Optimising the SHA256 Hashing Algorithm for Faster and More Efficient Bitcoin Mining[1]

by

Rahul P. Naik

Supervisor:

Dr. Nicolas T. Courtois

MSc Information Security

DEPARTMENT OF COMPUTER SCIENCE

September 2, 2013

[1] This report is submitted as part requirement for the MSc Degree in Information Security at University College London. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

Since its inception in early 2009, Bitcoin has attracted a substantial amount of users and the popularity of this decentralised virtual currency is rapidly increasing day by day. Over the years, an arms race for mining hardware has resulted with miners requiring more and more hashing power in order to remain alive in the Bitcoin mining arena. The hashing rate and the energy consumption of the mining devices used are of utmost importance for the profit margin in Bitcoin mining. As Bitcoin mining is fundamentally all about computing the double SHA256 hash of a certain stream of inputs many times, a lot of research has been aimed towards hardware optimisations of the SHA256 Hash Standard implementations. However, no effort has been made in order to optimise the SHA256 algorithm specific to Bitcoin mining.

This thesis covers the broad field of Bitcoin, Bitcoin mining and the SHA256 hashing algorithm. Rather than hardware based optimisations, the main focus of this thesis is targeted towards optimising the SHA256 hashing algorithm specific to the Bitcoin mining protocol so that mining can be performed faster and in a more efficient manner. These optimisations take advantage of the fixed or predictable nature of the input stream of data in Bitcoin mining and various shortcuts are discussed to calculate particular rounds or message schedules that achieve the same computational results as off-the-shelf SHA256.

Although these algorithm based optimisations can no longer allow generic SHA256 hashing, they are meant to radically optimise the process of Bitcoin mining. It has been claimed that if these improvements are to be implemented in mining devices, the double SHA256 computation reduces to a 1.8624 SHA256 computation which essentially means a faster hashing rate and lots of energy savings.

Keywords: Bitcoin, hash, SHA256, mining, ASIC, FPGA, algorithm optimisations, compression function, message schedule, Savings Factor, block, transactions, proof-of-work.

Acknowledgements

Firstly, I would sincerely like to thank my supervisor, Dr. Nicolas T. Courtois for his continued support and guidance throughout the duration of my dissertation. He has been keenly involved in the entire process and has provided me with timely suggestions and improvements that have helped me a lot. His mentoring, motivation and management has sparked tremendous ideas in my mind for this thesis and I am thankful for having him as my supervisor.

Furthermore, I am eternally grateful to The British Council for awarding me with the fully-funded Jubilee Scholarship for my MSc in Information Security at University College London. It would have been almost impossible for me to have met the expenses of my education without this scholarship. Without any financial barriers, this past year has been a wonderful experience and I shall cherish it for the rest of my life.

Finally, I am truly indebted to my parents and friends for encouraging and supporting me through tough times. I am here only because of your continued love and support. Thank you.

Contents

Chapter 1: Introduction

Bitcoin is a global, decentralised, pseudonymous virtual currency scheme that is not backed by any government or other legal entity. It was invented **[42]** and instigated back in 2008 by Satoshi Nakamoto (a pseudonym). Bitcoin depends on peer-to-peer networking and basic but ingeniously applied cryptography techniques to maintain its integrity and authenticity. It is based on the principal that for the currency to have any value, the creation of new Bitcoins must be limited. Bitcoins are thus slowly minted into existence through a computationally intensive process called Bitcoin Mining and Proof-of-Work generation.

With the current market price **[4] [41]** of about 1 BTC = $130 as compared to about 1 BTC = $0.2 exactly 3 years ago, the Bitcoin virtual currency is an apt example that all big things start small. With more vendors and merchants turning towards Bitcoin as a mode of payment as well as people starting to look at Bitcoin as digital gold **[51]** and a safe haven to the economic turmoil **[17] [49]** of fiat currencies, Bitcoin is definitely gaining popularity fast. With more than 60000 Bitcoin transactions being conducted per day **[11]** and the current incentive of 25BTC ≈ $3200 for every block mined, Bitcoin mining has also become very attractive as a prospective business. This attraction has led to many miners competing to be the first to mine a new Block and be awarded with the mining reward. As a result, both the Bitcoin network hash rate and difficulty are skyrocketing which is making it even harder to mine Bitcoins. Today, Bitcoin mining and has become computationally very intensive and with more and more participants joining this hunt to mine Bitcoins, the electricity consumption of the Bitcoin network has also skyrocketed. Current data estimates **[11]** that per day about 8700 megawatt hours of electricity is being spent on mining which on average amounts to more than 1 and a quarter million dollars being spent on electricity per day by the Bitcoin miners. Many have already started speculating that Bitcoin mining is a hazard to the environment **[12]** while others are against that conception **[28]**.

With the view of trying to stay alive in the Bitcoin mining arena, miners have entered into an arms race for Hashing power. As a result, many businesses[2] have emerged that provide specialised but expensive mining devices. Bitcoin mining in its infancy used CPUs which later

[2] Butterfly Labs - http://www.butterflylabs.com/, Cointerra - http://cointerra.com/, Bitfury - http://www.bitfury.org/, KnCMiner - https://www.kncminer.com/

evolved into using faster but more power consuming GPUs. After that, miners turned to FPGAs and finally in mid 2013, high speed dedicated ASIC devices entered the market. This has left the earlier methods of mining obsolete as they simply cannot compete with the hashing rate of ASICs. These ASIC mining devices offer huge hashing rates but higher energy consumption as well. Some dedicated ASIC chips, claim to provide very high hashing rates (Around 400[3]-600[4] GH/s). A 2 TH/s ASIC mining device[5] has also been announced by Cointerra.

1.1 Motivation and Goal

Much research effort has been spent on hardware optimisations and improvements on the SHA256 hashing algorithm which forms the basis of Bitcoin mining. These hardware optimisations have been aimed towards generic SHA256 hashing and are currently implemented and used in most mining devices. Improvements in SHA256 hardware have greatly increased the throughput and efficient implementations have also decreased the power consumption of the device. Mining typically involves calculating the double SHA256 hash of an input stream of data and the mining devices in the market use SHA256 cores to perform this double hashing during the mining computations.

The question now arises if there is a more efficient way to mine Bitcoins where mining devices would calculate something less than a double SHA256 and end up with the same result. Till date, to the best of the author's knowledge, no research effort has been made in optimising the SHA256 algorithm specific to Bitcoin mining. If this was possible and improvements at the SHA256 algorithm level were found, this would have a tremendous impact in the Bitcoin community as mining could then be performed faster and that too in a more efficient manner. Mining devices could then have a higher hash rate with the same power consumption as before. In other words, mining devices could have the same hashing rate but with a lower power consumption.

We thus study the SHA256 hashing algorithm and the Bitcoin block Hashing algorithm in detail and try and understand how the double SHA256 hashing can be improved so that

[3] See Bitfury 400GH/s MiningRig http://thegenesisblock.com/bitfury-400-ghs-bitcoin-mining-rig-hits-us-shores/
[4] See BFL 28nm Technology Bitcoin Mining Card - The Monarch http://thegenesisblock.com/butterfly-labs-tests-market-tolerance-with-600-ghs-pre-order-announcement/
[5] See TerraMiner IV http://thegenesisblock.com/cointerra-announces-2ths-asic-bitcoin-miner-for-15750/

lesser number of operations would need to be performed in order to achieve the same result as traditional double SHA256 hashing. The goal would be to take advantage of the fixed or predictable nature of the input data stream in Bitcoin mining. Any improvements achieved would potentially be worth a lot of money as they will bring about huge savings in the miners' electricity bills.

1.2 Structure of the Thesis

The rest of the thesis is organised as follows. Chapter 2 will contain an overview of the Bitcoin system and will touch upon important topics related to the Bitcoin protocol and will serve as background knowledge for what follows. Chapter 3 will be a detailed explanation of the SHA256 hashing algorithm which forms the basis of Bitcoin mining. Chapter 4 will concentrate on the hardware implementation of the SHA256 hashing algorithm. This chapter will also focus on a survey of the past research made in optimising SHA256 in hardware. Chapter 5 will bring us back to our research problem in question i.e. the Bitcoin block hashing algorithm which is the algorithm used in Bitcoin Mining. This chapter discusses what data is typically hashed by the miners in order to produce a block hash that will be accepted by the Bitcoin network. It also discusses the important point of the frequency by which the data changes which is a major factor involved in the suggestions of the optimisations. Chapter 6 is the most important that discusses the contribution of this thesis. Optimisations in the SHA256 hashing algorithm that are specific to Bitcoin mining have been suggested in ample detail in this chapter. Next, a quantitative analysis is performed in Chapter 7 that details the amount of savings made as a result of the suggested optimisations. These results are discussed in this chapter titled as Discussion. We also touch upon certain limitations as well as the future work associated with the aforementioned contributions. Finally, we finish with a conclusion that summarises the contributions made in this thesis.

Chapter 2: An Overview of Bitcoin

2.1 What is Bitcoin?

Bitcoin **[8] [44] [46] [50]** is a completely decentralised electronic global electronic currency that is not backed by any government or legal entity. It was designed, developed and launched by a pseudonym by the name of Satoshi Nakamoto back in 2009. Some wonder **[54]** if the pseudonym is actually a clever blend of the four technological companies viz. **SA**msung, **TOSHI**ba, **NAKA**michi and **MOTO**rola. Others speculate if it is not a single person but a group of individuals sitting at NSA or Google. Regardless, Satoshi Nakamoto exhibited his invention though his paper **[42]** in 2008 right about the time when the global financial crisis had hit the world. The paper discussed the concept of a purely peer-to-peer version of electronic currency without the need of a trusted third party or any financial institution. He proposed that his electronic payment system would purely rely on cryptographic proof instead of the ancient methodology that relied upon trust. This would allow any two parties to transact directly amongst each other without the need of a trusted third party to validate their transaction. Satoshi Nakamoto also discussed how the double-spending problem was addressed in his proposal without the use of third parties. He discusses the concept of hashing Bitcoin transactions into the longest hash-based proof-of-work which will make it possible for anyone in the network to verify transactions.

Bitcoin is considered to be a pseudonymous mode of payment meaning that it is partly-anonymous. This in turn means that one can practically enter into a Bitcoin transaction with anyone in the world without having to disclose one's identity. Although there are ways to link a person's identity with his Bitcoin transactions, there are also some steps that can be used to evade this. For a detailed analysis of the anonymity of the Bitcoin system, refer to **[45]**. Although Bitcoin is flourishing more than ever, its legal status and their implications still remain uncertain and **[29] [32]** try to cover and address the legal concerns and aspects that surround Bitcoin. Witnessing the success of Bitcoin, many similar currencies **[24] [53]** have emerged that have actually forked from the same Bitcoin open-source code but with minor technical and administrative alterations **[53]**.

2.1.1 Transactions

A Bitcoin transaction is basically a digitally signed chunk of data that is collected into blocks and broadcast into the peer-to-peer Bitcoin network. Bob can transfer Bitcoins to Alice for her services from the Bitcoins Bob earned from previous transactions. Bob could have been paid by someone else or Bob could have exchanged his fiat currency in exchange for Bitcoins using a Bitcoin exchange service like Mt. Gox **[41]**. Bob could have even mined those Bitcoins and/or would have gained Bitcoins as transaction fees. The point is that Bitcoin are exchanged through transactions and authenticity of these transactions is maintained using ECDSA **[1] [31]** signatures. The recipient needs proof that no double spending has occurred in the current transaction and this has been achieved in Bitcoin by making each node of the Bitcoin network aware of all transactions. Transactions can be seen as records that move Bitcoins to new addresses. A transaction will typically have one or more inputs and outputs where Bitcoins from the inputs are reassigned to one or more recipient addresses in the outputs. In each transaction, the sum of Bitcoins in all inputs must be more than or equal to the sum of Bitcoins in the outputs. If Bitcoins in the inputs are greater than the outputs, the difference is considered as a transaction fee which can be set at the discretion of the payer. The general format of a Bitcoin transaction can be found in **[9]**. Detailed information on all Bitcoin transactions conducted till date can be found at **[3]**.

2.1.2 Blocks

A block is a collection of all or some of the most recent Bitcoin transactions that do not exist in any previous blocks. New blocks are created through a process called Bitcoin mining and appended to a chain on previously accepted blocks called a block chain. For every newly created block, the miner is awarded with a mining reward which initially was 50 BTCs and has now halved to 25 BTCs. The transactions contained in this block thus get verified once the new block is accepted and by the Bitcoin network and is appended to the block chain. A block chain is the Bitcoin equivalent of a universal accounting ledger which ensures that no double-spending of Bitcoins can be performed. Among other things, the most important part of a block is its header which is very important in Bitcoin mining. Each block contains a reference to the previously created block in its header and so such a collection of blocks can be said to form a chain. A detailed description of the Bitcoin block header as well as the

process of Bitcoin mining has been explained in Chapter 5. Details of the Bitcoin block structure can be found in **[9]**.

2.1.3 Proof-of-work and the Longest Chain

Bitcoin mining and the creation of new blocks is essentially about trying to find the solution to the Bitcoin proof-of-work problem. Satoshi made use of this concept from a paper **[2]** that described a proof-of-work protocol to prevent Denial of Service attacks and email spam. A proof-of-work is actually a chunk of data which is computationally costly and time-consuming to produce so as to satisfy certain requirements enforced by the Bitcoin protocol. It is trivial to verify the solution but it takes a lot of trial and error on average before a valid solution to a proof-of-work problem is generated. Proofs of work are used in Bitcoin for generation of new blocks i.e. in Bitcoin mining. A metric called as the difficulty (explained next) is self adjusting that limits the creation of new blocks to 10 minutes per new block. The longest block chain created represents the majority decision of the Bitcoin network and has the greatest proof-of-work effort invested in it. It is thus believed that this particular chain has been backed my most Bitcoin nodes and is therefore legitimate. Thus, if an attacker decides to tamper a previous transaction or a block (known as the history revision attack), he will have to repeat all the proof-of-work of that block and all those follow it and then exceed the legitimate block chain so as to make it accepted by the network. In order to do this, the attacker would need at least half of the current network hash rate which at the moment of writing is 524.81 TH/sec. It would thus be extremely costly with rather meagre returns for someone to attempt such an attack. In Bitcoin, the proof-of-work is implemented by incrementing a field called nonce in the block header until the block header's hash value contains a certain number of preceding 0s. This requirement is determined by the current value of the target and difficulty which are explained next.

2.1.4 Target

Target is a 256 bit long integer that is broadcast and shared by the entire Bitcoin community. This value decides the difficulty of the finding a solution to the proof-of-work problem in Bitcoin. The primary requirement enforced by the Bitcoin protocol for a block to be accepted is that the hash if the block header must be less than or equal to the current

target[6]. Thus, as the target decreases, it becomes more difficult to mine Bitcoins i.e. generate a new block. The Bitcoin protocol relies on a feedback mechanism to dynamically adjust the target based on the speed in which the last 2016 blocks were mined. Each block is typically on average mined every 10 minutes and hence, 2016 blocks would typically take around 2 weeks. Thus, after about every 2 weeks, the target is adjusted that changes the difficulty of the proof-of-work problem. The actual time to create the current 2016 blocks is compared to the time taken to create the last 2016 blocks and the target is modified accordingly. If the new 2016 blocks were created rather quickly than the previous 2016 blocks, this means that it is easier for the network to mine Bitcoins and so the target is reduced accordingly to increase the proof-of-work difficulty so as to restore the criteria of 1 block getting created every 10 minutes.

2.1.5 Difficulty

This quantity is computed from the target and is a measure of how difficult it is to find a solution to the proof-of-work problem i.e. finding a new block compared to the easiest it ever was to find a new block. Similar to the target, the difficulty changes after every new 2016 blocks. Difficulty is thus given by the following equation:

$$\text{Difficulty}^{7} = \text{Maximum Target}/\text{Current Target}$$

Equation 1: Calculation of Difficulty

Adjusting the value of the Target and thereby the difficulty is the mechanism used by Bitcoin to maintain the rate of creation of new blocks i.e. creation of new Bitcoins to every 10 minutes. The average time required (in seconds) to find a new block relative to one's hash rate can be calculated as follows:

$$\text{Time} = \text{Difficulty} * 2^{32}/\text{hash rate}$$

We shall assume that we own the recently announced BFL Monarch with a hash rate of 600GH/s

Average Time to find a new block = 65750060*232/600G = 438333 seconds ≈ 5 days

[6] Current Target - http://blockexplorer.com/q/hextarget
[7] Current Difficulty - http://blockexplorer.com/q/getdifficulty

2.2 The Bitcoin Protocol Specification

2.2.1 Hashes

Bitcoin mostly uses the double SHA256 hashing algorithm i.e. SHA256(SHA256(x)). Hence, when a hash is computed in the Bitcoin system, it is usually a double SHA256 hash. Another hash that has a shorter length message digest called RIPEMD160 is also used but in the creation of the Bitcoin addresses (explained ahead).

2.2.2 Merkle Trees and Merkle Roots

The transactions are indirectly hashed into the merkle root. Satoshi Nakamoto has explained this in his paper **[42]** and claims that the transactions spent before these transactions can be discarded to save disk space. To enable this, the transactions that are selected by the miners to include in their block are hashed in a Merkle Tree, and only the root, called the Merkle Root is included in the block's header. Old blocks can then be compressed by chopping off branches of the Merkle Tree and the interior hashes need not be stored. The below image explains this well:

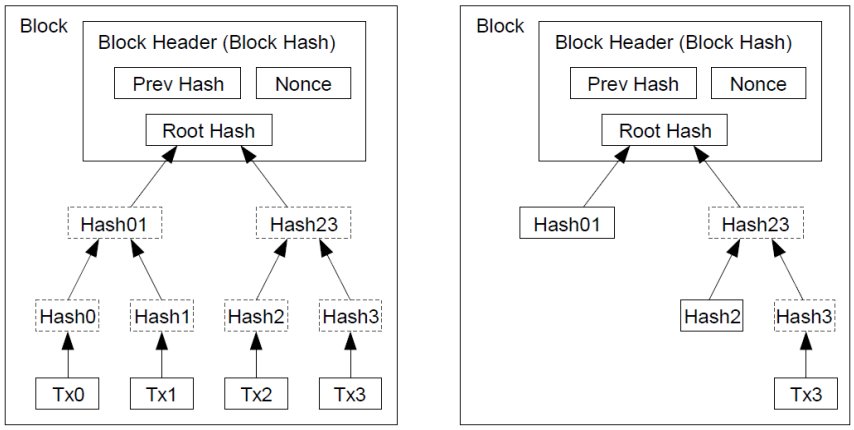

Figure 1: Merkle Tree and Merkle Root

Merkle trees are thus binary trees of hashes and they use double SHA256 hashing. Firstly, the bottom row of the tree is formed with the double SHA256 hashes of the transactions in the block. The row above it will now contain half that number of hashes as shown above in the image to the left. Two hashes from the row below are concatenated to form a 512 bit block and the double SHA256 hash of that is calculated. If a row has odd number of

elements, the final double hash is duplicated to ensure that each row has an even number of elements. This process is continued until the root of the tree is reached and a single 256 bit value remains. This value is called the Merkle Root and is stored in the block header.

Suppose that a miner has decided to include 3 transactions viz. t1, t2 and t3 in his new block. The Merkle Tree and the Merkle Root is calculated as follows:

H1 = DHash(t1) … // DHash(t1) = SHA256(SHA256(t1))

H2 = DHash(t3)

H3 = DHash(t3)

H4 = DHash(t3) … // Hash was duplicated as transactions were odd in number

H5 = DHash(H1 || H2)

H6 = DHash(H3 || H4)

H7 = DHash(H5 || H6)

Thus, H7 is the Merkle Root of these 3 transactions in this block.

Equation 2: Merkle Tree and Merkle Root Calculation Example. Source: [9]

2.2.3 Signatures

Bitcoin uses the Elliptic Curve Digital Signature Algorithm (ECDSA) **[1] [31]** to sign the transactions. New ECDSA public and private keys are generated for every Bitcoin address and the correlation is done internally by the Bitcoin client software.

2.2.4 Bitcoin Addresses

A Bitcoin address is analogous to an email address and Bitcoins can be sent to a person by sending them to one of their Bitcoin addresses. A person can have as many addresses as desired and it is in fact recommended that for additional privacy, it is best to use a unique address for each transaction. Websites that accept Bitcoin as donation often have a mechanism to generate a new Bitcoin addresses as required[8]. A Bitcoin address is a case-sensitive identifier that can be 27-34 alphanumeric characters long and begin with either the number 1 or 3. An example of a Bitcoin address is **1HB5XMLmzFVj8ALj6mfBsbifRoD4miY36v**. A Bitcoin address is actually the hash of an ECDSA public key and on a high level, is computed **[9]** as follows:

[8] See Wikileaks donation page - http://shop.wikileaks.org/donate#dbitcoin

Version = 1 byte of 0s

Keyhash = Version || RIPEMD160(SHA256(ECDSA_pubkey))

Checksum = 1st 4 bytes(SHA256(SHA256(Keyhash))

Bitcoin Address = Base58Encode(Keyhash || Checksum)

Equation 3: Calculation of a Bitcoin Address

2.3 Bitcoin Mining

Bitcoin mining essentially involves computing the double SHA256 hash of the Bitcoin block header such that the hash is less than or equal to the target and is preceded by certain number of zeroes. The number of 0s that need to precede the hash value is determined by the current value of target/difficulty and is explained in detail in section 5.2.6. The current section is concerned with what the miner is rewarded once he has created a new block which is accepted into the block chain by the Bitcoin network.

2.3.1 Mining Reward and Transaction Fees

When a miner is able to discover a new block that is accepted by the Bitcoin network, the miner receives a mining reward in the form of Bitcoins which are an incentive to the miner for the invested time and computational power. It is the generation transaction or a coinbase transaction contained in the block that grants these Bitcoins to the miners. The transaction fees of the transactions contained in that block are also credited via this generation/coinbase transaction. At the time of writing, for every new block, the miner is awarded with 25 BTCs. Earlier until 28th November 2012, the miners were rewarded with 50 BTCs per mined blocked. Roughly by the end of 2016, the reward will get halved again so that each newly found block will have a mining reward of 12.5 BTCs. This is because the Bitcoin protocol relies on the fact that for anything to have value, its supply must be limited. Hence, the Bitcoin protocol is designed such that the number of Bitcoins awarded per block halves after every 210000 blocks. We know that the target/difficulty self adjusts such that a new block can be found in about 10 minutes. This means that the mining rewards will halve after about every 4 years. The result of this is that Bitcoin has a limited and strictly fixed supply of about 21 million BTCs. The level of 21 million BTCs is expected to be reached sometime in the year 2140 but the practical number of 99% of the total Bitcoins mined will be attained sometime in the year 2032. The total number of Bitcoins currently mined and in

circulation are about 11.63 million BTCs **[4]**. One may speculate that this 21m cap isn't enough and Bitcoin isn't scalable. But it is believed that due to the limited supply, Bitcoin will deflate i.e. rise in value. Bitcoins are divisible up to eight decimal places and the smallest unit in Bitcoin is called a Satoshi and 1 Satoshi = 0.00000001 BTC. There are thus almost 2 quadrillion maximum possible atomic units in the Bitcoin protocol. When appropriate high levels of value are reached, people in the future can be seen dealing with smaller units like mBTC, µBTC and nBTC.

2.3.2 Improvement Proposal for the Mining Reward

The built-in mechanism of the mining reward halving is sort of a problem for miners. Notice how it forms a geometric decrement as it gets halved after every 210000 blocks. What this implies is that one day the mining reward suddenly becomes half of what it was before. This also means that suddenly, it becomes twice as costly to mine Bitcoins **[19]**. This hampers the income stability which is of utmost important to businesses. We thus propose to smooth the Bitcoin rewarding mechanism by introducing a linear decrement instead of the default geometric decrement. We propose that the block reward should be decremented after the creation of each block and not after 210000 blocks and we take advantage of the fact that 1 BTC is divisible up to 8 decimal places. The next reward halving is at 420000 blocks and next to next halving is at 630000 blocks. We propose that the following decrement should be introduced after block 420000:

12.5 BTC → 6.25 BTC and 420000 → 630000

Therefore, Reward = Reward - 6.25/(630000-420000) = Reward - 0.00002976

Thus, the number of Bitcoins awarded when block 420001 is mined will be 12.5 - 0.00002976 = 12.49997024 and so on

Equation 4: Calculation of the New Mining Reward

Chapter 3: The SHA256 Hashing Algorithm

3.1 An Overview of SHA256

A detailed description of the SHA256 hashing algorithm can be found in the official NIST standard **[26]**. This section provides an overview of the SHA256 algorithm that forms the backbone of the Bitcoin ecosystem. The integrity of Bitcoin transactions depends upon the collision resistance and pre-image resistance of the SHA256 hashing algorithm. It is important to remember the fact that in the Bitcoin protocol, the SHA256 hash is computed twice.

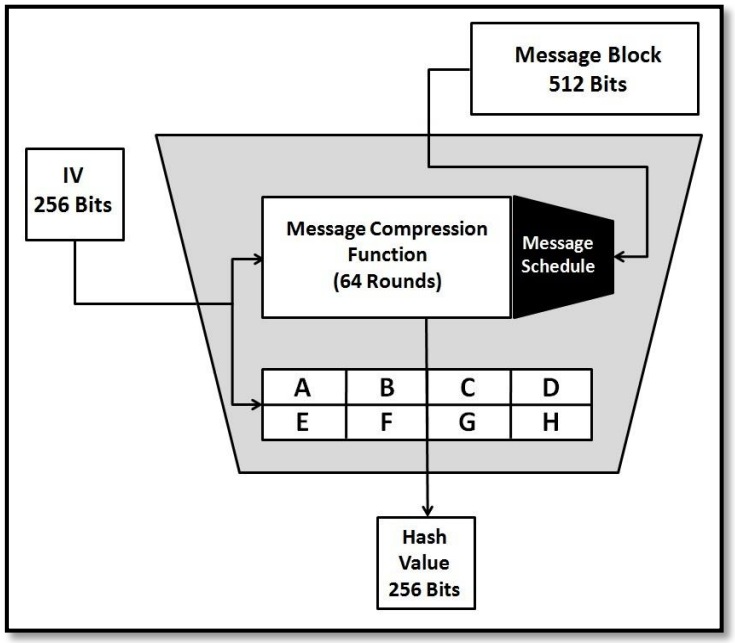

Figure 2: An Overview of the SHA256 Hashing Algorithm.

The SHA256 algorithm takes an input that has a length of less than 2^{64} bits. It has a block size of 512 bits which are represented as a sequence of sixteen 32-bit words. This 512 bit block enters a function called the message compression function in words of 32 bits (W_t) through a message scheduler. Both of these are explained in detail later on. The message scheduler expands the 512 bit message block into sixty-four 32-bit words. The operations inside the SHA256 hashing algorithm are performed on words that are 32-bit in length using eight working variables names as A, B, C, D, E, F, G and H that are also 32-bits in length. Hence, the word length of the SHA256 algorithm is of 32 bits. The values for these working variables are computed at every round and this process continues till 64 rounds have been

completed. Very importantly, it should be noted that all additions in the SHA256 hashing algorithm are performed modulo 2^{32}. Hence, the reader should interpret all additions mentioned henceforth in this text as additions performed modulo 2^{32}.

SHA256 also takes a 256 bit initialisation vector (IV) which is fixed for the first message block. An intermediate message digest obtained at the end of the first 64 rounds which serves as the IV for the next message block. The SHA256 hash function is built using the Davies-Meyer construction where the IV is added to the output at the end of 64 rounds. Thus, after 64 rounds of the message compression function and addition of the IV, the algorithm produces an intermediate message digest of 256 bits. After the entire message blocks have been hashed, a value on 256 bits is obtained that is the final message digest of the input message. The SHA256 hashing algorithm is thus comparable to a block cipher with a 256 bit message block size (IV) and a 512 bit key (message block) that is expanded into sixty-four 32 bit round keys using the message scheduler for each of the 64 rounds of this cipher. The Bitcoin protocol takes advantage of the avalanche property of the SHA256 algorithm that makes it very hard for attackers to find shortcuts in finding a new block that starts with the stipulated number of 0s. The next section will take a deep-dive into the insides of the SHA256 algorithm.

3.2 SHA256 Deep-Dive

The SHA256 hashing algorithm operation can be conveniently divided into three distinct operations. They are as follows:

- **Pre-processing:** Operation that performs padding logic and parses the input message
- **Message scheduler:** Function that generates sixty-four words from an 16 word input message block
- **Compression function:** Function that carries out the actual hashing operation of the message-dependent word that comes out of the message scheduler in each round

3.2.1 SHA256 Pre-processing

The SHA256 pre-processing is the initial step that needs to be performed before the message scheduling and the compression function can be applied. The pre-processing stage performs the following three tasks in order:

- Pad the message to make it a multiple of 512 bits,
- Parse this message into 512 bit blocks, and
- Set the initial hash value

3.2.1.1 Padding the Message

The message to be hashed needs to be padded first. Padding is done so as to ensure that the message to be hashed is a multiple of the block size for SHA256 i.e. 512 bits. Now, if we consider that the length of the message is l bits, the padding logic is such that it appends a bit "1" at the end of the actual message which is then followed by k number of zero bits. Here, k is the smallest, non negative solution to the following equation **[26]**:

$$l + 1 + k = 448 \bmod 512$$

Equation 5: SHA256 Padding Logic

The equation above is such because SHA256 allows an input message to have a length of up to 2^{64} bits. After the trail of 0 bits, a 64 bit block is appended at the end that is equal to l represented in a binary representation.

3.2.1.2 Parsing the Padded Message

After the message has been padded using the logic explained above, it is parsed into N 512-bit blocks so that the message scheduling and hash computation can be commenced.

3.2.1.3 Setting the Initial Hash Value (H0)

Before the hash computation commences, the initial hash value is set which consists of the following 32 bit words:

H_0^0	H_1^0	H_2^0	H_3^0	H_4^0	H_5^0	H_6^0	H_7^0
0x6a09e667	**0xbb67ae85**	**0x3c6ef372**	**0xa54ff53a**	**0x510e527f**	**0x9b05688c**	**0x1f83d9ab**	**0x5be0cd19**

Table 1: SHA256 Initialisation Vector. Source: [26]

It is interesting to know the origin of this 8 word value. They were obtained by taking the first 32 bits of the fractional parts of the square roots of the first 8 prime numbers. This initial hash value acts as the IV for the SHA256 algorithm as explained in the earlier section.

3.2.2 SHA256 Message Scheduler

After the pre-processing stage is completed, the message schedule block takes the first 512 bit message block and outputs the message dependant words W_t. The 32 bit message-dependant words that that are output by the message scheduler for every round are labelled as W_0, W_1,…, W_{63} (for t=0 to 63) and they are calculated as follows:

For $0 \leq t \leq 15$,

$W_t = M_t$

For $16 \leq t \leq 63$,

$W_t = \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-5}) + W_{t-16}$

Equation 6: SHA256 Message Scheduler. Source: [26]

Here, σ_0 and σ_1 are two logical functions specific to the SHA256 message scheduler that operate on a 32 bit word. The details of these functions are provided below:

$\sigma_0(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$

$\sigma_1(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$

Equation 7: Logical Functions σ_0 and σ_1. Source: [26]

The two logical functions σ_0 and σ_1 operate on a word of the input message and apply the above bitwise operations to it. $ROTR^x$ stands for bitwise rotate right for x bits, SHR^x stands for bitwise shift right and $\oplus$ stands for the bitwise exclusive or. This message schedule block is usually implemented in hardware by using 16 stages of 32 bit shift registers and three 32 bit adders **[33]** for the 512 bit data block processing.

The visual representation of the message scheduler has been shown in figure 3. The multiplexer is controlled by logic to allow either M_t or the computed W_t to pass depending on the value of t. Every round, the 32 bit value of W_t is shifted to the left using the shift registers as previous values of W_t are required to calculate future values of W_t. It is appealing to know that two logical functions σ_0, σ_1 and the message schedule logic

explained don't come into play until the 17th round. The 512 bit input message is fed as it is to the message compression function for the first 16 rounds.

3.2.3 SHA256 Message Compression Function

The message compression function performs the actual hashing operation and is the main operation that enforces the one-way property of SHA256. Other than the eight words of working variables A, B, C, D, E, F, G and H that are used and updated in each round, two temporary words T_1 and T_2 are also used by the message compression function for computation of the variables A and E in each round. The first step that the message compression function performs is that it initialises these 8 working variables with the IV (if it is the first block) or with the intermediate hash of the previous block (if it is not the first block being hashed).

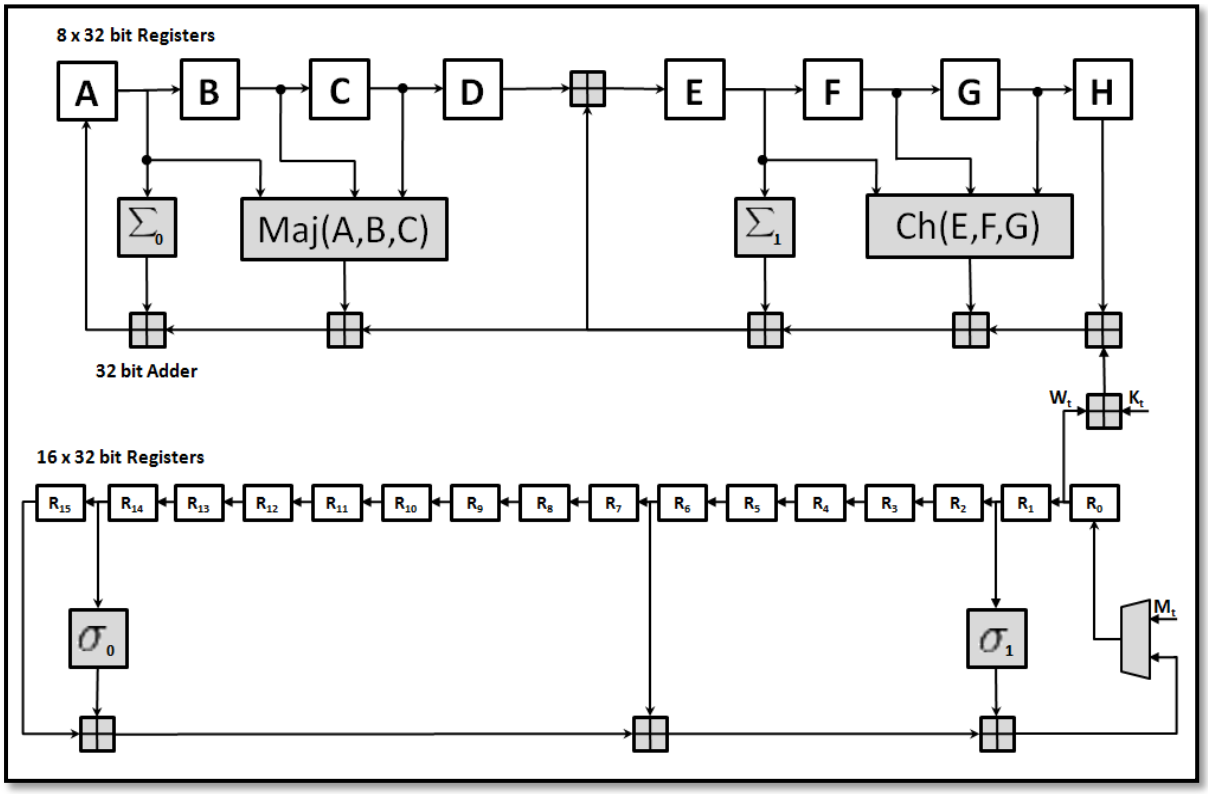

Figure 3: SHA256 Message Compression Function (Above) and Message Scheduler (Below)

The figure above shows the typical implementation of the message compression function as well as the message scheduler that operate in tandem. It can be clearly seen from the figure above that at every round, 6 out of 8 values of A, B, C, and E, F, G are shifted by one position to B, C, D and F, G, H respectively. W_t is the 32 bit data calculated by the message scheduler and fed to the compression function and K_t is a round specific 32 bit constant whose round

specific values are specified in appendix A. SHA256 uses 64 constants (1 for each round) that are 32 bit words and they have been obtained by taking the first 32 bits of the fractional parts of the cube roots of the first 64 prime numbers. Variables A and E are dependent on all input values and are computed in each round using equations explained next. After the 8 working variables are initialised as explained earlier, 64 rounds of the compression function are applied to them and intermediate round values of these variables are calculated as follows:

$$T_1 = H + \sum_1(E) + Ch(E, F, G) + K_t + W_t$$

$$T_2 = \sum_0(A) + Maj(A, B, C)$$

$$H = G;\ G = F;\ F = E$$

$$E = D + T_1 = D + H + \sum_1(E) + Ch(E, F, G) + K_t + W_t$$

$$D = C;\ C = B;\ B = A$$

$$A = T_1 + T_2 = H + \sum_1(E) + Ch(E, F, G) + \sum_0(A) + Maj(A, B, C) + K_t + W_t$$

Equation 8: Message Compression Function. Source: [26]

The four logical functions mentioned above perform the core operation of introducing the confusion and diffusion in W_t that enters in 32 bit words at each round. After applying the above equations to the working variables for 64 rounds, an appropriate level of the avalanche effect is observed. These 4 logical functions are now explained next.

$$Ch(X, Y, Z) = (X \wedge Y) \oplus (\neg X \wedge Z)$$

$$Maj(X, Y, Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)$$

$$\sum_0(X) = ROTR^2(X) \oplus ROTR^{13}(X) \oplus ROTR^{22}(X)$$

$$\sum_1(X) = ROTR^6(X) \oplus ROTR^{11}(X) \oplus ROTR^{25}(X)$$

Equation 9: Logical Functions Ch, Maj, $\sum_0$ and $\sum_1$. Source: [26]

Here, logical functions Ch and Maj take 3 words as input and produce a single word output. $\wedge$ stands for a 32 bit Bitwise AND operation while $\neg$ is the compliment operation. The Ch function always takes the working variables E, F and G as inputs while the Maj function always takes A, B and C as inputs. Variables A and E are the ones that need to be computed at each round. Functions $\sum_0$ and $\sum_1$ always take variables A and E as their input. We can

thus see some sort of symmetry in the message compression function that divides it into two parts. This symmetry is evident from figure 3.

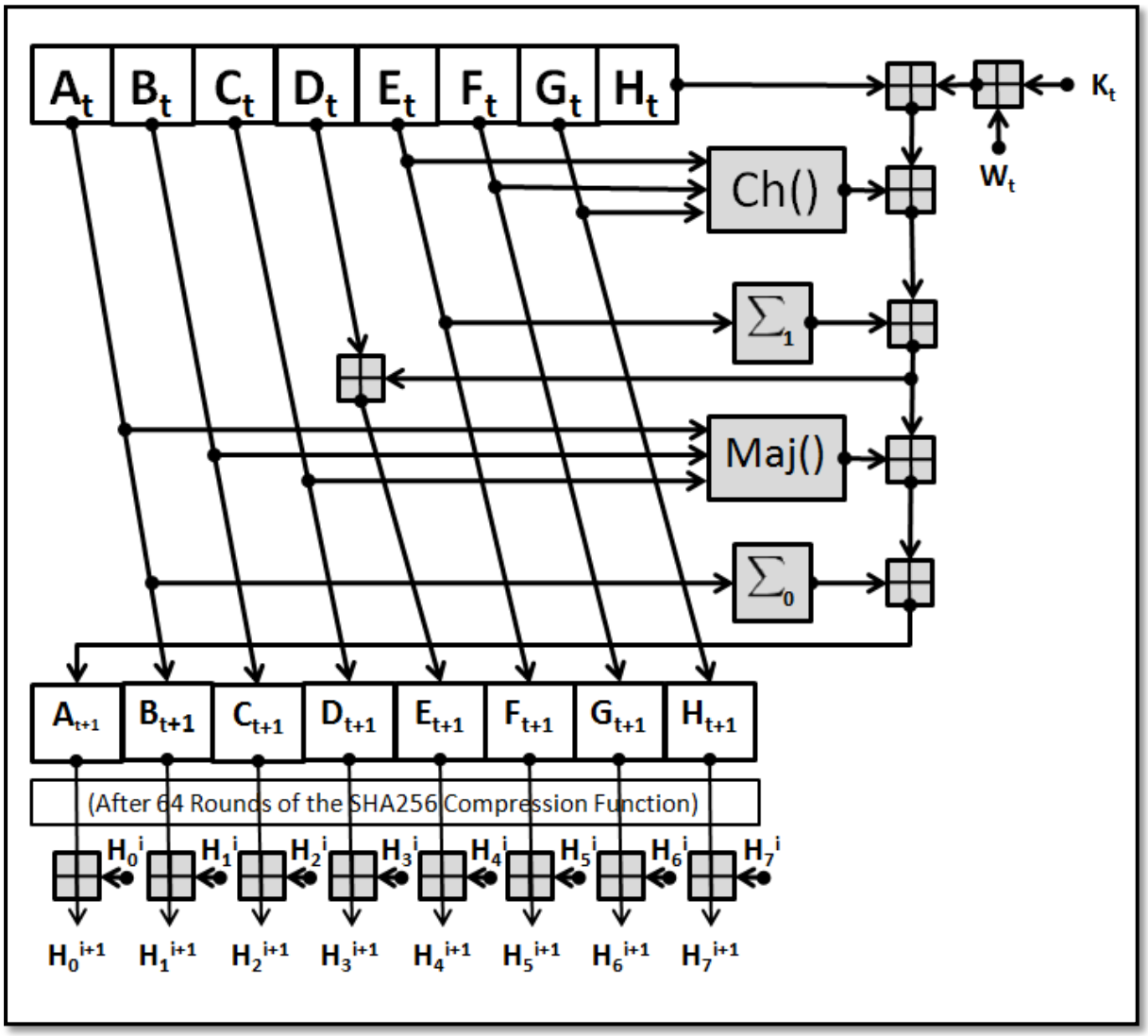

Figure 4: SHA256 Compression Function Along with the Final Additions.

The figure above represents a different look at the compression function but it conveys the same message. The point to take away from this figure is that after the compression function has been applied 64 times i.e. after the 64 rounds have been completed, the values contained in the working variables A to H are finally added to the 8 word data block that was fed to the compression function at the beginning. This value could either be the constant IV for SHA256 or an intermediate message digest. This is because of the fact that the SHA256 algorithm follows the Davies-Meyer construction where the input is added to the output at the end. Now, the intermediate/final hash is given by the following equation:

$$
\begin{aligned}
H_0^{i+1} &= A + H_0^i \\
H_1^{i+1} &= B + H_1^i \\
H_2^{i+1} &= C + H_2^i \\
H_3^{i+1} &= D + H_3^i \\
H_4^{i+1} &= E + H_4^i \\
H_5^{i+1} &= F + H_5^i \\
H_6^{i+1} &= G + H_6^i \\
H_7^{i+1} &= H + H_7^i
\end{aligned}
$$

Equation 10: Calculation of Intermediate/Final Hash Value

After all the message blocks including the final N^{th} message block has been processed in this manner, the final hash i.e. the 256 bit message digest of the message is represented in the following manner:

$$SHA256(M) = H_0^N \,||\, H_1^N \,||\, H_2^N \,||\, H_3^N \,||\, H_4^N \,||\, H_5^N \,||\, H_6^N \,||\, H_7^N$$

Equation 11: Resulting SHA256 Message Digest. Source: [26]

This 8 word data block (H_0^i - H_7^i) is the default constant SHA256 initialisation vector (IV) if the message was less than or equal to 512 bits (including the padding). If the length of the message (including padding) is greater than 512 bits, then this 8 word data block is the intermediate hash calculated of the previous 512 bit block. This arrangement where the intermediate hash value of the previous block is fed as IV to the hash computation of the next block is called the Merkle-Damgård construction. SHA256 is based on this construction called the Merkle-Damgård Paradigm and is built to be collision resistant as the underlying SHA256 compression function is collision resistant.

3.3 Analysis of the Operations Involved in SHA256

Below is an analysis of the number of different operations performed by the SHA256 algorithm on a 512 bit message block over 64 rounds. This analysis is imperative for the quantitative analysis done in the discussion part of this text that calculates the number of operations being saved due to the suggested optimisations. The table below summarises all the operations that take place in 64 rounds of the message compression function and the message scheduler as well as the 8 word addition that takes place at the end.

Additions (Mod 2^{32})	= (7*64) + (3*48) + 8 = 448 + 144 + 8 = **600**	(message compression) + (message scheduler) + (intermediate/final hash computation)
Bitwise Rotations (ROTR)	= (6*64) + (4*48) = 384 + 192 = **576**	$(\sum_0, \sum_1)$ + (σ_0, σ_1)
Bitwise Shifts (SHR)	= 2*48 = **96**	σ_0, σ_1
Bitwise AND ($\wedge$)	= 5*64 = **320**	Maj, Ch
Bitwise EX-OR ($\oplus$)	= (7*64) + (4*48) = 448 + 192 = **640**	(message compression) + (message scheduler)
Total Operations	= 600 + 576 + 96 + 320 + 640 = **2232**	

Table 2: Number of Operations in SHA256

The bitwise rotations and the bitwise shift operations involve just re-arranging the input word. It is said **[33]** that in the SHA256 architecture, the Mod 2^{32} additions are the most important and critical part that require many logic gates to implement. The additions involving 7 operands in the calculation of working variable A also forms the longest data path or the critical path. Hence, if we are able to cut down the number of additions being performed in the compression function even by a small amount, this will optimise the process to a great extent.

Chapter 4: Related Work - The Hardware Implementations and Optimisations of SHA256

The only practical way of a high speed SHA256 engine is to implement it in hardware be it either FPFAs or the recent technology of ASICs. Software implementations of Bitcoin mining used in CPU or GPU mining have become obsolete as they simply cannot compete with the hashing power of hardware implementations. These hardware implementations truly serve the meaning of a fast implementation and various hardware optimisations have been proposed over the years in order to increase their throughput and to reduce their power consumption. These optimisations, however, are aimed at the hardware implementation of the SHA256 hashing algorithm in general rather than SHA256 employed for Bitcoin mining. Most of these optimisations are aimed towards the longest data path or the critical path in the SHA256 core which is the calculation of working variable A in the message compression function that involves mod 2^{32} additions of 7 operands (see equation 8 in section 3.2.3). We shall now have a look at the various SHA256 hardware speedup proposals made.

4.1 SHA256 Hardware Optimisations

Many hardware implementations have been seen in the literature that are either FPGA **[15] [16] [25] [30] [39]** or ASIC designs **[21] [22] [33] [37] [47]**. These implementations designs contain one or a combination of the following optimisations so as to speed up the calculations i.e. the throughput of the SHA256 core. The main design difference for the hardware implementation of SHA256 lies in the trade-off between throughput and the area complexity which is measured in Gate Equivalents (GE). But, in our cases of Bitcoin mining, we typically have no area/space constrains and thus we shall concentrate on the throughput optimisations only. More the area, more is the throughput and lesser is the number of required clock cycles to perform the SHA256 computation.

4.1.1 Use of Carry-Save Adders (CSAs)

As mentioned before, the calculation of the working variable A for each round of the compression function forms the longest data path or the critical path in the SHA256 core.

This involves mod 2^{32} additions on 7 operands (K_t, W_t, H, $\sum_1(E)$, Ch(E, F, G), $\sum_0(A)$, Maj(A, B, C)). Architectures **[21] [22] [35] [36] [39]** employing Carry Save Adders (CSAs) minimise the delay caused by the carry propagation time by separating the sum and the carry paths. CSAs accept 3 operands as inputs and so, the working variable A can be computed using just 5 CSAs **[22]**. Having said that, CSAs require another 2-input adder for the recombination of the sum and carry paths. This 2 operand addition can either be performed by using CLAs i.e. Carry Look Ahead adders or by using CPAs i.e. Carry Propagation Adders. The net result of using CSAs for the critical path is that they reduce the carry propagation delay caused as compared to traditional CPAs used on the critical path.

4.1.2 Unrolling

Unrolled architectures **[20] [35] [36] [39]** reduce the number of clock cycles required to perform the SHA256 hash computation by implementing multiple rounds of the SHA256 compression function using combinational logic. These architectures help improve the throughput by optimising the data dependencies involved in the message compression function. Say if the SHA256 core was unrolled once, then this would effectively mean that the hash should be calculated in half the number of clock cycles. As a trade-off, unrolling the SHA256 core architecture comes at the cost of a decrease in the clock frequency and an increase in the area complexity.

4.1.3 (Quasi-) Pipelining

The goal of quasi-pipelining is to optimise the critical path and therefore increase the clock frequency. Quasi-pipelined SHA256 architectures **[21] [22] [36] [39]** use registers to break the long path or the critical path of the computation of the working variable A in the message compression function. Thus, such quasi-pipelines architectures allow higher data throughputs and higher frequencies of hash calculations by achieving very short critical paths. Pipelining is not as easy to achieve as it sounds due to the feedback associated due to the way in which the SHA256 compression function is designed. As a result, an external control circuitry is required such that the registers are enabled correctly.

4.1.4 Delay Balancing

Dadda et al. **[22]** have been the pioneers in hardware optimisations of SHA256 and they have also spent their research efforts on delay balancing along with the use of CSAs. Just as described earlier, a CLA is used to combine the sum and the carry paths output by the CSA but these sum and carry paths are first registered so that the CLA adder is removed from the critical path. This increases the throughput but this architecture requires additional control circuitry for the additional register introduced in the architecture.

4.1.5 Addition of K_t and W_t

Looking at figure 3 and figure 4, we can see that the addition of K_t and W_t can be performed independent of the message compression function. The architecture proposed in **[52]** uses this as an improvement by moving Kt + Wt to the message scheduler stage. This can be done because both W_t and K_t are available before and are independent of the other operands (see equation 8 in section 3.2.3). However, it is seen that quasi-pipelining architecture proposed by Dadda et. al. **[21] [22] [36]** performs a similar separation of the operands and the resulting critical path is even shorter than in **[52]**.

4.1.6 Operation Rescheduling

Architectures that employ operational rescheduling allow an efficient use of a pipelined structure without increasing the area complexity. This in turn allows higher throughputs. **[15] [16]** have claimed that they were able to reduce the critical path in a similar manner as unrolling techniques and gain a higher throughput without adding more area complexity.

Chapter 5: The Bitcoin Block Header Hashing Algorithm

The analyses and the details presented in the coming section have been collectively obtained from information provided in **[9] [19] [42]**. Important findings and understandings were also made by studying the open source Bitcoin mining program written in C called cgminer. The source code is available here **[34]**.

5.1 An Overview of the Bitcoin Block Header Hashing Algorithm

Mining devices use the Bitcoin Block Header Hashing Algorithm to find new blocks and thereby mine new Bitcoins. Looking from a purely technical perspective, the process of Bitcoin mining basically involves mining devices continuously calculating the double SHA256 hash of the Bitcoin block header and waiting for an output that would be accepted by the Bitcoin network. This section will emphasise on what constitutes this Bitcoin block header and how it is constructed. The construction of the Bitcoin block header will throw a light on how the data to be hashed actually enters the SHA256 hashing algorithm. It will also explain what part of this data typically remains constant throughout the mining process, what data changes but rather infrequently and what part of the data changes quite frequently.

Following the footsteps of **[19]**, the Bitcoin block header hashing algorithm is explained using a colour coded approach. Three colours viz. Green, Yellow and Red are used in order to explain the rate at which these values fluctuate relative to the process of Bitcoin mining. The green colour specifies that the value will either remain constant forever or for significantly long period of time. The yellow colour indicates that the value will change but rather quite infrequently i.e. relatively after some amount of time. The red colour indicates that this data value will change the fastest i.e. typically for every hash calculation. It needs to be pointed out that the colour coding and the data change frequencies mentioned are relative to the hashing speed of current hashing devices that are huge. The next figure shows the structure of the Bitcoin block header and how it is fed to the double SHA256 hashing algorithm in order to obtain a hash value that gets accepted by the Bitcoin network.

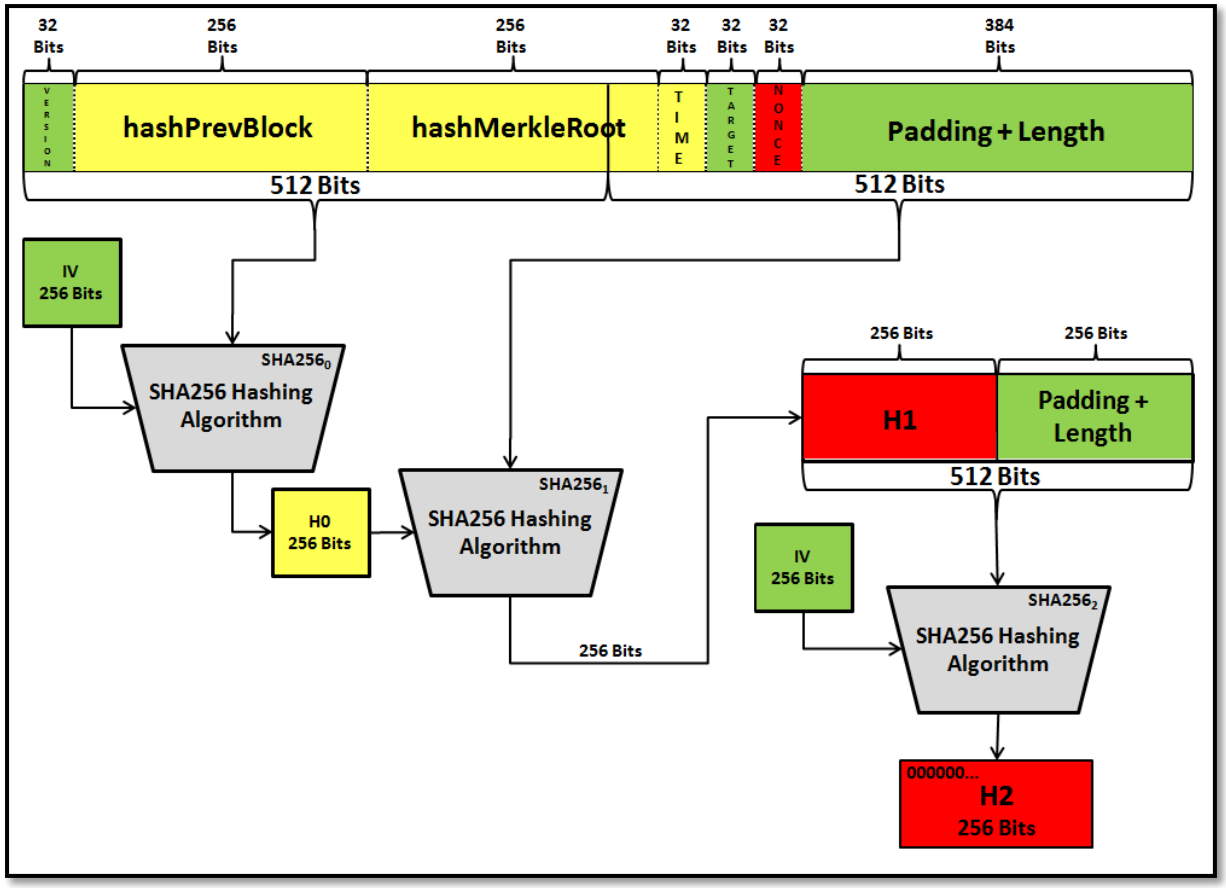

Figure 5: The Bitcoin Block Header Hashing Algorithm

From the figure, it is evident that the process of Bitcoin mining i.e. hashing this Bitcoin block header employs three applications of the SHA256 hashing algorithm. We shall name them $SHA256_0$, $SHA256_1$ and $SHA256_2$. These notations will henceforth be used throughout this text for addressing that particular application of the SHA256 hashing algorithm. The block header being greater than 512 bits in length, it is processed by two applications ($SHA256_0$, $SHA256_1$) of SHA256 (one 512 bit block at a time). $SHA256_0$ takes the first 512 bit block as input and after 64 rounds, produces the intermediate message digest H0. $SHA256_0$ takes the default IV on 256 bits that is described in **[26]** as well as in section 3.2. The default IV that $SHA256_0$ uses will be constant forever and hence it has been marked with the green colour. As the calculated intermediate message digest H0 depends on inputs marked with the yellow colour, H0 is also marked as yellow.

$SHA256_1$ uses H0 as its initialisation vector and takes the next 512 bits as its input block. The red coloured nonce is present in the input and hence, the final message digest, H1 produced by $SHA256_1$ is also marked as red. The process of hashing the block header does not stop

here. The final message digest H1 produced by $SHA256_1$ is applied through another SHA256 hashing which we name as $SHA256_2$. $SHA256_2$ takes the 256 bit block of H1 as its input message block and applies suitable padding (as explained in section 3.2.1.1) to make it a block of 512 bits. $SHA256_2$ being an additional application of SHA256 applied again, uses the same default IV as used by $SHA256_0$ which is marked as green. After 64 rounds of the compression function of $SHA256_2$, the final hash H2 is generated which for obvious reasons is marked as red in the earlier figure. H2 is then checked to see if it satisfies the current constraints of the Bitcoin protocol. If H2 does satisfy these constraints, the successful block with the correct nonce is broadcast immediately in the Bitcoin network for acceptance and to claim the mining reward. The Bitcoin mining process thus basically involves the below calculation repeated potentially billions of times with variable nonces:

$$H2 = SHA256(SHA256(Block\_Header))$$

Equation 12: Bitcoin Mining - Hashing the Block Header

The reader may question as to why an additional application of SHA256 is made at the end. One explanation **[7]** why Satoshi Nakamoto chose to have double SHA256 hashing is to prevent length extension attacks. The SHA256 hashing algorithm, like all hashes constructed using the Merkle-Damgård paradigm, is vulnerable to this attack. The length extension attack allows an attacker who knows SHA256(x) to calculate SHA256(x||y) without the knowledge of x. Although it is unclear how length extension attacks may make the Bitcoin protocol susceptible to harm, it is believed that Satoshi Nakamoto decided to play it safe and include the double hashing in his design. Another explanation **[6]** for this double hashing is that 128 rounds of SHA256 may remain safe longer if in the far future, a practical pre-image or a partial pre-image attack was found against SHA256.

Regardless of the reason behind it, what is important to know and to understand is that whenever a SHA256 hash is calculated in Bitcoin, it is a double SHA256 hash. Thus, a double hash of the block header is calculated and is then checked if the value of the hash conforms to the Bitcoin protocol proof-of-work constraints. The next section covers the details of the Bitcoin block header by explaining what each data block contains and the frequency by which these data change.

5.2 Details of the Bitcoin Block Header

The block header may also occasionally need to be updated while working on it during mining. It is important to know that it is the body of the block that contains the actual transactions and **NOT** the block header. All the transactions contained in the block are only hashed indirectly into the block header via the Merkle root (section 2.2.2). This ingenious method not only ensures the transaction integrity but another offshoot of this arrangement is that the time taken to hash the block header becomes independent of the number of transactions that it contains. The block header as described in the earlier figure and in **[9]** essentially contains the following fields (continuing the same colour coding scheme):

Field	Size	Description
Version	32 bits	Block version information that is based on the Bitcoin software version creating this block
hashPrevBlock	256 bits	The hash of the previous block accepted by the Bitcoin network
hashMerkleRoot	256 bits	Bitcoin transactions are hashed indirectly through the Merkle Root
Timestamp	32 bits	The current timestamp in seconds since 1970-01-01 T00:00 UTC
Target	32 bits	The current Target represented in a 32 bit compact format
Nonce	32 bits	Goes from 0x00000000 to 0xFFFFFFFF and is incremented after a hash has been tried
Padding + Length	384 bits	Standard SHA256 padding that is appended to the data above

Table 3: Bitcoin Block Header Fields Along With Their Brief Description

5.2.1 Version

This 32-bit value is an integer that represents the version of the rules that the Bitcoin software follows to create a new block. The current value is 2 and has changed ever since BIP0034[9] was accepted in July 2012. Before that, the value was 1. The point to take here is that this value can be considered as constant and is hence marked as green in the table above as well as in figure 5. The announcement that Version 1 blocks will soon be orphaned was made by Gavin Andresen, the lead core Bitcoin developer in his post **[5]** on Bitcointalk.

[9] Bitcoin Improvement Proposal BIP 0034 - https://en.bitcoin.it/wiki/BIP_0034

The BIP that has been accepted has implemented the rule that if 950 of the last 1,000 blocks are version 2 or greater, then reject all version 1 blocks in the community. **[19]** mentions that currently more than 90% of new blocks created are of version 2 and that the Bitcoin community will soon stop accepting blocks with version as 1.

5.2.2 hashPrevBlock

This is the 256 bit H2 of the previous block that was accepted by the Bitcoin network. By including this value in the new block header, the miner basically tries to further extend the longest proof-of-work chain as explained in section 2.1.3. It is important to know that the miner has to find a new block after the latest accepted block and he tries to be the first to solve the proof of work problem. The solution to the proof-of-work problem however, is **NOT** unique and is actually a race between different miners to be the first to solve and broadcast the new H2 that will be accepted by the network. If accepted, the miner will hence be awarded with the current mining reward of 25 BTC along with the transaction fees that were included in the individual transactions held by the block. As the Bitcoin protocol is designed such that a new block is generated by the network in approximately every 10 minutes, it is safe to assume that on average, hashPrevBlock needs to be updated after around every 10 minutes. For this reason, we have marked it with the yellow colour.

5.2.3 hashMerkleRoot

hashMerkleRoot is the 256 bit value of the Merkle Root as explained in section 2.2.2. Similar to hashPrevBlock, hashMerkleRoot will typically on average change in around 10 minutes time and hence even this is marked in yellow. There is another scenario where hashMerkleRoot changes and this will be explained ahead in section 5.2.6.

5.2.4 Timestamp

This 32 bit value is the current time in seconds since 1970-01-01 T00:00 UTC. The miner may have some flexibility of varying it to his advantage but this is very risky as there are only 600 seconds in that 10 minute window and every microsecond counts. Considering the hashing rate of current miners, 1 second is relatively a large timeslot and we have thus marked the timestamp field as yellow; indicating that it changes but relatively rarely.

5.2.5 Target

This is the same Target as explained earlier in section 2.1.4. The only difference is that this value is a compact version for it and is expressed in 32 bits rather than 256 bits. This is a particular sort of floating-point encoding that uses 3 bytes mantissa, the leading byte as exponent (where only the 5 lowest bits are used) and the base is 256 **[9]**. The target changes after every 2016 new blocks which takes about 2 weeks time. As the target changes in about two weeks' time, we have marked it as green.

5.2.6 Nonce

This 32 bit value is the only value in the block header that is the most volatile as it changes on every attempt of the double hash on the block header. We have thus marked the nonce field in red. The nonce starts at 0 and it is incremented strictly in a linear manner for each H2 attempted. One interesting question that needs to be brought up is that if one knows the current target, what would be the probability of finding H2 that will be accepted by the Bitcoin network? This probability **[10] [19]** is given by:

Probability = Target/2^{256} = 1/(Difficulty*2^{32})

With the current Difficulty[10] at the time of writing being 65750060,

Probability = 1/(65750060*2^{32}) = $\mathbf{2^{-57.97}}$

Hence, the average number of hashes that need to be tried to solve a block

= 1/Probability = $\mathbf{2^{57.97}}$

That been said, we know that there are only 2^{32} possible values for the nonce! This means that the nonce is probably going to overflow more often than not. If this happens, there is a provision in the Bitcoin protocol such that whenever the nonce overflows, the "extraNonce" portion of the generation transaction in the block is incremented which ultimately changes the Merkle Root. Once this updated Merkle Root is added to the block header, calculation commences again with nonce at 0 until an acceptable H2 is found. Else, this process is repeated. This is the second scenario in which hashMerkleRoot might change while a miner is solving for a new block. This value of 57.97 also means that H2 will need to start with 58

[10] Current Difficulty: http://blockexplorer.com/q/getdifficulty

or more 0s and also be less than the target so as to be accepted in the block chain by the Bitcoin network.

5.2.7 Padding + Length

For $SHA256_1$, padding + length are 384 bits long while for $SHA256_2$, it is 256 bits long. As the specification of SHA256 is known and the length of the input message to $SHA256_1$ and $SHA256_2$ is fixed i.e. 640 bits and 256 bits respectively; the padding + length field will always remain constant. We have hence marked these in green.

After comprehending all of this, the reader may argue that given all these fields, the same sequence of hashes will be generated by all miners and the miner with the most mining capability will always be able to solve the block first. This is in fact not true as it is almost impossible for two miners to end up with the same hashMerkleRoot. This is because each block has a unique transaction called the "Generation Transaction" or the "Coinbase Transaction". This transaction grants the mining reward and the transaction fees to the miner once the block is accepted by the network. As this generation transaction is unique, hashMerkleRoot is generally unique for all miners and every hash calculated by a miner has the same chance of solving the block as every other hash calculated in the entire Bitcoin network. Therefore, it can be said that the process of Bitcoin mining is analogous to a lottery draw where each participant has an equal chance of winning. But as people tend to buy more and more lottery tickets in order to have a better chance at winning the lottery, same is evident in the Bitcoin world where there is an arms race between miners to obtain mining devices with the fastest hashing rate. This is because a mining device with a faster hashing rate can make more attempts at solving the block in a given time. Winning the Bitcoin lottery is getting harder every two weeks as the network hash rate is constantly on the rise which is driving the Difficulty up as well. But as rightfully claimed in **[19]**, Bitcoin miners are now clever enough to participate in mining clusters or mining pools that helps to smooth their gains in mining and remove the lottery aspect from their earnings. Each miner is then paid out on the basis of the hashing rate contributed to the cluster/pool.

Chapter 6: SHA256 Algorithm Optimisations

6.1 Optimisation#1: The Calculation of H0 for $SHA256_0$

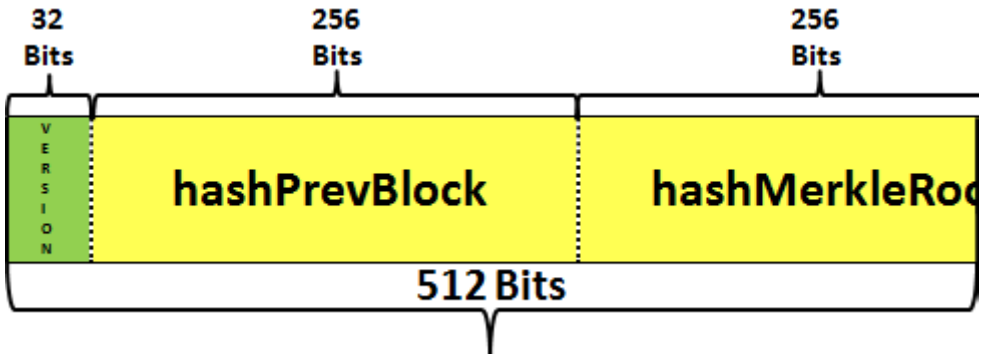

Figure 6: Input Message Block to $SHA256_0$

As evident from figure 5 and according to the Block header hashing algorithm, the hash value generated after the first application of the hash function ($SHA256_0$) is H0. Looking at an excerpt of figure 4 above, it is evident that H0 depends upon 32 bits of Version, 256 bits of hashPrevBlock and 224 bits of hashMerkleRoot. Version is marked as green and will remain constant throughout. So, hashPrevBlock and hashMerkleRoot are the only two variables involved. Thus, as rightfully pointed out in **[19]** and by independent observation, H0 needs to be calculated only once during the mining computation. Calculation of H0 therefore costs nothing and can be amortized over many computations with various nonces.

The 256 bits of hashPrevBlock will remain constant until someone else finds a new block. If a new block is found, the existing mining needs to be terminated. hashPrevBlock and hashMerkleRoot will obviously change and H0 will have to be calculated again. It is the responsibility of the miner to constantly check if a new block was found in the Bitcoin network. If the new block is yet to be found, hashMerkleRoot will change only when the nonce overflows (see section 5.2.6).

This optimisation is understandably trivial and it is believed that this optimisation logic should have already been implemented in most mining devices.

6.2 Optimisation#2: Early Rejection at Rounds 61 and 62 for $SHA256_2$

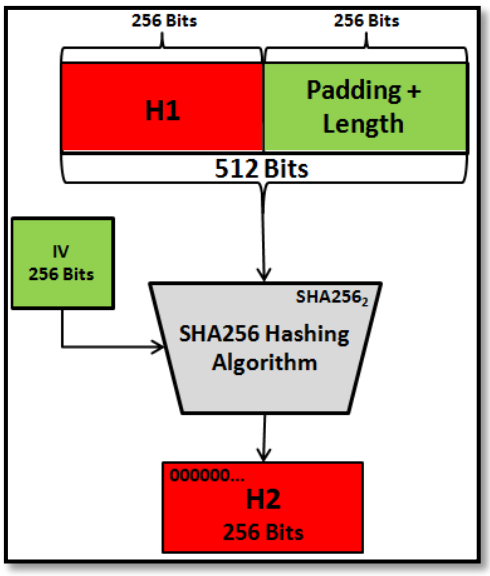

Figure 7: $SHA256_2$

[19] also presents the idea of an early rejection technique where during the application of $SHA256_2$ for the calculation of H2, due to the nature of the SHA256 algorithm, the necessary values to be checked if H2 < target can be obtained at rounds 61 and 62 itself. It is thus possible for mining devices to know in advance from the value of working variable E at rounds 61 and 62 if a particular nonce has produced the required number of 0s and whether H2 is less than the target. Hence for most of the time, there is no need to calculate rounds 63 and 64. In fact, with the pseudo code provided next, many times even round 62 need not be calculated for most of the time.

	A	B	C	D	E	F	G	H
t=59:	B6AE8FFF	FFB70472	C062D46F	FCD1887B	B21BAD3D	6D83BFC6	7E44008E	9B5E906C
t=60:	B85E2CE9	B6AE8FFF	FFB70472	C062D46F	961F4894	B21BAD3D	6D83BFC6	7E44008E
t=61:	04D24D6C	B85E2CE9	B6AE8FFF	FFB70472	948D25B6	961F4894	B21BAD3D	6D83BFC6
t=62:	D39A2165	04D24D6C	B85E2CE9	B6AE8FFF	FB121210	948D25B6	961F4894	B21BAD3D
t=63:	506E3058	D39A2165	04D24D6C	B85E2CE9	5EF50F24	FB121210	948D25B6	961F4894

Figure 8: Last 5 Rounds of SHA256 (Example). Source: [43]

We know that by the end of 64 rounds, in order for the new block to be accepted by the network, H + 0x5BE0CD19 must be equal to 0x00000000. With the current target, little_endian(G + 0x1F83D9AB) must have a value less than target32 (explained ahead). It is evident from the figure above that the two values to be checked are obtained at round 61 (E

at t=60) and at round 62 (E at t=61). The logic to be implemented in mining devices is explained below in the form of a pseudo code:

```
At t=60 if (E + 0x5BE0CD19 = 0x00000000)
{
      calculate_round62()
      At t=61 if (little_endian(E + 0x1F83D9AB) <= target32)
      {
            calculate_remaining_rounds()
      }
}
else next_nonce()
```

Here, target32 is the 32 bit value (bits 32-63) of the current target. Hence, for most of the cases, the nonce can be early rejected at round 61 thus saving the calculation of three rounds of the SHA256 compression function. For some cases, round 62 will also need to be calculated and as explained in the pseudo code above, early rejection of the nonce can be performed at this round as well. In very few cases all 64 rounds will need to be calculated.

6.3 Optimisation#3: First 3 Rounds of $SHA256_1$

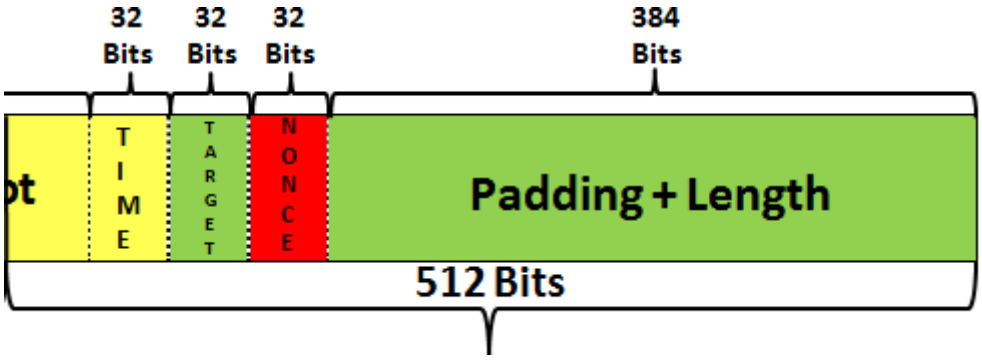

Figure 9: Input Message Block to $SHA256_1$

Referring to another excerpt of figure 5 above and as per the message schedule, the 32 bit values of W_t to enter the compression function of $SHA256_1$ at rounds 1, 2 & 3 will be the last 32 bits of hashMerkleRoot, timestamp and target respectively. Two of these values have been marked yellow indicating that they will change but relatively very slowly. Target has been marked green as it will change relatively after a long time (2016 blocks i.e. after 2 weeks time).

As explained earlier, hashMerkleRoot will change only when someone else finds a new block or when the nonce overflows. Even if we assume a very modest hashing rate of our mining device to be 10 GH/s, as 10G >> 2^{32}, we can safely claim that the nonce would overflow before needing to increment the timestamp by 1. The timestamp will thus be updated only when hashMerkleRoot changes. Hence, round 1, 2 & 3 calculations for SHA2561 need to be calculated only once i.e. initially or when hashMerkleRoot/timestamp change. The values of the working variables A-H at the end of round 3 can be stored as they will remain constant for different nonces. Thus, for every new nonce, round calculation can resume from round 4 where the nonce enters the compression function. This was independently observed and has also been pointed out in **[19]**.

6.4 Optimisation#4: Round 4 Incremental Calculations for $SHA256_1$

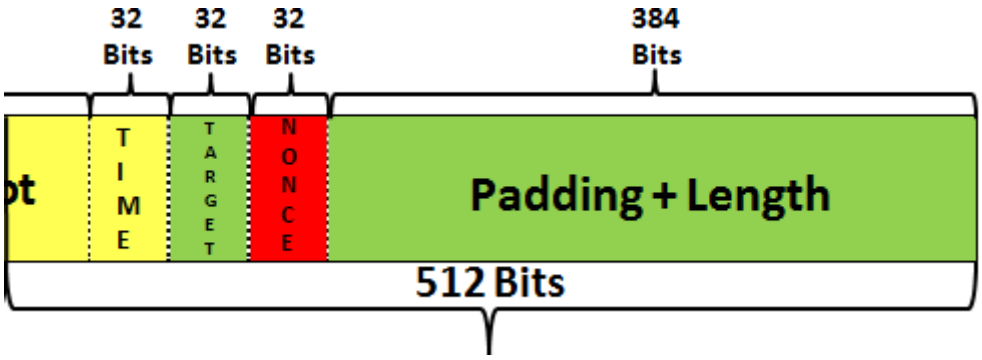

Figure 10: Input Message Block to $SHA256_1$ (Nonce)

As seen from the figure above & the message scheduler equation in section 3.2.2, the 32 bit nonce enters at round 4 for $SHA256_2$. Recall the improvement claimed in the previous section that the first 3 round computations only need to be done once and the result can be used for different nonces. Also, recall the equations **[26]** for variables A and E in the compression function:

$$A = H + \sum_1(E) + Ch(E, F, G) + \sum_0(A) + Maj(A, B, C) + K_t + W_t$$

$$E = D + H + \sum_1(E) + Ch(E, F, G) + K_t + W_t$$

Equation 13: Working Variables A and E

Now, for round 4 of $SHA256_1$, W_3 will be the 32 bit nonce. It was observed that for round 4, all variables except W_t in both the equations above remain constant. This is because the

values of all the working variables are of round 3 and as claimed in the previous optimisation, these values will remain constant for most of the time even for different nonces.

Hence, it can be claimed that the entire round 4 calculation only needs to be completed once for the initial nonce of 0x00000000. For future nonces, round 4 values of the variables A & E can be trivially calculated by incrementing their values from the previous nonce by 1. The values for the rest of the working variables will be the same as their values from all the previous nonces. Thus, by this optimisation, an entire round of the compression function is reduced to two trivial increments. This property was tested practically by running the SHA256 algorithm source code taken from **[23]** and was indeed found to be true. The modified source code can be found towards the end of this text in appendix B. The round 4 values of working variables A & E obtained on execution of the code for nonces 0x00000000 to 0x00000005 are as follows:

Nonce	A	B	C	D	E	F	G	H
0x00000000	c14c28c**6**	fdd86aa7	1184d36	2703413e	346785c**7**	c1abdbc7	8f925db9	a4b56f21
0x00000001	c14c28c**7**	fdd86aa7	1184d36	2703413e	346785c**8**	c1abdbc7	8f925db9	a4b56f21
0x00000002	c14c28c**8**	fdd86aa7	1184d36	2703413e	346785c**9**	c1abdbc7	8f925db9	a4b56f21
0x00000003	c14c28c**9**	fdd86aa7	1184d36	2703413e	346785c**a**	c1abdbc7	8f925db9	a4b56f21
0x00000004	c14c28c**a**	fdd86aa7	1184d36	2703413e	346785c**b**	c1abdbc7	8f925db9	a4b56f21
0x00000005	c14c28c**b**	fdd86aa7	1184d36	2703413e	346785c**c**	c1abdbc7	8f925db9	a4b56f21

Table 4: Round 4 Optimisation for SHA2561: Code Execution Results

For the above execution of code, the value of the last 32 bits of hashMerkleRoot, timestamp and target i.e. W_0, W_1 and W_2 respectively, were kept constant at 0xFFFFFFFF. Hence, from the above table it is evident that round 4 values of A and E can be trivially computed by incrementing their previous nonce values by 1. It can also be seen that all the remaining working variables remain constant for different nonces at round 4. Now with this optimisation in mind and the one before, we can now claim that for most of the time, the computation of $SHA256_1$ can be directly started from round 5!

6.5 Optimisation#5: Saving Additions Using the Long Trail of 0s for $SHA256_1$ and $SHA256_2$

According to the Bitcoin block header hashing algorithm discussed in section 5.1, we can take advantage of the fact that the length of the input given to $SHA256_1$ and $SHA256_2$ never changes. Since the input never changes, as explained in section 3.2.1.1, regarding the padding scheme of SHA256, we can pin point exactly what data is contained & where in the padding of the input to $SHA256_1$ as well as $SHA256_2$. Recall the padding equation of the SHA256 pre-processing stage mentioned in section 3.2.1.1. We shall use that equation to calculate the exact value of the inputs to $SHA256_1$ and $SHA256_2$. The padding equation **[26]** is as follows:

$$l + 1 + k = 448 \bmod 512$$

Looking back at figure 5 and the block header hashing algorithm, we can conclude that the length of the message for $SHA256_1$ is 640 bits (32+256+256+32+32+32). Hence, by the above equation,

$$k = 1024 - (640 + 1 + 64) = 319$$

The input to SHA2561 hence contains 640 bits of the message, followed by bit “1”, then 319 zero bits and finally the message length (640 = 0x00000280) expressed using 64 bits. Similarly for SHA2562,

$$k = 512 - (256 + 1 + 64) = 191$$

The input to SHA2562 hence contains 256 bits of the message, followed by bit “1”, then 191 zero bits and finally the message length (256 = 0x00000100) expressed using 64 bits. Based on these calculations, the following table contains the values contained in the inputs for $SHA256_1$ and $SHA256_2$ along with the round in which they enter the message compression algorithm:

$SHA256_1$ (For H1)		
Round (t)	32 bit W_t (In Hex)	Description
0	XXXXXXXX	Last 32 Bits of hashMerkleRoot
1	XXXXXXXX	Timestamp
2	XXXXXXXX	Target
3	XXXXXXXX	Nonce (00000000 to FFFFFFFF)
4	0x80000000	Padding Starts
5	0x00000000	|
6	0x00000000	|
7	0x00000000	|
8	0x00000000	|
9	0x00000000	|
10	0x00000000	|
11	0x00000000	|
12	0x00000000	|
13	0x00000000	Padding Ends
14	0x00000000	Length 1
15	0x00000280	Length 2

$SHA256_2$ (For H2)		
Round(t)	32 bit W_t (In Hex)	Description
0	XXXXXXXX	$H1_0$
1	XXXXXXXX	$H1_1$
2	XXXXXXXX	$H1_2$
3	XXXXXXXX	$H1_3$
4	XXXXXXXX	$H1_4$
5	XXXXXXXX	$H1_5$
6	XXXXXXXX	$H1_6$
7	XXXXXXXX	$H1_7$
8	0x80000000	Padding Starts
9	0x00000000	|
10	0x00000000	|
11	0x00000000	|
12	0x00000000	|
13	0x00000000	Padding Ends
14	0x00000000	Length 1
15	0x00000100	Length 2

Table 5: W_t Values for the First 16 Rounds ($SHA256_1$ and $SHA256_2$)

From the table it is evident that rounds 6 to 15 (10 rounds) for $SHA256_1$ and rounds 10 to 15 (6 rounds) for $SHA256_2$ will always have W_t = 0x00000000. It is also worth noticing that the values follow the same colour coding as mentioned earlier. All variable values have been shown as 0xXXXXXXXX. Mining devices can take advantage of these long trails of 0s as for the rounds mentioned above, we can potentially save an addition per round. Mining devices can implement some logic where for the mentioned rounds; the value of K_t can be directly fed instead of the value of $K_t + W_t$. Hence with this optimisation, 10 additions per nonce can be saved for $SHA256_1$ and 6 additions per nonce can be saved for $SHA256_2$. Therefore, a

total of 16 additions can be saved per nonce using this optimisation. As nonces are in billions, this improvement is definitely non-negligible and will aid in faster and more efficient Bitcoin mining. It needs to be mentioned that this optimisation wouldn't have been possible for generic SHA256 hashing as the length of the message is always variable. In Bitcoin mining, we are taking advantage of the fact that the length of the message will always remain constant and hence saving a non-negligible amount of additions (16 to be exact) per nonce. It is also important to remark that there is a possibility for a similar optimisation in $SHA256_0$ during the calculation of H0. This is due to the presence of the trail of 0s in hashPrevBlock. However, referring to optimisation#1 in section 6.1, we now know that H0 needs to be calculated only once. Hence, such optimisation for SHA2560 doesn't really have a non-negligible impact on the mining device's throughput or power consumption.

6.6 Optimisation#6: Saving Additions with Hard Coding

$SHA256_1$ (For H1)		
Round(t)	32 bit W_t (In Hex)	Description
4	0x80000000	Padding Starts
15	0x00000280	Length 2

$SHA256_2$ (For H2)		
Round(t)	32 bit W_t (In Hex)	Description
8	0x80000000	Padding Starts
15	0x00000100	Length 2

Table 6: Where $W_t + K_t$ Can Be Hardcoded ($SHA256_1$ and $SHA256_2$)

From the table above, it is observed that the value of W_t for $SHA256_1$ will always be 0x80000000 and 0x00000280 for rounds 5 and 16 respectively. Similarly for $SHA256_2$, the value of Wt for rounds 9 and 16 will always be 0x80000000 and 0x00000100 respectively. We can take advantage of this fact and change the table of constants for $SHA256_1$ and $SHA256_2$ with values calculated as follows[11]:

[11] Refer to the appendix for the SHA256 round specific constants (K_t)

- For $SHA256_1$, at round 16, $W_{15}+K_{15}$ can be hardcoded as 0x00000280+0xc19bf174=**0xc19bf3f4**. The same is true in Round 16 for $SHA256_2$ where $W_{15}+K_{15}$ can be hardcoded as 0x00000100+0xc19bf174=**0xc19bf274**.

- A similar technique can be applied to Round 5 for $SHA256_1$ and Round 9 for $SHA256_2$. Hardcode with 0x80000000+0x3956c25b=**0xb956c25b** for $SHA256_1$ and 0x80000000+0xd807aa98=**0x5807aa98** for $SHA256_2$.

The point in doing this is that we are saving 4 more additions ($W_t + K_t$) per nonce. These new constants can be updated in the constants table K_t for $SHA256_1$ as well as $SHA256_2$. The new, Bitcoin specific constants are provided in the table below. It is important to remember that the rest of the constants still remain the same.

$SHA256_1$ (For H1)		
Round(t)	Previous value for K_t	New Value for K_t
4	0x3956c25b	**0xb956c25b**
15	0xc19bf174	**0xc19bf3f4**

$SHA256_2$ (For H2)		
Round(t)	Previous value for K_t	New Value for K_t
8	0xd807aa98	**0x5807aa98**
15	0xc19bf174	**0xc19bf274**

Table 7: New Values for Bitcoin Specific Constants ($SHA256_1$ and $SHA256_2$)

This improvement combined with the previous optimisation in section 7.5 will now allow mining devices to save 20 additions per nonce. This is indeed a significant amount of savings in calculations. It can be speculated that having different table of constants for different applications of the SHA256 algorithm would be sort of an overhead but this overhead is negligible towards the number of additions being saved per round. The importance of these savings can be further emphasised from the fact that for each nonce, 2X64 rounds of SHA256 hashing is performed by mining devices. These mining devices frequently overflow the nonce which typically goes from 0 to 2^{32} which is as large as 4294967296!

6.7 Optimisation#7: Message Scheduler Bypass for Certain Rounds

This optimisation is an offshoot of the previous two optimisations viz. section 6.5 and 6.6. With these two optimisations in place, the compression function does not depend on the message scheduler for some particular rounds. At these rounds, calculations can thus be

made directly by the compression function without waiting for the message scheduler. There will be no propagation delay as the value of K_t will be directly fed to the compression function. The rounds for which this optimisation is possible are mentioned below:

For $SHA256_1$	Rounds 5 to 16 (12 in total)
For $SHA256_2$	Rounds 9 to 16 (8 in total)

It is worth mentioning that the message scheduler will still have to keep track of all W_t for future round values.

6.8 Optimisation#8: Constant Message Schedule for $SHA256_1$

For Round 17 of $SHA256_1$, W_{16} need not be calculated most of the times as it is observed that it will mostly remain constant and independent of the nonce. The proof of which is mentioned below:

For $16 \leq t \leq 63$, we have,

$W_t = \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-5}) + W_{t-16}$

Therefore, $W_{16} = \sigma_1(W_{14}) + W_9 + \sigma_0(W_1) + W_0$

Hence, $W_{16} = 0 + 0 + \sigma_0(W_1) + W_0$

Equation 14: Calculation for W_{16}

This is because W_{14} and W_9 will always be equal to 0x00000000 for $SHA256_1$. This means that W_{16} only depends on W_1 (Timestamp) and W_0 (Last 32 bits of hashMerkleRoot). This in turn means that W_{16} will only have to be calculated once and it will remain constant even for different nonces. Like earlier optimisations, W_{16} will have to be calculated again after the timestamp gets incremented or if hashMerkleRoot changes. But this will relatively be a very uncommon event.

Similarly, for Round 18, W_{17} need not be calculated most of the times as it is observed that it will mostly remain constant and independent of the nonce. The proof of which is mentioned next:

For $16 \leq t \leq 63$, we have,

$W_t = \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-5}) + W_{t-16}$

Therefore, $W_{17} = \sigma_1(W_{15}) + W_{10} + \sigma_0(W_2) + W_1$

Hence, $W_{17} = \sigma_1(0x00000280) + 0 + \sigma_0(W_2) + W_1$

Equation 15: Calculation for W_{17}

This is because W_{15} and W_{10} will always be equal to 0x00000280 and 0x00000000 respectively for $SHA256_1$. This means that W_{17} only depends on W_2 (Target) and W_1 (Timestamp). This in turn means that W_{17} will only have to be calculated once and it will remain constant even for different nonces. Like earlier optimisations, W_{17} will have to be calculated again after the timestamp gets incremented. But this will relatively be a very uncommon event. Moreover, there is also a short cut method to calculate W_{17} after the timestamp is incremented by 1. The mining device will just have to be increment the previous value of W_{17} by 1 after the timestamp gets incremented by 1. The claims made in this optimisation were confirmed practically by executing the message scheduler code for different nonces. The source code is available in appendix C and the results of this are posted below:

W_0	0xffffffff	0xffffffff	0xffffffff	0xffffffff	0xffffffff
W_1	0xffffffff	0xffffffff	0xffffffff	0xffffffff	0xffffffff
W_2	0xffffffff	0xffffffff	0xffffffff	0xffffffff	0xffffffff
W_3	0x00000000	0x00000001	0x00000002	0x00000003	0x00000004
W_4	0x80000000	0x80000000	0x80000000	0x80000000	0x80000000
W_5	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
W_6	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
W_7	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
W_8	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
W_9	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
W_{10}	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
W_{11}	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
W_{12}	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
W_{13}	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
W_{14}	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
W_{15}	0x00000280	0x00000280	0x00000280	0x00000280	0x00000280
W_{16}	0x1ffffffe	0x1ffffffe	0x1ffffffe	0x1ffffffe	0x1ffffffe
W_{17}	0x210ffffe	0x210ffffe	0x210ffffe	0x210ffffe	0x210ffffe

Table 8: Code Execution Results for Constant W_t with Different Nonces

6.9 Optimisation#9: Incremental Message Schedule Calculation at Round 20 for $SHA256_1$

For Round 20, W_{19} for most of the time can be calculated without the message scheduling algorithm by simply incrementing the W_{19} value from the previous nonce by 1. The proof of which is mentioned below:

$$W_{19} = \sigma_1(W_{17}) + W_{12} + \sigma_0(W_4) + W_3$$

$$\text{Hence, } W_{19} = \sigma_1(W_{17}) + 0 + \sigma_0(0x80000000) + W_3$$

Equation 16: Calculation for W_{19}

W_{17} (as explained in the previous section) will remain constant for most of the time. W_{12} and W_4 will always remain constant as explained earlier with values 0x00000000 and 0x80000000 respectively. Thus, W_{19} only depends on the value of W_3 which happens to be the nonce. Thus with every increment of the nonce, W_{19} can be directly calculated by incrementing W_{19} from the previous nonce by 1. A noteworthy remark is that W_{19} will have to be calculated again whenever the timestamp is incremented. This is because W_{17} will change with the timestamp. Using this optimisation, we are saving an entire message schedule calculation for a round and reducing it to just one increment. The source code is available in appendix C and the results of that code execution are given below:

W_0	0xffffffff	0xffffffff	0xffffffff	0xffffffff	0xffffffff
W_1	0xffffffff	0xffffffff	0xffffffff	0xffffffff	0xffffffff
W_2	0xffffffff	0xffffffff	0xffffffff	0xffffffff	0xffffffff
W_3	0x00000000	0x00000001	0x00000002	0x00000003	0x00000004
W_{19}	0x1108b75**9**	0x1108b75**a**	0x1108b75**b**	0x1108b75**c**	0x1108b75**d**

Table 9: Code Execution Results for W_{19} with Different Nonces

6.10 Optimisation#10: Saving Additions by Dynamic Hard Coding for $SHA256_1$

This optimisation is an offshoot of the combined effect of optimisations mentioned in sections 6.6, 6.7, 6.8 and 6.9. With the results obtained from the optimisations presented in these sections, it can be claimed that once W_t is calculated for rounds 17, 18 & 20 for the initial nonce of 0x00000000, it is possible to predict their values for the next nonces. W_{16} and W_{17} will remain constant while W_{19} can be calculated by a mere increment. As they are

predictable, the table of constants can be dynamically updated after the initial calculation has been performed. The table of constants can be populated with the following values:

Dynamically hardcoded new values:

K_{16} = 0xXXXXXXXX + 0xe49b69c1

K_{17} = 0xXXXXXXXX + 0xefbe4786

K_{19} = 0xXXXXXXXX + 0x240ca1cc

Equation 17: Calculation of Dynamic Hard coding Values for K_{16}, K_{17} and K_{19}

Here, 0xXXXXXXXX represents the variable values of W_t at t = 16, 17 and 19 that technically need to be calculated only once. The calculations for K_t mentioned above will only need to be done once and then replace the original constants for those rounds. By doing so, the mining devices will not have to perform the additions of $W_t + K_t$ for rounds 17, 18 and 20 as well.

For rounds 17 and 18, the new value of K_t can be fed directly to the message compression function instead of $W_t + K_t$ until the values of hashMerkleRoot and Timestamp remain constant. As for round 20, the new value of K_t will have to be incremented by 1 for every increment in the nonce and fed directly to the message compression function. Doing so will save 3 additions per nonce. Also, as explained earlier, directly feeding K_t instead of $W_t + K_t$ will also reduce the propagation delay involved as the compression function can be directly fed with the needed values without the need of the message scheduler. Hence, combining these results with our previous saving of 20 additions per nonce, we can now claim of saving 23 additions per nonce.

Detailed quantitative analyses of the optimisations presented in this chapter are presented in the discussion section that follows next. We shall make use of the SHA256 bitwise operations analysis presented in section 3.3 to decide the number of bitwise operations saved by the optimisation suggestions made in this chapter. By doing do, the reader will have an idea of the impact of these improvements in the throughput and power consumption of Bitcoin mining devices.

Chapter 7: Discussion

7.1 Analysis of the Savings Made in Bitcoin Mining Calculations

The optimisations suggested in the previous chapter need to be analysed in order to quantitatively determine the savings introduced in the calculations. We will assume that optimisation#1 in section 6.1 has already been implemented in most Bitcoin mining devices and thus, we can arrive at a decision that Bitcoin mining essentially involves a dual application of the SHA256 hashing algorithm ($SHA256_1$ and $SHA256_2$) rather than a triple application of SHA256 ($SHA256_0$, $SHA256_1$ and $SHA256_2$). Therefore Bitcoin mining essentially involves applying SHA256 with a factor of 2 i.e. the SHA256 algorithm applied twice to the Bitcoin block header in order to obtain H2. Recalling the calculations performed in section 3.3, we shall perform similar calculations in order to determine the total number of rounds and/or operations saved with the optimisations suggested. We shall perform separate calculations of the savings made in $SHA256_1$ and $SHA256_2$. Also, these savings will be calculated on a per-suggested-optimisation basis so that each saving made is clearly understood by the reader. We follow that up with a summary of the savings. The table below performs the calculations:

SHA256 Application	Optimisation	Calculations Saved
$SHA256_0$	#1 - The Calculation of H0 for $SHA256_0$	None
$SHA256_1$	#3 - First 3 Rounds of $SHA256_1$	SHA256 Rounds: 3
	#4 - Round 4 Incremental Calculations for $SHA256_1$	SHA256 Rounds: 1
	#5 - Saving Additions Using the Long Trail of 0s for $SHA256_1$	Mod 2^{32} additions: 10
	#6 - Saving Additions with Hard Coding	Mod 2^{32} additions: 2
	#8 - Constant Message Schedule for $SHA256_1$	2 calculations of Message Scheduler Mod 2^{32} additions: 3*2=6 Bitwise Rotations: 4*2=8 Bitwise Shifts: 2*2=4 Bitwise AND: 0 Bitwise EX-OR: 4*2=8

	#9 - Incremental Message Schedule Calculation at Round 20 for $SHA256_1$	1 calculation of Message Scheduler Mod 2^{32} additions: 3*1=3 Bitwise Rotations: 4*1=4 Bitwise Shifts: 2*1=2 Bitwise AND: 0 Bitwise EX-OR: 4*1=4
	#10 - Saving Additions by Dynamic Hard Coding for $SHA256_1$	Mod 2^{32} additions: 3
$SHA256_2$	#2 - Early Rejection at Rounds 61 and 62 for SHA2562	SHA256 Rounds: 3
	#5 - Saving Additions Using the Long Trail of 0s for $SHA256_2$	Mod 2^{32} additions: 6
	#6 - Saving Additions with Hard Coding	Mod 2^{32} additions: 2

The table that now follows is a summary of the total savings introduced by these algorithm optimisations suggested in this thesis:

$SHA256_1$	SHA256 Rounds: 4 Mod 2^{32} additions: 24 Bitwise Rotations: 12 Bitwise Shifts: 6 Bitwise AND: 0 Bitwise EX-OR: 12
$SHA256_2$	SHA256 Rounds: 3 Mod 2^{32} additions: 8 Bitwise Rotations: 0 Bitwise Shifts: 0 Bitwise AND: 0 Bitwise EX-OR: 0

Table 10: Summary of Savings Made Due to the Algorithm Optimisations

We shall now calculate the average number of different operations calculated per round of the SHA256 Hashing Algorithm. We calculate the average as not every round has the same number of operations. This is because the message scheduler calculations start from round 17 (i.e. t=16) onwards. The table below mentions the details of the average operations per round of SHA256. This calculation needs to be done as it will provide us with an approximation of converting the saved operations to SHA256 rounds. Knowing this will allow us to calculate a constant Savings Factor as compared to applying SHA256 twice.

Additions (Mod 2^{32})	7 + (3*48/64) = 7 + (3*0.75) = 9.25
Bitwise Rotations (ROTR)	6 + (4*48/64) = 6 + (4*0.75) = 9
Bitwise Shifts (SHR)	2*48/64 = 1.5
Bitwise AND (∧)	5
Bitwise EX-OR (⊕)	7 + (4*48/64) = 7 + (4*0.75) = 10

Table 11: Average Operations per Round of SHA256

With the above calculations in mind, we first calculate the savings based on the complete round computations that we have saved. For most of the time, we essentially need to compute only 60 out of 64 rounds of $SHA256_1$ and 61 out of 64 rounds of $SHA256_2$. Hence:

Savings Factor = 60/64 + 61/64 ≈ 0.9375 + 0.9535 ≈

1.891

Equation 18: Savings Factor Initial Calculation

The savings factor calculated till now means that instead of computing 2*SHA256, miners will now need to compute only 1.891*SHA256 in order to calculate the same H2. Now, in order to include the other granular savings, we will need to make an approximation opting for a simplifying assumption that all operations take the same time to execute i.e. they have the same latency. Although this isn't accurately true, we are making this assumption to simplify our calculations whilst keeping the results of our calculations reasonably close to the truth. It was seen from **[27]** that almost all the involved operations use the same amount of clock cycles in all types of CPUs. We now try and include the other savings introduced with the rest of the suggested optimisations:

For $SHA256_1$: ((24/9.25)+(12/9)+(6/1.5)+0+(12/10))/5 = (2.5946+1.334+3+0+1.2)/5 ≈ 1.6257

For $SHA256_2$: ((8/9.25)+0+0+0+0)/5 ≈ 0.8649/5 ≈ 0.173

We now include these savings in our calculation for our Savings Factor:

Savings Factor = (60-1.6257)/64 + (61-0.173)/64 ≈ 0.912 + 0.9504 ≈

1.8624

Equation 19: Savings Factor Final Calculation

What this number effectively means is that by implementing the suggested optimisations, miners will now be able to calculate H2 not by applying SHA256 twice but rather applying SHA256 1.8624 times which is a significant reduction considering the fact that around 500GigaHashes/sec can be calculated by a single ASIC mining device and 507.38 TeraHashes/sec are currently being calculated the entire Bitcoin network!

7.2 Summary, Limitations and Future Work

We can now summarize the contribution of this thesis by mentioning that by optimising the SHA256 hashing algorithm, we were able to improve the Bitcoin mining process from 2xSHA256 to approximately 1.8624xSHA256. This effectively means that Bitcoin mining devices can now achieve the same hashing rate with lower power consumption or mining devices can have a higher hash rate with the same power consumption as earlier.

Having said that, it can be acknowledged that the Savings Factor of 1.8624 determined in this thesis is not completely accurate. We have crudely tried to represent the operations in terms of SHA256 rounds and while we have a figure quite close to accuracy, for a truly accurate figure, we will need to implement these optimisations. For a perfectly accurate determination of the computational improvements introduced with these optimisations, ideally, mining devices employing both off-the-shelf and the optimised version of SHA256 will need to be implemented on a common platform. They would then need to be benchmarked and their performances would need to be compared. This would give a measure of the real world performance of the implemented optimisations and give us a more accurate Savings Factor. It can be claimed that the Savings Factor presented in this thesis will be reasonably close to the one calculated after implementation, benchmarking and comparison.

It would also be important to mention that much of the optimisations were designed and suggested by taking advantage of the fixed or predictable nature of the input given to the hashing function in Bitcoin mining. Hence, generic SHA256 hashing cannot be performed once these improvements have been implemented. Also, much of the omprovements have been more concentrated on $SHA256_1$ and much less optimisations have been suggested for $SHA256_2$. Perhaps more complex methodologies like logic minimisation techniques and circuit optimisations **[13] [14] [18]** could be employed to achieve further optimisations.

Possibly some progress in more optimisations could also be made by applying the research findings on analyses of simplified and step-reduced SHA256 **[38] [40] [48]**. Maybe such techniques can even make use of the fixed or predictable nature of the Bitcoin block hashing algorithm process and perhaps be able to achieve more optimisations that are specific for Bitcoin mining.

This thesis was more of a theoretical approach towards studying and optimising SHA256 for Bitcoin mining. Although an open source SHA256 program **[34]** in C was utilised in order to verify the claims made in the optimisations suggested, we would need these improvements to be implemented in hardware and its performance will have to be compared against a standard SHA256 core used for Bitcoin mining. This shall be left as future work.

Moreover, how these optimisations will work in tandem with the existing hardware optimisations of SHA256 mentioned in chapter 4 needs to be determined as well. A critical evaluation of these algorithm optimisations will need to be performed and their compatibility with existing hardware optimisations will need to be assessed as well. Based on the results of these findings, either of the two i.e. algorithm optimisations or hardware optimisations will need to be tweaked or amended as necessary.

Chapter 8: Conclusion

The foremost aim of this thesis was to propose optimisations in the SHA256 hashing algorithm that were specific to Bitcoin mining. This aim was motivated by the fact that many hardware based optimisations in SHA256 hardware implementations have already been suggested but they have been aimed at the SHA256 hashing algorithm in general.

With that in mind, the primary contribution made in this thesis has been the SHA256 algorithm optimisation suggestions that are specific to Bitcoin mining. Due to these suggested optimisations, it has been claimed that Bitcoin miners will now need to compute SHA256 only 1.8624 times in order to calculate H2 once as opposed to the normal 2 times. An improvement proposal was also made regarding the Bitcoin mining reward halving cycle with a view to introduce linearity in the reward given to miners for mining new Bitcoins. The thesis also made an attempt to organise background information as well as the related information which would be needed in order to fully comprehend what was being suggested. A discussion has also been made regarding the accuracy of the Savings Factor and the need for implementing and comparing Bitcoin mining as performed by off-the-shelf SHA256 and the optimised version of SHA256 for a more accurate quantification of this Savings Factor. The need for a critical analysis of the algorithm optimisations' compatibility with existing hardware optimisations has also been discussed.

It is believed that the suggested optimisations will bring about radical throughput improvements in Bitcoin mining devices. They will also allow Bitcoin miners to make a lot of savings in their electricity bills. It was decided to make all this information public with the vision of the betterment of the Bitcoin community and it is hoped that these findings will be a stepping stone to faster and more efficient Bitcoin mining.

"Vires in Numeris"

Bibliography

[1] ANSI. “X9. 62: 2005: The Elliptic Curve Digital Signature Algorithm (ECDSA).” Public Key Cryptography for the Financial Services Industry, 2005.

[2] Back, Adam. “Hashcash - A Denial Of Service Counter-Measure.” http://www.hashcash.org/papers/hashcash.pdf, 2002.

[3] Bitcoin Block Explorer. http://blockexplorer.com/ (accessed September 1, 2013).

[4] Bitcoin Charts. http://bitcoincharts.com/ (accessed September 1, 2013).

[5] Bitcoin Forum. “block.version=1 blocks will all be orphaned soon.” 18 March 2013. https://bitcointalk.org/index.php?topic=154521.0 (accessed September 1, 2013).

[6] Bitcoin Forum. “Cryptographic reasoning for double-hash?” https://bitcointalk.org/index.php?topic=45456.0 (accessed September 1, 2013).

[7] Bitcoin Stack Exchange. “Why does Bitcoin use two applications of SHA256?” http://bitcoin.stackexchange.com/questions/4317/why-does-bitcoin-use-two-rounds-of-sha256 (accessed September 1, 2013).

[8] Bitcoin Wiki. https://en.bitcoin.it/wiki/Main_Page (accessed September 1, 2013).

[9] Bitcoin Wiki. “Bitcoin Protocol Specification.” https://en.bitcoin.it/wiki/Protocol_specification (accessed September 1, 2013).

[10] Bitcoin Wiki. “Generation Calculator.” https://en.bitcoin.it/wiki/Generation_Calculator (accessed September 1, 2013).

[11] Blockchain.info. “Bitcoin Currency Statistics.” https://blockchain.info/stats (accessed September 1, 2013).

[12] Bloomberg. “Virtual Bitcoin Mining Is a Real-World Environmental Disaster.” 12 April 2013. http://www.bloomberg.com/news/2013-04-12/virtual-bitcoin-mining-is-a-real-world-environmental-disaster.html (accessed September 1, 2013).

[13] Boyar, Joan, and René Peralta. “A New Combinational Logic Minimization Technique with Applications to Cryptology.” Vol. 6049, in Experimental Algorithms, 178-189. Springer Berlin Heidelberg, 2010.

[14] Boyar, Joan, Philip Matthews, and René Peralta. “Logic Minimization Techniques with Applications to Cryptology.” Journal of Cryptology (Springer-Verlag) 26, no. 2 (April 2013): 280-312.

[15] Chaves, Ricardo, Georgi Kuzmanov, Leonel Sousa, and Stamatis Vassiliadis. “Cost-Efficient SHA Hardware Accelerators.” IEEE Transactions on Very Large Scale Integration (VLSI) Systems 16, no. 8 (2008): 999-1008.

[16] Chaves, Ricardo, Georgi Kuzmanov, Leonel Sousa, and Stamatis Vassiliadis. “Improving SHA-2 Hardware Implementations.” In Cryptographic Hardware and Embedded Systems, 298-310. 2006.

[17] CNN Money. “Bitcoin Prices Surge Post-Cyprus Bailout.” 28 March 2013. http://money.cnn.com/2013/03/28/investing/bitcoin-cyprus/index.html (accessed September 1, 2013).

[18] Courtois, Nicolas T., Daniel Hulme, and Theodosis Mourouzis. “Solving Circuit Optimisation Problems in Cryptography and Cryptanalysis.” IACR Cryptology ePrint Archive 2011, 2011: 475.

[19] Courtois, Nicolas T., et. al. “The Unreasonable Fundamental Incertitudes Behind Bitcoin (extended version).” First Draft. 2013.

[20] Crowe, Francis, Alan Daly, Tim Kerins, and William Marnane. “Single-Chip FPGA Implementation of A Cryptographic Co-Processor.” Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on. IEEE, 2004. 279-285.

[21] Dadda, Luigi, Marco Macchetti, and Jeff Owen. “An ASIC Design for a High Speed Implementation of the Hash Function SHA-256 (384, 512).” Proceedings of the 14th ACM Great Lakes Symposium on VLSI. ACM, NY, USA, 2004. 421-425.

[22] Dadda, Luigi, Marco Macchetti, and Jeff Owen. “The Design of a High Speed ASIC Unit for the Hash Function SHA-256 (384, 512).” Proceedings of Design, Automation and Test in Europe Conference and Exhibition. IEEE, 2004. 70-75.

[23] Devine, Christophe. “FIPS-180-2 Compliant SHA-256 Implementation.” https://github.com/jessek/hashdeep/blob/master/src/sha256.c (accessed September 1, 2013).

[24] Dustcoin.com. “Cryptocoin Mining Information.” http://dustcoin.com/mining (accessed September 1, 2013).

[25] Esuruoso, Olakunle. “High Speed FPGA Implementation of Cryptographic Hash Function.” Doctoral Dissertation, University of Windsor. Electronic Theses and Dissertations, 2011.

[26] FIPS PUB 180-2. “SHA256 Standard.” National Institute of Standards and Technology (NIST). 1 August 2002. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf.

[27] Fog, Agner. “Instruction Tables: Lists of Instruction Latencies, Throughputs And Micro-Operation Breakdowns For Intel, AMD and VIA CPUs.” http://www.agner.org/optimize/instruction_tables.pdf (accessed September 1, 2013).

[28] Forbes. “Bitcoin Mining Is Not A Real World Environmental Disaster.” 14 April 2013. http://www.forbes.com/sites/timworstall/2013/04/14/bitcoin-mining-is-not-a-real-world-environmental-disaster/ (accessed September 1, 2013).

[29] Grinberg, Reuben. “Bitcoin: An Innovative Alternative Digital Currency.” Hastings Science & Technology Law Journal 4, no. 1 (2011): 159-208.

[30] Imtiaz, Ahmad, and A. Shoba Das. “Hardware Implementation Analysis of SHA-256 and SHA-512 Algorithms on FPGAs.” Computers & Electrical Engineering 31, no. 6 (2005): 345-360.

[31] Johnson, Don, Alfred Menezes, and Scott Vanstone. “The Elliptic Curve Digital Signature Algorithm (ECDSA).” International Journal of Information Security (Springer-Verlag) 1, no. 1 (August 2001): 36-63.

[32] Kaplanov, Nikolei M. “Nerdy Money: Bitcoin, the Private Digital Currency, and the Case Against its Regulation.” Temple University Legal Studies Research Paper, 2012.

[33] Kim, Mooseop, Jaecheol Ryou, and Sungik Jun. “Efficient Hardware Architecture of SHA-256 Algorithm for Trusted Mobile Computing.” Information Security and Cryptology. Springer Berlin Heidelberg, 2009. 240-252.

[34] Kolivas, Con. “Github - cgminer.” https://github.com/ckolivas/cgminer (accessed September 1, 2013).

[35] Lien, Roar, Tim Grembowski, and Kris Gaj. “A 1 Gbit/s Partially Unrolled Architecture of Hash Functions SHA-1 and SHA-512.” Vol. 2964, in Topics in Cryptology - CT-RSA 2004, 324-338. Springer Berlin Heidelberg, 2004.

[36] Macchetti, Marco, and Luigi Dadda. “Quasi-Pipelined Hash Circuits.” Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on. IEEE, 2005. 222-229.

[37] Madhavi, V.C., Dr.Ch.Ravi Kumar, and G.Rama Krishna Prasad. “New Techniques for Hardware Implementations of SHA.” Global Journal of Researches in Engineering Electrical and Electronics Engineering 12, no. 7 (2012): 34-38.

[38] Matusiewicz, Krystian, Josef Pieprzyk, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. “Analysis of Simplified Variants of SHA-256.” Western European Workshop on Research in Cryptology 74 (2005): 123-134.

[39] McEvoy, Robert P., Francis M. Crowe, Colin C. Murphy, and William P. Marnane. “Optimisation of the SHA-2 family of Hash Functions on FPGAs.” Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on. Karlsruhe: IEEE, 2006.

[40] Mendel, Florian, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. “Analysis of Step-Reduced SHA-256.” In Fast Software Encryption, 126-143. 13th International Workshop: Springer Berlin Heidelberg, 2006.

[41] Mt.Gox. https://www.mtgox.com/ (accessed September 2013, 1).

[42] Nakamoto, Satoshi. “Bitcoin - A Peer-to-Peer Electronic Cash System.” 2008. http://bitcoin.org/bitcoin.pdf.

[43] NIST. “SHA-256 Example.” http://csrc.nist.gov/groups/ST/toolkit/documents/Examples/SHA256.pdf.

[44] P2P Foundation. “Bitcoin - A New Open Source P2P E-Cash System.” http://p2pfoundation.net/bitcoin (accessed September 1, 2013).

[45] Reid, Fergal, and Martin Harrigan. “An Analysis of Anonymity in the Bitcoin System.” In Security and Privacy in Social Networks, 197-223. Springer New York, 2013.

[46] Rosenberg, Paul, and Thomas Anderson. “The Basics of Bitcoin - A Freeman's Perspective Primer.” FreemansPerspective.com. May 2013. http://www.freemansperspective.com/issues/FMP-bitcoinprimer-may2013.pdf.

[47] Satoh, Akashi, and Tadanobu Inoue. “ASIC-Hardware-Focused Comparison For Hash Functions MD5, RIPEMD-160, and SHS.” Integration, the VLSI Journal 40, no. 1 (2007): 3-10.

[48] Selvakumar, A. Arul Lawrence, and C. Suresh Gnandhas. “Analysis of Building Blocks in SHA256.” Research Journal of Applied Sciences, Engineering & Technology 1, no. 1 (2009): 10-15.

[49] Sky NEWS. “Cyber Currency Surge Amid Eurozone Crisis.” 30 March 2013. http://news.sky.com/story/1071652/cyber-currency-surge-amid-eurozone-crisis (accessed September 1, 2013).

[50] The Bitcoin Foundation. “Bitcoin.” http://bitcoin.org (accessed September 1, 2013).

[51] The Economist. “Mining Digital Gold.” 13 April 2013. http://www.economist.com/news/finance-and-economics/21576149-even-if-it-crashes-bitcoin-may-make-dent-financial-world-mining-digital (accessed September 1, 2013).

[52] Ting, Kurt K., Steve C. L. Yuen, K. H. Lee, and Philip H. W. Leong. “An FPGA Based SHA-256 Processor.” Vol. 2438, in Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream, 577-585. Springer Berlin Heidelberg, 2002.

[53] Wired.co.uk. “Wary of Bitcoin? A guide to Some Other Cryptocurrencies.” 7 May 2013. http://www.wired.co.uk/news/archive/2013-05/7/alternative-cryptocurrencies-guide/viewall (accessed September 1, 2013).

[54] Wired.com. “The Rise and Fall of Bitcoin.” 23 November 2011. http://www.wired.com/magazine/2011/11/mf_bitcoin/all/ (accessed September 1, 2013).

List of Figures

List of Tables

List of Equations

Appendix A: SHA256 Constants (K_t)

K_0 = 0x428A2F98; K_1 = 0x71374491; K_2 = 0xB5C0FBCF; K_3 = 0xE9B5DBA5;
K_4 = 0x3956C25B; K_5 = 0x59F111F1; K_6 = 0x923F82A4; K_7 = 0xAB1C5ED5;
K_8 = 0xD807AA98; K_9 = 0x12835B01; K_{10} = 0x243185BE; K_{11} = 0x550C7DC3;
K_{12} = 0x72BE5D74; K_{13} = 0x80DEB1FE; K_{14} = 0x9BDC06A7; K_{15} = 0xC19BF174;
K_{16} = 0xE49B69C1; K_{17} = 0xEFBE4786; K_{18} = 0x0FC19DC6; K_{19} = 0x240CA1CC;
K_{20} = 0x2DE92C6F; K_{21} = 0x4A7484AA; K_{22} = 0x5CB0A9DC; K_{23} = 0x76F988DA;
K_{24} = 0x983E5152; K_{25} = 0xA831C66D; K_{26} = 0xB00327C8; K_{27} = 0xBF597FC7;
K_{28} = 0xC6E00BF3; K_{29} = 0xD5A79147; K_{30} = 0x06CA6351; K_{31} = 0x14292967;
K_{32} = 0x27B70A85; K_{33} = 0x2E1B2138; K_{34} = 0x4D2C6DFC; K_{35} = 0x53380D13;
K_{36} = 0x650A7354; K_{37} = 0x766A0ABB; K_{38} = 0x81C2C92E; K_{39} = 0x92722C85;
K_{40} = 0xA2BFE8A1; K_{41} = 0xA81A664B; K_{42} = 0xC24B8B70; K_{43} = 0xC76C51A3;
K_{44} = 0xD192E819; K_{45} = 0xD6990624; K_{46} = 0xF40E3585; K_{47} = 0x106AA070;
K_{48} = 0x19A4C116; K_{49} = 0x1E376C08; K_{50} = 0x2748774C; K_{51} = 0x34B0BCB5;
K_{52} = 0x391C0CB3; K_{53} = 0x4ED8AA4A; K_{54} = 0x5B9CCA4F; K_{55} = 0x682E6FF3;
K_{56} = 0x748F82EE; K_{57} = 0x78A5636F; K_{58} = 0x84C87814; K_{59} = 0x8CC70208;
K_{60} = 0x90BEFFFA; K_{61} = 0xA4506CEB; K_{62} = 0xBEF9A3F7; K_{63} = 0xC67178F2;

Appendix B: SHA256 Implementation in C

```
#include <stdio.h>
#include <math.h>


// Shift right
#define SHR(x,n) ((x & 0xFFFFFFFF) >> n)


// Rotate right
#define ROTR(x,n) (SHR(x,n) | (x << (32 - n)))


// σ0 and σ1
#define S0(x) (ROTR(x, 7) ^ ROTR(x,18) ^  SHR(x, 3))
#define S1(x) (ROTR(x,17) ^ ROTR(x,19) ^  SHR(x,10))


// Σ0 and Σ1
#define S2(x) (ROTR(x, 2) ^ ROTR(x,13) ^ ROTR(x,22))
#define S3(x) (ROTR(x, 6) ^ ROTR(x,11) ^ ROTR(x,25))

// Maj and Ch
#define F0(x,y,z) ((x & y) | (z & (x | y)))
#define F1(x,y,z) (z ^ (x & (y ^ z)))

void main ()
{
    unsigned long W[64], K[64];
    unsigned long A, B, C, D, E, F, G, H, temp1, temp2;
    int t;

// Initialising working variables with some random hash value
    A = 0x641711d4; B = 0x71a975e1; C = 0x80b27340; D = 0xaa475500;
    E = 0x8ef0a0b9; F = 0x7d2f14fd; G = 0x87dca129; H = 0x215da880;

// Hard coding the fixed W_t for the first 16 rounds
    W[0] = 0xFFFFFFFF; W[1] = 0xFFFFFFFF; W[2] = 0xFFFFFFFF; W[3] = 0x00000005;
    W[4] = 0x80000000; W[5] = 0x00000000; W[6] = 0x00000000; W[7] = 0x00000000;
    W[8] = 0x00000000; W[9] = 0x00000000; W[10] = 0x00000000; W[11] = 0x00000000;
    W[12] = 0x00000000; W[13] = 0x00000000; W[14] = 0x00000000; W[15] = 0x00000280;
```

```
//SHA256 constants Kt
    K[0] = 0x428A2F98; K[1] = 0x71374491; K[2] = 0xB5C0FBCF; K[3] = 0xE9B5DBA5;
    K[4] = 0x3956C25B; K[5] = 0x59F111F1; K[6] = 0x923F82A4; K[7] = 0xAB1C5ED5;
    K[8] = 0xD807AA98; K[9] = 0x12835B01; K[10] = 0x243185BE; K[11] = 0x550C7DC3;
    K[12] = 0x72BE5D74; K[13] = 0x80DEB1FE; K[14] = 0x9BDC06A7; K[15] = 0xC19BF174;
    K[16] = 0xE49B69C1; K[17] = 0xEFBE4786; K[18] = 0x0FC19DC6; K[19] = 0x240CA1CC;
    K[20] = 0x2DE92C6F; K[21] = 0x4A7484AA; K[22] = 0x5CB0A9DC; K[23] = 0x76F988DA;
    K[24] = 0x983E5152; K[25] = 0xA831C66D; K[26] = 0xB00327C8; K[27] = 0xBF597FC7;
    K[28] = 0xC6E00BF3; K[29] = 0xD5A79147; K[30] = 0x06CA6351; K[31] = 0x14292967;
    K[32] = 0x27B70A85; K[33] = 0x2E1B2138; K[34] = 0x4D2C6DFC; K[35] = 0x53380D13;
    K[36] = 0x650A7354; K[37] = 0x766A0ABB; K[38] = 0x81C2C92E; K[39] = 0x92722C85;
    K[40] = 0xA2BFE8A1; K[41] = 0xA81A664B; K[42] = 0xC24B8B70; K[43] = 0xC76C51A3;
    K[44] = 0xD192E819; K[45] = 0xD6990624; K[46] = 0xF40E3585; K[47] = 0x106AA070;
    K[48] = 0x19A4C116; K[49] = 0x1E376C08; K[50] = 0x2748774C; K[51] = 0x34B0BCB5;
    K[52] = 0x391C0CB3; K[53] = 0x4ED8AA4A; K[54] = 0x5B9CCA4F; K[55] = 0x682E6FF3;
    K[56] = 0x748F82EE; K[57] = 0x78A5636F; K[58] = 0x84C87814; K[59] = 0x8CC70208;
    K[60] = 0x90BEFFFA; K[61] = 0xA4506CEB; K[62] = 0xBEF9A3F7; K[63] = 0xC67178F2;

// Calculating Wt values after round 16
    for(t=16;t<64;t++)
    {
        W[t] = S1(W[t -  2]) + W[t -  7] + S0(W[t - 15]) + W[t - 16];
    }

// Message compression
    for(t=0;t<64;t++)
    {
       temp1 = H + S3(E) + F1(E,F,G) + K[t] + W[t];
       temp2 = S2(A) + F0(A,B,C);
       H = G; G = F; F = E; E = D + temp1;
       D = C; C = B; B = A;
       A = temp1 + temp2;

        // Printing out the values at the end of round 4
        if(t==3)
        {
              printf("%x %x %x %x %x %x %x %x\n", W[t], A, B, C, D, E, F, G, H);
       }
    }
}
```

Appendix C: H1 Message Schedule Calculation in C

```
#include <stdio.h>
#include <math.h>

// Shift right
#define SHR(x,n) ((x & 0xFFFFFFFF) >> n)

// Rotate right
#define ROTR(x,n) (SHR(x,n) | (x << (32 - n)))

// σ0 and σ1
#define S0(x) (ROTR(x, 7) ^ ROTR(x,18) ^  SHR(x, 3))
#define S1(x) (ROTR(x,17) ^ ROTR(x,19) ^  SHR(x,10))

void main ()
{
    unsigned long W[64];
    int t;

// Hard coding the fixed Wt for the first 16 rounds
    W[0] = 0xFFFFFFFF; W[1] = 0xFFFFFFFF; W[2] = 0xFFFFFFFF; W[3] = 0x00000005;
    W[4] = 0x80000000; W[5] = 0x00000000; W[6] = 0x00000000; W[7] = 0x00000000;
    W[8] = 0x00000000; W[9] = 0x00000000; W[10] = 0x00000000; W[11] = 0x00000000;
    W[12] = 0x00000000; W[13] = 0x00000000; W[14] = 0x00000000; W[15] = 0x00000280;

// Calculating Wt values after round 16
    for(t=16;t<64;t++)
    {
        W[t] = S1(W[t -  2]) + W[t -  7] + S0(W[t - 15]) + W[t - 16];
    }
// Printing Wt values for rounds 17, 18 and 20
    for(t=16;t<18;t++)
    {
        printf("%d) %x\n",t+1,W[t]);
    }
    printf("20) %x\n", W[19]);
}
```