin{aligned} C :=\ & \mathtt{x_1{=}x_0{+}y_0} \ \land \\ & \mathtt{x_2{=}2} \ \land \\ & \mathtt{x_3{=}x_1{+}1} \ \land \\ & \mathtt{x_4{=}(x_1\,!{=}1)?x_2{:}x_3} \end{aligned}$$

$$P := \mathtt{x_4} \leq 3$$

PROPERTY CHECKING



```
x=x+y;
if(x!=1)
  x=2;
else
  x++;

assert(x<=3);
```

→

```
x1=x0+y0;
if(x1!=1)
  x2=2;
else
  x3=x1+1;

x4=(x1!=1)?x2:x3;
assert(x4<=3);
```

→

$$
\begin{aligned}
C &:= x_1 = x_0 + y_0 \;\wedge \\
&\quad x_2 = 2 \;\wedge \\
&\quad x_3 = x_1 + 1 \;\wedge \\
&\quad x_4 = (x_1 != 1) ? x_2 : x_3 \\
P &:= x_4 \leq 3
\end{aligned}
$$

$$C \wedge \neg P$$

Passed to decision procedure. Only satisfiable IFF property P violated

Counterexample: execution path to violation of P

DECISION PROCEDURE: SATISFIABILITY SOLVER

Decide whether logic formula has a solution (is satisfiable)

Very active and competitive research area

Solvers based on Davis–Putnam–Logemann–Loveland (DPLL) algorithm

Extremely efficient: 100k's vars, millions of clauses

CONJUNCTIVE NORMAL FORM (CNF)

Formula in CNF: 'ands of ors'

$$(\neg a \lor b) \land (\neg a \lor c) \land (\neg b \lor \neg c \lor d)$$

For each clause, at least one literal true

All clauses true in order to be SAT

DPLL ALGORITHM

Depth-first search by picking literals

Propagate decision

Backtrack on conflict

Lots of variations and heuristics

SAT AND CRYPTOGRAPHY

Many papers on using SAT solvers for attacking ciphers

Represent cipher as equations, solve using SAT

Special solvers with XOR, Gauss elimination, variable activity support, ..

Cryptominisat (Mate Soos)

ENCODE SPECIFICATION USING CBMC

Translate specification into C code

Annotate with CBMC specific assumptions and assertions

SAT-BASED MINING

ENCODE SPECIFICATION USING CBMC

```
nonce = nondet_int()
hash = sha2(sha2(block_header))
assume(hash[0] == 0 && hash[1] == 0 && ..) // assu
assert(hash > target)                      // prop
```

Nonce is a non-deterministic value

Known structure of valid hash: leading zeros are assumed

Assertion that valid nonce does not exist

SAT-BASED FRAMEWORK

```
void satcoin(unsigned int *block) {
   unsigned int *nonce = block+N;
   *nonce = nondet_int();

   // 'sha' is a standard SHA-256 implementation
   hash = sha(sha(block));

   // assume leading zeros
   assume(hash[0] == 0x00 && ...);

   // encode a state where byte M of hash is bigge
```

Demo Time

COMPARISON

SAT VS BRUTEFORCE

Clearly, brute force much faster. Only direction is making f faster though

---

Encode richer specification: leading zeros, tricks in SHA2, set individual bits in nonce, ...

Specialised SHA2 encoding: Vegard Nossum, sha256-sat-bitcoin

Take advantage of SAT solvers: learnt clauses, variable activity, cryptominisat, portfolio solvers

COMPARISON

INCREASING DIFFICULTY

Increasing difficulty results in more leading 0 in hash

Conceptually restricts search space

Does this lead to more efficient SAT solving?

REFERENCES

SOME RELEVANT PAPERS

SAT Solving - An alternative to brute force bitcoin mining

SAT-based preimage attacks on SHA-1

The Unreasonable Fundamental Incertitudes Behind Bitcoin Mining

Algebraic Fault Attack on the SHA-256 Compression Function

THANK YOU