Towards Timelock Encryption

Wesley Smith

THE UNIVERSITY OF EDINBURGH

4TH YEAR PROJECT REPORT

SCHOOL OF INFORMATICS

supervised by
Dr. Myrto ARAPINIS

April 2019

1 Abstract

Applications of distributed technologies is a burgeoning area of research; the success of cryptocurrencies has sparked investigation into other use cases of blockchain technology such as e-voting. Timelock encryption, or encrypting a message such that anyone can decrypt after a set amount of time has passed, is a concept with unexplored utility pertaining to decentralized technology. Functional timelock encryption could allow users to publish sensitive information such as candidate choice to the public that could eventually be easily decrypted by anyone. This work contains a partial implementation of a recently proposed timelock encryption scheme, including the creation of an original SAT formulation of blockchain validity contraints, the creation of a vector family corresponding to an equivalent Subset-Sum problem instance, and a proposed proof-of-concept witness encryption scheme built from multipartite Diffie-Hellman key exchange via cryptographic multilinear maps. In addition, it contains discussion of the feasibility of realizing schemes built on multilinear maps.

Acknowledgements

Many thanks to Myrto Arapinis, my supervisor, for outstanding and thorough mentorship and guidance.

In addition, thanks to Graham Dutton for invaluable computing support and Markulf Kohlweiss for help with cryptography.

Contents

2 Introduction

2.1 Motivation

Rivest, Shamir, and Wagner's 1996 paper *Time-lock Puzzles and Timed-release Crypto* [23] was the first academic publication to investigate the concept of *timelock encryption*: encrypting data such that decryption doesn't require "key" in the traditional sense, but instead requires that a set amount of time has passed. They discuss two major approaches: encrypting through a set-difficulty computational problem, and using trusted third parties to release decryption information after the set time has passed. Many implementations of the first concept exist [4], and the second, less technical approach is similar to the way time-sensitive goods and information are handled in the real world by trusted third parties today (wills, trusts, etc). However, these approaches beg the question: can timelock encryption be achieved without a trusted third party and without requiring expensive computation on the part of the decrypter?

2.1.1 Decentralized Technology

The success of Bitcoin, one of the first mainstream decentralized technologies (and easily the most prominent), has elevated decentralized tech from an intellectual curiosity to an active area of research. Although distributed ledgers exist in the real world, the paper that introduced Bitcoin [21] was the first to mathematically realize the concept of a *blockchain*. *Bitcoin: A Peer-to-Peer Electronic Cash System* leverages the blockchain concept as a tool to make a peer-to-peer currency practical. In the years following its publication it has become clear that the potential of blockchain and similar technologies far surpasses what Bitcoin utilizes it for.

The goal of most decentralized technologies is the same - eliminate the need for trusted third parties. Bitcoin is the most intuitive example of this: by making all transactions peer-to-peer and recordkeeping with a distributed ledger it eliminates the need for a central financial authority. Voting is an alternative use for blockchain technology, as an on-chain voting protocol would be "transparent" and "easily auditable" [19] while improving accessibility, all without the threat of election fraud conducted by the voting authority.

2.1.2 E-cclesia

E-cclesia: Self-tallying E-voting Protocol in the UC Framework [19] introduces a self-tallying voting protocol that seeks to achieve these goals. One crucial property of any voting scheme is *fairness*: that "no partial tally is revealed that could influence the votes of the remaining voters" [19]. A scheme fails this property if "a protocol releases any information on votes while still accepting new votes."

One approach the authors of E-cclesia are currently considering is to use timelock encryption on the votes to guarantee fairness. Neither of the primitive forms of timelock encryption discussed in the original paper will work; E-cclesia is decentralized and potentially on-chain, so there can be no trusted third party to hold votes. In addition, E-cclesia cannot implement a computational form of timelock encryption as the cost would be prohibitive.

To this end, we are working with a novel timelock encryption scheme proposed in Liu et al.'s recently published *How To Build Time-lock Encryption* [18]. This scheme promises efficient decryption without a trusted third party, a type of timelock encryption not considered by Rivest et al. To realize this, the Liu scheme uses the Bitcoin blockchain to develop a *witness encryption* scheme: the predictable nature of Bitcoin block generation means that the BTC blockchain functions as an excellent computational reference clock.

2.2 Building Timelock Encryption

The scheme in *How To Build Time-lock Encryption* has three major parts:

- Deriving an NP-relation from the BTC blockchain
- Building witness encryption from that NP-relation
- Combining the witness encryption scheme with known properties of the blockchain to realize timelock encryption

Under this scheme, Liu et al. guarantee the following properties [18]:

- Non-interactivity: the encrypter is not needed for decryption
- No trusted setup: the scheme involves no trusted third parties
- No resource restrictions: there is minimal [1] computational expenditure on the part of the decrypter (assuming the Bitcoin blockchain continues in its current form)

[1] "Minimal" in this case means any computational overhead on the part of the decrypter is a side-effect of the implementation details and not intended to contribute to the overall difficulty of decryption.

3 Background

3.1 The Bitcoin Blockchain

A blockchain is typically a publicly available distributed ledger. It's divided into discrete blocks, each of which contains two things: information defining that block in the context of the larger chain, and record-keeping information. In the cryptocurrency use-case, this record-keeping information details financial transactions involving digital cash. In addition, since anyone who wishes to has a distinct copy of the blockchain, there must exist some notion of *consensus*, or resolving differences in different parties' versions of the chain. Blockchain "mining" refers to extending the existing blockchain; in real blockchains this consists of solving a cryptographic puzzle. The miner that solves it then appends the newly mined block to the chain, and all other miners verify his or her solution. From the perspective of a ledger, mining is updating the blockchain to contain all transactions performed since the last block was mined.

As a data structure, a blockchain is essentially a linked list: each block is an item composed of transaction information and header information. The blocks are "chained" by the existence of some function $H()$: any consecutive blocks B_i and B_{i+1} must be related by this $H()$. This is how miners check that a proposed new block is indeed a valid block. Figure 1 illustrates this concept:

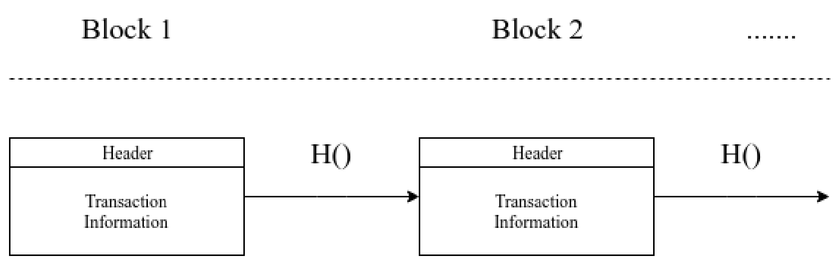

Figure 1: A simplified view of blockchain structure

Example
For clarity, we will present a small example of a toy blockchain. Here, blocks will consist of only one thing: a 3-bit block value $000 - 111$ interpreted as an integer. $H()$ will be adding 1 to the block value: in other words, each block's value must be one greater than the previous block's value.

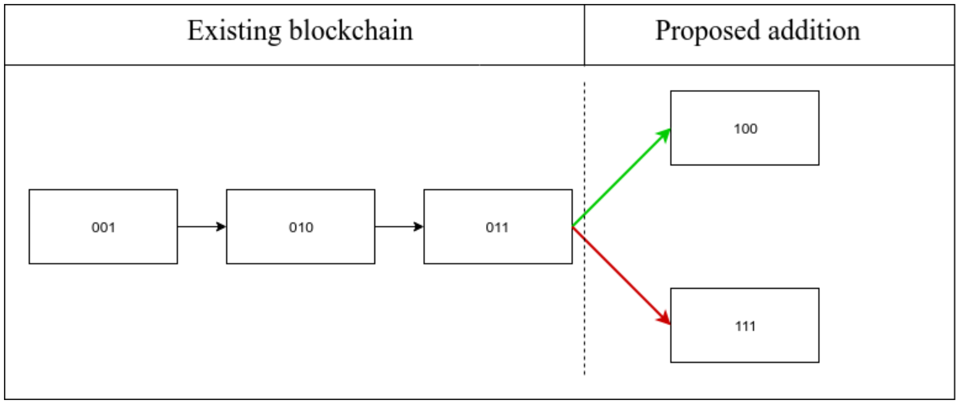

Figure 2: A toy blockchain example

In Figure 2, three valid blocks exist in the chain, with values 1, 2, and 3. Two proposed extensions exist: one has value $100 = 4$, and one has value $111 = 7$. Because $H()$ requires incrementing block value by exactly 1, block 100 will be appended and block 111 will be discarded. Clearly this blockchain is trivial to extend and contains no transaction information, so it would have no utility as a real public ledger. However, it illustrates important blockchain concepts in the context of this work.

We can now examine the Bitcoin blockchain. Bitcoin is "peer-to-peer electronic cash" [21] and is the the most successful example of blockchain technology. Bitcoin uses two rounds of SHA-256 as $H()$, or the function giving the chaining property. SHA-256 is a member of the SHA-2 family of hash functions built from the Merkle-Damgard construction; like other hash functions, its most important properties include collision resistance and one-way behavior. SHA-256 takes inputs of any length and outputs a 256 bit hash value. The cryptographic puzzle Bitcoin requires miners to solve involves finding an output of SHA-256 that is smaller than a target value defined by a difficulty parameter. Because SHA-256 is unpredictable, this is computationally difficult.

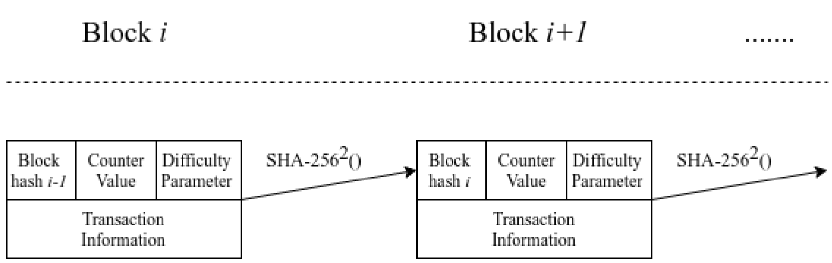

Figure 3: A simplified view of Bitcoin blockchain structure

Figure 3 updates Figure 1 to contain detail specific to the Bitcoin blockchain. Note that each block has four major components.

Now for a more rigorous look. Define the following:

- T: a list of transactions

- r: a counter value
- D: a difficulty parameter [2]
- B: a block hash value

We then express the Bitcoin blockchain as the sequence of tuples ("blocks")

$$(T_1, r_1, D_1, B_1)...(T_n, r_n, D_n, B_n)$$

where the following relations must hold:

- $B_i \leq D_i$
- $B_i = H(T_i, r_i, D_i, B_{i-1})$, where H consists of two rounds of SHA-256.

These constraints are what will later allow us to build an NP-relation from the Bitcoin blockchain. Note that D_i is set dynamically in response to the current mining environment and, as such, future difficulty parameters are unknown.

The difficulty parameter dictates the amount of computational power required to "mine" the next block; this is set such that the time required for block generation is approximately 10 minutes.

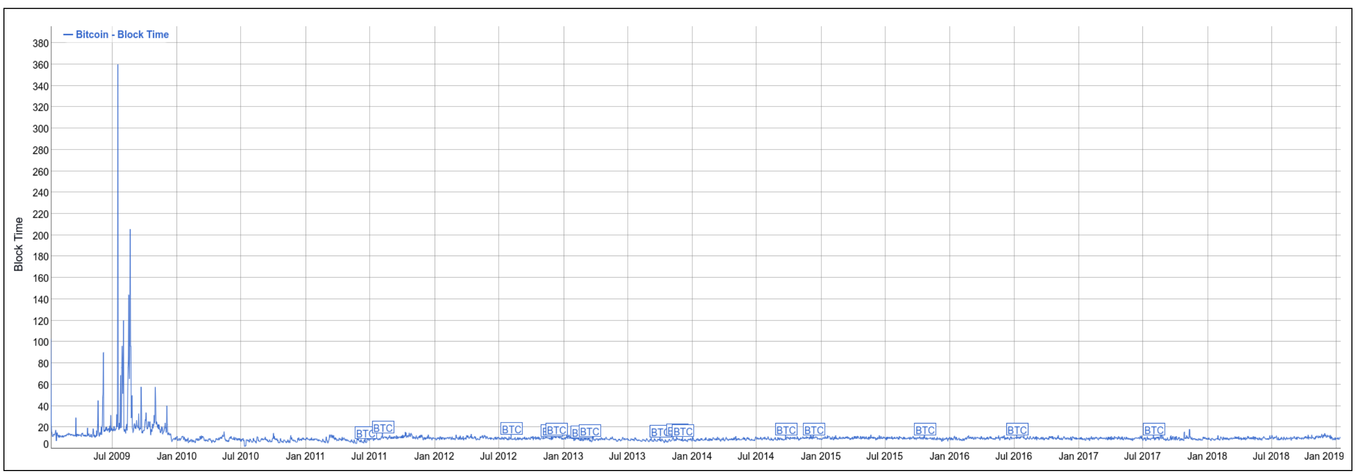

Figure 4: BTC block generation times

Figure 4 [1] shows that this is remarkably consistent; since 2012 there has never been a block generation time above 20 minutes or less than 5. This clear correspondence between time and blockchain length [3] is crucial to the concept of timelock encryption, as a clear correspondence exists between blockchain length and elapsed time.

[2] More literally D is a *target value* generated from a difficulty parameter: we will refer to it as a difficulty parameter for simplicity.

[3] Note that "length" in the context of the BTC blockchain has a more rigorous definition involving block difficulty. In this paper, however, "length" will only mean "number of blocks".

3.2 CNF-SAT and Subset-Sum

This project is concerned with two major NP-complete problems: CNF-SAT and Subset-Sum. A problem is in the NP complexity class if solutions can be verified but not found deterministically in polynomial time, and it is NP-hard if it is "at least as hard" as the hardest NP problems. If a problem is both NP and NP-hard, it is NP-complete [27].

The Cook-Levin theorem [26] proved SAT, or the Boolean satisfiability problem, to be the first known NP-complete problem. Its formulation is as follows:

Given propositional statement **x** with literals $x_1...x_n$, **does an assignment of** $x_1...x_n$ **exist such that x evaluates to true**?

More specifically, we work with CNF-SAT, which stipulates the the propositional statement be in conjunctive normal form: a conjunction of disjunctions (e.g $(x_1 \bigvee x_2) \bigwedge (x_2 \bigvee x_3)...$).

The other NP-complete problem we're interested in is Subset-Sum, concerning whether or not a subset of a vector family sums to a target vector. Its formulation is as follows, where v_i denotes a vector and l_i a corresponding integer:

Given vector multiset $\Delta = (v_i : l_i)$ and target sum **s**, **does there exist a subset of** Δ **that sums to s** (where each v_i can be used at most l_i times)?

Mathematically:

$$\text{Is } \sum_{\mathrm{v}_i \in \Delta} b_i \mathrm{v_i} = \mathbf{s} \text{ for some } b_i\text{s, where } 0 \leq b_i \leq l_i?$$

An instance of any NP-complete problem can be translated to an instance of any other NP-complete problem; later we will discuss one such conversion from CNF-SAT to Subset-Sum.

3.3 Witness Encryption

The concept of *witness encryption* is central to our timelock encryption scheme. First introduced in 2013 [11], witness encryption requires that instead of knowing a secret key corresponding to a public key, the decrypter must know a *witness* corresponding to an *instance* of an NP-relation. As Garg et al. put it, "What if we don't really care if [the decrypter] knows a secret key, but we do care if he knows a solution to a crossword puzzle that we saw in the Times? Or if he knows a short proof for the Goldbach conjecture? Or, in general, the solution to some NP search problem?" [11]

We say a *language* is a subset of (0,1)* that specifies a decision problem. An NP-relation R associated with a language L serves to codify the relationship between problem instances and solutions within the language L. More specifically, for an *instance* x and a *witness* w, $x \in L \iff \exists w$ such that $(x, w) \in R$.

Thus, witness encryption requires the following:

- A message space M: this is the set of all messages compatible with the witness encryption scheme
- A ciphertext space C: this is the set of all possible ciphertexts under the witness encryption scheme
- An NP-relation R on a language L. Our work contains discussion of three NP-relations: blockchain validity, Boolean satisfiability, and Subset-Sum.
- An instance x and witness w of L such that $(x, w) \in R$. An example of this for CNF-SAT is given later in this section.
- An encryption algorithm $WE.encrypt(1^p, x, m)$ for $m \in M$. $WE.encrypt$ takes a message m from the message space and returns a ciphertext c from the ciphertext space. [4]
- A decryption algorithm $WE.decrypt(c, w)$ for $c \in C$. $WE.decrypt$ takes a ciphertext c from the ciphertext space and returns a message m from the message space.

Of course, correctness must be satisfied: if $(x, w) \in R$,

$$WE.decrypt(WE.encrypt(x, m), w) = m.$$

Garg et al. focus on NP-complete problems (belonging to both the NP and NP-hard complexity classes) to build timelock encryption. Well known NP-complete problems form good examples for the intuition behind witness encryption: take the Boolean satisfiability problem (SAT). Here, an instance used for encryption would be a propositional statement, and a witness used for decryption would be a satisfying valuation of the literals in the sentence. As Boolean satisfiability is an NP-complete problem, such a satisfying valuation is both easy to verify and hard to find.

Take as an example a small case with CNF-SAT as our relation. Our CNF-SAT instance is

$$\mathbf{x} = (x_1 \bigvee x_2) \bigwedge (x_3 \bigvee x_4) \bigwedge (x_1 \bigvee x_4)$$

A witness to this instance is a valuation of the literals $x_1 ... x_4$ for which the sentence evaluates to true: for example, $[x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 1]$. Then, for some $c \in C, m \in M$:

- $WE.encrypt(1^p, ((x_1 \bigvee x_2) \bigwedge (x_3 \bigvee x_4) \bigwedge (x_1 \bigvee x_4)), m) = c$
- $WE.decrypt(c, [x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 1]) = m$

Witness encryption has nothing intrinsically to do with timelock encryption; one of the major contributions of Liu et al. is to combine witness encryption with a *computational reference clock* in a way that makes timelock encryption feasible. Liu et al. provide the insight that **the Bitcoin blockchain allows us to define an instance for which there will only be a corresponding witness at some point in the future**.

[4] 1^p here is a security parameter and not crucial to understanding the scheme at this level.

3.4 Multilinear Maps

A cryptographic multilinear map is a mathematical structure through which objects, usually vectors, can be encoded. Multilinear maps guarantee important properties detailed below; these properties give multilinear maps potential utility as a component of many cryptographic primitives and schemes.

The witness encryption scheme proposed by Liu et al. is built on cryptographic multilinear maps. Boneh et al. [2] proposed the use of multilinear maps in cryptography in 2003, claiming possible uses (if the technology were to be realized) include key-exchange protocols and signature schemes. Since, three constructions have been proposed that bring the concept closer to practicality: Garg et al. [10] introduced a lattice-based construction, Gentry et al. [12] introduced a graph-based construction, and Coron et al. in 2013 [6] and 2015 [7] introduced a "practical" multilinear map scheme over the integers.

Per Boneh et al. [2], a cryptographic multilinear map is a map e between groups G_1^n, G_2 such that the following holds for $e : G_1^n \longrightarrow G_2$:

- G_1, G_2 are groups of the same prime order
- for $a_1, ...a_n \in \mathbb{Z}, x_1, ...x_n \in G_1$:
$$e(x_1^{a_1}, ...x_n^{a_n}) = e(x_1, ...x_n)^{\prod_{i=1}^{n} a_i}$$

Note that a multilinear map is just a mathematical structure with the above guaranteed properties. Multilinear map research such as that described at the beginning of the section attempts to use various mathematical objects to realize such a structure. Section 6.2 will describe one such approach in more detail.

The security guarantees provided by multilinear maps are not fully understood. Cryptanalysis attacks have been shown to exist against primitives instantiated with this concept. One such attack was against an older version of the Coron et al. scheme: Cheon et al. [3] derive a diagonalizable matrix from the parameters of a set of multilinear maps and recover the secret primes upon which security of the maps depends. A second attack breaks the graph-based construction [5]. The scheme we will discuss and implement in this work has been shown to be secure against this attack specifically, but it's worth noting that the security of the constructions we will discuss depends on the security of multilinear map schemes as a whole.

3.5 Timelock Encryption

Timelock encryption consists of encrypting data such that the decrypter needs to "show" that a set amount of time has passed. There are three main approaches:

-*Timelock puzzles* First introduced by Rivest et al., timelock puzzles are set-difficulty computational problems. By requiring a specific amount of computation to decrypt, and by approximating the computational power of the decrypter, timelock puzzles can relate a difficulty to an amount of time. This puzzle needs to be in no way parallelizable; Rivest et al. propose successive squarings, or computing ($2^t \bmod p$) for p, t defined in such a way that the security of the scheme is based on the discrete logarithm problem. This approach has been implemented [4] and further investigated, but is intrinsically unsuitable for our problem as it requires significant computational effort on the part of the decrypter.

- *Trusted Party Cryptography* Rivest et al. also describe a somewhat less technical approach: using a trusted party to release information at a set time. This exists in the world and is used for many applications (wills, trusts, sensitive information handling, etc). Even the more technical of these approaches, such as key-sharing, are clearly completely unsuitable for use in a decentralized voting scheme. However, more cutting edge options now exist - such a "trusted" party could be replaced by decentralized technologies. Chao Li and Palanisami Balaji [16] investigate using Ethereum smart contracts to eliminate trust while maintaining enforceability and security properties. This technique and their results are promising, but the required expense and effort make it unfeasible for our decentralized voting use-case.

- *Timelock Encryption from Witness Encryption* As described before, this approach requires a "puzzle" where the solution is only available at a set point in the future. Using the blockchain as our NP-relation allows exactly this.

Building timelock encryption this way is surprisingly intuitive. Say we want to encrypt for a minutes. The problem facing anyone who wishes to decrypt our ciphertext is as follows:

Describe a valid blockchain of length ($a/10$) where the initial block is the current block at the time of encryption.

Note that this does not truly differ from a timelock puzzle in any fundamental way; finding a valid blockchain *is* a computational problem, and it *is* possible for an adversary to brute-force a solution. However, there is an extreme financial disincentive - anyone who has the ability to outpace the Bitcoin mining pool could become enormously rich instead of cracking our votes. The Liu scheme leverages the enormous computational resources being put towards Bitcoin mining to mitigate an adversary's ability to decode early. Bitcoin miners constitute an enormous and consistent computational resource pool, with an estimated 2^{62} hashes being computed every second worldwide [18], and using that to circumvent computation on the part of a decrypter is a clever way to increase the utility of time-lock puzzles.

3.6 Diffie-Hellman Key Exchange

This topic represents a slight digression from the previously introduced material; while it is not directly related to timelock encryption, part of our work with witness encryption is directly related to the Diffie-Hellman key exchange and as such we overview it here.

In their groundbreaking 1976 paper *New Directions in Cryptography*, Whitfield Diffie and

Martin Hellman introduced the concept of public-key cryptography [9]. A fundamental form of this is Diffie-Hellman key exchange, which involves joint creation of a "shared secret" by two or more parties, often a key allowing for symmetric-key encryption. Each party uses the public keys, denoted *pk*, of the others in conjunction with their own secret key *sk* in such a way that each user obtains the same result. Thus, parties wanting to communicate securely can separately compute a key that an attacker would find "computationally infeasible [to recompute] from the information overhead" [9].

Diffie-Hellman key exchange can be realized with finite cyclic groups as follows. Here, Alice and Bob are the two parties wishing to establish a shared secret, and Eve is an eavesdropping adversary.

1. Alice and Bob choose a cyclic, finite multiplicative group G and a generator $g \in G$. These are both known to Eve.

2. Alice chooses a secret key a and Bob chooses a secret key b. a, b must be less than $|G|$ (the number of elements in G).

3. Alice sends g^a to Bob, and Bob sends g^b to Alice.

4. Both parties compute $g^{ab} = (g^a)^b = (g^b)^a$. This is the shared secret.

Eve knows only G, g, g^a, g^b and as such cannot efficiently recompute g^{ab}. The difficulty sources from the discrete logarithm problem, or DLOG: for group elements g, h and group G, if g, h, G are sufficiently chosen it is difficult to find x such that $g^x = h$.

4 Main Contributions

The main contributions of the project are as follows:

1. A CNF-SAT problem instance representing an original conversion from Bitcoin blockchain validity constraints

2. A Python implementation of the above conversion that builds a SAT problem instance from a CNF expression of SHA-256. This SAT problem was created with software implementing the Tseitin transform, which for readability is discussed in Chapter 5 instead of with other background information

3. A Python implementation of an existing conversion from SAT to Subset-Sum, generating a multiset of vectors

4. Optimizations of the above to bring the time and space costs within feasability given our computational resources. Much of the difficulty was in circumventing engineering problems stemming from the enormity of a logic gate representation of SHA-256

5. An original, proof of concept witness encryption scheme built from cryptographic multilinear maps that is compatible in concept with encrypting to Subset-Sum problem instances

6. A C++ implementation of the above scheme built from an existing implementation of multipartite Diffie-Hellman key exchange via cryptographic multilinear maps

5 Problem Conversion

The witness encryption scheme proposed in *How to Build Time-Lock Encryption* is tailored specifically for Subset-Sum. Thus, the end result of this section should be a Subset-Sum problem instance. Our starting point is an instance of our NP-relation: blockchain validity constraints defined by a length and a series of difficulty parameters. Liu et al. give a conversion from CNF-SAT to Subset-Sum. Thus, Chapter 5 will contain the following:

- An original theoretical conversion from blockchain validity constraints to CNF-SAT and the implementation process for that conversion
- Description and discussion of the resultant SAT problem, btcSAT
- The implementation process for the given conversion from CNF-SAT to Subset-Sum
- Description and discussion of the resultant Subset-Sum problem, Δ

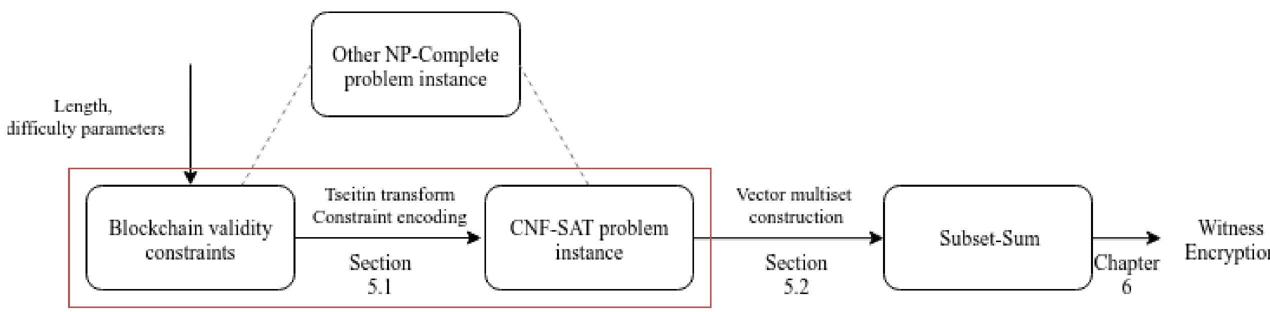

Figure 5: Overview of the problem conversions required

Figure 5 details this process. The grey dashes indicate a potential alternate approach that was considered but was ultimately unsuccessful. The red box indicates an original theoretical conversion; Liu et al. assume that a translation exists from blockchain validity constraints to CNF-SAT but do not give one.

5.1 BTC to CNF-SAT

The first mathematically rigorous result in the problem conversion above is a CNF-SAT problem instance. SAT, or the Boolean satisfiability problem, involves finding a satisfying assignment of literals for a propositional sentence. CNF-SAT is a slight refinement; it requires the propositional sentence to be in conjunctive normal form (a conjunction of disjunctive clauses).

To recap, we're interested in two specific blockchain constraints to try and build an NP relation. They are as follows:

- $B_i \leq \mathrm{D}_i$
- $(T_i, r_i, D_i, B_i) = H(B_{i-1})$, where H consists of two rounds of SHA-256.

where B_i is the block hash value, D_i is the difficulty parameter[5], and $H()$ is a hash function. Specifically, $H()$ is two rounds of SHA-256.

Initially we investigated a possible conversion from blockchain parameters to a problem other than CNF-SAT, as the Liu paper does not explicitly state how an implementation of their scheme would arrive at a CNF-SAT representation. Minimizing the size of the problem was important at this stage, as we wanted to avoid engineering problems later stemming from the size of the components of our Subset-Sum vector family. Any NP-complete problem instance can be converted to an instance of any other NP-complete problem [27], and we hoped that the constraints above would lend themselves intuitively to a particular NP-relation.

Unfortunately no clever options presented themselves; the issue underlying all possibilities was the complex and unpredictable nature of SHA-256. The second constraint, encapsulating the "chain" property of a valid blockchain, constitutes the vast majority of the complexity and difficulty of deriving an NP problem instance where a validating literal assignment corresponds to a valid instance of the NP language defined by $H()$: $(x, w) \in R \iff H(x) = w$. In this case, x is a propositional sentence and w an assignment of its literals. To this end, we opted to go with a difficult but somewhat intuitive approach: encode the second constraint with a CNF-SAT representation of two rounds of SHA-256 as a logical circuit. In this model, the literals $x_1 ... x_{512}$ would represent the input (the bits of the previous block padded) and $x_{513} ... x_{768}$ would represent the output. Our propositional sentence would emulate the bitwise operations of SHA-256, and a valuation of $x_1 ... x_{768}$ would satisfy the sentence if and only if $x_{513} ... x_{768}$ interpreted as a sequence of bits is the correct output of SHA-256 performed twice on $x_1 ... x_{256}$.

Question 1 (Bitcoin Validity) *Do a given series of block hash values and associated difficulties describe a valid [6] blockchain, where the chaining function in question is two rounds of SHA-256?*

Example
Do the following three blocks define a valid section of blockchain, where $a, b, c \in [0, 2^{256} - 1]$?

$$(B_1 = a, D_1 = 2^{64} - 1), (B_2 = b, D_2 = 2^{60} - 1), (B_3 = c, D_3 = 2^{56} - 1)$$

Question 2 (Boolean Satisfiability) *Does a satisfying valuation for a propositional sentence $\boldsymbol{x}$ composed of literals $x_1 ... x_n$ exist?*

[5] In this case it is *our* difficulty parameter and not the actual Bitcoin parameter (see Section 7.3).

[6] A simplified definition of the Bitcoin validity constraint is used here. See Section 7.3 for more details.

Example Can an assignment of the literals $x_1 ... x_5$ cause the sentence to evaluate to true?

$$(x_1 \bigvee x_2) \bigwedge (x_3 \bigvee \tilde{x}_4) \bigwedge (\tilde{x}_3 \bigvee x_5)$$

Theorem 1 links the above questions by describing a SAT problem encapsulating Bitcoin blockchain validity per the two constraints at the beginning of the section.

Theorem 1 (Bitcoin Validity in SAT) *Let $\boldsymbol{x}$ be a propositional sentence encoding a preset blockchain length and set of difficulty parameters. $\boldsymbol{x}$ has literals $x_1 ... x_n$.*
Then a valuation of $x_1 ... x_n$ can satisfy $\boldsymbol{x}$ if and only if ***the literals corresponding to each B_i, interpreted as a bit string, are the correct output of SHA-256 performed twice on the literals corresponding to B_{i-1} interpreted as a bit string****. In addition, the literals corresponding to each D_i interpreted as an integer must be greater than the the literals corresponding to each B_i interpreted as an integer.*

The following sections describe building the components of btcSAT, **a CNF-SAT problem instance consistent with Theorem 1**.

5.1.1 First Constraint: $B_i \leq D_i$

We use bitwise comparisons to encode this inequality in CNF: here B_i and D_i are canonically interpreted as 256-bit integers. We denote the jth bit of B_i as $B_{i,j}$. A general SAT expression for $B_i \leq D_i$ can be built as follows:

$$(\tilde{B}_{i,1} \bigwedge D_{i,1}) \bigvee (((B_{i,1} \bigwedge D_{i,1}) \bigvee (\tilde{B}_{i,1} \bigwedge \tilde{D}_{i,1})) \bigwedge ((\tilde{B}_{i,2} \bigwedge D_{i,2}) \bigvee$$
$$(((B_{i,2} \bigwedge D_{i,2}) \bigvee (\tilde{B}_{i,2} \bigwedge \tilde{D}_{i,2}))...(\tilde{B}_{i,256} \bigwedge D_{i,256}) \bigvee (((B_{i,256} \bigwedge D_{i,256}) \bigvee (\tilde{B}_{i,256} \bigwedge \tilde{D}_{i,256}))))))))...$$

This expression, while unattractive, is straightforward: for $B_i \leq D_i$, either the first bit of D_i must be 1 and the first bit of B_i 0, or they must be equal. If they are equal, we consider the substring starting from the second bit in exactly the same way.

Note that this is a SAT formula but not a CNF formula as required by our translation. While the part of the expression involving each bit consists of only a few clauses, the expressions are nested within each other. Translating a SAT expression to CNF in the traditional way (applying DeMorgan's Law and the distributive property) can cause exponential size blowup [28], and given the structure of the sentence certainly would in this case. To avoid this, we made the following observation and adaptation:

The Bitcoin blockchain allows any 256 bit integer as the difficulty parameter: $D_i \in [0, 2^{256} - 1]$. Consider instead the following restricted domain: $D_i \in \{2^n - 1 : 0 < n \leq 256\}$. All elements of this new domain are of the form $0^a 1^b$, where $a + b = 256$. All elements of the form $0^a \{0, 1\}^b$ are less than or equal to $0^a 1^b$: it follows that comparing a string to one of the form $0^a 1^b$ only requires checking that the first a bits are 0:

$$(\tilde{B}_{i,1} \bigwedge \tilde{B}_{i,2} \bigwedge ... \bigwedge \tilde{B}_{i,a})$$

To this end, we decided to always choose our difficulty parameters from the restricted domain. As we choose our own D_is to mimic those of the real Bitcoin blockchain, we're free to round; in this case, we round to one less than the nearest power of two. As such, **for each block in the length of our instance, we add at most 256 clauses**, each with one literal. We use simple Python code to add these clauses to our CNF file.

5.1.2 Second Constraint: $B_i = H(T_i, r_i, D_i, B_{i-1})$

The section discusses encoding SHA-256 as a CNF-SAT instance. It examines two of our approaches to this, **makeSAT v1** and **makeSAT v2**, discusses the challenges they presented, and describes the workaround we found.

The logical structure of SHA-256 is necessarily quite complex. However, circuit descriptions do exist. A circuit description of SHA-256 gives a functionally equivalent network of logical gates. Two that we found had the following compositions:

Gate Type	Bristol Circuit	Goldfeder Circuit
AND	90,825	22,272
INV	103,258	2,194
XOR	42,029	91,780

Table 1: Structures of SHA-256 circuits

We moved forward with the Goldfeder circuit [13]. Although it was built from the Bristol circuit, the optimizations they performed for efficiency made it more appealing. Note the size of the circuits: even the smaller of the two (the Goldfeder circuit) has well over 100,000 logic gates in it. This is unsurprising, given how heavy duty SHA-256 is, but it is still problematic given the context for the CNF-SAT version of SHA in our implementation.

All coding in this section was done in Python.

makeSAT v1
Our first approach takes as input a value from 1 to 116758 and travels backwards through a circuit form of SHA-256 towards the inputs. These values correspond to gates in the logical structure of SHA-256. On each call it breaks a gate down into its inputs and an operation, then calls itself on the inputs. This terminates only on the input bits, i.e input values 1-512; remember that **we want to define the output bits of SHA-256 in terms of only variables x_1 to x_{512}**.

- Inputs: an integer specifying gate to be reduced. sha256.txt, a Boolean circuit description of SHA-256.
- Output: a propositional representation of the input gate containing only the input literals, $x_1 ... x_{512}$. To be used on gates 116503-116758, the output gates in the Goldfeder circuit.

- Helper functions: *gateDescription(int)*, accessing sha256.txt and returning the inputs of gate *int* and the gate type.

```
Data   : sha256.txt
Input  : int a
Output: propositional statement representing the gate specified by a

%if gate is an input bit, do not recur.  Otherwise, call self on
 gate's inputs and note the operation represented by the gate

if a ≤ 512 then
 | return a;
else
 | (in1, in2, op) ← gateDescription(a);
 | return(makeSAT(in1) OP makeSAT(in2));
end
```

Algorithm 1: makeSAT v1

In theory, calling **makeSAT** would result in the generation of a text file, the contents of which would contain a SAT problem instance representing the output of that gate in terms of the inputs. Note that this is not in CNF: the Goldfeder circuit contains AND gates, INV gates, and XOR gates (see Table 1). CNF converters exist, and as such we planned to do an automated conversion once **makeSAT** was functional and the SAT files were generated.

makeSAT v1 did not work as intended. The issue was in engineering rather than concept; we ran makeSAT on shallow gates (near the inputs) and used sha256.txt to verify the small resultant propositional sentences without error. However, running on an output gate as intended invariably caused a crash; system memory was insufficient to hold the enormous propositional statements generated this way. Both DICE machines and *student.compute* resources had insufficient memory. While other computational platforms such as AWS and Google Cloud Compute could have provided more resources, we weren't sure what the actual requirements were. If **makeSAT** v1 required only slightly more RAM than *student.compute* provides, cloud computing would have made it feasible, but the same is not true of more extreme amounts. Given the size of sha256.txt ($> 100,000$ gates), we decided to rethink our method.

makeSAT v2

Since RAM was the bottleneck in **makeSAT** v1, for v2 we opted to edit text files in-place instead of holding the entire statement in memory. **makeSAT** v2 takes as inputs a set of gates to be reduced, finds them in a text file containing a partially complete CNF, and reduces those gates by one level. This approach shifted the space cost from RAM to system storage, and would allow us to progress significantly farther than the recursive approach in **makeSAT v1**. In addition, specifying the gates to be reduced one at a time increased our operational control; simple for loop scripts expedited the process when the file was still small, and once each call to **makeSAT** v2 had significant time/space cost we processed gates in smaller batches.

- Input: sha256.txt, a Boolean circuit description of SHA-256. Int a, this time not specifying the start point for a full recursive traverse of the circuit, but specifying the gate to be expanded in the current partially complete .txt file.
- Output: 116503.txt - 116758.txt, 256 files (generated one at a time) each containing a propositional representation of one of SHA-256's output bits. So named as 116503-116758 are the gate numbers representing output in the Goldfeder circuit. Each file contains only the input literals, $x_1 ... x_{512}$.
- Helper functions: *gateDescription(int)*, accessing sha256.txt and returning the inputs of gate *int* and the gate type.

Data : sha256.txt
Input : int a, sentence.txt
Output: sentence.txt. which now contains no instances of gate a

```
% get description of gate with value a.  replace instances of a with
 gate expansion
```

(in1, in2, op) ← `gateDescription(`a`)`
for $a \in$ *sentence.txt* **do**
| $a \leftarrow (in1 \; OP \; in2)$
end

Algorithm 2: makeSAT v2

The size of the resultant .txt files illuminated exactly how far from feasible **makeSAT** v1 was. The space cost of a single .txt file, containing just one output bit, appeared to grow exponentially with the number of gates processed. sha256.txt contains over 100,000 gates; during a trial run we first ran a script to process the top 1000, then ran another to process the next 10.

Gates Processed	File Size
1	6B
1000	100MB
1010	4GB

Table 2: Size of .txt files generated by **makeSAT** v2

Were the relationship linear, the size of the file after processing 1000 gates would have been reassuring; the final file would have been on the order of 100GB, well within the storage capacity of modern computer systems. The next 10 gates, however, blew the filesize up far beyond what we expected and made it clear that reducing gate by gate would never be feasible. It's difficult to estimate what the size of a completed file would be, but based on the data above it's reasonable to assume it could easily be on the order of petabytes or more. We decided optimization was futile, as clever storage of the partial statements or similar could not hope to close that gap.

The Tseytin Transformation

Facing seemingly intractable costs to represent SHA-256 as a SAT problem, we reevaluated what we were trying to do in a broader context. To be a valid hash function, SHA-256 *cannot* be solvable as a SAT problem. At the moment, no pre-image attacks exist; what we were trying to do could enable exactly that. Through constructing satisfiability problems for the output variable in terms of the input variables, it would be straightforward (if currently computationally unfeasible) to set the output bits to a predetermined hash value and run an automated SAT solver to find a valid pre-image.

SAT solvers run in $O(1.3^n)$ [25], where n is the number of literals in the expression. Our approach would have 512, but only 256 are truly variables; the rest are padding and can be ignored/set. 1.3^{256}, while very large (on the order of 10^{29}) still represents a tremendous reduction in the worse case cost of a pre-image attack. It makes sense, then, that there exists a tradeof: compactness of expression for number of variables.

The Tseytin transformation served as a workaround. The Tseytin transformation generates a CNF-SAT representation of a logical circuit. In *On the Complexity of Derivation in Propositional Calculus*, [24] G.S Tseytin proved that the output CNF and the CNF resulting from applying DeMorgan's Law and the distributive property to the circuit are equisatisfiable; it is satisfiable if and only if the original is. A key aspect of the Tseytin transformation used for our purpose is its introduction of new variables: additional literals are added that correspond to each gate in the circuit. For use with SHA-256, this corresponds to more than 100,000 literals; while this removes from the realm of possibility solving the resulting SAT instance, it allows for a compact representation.

Example

Here we will apply the Tseitin transformation to a simple logical circuit composed of two OR gates and an AND gate: Figure 6 describes this circuit, which takes as input a 4-bit integer $(a_1a_2a_3a_4)$ and returns true if the number is greater than 5 but not equal to 8 or 12.

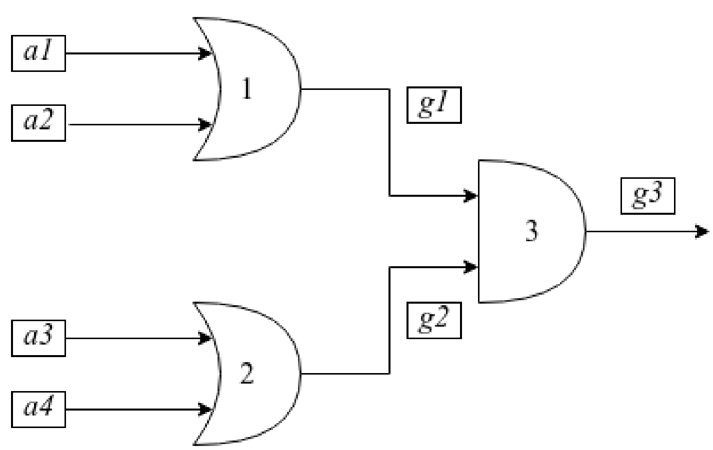

Figure 6: A simple logical circuit

To create a SAT instance from this circuit, the Tseytin transformation requires that we introduce a new boolean literal corresponding to each gate: these are denoted g_1, g_2, g_3 in the figure above. In total we now have seven literals: the inputs $a_1...a_4$ and $g_1...g_3$. We now generate CNF expressions for each gate variable relating the inputs and outputs with respect to the gate type (OR or AND):

$$g_1 : (g_1 \bigvee \tilde{a}_1) \bigwedge (g_1 \bigvee \tilde{a}_2) \bigwedge (g_1 \bigvee \tilde{a}_1 \bigvee \tilde{a}_2)$$

$$g_2 : (g_2 \bigvee \tilde{a}_3) \bigwedge (g_2 \bigvee \tilde{a}_4) \bigwedge (g_2 \bigvee \tilde{a}_3 \bigvee \tilde{a}_4)$$

$$g_3 : (\tilde{g}_3 \bigvee g_1) \bigwedge (\tilde{g}_3 \bigvee g_1) \bigwedge (g_3 \bigvee \tilde{g}_1 \bigvee \tilde{g}_3)$$

We then combine the above with a single clause enforcing the final output of the circuit, g_3 to obtain a CNF-SAT instance containing 10 clauses that is equisatisfiable to the logical circuit:

$$\begin{aligned}(g_1 \bigvee \tilde{a}_1) \bigwedge (g_1 \bigvee \tilde{a}_2) \bigwedge (g_1 \bigvee \tilde{a}_1 \bigvee \tilde{a}_2) \bigwedge (g_2 \bigvee \tilde{a}_3) \bigwedge (g_2 \bigvee \tilde{a}_4) \\ \bigwedge (g_2 \bigvee \tilde{a}_3 \bigvee \tilde{a}_4) \bigwedge (\tilde{g}_3 \bigvee g_1) \bigwedge (\tilde{g}_3 \bigvee g_1) \bigwedge (g_3 \bigvee \tilde{g}_1 \bigvee \tilde{g}_3) \bigwedge g_3\end{aligned}$$

In looking for software implementing the Tseytin transformation we had an unexpected stroke of luck - a github repository [20] containing software built to generate CNF-SAT instances of hash functions turned up. Vegard Nossum (who also built the SHA-1 instance generator Google used to successfully break SHA-1) built an instance generator implementing the Tseytin transformation that had options to work for one or two rounds of SHA-256. Understanding and using this software was straightforward, and resulted in *btcSHA.cnf*, encoding $H()$ from the constraint.

Combining the constraints

At this stage, btcSHA.cnf represented only two rounds of SHA, not the full BTC constraints we need to emulate. As such, before moving to the next conversion we appended the following to btcSHA.cnf:

- Clauses forcing equality between new variables representing the hash value of the previous block.
- Clauses forcing the output of SHA-256 to be less than new variables representing the difficulty parameter (interpreted as an integer). As proof-of-concept, we chose $2^{128} - 1$ as our difficulty here.

Equality is easy to encode: $a = b$ becomes $(a \bigwedge b) \bigvee (\tilde{a} \bigwedge \tilde{b})$. In CNF: $(a \bigvee \tilde{b}) \bigwedge (\tilde{a} \bigvee b)$. Two clauses were added to btcSHA.cnf for each of $x_1...x_{256}$, and 256 new variables were introduced representing the previous block hash. Section 5.1.1 details what was added for the difficulty constraint: the only difference is that the bits chosen here were the output bits of our chaining function H.

5.1.3 btcSAT: a CNF-SAT problem instance

Our resultant file, *btcSAT.cnf*, is one of the major original results of this project. It contains a CNF-SAT problem instance corresponding to a valid blockchain of length 2, and was created in the following three steps:

- Using Vegard Nossum's implementation of the Tseytin transformation to build a CNF expression from the logic gate structure of two rounds of SHA-256.
- Appending clauses forcing a simplified "chaining" property of the blockchain (i.e $B_i = H(B_{i-1})$) [7]
- Developing and appending a concise expression of the difficulty property of the blockchain (i.e $B_i \leq D_i$)

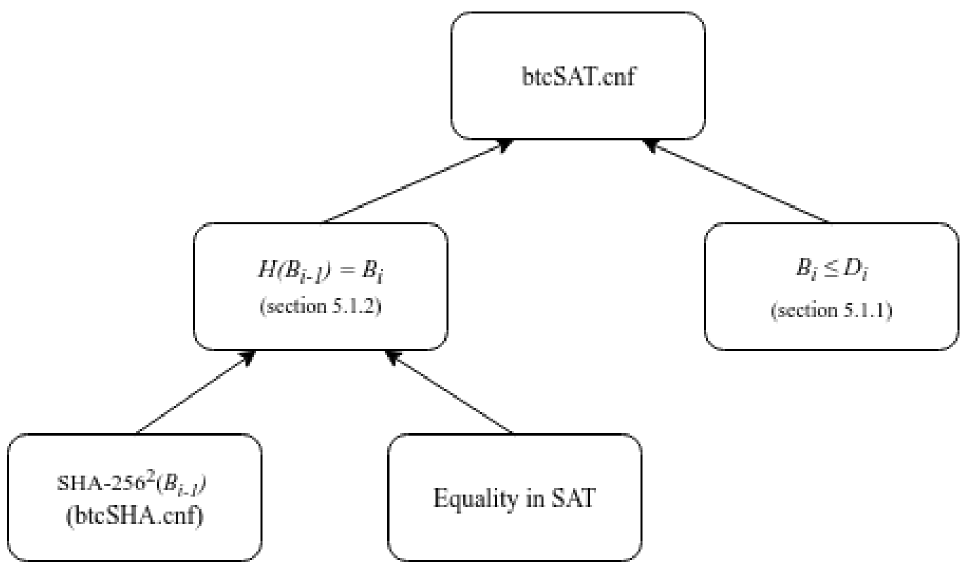

Figure 7: A visual overview of the construction of btcSAT

Figure 7 shows the structure of btcSAT.cnf; it contains the block hash constraint and the difficulty constraint. The block hash constraint is built by combining btcSHA.cnf, generated by software implementing the Tseitin transform, and equality in SAT. Table 3 contains a technical description of btcSAT.cnf.

Filename	File Size	Num. of BTC blocks	Num. of literals (n)	Num. of clauses (k)
btcSAT.cnf	19.3MB	2	260929	901841

Table 3: Description of *btcSAT.cnf*

[7]See Section 7.3 for discussion of this property

While btcSAT only represents a blockchain of length 2, the procedure can be easily generalized and used to form a CNF problem instance corresponding to any length, and by extension any duration of encryption. Both n, the number of literals, and k, the number of clauses, are essentially linear in the length of the chain. Longer chains just require duplicating *btcSHA.cnf* and editing the appended clauses listed above: the difficulty can vary per block, and the "chaining" property needs to be implemented using the correct variables.

5.2 CNF-SAT to Subset-Sum

Subset-Sum is another well-known NP-complete problem. A problem instance consists of the following:

Given vector multiset $\Delta = (v_i : l_i)$ and target sum s, **does there exist a subset of** Δ **that sums to s** (where each vector v_i can be used at most l_i times)?

Mathematically:

$$\text{Is } \sum_{\mathrm{v}_i \in \Delta} b_i \mathrm{v_i} = \mathbf{s} \text{ for some } b_i\text{s, where } 0 \leq b_i \leq l_i?$$

Our next step was to build Δ, the multiset of vectors resulting from converting our CNF-SAT problem instance to a Subset-Sum problem instance. The conversion is outlined in *How to Build Time-Lock Encryption* [18]. Section 5.2 contains the following:

- An overview of an existing theoretical conversion from SAT to the vector component of Subset-Sum
- Descriptions of and pseudocode for three approaches to constructing the vector component of Δ: **makeVecs** v1, v2, and v3
- An overview of an existing theoretical conversion from SAT to the integer component of Subset-Sum
- Description and discussion of Δ, the resulting Subset-Sum instance

5.2.1 $\Delta = (v_i : l_i)$ - building v_i

This section overviews constructing the v_is (the vector components of a Subset-Sum problem instance).

Define the following:

- n: the number of variables $x_1 ... x_n$ in a CNF expression
- k: the number of clauses
- m_i for $1 \leq i \leq k$: the number of literals in clause k

Then we construct the following binary vectors, all of length $(n + 2k)$:

- $u_1, u_2 ... u_n$: the ith element of each u_i is 1, and each $n + j$th element is 1 if and only if literal x_i occurs positively in clause j. All other elements 0.
- $\tilde{u}_1, \tilde{u}_2 ... \tilde{u}_n$: the ith element of each $\tilde{u}_i$ is 1, and each $n + j$th element is 1 if and only if literal x_i occurs negatively in clause j. All other elements 0.
- $v_1, v_2, ... v_k$: the $n+i$th and $n+k+i$th elements of each v_i are 1. All other elements 0.
- $z_1, z_2, ... z_k$: the $n + k + i$th element of each z_i is 1. All other elements 0.

We also must construct a target vector s: this is an integer vector (non-binary) also of length $n + 2k$.

- $1 \leq i \leq n$: $s_i = 1$
- $(n + 1) \leq i \leq (n + k)$: $s_i = m_i$
- $(n + k + 1) \leq i \leq (n + 2k)$: $s_i = m_i - 1$

The u and ũ vectors correspond to the variables $x_1 ... x_n$ in the CNF - specifically the two possible assignments of each variable. Vector v_j corresponds to an unsatisfied literal in clause j, while z_j corresponds to an satisfied literal in clause j other than the first.

Theorem 2 (Subset-Sum Creation Correctness) *A CNF-SAT instance $\boldsymbol{x}$ is satisfiable if and only if there exists a solution to its corresponding Subset-Sum problem built in the above way.*

For a proof of Theorem 2, see [18].

makeVecs v1
This first approach to vector generation is a naive implementation that builds the vectors described in Section 5.2. It builds the vectors sequentially and stores them exactly as described in the paper. Its runtime is $O(n(n + k))$.

- Input: A CNF expression to be converted to Subset-Sum. n, the number of literals in the CNF to be parsed. k, the number of clauses.
- Output: one of the sets of vectors described in Section 5.2. Here we build $u_1 ... u_n$, but each set is almost identical to build. Each vector is of length $n + 2k$.

```
Data    : btcSAT.cnf with clauses C_1...C_k
Input   : int n, int k
Output: binary vectors u_1...u_n

%for each variable
for i from 1 to n do
    %assign 1 to the vector index equal to that variable's index
    u_{i,i} ← 1
    %assign 0 to all others in that range
    for m from 1 to n do
        if m ≠ i then
            u_{i,m} ← 0
        end
    end
    %for each clause
    for j from 1 to k do
        %assign relevant vector element to 1 if variable is in clause.
         0 otherwise
        if x_i ∈ C_j then
            u_{i,n+j} ← 1
        else
            u_{i,n+j} ← 0
        end
    end
    %all other elements 0
    for p from 1 to n do
        u_{i,n+k+p} ← 0
    end
end
```

Algorithm 3: makeVecs v1

Given that in our instance $n \approx 250,000$ and $k \approx 1,000,000$, a runtime of $O(n(n+k)$ is clearly problematic. Indeed, our implementation of **makeVecs** v1 took hours to generate a single vector out of more than two million. Our next approach was to traverse the CNF only once:

makeVecs v2

Our second attempt is much more time efficient: it traverses *btcSHA.cnf* only once and generates the vectors in parallel. Since $|C_j| \leq 4$ ($|C_j|$ denotes the number of literals in clause j), the runtime is $O(n+k)$. This time, we were able to execute the code on our CNF expression and generate vectors.

- Input: A CNF expression to be converted to Subset-Sum. n, the number of literals in the CNF to be parsed. k, the number of clauses.

- Output: one of the sets of vectors described in Section 5.2. Here we build $u_1 ... u_n$, but each set is almost identical to build. Each vector is of length $n + 2k$.

```
Data   : btcSAT.cnf with clauses C_1...C_k
Input  : int n, int k
Output: binary vectors u_1...u_n

%create zero vectors

for i from 1 to n do
|  u_i ← 0
end

%for each variable in each clause, change the associated element of
 that variable's vector to 1

for j from 1 to k do
|   for x_m ∈ C_j do
|   |  u_{m,n+j} ← 1
|   end
end
```

Algorithm 4: makeVecs v2

Unfortunately, we only partially generated the required amount before we realized that space cost was still an issue. We need a total of $2n + 2k + 1$ vectors, each of length $n + 2k$. With *btcSHA.cnf*, that's $2,325,541$ vectors of length $2,064,611$. Storing each vector element as a 8-bit integer, this would require approximately 4.8TB of storage - far more than is practical for our purposes.

makeVecs v3

The third and final approach to generating our vector multiset Δ leverages the fact that the vectors as described in *How to Build Timelock Encryption* are both sparse and binary; they are of length $n + 2k$ but are composed of almost entirely zeros. We take advantage of this and compress the vectors: instead of storing them naively, we turn them into lists of integers where the values denote the indices of 1s. Storing the vectors this way resulted in immense space savings, and the overall space cost went from $\approx$ 5TB to $\approx$ 50MB.

- Input: A CNF expression to be converted to Subset-Sum. n, the number of literals in the CNF to be parsed. k, the number of clauses.

- Output: one of the sets of vectors described in Section 5.2. Here we build $u_1 ... u_n$, but each set is almost identical to build. Each vector is of variable length, but $|u_i| << (n + 2k)$.

Data : btcSAT.cnf with clauses $C_1...C_k$
Input : int n, int k
Output: small integer lists $\mathrm{u}_1...\mathrm{u}_n$

```
%create empty lists for each vector
```

for *i from 1 to n* **do**
  $\mathrm{u}_i \leftarrow []$
end

```
%for each variable in each clause, append the clause index to the
 relevant list
```

for *j from 1 to k* **do**
  for $x_m \in C_j$ **do**
    $\mathrm{u}_m.\mathrm{append}(n + j)$
  end
end

Algorithm 5: makeVecs v3

The only vector not stored in this sparse way is the target vector; as there is only one, **makeVecs** v2 sufficed as the space cost was not excessive. In addition, the target vector is not sparse, so this compression would make little difference.

Appendix B contains pseudocode generating the other three sets of vectors, ũ, v, and z in the same way as **makeVecs** v3.

5.2.2 $\Delta = (v_i : l_i)$ - building l_i

In the context of Δ, l_i (an integer) denotes the maximum number of times vector v_i can be used in creating the target sum.

The way the v_is are defined makes constructing the l_is trivial. Consider the following:

- $\mathrm{u_i}$ and $\tilde{\mathrm{u}}_\mathrm{i}$ correspond to opposite assignments of the same literal. A witness will assign each literal to exactly one; therefore for $\mathrm{u_1...u_n}$ and $\tilde{\mathrm{u}}_1...\tilde{\mathrm{u}}_\mathrm{n}$, $l_i = 1$.

- $\mathrm{v_j}$ corresponds to an unsatisfied literal in clause j. There are m_j literals in clause j, and of course a satisfying valuation can have at most $m_j - 1$ of them evaluate to false. So $l_{v_j} = m_j - 1$.

- $\mathrm{z_j}$ corresponds to a satisfied literal in clause j *other than the first*. There are m_j literals in clause j, and of course a valuation can have at most m_j of them evaluate to true. So $l_{z_j} = m_j - 1$.

5.2.3 Δ: a Subset-Sum problem instance

A Subset-Sum problem instance corresponding to the validity of a Bitcoin blockchain of length 2 is another significant original result of this project. Tables 4 and 5 summmarize

Δ, its components, and its creation. This marked the end of the problem conversion in Figure 3.

Algorithm version	**Vector generation**	**Runtime**	**Vector storage**	**File Size**
makeVecs v1	Sequential	$O(n(n+k))$	Naive	4.8TB
makeVecs v2	Parallel	$O(k)$	Naive	4.8TB
makeVecs v3	Parallel	$O(k)$	Compressed	50MB

Table 4: Summary of generating Δ

Notation	l_i	**Intuition**	**Number of vectors**	**Filesize**
u	1	Positive literal occurrences	260929	10.8MB
ũ	1	Negative literal occurrences	260929	11.1MB
v_j	$m_j - 1$	Unsatisfied literals in clause j	901841	7.2MB
z_j	$m_j - 1$	Satisfied literals in clause j	901841	3.6MB
s	-	Target	1	4MB
Δ	-	Complete vector multiset	2325541	36.7MB

Table 5: Composition of Δ

5.2.4 Computing a witness $w = b_1 \ldots b_m$ from the blockchain

While a length of blockchain serves as a witness to an instance used during encryption, the blockchain is not the witness itself. As the problem we're interested in is Subset-Sum, a witness consists of coefficients $(b_0 \ldots b_{|\Delta|})$ such that $\sum_{\mathrm{v}_i \in \Delta} b_i \mathrm{v_i} = \mathbf{s}$.

The following details the construction of each such coefficient from a blockchain:

- $\mathrm{u_i}$: $b_{\mathrm{u_i}} = 1$ if the i-th bit of the relevant block hash value is 1, 0 otherwise.
- $\tilde{\mathrm{u}}_\mathrm{i}$: $b_{\tilde{\mathrm{u}}_\mathrm{i}} = 1$ if the i-th bit of the relevant block hash value is 0, 0 otherwise.
- all $\mathrm{v_j}$: traverse the CNF. For each clause j, for each unsatisfied literal add one to b_{v_j}.
- all $\mathrm{z_j}$: traverse the CNF. For each clause j, for each satisfied literal beyond the first add one to b_{z_j}.

This process is fully general, and will work for a Subset-Sum instance corresponding to a blockchain of any length.

Example
As an example, consider the following situation; note that it is fabricated for the purpose of illustration and does not correspond to any real blockchain validity instance:

1. We have a three bit blockchain, and as such have literal vectors $u_1 \ldots u_3$ and $\tilde{u}_1 \ldots \tilde{u}_3$
2. Our CNF has 4 clauses, and as such we have clause vectors $v_1 \ldots v_4$ and $z_1 \ldots z_4$.

Take our CNF-SAT instance representing blockchain validity to be

$$(x_1 \bigvee x_2) \bigwedge (x1 \bigvee \tilde{x}_2) \bigwedge (x_1 \bigvee x_3) \bigwedge (x_2 \bigvee x_3)$$

Now assume a block hash value we see is (101). From this and the CNF we will compute a witness (coefficients of each vector).

From that we see $x_1 = 1, x_2 = 0, x_3 = 1$, and by extension $b_{u_1} = b_{u_3} = 1$ and $b_{u_2} = 0$. Conversely, $b_{\tilde{u}_1} = b_{\tilde{u}_3} = 0$ and $b_{\tilde{u}_2} = 1$.

Clause 1 has 1 unsatisfied literal (x_2), so $b_{v_1} = 1$. Similarly, $b_{v_2} = 0, b_{v_3} = 0$, and $b_{v_4} = 1$. Clause 1 has 1 satisfied literal (x_1), so $b_{z_1} = 0$. Similarly, $b_{z_2} = 1, b_{z_3} = 1$, and $b_{v_4} = 0$.

We have defined a coefficient b for each of $2n + 2k = 2(3) + 2(4) = 14$ vectors in our Subset-Sum instance, and as such have computed a complete witness.

6 Witness Encryption

With Δ, our proof-of-concept Subset-Sum instance, generated, the next step was implementing a witness encryption scheme tailored to Subset-Sum.

As a recap, witness encryption involves encrypting to an *instance* x of an NP-language L, and decryption involves providing a *witness* w such that $(x, w) \in R$, some NP-relation. It's worth noting that some conventional cryptography problems can be viewed as witness encryption: take as an example DLOG as discussed in Section 3.6. Given g, h and group G, finding x such that $g^x = h$ could be viewed as x serving as a witness to the more general $g^? = h$ problem instance. However, witness encryption as a paradigm allows for NP-relations well outside the realm of what we find in traditional cryptography problems like DLOG.

Our instance is a Subset-Sum problem: this corresponds to the desired length of blockchain, the starting block (the current Bitcoin block), and difficulty parameters. Regularity of Bitcoin block generation ensures that a well-defined relation exists between blockchain length and time. A witness to that instance would be a blockchain of the given length that starts with the given initial block and is valid both with respect to the "chaining" property and the difficulty property.

We require a witness encryption scheme to apply to Δ. While Δ is a specific instance (length 2, arbitrary difficulty values), an encryption scheme working for a specific Δ can be later generalized.

Chapter 6 is theoretically and notationally dense; Appendix A contains a notational reference table to help resolve confusion.

6.1 Liu et al.: WE.encrypt, WE.decrypt

The following section describes the witness encryption scheme suited for encrypting a Subset-Sum problem instance proposed in *How to Build Timelock Encryption.*

Define the following:

- Δ: a Subset-Sum problem instance with target vector $\mathbf{s}$ (all vectors length d)
- $e()$: the mapping operation corresponding to a set of cryptographic multilinear maps tailored to Δ
- w: a witness to Δ (e.g $b_1 ... b_{|\Delta|}$ such that $\sum_{\mathrm{v}_i \in \Delta} b_i \mathrm{v_i} = \mathbf{s}$)
- α: a random vector of length d
- α^{v_i}: the operation $\alpha_1^{v_{i,1}} * \alpha_2^{v_{i,2}} ... * \alpha_d^{v_{i,d}}$
- g_v: a group generator (generating one of the groups used in the map set) corresponding to vector v

Then the following pair of algorithms constitute witness encryption:

- WE.encrypt(Δ, m): choose random α and generate a set of cryptographic multilinear maps compatible with Δ (denoted **params**).
Then ciphertext $c =$(**params**, $\{g_{v_i}^{\alpha^{v_i}}\}_{i \in \Delta}$, $m * g_s^{\alpha^s}$)

- WE.decrypt(c, w): compute key K:

$$K = e(\underbrace{g_{v_1}^{\alpha^{v_1}}, ...g_{v_1}^{\alpha^{v_1}}}_{b_1}, \underbrace{g_{v_2}^{\alpha^{v_2}}, ...g_{v_2}^{\alpha^{v_2}}}_{b_2}, ... \underbrace{g_{v_{|\Delta|}}^{\alpha^{v_{|\Delta|}}}, ..., g_{v_{|\Delta|}}^{\alpha^{v_{|\Delta|}}}}_{b_{|\Delta|}})$$

which then can be used to try and invert $m * g_s^{\alpha^s}$

This scheme directly translates Subset-Sum from a problem involving integer arithmetic to a problem involving mapping group elements.

Remember that multilinearity guarantees the property $e(x_1^{a_1}, ...x_n^{a_n}) = e(x_1, ...x_n)^{\Pi_{i=1}^n a_i}$. It follows that if a correct witness $w = b_1...b_{|\Delta|}$ is provided,

$$K = g_{\sum b_i v_i}^{\alpha^{\sum b_i v_i}} = g_s^{\alpha^s} \text{ since of course } \sum b_i v_i = \mathbf{s}$$

To implement this, we needed to find such a multilinear map scheme.

6.2 New Multilinear Maps over the Integers

As described in Section 3.3, only three major candidate constructions exist for multilinear maps: a lattice-based construction GGH13 [10], a graph-based construction GGH15 [12], and a scheme over the integers CLT which is built on similar underlying principles as the lattice scheme [6]. *How to Build Timelock Encryption* refers directly to a lattice based scheme like the one proposed by Garg et al., but the authors indicate that their witness encryption construction should work independently of the multilinear map instantiation.

Of the three, only Coron et al.'s *New Multilinear Maps Over the Integers* has a suitably general publicly available codebase [14]; as it is outside the scope of this project to build a multilinear map library from scratch, we decided to move forward with the CLT scheme.

Under the CLT scheme, vectors are encoded as integers. Define the following (note that variables such as n have values independent of prior sections):

- $a = (a_1, ...a_n)$: a vector to be encoded
- $p_1...p_n$: secret primes
- z: a random integer modulo $\Pi_{i=1}^n p_i$
- $g_1...g_n$: small secret primes

- $r_1 ... r_n$: small random integers (noise)

Then a level-k encoding of m is the integer d such that:

$$\forall i \in [1, n] : d = \frac{r_i * g_i + a_i}{z^k} \bmod p_i$$

Level-k refers to the level of multilinearity of the encoding - the most important aspect of this concept is that **pairing a level-k_1 encoding with a level k_2 encoding results in an encoding of level $k_1 + k_2$**.

Note that a multilinear map is just a mathematical structure with certain guaranteed properties (see Section 3.4). Described here is one specific construction to allow us to create such a mathematical structure.

6.2.1 Diffie-Hellman via Multilinear Maps

As described in Section 3.5, conventional Diffie-Hellman key exchange uses finite cyclic groups; the difficulty of recomputing the shared secret comes from the discrete logarithm problem. The Decisional Diffie-Hellman computational hardness assumption is related to DLOG, and formalizes the security of Diffie-Hellman key exchange from a more specific perspective.

Coron et al. describe how the CLT multilinear map scheme can be used to realize multipartite Diffie-Hellman key exchange. In their words, "The security of the protocol relies on a new hardness assumption which is a natural extension of the Decisional Diffie-Hellman assumption" [7]. Because multiplying encodings results in summing their multilinearity levels, we can use a level-κ multilinear map instance to do multipartite Diffie-Hellman with $\kappa + 1$ users as follows:

1. Encode each user's private key (level-0) as a level-1 encoding. Publish these encodings.
2. For each set of $\binom{\kappa+1}{\kappa}$ level-1 encodings, compute the product as a level-κ encoding.
3. Each user computes the product of their private key and the level-κ encoding of the other users' keys. This will be the same for all users - the shared secret.

Figure 8 provides a visual example of this. pk_i and sk_i denote the public and secret keys of user i. Similarly to how in conventional Diffie-Hellman an adversary cannot compute g^{ab} from g^a, g^b, here an adversary cannot compute the key from each of the level-κ encodings.

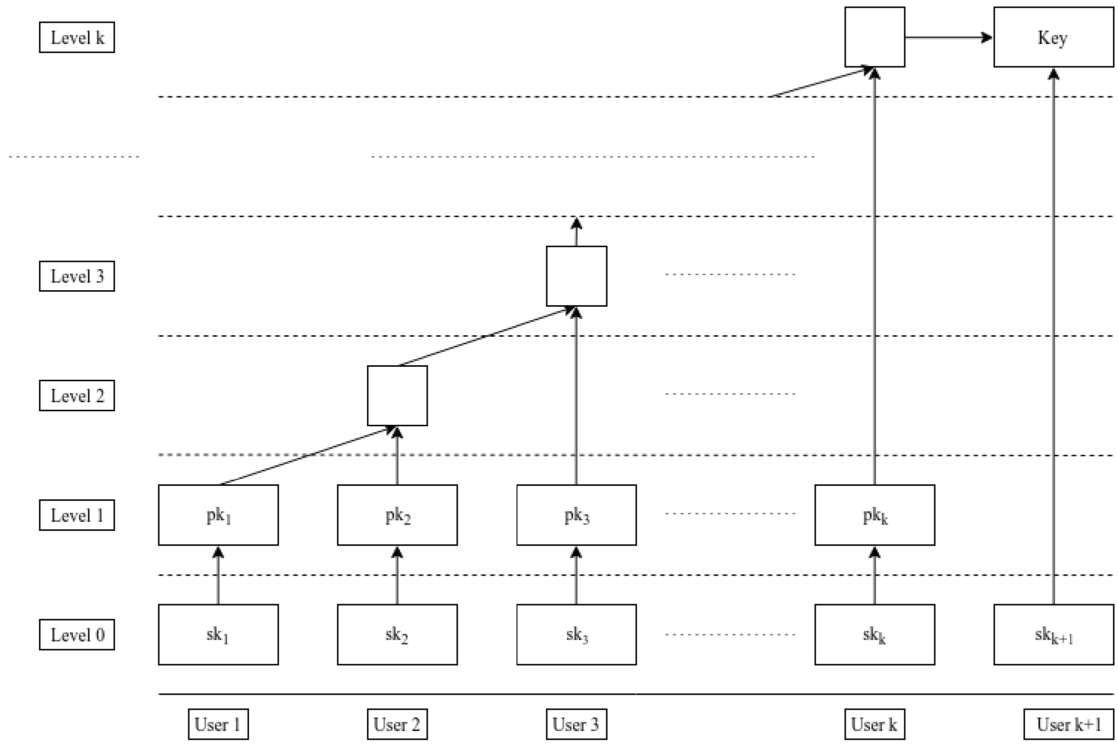

Figure 8: Diffie-Hellman key exchange for $k+1$ users via a level-k multilinear map

Coron et al. provide a C++ implementation of multipartite Diffie-Hellman via multilinear maps [14]. Their code generates multilinear map parameters and performs the key exchange as proof of the above concept.

6.2.2 Witness Encryption from Diffie-Hellman

In the following section we propose a primitive form of witness encryption to a Subset-Sum problem instance via cryptographic multilinear maps; the scheme is built from the above implementation of Diffie-Hellman multipartite key exchange. In addition, we provide an implementation of this scheme and give an example execution on a small, proof-of-concept Subset-Sum problem instance.

While our original goal was to implement the Liu et al. witness encryption scheme from Section 6.2, we found this to be unfeasible; our scheme here serves as a proof-of-concept workaround. For more details, see Chapter 7.

What follows is essentially repurposing the structure of Diffie-Hellman. Since we are unable to directly encode vectors (see Section 7.1), we instead generate random integers corresponding to the vectors and for them create level-1 encodings. From these level-1 encodings and a witness we generate a level-κ encoding (where κ is $\sum_{i\in|\Delta|} b_i$) that corresponds to the target sum and use it to encrypt a message. On the other side, **the decrypter uses his/her own knowledge of the witness to recompute the level-κ encoding**. In essence, we have replaced the addition operation on integer vectors with the mapping operation on encodings (integers).

Both the encrypter and the decrypter compute key K as follows:

$$K = s_{enc} = e(\underbrace{g_1, ...g_1}_{b_1}, \underbrace{g_2, ...g_2}_{b_2}, ..., \underbrace{g_{|I|}, ...g_{|I|}}_{b_{|I|}})$$

where g_i is a level-1 encoding. Each g_i is built from a level-0 random encoding α_i corresponding to $v_i \in \Delta$ and $w = b_1, b_2...$ is a witness to Δ. As before, $e()$ is the mapping operation. We refer to the key as s_{enc} because in a sense it represents an encoding of the Subset-Sum target vector s. While s is computed with vector addition, s_{enc} is computed by pairing encodings.

Figure 9 shows how witness encryption is built from Figure 8 (Diffie-Hellman). Encoding δ is at level $b_1 + b_2$ because it is obtained by mapping the level-1 encoding of vector 1 b_1 times and the level-1 encoding of vector 2 b_2 times - remember that multiplying two encodings results in an encoding at the sum of their levels. By the same logic, the target sum encoding s_{enc} is at level $\sum_{i\in|\Delta|} b_i$, and **this is the required level of multilinearity of our map parameters**. Both the encrypter and the decrypter compute s_{enc} in this way.

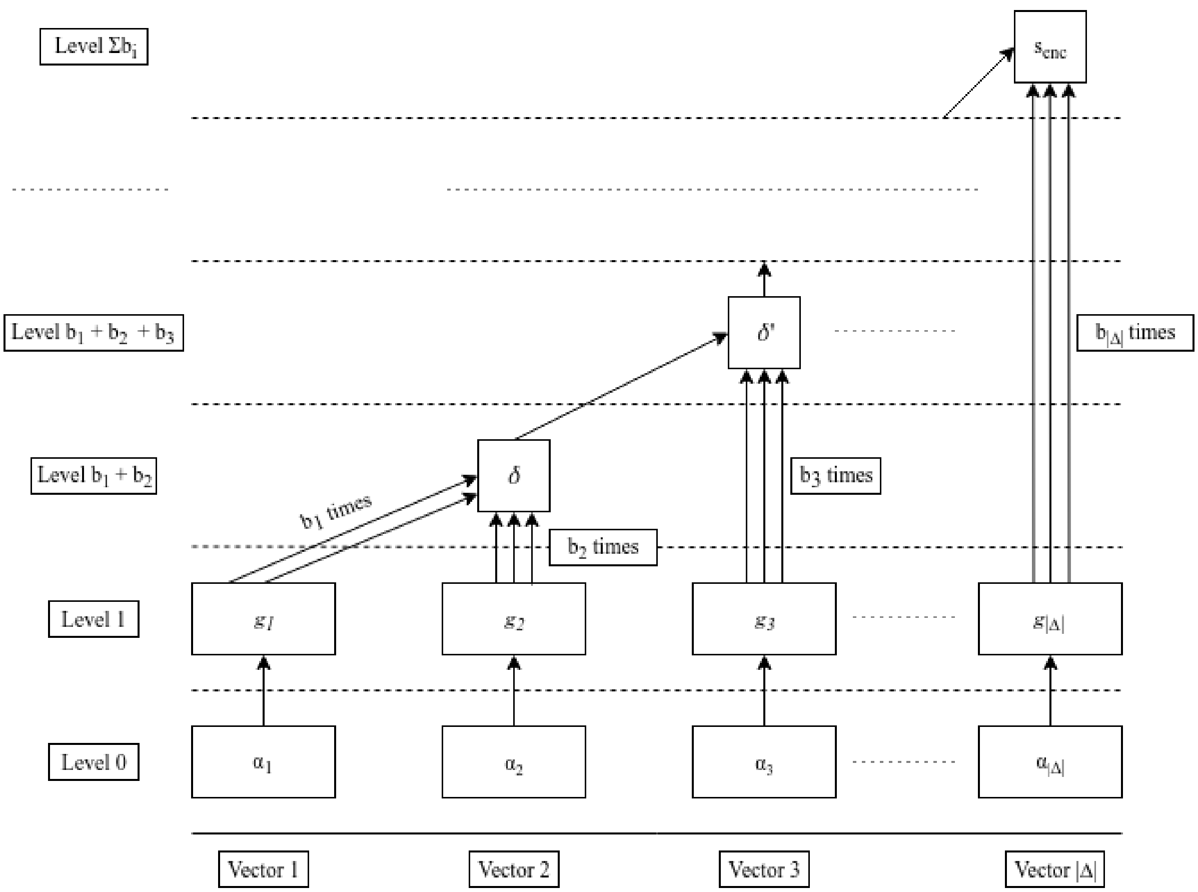

Figure 9: Computing s_{enc}, our encryption and decryption key

Algorithms 6 and 7 give encryption and decryption procedures built in this way. It's crucial to note that WE.encrypt takes as input a witness to the related Subset-Sum instance; this is not a characteristic of the scheme in Section 6.1 and has strong consequences on its utility for the timelock encryption use case (to be detailed later).

Input : (params), $\Delta = \{v_i : l_i\}_{i \in I}$, target s, witness $w = (b_1, b_1 ... b_{|I|})$, message m
Output: Ciphertext c corresponding to target sum s, randomness $[\alpha_1, \alpha_2, ..., \alpha_{|I|}]$

```
%generate random parameters
```

$[\alpha_1, \alpha_2, ..., \alpha_{|I|}] \leftarrow random$
$\mathrm{s}_{enc} \leftarrow 1$

```
%compute top-level encoding by pairing encodings representing each
 vector the amount of times dictated by the witness
```

for *i from 1 to* $|I|$ **do**
  $g_i \leftarrow params.encode(\alpha_i)$
end
for *i from 1 to* $|I|$ **do**
  for *j from 1 to* b_i **do**
    $s_{enc} \leftarrow s_{enc} * g_i$
  end
end

```
%use top-level encoding corresponding to witness to create and
 publish ciphertext
```

$c = m + s_{enc}$
return c, $[\alpha_1, \alpha_2, ..., \alpha_{|I|}]$

Algorithm 6: Pseudocode for witness encryption under our scheme

Input : (params), $\Delta = \{v_i : l_i\}_{i \in I}$, target s, witness $w = (b_1, b_1 ... b_{|I|})$, *randomness* $(\alpha_1, \alpha_2, ... \alpha_n)$
Output: Message m corresponding to ciphertext c

$s_{enc} \leftarrow 1$

```
%encode received randomness
```

for i *from 1 to* $|I|$ **do**
  $g_i \leftarrow params.encode(\alpha_i)$
end

```
%use witness to recompute top-level encoding
```

for i *from 1 to* $|I|$ **do**
  for j *from 1 to* b_i **do**
    $s_{enc} \leftarrow s_{enc} * g_i$
  end
end

```
%use recomputed top-level encoding to recover message from
 ciphertext
```

$m = c - s_{enc}$
return m

Algorithm 7: Pseudocode for witness decryption under our scheme

We implemented this in C++ using the CLT Diffie-Hellman implementation as a starting point. We now give an example of encryption and decryption on a small and simple Subset-Sum problem.

Witness Encryption Example

Define our Subset-Sum instance as follows:

- $\Delta = (1,0,0,0,0,0), (0,1,0,0,0,0)...(0,0,0,0,0,1)$, the standard basis vectors for $\mathbb{R}^6$. All l_is are 1.
- target vector $s = (1,1,1,1,1,1)$

Our witness w is trivial: $w = [1,1,1,1,1,1]$. It follows that our required level of multilinearity $(\sum_{i \in \Delta} b_i) = 6$. We use the CLT software to generate (**params**), a level-6 multilinear map set. The following is the command-line syntax for using our code implementing Algorithms 6 and 7:

Encryption:

```
./key-exchange encrypt [MESSAGE] [WITNESS]
```

Decryption:

```
./key-exchange decrypt [CIPHERTEXT] [WITNESS] [α1] [α2] ...
```

Where MESSAGE is an integer, WITNESS is w written as an integer, and ($\alpha 1$, $\alpha 2$...) are integers randomly generated and output by the encryption script.

CIPHERTEXT, and indeed any numerical representation of an encoding, is an element of $\mathbb{Z}_{x_0}$, where x_0 is a random product of primes generated by the CLT implementation.

Now we will show execution of our implementation on this trivial Δ. Firstly, we generate a multilinear map family of multilinearity level 6. Now we choose an integer as our message: take 12345678. Figure 10 shows the in-terminal result of using Algorithm 6 to encrypt message 12345678 to a Subset-Sum instance with witness [1,1,1,1,1,1] under our generated maps:

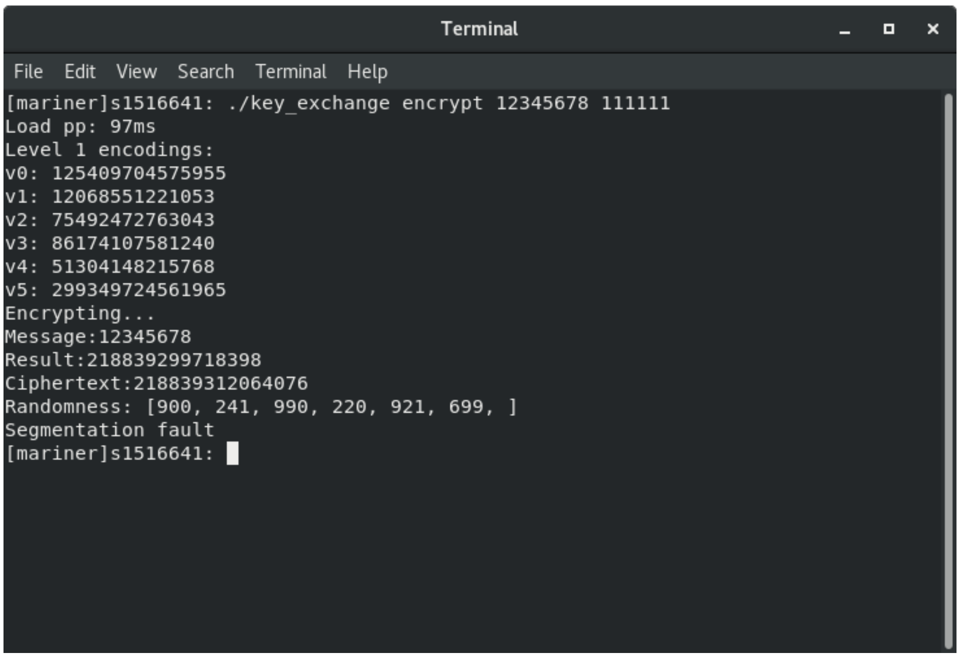

Figure 10: The result of encrypting message "12345678" to a simple Subset-Sum instance

Algorithm 6 tells us that encryption should output a ciphertext and generated randomness. Here we see:

- ciphertext: $c = 218839312064076$
- randomness: $[\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6] = [900, 241, 990, 220, 921, 699]$

The next stage is decryption. Algorithm 7 shows us that decryption requires as input the ciphertext, a witness, and the randomness used during encryption. Figure 11 shows the in-terminal result of this execution:

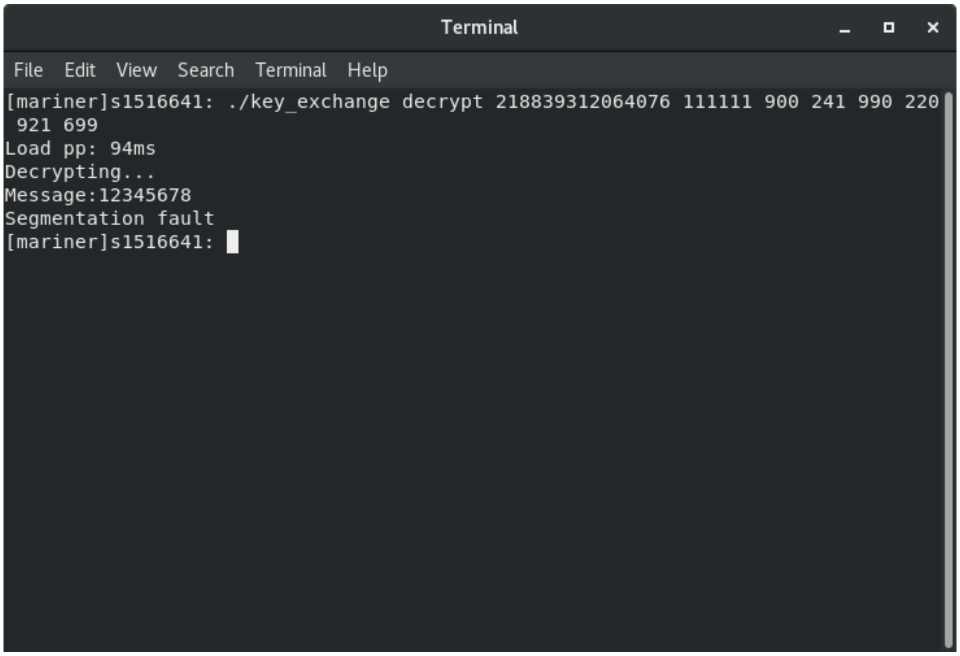

Figure 11: The result of decrypting the ciphertext generated in Figure 6

The decryption outputs 12345678 as the message, so correctness is satisfied:

$$WE.decrypt(WE.encrypt(\Delta, w, m), w) = m$$

Figure 12 details the process of computing s_{enc} in this example; Figure 9 contains the general case. Our vectors, the standard basis vectors for $\mathbb{R}^6$, are encoded as $\alpha_1 ... \alpha_6$. From $\alpha_1 ... \alpha_6$, $g_1 ... g_6$ are computed. Our witness is $w = [1, 1, 1, 1, 1, 1]$, so each g_i is paired *exactly once*. The intermediate encodings are denoted δ, δ'.... The result of this, s_{enc}, is our key for encryption and decryption.

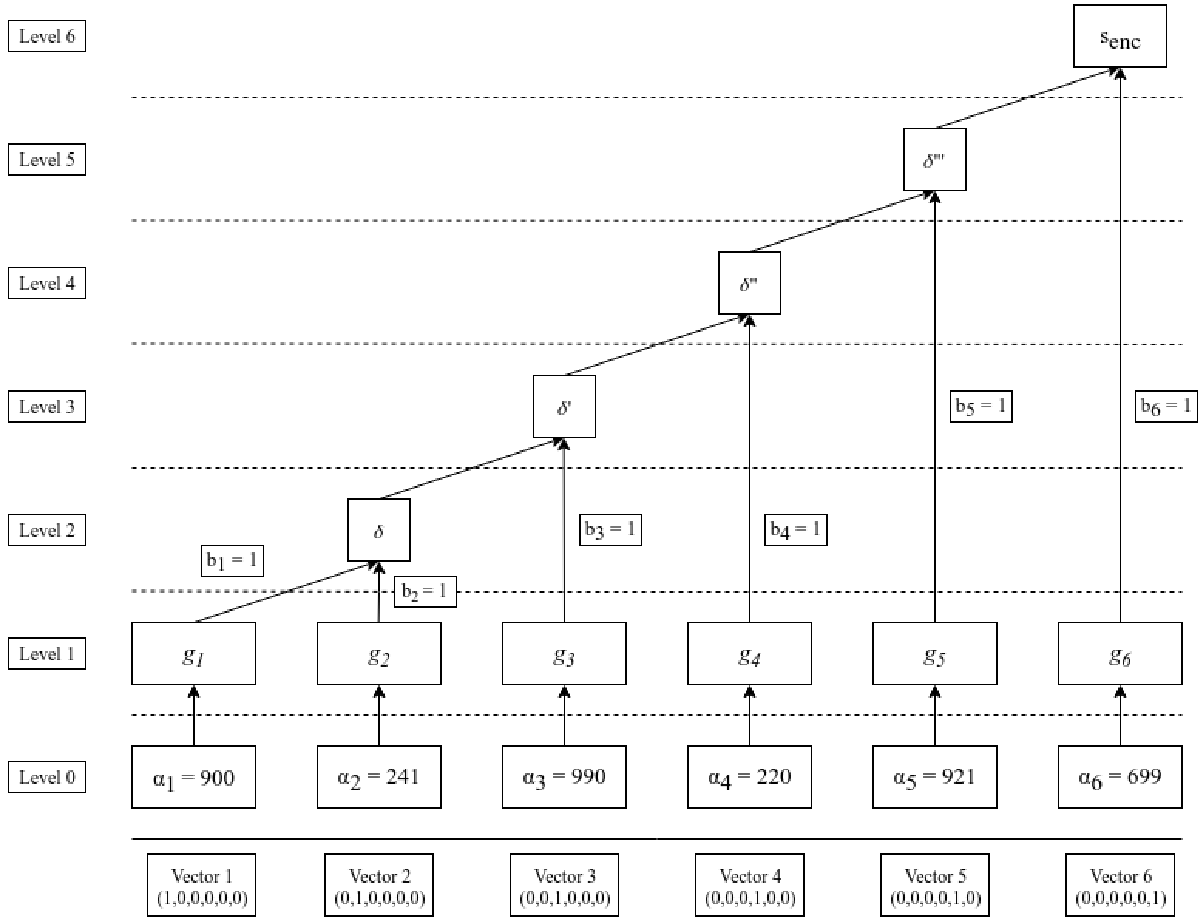

Figure 12: The map structure of our witness encryption example

7 Evaluation, Limitations and Future Work

In the preliminary stages of this project, we determined the original goals to be in the scope of an honours dissertation. The feasibility of achieving these goals within the timeframe (completing the problem conversion detailed in Chapter 5 and implementing witness encryption on the resulting Subset-Sum problem instance) was predicated on two main assumptions:

- The problem conversion work would be mostly theoretical, and the implementation aspects would be straightforward exercises in scripting.
- The witness encryption stage would be mostly about implementing the scheme in Section 6.1 using available multilinear map software.

Both of these assumptions turned out to be false. The problem conversion (Bitcoin blockchain $\rightarrow$ CNF-SAT $\rightarrow$ Subset-Sum) ended up being significantly more of an engineering problem than expected, with extensive code redesigns necessary to bring computational costs down to a feasible level. The witness encryption assumption was also incorrect; existing multilinear map software is extremely underdeveloped, with no generic out-of-the box models available. Additionally, even if more user-friendly software existed, current schemes are not capable of generating a multilinear map family of the size required to be compatible with Δ, our Subset-Sum problem instance.

As such, the original goals of the project were not all realized this year, with each step of the process being more difficult and exploratory than anticipated. Figure 13 overviews the original project structure; the green arrows indicate completion and the red arrows indicate impossibility under our approach. Note that this does not include our investigation into witness encryption built from Diffie-Hellman, as key aspects of our scheme make it not well suited to timelock encryption with the Bitcoin blockchain.

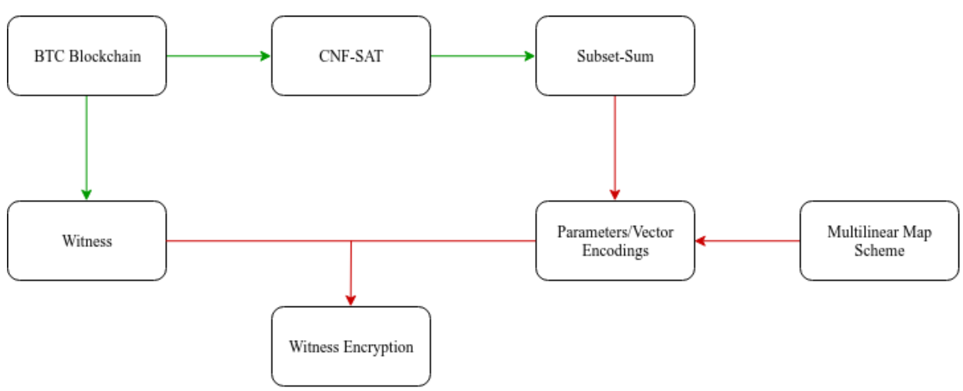

Figure 13: Feasibility of the original project progression

The following sections detail the ways in which our work this year could be extended and evaluated.

7.1 Multilinear Maps with Δ

At the outset, we were unaware of the embryonic state of multilinear map implementations. To build the Liu et al. witness encryption scheme, in an ideal world we would have an API capable of the following:

1. Generation of multilinear map parameters with given multilinearity level/security guarantee
2. Direct encoding of vectors as elements at any level of multilinearity
3. Encoding specific vectors as group generators during creation of parameters
4. User-level operations on encodings to compute encoding products etc.

The only code we could base our approach on was the CLT [7] scheme C++ implementation of Diffie-Hellman key exchange. This code is extremely narrow: while it allows for generation of multilinear map parameters and operations on encodings, it doesn't appear to allow for direct encodings. Diffie-Hellman is done using random sampling of the map parameters to obtain the secrets to be shared. We could not realize witness encryption as discussed in Section 6.1 with the utility provided by this code. There is also no way to encode vectors as group generators during parameter creation.

Additionally, the time and space costs of the CLT multilinear map implementation are both excessive. Table 6 contains data taken from *New Multilinear Maps over the Integers* [7] detailing time and space costs of generating small multilinear map parameters of multilinearity level 6:

Security Parameter	κ	**Setup Time**	**Storage cost**
52	6	5.9s	27MB
62	6	36s	175MB
72	6	583s	1.2GB
80	6	4528s	6.1GB

Table 6: Time and space costs for generating small multilinear map parameters under the CLT scheme

Note that increasing the security parameter (a security parameter λ denotes insecurity against an adversary with 2^λ clock cycles) results in near-problematic time and space costs for generation and storage **even for multilinearity level 6**. Δ, our Subset-Sum instance, would require a **multilinearity level greater than 1,000,000**. The resources required to realize this would be astronomical, both in computational power and storage.

The result of these concerns was a reduction of the problem scope we're interested in. Since it's impossible to use the Liu et al. witness encryption scheme with current multilinear map implementations for Δ, our witness encryption work aimed to perform witness encryption on small, token Subset-Sum instances. Future work would require fundamentally changing our approach to encrypting to Δ in one of the following ways:

- Encrypt to Δ with a witness encryption scheme not built from multilinear maps.
- Use a multilinear map scheme for which there is faster software with more utility than any that currently exist.
- Conduct a more rigorous analysis of the CLT multilinear map scheme and the existing codebase to determine if it can be extended/optimized enough to serve our purposes.

7.2 Our Witness Encryption Scheme

As described above, our witness encryption proposal built from Diffie-Hellman multipartite key exchange is a result of a reduction in problem scope; it's not intended to work for encrypting to Δ.

Algorithms 7-8 and the C++ implementation thereof are entirely proof-of-concept; the goal was witness encryption for Subset-Sum of any size. Development was constrained by inadequate multilinear map resources, as described in Section 7.1. The most fundamental way in which our scheme is not suited for timelock encryption is that **encryption requires the user to know a witness to the instance**. In the case where a witness is a blockchain, this presents a clear problem; the blockchain is public.

In addition, time constraints meant we were not able to perform any rigorous analysis of our scheme. The only property we propose is correctness (e.g $WE.dec(WE.enc(x, m), w) = m$) and even that has only been experimentally verified and not theoretically proven. Note, however, that **the hardness assumption proposed in [7]** and discussed in Section 6.2.1 should directly relate to the security of our scheme, as our witness encryption is built directly from Diffie-Hellman. Future work could consist of the following:

- Theoretical extension of building witness encryption from Diffie-Hellman (generalizing, adding functionality)
- Analysis of the effect of the range of generated randomness (currently: $\alpha \in [1, 1000]$) on security
- Analysis of the scheme's functionality in cases where multiple witnesses exist
- Analysis of how witness size affects security (e.g witnesses with small b_i values are easy to brute force)
- Analysis of the effects of known multilinear map attacks on security
- Proofs of desirable properties in a encryption scheme (or lack thereof)

7.3 The Real Bitcoin Blockchain: Predicting D_i and Hashing Transactions

Early on in the project, we identified two ways in which our construction of btcSAT differs from the true Bitcoin blockchain. However, as we expected the witness encryption portion of the project to be difficult and time-consuming, we moved forward with btcSAT as a prototype. Our work on witness encryption resulted in nothing compatible with Δ, our Subset-Sum problem instance; as such there was neither the need nor the time to refine btcSAT. The two differences between btcSAT and the real Bitcoin blockchain are as follows:

1. btcSAT uses set, dummy difficulty parameters
2. btcSAT uses a simplified version of the block chaining property $B_i = H(T_i, r_i, D_i, B_{i-1})$

Extension of our work to function as originally intended would require addressing both of these deviations: the second requires a simple restructuring of the SHA-256 component of our CNF-SAT, but the first could prove quite challenging.

Real future Bitcoin difficulty parameters are impossible to know. However, a timelock encryption requiring a length **a** blockchain to decrypt requires setting a difficulty value for each block that mimics the real value as closely as possible. Consider the following two cases:

- If the difficulty values mean that the block hash targets are too large, an adversary can brute force a fake blockchain faster than the "real" Bitcoin blockchain will be generated, allowing for early decryption.
- If the difficulty values mean that the targets are too small, a real length of Bitcoin blockchain *might not serve as a decryption tool.* This could lead to ciphertexts never being decrypted.

btcSAT contains no predictions: we use a constant difficulty value for proof-of-concept purposes. We leave it as future work to implement a sufficiently accurate prediction algorithm.

In addition, someone extending our scheme to functionally encrypt on the Bitcoin blockchain would have to take into account the way block hashes are computed. In the background we describe the blockchain's "chaining" property as $B_i = H(T_i, r_i, D_i, B_{i-1})$; the blockchain hashes a list of transactions, a counter value, a target generated from a difficulty parameter, and the previous block hash. This differs slightly from what we implemented. Our implementation assumed 512 bit input (with padding) to the first round of SHA-256: in other words, it built into SAT $H(B_{i-1}) = B_i$. It did not take into account the structure of the rest of the block. However, extending btcSAT to take into account the true chaining property should be straightforward; anyone wishing to extend our work would simply have to add variables to the CNF file corresponding to the rest of the relevant block data and use CNF structure of SHA-256 with the additional variables as well.

8 Summary

The timelock encryption scheme proposed by Liu et al. in *How to Build Time-lock Encryption* is built on two ideas: using a large public computation to create a computationally hard problem for which a publicly available solution will exist at a set time in the future, and realizing witness encryption on a problem of this nature. Specifically, the Bitcoin blockchain is used due to its predictable block generation and the size of its computational resources; an attacker would need to outpace the entire Bitcoin mining pool to decrypt early via brute force.

This work contains a partial implementation of the above scheme. We have:

1. Introduced and implemented in Python an original formulation of Bitcoin blockchain validity constraints as a Boolean satisfiability problem (section 5.1)

2. Implemented in Python an existing conversion from SAT to Subset-Sum (section 5.2)

3. Found Liu et al.'s proposed witness encryption scheme to be infeasible given the state of existing multilinear map resources (sections 6.1, 7.1)

4. Introduced and implemented in C++ an original form of witness encryption via multilinear maps that is suited to small Subset-Sum problem instances (section 6.2)

While our proposal is not compatible with problems on the scale of our created Subset-Sum instance Δ, it has sufficient functionality to be considered proof-of-concept. This project was ambitious, and every aspect of our results could benefit from further work; extensions of and adjustments to our ideas and implementations are proposed in Chapter 7.

9 Related Work

As this paper is theoretically and conceptually dense, we present the related work here at the end to improve readability.

Timelock encryption as described in this paper is quite ambitious; it relies on functional implementations of two nascent concepts, cryptographic multilinear maps and witness encryption. The academically complex nature of this approach is justified by theoretical characteristics of the end result that are difficult to achieve with conventional approaches, such as no significant decryption overhead. One approach emulating timelock encryption from conventional primitives is proposed in *Timed-Release of Self-Emerging Data using Distributed Hash Tables* [15]. Li et al. encrypt data conventionally, required a key to decrypt, but "hide" the keys in a distributed hash table such that they automatically appear after a set duration. Their paper consists of a timed-release cryptography model that they describe as "highly distributed", descriptions of attacks against their model, and an experimental evaluation of its resilience to these attacks. However, it contains limited analysis of efficiency and scalability of their ideas. The same authors, in a more recent paper [16], build on their existing ideas to introduce and deploy a time-release scheme on the Ethereum blockchain through the use of smart contracts. Similarly, the authors of *Keeping Time-Release Secrets through Smart Contracts* [22] propose an incentive-or-punishment based scheme for time-release cryptography.

Practical Witness Encryption for Algebraic Languages And How to Reply an Unknown Whistleblower [8] introduces a new construction under the witness encryption paradigm. Their construct allows for witness encryption on any NP language via a reduction to bilinear pairings. This could be useful to us; directly encrypting to CNF-SAT would allow circumvention of our translation to Subset-Sum. Derler et al. emphasize the efficiency of their approach, but evaluating its compatibility with our needs would require further analysis.

Our use case for timelock encryption was to fill a gap in E-cclesia, a voting protocol. Other constructions of self-tallying decentralized e-voting protocols exist; one such scheme was introduced in Li et al.'s *A Blockchain-based Self-tallying Voting Scheme in Decentralized IoT* [17]. As described in Section 2.2, a protocol such as this must guarantee *fairness* to its participants: a partial tally cannot be revealed and used to influence remaining voters. Li et al. investigate using zero-knowledge proofs to this end.

10 Appendices

10.1 Appendix A: Notation Reference

The following table explains the notation used in our descriptions of witness encryption and multilinear maps. Readers are encouraged to refer here if confused about object types etc. For notation appearing more than once, note the section it appears in.

Notation	Type	Explanation	Relevant Section
c	Group element (integer)	Ciphertext	Witness Encryption
m	Integer	Message to be encrypted	Witness Encryption
w	Vector	A Subset-Sum witness	Witness Encryption
α	Vector	Random vector	Witness Encryption (Liu Scheme)
α^{v_i}	Integer	$\alpha^{v_i} = \alpha_1^{v_1} * \alpha_2^{v_2}...$	Witness Encryption (Liu Scheme)
g_v	Group element (integer)	A group generator corresponding to vector v	Witness Encryption (Liu Scheme)
$g_s^{\alpha^s}$	Group element (integer)	Grp element corresponding to target vector	Witness Encryption (Liu Scheme)
$\alpha_1..\alpha_n$	Integers	Randomness generated for vector encoding	Witness Encryption (Our Scheme)
s_{enc}	Group element (integer)	encoding of target sum	Witness Encryption (Our Scheme)
$g_1...g_n$	Group elements (integers)	level-1 vector encodings	Witness Encryption (Our Scheme)
a	Vector	Object to be encoded	Multilinear Maps
b	Group element (integer)	Encoding of vector	Multilinear Maps
$e()$	Function	The mapping operation w.r.t multilinear maps	Multilinear Maps
κ	Integer	Level of multilinearity	Multilinear Maps
$p_1...p_n$	Integers	Randomly generated secret primes	Multilinear Maps
$g_1...g_n$	Integers	Randomly generated small secret primes	Multilinear Maps
$r_1...r_n$	Integers	Randomly generated noise	Multilinear Maps
x_0	Integer	$\Pi_1^n p_i$: product of primes	Multilinear Maps

Table 7: Notation reference for witness encryption/multilinear maps

10.2 Appendix B: Pseudocode for generation of Δ

Our Subset-Sum instance Δ is composed of four vector sets: u, ũ, v, and z. Algorithm 5, describing **makeVecs** v3, constructs u. Here we give pseudocode for construction of the other three sets.

$\tilde{u}_1 ... \tilde{u}_n$:

Data : btcSAT.cnf with clauses $C_1 ... C_k$
Input : int n, int k
Output: small integer lists $\tilde{u}_1 ... \tilde{u}_n$

```
%create empty lists for each vector
```

for *i from 1 to n* **do**
  $\tilde{u}_i \leftarrow []$
end

```
%for each negated variable in each clause, append the clause index
 to the relevant list
```

for *j from 1 to k* **do**
  for $\tilde{x}_m \in C_j$ **do**
    $\tilde{u}_m$.append($n + j$)
  end
end

Algorithm 8: Generating $\tilde{u}_1 ... \tilde{u}_n$

$v_1 ... v_k$:

Data : btcSAT.cnf with clauses $C_1 ... C_k$
Input : int n, int k
Output: small integer lists $v_1 ... v_k$

```
%create empty lists for each vector
```

for *i from 1 to k* **do**
  $v_i \leftarrow []$
end

```
%append predetermined clauses to each list
```

for *j from 1 to k* **do**
  v_j.append($n + j$) v_j.append($n + k + j$) **end**

Algorithm 9: Generating $v_1 ... v_k$

$z_1 ... z_k$:

Data : btcSAT.cnf with clauses $C_1 ... C_k$
Input : int n, int k
Output: small integer lists $z_1 ... z_k$

```
%create empty lists for each vector
```

for i *from 1 to* k **do**
  $z_i \leftarrow []$
end

```
%append predetermined clauses to each list
```

for j *from 1 to* k **do**
  z_j.append$(n + k + j)$ **end**

Algorithm 10: Generating $v_1 ... v_k$

References

[1] Bitcoin block time chart. BitInfoCharts. Online; accessed 1 April 2019.

[2] Dan Boneh and Alice Silverberg. Applications of multilinear forms to cryptography. *Contemporary Mathematics*, 324(1):71–90, 2003.

[3] Jung Hee Cheon, Kyoohyung Han, Changmin Lee, Hansol Ryu, and Damien Stehlé. Cryptanalysis of the multilinear map over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 3–12. Springer, 2015.

[4] William Clarkson. Time-lock cryptography, Dec 2013. Tufts University.

[5] Jean-Sébastien Coron, Moon Sung Lee, Tancrede Lepoint, and Mehdi Tibouchi. Cryptanalysis of ggh15 multilinear maps. In *Annual International Cryptology Conference*, pages 607–628. Springer, 2016.

[6] Jean-Sébastien Coron, Tancrede Lepoint, and Mehdi Tibouchi. Practical multilinear maps over the integers. In *Annual Cryptology Conference*, pages 476–493. Springer, 2013.

[7] Jean-Sébastien Coron, Tancrede Lepoint, and Mehdi Tibouchi. New multilinear maps over the integers. In *Annual Cryptology Conference*, pages 267–286. Springer, 2015.

[8] David Derler and Daniel Slamanig. Practical witness encryption for algebraic languages and how to reply an unknown whistleblower. *IACR Cryptology ePrint Archive*, 2015:1073, 2015.

[9] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[10] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–17. Springer, 2013.

[11] Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. Witness encryption and its applications. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 467–476. ACM, 2013.

[12] Craig Gentry, Sergey Gorbunov, and Shai Halevi. Graph-induced multilinear maps from lattices. In *Theory of Cryptography Conference*, pages 498–527. Springer, 2015.

[13] Steven Goldfeder. sha256final.txt.

[14] Tancrède Lepoint. new-multilinear-maps, Feb 2015. github.com/tlepoint/new-multilinear-maps.

[15] Chao Li and Balaji Palanisamy. Timed-release of self-emerging data using distributed hash tables. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2344–2351. IEEE, 2017.

[16] Chao Li and Balaji Palanisamy. Decentralized release of self-emerging data using smart contracts. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 213–220. IEEE, 2018.

[17] Yannan Li, Willy Susilo, Guomin Yang, Yong Yu, Dongxi Liu, and Mohsen Guizani. A blockchain-based self-tallying voting scheme in decentralized iot. *arXiv preprint arXiv:1902.03710*, 2019.

[18] Jia Liu, Tibor Jager, Saqib A. Kakvi, and Bogdan Warinschi. How to build time-lock encryption. *Des. Codes Cryptography*, 86(11):2549–2586, 2018.

[19] Lenka Marekova. E-cclesia: Self-tallying e-voting protocol in the uc framework, 2018. The University of Edinburgh.

[20] Martin Maurer and Vegard Nossum. sha256-sat-bitcoin, 2015. github.com/tlepoint/new-multilinear-maps.

[21] Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

[22] Jianting Ning, Hung Dang, Ruomu Hou, and Ee-Chien Chang. Keeping time-release secrets through smart contracts. 2018.

[23] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996.

[24] Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

[25] Wikipedia contributors. Boolean satisfiability problem, 2018. Online; accessed 1 April 2019.

[26] Wikipedia contributors. Cook-levin theorem, 2018. Online; accessed 1 April 2019.

[27] Wikipedia contributors. Np-completeness, 2018. Online; accessed 1 April 2019.

[28] Wikipedia contributors. Tseytin transformation, 2018. Online; accessed 1 April 2019.