TDK Thesis

Tóth Zoltán

University of Szeged
Faculty of Science and Informatics
Department of Software Engineering

Analyzing the potential of SAT solvers for alternative Bitcoin mining

Tóth Zoltán
Computer Science M.Sc II. year

Hamza Baniata Ph.D.
Research Assistant

Attila Kertesz Ph.D.
Associate Professor

Szeged
2024

Összefoglaló

Mióta a névtelen személy vagy csoport, a Satoshi Nakamoto álnevet használva elkészítette a Bitcoint, azóta a fizetőeszközeinkről alkotott képünk örökre megváltozott. A Bitcoin létrehozásával megjelent az első digitális, decentralizált valuta, mely a bankok fizetési rendszereivel szállt versenybe. A Bitcoin megalkotásában fontos szerepet játszott a Proof of Work (POW) nevű konszenzus algoritmus.

Az új Bitcoinokat bányászattal lehet előállítani. Jelenleg a legjobb és legszélesebb körben használt algoritmus a Bitcoin bányászathoz a brute-force. Az algoritmus során a Bitcoin bányászok egyesével növelik a nonce értékét a blokk fejlécében, remélve, hogy ezzel egy megfelelő hash-t kapnak. A probléma ezzel a módszerrel az, hogy a hash-ek megtalálása véletlenszerű, minden lehetséges érték ki van próbálva.

A dolgozatomban megvizsgálok egy módszert, amit Jonathan Heusser tett közzé 2013-ban. Ez egy újszerű módszer, amely SAT megoldókat használ a bányászathoz. Feltehetően Heusser volt az első aki ezzel kísérletezett. A bányászathoz használt algoritmus C-ben van megírva, és a CBMC nevű programmal lehet belőle CNF-et készíteni, amelyet különböző SAT megoldókkal lehet megoldani. Az ő módszerének egyik nagy előnye, hogy az algoritmusban megadhatjuk egy elfogadható hash struktúráját, vagyis, hogy mennyi nulla szerepeljen a hash-ben, és ezzel csökkenthetjük a keresési teret a SAT megoldók számára, legalábbis Heusser állítása szerint. Továbbá, Heusser azt is állítja, hogy emiatt az ő módszere hatékonyabbá válik ahogy növekszik a Bitcoin bányászat nehézsége (minél több nulla kell a hash elején).

Heusser eredményeit reprodukáltam dolgozatomban és megvizsgáltam az állításait. Az eredményeim részben hasonlóak voltak, de több teszt is ellentmond Heusser állításainak. Több blokkot is kielemeztem, és sokszor volt, hogy a használt SAT megoldó le sem futott annak ellenére, hogy a Bányászati feltételek azonosak voltak az előző blokkal. Heusser algoritmusában felfedeztem egy hibát, ami cáfolja azt, hogy a keresési teret csökkenteni tudnánk a SAT megoldók számára. Az eredményeim táblázatokban összesítem, és megállapítom a SAT megoldók átlagos szükséges megoldási idejét is az adott bányászati feltételek mellett. Kifejtem, hogy Heusser módszerét miért nem lehet valódi bányászatra használni.

Kulcsszavak: Bitcoin, SAT, Mining, Blockchain, CBMC

Abstract

Since the anonymous person or group, under the name, Satoshi Nakamoto announced Bitcoin, our view of payment currencies changed forever. By creating Bitcoin, the first decentralized digital currency, Satoshi created an alternative payment system to banks which people can use.

New Bitcoins are created by mining. The mining process is a hard mathematical puzzle which must be solved by miners. Bitcoin generates hashes with SHA256, and the task is to find a hash which is smaller than the target, i.e: it has enough leading zeros.

Currently the best, and most used method for Bitcoin mining is brute-force, where miners slightly alter the block header, and hash it each time in the hope of finding a valid hash. The problem with this method is that the mining is totally random, and finding a valid hash is pure luck.

In my thesis, I have examined a method that was proposed by Jonathan Heusser in 2013. He proposed a novel method that uses SAT solving. The main advantage of this method is that we can make assumptions about the structure of a valid hash, and it can limit the search space for SAT solvers, so instead of blindly computing every hash, the SAT solvers will compute only the ones which comply with the mining specification, at least he claims. One interesting property of this method is that as the mining difficulty gets harder, the SAT solving will become more efficient, at least in theory.

I show the method in details, and analyze the claims of Heusser. I also show my own results with the programs and SAT solvers I have used. Based on these, I make a conclusion about the efficiency of the method, both in practice and in theory.

Keywords: Bitcoin, SAT, Mining, Blockchain, CBMC

Contents

1 Introduction **1**

1.1 Blockchain Technology . . . 1

1.2 Bitcoin . . . 1

1.3 Research problem and Thesis objectives . . . 3

1.4 Thesis structure . . . 4

2 Background **5**

2.1 Bitcoin mining . . . 5

2.1.1 GetWork . . . 5

2.1.2 GetBlockTemplate (GBT) . . . 6

2.1.3 Stratum . . . 6

2.1.4 Solo mining . . . 7

2.1.5 Mining pools . . . 7

2.1.6 How to encode Bitcoin's blocks in C . . . 8

2.2 C Bounded Model Checker . . . 9

2.3 Satisfiability Solvers . . . 10

2.4 How CBMC works . . . 12

2.4.1 Example run . . . 12

2.4.2 Assumptions and assertions . . . 13

2.4.3 Generating random numbers . . . 14

2.4.4 Limitation on loops . . . 14

2.5 Related Work . . . 15

3 Analysis of Heusser's method **17**

3.1 How Heusser's SHA256 works . . . 20

3.2 Heusser's SAT solving results . . . 21

3.3 Discussion . . . 24

3.3.1 Inaccuracies in Heusser's main concept . . . 24

3.3.2 SAT based method in practice . . . 24

4 My Results **28**
4.1 My SAT solving results 28
4.2 Automated tests 30

5 Conclusion **32**

Bibliography **33**

Chapter 1

Introduction

1.1 Blockchain Technology

A blockchain is a distributed ledger with growing lists of records (blocks) that are securely linked together via cryptographic hashes. Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data (generally represented as a Merkle tree, where data nodes are represented by leaves). Since each block contains information about the previous block, they effectively form a chain (compare linked list data structure), with each additional block linking to the ones before it. Consequently, blockchain transactions are irreversible in that, once they are recorded, the data in any given block cannot be altered retroactively without altering all subsequent blocks.

Blockchains are typically managed by a peer-to-peer (P2P) computer network for use as a public distributed ledger, where nodes collectively adhere to a consensus algorithm protocol to add and validate new transaction blocks. Although blockchain records are not unalterable, since blockchain forks are possible, blockchains may be considered secure by design and exemplify a distributed computing system with high Byzantine fault tolerance.

The first blockchain was created for Bitcoin to serve as the public distributed ledger for bitcoin cryptocurrency transactions, based on previous work by Stuart Haber, W. Scott Stornetta, and Dave Bayer. The implementation of the blockchain within bitcoin made it the first digital currency to solve the double-spending problem without the need for a trusted authority or central server [1].

1.2 Bitcoin

Bitcoin is a decentralized digital currency, meaning it operates without a central bank or government control. Instead, it relies on a network of computers around the world to verify and record transactions on a public ledger called the blockchain. This ledger is like

a giant, unalterable record of every Bitcoin transaction ever made. Bitcoin is rewarded to blockchain miners for verifying transactions and can be purchased on several exchanges.

In October 2008, a person or group using the false name Satoshi Nakamoto announced to the cryptography mailing list at metzdowd.com: "I've been working on a new electronic cash system that's fully peer-to-peer, with no trusted third party." This now-famous white paper published on Bitcoin.org, entitled "Bitcoin: A Peer-to-Peer Electronic Cash System," would become the Magna Carta for how Bitcoin operates today. It was introduced to the public in 2008 [2]. It has since become the most well-known cryptocurrency in the world. Its popularity has inspired the development of many other cryptocurrencies.

The process of adding transaction records to the blockchain is called "mining." Bitcoin miners use powerful computers to solve a complex mathematical puzzle. The first miner to crack the puzzle for a new block of transactions adds that block to the blockchain. In return, they receive a reward in the form of newly created Bitcoin and also collect transaction fees from the transactions they confirmed. In case of Bitcoin, the mining puzzle is to find a new hash for the next block, which is smaller than the target difficulty (i.e: the hash must contain the required number of leading zeros). Bitcoin uses the SHA256 algorithm to calculate new hashes.

This puzzle-solving process is intentionally difficult and computationally expensive. It serves two purposes. Firstly, it acts as a lottery system controlling the steady release of new Bitcoin into circulation. Secondly, it secures the network by making it very difficult for anyone to manipulate the blockchain or create fake transactions.

Bitcoin mining consumes enormous amount of energy each year. *"The CBECI (Cambridge Bitcoin Electricity Consumption Index) estimated that global electricity usage associated with Bitcoin mining ranged from 67 TWh to 240 TWh in 2023, with a point estimate of 120 TWh. [3] The International Energy Agency estimated the global consumption of electricity during 2023 to have been 27,400 TWh. So, the CBECI estimates put electricity supporting Bitcoin mining in 2023 at about 0.2% to 0.9% of global demand for electricity. Based on those estimates, global electricity use in cryptocurrency mining was about the same as total electricity consumption in Greece or Australia, respectively."* [3]

The number of miners has increased since the release of Bitcoin, and caused the increase in the puzzle solving difficulty. It is ensured by the Bitcoin network, that a block gets mined every 10 minutes on average. The network always adjusts the mining difficulty based on the current global hash rate, which is 608 EH/s at the time of writing [4]. This increase in the number of miners and the global hash rate is the main cause of Bitcoin's high energy consumption.

While anyone can try and become a Bitcoin miner, the energy consumption and specialized hardware needed create high barriers to entry. This has led to the rise of large-

scale mining operations and pools where miners combine their computational capacities to increase the probability of solving more puzzles and, thus, earning higher rewards.

1.3 Research problem and Thesis objectives

As Bitcoin mining is basically hashing repeatedly with SHA256 until we find a valid hash, we need to address pre-image attack possibilities for the algorithm. For any given input, there are 2^{256} possible SHA256 hashes. This is roughly $1.157 \cdot 10^{77}$ different hashes. Looking at the history of Bitcoin's hash rates, the maximum estimate on the total hash rate was about 750 EH/s in 2024 [4]. With the help of this data, we can estimate the number of years needed to perform a pre-image attack using the classical brute-force method ($2^{256} \div 750000000000000000000 \div 60 \div 60 \div 24 \div 365$), which is about $4.89 \cdot 10^{48}$ years. Although computational power of used hardware is increasing every year, pre-image attacks on SHA256 will likely remain unfeasible in our lifetime, using the brute-force method. If someone could make a new method which is significantly faster, that would be very valuable.

If such a method became possible, it would mean that the security assumptions underlying Bitcoin's proof-of-work algorithm are compromised. Attackers could use the new method to mine faster than everyone else. They could also try to execute a 51% attack if they had enough computing power, and execute double spend attacks. Bitcoin's reputation and value would likely fall, as users may lose their trust in the security and reliability of Bitcoin.

To protect Bitcoin's network, SHA256 would need to be replaced, with a new, secure hashing algorithm, but such a modification needs widespread consensus among the users, developers, miners and other stakeholders. This transition to a new hashing algorithm could introduce new risks and challenges. If the new algorithm would generate hashes which are not 256 bits, then Bitcoin's block structure would need to be changed. That would also mean that mining algorithms and protocols needed a change too. Breaking SHA256 could cause security problems in many territories other than Bitcoin.

In section 2.5 I introduce some previous works, which addressed a similar issue to pre-image attacks on SHA256. In my thesis I will try to address this issue myself, with Heusser's SAT based method.

The main objectives of my thesis are:

1. Following the formal insights presented in [5], I will attempt to experimentally confirm whether Heusser's approach is practical.

2. To do so, I will reproduce Heusser's results by using his original PoC code. Then,

I will implement new SAT-based versions aiming to analyse the approach, in its generality.

3. For tested SAT solvers, I will evaluate the average runtime, and I will discuss implementation and deployment challenges and limitations.

1.4 Thesis structure

The remaining part of my thesis is structured as follows: Chapter 2 presents the necessary technical background and discusses related works. In Chapter 3, I analyze Heusser's method and I present the methods used during my research, including previous SAT solvers and my proposed modifications. In this chapter I also discuss the results of running previous and current implementations, and I present the conceptual and design limitations of the mining approach using SAT solvers. Finally, I list my results in Chapter 4 and conclude my thesis in Chapter 5.

Chapter 2

Background

2.1 Bitcoin mining

Miners today has two options to choose from. They can either mine Bitcoin solo, or join a mining pool. Each of them has their own pros and cons. What's common in them, is that they both use Bitcoin's core implementation (bitcoind) under the hood. This way, solo miners and pools can communicate with the Bitcoin network [6].

2.1.1 GetWork

This is the old protocol used by Bitcoin miners. In 2012 it has become outdated, as more miners joined the network. One problem with this protocol is that it doesn't maintain a connection between the miner and the network. If a miner is not able find a valid hash by looping through all nonces, it has to request the network for new hashing work. A pool for example, would have to serve ten thousands of requests each second, if they would still use this protocol. An example hashing work extracted from the response of a getWork request, looks the following:

```
000000020597ba1f0cd423b2a3abb0259a54ee5f783077a4ad45fb6200000218
000000008348d1339e6797e2b15e9a3f2fb7da08768e99f02727e4227e02903e
43a42b31511553101a051f3c0000000000000080000000000000000000000000
0000000000000000000000000000000000000000000000000000000080020000
```

This data can be processed in the following way:

1. Save the first 160 hex
2. Byte-swap every 32-bit chunk of data
3. Hash the result: SHA256(SHA256(result))

The full response from the network includes a few other data in the JSON it returns, other than the hashing work (which is the block's header). One of them is the midstate, which miners can use to further optimize their algorithm. By using the midstate, miners are able to compute only the second SHA256 run.

Another problem with this protocol is that miners can't construct their own blocks. They receive the block header, but they can't decide which transactions they would like to add to the block. In case of mining pools, miners blindly accept their hashing work, and the responsibility to add new transactions to the block is moved to the pool. A corrupt pool might use it's collected hashing power to execute double spend attacks and other similar attacks. Because of the previously mentioned reasons, pools disallow this protocol to use, but solo miners are still able to use this protocol through an older version of bitcoind.

2.1.2 GetBlockTemplate (GBT)

This protocol superceds the old getWork. It was openly developed by the Bitcoin community through the mid 2012s. It solves both problems of the previously mentioned getWork protocol. Miners who use GBT, are connected to the Bitcoin network through a HTTP longpoll connection. They leave this connection open, and this make transmitting data much faster. This highly reduces network load, and makes solo mining possible again, while pool servers benefit from it too. The block creation is done by the miner, and it makes the network more decentralized and safe. GBT was designed to be flexible, and it's prepared for new extensions.

Miners initially have to query a block template once from the network. Then they can build their own merkle tree, deciding which transactions to include. It's sometimes required though, to refresh the block template. It's usually best if the pool is let to decide when to do that, as it's aware of important changes. After a refresh, the miner can start it's work again. GBT interpreters are available for C and python, which makes creation of mining softwares easier.

2.1.3 Stratum

Since 2012, the stratum protocol is the most used protocol among bitcoin miners and pools. The v1 of stratum released near the same time as GBT, eliminated GBT's wider adoption, and people quickly began to use stratum. Stratum provides miners the minimal information they need to construct blocks on their own. Stratum miners can't inspect or add transactions to the block they're currently mining. They alter the coinbase field by adding an extra nonce to it, then hash it and add it to the merkle tree. The merkle root

is recalculated as needed. Stratum was criticized, because the v1 of it was developed as a secret. Under the hood, it uses GBT's features with some improvements. A downside of v1 Stratum is that miners are unable to construct their own blocks (same problem as with getWork). Miners would receive a block template, which they can work on. A newer version of Stratum is v2, which eliminates this problem, and allows miners to construct their own blocks and decide which transactions to include. Stratum V2 simplifies the mining process, increases safety, and encourages a more distributed and decentralized approach to mining. V2 is not widely adopted yet, and most pools operate using Stratum v1, but there are a few pools, which already adopted the new mining protocol.

2.1.4 Solo mining

This form of mining is what's the least profitable nowadays for most of the people. The chances of hitting a block solo, are so low, that if you don't have an enormous amount of hashing power, then you need to wait years, until you find a block. According to SoloChance.com [7] which is the most widely used calculator for solo mining, you'll need about 36 years, if you are solo mining with one Bitmain Antminer S21 Hyd (335 Th/s), which is at the time of writing, one of the most powerful ASIC miners (of course, you might have more miners, which will increase your chances).

Solo miners use bitcoind, along with GBT to get new transactions and create their own blocks. The communication between the miner and the network (bitcoind) occurs with the help of mining softwares. Solo miners can choose from a wide range of different mining softwares to use. These are mostly reputable and easy to setup. Solo miners are even able to write their own mining programs, to be part of the network.

2.1.5 Mining pools

Pooled mining is what's most used nowadays. It is more feasible for the majority of people, than solo mining. Each pool has it's own set of rules which declare how rewards will be shared among participants. There are more than 50 pools available to mine Bitcoin, but the market (and the Bitcoin network itself) is dominated by 5 large pools. They share almost 80% of the total hashrate of the network. While pooled mining is prefered, it still has it's own disadvantages, such as you never get the rewards for the whole block, as they will be splitted and shared among all eligible participants.

The mining pool gets new transactions from bitcoind. Pool miners use their mining softwares, to connect to a pool. Then they request the information needed to construct their block header. Pools have their own target, which is higher in difficulty, than the network's target (i.e: less leading zeros required). This way, pools can decide who is

eligible for a share, and they know who opted enough hashing power.

2.1.6 How to encode Bitcoin's blocks in C

Heusser retrieved the data of the genesis block from Bitcoin.it website [8]. Although he encoded the block in decimal format, he used the values of the raw hex data of the block which can be seen in Figure 1.

A block header is a 80 byte long hexadecimal string in raw data format, and it consists of the following parts:

- 4-byte long Bitcoin version number
- 32-byte previous block hash
- 32-byte long Merkle root
- 4-byte long timestamp of the block
- 4-byte long difficulty target for the block
- 4-byte long nonce used by miners

```
00000000   01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000010   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000020   00 00 00 00 3B A3 ED FD  7A 7B 12 B2 7A C7 2C 3E   ....;£íýz{.²zÇ,>
00000030   67 76 8F 61 7F C8 1B C3  88 8A 51 32 3A 9F B8 AA   gv.a.È.Ã^ŠQ2:Ÿ¸ª
00000040   4B 1E 5E 4A 29 AB 5F 49  FF FF 00 1D 1D AC 2B 7C   K.^J)«_Iÿÿ...¬+|
00000050   01 01 00 00 00 01 00 00  00 00 00 00 00 00 00 00   ................
00000060   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000070   00 00 00 00 00 00 FF FF  FF FF 4D 04 FF FF 00 1D   ......ÿÿÿÿM.ÿÿ..
00000080   01 04 45 54 68 65 20 54  69 6D 65 73 20 30 33 2F   ..EThe Times 03/
00000090   4A 61 6E 2F 32 30 30 39  20 43 68 61 6E 63 65 6C   Jan/2009 Chancel
000000A0   6C 6F 72 20 6F 6E 20 62  72 69 6E 6B 20 6F 66 20   lor on brink of 
000000B0   73 65 63 6F 6E 64 20 62  61 69 6C 6F 75 74 20 66   second bailout f
000000C0   6F 72 20 62 61 6E 6B 73  FF FF FF FF 01 00 F2 05   or banksÿÿÿÿ..ò.
000000D0   2A 01 00 00 00 43 41 04  67 8A FD B0 FE 55 48 27   *....CA.gŠý°þUH'
000000E0   19 67 F1 A6 71 30 B7 10  5C D6 A8 28 E0 39 09 A6   .gñ¦q0·.\Ö¨(à9.¦
000000F0   79 62 E0 EA 1F 61 DE B6  49 F6 BC 3F 4C EF 38 C4   ybàê.aÞ¶Iö¼?Lï8Ä
00000100   F3 55 04 E5 1E C1 12 DE  5C 38 4D F7 BA 0B 8D 57   óU.å.Á.Þ\8M÷º..W
00000110   8A 4C 70 2B 6B F1 1D 5F  AC 00 00 00 00            ŠLp+kñ._¬....
```

Figure 1: Genesis block's raw hex data

In the raw data presented, the first 160 hexadecimal characters make up the genesis block's header. Note, that 1 hexadecimal character is only 4 bits, so 1 byte equals to 2 hexadecimal characters, that's why they are separated two by two in the raw data.

```c
unsigned int genesis_block[20] = {
    0x01000000,
    0x00000000,
    0x00000000,
    0x00000000,
    0x00000000,
    0x00000000,
    0x00000000,
    0x00000000,
    0x00000000,
    0x3ba3edfd,
    0x7a7b12b2,
    0x7ac72c3e,
    0x67768f61,
    0x7fc81bc3,
    0x888a5132,
    0x3a9fb8aa,
    0x4b1e5e4a,
    0x29ab5f49,
    0xffff001d,
    0x1dac2b7c};
```

Figure 2: Genesis block in C

Now that we have all the data we need, we can copy paste the values in C. Because of how Heusser's SHA256 implementation works, which is explained in section 3.1, we need to create an unsigned int array which consists of 20 elements. We need to split the raw data to 20 elements, and paste them into the newly created array. When pasted to C, in case of the genesis block it should look as presented in Figure 2.

To create new blocks, and to encode the latest blocks in the Bitcoin blockchain, we need a source of data. Throughout my thesis I've tried many of them, including btc.com, btcscan.org and blockexplorer.one, but I found Blockchain.com the most suitable as it provides a free API which gives us the raw hexadecimal data of any block. We can use Postman, CURL or any other suitable tool to access the API. I used Postman for my requests. An example request is the following:

https://blockchain.info/rawblock/000000000019d6689c085ae165831e934ff763ae
46a2a6c172b3f1b60a8ce26f?format=hex

This is a GET http request which provides the genesis block's raw data. To access other blocks, change the block hash in the request. Block hashes can be retrieved from the block explorer tool of the website.

2.2 C Bounded Model Checker

C Bounded Model Checker (CBMC) is a Bounded Model Checker for C and C++ programs. It supports older versions of the C language, and C11 too. CBMC has a variant, named JBMC, which is used to analyze Java code. CBMC and JBMC are bug hunting tools,

which are used to detect flaws in programs. These tools can analyze various problems within a code, such as memory flaws, array bounds checks, safe use of pointers, exceptions, manual assertions etc [9]. CBMC is available for most versions of Linux (pre-packaged on Debian, Ubuntu and Fedora), Solaris 11, Windows and MacOS X. CBMC comes with a built in solver for SAT. As of writing these lines, the current state-of-the-art version of CBMC, which is 5.95.1, uses Minisat 2.2.1 with simplifier for it's built in SAT solver. As an alternative, CBMC provides many other built-in SAT and SMT solvers for us to use, including MathSAT, Boolector, Z3 and CVC4. Understanding the internals of CBMC are essential, as the method presented by Heusser heavily depends on the use of this program. It's operation is described in section 2.4.

2.3 Satisfiability Solvers

In the realm of theoretical computer science and mathematical logic, the Boolean Satisfiability Problem (SAT) stands as a cornerstone concept. SAT revolves around the fundamental question: does there exist an assignment of truth values to variables that can satisfy a given Boolean formula? *In other words, it asks whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE [10].* Although deceptively simple in its formulation, SAT holds profound implications for diverse areas such as constraint solving, algorithm verification, artificial intelligence, and computational complexity.

A Boolean formula is constructed from variables (e.g: x, y or z), Boolean operators (AND - $\land$, OR - $\lor$, NOT - $\neg$), and parentheses. An example formula is the following

$$\varphi = x \land (x \lor \neg y) \land (x \lor y) \tag{2.1}$$

This formula is also in CNF form, which stands for Conjuctive Normal Form. CNF is any formula which purely consists of conjunctions of clauses. A clause is a disjunction of literals, and a literal is simply a variable or its negation. The satisfiability of this formula can be easily verified by just looking at it. If we set variable x to *true*, then the formula will be satisfied.

SAT is the first problem that was proven to be NP-complete, by the Cook–Levin theorem. This means that all problems in the complexity class NP are at most as difficult to solve as SAT. Currently there is no known algorithm which could efficiently solve each SAT problem. It is believed by the majority of people that no such algorithm exists, although this belief is not proved mathematically. Resolving the question of whether SAT has a polynomial-time algorithm is equivalent to the P versus NP problem, which is a famous open problem in the theory of computing.

SAT solvers are algorithms designed to determine whether CNF formulas have any assignment of True or False to their variables that renders the entire formula True. Decades of research have produced remarkable advancements in SAT solvers. At the heart of these solvers lie sophisticated techniques like Conflict-Driven Clause Learning (CDCL), backtracking search, intelligent heuristics, and efficient data structures. These developments have unlocked SAT's potential for solving remarkably complex real-world problems. Applications of SAT include:

- Hardware and Software Verification: Ensuring the correctness and security of hardware designs and software systems.
- Planning and Scheduling: Optimizing schedules and resource allocation in logistics, manufacturing, and project management.
- Bioinformatics: Analyzing biological data and modeling gene regulatory networks.

Even though SAT solvers improved significantly over the past decades, still none of them can efficiently solve (i.e: in polynomial time) every SAT instance. It's also intriguing, that even today, it is unknown what makes a SAT instance hard to solve. Some instances which consist of millions of clauses and variables can be solved efficiently, while other, much smaller formulas can not.

The DIMACS CNF format is a widely accepted standard for representing Boolean formulas in a text-based file format. It plays a crucial role in the field of SAT solving. The precise origin of the DIMACS CNF format is unclear, but it emerged within the research community focused on the SAT problem, likely in the late 1980s or early 1990s, associated with the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS). Due to its simplicity and clarity, the DIMACS CNF format has become the de facto standard for representing Boolean formulas used in SAT solvers and benchmarks. Major SAT solver competitions like SAT Competition utilize DIMACS CNF as the primary input format. The DIMACS CNF format's popularity stems from its ease of use, human-readability, and broad support by various SAT solvers. This format facilitates the exchange of SAT problems across different solvers and research groups. Focus on Functionality: It's important to note that the DIMACS CNF format is designed for efficient processing by SAT solvers rather than human interpretation. While relatively simple, understanding the format's structure allows researchers to work with SAT solvers more effectively.

A DIMACS CNF file consists of plain text lines representing the Boolean formula. Lines starting with the character "c" are ignored by the solver and can be used for human-readable comments. The first non-comment line typically starts with the letter "p" followed by "cnf". This line specifies that the file is in CNF format and provides the

number of variables (n) and the number of clauses (m) in the formula. An example header line might be the following:

$$p\ cnf\ 4\ 3 \tag{2.2}$$

This means, that the cnf will have 4 variables and 3 clauses. Each line then represents a clause in the text file. At the end of these lines there's always a 0 character, which separates the clauses. Then a new clause, will start in the next line. A positive literal signifies the variable itself (e.g., "2"). A negative literal is the same but with a minus sign (e.g: "-2"). Whitespace separates literals within a clause. A full example of a DIMACS CNF might be the following:

$$\begin{aligned} &p\ cnf\ 3\ 3 \\ &1\ 0 \\ &-2\ 0 \\ &-1\ 2\ 3\ 0 \end{aligned} \tag{2.3}$$

This represents the following CNF formula:

$$\varphi = x \wedge \neg y \wedge (\neg x \vee y \vee z) \tag{2.4}$$

In conclusion, the DIMACS CNF format serves as a cornerstone for communication and collaboration within the SAT solving community. Its simplicity and efficiency have cemented its place as the standard for representing Boolean formulas used in SAT solvers and benchmarks. Most of the SAT solvers I used, work with this format too.

2.4 How CBMC works

CBMC and JBMC are effectively the same tools, and they almost work the same, except a few differences. I explain, how CBMC works, as this is used in Heusser's method, and my own modified version of his code is written in C too.

2.4.1 Example run

Consider the following program in Listing 2.1.

```
int main() {
    int x = 10;
    assert(x == 5);
    return 0;
}
```

Listing 2.1: Simple C code

This can be easily analyzed with CBMC by running the following command in cmd:

```
cbmc filename.c --trace
```

CBMC will produce the following result:

```
** Results:
testcbmc.c function main
[main.assertion.1] line 6 assertion x == 5: FAILURE

Trace for main.assertion.1:

State 12 file testcbmc.c function main line 5 thread 0
----------------------------------------------------
  x=10 (00000000 00000000 00000000 00001010)

Violated property:
  file testcbmc.c function main line 6 thread 0
  assertion x == 5
  x == 5


** 1 of 1 failed (2 iterations)
VERIFICATION FAILED
```

It analyzed the code, by converting the program to SSA [11], then converting it to CNF, then running the built-in solver. From the trace, we can see which property caused the error. This works the same on more complex programs too.

2.4.2 Assumptions and assertions

These two are the most important concepts to understand, because Heusser cut the search space by giving his program clever assumptions, and used assertions to solve SAT with CBMC, and retrieve the correct nonce. Consider the following program:

```
1 #include <assert.h>
2 int main()
3 { int x = 10;
4     for(int i = 0; i < 20; i++) {
5         x += i;
6         __CPROVER_assume(x > 15);
7         assert (x > 15); }}
```

Listing 2.2: Assumptions and assertions with CBMC

Because of the assumption, CBMC will execute only the paths which satisfy the assumption. That's why the analyzation will not fail in this case, because we assumed that x $>$ 15, so the assertion will only execute on paths where x $>$ 15. Please note, that if we change to assumption to x $>$ 10, CBMC will still verify the program successfully, because if the assumption is not satisified in the for loop, then the whole loop will be skipped. Normally, we would expect CBMC to fail at the assertion, when x is 11, or 12 etc. If we set the assumption to x $>$ 9, then CBMC will fail at the assertion, because initially x is 10, which satisfies the assumption, but the assertion will fail.

In the SAT based miner, assumptions are used to skip paths, where the hash doesn't comply with the target. Assertions are used to raise errors. Heusser does it intentionally. He directs CBMC to catch an assertion error exactly when the program finds a valid hash with the valid nonce. Then, in the counterexample we can see the valid nonce which produced a valid hash.

2.4.3 Generating random numbers

CBMC can generate random numbers, which are also an important part of the SAT based method. It's important that when CBMC generates the numbers, it will try all possible values, if we don't set a limitation on them. We can use assumptions, to limit the range of numbers. Consider the following code:

```
int main() {
    #ifdef CBMC
        int x = nondet_uint(); int y = 10;
        __CPROVER_assume(x > 0 && x < 100);
        y += x;
        assert(y > 10 && y < 110);
    #endif
}
```

Listing 2.3: Random numbers with CBMC

The above program will be verified successfully because the assumption ensures that y after the addition will be in the given range. If we remove the assumption, then the assertion will fail because x can be any random number.

2.4.4 Limitation on loops

Loops cannot be always effectively analyzed. Consider the following program:

```
int binsearch(int x)
{
  int a[16];
  signed low = 0, high = 16;

  while(low < high)
  {
    signed middle = low + ((high - low) >> 1);

    if(a[middle] < x)
      high = middle;
    else if(a[middle] > x)
      low = middle + 1;
```

```
14      else // a[middle]==x
15        return middle;
16    }
17
18    return -1;
19  }
```

Listing 2.4: CBMC's binary search example

CBMC will not be able to determine the upper bound for the while loop, and it won't stop on it's own. This is because the upper bound for the loop in the program is calculated during runtime. In this case, we have to explicitly provide how many times will the loop run at maximum. The –unwind property sets the number of loop unwindings globally, but we can limit just one loop too.

If we tried to analyze a full implementation of SHA256, CBMC would fail, as it is not able to unwind all loops in the algorithm, because of the previously mentioned reasons. It could only analyze it if we set the –unwind property or if we modify the algorithm in a way that cbmc will be able to analyze it.

2.5 Related Work

The current state of literature includes many attempts (e.g. [12]) to find a practical way for executing pre-image attacks on SHA256, among other variants (e.g. [13]). This is because SHA256 is deployed for insuring the security of several real-world applications, including access control, e-Health, e-Voting, and Banking systems. The full algorithm is still generally safe to use nowadays despite the continuous efforts spent to find computational vulnerabilities. Such vulnerabilities might also be found for specific applications, such as ASICBoost [14], resulting in nearly 20% enhancement in the Bitcoin mining application.

In 2013, Jonathan Heusser presented a novel alternative Bitcoin mining approach [15] in which he deployed SAT solvers to solve the mining problem. Specifically, the methods for utilizing SAT solvers were described and some exclusive experiments were discussed. The experimental results in this work claimed two main conclusions as follows:

1. The proposed algorithm gets more efficient with increasing the puzzle difficulty.

2. The search time of the proposed algorithm is not linearly correlated with the nonce range.

3. The algorithm is not brute-force, as we can take advantage of the structure of a valid hash, thus only valid hashes will be computed

The proposed method was tested for mining few pre-mined blocks using several solvers. Furthermore, the number of different trials performed, with different number of assumptions or assertions, before achieving the final presented results, was not declared. Additionally, a modified version of SHA256 was implemented and used in the experiments, as Heusser made it only compatible with Bitcoin's blocks, which is explained in section 3.1.

Other works attempted to reproduce and confirm Heusser's experiments and results. However, this remained an open issue as the full details on the implementation and deployment were not easy to deduce.

Similarly, Trevor Phillips attempted to execute pre-image attacks on SHA256 using SAT solvers [16]. In this work, some methods could indeed solve up to 16 rounds of the full SHA256 algorithm. However, this was only realized as a partial attack as the remaining rounds could not have been broken using the presented method.

Recently, Baniata and Kertesz [5] have formalized Heusser's approach using the Birthday paradox and formally confirmed Heusser's two observations under the condition that the presented mining method is valid. They have also proposed new concepts for benchmarking the security of PoW-based systems, including Critical Difficulty and Critical Difficulty per given portion. However, it was out of the scope of this study to reproduce Heusser's results and experimentally confirm Heusser's conclusions using other blocks or SAT solvers. Thus, this is the open issue that I am aiming to address in my thesis.

Chapter 3

Analysis of Heusser's method

Heusser's proof of concept code can be retrieved from his github repository which linked in his article [15]. The SAT based method is entirely written in C. He also included some CNF files which he benchmarked. Those files are CNFs generated with CBMC. The out_1k_sat.cnf represents the Genesis Bitcoin Block. The out_blk218430_1k_sat.cnf as the name suggests, represents Bitcoin's 218430Th block. The 1k in the file names indicate the nonce range used for these CNFs. The sat at the end means, that the formula is satisfiable, therefore, a valid hash was found. He included a few other verions of these CNFs, with different nonce range, or when the cnf is unsat, which means the formula is unsatisfiable. All CNF files are in DIMACS format, as CBMC supports this format to generate CNF files. In section 2.3 I explain the Dimacs CNF format.

Heusser's CNF files differ from the ones I'm able to generate, by using the original proof of concept code. When I generate the CNF of the genesis block with 1k nonce range, my cnf has 141360 variables and 645987 clauses. The uploaded cnf file has 253808 variables and 867382 clauses. This is probably because I'm using a newer version of CBMC, while he used a version from 2013. Unfortunately, versions of CBMC from 2013 are not available, as the oldest version on Github is 5.6, from 2016. CBMC has improved a lot since 2013, and it has a detailed documentation about the history and improvements of it throughout the years [9].

Most of Heusser's code is the implementation of SHA256, along with the CBMC specific directives which help with the SAT solving. There's an unused method in the PoC code, but it doesn't affect the performance of CBMC and the Solvers.

The genesis block in his code is the actual genesis block of Bitcoin. I explain in section 2.1.6, how to encode Bitcoin's blocks in C, to be able to work with them using the SAT based method. Heusser implemented SHA256 in a way that it's friendly to CBMC. This means, he modified on a few parts of the original method. The SHA256 implementation used by him is explained in section 3.1.

An important part of his method is when he generates a non-deterministic value for the nonce.

```
#ifdef CBMC
    // set the nonce to a non-deterministic value
    *u_nonce = nondet_uint();

#ifdef SATCNF
    // make sure the valid nonce is in the range
    unsigned nonce_start = 497822588 - SATCNF;
    /* other code..... */
```

Listing 3.1: Heusser's non-deterministic nonce

He claims, that this variable is the only free variable in the generated model and *"in our case it is a way of moving the search from outside of the actual computation (a brute force loop) to be part of the SAT solver search"*.

The random nonce is indeed needed because, this way CBMC is let to play with this variable instead of being part of the normal computation flow.

But, any variable we add to our program, can lead to an increase in the number of variables in the CNF, even if it's not random or if it's not an essential part of the code and could safely be deleted.

The problem with Heusser's method is that it's very inconsistent. Initially his code forces CBMC to find the correct nonce by limiting the nondet_uint() function to generate that exact nonce value which is good for the genesis block. This is done by an assumption, and looks the following:

```
/* ========================== GENESIS BLOCK======================== */
    //__CPROVER_assume(*u_nonce > 0 && *u_nonce < 10);
    __CPROVER_assume(*u_nonce > 497822587 && *u_nonce < 497822589);
    // 1 nonces only
```

Listing 3.2: Heusser's assumption to find the valid nonce

If we run CBMC with this assumption, it finds the valid nonce, because we forced it to try only this one. If we use 1k for the nonce range, then CBMC will run too long. It doesn't finish under 10 minutes. Even if we set the nonce range to a very small number like 10, CBMC will still not be able to finish under 10 minutes. I use 10 minutes as the threshold, because Bitcoin miners can create a new block every 10 minutes on average.

Heusser doesn't use the best possible assumptions for the genesis block's hash. The genesis block's hash target is 8 leading zeros.

```
00000000ffff0000000000000000000000000000000000000000000000000000
```

We should assume all of them, to get the best results. Assuming the leading zeros, cuts down all search paths where the hash doesn't have the number of leading zeros as we assumed, at least that's what Heusser claimed. He used the following assumption:

```
__CPROVER_assume(
    (unsigned char)(state[7] & 0xff) == 0x00 &&
    (unsigned char)((state[7] >> 8) & 0xff) == 0x00 &&
    (unsigned char)((state[7] >> 16) & 0xff) == 0x00);
```

Listing 3.3: Heusser's assumption on the hash

One line of code assumes 2 leading zeros. We need to assume the zeros from the trail of the hash (that's why we use the last element of the state variable, which stores the hash), because in Bitcoin the hash is displayed in big-endian, so every byte is reversed. It's important to note, that 1 byte is 2 hexadecimal digits. If this is the hexadecimal representation of the nonce: 0x26b7c92d, then Bitcoin will display it in the following way: 0x2dc9b726. When we look at any hash in Bitcoin, like the genesis block's hash:

`0x000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f`

note, that all of them are displayed in big-endian order.

Back to Heusser's assumption which is displayed above, we can see that he only assumed 6 leading zeros, which is not optimized, as it keeps the possibility that the internal SAT solving will execute paths where the leading zeros are not sufficient for the genesis block.

Although, in practice there's a really low chance that we could find another nonce valid, because it generated a hash that has 6 or 7 leading zeros which comply with Heusser's assumption, we should always strive to assume as many leading zeros as possible.

The next problem is Heusser's assertion, which is reversed in logic. *"You specify the invariant of your system, which should always hold, and the model checker will try to find an execution where this invariant is violated (i.e. where there was a satisfiable solution). That is why the P above is negated in the formula. Thus, the invariant, our P, is set to No valid nonce exists"*. In his implementation it looks the following:

```
    int flag = 0;
    // if((unsigned char)((state[6]) & 0xff) != 0x00) {
    if ((unsigned char)((state[7] >> 24) & 0xff) != 0x00)
    {
        flag = 1;
    }
    // counterexample to this will contain an additional leading 0 in
   the hash which makes it below target
    assert(flag == 1);
```

Listing 3.4: Simplified assertion

This is what it encodes: If the hash doesn't contain the zeros at the specified position, then flag is set to 1 and the assertion will pass. If the hash has the leading zeros at the specified position, it means that we found a valid hash. In this case, the if statement will be false, and the flag variable will remain 0, so the assertion will fail.

For CBMC, this reversed logic is not needed. It doesn't care about how we encoded the state which leads to the counterexample (assertion fail). We can just write assert(0); and it will generate the counterexample. The following assertion has the same effect as Heusser's:

```
if ((unsigned char)((state[7] >> 24) & 0xff) == 0x00)
{
    assert(0);
}
```

Listing 3.5: Simplified assertion

This can be translated as the following: If the hash is valid, then generate a counterexample.

3.1 How Heusser's SHA256 works

The SHA256 algorithm implemented by Heusser is a properly working implementation, but it has some modifications compared to the original algorithm. The implementation of Heusser is friendly to CBMC, as it doesn't involve any strlen or similar methods. This is important because in full SHA256, we have loops, which have dynamic upper bound, based on parameters, and these loops can't be automatically unwinded by CBMC.

Heusser's implementation only works on Bitcoin's blocks as inputs. But this is perfectly fine for a bitcoin miner, because the block header what gets hashed, is always the same size (80 bytes). In the first round of SHA256, he processes 2 chunks, because the 80 bytes message, which is 640 bits, is too large for one chunk. One chunk in SHA256 is 512 bits. The first chunk simply gets loaded with the first 512 bits of the block header. The second chunk will have 128 + 1 + 319 + 64 bits where:

- The first 128 bits are the remaining part of the original message
- A single 1 bit gets appended to the message
- 319 zeros will be used as padding
- This is the last part of the algorithm. The last 64 bits are always the length of the original message, encoded as a 64 bit big-endian integer.

Heusser processes both chunks correctly, and preserves the state of SHA256 between the two chunks procession. After the first round, he processes the resulting hash in the second

round. When the second round of SHA256 finishes, we have the final result of hashing our block. My own tests on hundreds of different blocks validated the correctness of Heusser's SHA256 implementation.

3.2 Heusser's SAT solving results

Heusser used many different SAT solvers. We can generate CNF formulas from our code with CBMC, by using the following parameters:

```
--dimacs --outfile filename.cnf
```

With dimacs we can specifiy the format of the CNF file, and with the other parameter we can set the name of the CNF file.

Although, the generated CNF can be solved more efficiently with external solvers like Manysat or Zchaff, the problem with them is we can't retrieve the nonce from their output. SAT solvers on their own are not tools designed specifically for bug hunting, and they can only output whether the formula is satisfiable or not. CBMC, and some other tools, are built on top of SAT solvers, to provide counterexamples which lead to bugs. In our case, it will lead us to a correct nonce. Heusser only mentioned he retrieved the correct nonce, when he used CBMC's built-in solver to test the genesis block.

But external solvers can still be used, and we know if the valid nonce is found by them, because:

- If the instance is SAT (satisfiable), it means that an assertion error occured in the program, and we encode that it only occurs, if the hash is valid, which indicates that the nonce must be valid too.
- We assume the nonce range, in a way, that it includes the correct one

For real Bitcoin mining this is not practical as for a new block in Bitcoin, we don't know the nonce, so the nonce range can't be pre-assumed this simply. In section 3.3.2 I discuss the method for real Bitcoin mining.

The table 3 shows Heusser's results on the genesis block, when the instance is unsatisfiable. He claimed the following: *"In UNSAT, every solver has to perform the same amount of basic work of trying out 1000 nonce values"*. When the instance is UNSAT, it means that he encoded an example, where he used a nonce range which doesn't include the correct nonce for the genesis block. This way, the solver is forced to try all values, and when it's done, it will output UNSAT as the result.

The table 4 shows Heusser's results on the genesis block, when the instance is SAT.

He claims the following: *"First, it is unsurprising that the SAT timings are generally lower than UNSAT as the solvers do not have to try all possible nonce values"*. Although

SAT Solver	Time to UNSAT
Cryptominisat 2.9.2	49s
Cbmc 4.0 Minisat	1m18s
Rsat 2.01	2m08s
restartsat	1m35s
Minisat 2.0	2m09s
ManySat 2.0	2m24s
Precosat 576	2m30s
Glucose 2.1	2m35s
ZChaff 2004.11.15	4m10s
Lingeling ala	Did not terminate (40m+)

Figure 3: Heusser's SAT solving results - UNSAT

SAT Solver	Time to SAT
Cryptominisat 2.9.2	42s
Cbmc 4.0 Minisat	1m05s
Rsat 2.01	1m29s
restartsat	38s
Minisat 2.0	1m70s
ManySat 2.0	1m16s
Precosat 576	1m23s
Glucose 2.1	45s
ZChaff 2004.11.15	22s
Lingeling ala	Did not terminate

Figure 4: Heusser's SAT solving results - SAT

SAT Solver	Time to SAT	Speedup
Cryptominisat 2.9.2	54s	-10%
Cbmc 4.0 Minisat	1m11s	-9%
Rsat 2.01	1m10s	+27%
restartsat	26s	+42%
Minisat 2.0	Did not terminate	
ManySat 2.0	39s	+94%
Precosat 576	25s	+232%
Glucose 2.1	1m17s	-42%
ZChaff 2004.11.15	Did not terminate	
Lingeling ala	Did not terminate	

Figure 5: Heusser's SAT solving results on Block 218430 - SAT

it's true, there's one solver which performed worse. Minisat 2.0 took 130 seconds to SAT, and 129 seconds to UNSAT. This is a very low difference, but it projects forward to a flaw of the SAT based method.

I mentioned it earlier in my thesis, but it is generally unknown what makes a CNF instance hard to solve, which leaves only speculation or analysis of the stats provided by the SAT solvers to come to useful conclusions. This is exactly what Heusser found too.

The table 5 shows Heusser's test on Block 218430, when the instance is SAT.

The geometric mean runtime for the genesis block over all solvers is 59s, while block 218430 clocks in at 47s. But Heusser excluded the non terminating solvers from the calculation. If we add just 10 minutes for the non terminating solvers, then block 218430 will be slower on average.

The fact, that two more solvers did not terminate, just backs the claim that it is unknown what makes a SAT instance hard. If we count the number of solvers which performed worse on Block 218430, we can see that more leading zeros didn't help as 5 performed worse, while only 4 performed better. Precosat 576 for Heusser is clearly the biggest winner. With a different solver, I have similar results. First, I performed much better on blocks which contained more leading zeros, but after I tested more blocks, I got sometimes worse results with some later blocks. If leading zeros would really help, and the SAT solving wouldn't be random, then the same solver should always perform better on blocks with more leading zeros, with the same settings for the solver (int this case, no parameter tuning).

3.3 Discussion

3.3.1 Inaccuracies in Heusser's main concept

In Heusser's method, there's a big mistake which I only found later as I understood both his method and CBMC better. The main concept of his method is that instead of brute force loops, we compute only the paths where the hash is valid. This is done by assumptions, where we assume the leading zeros. How would CBMC know which hash is correct, if we don't compute it? We always need to compute the hash first, in order to be able to check if it complies with the target. Heusser placed the assumptions after the SHA256 algorithm, which makes no sense, since the hash is already computed. It doesn't cut down the formula, instead, assumptions make the CNF larger. Write any simple C code and add an assumption to it, the CNF will always have more variables and clauses, because this way it expresses the assumptions with the newly created clauses and variables, but this doesn't always lead us to better SAT solving, especially in the case of the SAT miner.

Heusser wrote the following *"Assumptions on the outputs will result in restrictions of the input – in our case this means only valid nonces will be considered"*. If this would be true, then it would mean that only the nonce which is valid, will be considered. In this case, CBMC should finish in a few seconds, as it only have to execute the SHA256 algorithm once.

An assumption can be placed before SHA256, but it's still unfeasible to use in practice. We can't assume leading zeros before SHA256, but the nonce can be restricted before any iteration of the algorithm happens. Heusser placed his non deterministic nonce and it's assumption after the first chunk is processed by SHA256. If we place it before the algorithm starts, we should see better results, in theory. In practice this will not affect the runtime, and we get the same results as before.

As wee need to compute hashes with SHA256, the algorithm needs to be analyzed by CBMC. SHA256 is a well written hash function which currently has no flaws, and no one could break it. What we need is a pre-image attack, because we want to find an input which produces a correct hash, but this is unfeasible at the time of writing. SHA256 is prepared for a lot of different kind of attacks. This involves encoding the algorithm as a SAT problem too [16].

3.3.2 SAT based method in practice

While the SAT based method is not entirely unusable, it's not a good way to mine Bitcoin. While external SAT solvers can solve SAT instances sometimes very effectively, the nonce can't be retrieved from their output. Only CBMC can be used. I tested it's other solvers

like mathsat or z3, but none of them was more effective than the default one, so the best we can do is to use CBMC with it's plain, default settings, to mine Bitcoin using SAT solving. In real Bitcoin mining we can't limit the range of the nonce, because the correct nonce is unknown. We need to leave to nonce unlimited and let CBMC try all possible nonces. This makes things even more complicated, and CBMC will likely run forever. We might start several CBMC runs in parallel, but this is unfeasible, as even if we set the nonce range to 1 million, we would need more than 4000 different runs in parallel, to compute the hash "effectively". Even if this would be feasible, we would still be required to change the timestamp or the merkle root of the block we hash, because most of the time, for one istance of the block, the 4 billion nonce values will not be sufficient. Today, the global hashing power of Bitcoin can easily reach 750 EH/s. That's $7.5 \cdot 10^{20}$ hashes per seconds, which is far more than just 4 billion ($4 \cdot 10^{9}$) hashes.

The reason why the SAT based method was effective sometimes, and solvable with CBMC, is because we used past blocks. All of them has known, fixed values. In the SAT based method we don't have to bother with changing the timestamp, re-calculate the merkle root etc. But real mining is far more complicated.

If we assume the SAT based method is effective, it would still be complicated, probably unfeasible, to use CBMC on a mining program. Miners need networking, which makes things complicated. Network requests are not instant, and they will hardly be analyzed with CBMC. Mining programs continously communicate with the pool and with other nodes. Miners need to track changes all the time. It must also be checked that no one found a valid block before us.

There's a library, called libblkmaker, which implements the getBlockTemplate (GBT) mining protocol of Bitcoin. The official documentation of GBT says the following: *"All your miner needs to do then is handle the networking, and ask the blkmaker module for data (block headers to search)"* [17]. Although it sounds simple, building our own miner still remains a challenging task. There are some open-source mining programs which we can use if we decide not to implement it for ourselves. An example is ccminer, which can only be used with Nvidia CUDA GPUs [18].

If we have a miner program written in C, then we can use CBMC to mine. We will likely encounter loops which can not be unbounded. I tried to run libblkmaker's example.c file too, just to see if the analyzation is possible at least on the mining protocol itself, but the results were the same, as expected with mining programs. The following shows how I used CBMC on the example file:

```
cbmc example.c blkmaker.c base58.c blkmaker_jansson.c blktemplate.c
--trace -I/mnt/d/Diplomamunka/SAT_for_Bitcoin_mining/libblkmaker >
libblkmaker-test.txt
```

```
5798  Unwinding loop strcasecmp.0 iteration 6 file <builtin-library-strcasecmp> line 44 function strcasecmp thread 0
5799  Unwinding loop strcasecmp.0 iteration 7 file <builtin-library-strcasecmp> line 44 function strcasecmp thread 0
5800  Unwinding loop strcasecmp.0 iteration 8 file <builtin-library-strcasecmp> line 44 function strcasecmp thread 0
5801  Unwinding loop strcasecmp.0 iteration 9 file <builtin-library-strcasecmp> line 44 function strcasecmp thread 0
5802  Unwinding loop blktmpl_getcapability.0 iteration 23 file blktemplate.c line 54 function blktmpl_getcapability thread 0
5803  Unwinding loop strcasecmp.0 iteration 1 file <builtin-library-strcasecmp> line 44 function strcasecmp thread 0
5804  Unwinding loop strcasecmp.0 iteration 2 file <builtin-library-strcasecmp> line 44 function strcasecmp thread 0
5805  Unwinding loop strcasecmp.0 iteration 3 file <builtin-library-strcasecmp> line 44 function strcasecmp thread 0
```

Figure 6: Results of libblkmaker analyzation with CBMC

```
**** WARNING: no body for function b58tobin
**** WARNING: no body for function _blkmk_bin2hex
**** WARNING: no body for function b58check
**** WARNING: no body for function json_object
**** WARNING: no body for function json_array
**** WARNING: no body for function json_string
**** WARNING: no body for function json_array_append_new
**** WARNING: no body for function json_object_set_new
```

Figure 7: Missing body for methods

I tried this process under WSL-2 Ubuntu. All .c files which are part of the implementation, have to be provided to CBMC. Otherwise, it will output that implementation for some methods are missing. The -I parameter sets where CBMC should search for header files. The last part of the command outputs the results to a text file. The results can be seen in Figure 6. The methods shown couldn't be unwinded. There were also warnings about missing body for some methods which can be seen in Figure 7.

These methods belong to libraries which I haven't included for CBMC. The methods with the "json" prefix, are all part of the jansson library for C, which is not built-in for compilers. But these warnings doesn't affect the results, as:

- The unwinding problem was caused by other methods
- If there's no body for a function, CBMC simply skips the proper analyzation of it, which makes the run even faster. I tested it under a simple networking program which uses the curl library.

A solution would be to separate the miner implementation and the SAT solver run. We could write a miner, and call CBMC from the code. Then, we only need to adjust the separate C file which will be parsed, the one that was used in Heusser's method (satcoin.c). The miner could first change the hard-coded block in satcoin.c, then it can call CBMC. If

the network changes something, or simply the block template needs to be changed, these can all be adjusted from the miner code, before every CBMC call. If we implement a miner this way, then we can use Heusser's SAT based method for real Bitcoin mining, but the inaccuracies with the method still remain.

Chapter 4

My Results

4.1 My SAT solving results

The Table 4.1 shows my results on the genesis block. I used the same parameters as Heusser (6 leading zeros assumed, same assertion, 1k nonce range). Note, that I modified his code a bit, but it works the same way. I used the default settings for every solver, in all of my tests (no parameter tuning).
Compared to Heusser, none of my tested solvers could finish on the genesis block, at least with his original settings for the block.

If I modify the assumptions, the results will be different. Two of my solvers could solve the genesis block, by assuming 10 leading zeros, instead of 6 (although the target was 8 leading zeros for the genesis block, the accepted hash contained 10 leading zeros).

Table 4.3 will show my results on Block 780000. I assumed 20 leading zeros, and used 1k for the nonce range.

Manysat performed better and was able to solve block 780000 in 5 seconds, but zchaff performed worse on this block, than on the genesis block. The rest of the solvers still couldn't finish.

Solver	Result
Cryptominisat 5.11.21	did not terminate (10m+)
Minisat 2.2.0	did not terminate (10m+)
Manysat 2.0	did not terminate (10m+)
Rsat 3.01	did not terminate (10m+)
Zchaff 2004.11.15	did not terminate (10m+)
Z3	did not terminate (10m+)

Table 4.1: My results on the genesis block

Solver	Result
Cryptominisat 5.11.21	did not terminate (10m+)
Minisat 2.2.0	did not terminate (10m+)
Manysat 2.0	19s
Rsat 3.01	did not terminate (10m+)
Zchaff 2004.11.15	34s
Z3	did not terminate (10m+)

Table 4.2: My results on the genesis block with 10 zeros assumed

Cryptominisat 5.11.21	did not terminate (10m+)
Minisat 2.2.0	did not terminate (10m+)
Manysat 2.0	5s
Rsat 3.01	did not terminate (10m+)
Zchaff 2004.11.15	40s
Z3	did not terminate (10m+)

Table 4.3: My results on block 780000 with 20 zeros assumed

Other tests show the same inconsistencies. Manysat was the best performing solver for me. It solved Block 780000 in 5 seconds, but Block 780900 took 23 seconds to solve. Then I tried to solve Block 798000 with manysat, and It couldn't solve it (did not terminate in 10 minutes). I also tried Block 756951 which has 24 leading zeros in it's hash, but it did not terminate.

Testing CBMC's built-in solver, the results were the same. Sometimes using fewer assumptions on the same block helped, and produced better results. I have a test with Block 780000 when I didn't assume any leading zeros, and used only the assertion itself to check the hash. Nonce range is 1k. The following is my assertion:

```
1     int flag = 0;
2     if ((unsigned char)(state[7] & 0xff) == 0x00 &&
3         (unsigned char)((state[7] >> 8) & 0xff) == 0x00);
4     {
5         flag = 1;
6     }
7     assert(flag == 0);
```

Listing 4.1: Assertion for Block 780000

It finished in 289 seconds. But another test, where I assumed these 4 leading zeros too, did not terminate. Another one, where I used no assumptions but asserted all 20 leading zeros, did not terminate. And I have a really interesting test too, where everything was

the same, but I limited the nonce range to just 10 values. Although theoretically, a much lower nonce range should make the SAT solving process faster, in this case, it performed much worse, and it took 1038 seconds (more than the 10 minutes deadline) for CBMC to solve it.

4.2 Automated tests

Manually testing and running SAT solvers is uncomfortable. Some tests can take many hours. To automate the SAT solving process I created a few python scripts. The scripts are able to query real Bitcoin blocks with Blockchair.com's API, and based on them they create the CNF files. Running the any solver is automated too, the solver can be passed as a command line argument for the script.

The first test can be seen in Figure 8

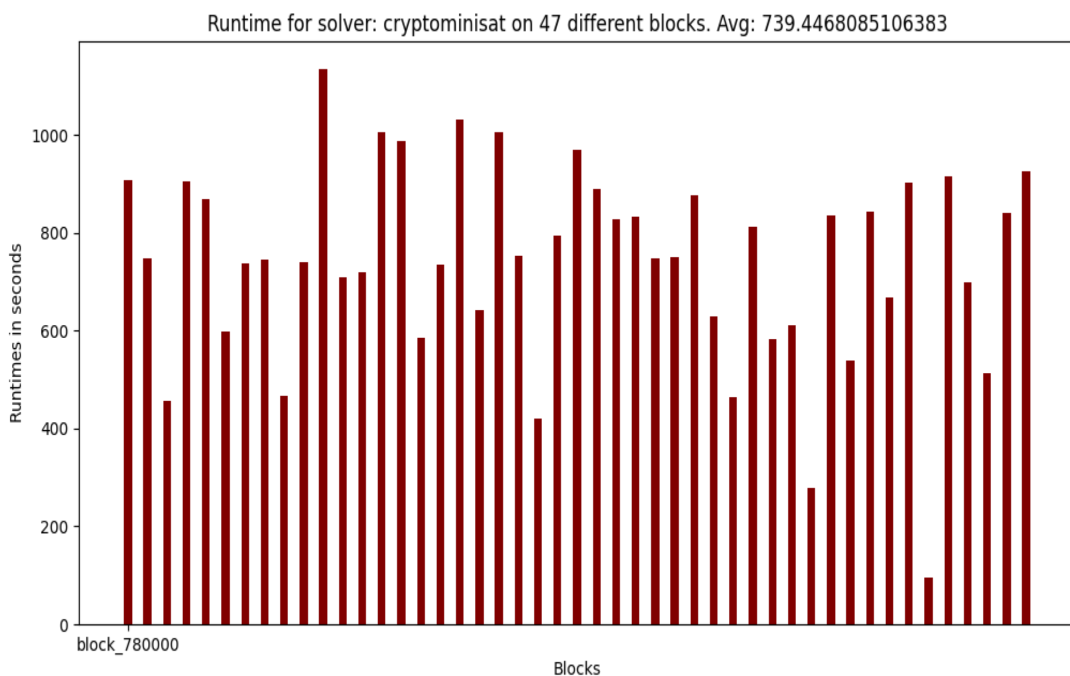

Figure 8: Runtimes on 47 blocks from Block 780000

The nonce range was 1000 for all blocks. The tests were running on a Linux system with an older version of cryptominisat. The next test is in Figure 9.

The two tests show on many blocks that leading zeros doesn't help with the SAT solving. The result in the average runtime is minimal, but the blocks starting from 780000 have two times more leading zeros than the blocks from 10000. Using 130 blocks starting

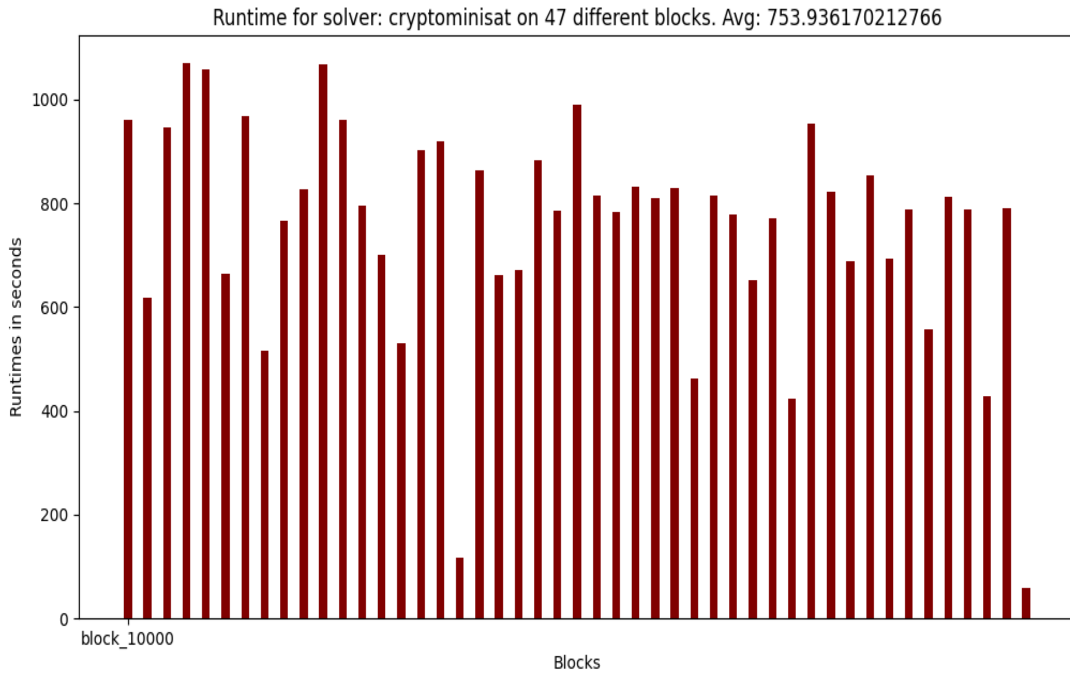

Figure 9: Runtimes on 47 blocks from Block 10000

from block 780000, the results will be worse, and the average will be more than on those 47 blocks from starting from block 10000.

I also tested higher nonce ranges. If I set the nonce range to 10 million, then starting from block 10000, the first block finished under 2500 seconds, but the following one couldn't finish in hours.

Chapter 5

Conclusion

Based on all my tests, including the tests with CBMC's built in solver and the external solver's tests too, and taking note of Heusser's inaccuracy, it seems more leading zeros in the hash don't help with SAT solving. The runtime doesn't grow linearly with the nonce range. It is unknown what makes a CNF instance hard to solve, so it's essentially random, what blocks will finish fast. Every bit of modification to the code, which CBMC uses to generate the CNF, will highly affect the results. On average, the SAT based method is much slower than regular brute-force, and it could hardly be used for real Bitcoin mining.

Bibliography

[1] Massimo Di Pierro. What is the blockchain? *Computing in Science & Engineering*, 19(5):92–95, 2017. doi: 10.1109/MCSE.2017.3421554.

[2] Satoshi Nakamoto. Bitcoin whitepaper. *Bitcoin*, 2008. Accessed: 2024-04-13.

[3] U.S. Energy Information Administration. Tracking electricity consumption from u.s. cryptocurrency mining operations. https://www.eia.gov/todayinenergy/detail.php?id=61364, 2024. Accessed: 2024-04-20.

[4] CoinWarz.com. Bitcoin hashrate chart. https://www.coinwarz.com/mining/bitcoin/hashrate-chart, 2024. Accessed: 2024-04-20.

[5] Hamza Baniata and Attila Kertesz. Partial pre-image attack on proof-of-work based blockchains. *Blockchain: Research and Applications*, page 100194, 2024.

[6] Bitcoin developer. Bitcoin mining guide. https://developer.bitcoin.org/devguide/mining.html, 2024. Accessed: 2024-04-20.

[7] SoloChance. Solochance - solo mining calculator. https://solochance.com/, 2024. Accessed: 2024-04-20.

[8] Bitcoin Wiki. Bitcoin genesis block. https://en.bitcoin.it/wiki/Genesis_block, 2024. Accessed: 2024-04-20.

[9] Daniel Kroening, Peter Schrammel, and Michael Tautschnig. Cbmc: The c bounded model checker, 2023.

[10] Wikipedia contributors. Boolean satisfiability problem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Boolean_satisfiability_problem&oldid=1219369085, 2024. [Online; accessed 20-April-2024].

[11] Wikipedia contributors. Static single-assignment form — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Static_

single-assignment_form&oldid=1209495970, 2024. [Online; accessed 20-April-2024].

[12] Yu Sasaki, Lei Wang, and Kazumaro Aoki. Preimage attacks on 41-step sha-256 and 46-step sha-512. *Cryptology ePrint Archive*, 2009.

[13] Emanuele Bellini, Alessandro De Piccoli, Rusydi Makarim, Sergio Polese, Lorenzo Riva, and Andrea Visconti. New records of pre-image search of reduced sha-1 using sat solvers. In *Proceedings of the Seventh International Conference on Mathematics and Computing: ICMC 2021*, pages 141–151. Springer, 2022.

[14] Timo Hanke. Asicboost - a speedup for bitcoin mining, 2016.

[15] Norbert Manthey and Jonathan Heusser. Satcoin - bitcoin mining via sat. *Proceedings of SAT Competition*, 2018:67–68, 2018.

[16] Trevor Phillips. Sat-solver, optimization, and belief propagation attacks on sha-256, 2024.

[17] Bitcoin Wiki. Bitcoin - getblocktemplate protocol. https://en.bitcoin.it/wiki/Getblocktemplate, 2024. Accessed: 2024-04-20.

[18] Tanguy Pruvot. ccminer. https://github.com/tpruvot/ccminer, 2019. Accessed: 2024-04-20.