Spring 1-1-2016

# Applications of Cryptographic Hash Functions

Jared Nishikawa
*University of Colorado at Boulder*, jared.nishikawa@gmail.com

Applications of Cryptographic Hash Functions

by

Jared Nishikawa

B.A., Willamette University, 2010

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirement for the degree of

Doctor of Philosophy

Department of Mathematics

2016

This thesis entitled:
Applications of Cryptographic Hash Functions
written by Jared Nishikawa
has been approved for the Department of Mathematics

_____

John Black

_____

David Grant

Date: _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Nishikawa, Jared (Ph.D., Mathematics)

Applications of Cryptographic Hash Functions

Thesis directed by Associate Professor John Black

This is a cryptography thesis, and the focus is on applications of cryptographic hash functions.

Hash functions, as a cryptographic primitive, are ubiquitous in their potential for application. In essence, a hash function distills data of any size into a fingerprint, where inverting this function is considered computationally infeasible.

After the introduction and preliminary work, the body of this paper is divided into chapters which report the results of three significant research problems.

The first problem is the study of building trapdoors into hash functions. How do we make a hash function that can be inverted with knowledge of a secret key? There are two main problems: the algorithm must not reveal the existence of the trapdoor, and use of the trapdoor must not reveal its existence. Finally, we offer a solution using elliptic curves.

The second problem is that of time-lock encryption, that is, how to encrypt a message such that it can not be opened before a given date. We start with the assumption of a centralized authority issuing time-lock encryption keys, and study ways for that party to certify intermediate authorities to do the same task, while retaining verifiability. We offer a solution using a structure developed for the Bitcoin wallet client.

The third is constructing a protocol for increased privacy for users on the Bitcoin network. A significant portion of this thesis is given over to the background of Bitcoin, as it also turned out to be helpful for our time-lock encryption solution. We construct a method for Bitcoin spending privacy using contracts, a simple but powerful mechanism for automating transactions and minimizing trust in the world of cryptocurrencies.

Dedication

To my brothers, Gregory and Regan Nishikawa.

Acknowledgements

I would like to express my sincere gratitude to my primary advisor Dr. John Black for his constant support, suggestions, and critiques throughout this process. His patience and optimism played no small part in inspiring me to complete this project. It has been a pleasure getting to know and work with such an intelligent and amicable person.

Also, I thank Dr. David Grant and Dr. Katherine Stange for their input and expert advice. I owe a special word of thanks to Dr. Stange for her assistance in my search for employment this past year.

Thank you also to Dr. Robert Tubbs and Dr. Dirk Grunwald for agreeing to serve on my committee.

Finally, I am thankful for my friends and family for their unending support and kind words during my time as a graduate student.

# Contents

**List of Tables** **x**

**List of Figures** **xii**

**1 Introduction** **1**

1.1 Trust . . . . . 1

1.1.1 Trapdoors . . . . . 2

1.1.2 Time-Lock Encryption . . . . . 2

1.1.3 Bitcoin and Anonymity . . . . . 3

1.2 Scope . . . . . 4

1.3 Outline . . . . . 4

**2 Preliminaries** **6**

2.1 Cryptographic Security . . . . . 6

2.2 Integer Factorization . . . . . 7

2.3 The Discrete Logarithm Problem (DLP) . . . . . 8

2.4 Elliptic Curves . . . . . 9

2.4.1 Elliptic Curves Over Finite Fields . . . . . 13

2.4.2 ECDLP . . . . . 14

2.5 Symmetric-key Cryptography . . . . . 15

2.5.1 DES . . . . . 15

2.5.2 AES . . . 15
2.6 Public-key Cryptography . . . 17
2.6.1 RSA Cryptosystem . . . 18
2.6.2 The ElGamal Cryptosystem . . . 19
2.7 Signature Schemes . . . 21
2.7.1 RSA Signature Scheme . . . 21
2.7.2 ECDSA . . . 22
2.8 Cryptographic Hash Functions . . . 22
2.8.1 Definitions . . . 22
2.8.2 The Birthday Paradox . . . 24
2.8.3 Applications of Hash Functions . . . 27
2.8.4 Examples . . . 31
2.8.5 MACs and HMACs . . . 35
2.9 Bitcoin . . . 36
2.9.1 A Public Ledger . . . 36
2.9.2 Double-Spending . . . 37
2.9.3 Proof-of-work . . . 38
2.9.4 Blocks . . . 39
2.9.5 Miners . . . 40
2.9.6 Verifications . . . 41
2.9.7 51% Attack . . . 42
2.9.8 The Profitability of Mining . . . 44

**3 Trapdoors** **46**

3.1 Overview . . . 46
3.2 Using a Trapdoor Hash Function . . . 48
3.3 Motivation and Previous Work . . . 49
3.4 Goal . . . 51

3.5 Cryptographic Hash Functions . . . 51
3.6 Trapdoor Hash Functions . . . 52
3.6.1 Simple Example . . . 53
3.6.2 Trapdoor Hash Function Based on RSA . . . 54
3.6.3 Trapdoor Hash Function Based on VSH . . . 55
3.7 Contribution: Elliptic Curve Trapdoor Variant . . . 56
3.8 GHS Weil Descent Attack . . . 59
3.9 Summary . . . 61

**4 Time-lock** **63**
4.1 Overview . . . 63
4.2 Definitions and Background . . . 64
4.2.1 Hash Chains . . . 64
4.2.2 Time-lock Encryption . . . 65
4.2.3 Previous Research . . . 67
4.3 The Beacon . . . 69
4.3.1 The Extended Beacon . . . 71
4.3.2 The Ideal Solution . . . 71
4.3.3 Hierarchical Deterministic Wallets . . . 73
4.4 Main Result . . . 76
4.4.1 Contribution . . . 76
4.4.2 Construction 1 . . . 76
4.4.3 Construction 2 . . . 80
4.5 Summary . . . 82

**5 Bitcoin Mixing with Contracts** **84**
5.1 Overview . . . 84
5.2 Transactions . . . 85

5.2.1 Structure . . . 85
5.2.2 Inputs and Outputs . . . 86
5.2.3 Transaction Fees . . . 86
5.3 Scripts and Contracts . . . 87
5.4 Description of the Problem . . . 95
5.5 Previous Work . . . 95
5.5.1 Bitcoin Exchanges . . . 95
5.5.2 Mixing Services . . . 95
5.5.3 CoinJoin . . . 97
5.5.4 Other . . . 98
5.6 Contribution . . . 99
5.6.1 Phase 1: Meet-and-Freeze . . . 100
5.6.2 Phase 2: Coin Melt . . . 101
5.6.3 Persistence . . . 104
5.6.4 Key Exchange . . . 104
5.6.5 Analysis of DoS Attacks . . . 106
5.7 Summary . . . 107

**6 Conclusion and Final Remarks** **108**

**Bibliography** **109**

**A Script** **113**

# List of Tables


2.1 Computational Feasibility . . . . . 7

2.2 Python 3 Code: Computing a Discrete Logarithm . . . . . 8

2.3 Sage Code: Computing a Discrete Log over an Elliptic Curve . . . . . 14

2.4 Public and Private Key Usage . . . . . 18

2.5 RSA Cryptosystem . . . . . 19

2.6 ElGamal Cryptosystem . . . . . 20

2.7 RSA Signature Scheme . . . . . 21

2.8 Elliptic Curve Digital Signature Algorithm . . . . . 23

2.9 Cryptographic Hash Function Properties . . . . . 24

2.10 Google's suggestion for creating a secure password. . . . . 29

2.11 Initial Hash Values for **sha1** . . . . . 33

2.12 Key Schedule for **sha1** . . . . . 33

2.13 **sha1** Pseudocode . . . . . 34

2.14 Python 3 Code: Computing a Proof-of-Work . . . . . 38

3.1 Cryptographic Hash Function Properties, Revisited . . . . . 52

3.2 Trapdoor Hash Function Properties . . . . . 53

3.3 Unified Pseudocode for VSH . . . . . 56

3.4 Pseudocode for ECTA . . . . . 58

4.1 Alternate Approach to Hash Chaining . . . . . 66

4.2 HD Wallet Key Derivation: Algorithm . . . . . 74

5.1 Transaction Structure . . . . . 86

5.2 CoinFreeze Locking Script . . . . . 92

5.3 Two Person Coin-Freeze Locking Script . . . . . 100

5.4 Coin Melt Locking Script . . . . . 102

5.5 CoinJoin Locking Script . . . . . 103

5.6 CoinJoin++ Locking Script . . . . . 103

# List of Figures


2.1 Elliptic Curve of the Form $y^2 = x^3 + ax + b$, with $a = -1$ and $b = 1$. . . . . 9

2.2 Elliptic Curve of the Form $y^2 = x^3 + ax + b$, with $a = -4$ and $b = 0$. . . . . 10

2.3 Feistel Cipher, Encryption and Decryption . . . . . 16

2.4 Merkle Tree . . . . . 30

2.5 The Block Chain . . . . . 40

4.1 Parallel Hash Chains . . . . . 71

4.2 Commutative Parallel Chains . . . . . 72

4.3 RSA Parallel Chains . . . . . 73

4.4 HD Wallet Key Derivation: Illustration . . . . . 75

4.5 The Extended Beacon: Attempt 1 . . . . . 77

4.6 The Extended Beacon: Attempt 2 . . . . . 78

4.7 The Extended Beacon: Attempt 3 . . . . . 80

4.8 The Extended Beacon: Variation . . . . . 82

5.1 Evaluating a Script . . . . . 89

5.2 Evaluating a Script (cont.) . . . . . 90

5.3 Mixing . . . . . 96

5.4 CoinJoin . . . . . 98

A.1 Constants . . . . . 114

A.2 Flow Control Operators . . . . . 114

A.3 Stack Operators . . . . . 115
A.4 Bit Logic Operators . . . . . 115
A.5 Arithmetic Operators . . . . . 116
A.6 Crypto Operators . . . . . 117
A.7 Locktime Operators . . . . . 117

# Chapter 1

# Introduction

The research for this dissertation is ultimately rooted in the study of cryptographic hash functions, although our final conclusions extend beyond that. This introduction is intended to give the reader a brief overview of the ground that will be covered, and hopefully provide context and motivation for the results. Chapter 2 will contain technical details of the relevant cryptographic primitives and schemes involved in the rest of the paper.

## 1.1 Trust

Cryptography, lightly put, is the study of secure communication.

If everybody trusted everybody, there would be no need for secure communication. As it is, if Alice wants to send a message to Bob that she knows might intercepted, she would like some assurance that only Bob can understand it. For example, if Alice shifts each of the letters down the alphabet a fixed amount (encryption), she easily turns the message into incomprehensible garbage. If Bob knows the correct amount, he can shift back to restore it (decryption). This method is believed to have been used by Julius Caesar and is a simple example of a substitution cipher. Of course, this is insecure, and we now have computers to speed up the process of breaking a cipher (cryptanalysis). As an aside, we adopt the standard in this paper of using Alice and Bob to represent two (usually neutral) parties

who wish to communicate securely (or enact some other cryptographic protocol) with each other. We will name an eavesdropper Eve, as the need arises. In practice, these entities are probably not people, but instead computers, ATMs, internet browsers, website servers, and so on.

### 1.1.1 Trapdoors

Since a community of people will almost certainly give rise to hierarchies of trust, it is often convenient to have trapdoors built into the system. For example, all the graduate students at a university have keys to their own offices, but the management possesses a literal master key that opens all the doors in the building. The keys and locks could also be adjusted so the graduate students can be trusted to have access to their offices as well as a given community room. In this case, it is desirable and robust to have not just hierarchies of access, but also *knowledge* of said hierarchies. It would be another case entirely if a university's president promised the students he could not access their e-mail, but actually possessed a secret key (digital, not literal) that unlocked everyone's inbox. Speaking of trust, how much do we trust that power is not being exploited like this?

The truth is that hidden trapdoors are a major question for researchers in the cryptography community, and articles are released every year about security agencies privately installing trapdoors in public security algorithms. Of course, there has been no evidence yet of a hash function with a trapdoor (more detail in Section 2.8), and one goal of this thesis is to create a model of a trapdoor cryptographic hash function.

### 1.1.2 Time-Lock Encryption

Another open problem in cryptography is that of timed encryption. Alice may want to send Bob a message on January 1, with the stipulation that he will not be able to decrypt it before July 1. Of course, it is given that no more action should be needed from Alice, otherwise she would just wait until July 1 to give Bob the message. One example of when

this might be useful: if Alice is bidding in a sealed auction, she submits a time-locked bid to the auctioneer. The auctioneer collects everyone's secure bids (the encryption prevents the auctioneer from colluding with a bidder), and can only open them after the bid ends. Furthermore, another concept this introduces is a commitment: Alice *commits* to the bid by encrypting and sending it to the auctioneer. After he receives the bid, she cannot change it.

A simple solution to time-lock encryption: on January 1, Alice sends Bob the encrypted message and sends her lawyer the decryption key with the instruction not to give Bob the key until July 1. Here, the lawyer is playing the role of a Trusted Third Party (TTP), and we will see in Chapter 4 the problems with this solution. We do not approach the time-lock problem directly; instead, we start by assuming a central authority has the power to issue time-lock encryption keys and ask whether that entity can authorize intermediate authorities to issue encryption keys, while maintaining a trusted mathematical relationship with the parent keys. The second goal of this thesis is to provide a solution to this problem.

### 1.1.3 Bitcoin and Anonymity

Our third and final contribution is related to Bitcoin.

Historically, cryptography has been primarily concerned with secure communication. As more and more human activities take place in the digital world, we must find new and innovative ways to ensure (or minimize the need for) trust between strangers on the internet. Bitcoin, a *cryptocurrency*, is a form of money that does not require either physical coins or banks. The transition away from an authoritative body (like a bank) is known as *decentralization* and is desirable because it eliminates a single point of failure. In general, trust in humans and human-controlled companies can be reduced by increasing automation based on secure mathematical principles. Bitcoin also brings to the community one of the newest tools of trust: the *block chain* (more in Section 2.9).

The block chain effectively distributes trust to a network of users, such that in order for a group of people to act maliciously, they would need more than 50% of the computational

power available to the entire network. (The truth is more complicated than that, but this is an overview). Now, one necessary evil of the block chain is that it is a public record of all monetary transactions. Even though it is possible to hide behind a digital address, it would be desirable to have *untraceable* bitcoins. One method to achieve partial untraceability: users pool their money together in one large transaction and send the output money to new addresses (input and output addresses cannot be linked this way); this method is called *mixing*. There are problems with this and there are many papers already which seek to extend, alter, or revamp this approach. The final goal of this thesis is to construct a new, feasible, and readily-compatible method for mixing Bitcoins using *contracts*.

## 1.2 Scope

With the results in this paper, we hope to extend the body of knowledge of cryptography. We believe our approaches to be new, interesting, and worthy of review. Furthermore, if our results inspire more research in these topics, then we will be more than satisfied.

## 1.3 Outline

In Chapter 2, we will discuss the necessary background for this paper. This chapter covers a broad range of topics, but we will give more weight and space to those which feature prominently in the rest of the text.

In Chapter 3, we will discuss in detail the relevant aspects of trapdoors in cryptography, and describe our construction of a trapdoor hash function.

Continuing, Chapter 4 introduces the concept, history, and current state of time-lock encryption. At this point, we will introduce an interesting cryptographic primitive used in the Bitcoin client. Obviously, it was developed and deployed with only Bitcoin wallets in mind; nevertheless, we use it for a novel approach to timed encryption.

Finally, with Chapter 5, we break down the intricacies of the Bitcoin protocol, including

scripts and contracts. We will address the history and need for Bitcoin mixing and describe our approach using contracts.

Chapter 6 will contain our summary and final remarks.

# Chapter 2

# Preliminaries

Since this paper does cover so many different topics, we will spend this chapter reviewing the necessary definitions, cryptographic primitives, schemes, and cryptosystems that we will take for granted in later chapters.

Furthermore, we give ample space to the history and development of Bitcoin, so we can delve deeper into some of the more complicated nuances of the transaction protocols in Chapter 5.

## 2.1 Cryptographic Security

To begin with, we want to give some idea about how strong an encryption key (a bitstring) of length $n$ is, just based on a brute-force search.

A key should be private, but if the length is known, then a *brute-force* attack means the adversary simply just guesses and tries every key of length $n$. Since keys are bitstrings, there are $2^n$ possible keys. Table 2.1 gives a rough estimate of how long it would take for a (not-particularly powerful) home computer to compute a given number of operations. For example, a 1.2 GHz processor runs around 1.2 billion cycles per second. As a lower bound, we suppose that each operation costs exactly 1 cycle (probably not true for actual computations). As a further lower bound, assume even that a single key can be guessed with

exactly 1 cycle.

| Number of operations | Feasibility |
|---|---|
| $2^{30}$ | 0.8 seconds |
| $2^{40}$ | 13 minutes |
| $2^{50}$ | 9 days |
| $2^{60}$ | 25 years (feasible with parallel processing) |
| $2^{70}$ | 25 millennia (infeasible) |
| $2^{80}$ | 25,000 millennia (infeasible++) |

Table 2.1: Computational Feasibility

This table shows why cryptographic protocols that are reported to have "80 bits of security" are considered secure.

In practice, the numbers can vary greatly. DES (Data Encryption Standard) was originally shipped with 56-bit encryption keys, and the EFF (Electronic Frontier Foundation) built a machine (Deep Crack) in 1998 that used a brute force attack to decrypt a 56-bit DES key in less than 24 hours. [17]

This table is intended as a general rule of thumb, and we will refer to it as necessary when discussing computational feasibility. We will also used the word *intractable* to describe a problem whose solution cannot be computed with a feasible amount of time or resources.

## 2.2 Integer Factorization

Speaking of intractability, secure algorithms frequently make use of difficult math problems. These are problems that are considered infeasible given current mathematical theory and computational resources.

*Factorization* is, of course, the decomposition of an integer into its prime components, and it is considered infeasible if the integer is at least 309 decimal digits (1024 bits). We will make use of this in the RSA cryptosystem in Section 2.6.1.

```
python: def dlptest():
python:     cur = time.time()
python:     n = 1
python:     p = 8000009
python:     target = 993256
python:     while( pow(2,n,p) != target):
python:             n += 1
python:     print( time.time() - cur, n)
python: dlptest()
output: 22.828850984573364 2991024
```

Table 2.2: Python 3 Code: Computing a Discrete Logarithm

## 2.3 The Discrete Logarithm Problem (DLP)

Another hard mathematical problem frequently used in security algorithms is the Discrete Logarithm Problem.

**Problem 2.3.0.1** (DLP)**.** *For a finite group $G$, let $a$ be a non-identity element, and let $\langle a \rangle$ be the subgroup generated by $a$. If $b$ is in $\langle a \rangle$, then by definition there exists some integer $x$ such that $a^x = b$.*

*The Discrete Logarithm Problem asks: given $a$ and $b$, what is the value of $x$?*

**Example 2.3.0.1.** *For small groups, finding a solution to DLP is not very difficult. Let $G = (\mathbb{Z}/p\mathbb{Z})^\star$, with $p = 8000009$ (which is prime). Let $a = 2$, and let $b = 993256$. Running a simple while loop in Python (Table 2.2) returns a result in 22.83 seconds on my 2.66 GHz computer.*

*Evidently,*

$$2^{2991024} \equiv 993256 \pmod{8000009}.$$

For large enough groups, (it is generally believed that) computing a discrete logarithm is a computationally infeasible task, which is why exponentiation—easy by comparison—is often used in cryptographic protocols.

## 2.4 Elliptic Curves

Elliptic curves are a double-edged sword. On the one hand, elliptic curve cryptography requires shorter keys (in fact, this is the primary advantage of elliptic curve cryptography: 160-bit ECC keys are considered stronger than 1024-bit RSA keys [11]). On the other hand, elliptic curve research is much younger, so it could be that mathematical theory has yet to develop simpler attacks on elliptic curves. Either way, use of elliptic curves in security algorithms is becoming fairly ubiquitous.

The theory of elliptic curves is deep and and still worthy of much research. For a thorough and rigorous treatment of the subject, Silverman's books are an invaluable resource [47] [48].

For our purposes, we will content ourselves with the basic definitions.

**Definition 2.4.1** (Elliptic Curves over $\mathbb{R}$ or $\mathbb{C}$)**.** *If $a, b$ are constants in $\mathbb{R}$ (or $\mathbb{C}$) that satisfy $4a^3 + 27b^2 \neq 0$, then an **elliptic curve over $\mathbb{R}$ (or $\mathbb{C}$)** is the set $E$ of points $(x, y)$ that satisfy the following equation:*

$$y^2 = x^3 + ax + b,$$

*along with a point $\mathcal{O}$, called the point at infinity.*

See Figures 2.1 and 2.2 for graphs of two example elliptic curves.

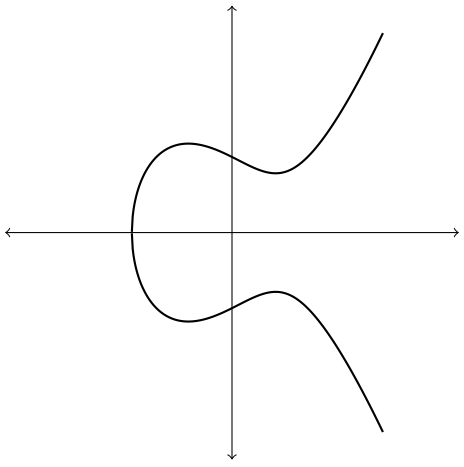

Figure 2.1: Elliptic Curve of the Form $y^2 = x^3 + ax + b$, with $a = -1$ and $b = 1$.

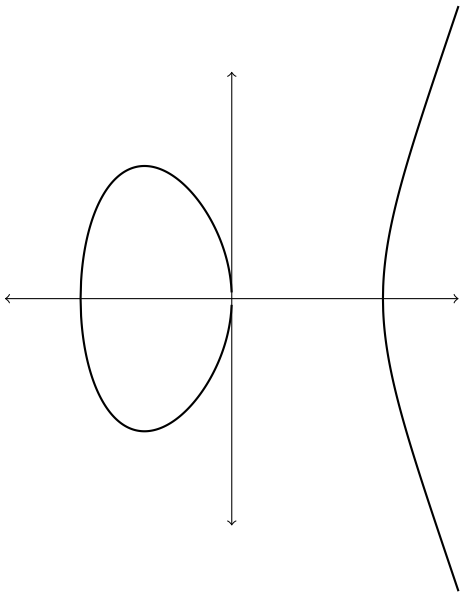

Figure 2.2: Elliptic Curve of the Form $y^2 = x^3 + ax + b$, with $a = -4$ and $b = 0$.

We can define an addition on the points of $E$ which makes $(E, +)$ into an abelian group, with $\mathcal{O}$ as the identity element.

**Definition 2.4.2** (Point Addition)**.** *Suppose $P$ and $Q$ are points on an elliptic curve $E$, but not the point at infinity $\mathcal{O}$. Write $P = (x_1, y_1)$ and $Q = (x_2, y_2)$. There is a line $L$ through points $P$ and $Q$. If $L$ is not vertical, then it intersects $E$ at a third point, call it $R'$. If we reflect $R'$ across the $x$-axis to obtain $R$, then define elliptic curve addition by $P + Q = R$.*

*If $L$ is vertical, then define $P + Q = \mathcal{O}$.*

*If $Q = \mathcal{O}$, then define $P + \mathcal{O} = P$.*

This definition requires some justification, which we will address in three cases, regarding the coordinate relationship between $P = (x_1, y_1)$ and $Q = (x_2, y_2)$. The argumentation is well-known, though concise formulations are readily available in Stinson [49] and Washington [53]:

*Case 1.* $x_1 \neq x_2$ and $y_1 \neq y_2$.

The line $L$ through $P$ and $Q$ can be written as

$$y = mx + v,$$

where

$$m = \frac{y_2 - y_1}{x_2 - x_1},$$

and

$$v = y_1 - mx_1.$$

To determine the third point on the intersection between $E$ and $L$, we substitute the equation for $L$ into the elliptic curve equation:

$$\begin{aligned}
(mx + v)^2 &= x^3 + ax + b \\
x^3 + ax + b - (mx + v)^2 &= 0 \\
x^3 - (mx)^2 - 2mxv - v^2 + ax + b &= 0 \\
x^3 - m^2x^2 + (a - 2mv)x + b - v^2 &= 0 \qquad (2.1)
\end{aligned}$$

Since this is a cubic with at least two real roots ($x_1$ and $x_2$), the third root must also be real. Let $x_3$ be the third real root. That is, we could rewrite the cubic as:

$$(x - x_1)(x - x_2)(x - x_3) = 0.$$

The quadratic coefficient will be $-(x_1 + x_2 + x_3)$. Comparison with Equation 2.1 gives us:

$$x_3 = m^2 - x_1 - x_2.$$

The coordinates of $R'$ will be $(x_3, y')$, and we can again use $m$ to find a relationship

between $y'$ and the points we already know:

$$m = \frac{y' - y_1}{x_3 - x_1},$$

for example. So,

$$y' = m(x_3 - x_1) + y_1.$$

Since, ultimately, we wanted $R$ (the reflection of $R'$ across the $x$-axis), we now have $P + Q = R$, with:

$$R = (x_3, m(x_1 - x_3) - y_1),$$

where

$$x_3 = m^2 - x_1 - x_2,$$

and

$$m = \frac{y_2 - y_1}{x_2 - x_1}.$$

*Case 2.* $x_1 = x_2$ and $y_1 = -y_2$.

Note that this covers the case where $y_1 = y_2 = 0$. In Case 2, simply define $P + Q = \mathcal{O}$. Thus, $P$ and $Q$ are inverse elements. The points $(x, 0)$ on an elliptic curve $E$ are self inverse.

*Case 3.* $x_1 = x_2$ and $y_1 = y_2$.

Assume, of course, that $y_1 \neq 0$ (otherwise, Case 2 applies). Since we are adding a point $P$ to itself, the line $L$ we are considering is the tangent at $P$. Simple implicit differentiation of the elliptic curve equation gives us:

$$2y\frac{dy}{dx} = 3x^2 + a.$$

Therefore, the slope of $L$ at $P = (x_1, y_1)$ is

$$m = \frac{3x_1^2 + a}{2y_1}.$$

Computing $R'$ and reflecting to obtain $R$ is the same as in Case 1.

Clearly, addition on $E$ is closed and commutative, and every element has an inverse. Showing addition is also associative is outside the scope of our goals and definitions, but ultimately, $E$, together with the elliptic point addition that we just defined, forms an abelian group.

### 2.4.1 Elliptic Curves Over Finite Fields

Because we are mainly concerned with the application of elliptic curves in a cryptographic setting, we would like to consider the case where the base field has finite characteristic.

Fortunately, the definition for elliptic curves over $\mathbb{F}_q$ (where $q = p^n$ for some prime $p$) is mostly the same.

**Definition 2.4.3** (Elliptic Curves over $\mathbb{F}_q$)**.** *If $a, b$ are constants in $\mathbb{F}_q$ that satisfy $4a^3+27b^2 \not\equiv 0 \pmod{p}$, then a **elliptic curve over** $\mathbb{F}_q$ is the set $E$ of points $(x, y)$ that satisfy the following congruence:*

$$y^2 \equiv x^3 + ax + b \pmod{p},$$

*along with a point $\mathcal{O}$, called the point at infinity.*

The derivation for is addition laws are analogous. We summarize the rules again, for brevity:

For the point $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, (with $Q \neq -P$), we define $P + Q = R = (x_3, y_3)$. We can deduce:

$$x_3 = m^2 - x_1 - x_2,$$

and

$$y_3 = m(x_1 - x_3) - y_1,$$

```
sage: F.<a> = GF(37^2, 'a')
sage: E = EllipticCurve(F, [1,1])
sage: P = E(25*a + 16, 15*a + 7)
sage: Q = E(36*a + 32, 5*a + 12)
sage: discrete_log(Q, P, P.order(), operation = '+')
output: 39
```

Table 2.3: Sage Code: Computing a Discrete Log over an Elliptic Curve

where

$$m = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1}, & \text{if } P \neq Q \\ \frac{3x_1^2 + a}{2y_1}, & \text{if } P = Q \end{cases}.$$

Of course, if $Q = -P$, then $P + Q = \mathcal{O}$.

### 2.4.2 ECDLP

Now that we have a new type of finite group (other than $(\mathbb{Z}/n\mathbb{Z})^\star$), let us formulate the DLP for elliptic curves:

**Problem 2.4.3.1** (ECDLP)**.** *For an elliptic curve $E$ over $\mathbb{F}_q$, let $P$ be a non-identity element, and let $\langle P \rangle$ be the subgroup generated by $P$. If $Q$ is in $\langle P \rangle$, then by definition there exists some integer $x$ such that $xP = Q$.*

*The Elliptic Curve Discrete Logarithm Problem asks: given $E$, $\mathbb{F}_q, P, Q$, what is the value of $x$?*

Note: integer multiplication of a point is simply defined to be repeated elliptic point addition.

Table 2.3 shows an example computation in Sage of a discrete logarithm on an elliptic curve. Line 1 names a generator $a$, and Line 2 specifies the elliptic curve to be used. This computation took just under 5 seconds.

Evidently, in the given curve:

$$Q = 39P.$$

## 2.5 Symmetric-key Cryptography

We now step away from the discussion of difficult mathematical problems to address practical encryption. Encryption should be *fast*, and unfortunately, exponentiation, and elliptic curve multiplication are *slow*. For this reason, encryption that relies only on addition, bit rotation, and XOR (operations that are fast for a computer) is desirable.

If Alice and Bob possess a shared key, then there are standard methods of fast, secure encryption.

### 2.5.1 DES

DES (Data Encryption Standard) is constructed as a 16-round *Feistel Cipher* (Figure 2.3). The parallel structure ensures that encryption and decryption are identical algorithms (which cuts down space requirements).

Standardized in 1977, DES was implemented with 56-bit security keys. As we now know (Table 2.1), 56 bits is no longer acceptable as a security standard.

#### Attacks and Triple DES

To increase the security of DES without creating a brand new block cipher, the first idea that strikes is to simply encrypt twice using two keys. As it turns out, however, iterating DES twice still only provides 56 bits of security. Attacking a double-DES encryption from both ends is only twice as hard (computationally speaking) as attacking a single iteration of DES. This is called a *meet-in-the-middle* attack. To combat this problem, triple DES effectively increases the security to 112 bits without needing to modify the block cipher.

### 2.5.2 AES

The Advanced Encryption Standard (AES) was published in 1998 to replace DES. Instead of a Feistel Cipher, AES incorporates a *substitution-permutation network*. AES operates on

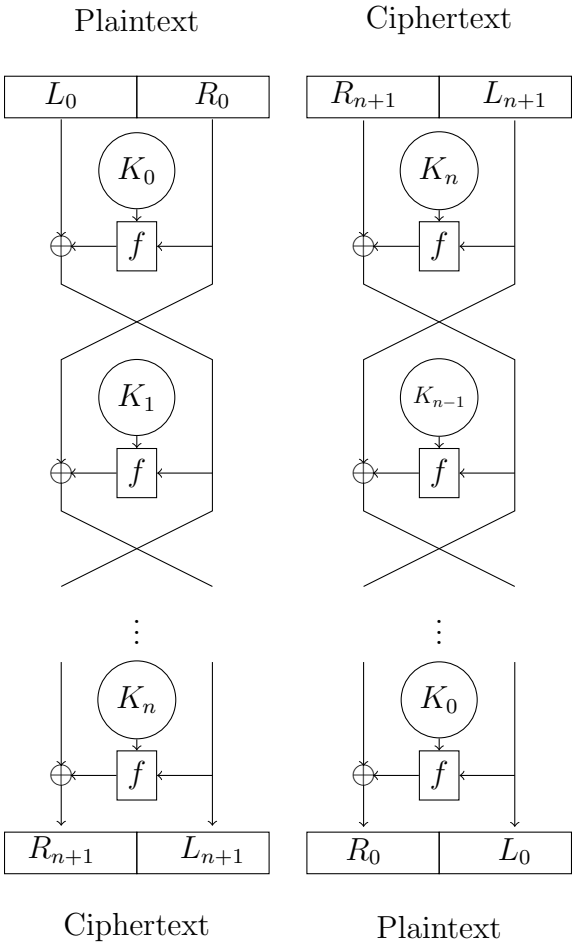


Figure 2.3: Feistel Cipher, Encryption and Decryption

an internal state matrix. Loosely speaking, the algorithm iterates rounds of the following operations:

- Replace bytes according to an embedded lookup table
- Shift rows
- Mix columns
- Add key

AES uses a 128-bit key, which provides acceptable cryptographic security.

**Remark 1.** *Why do we have both symmetric and public-key cryptography? Symmetric encryption (AES and DES, for example) is ARX-based (Addition, Rotation, and XOR) and is therefore orders of magnitude faster than public-key encryption. Intuitively, since public-key encryption is based on difficult mathematical operations (like exponentiation or elliptic curve multiplication), it is slower to compute.*

*Since symmetric key encryption is so much faster than asymmetric encryption, why would we use the latter? The reason is that in order for two parties to begin symmetric key encryption, they must use a public-key encryption system to exchange a symmetric key in the first place.*

The next section discusses public-key cryptography.

## 2.6 Public-key Cryptography

The theory behind a *public key cryptosystem* is that it is possible for Alice to generate a public and private key pair $(e, d)$ (where $e$ is private, and $d$ is public), such that encryption with $e$ can be decrypted with $d$. In this way, Alice can publicly announce $e$, so anyone can send Alice encrypted messages knowing that she is the only one with the corresponding decryption key. Furthermore, this idea of key pairs will be useful in signature schemes: Alice

| Private key | Public key |
|---|---|
| Alice | Bob |
| $d$ | $e$ |
| Decrypt | Encrypt |
| Sign | Verify |

Table 2.4: Public and Private Key Usage

can sign with her private key $d$, such that anyone else can check the validity of the signature with her public key $e$. See Table 2.4 for a handy guide regarding the major uses of public and private keys.

A function that is easy to evaluate but difficult to invert is called a *one-way function*. In the context of a public-key cryptosystem, it is desirable that encryption be an injective, one-way function (but actually, decryption can be performed easily with the correct private key).

### 2.6.1 RSA Cryptosystem

For example, let $p$ and $q$ be primes, and let $N = pq$ (in this case, $N$ is called an *RSA modulus*). Furthermore, suppose $a$ is an integer in $(\mathbb{Z}/N\mathbb{Z})^\star$. Define

$$f : \mathbb{Z}/N\mathbb{Z} \to \mathbb{Z}/N\mathbb{Z}$$

by

$$f : x \mapsto x^a \pmod{N}.$$

It is generally believed that $f$ is a one-way function, provided $p$ and $q$ are sufficiently large. Even if $a$ and $x^a$ are known, there is currently no known feasible way to recover $x$. Factoring $N$ would be sufficient, although it is not known whether or not it is necessary. That is,

Alice wants Bob to send her an encrypted message. Alice and Bob agree on an RSA modulus $N = pq$ (only Alice knows $p$ and $q$), and Alice announces her public key, an integer $e$ in $[1, N-1]$ (Alice picks $e$ relatively prime to $N$). Furthermore, since Alice knows the factorization of $N$, she can compute $d = e^{-1} \pmod{\phi(N)}$. Note: this means that $ed = 1 + k\phi(N)$.

*Encryption.*
For Bob to encrypt a message $M$ with Alice's public key $e$, he computes

$$C = M^e \pmod{N},$$

and sends $C$ to Alice.

*Decryption.*
For Alice to decrypt $C$, she computes

$$\begin{aligned} C^d \pmod{N} &= (M^e)^d \pmod{N} \\ &= M^{ed} \pmod{N} \\ &= M^{1+k\phi(N)} \pmod{N} \\ &= MM^{\phi(N)} \pmod{N} \\ &\equiv M \pmod{N}. \end{aligned}$$

The last line is true because $(\mathbb{Z}/N\mathbb{Z})^\star$ has group size $\phi(N)$, and therefore $M^{\phi(N)} \equiv 1 \pmod{N}$.

Table 2.5: RSA Cryptosystem

recovering $x$ is *no harder* than finding the factorization of $N$.

This gives rise to the RSA public-key cryptosystem, Table 2.5

### 2.6.2 The ElGamal Cryptosystem

The ElGamal cryptosystem is based on the hardness of the DLP. See Table 2.6 for the full details of encryption and decryption.

If an adversary intercepts $p, g, h, C_1$, and $C_2$, recovering $a$ or $b$ is presumed hard based on the best current algorithms for solving the DLP. Since $s$ and $m$ are both secret, seeing $C_2$ gives no information about either.

This method of exchanging a shared secret with two private keys is known as the *Diffie-Hellman key exchange*.

Alice wants Bob to send her an encrypted message. They must agree on a prime $p$ such that the DLP in $(\mathbb{Z}/p\mathbb{Z})^\star$ is intractable, and a generator $g$ of the group.
Alice selects $a$ randomly in $(\mathbb{Z}/p\mathbb{Z})^\star$, and computes $h = g^a$. She publishes $h$, but retains $a$ as her private key.

*Encryption.*
For Bob to encrypt a message $m$, first assume for the ease of notation that $m$ can be represented as an element of $(\mathbb{Z}/p\mathbb{Z})^\star$. (The only consequence is that might mean the message must be encrypted in blocks, which is not an issue).
Bob selects $b$ randomly in $(\mathbb{Z}/p\mathbb{Z})^\star$, and computes

$$C_1 = g^b \pmod{p},$$

$$s = h^b \pmod{p},$$

and

$$C_2 = ms \pmod{p}.$$

Note: $s$ is known as a *shared secret* as Alice will be able to compute $s = C_1^a$.
Bob sends the ciphertext $(C_1, C_2)$ to Alice.

*Decryption.*
Alice computes the shared secret $s = C_1^a \pmod{p}$. She then computes $C_2 s^{-1}$, and we can see that

$$\begin{aligned} C_2 s^{-1} &= (ms)s^{-1} \pmod{p} \\ &= m \pmod{p} \end{aligned}$$

Table 2.6: ElGamal Cryptosystem

Alice wants to send a signed message to Bob. They must agree on an RSA modulus $n$ (but Alice knows the factorization $n = pq$), as well as Alice's public exponent $e$. Alice retains the private exponent $d$ such that $de \equiv 1 \pmod{\phi(n)}$.

*Signing.*
To sign a message $M$, Alice computes:

$$\mathbf{sig}(M) = M^d \pmod{n}.$$

*Verification.*
To verify a signature $s$ for a message $M$, Bob computes

$$\mathbf{ver}(s, M) = \mathbf{True} \iff M \equiv s^e \pmod{n}.$$

Table 2.7: RSA Signature Scheme

The ElGamal cryptosystem can be implemented with elliptic curves as well, and the security is based on the assumed intractability of the ECDLP.

## 2.7 Signature Schemes

Signature schemes are a vital part of digital communication. A signature scheme allows one user to prove ownership of a private key to another user without revealing that key. The theory behind these sorts of "proofs" often rely on the computational infeasibility of mathematical problems like the DLP (or ECDLP).

A signature scheme will consist of a signing algorithm **sig**, and a verification algorithm **ver**. The inputs will vary, but in general, the signing algorithm will accept as input a message and a private key (and output a signature), and the verification algorithm will accept a message and signature (and output **True** or **False**).

### 2.7.1 RSA Signature Scheme

The RSA encryption algorithm naturally lends itself to a simple signature scheme based. See Table 2.7.

### 2.7.2 ECDSA

We introduce the Elliptic Curve Digital Signature Algorithm (ECDSA) because of its use in signing Bitcoin transactions. It is originally based on the equivalently formulated Digital Signature Algorithm (DSA), although ECDSA is favored over DSA because (as is the case with elliptic curve cryptography) the key sizes are significantly smaller. Table 2.8 is adopted from Stinson [49].

## 2.8 Cryptographic Hash Functions

We are now ready to introduce a very important object of study: *hash functions*. A solid understanding of this section is necessary before embarking on the Bitcoin section 2.9.

Informally, a cryptographic hash function returns a unique identifier or "fingerprint" for the input (or *message*). Of course, there are messages that return the same fingerprint, so the use of the word "unique" is questionable. Nevertheless, for a good cryptographic hash function, *finding* messages that return the same output (or *digest*) is considered computationally infeasible given current resources (Table 2.1). Furthermore, changing the message even slightly should result in unpredictable changes in the digest. Digests are typically represented as fixed-length bitstrings (at least 160 bits in length). Because of the (near) uniqueness and short length of the digests, hash functions are ubiquitous in use as cryptographic primitives. Essentially, a hash function compresses a piece of data (which could be quite large) into a manageably small bitstring.

### 2.8.1 Definitions

**Definition 2.8.1** (Hash Function)**.** *Formally, a* ***hash function*** $H$ *is a map*

$$H : \{0,1\}^* \to \{0,1\}^n,$$

Alice wants to send a signed message to Bob. They must first agree on an elliptic curve $E$ defined over $\mathbb{F}_p$ ($p$ a large prime). They must also agree on a base point $G$, with prime order $q$ such that the DLP in $\langle G \rangle$ is intractable.

Alice generates key pair $(m, Q)$ (where $1 \leq m \leq q-1$ and $Q = mG$). Then $m$ is the private key, and $Q$ (along with $p, q, E, G$) is the public key.

*Signing.*
To sign a message $M$ (as we will see later, it is actually better to sign $x = \mathbf{hash}(M)$ where **hash** could be any standard cryptographic hash function, like **sha1**), Alice generates a session private key $k$ ($1 \leq k \leq q-1$), and computes:

$$\mathbf{sig}(x, k) = (r, s),$$

where

$$\begin{aligned} (u, v) &= kG \\ r &\equiv u \pmod{q} \\ s &\equiv k^{-1}(x + mr) \pmod{q}. \end{aligned}$$

(If either $r = 0$ or $s = 0$, a new value of $k$ should be generated).

*Verification.*
If Bob wants to verify the validity of $(r, s)$ as the signature for $M$, he first calculates $x = \mathbf{hash}(M)$, then performs the following computations:

$$\begin{aligned} w &\equiv s^{-1} \pmod{q} \\ i &\equiv wx \pmod{q} \\ j &\equiv wr \pmod{q} \\ (u, v) &= iG + jQ \\ \mathbf{ver}(x, (r, s)) = \mathbf{True} &\iff r \equiv u \pmod{q}. \end{aligned}$$

Table 2.8: Elliptic Curve Digital Signature Algorithm

*for some $n$.*

Clearly, since $H$ maps an infinite set to a finite set, digests are not unique to a given message. We call two messages with the same digest a *collision*. Here are some natural problems that arise when talking about hash functions.

**Problem 2.8.1.1** (Collision)**.** *Given a hash function* ***hash****, find two messages $x_1, x_2$ such that* $\boldsymbol{hash}(x_1) = \boldsymbol{hash}(x_2)$.

**Problem 2.8.1.2** (Pre-image)**.** *Given a hash function and a digest $y$, find a message $x$ such that* $\boldsymbol{hash}(x) = y$.

**Problem 2.8.1.3** (Second Pre-image)**.** *Given a hash function and a message $x_1$, find a message $x_2$, such that* $\boldsymbol{hash}(x_1) = \boldsymbol{hash}(x_2)$.

Since the main property we will want to exploit is that of computational uniqueness, we define security parameters that a *cryptographic hash function* should have. See Table 2.9.

- Collision resistance: It should be computationally infeasible to find distinct $x, x'$ with $\mathbf{hash}(x) = \mathbf{hash}(x')$.
- Pre-image Resistance: For a given digest $y$, it should be computationally infeasible to find an $x$ with $\mathbf{hash}(x) = y$. We also refer to this as the "one-way property."
- Second Pre-image Resistance: For a given $x$, it should be computationally infeasible to find $x' \neq x$ with $\mathbf{hash}(x) = \mathbf{hash}(x')$.

Table 2.9: Cryptographic Hash Function Properties

### 2.8.2 The Birthday Paradox

Analyzing the difficulty of finding a collision simply by a brute-force search is related to the well-known "birthday paradox." Probabilistically, it is clear that in a gathering of $n$ people, the probability that at least two have the same birthday is:

$$P(n) = 1 - \frac{365!/(365-n)!}{365^n},$$

(one minus the probability that no two people have the same birthday). Evidently,

$$P(23) \approx .507.$$

In other words, in a gathering of 23 people, the probability that at least two people have the same birthday is more than 50%. The paradox lies in the fact that the low number of people needed is counterintuitive.

If we formulate birthdays as digests and people as messages, than the map from people to birthdays is a hash function (note: not cryptographically secure).

The following theorem and argument regarding the expected number of tries to find a collision is paraphrased from Stinson's cryptography textbook [49], although it is a well known line of reasoning:

**Theorem 2.8.1.1.** *If we throw $Q$ balls into $M$ bins, then the probability that no bin contains more than one ball (no collisions) is:*

$$P = \left(\frac{M-1}{M}\right)\left(\frac{M-2}{M}\right)\cdots\left(\frac{M-Q+1}{M}\right).$$

Note: if $Q = 23$ and $M = 365$, we have the birthday paradox equation. The reason we formulate it like this is that we would like to fix $P$ and solve for $Q$:

$$P = \left(1-\frac{1}{M}\right)\left(1-\frac{2}{M}\right)\cdots\left(1-\frac{Q-1}{M}\right) = \prod_{i=1}^{Q-1}\left(1-\frac{i}{M}\right).$$

If $x$ is a small real number, $1 - x \approx e^{-x}$ (from the series expansion for $e^{-x}$). Therefore,

$$\begin{aligned} P &\approx \prod_{i=1}^{Q-1} e^{\frac{-i}{M}} \\ &= e^{-\sum_{i=1}^{Q-1}\frac{i}{M}} \\ &= e^{\frac{-Q(Q-1)}{2M}} \end{aligned}$$

We can approximate the probability of finding at least one collision to be:

$$1 - e^{\frac{-Q(Q-1)}{2M}}.$$

If we call this probability $\epsilon$, then we can solve for $Q$ as a function of $M$ and $\epsilon$:

$$\begin{aligned} e^{\frac{-Q(Q-1)}{2M}} &\approx 1 - \epsilon \\ \frac{-Q(Q-1)}{2M} &\approx \ln(1-\epsilon) \\ Q^2 - Q &\approx 2M \ln \frac{1}{1-\epsilon}. \end{aligned}$$

As $M$ gets large (for example, for a hash function whose output is 160 bits, $M = 2^{160}$), we can ignore the smaller $Q$ term:

$$Q \approx \sqrt{2M \ln \frac{1}{1-\epsilon}}.$$

If we take $\epsilon = 0.5$ (i.e., if we are hoping for a 50% chance of finding a collision), then the approximation is:

$$Q \approx 1.17\sqrt{M}.$$

So, the number of hashes needed to have a probability of 50% chance of finding a collision is proportional to the square root of the size of the hash space. This immediately tells us why the standard hash length is 160. To brute force a collision would take (proportional to) $2^{80}$ computations, which is out of the realm of computational feasibility (Table 2.1). As a side note (even though the approximations are not as indicative when the numbers are small), it just so happens that:

$$1.17\sqrt{365} \approx 22.35.$$

### 2.8.3 Applications of Hash Functions

Cryptographic hash functions are immediately useful in a number of different protocols. There are a few major reasons for this:

- Short digests
- Uniqueness of digests (relative to computational feasibility)
- Speed of algorithm

#### Data Integrity

The first obvious application of cryptographic hash functions is as a tool for file integrity. If Alice sends Bob a file, and Bob wants to be sure there was no corruption during the file transfer, Alice can also send Bob the digest of the file. When Bob computes the digest of the file he received, he checks it with the digest that Alice sent. If they match, then the file transfer was uncorrupted. Now, we have not discussed the possibility of an adversary Eve posing as Alice in which case Eve could send Bob both a bogus file and the (matching) bogus digest. In this sort of case, it would be desirable for Alice to also sign the message using a signature scheme.

#### Signatures

On that note, hash functions also make signatures significantly quicker and in some cases, more secure. A simple signature scheme uses RSA exponents. Alice has a private key $d$, and she signs a message $m$ with $m^d$. If $m$ is large, than this computation is slow. Especially if Alice is signing (say) a multi-page document, then it standard practice to sign each page individually. Furthermore, if an adversary sees Alice sign message $m_1$ and message $m_2$, then the signed message $(m_1 m_2)^d$ could be forged with simple multiplication.

Instead, Alice should first compute the digest $h = \mathbf{hash}(m)$, and then sign with $h^d$. Now, if an adversary sees the signed messages $h_1^d$ and $h_2^d$, no information is gained with

$(h_1 h_2)^d$, since it is computationally infeasible to find a pre-image with $h_1 h_2$ as a digest. Essentially, using a hash function in the middle of signing creates a firewall based on the one-way property. Also, signing a digest is much faster than signing a large message.

### Commitment Schemes

Cryptographic hash functions also provide a way to enforce simple commitment schemes. Alice wants Bob to commit to a value that she can guess before Bob reveals it. The classic example is: how do two people flip a coin over the phone?

*The Commitment Phase.* For a single bit commitment (akin to flipping a coin), Bob pseudo-randomly generates a bitstring $c$ of length 80, the last bit of which is the coin flip. He computes the digest $h$ and sends it to Alice. He tells Alice that she must guess the last digit of $c$. On the one hand, because of the one-way property of the hash function, Alice cannot directly compute the pre-image. However, she also cannot brute-force guess the message, considering that Bob specifically picked an input of length 80 (which is cryptographically secure, as far as brute-force computation is concerned). Furthermore, now that Alice has $h$, Bob cannot change $c$, because altering it even slightly would change the digest.

*The Reveal Phase.* Alice guesses the last bit of $c$. Bob reveals $c$, and also whether Alice was right or wrong. Alice can trust that Bob did not tamper with the message, and Bob can trust that Alice did not brute-force guess it.

### Password Hashing

Alice logs into her e-mail account. The server asks for a username and password and Alice provides it. The server checks to make sure the username and password match a username and password combination on file in the database. If this check succeeds, the server lets Alice log in.

This is bad. An e-mail server absolutely should not store its users' passwords. Instead, if the server stored the hash of Alice's password, it would still be simple to verify, and in

| |
|---|
| Use at least 8 characters. Don't use a password from another site, or something too obvious like your pet's name. |

Table 2.10: Google's suggestion for creating a secure password.

the event of leakage, an adversary would only gain access to the hash of Alice's password, and *not* the password itself. The truth is that even leaking hashed passwords is dangerous because many user passwords are simple-to-guess combinations of words and numbers. In actual fact, using a password like

`H0wn0wbl20wnc0w`

is just as insecure as using

`hownowbrowncow`.

Dictionary attacks compute hashes of common word combinations (as in `hownowbrowncow`) and check against the leaked hashes. Intelligently, they use variants of words with number and letter replacement (as in `H0wn0wbl20wnc0w`), and password hash leakage events have shown us that hackers can crack up to 90% of the leaked passwords using these attacks [22]. Note: Google rates the password `H0wn0wbl20wnc0w` as "Strong" and Table 2.10 gives Google's suggestions for creating secure passwords.

As an aside: our suggestion for creating secure passwords is 11 (or more) pseudo-randomly generated characters.

### Source of Pseudo-Randomness

Hash functions can also be used in the generation of pseudo-random bits. See [36] for a discussion of hash functions, randomness extractors, and universal hashing.

Since hash function constructions are typically lightweight and fast (based on ARX models: addition, rotation, and XOR), generating billions of pseudo-random bits from a small seed is easy.

### Merkle Trees

A *Merkle tree* is a structure that produces a hash that depends on a set of messages $m_1, \ldots, m_n$. A nice property of Merkle trees is that verifying inclusion of a single $m_i$ in the hash is fast.

To construct a Merkle tree with messages $m_1, \ldots, m_n$, each message is hashed. The hashes are paired, concatenated, and hashed again (if there is an odd number of hashes on any given row, the last hash is duplicated).

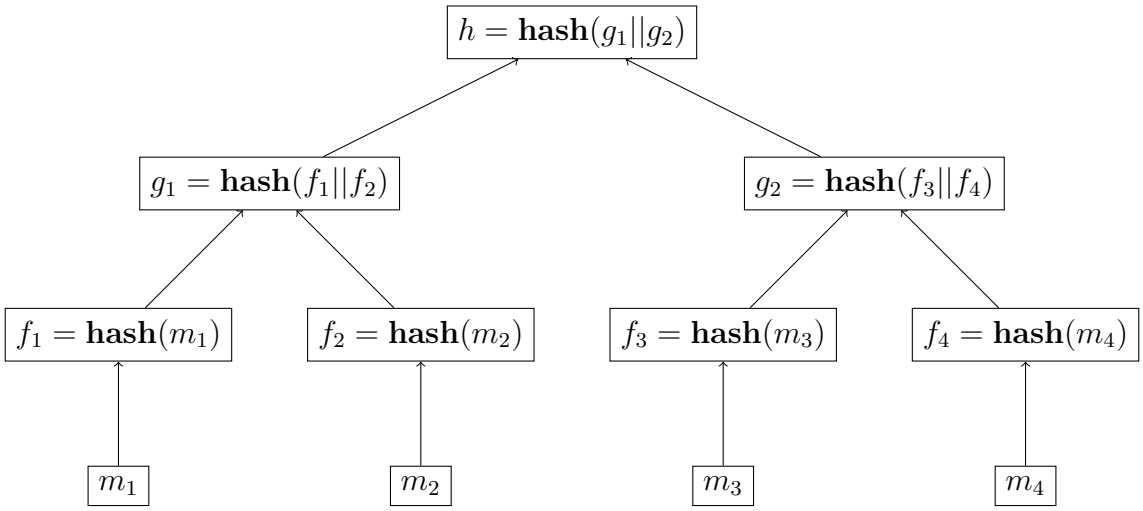


Figure 2.4: Merkle Tree

In Figure 2.4, $h$ is called the *Merkle root*. Suppose Bob, in order to commit to all the messages $m_1, \ldots, m_4$, gives the Merkle root $h$ to Alice. If Alice wants proof that $m_4$ was included in the commitment, Bob can provide her with $f_3$ and $g_1$. Alice computes:

$$\mathbf{hash}(g1||\mathbf{hash}(f_3||\mathbf{hash}(m_4))),$$

which evaluates to $h$ (—— denotes concatenation). For Merkle trees that correspond to a large number of messages, the verifier needs at most $\log_2 n$ hashes, and this sequence of hashes is called a *Merkle path*.

Merkle trees will be important later on, when we are discussing Bitcoin.

### 2.8.4 Examples

#### Merkle-Damgård Construction

Here we discuss Merkle-Damgård constructions. In not so many words, a Merkle-Damgård hash function starts with an initialization vector $IV$ (algorithm-dependent) as its internal state, breaks the message into blocks, and iteratively uses the blocks to update the internal state, using a one-way compression function. The final internal state (possibly after a finalization function) is output as the hash.

#### SHA1 and MD5

There is a tradeoff between cryptographic security and speed. That is, there are hash functions which are provably reducible to computationally infeasible problems (like the DLP, or other hard problems), and there are hash functions which are fast, but whose constructions are more or less ad hoc. For example, **sha1** is an *ARX algorithm*, in that it uses only the operations Addition, Rotation, and XOR. These operations are very fast for computers to process (compared to multiplication, exponentiation, or elliptic addition). On the other hand, the only security "proof" is simply that it has not been broken yet (this is actually semantically false, because as we will see, there are theoretical attacks within computational feasibility).

In practice, the fast cryptographic hash functions are given preference. In this section, we will discuss two of the most ubiquitous (and therefore significant) hash functions in standard use.

The Secure Hash Algorithm family is a sequence of cryptographic hash functions published by the National Institute of Standards and Technology (NIST).

- 1993: **sha0** is published, rescinded shortly after, and eventually replaced by **sha1**.
- 1995: **sha1** is published and remains in popular usage through most of the 2000s. In 2005, Wang et al. described theoretically feasible attacks on the collision resistance in

**sha1** [52] [51]. The hash function was not immediately revoked as a security standard, although it has been gradually phasing out since.

- 2001: **sha2** is designed by the NSA and published by NIST. There have been no significant attacks on the full algorithm, and **sha256** (a variant) is used for finding proof-of-work in Bitcoin.

- 2008-2010: NIST organizes the NIST hash function competition, where Keccak is accepted as one of 51 candidates.

- 2012: Keccak is selected as the winner of the competition.

- 2015: **sha3** (originally Keccak) is published by NIST as a hashing standard.

A layman's description of **sha1** is outlined in Table 2.13, though the pseudocode can be found in any modern cryptography book (like [49]), or relevant informational website. The $h_i$ and $K_i$ that are mentioned are pre-determined constants, commonly referred to as *nothing-up-my-sleeve* numbers. These are recorded as hex values in Tables 2.11 and 2.12. It is clear that the $h_i$ are chosen to be visually innocuous, while the $K_i$ are $2^{30}$ times $\sqrt{2}, \sqrt{3}, \sqrt{5}, \sqrt{10}$. MD5 (published in 1994) is constructed similarly, though **sha1** has more rounds, a larger internal state, and the extension from sixteen words $w_i$ to eighty is not present in MD5. In fact, **sha1** was an improvement over **sha0**, and the most significant change was an extra bit shift in this extension.

Currently, MD5 is considered to be cryptographically broken, as it is demonstrably not collision-resistant. The persistent use of **sha1** , despite the fact that it is on the edge of being similarly broken, is probably because of historical and ubiquitous use, and because there *is not* a demonstrated collision yet. Regardless, **sha2** and **sha3** are both endorsed by NIST and considered secure... for now.

$$
\begin{aligned}
h_0 &= 67452301 \\
h_1 &= EFCDAB89 \\
h_2 &= 98BADCFE \\
h_3 &= 10325476 \\
h_4 &= C3D2E1F0
\end{aligned}
$$

Table 2.11: Initial Hash Values for **sha1**

$$
K_i = \begin{cases}
5A827999 & 0 \leq i < 20 \\
6ED9EBA1 & 20 \leq i < 40 \\
8F1BBCDC & 40 \leq i < 60 \\
CA62C1D6 & 60 \leq i < 80
\end{cases}
$$

Table 2.12: Key Schedule for **sha1**

### SHA3 (Keccak)

Originally known as Keccak, it has been adopted and approved by NIST as the next iteration in the Secure Hash Algorithm family. Keccak uses a *sponge* construction as opposed to a Merkle-Damgård construction. A sponge construction starts with an initial vector of all zeroes, breaks the message into blocks, "absorbs" all the blocks into the internal state using a two-step process of XOR and an algorithm-dependent permutation function, then "squeezes" output bits using the internal state and the permutation. The major advantage of this construction over a Merkle-Damgård approach is that not only is the entire internal state *not* output at the end, the reserve bits (called the *capacity*) is required to be at least double the desired security (in bits). The reason this is helpful is that it prevents length-extension attacks.

There are a handful of variations, with capacity ranging from 256 to 1024 (128-bit security to 512-bit security, See Table 2.1).

**sha1**$(m)$ :

$y \leftarrow \texttt{pad}(m)$
partition $y$ into 512-bit blocks
for each block $M$:
    partition $M$ into sixteen 32-bit words
    extend (using XOR and leftrotate) into eighty 32-bit words
    initialize hash value for this block:
    $a \leftarrow h_0$
    $b \leftarrow h_1$
    $c \leftarrow h_2$
    $d \leftarrow h_3$
    $e \leftarrow h_4$
    for $i$ from 0 to 79:
        update $a, b, c, d, e$ using current values and key schedule $K_i$
    $h_0 \leftarrow h_0 + a$
    $h_1 \leftarrow h_1 + b$
    $h_2 \leftarrow h_2 + c$
    $h_3 \leftarrow h_3 + d$
    $h_4 \leftarrow h_4 + e$
**hash**$\leftarrow$ *combine*$(h_0, h_1, h_2, h_3, h_4)$
return **hash**

Table 2.13: **sha1** Pseudocode

### 2.8.5 MACs and HMACs

MAC stands for *message authentication code.* A MAC algorithm provides a small piece of information (a *tag*) that can be used to verify the integrity of a message (like a hash function) as well as the sender (like a signature). MACs are different from (and can be built using) hash functions in that a MAC requires a symmetric key between two parties. Also, while a hash function's collision resistance might be broken (like MD5), the security of the corresponding MAC would not be compromised because of the secret key. Although MACs could be generated using signature schemes (signers and verifiers would need both the public and private keys), MACs based on hash functions are faster by several orders of magnitude.

A *keyed-hash message authentication code* is known as an HMAC. Suppose Alice and Bob possess a shared key $K$. If Alice wants to send a message $m$ to Bob and include a MAC using, for example, **sha1** (an HMAC using **sha1** would be known as HMAC-**sha1**), a naive construction might simply have her compute:

$$\text{HMAC} = \mathbf{sha1}(K||m).$$

However, using a Merkle-Damgård based hash function leaves this approach open to *length-extension* attacks. A length-extension attack is based on the fact that Merkle-Damgård hash functions (like MD5 or **sha1**) operate iteratively and output their internal state. Suppose Alice sends Bob the tuple $(m, t)$ (where $m$ is a message, and $t$ is the corresponding tag) and Eve intercepts it. She can append data to the end of $m$, and continue feeding $t$ through **sha1**(as if $t$ was simply an intermediate internal state) along with the appended data. Thus, Eve can produce a valid tag corresponding to the larger message even though she did not know the key. This works because the original input to the hash function was $K||m$.

If we try

$$\text{HMAC} = \mathbf{sha1}(m||K),$$

then collisions in **sha1** would allow collisions in the MAC.

Therefore, the typical construction of an HMAC using a Merkle-Damgård hash function is as follows:

$$\text{HMAC } = \mathbf{sha1}((K \oplus \text{pad}_1)||\mathbf{sha1}((K \oplus \text{pad}_2)||m)).$$

Since **sha3** (Keccak) does not output its entire internal state, it is not vulnerable to length-extension attacks, and therefore building an HMAC by simply prepending the message with a key is secure. Note that Keccak withholds $c$ bits in its internal state that are never output, and therefore, to resist a generic birthday attack, $c$ should be at least twice the desired security (in bits).

This concludes the section on cryptographic hash functions. We will use the information given here pervasively throughout the paper, so we will reference or remind the reader as needed.

## 2.9 Bitcoin

At this point, our preliminary chapter changes theme dramatically. Having discussed all the necessary cryptography background, we now give way to a discussion of currency, cryptocurrencies, and Bitcoin.

Bitcoin is a *cryptocurrency* (a digital form of money grounded in cryptographic security), first discussed in Satoshi Nakamoto's famous white paper [39].

### 2.9.1 A Public Ledger

We will build up the picture of Bitcoin (which may not seem intuitive at first) starting with simple examples and working our way up to the current implementation.

Imagine Alice and Bob, members of a small and primitive village. Alice gives Bob a goat in return for a basket of eggs. They both agree that the goat is worth more, but seeing as they cannot exchange smaller denominations of goat, Bob also gives Alice a note worth the equivalent of 1 goat minus 1 basket of eggs. For the purposes of reading comprehension, let's

say the note is worth 10 dollars. Alice can now exchange the note (though it has no intrinsic value) for goods and services.

Now, to take the analogy one step further: suppose the members of the village grow tired of carrying around dozens of notes. They decide to elect one person, Charlie, to keep a record of everyone's credit exchanges. In this story, Alice gives Bob the goat in exchange for a basket of eggs, and they tell Charlie that Alice's credit has increased by 10 dollars. A few problems: what if Charlie messes up? What if he loses all the records in a fire? What if he is corrupted by power and gives some of the villagers unfair favors?

Instead, suppose the villagers keep a public bulletin board where they post transactions. Alice gives Bob a goat, Bob gives Alice the eggs, and they post a note on the board that says "Bob gives Alice 10 dollars." If Alice wants to buy a pig from Dave for 8 dollars, she merely shows Dave the note on the board, he gives her the pig, and they post another note that says "Alice gives Dave 8 dollars." If anyone is curious how much money Alice has, they check the board, do the math, and find that she has 2 dollars to her name.

In a very oversimplified sense, money can be thought of as a long list of transactions between people posted on a bulletin board. If everybody trusted everybody, then the bulletin board would be the end of the story. But there are many problems now.

### 2.9.2 Double-Spending

Suppose Alice tries to double-spend her 10 dollars. She shows Dave her 10 dollar note on the bulletin board, he gives her the pig, and she writes him a note giving him 8 dollars. But now, when Dave is distracted, she pockets the note, and pays Erin a visit. Erin is selling chickens for 5 dollars each. Alice shows Erin the 10 dollar note on the board, Erin gives her a chicken, and they post a note that says "Alice gives Erin 5 dollars." At this point, Dave checks the bulletin board and realizes that Alice never posted the note giving him the 8 dollars. From the point of view that the bulletin board is law, nobody recognizes that note as valid anymore. Dave is frustrated, but at this point, there's nothing he can do.

### 2.9.3 Proof-of-work

At this point, the analogy starts to break down, so let us simply say that one part of Bitcoin is a public bulletin board that records all transactions of money between users. The problem of double-spending remains. The solution to this: Alice broadcasts a message "Alice gives Bob 10 dollars" that goes into a candidate pool. This message is not yet ratified, though anyone can check the past transactions to make sure that Alice actually has 10 dollars to spend.

In order for the transaction to be considered valid, Alice (or any other user, really) must demonstrate some amount of computational work using the transaction as an input. For example, let $m =$ `Alice gives Bob 10 dollars`. Suppose the requirement is that Alice must find a suffix to attach to $m$ such that the **sha1** hash of the message plus the suffix outputs a digest beginning with sixteen zeroes (the hex digest would begin with four zeroes). We use the word *nonce* to refer to the suffix that Alice appends to her message.

```
python: nonce = 0
python: while(True):
python:     m = "Alice gives Bob 10 dollars" + str(nonce)
python:     h = hashlib.sha1()
python:     h.update(m.encode())
python:     digest = int( h.hexdigest(),16)
python:     if( digest < 2**144 ):
python:         print(nonce)
python:         break
python:     nonce += 1
output: 36922
```

Table 2.14: Python 3 Code: Computing a Proof-of-Work

This code in Table 2.14 ends with a nonce of 36922, which gives us

$$m = \texttt{Alice gives Bob 10 dollars36922}$$

$$\textbf{sha1}(m) = \texttt{0000c6acb2748f81d981c71dd4a6290e65e96046}$$

The hash is easy to verify, but hard to fake, because of the one-way property of hash functions. This is called a *proof-of-work*, and it shows that Alice had to spend some computational time in order to compute it. This indicates that Alice would not have time to try to double-spend her money because of the lag time involved in broadcasting a transaction and then ratifying it with a proof-of-work.

### 2.9.4 Blocks

In practice, all broadcasted transactions are sent to a candidate pool. Anyone can check the validity of the transactions by checking the public bulletin board to make sure the corresponding users have the available funds.

A user (any user) aggregates a set of transactions, called a *block*, and hashes the block together with a nonce (proof-of-work). The nonce is incremented until the hash lies below an agreed upon threshold, or *target*, and then the block, nonce, and proof-of-work hash are broadcast to the entire network. If the block and hash are valid, then the users update their list of blocks so far with the new block.

To be more specific, there are actually two parts to the block: the *block header* and the transactions. Since a block may contain as many as 1,500 transactions, it is convenient to use a data structure that is one-way dependent on all of the transactions (See Figure 2.4). The user constructs a Merkle tree with all the transactions to be included in the block, and appends the Merkle root to the block header. Also included in the block header are the time, the nonce, and the hash of the previous block (this last requirement is to guarantee sequentiality of blocks, Figure 2.5). The block header is much smaller than the rest of the block (i.e., the transactions) by comparison, so computing the hash is faster. When we say that a user hashes a block, it is actually a misnomer: they are hashing the block *header*.

As blocks percolate through the peer-to-peer network, users update their ledger, which is called the *block chain*. This process guarantees that every transaction must have been put through a proof-of-work before inclusion into the block chain, and therefore drastically

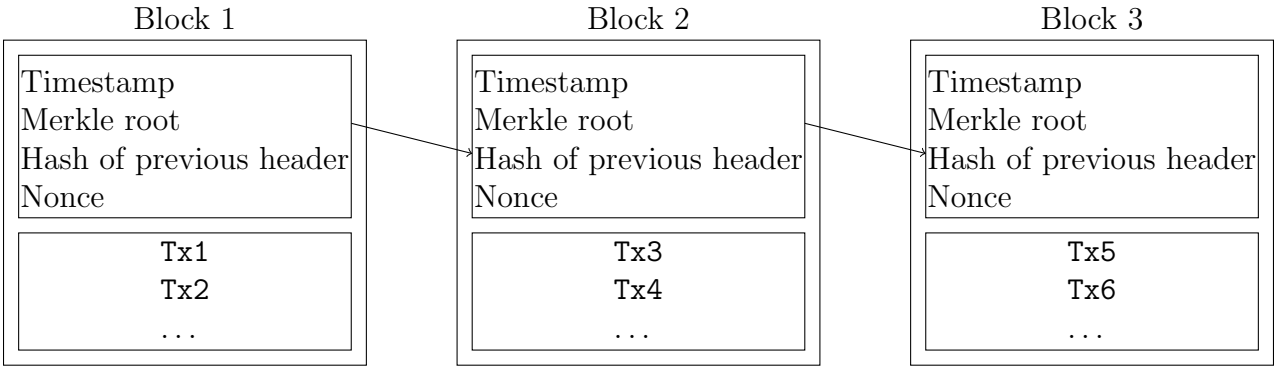


Figure 2.5: The Block Chain

reduces the risk of double spending. In practice, especially for large exchanges of bitcoins, users will wait until a transaction is six blocks deep into the block chain, as the probability of an adversary subverting the block chain is statistically negligible after that point (more in Section 2.9.7).

**Remark 2** (Proof of Existence)**.** *Clearly, the block chain can be used as a public, trusted, and timestamped bulletin board. Hence, it is theoretically possible to lay claim to digital information by recording a hash of the information in a (valid) transaction and embedding it in the block chain. This has already been into practice as a service at* `https://www.proofofexistence.com/` *(Check URL).*

### 2.9.5 Miners

The function of a Bitcoin *miner* is to gather verified transactions into a block and provide a proof-of-work for the block. When a proof-of-work is produced, the block is submitted to the network, and everyone updates their ledgers (the block chain) to include the new block.

Once found, the proof-of-work hash is trusted by the network, because of the reliability of the cryptographic hash functions involved (in this case, SHA256). Nobody needs to trust individual parties, but they can trust that SHA256 has all the collision resistance properties, and therefore forging a hash is computationally infeasible. This is called *distributed consensus*, and the largest and most attractive advantage over the traditional banking system is

that it removes centralization of monetary transactions.

Occasionally, because of the lag time involved in the peer-to-peer network, two miners will mine a block around the same time. When that happens, miners must make a choice regarding which block to extend. This is called a *fork*. Eventually, one tine of the fork will be longer, and by default, miners mine on the chain with the most proof-of-work. Note that this does not necessarily mean the longest chain.

Miners have an incentive to mine blocks, as there is a Bitcoin (BTC) reward for finding the next valid block in the block chain. When Bitcoin was born, the reward was 50 BTC per block, but the reward halves every 210,000 blocks. Currently, the reward is 25 BTC. Miners also receive transaction fees (more in Section 5.2.3).

### 2.9.6 Verifications

There are three main types of users on the Bitcoin network.

*Full Nodes.*

A user with a full node holds the entire block chain in computer memory. Currently, the total size of the block chain is nearing 60 gigabytes. When a candidate transaction `Tx1` is broadcast, a full node can verify that the inputs to this transaction are correctly signed and come from unspent transaction outputs (UTXO) recorded in the block chain. Along with this verification, for each transaction that `Tx1` references, at least one full node must announce a Merkle path verifying that that transaction is included in a block.

It is estimated that there are only around 5000 full nodes in the entire world. It is impossible to deduce the number of users (full nodes or not) on the network because one person may have many addresses, but guesses range from 500,000 to 4 million.

*Simplified Payment Verification (SPV) Clients.*

SPV clients hold only the block headers, and cannot fully verify transactions on their own. Regardless, after a full node computes a Merkle path and announces it, SPV clients can check that the Merkle path for the transaction matches the Merkle root in the stored

block header.

*Miners.*

Miners need not hold the entire block chain, nor even verify transactions at all. Since transactions are verified by full nodes, miners can merely collect verified transactions and compute hashes.

### 2.9.7 51% Attack

If a user (or group of users working together) control more than 50% of the computational mining power on the network, then they can enact a *majority attack* on the block chain.

**Example 2.9.0.1.** *Matt the miner controls more than 50% of the Bitcoin mining power. He buys a car from Alice for 50 BTC. The transaction is announced over the network, and included into the block chain (it is a valid transaction, since Matt has at least 50 BTC in UTXO). Let us say that the transaction is in block 100. When Alice is satisfied that the transaction is verified and safely recorded in the block chain, she gives the car to Matt.*

*Shortly after (say, around block 110), Matt throws all his computing power into creating a new block to succeed block 99 (a fork). This block does not contain Matt's transaction to Alice. Even when Matt broadcasts this block, the other miners will ignore it because they are working on extending the chain with the most proof-of-work. However, since Matt has more hashing power than all the rest of the other miners combined, he will eventually catch up and his version of the block chain will contain more proof-of-work. By default, the miners will then switch to his block chain, and the "honest" chain will be considered orphaned.*

*At this point, the public ledger now contains no record of Matt giving Alice 50 BTC. Matt has successfully stolen Alice's car.*

It seems gravely unlikely that anyone will be able to enact a 51% attack on the block chain. Regardless, there are still vulnerabilities that lie in the danger of entities controlling just a fairly large percentage of the hashing power. Satoshi Nakamoto foresaw the possibility even in his original Bitcoin paper [39].

We will not reproduce all the calculations, but simply state the result.

**Theorem 2.9.0.1** (Attacker Success Probability)**.** *Suppose Matt is a dishonest miner. Given that:*

- *The probability of Matt mining the next block is $q$*
- *Matt is catching up from behind after $z$ blocks,*

*then Matt's potential progress will be a Poisson distribution with expected value*

$$\lambda = z\frac{q}{1-q}.$$

*The probability that Matt could catch up is:*

$$P = 1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!} \left(1 - \left(\frac{q}{1-q}\right)^{z-k}\right).$$

For example, if Matt has 10% of the mining power, and is 5 blocks behind, then his success probability is $< .1\%$. This is the reason that it is considered good practice (especially for the sale of expensive items) to make sure a transaction is 6 blocks deep in the block chain before transferring the item.

So, at this point, we can loosely define three hierarchies of ratification in the lifetime of a transaction `Tx`:

- `Tx` is broadcast to the pool, and full nodes check to make sure it validly references UTXO from the block chain.
- A miner includes `Tx` in a block, computes a valid proof-of-work, and submits it to the network.
- The block that includes `Tx` is 6 blocks deep in the block chain

There has never been a successful 51% attack on the network. For research into attacks using less than half of the computational power, we refer to the following papers:

- In 2012, Karame et al. focuses on double-spending attacks with low cost, and analyzes the countermeasures adopted by Bitcoin developers [28].

- In 2013, Bahack argues that the computational threshold for launching a successful forking attack is not as high as 50%. Rather, it is closer to 25%, exploiting attacks that he calls "Block Discarding" and "Difficulty Raising" [4]. The main ideas are that an attacker with many network connections (much more than the average user) can control to some extent which blocks propagate through the network, then withhold valid blocks while mining successive blocks. Difficulty-raising refers to the fact that if blocks are mined quickly, then the target difficulty increases automatically. Therefore, an attacker could theoretically mine several blocks in a row, timestamping them accordingly so that the target difficulty is always raised to a point where they still have a good chance of mining the next block first.

- In 2015, Heilman et al. investigates the power an attacker would have by simply "eclipsing" a victim node by occupying all of its input and output connections [21].

### 2.9.8 The Profitability of Mining

It seems appropriate to end this chapter with a light discussion regarding the profitability of being a Bitcoin miner.

There are application-specific integrated circuits (ASICs) specifically built to mine bitcoins. The current top model is the Antminer S7, which computes 4.73 terra-hashes per second ($4.73 \cdot 10^{12}$) , and runs at 1210 watts (1.21 kilowatts). The Antminer S7 costs 987 USD on Amazon.

The mining power on the entire network totals in excess of 1.24 exa-hashes per second ($1.24 \cdot 10^{18}$). Since Bitcoin blocks are being mined on average once every 10 minutes, there are around 4320 blocks mined per month. The current Bitcoin reward is 25 BTC per block, so there are 108,000 newly minted bitcoins every month. At the current exchange (1 BTC =

408.27 USD), that gives us 44 million USD every month. A single Antminer S7 represents

$$\frac{4.73 \cdot 10^{12}}{1.24 \cdot 10^{18}} = 3.8 \cdot 10^{-6}$$

$$= 0.00038\%$$

of the mining power. Assuming rewards are mined exactly equal to this percentage, we estimate that one Antminer S7 running at full capacity brings in a revenue of

167.84 USD

per month. If electricity (in Colorado, for example) is 11.46 cents per kilowatt-hour, then running a 1.21-kilowatt machine 24 hours a day, 30 days a month gives us 871.2 kilowatt-hours, and therefore an electricity cost of

99.84 USD

per month.

This gives us a profit of around 68 USD per month. In other words, the Bitcoin mining machine turns electricity into money at the profitable rate of 9.4 cents an hour. (Of course, owning a mining rig consisting of 1500 Antminer S7s running at full capacity with cheaper electricity costs—that is, in China—is a different story).

Because colloquial discussion inevitably leads to the question "Should I mine for bitcoins?" the answer is that any hashrate below 3 trillion hashes per second is not worth the electricity spent in hashing (the threshold is 2 trillion hashes per second for China).

This concludes the chapter of preliminary information. Chapters 3, 4, and 5 of this dissertation describe in detail our results and contribution to the fields of cryptography and cryptocurrency.

# Chapter 3

# Trapdoors

## 3.1 Overview

The potential for trapdoors in security algorithms has always been an issue.

In a recent case, the pseudorandom number generator Dual_EC_DRBG was revealed to have a trapdoor: the algorithm employed the use of two (supposedly) randomly chosen points $P$ and $Q$ on an elliptic curve. This itself is a red flag: there is simply no guarantee that $P$ and $Q$ fit the bill of nothing-up-my-sleeve numbers. There were other complaints raised about the algorithm (regarding the output bit truncation) but nevertheless, it was standardized in the early 2000s.

In 2007, an informal presentation by Shumow and Ferguson called attention directly to the fact that if one elliptic point was chosen to be a multiple of the other, then an adversary could predict future outputs of DUAL_ECDRBG [46].

Finally, after Edward Snowden's leak of NSA documents, it was revealed that a program existed in the NSA (Bullrun) designed "to covertly introduce weaknesses into the encryption standards followed by hardware and software developers around the world," [41] and furthermore that an intentional relationship had been built into the elliptic points (therefore introducing a security flaw in Dual_EC_DRBG).

This flaw does not allow an attacker to reverse the Dual_EC_DRBG algorithm. Rather, knowing the current internal state of the PRNG and the relationship between $P$ and $Q$ would allow an attacker to predict future output bits. This is not the same as a trapdoor in a hash function. Fortunately, there has not yet been a known case of a widely-used hash function containing such a security fault.

Most widely used cryptographic hash functions are constructed with speed in mind: the basic components are bitwise operations (and, or, and xor) and bit shifts. **sha1** and MD5 are good examples of this; however, the only measure of security **sha1** has is that no one has broken it yet. Actually, MD5 has been broken (a collision has been produced) [52] and there is a theoretical feasible ($2^{63}$) attack on **sha1** [51].

There is another class of hash functions which are provably secure: the task of finding a collision is provably reducible to solving a problem which is mathematically unsolvable in polynomial time. These are called *algebraic hash functions*. The trade-off is that these hash functions are orders of magnitude slower.

In both cases, there are ways to build in trapdoors. For example, most cryptographic hash functions employ nothing-up-my-sleeve numbers (similar to the seemingly unbiasedly chosen elliptic points in Dual_EC_DRBG). These numbers are often derived from accepted mathematical constants like $\pi$ or $\sqrt{2}$, so as to allay suspicions of a trapdoor. **sha1** uses $2^{30}$ times the square roots of 2, 3, 5, and 10 as its initialization vectors. It is certainly not the intent of this chapter to suggest **sha1** contains a trapdoor; nevertheless, it is conceivable that a trapdoor *could* be built into a hash function using these constants as a disguise.

Section by section: we will provide a background of hash functions, formalize the idea of trapdoor hash functions, discuss previous work, build up a progression of functions toward one that fits all of the desired properties. Finally, we will discuss ongoing and future work.

## 3.2 Using a Trapdoor Hash Function

Since the most important characteristic of a hash function is its one-way property, a trapdoor hash function would compromise all cryptographically secure information or exchanges which depend on this property.

Not all applications of trapdoor hash functions are malicious (although those are the easiest to concoct). We give some examples of applications, both malicious or otherwise.

- *Signature schemes.* As we will discuss later, *chameleon hash functions* have known trapdoors. This is useful in building a signature scheme: the recipient is the only one with access to the trapdoor and therefore cannot prove the validity of the signature to anyone else.

- *Public key exchange.* Suppose Alice and Bob want to exchange information privately. To set up a shared key, Alice sends Bob a hash $H$, which she knows Bob can open using the trapdoor to obtain pre-image $M$. Since only Alice and Bob know the trapdoor, they are the only ones who can feasibly construct this pre-image. They can now use $M$ as a shared key to encrypt data using AES, or other encryption standard.

- *Built-in weaknesses.* Alice is hired to edit and ratify the newest standardized hash function. Alice builds into it changes which seem benign but allow Alice to break all three resistances. After the hash function is released, Alice can forge signed documents, send viruses by tricking hash verifiers, and potentially break into password-protected information. Furthermore, Alice can simply wait until the hash function is in wide usage and then anonymously release instances of collisions, thereby triggering national panic and, as Aumasson calls it, cryptoarmageddon. [3]

- *Bitcoin.* Since the whole backbone of Bitcoin is dependent on proof-of-work, if Alice can quickly manufacture pre-images, she can single-handedly subvert the entire block chain. If she is clever enough, she can probably pull this off undetected, as long as

she works anonymously and always under different addresses. With this amount of control, she can:

- Reap all of the mining rewards
- Double spend bitcoin
- Enact DOS attacks against specific businesses or companies, as she essentially controls which transactions are included in the block chain.

Not to mention the fact that she can spend the unspent transaction outputs on almost every transaction by forging bitcoin addresses and signatures.

## 3.3 Motivation and Previous Work

Before we begin, it seems relevant to discuss the goal and scope of this chapter. Currently, there does not yet exist a detailed treatment and construction of a trapdoor hash function. One goal is to decide what properties we want in a trapdoor, and we will paint a clearer image of what we want by building up more and more focused examples. In terms of incremental progress, this chapter should serve as a stepping stone toward a more foolproof ARX-based trapdoor hash function, which, regrettably, we have not created. Such a hash function would have profound implications in the arena of public key cryptography. For example, a trapdoor hash function using only addition, rotation, and XOR operations could be used to enact a public key exchange.

A step in a different direction would be to modify an existing hash function (such as **sha1**) that would allow for the existence of a single collision. Even though this is a far cry from a trapdoor hash function, it would still have malicious applications. The creator of the modified hash function could publish it and then later, when it was widely used, pseudonymously publish the single collision while claiming to be able to produce more. Whether the intent is to incite fear, or to blemish the reputation of the government, it is

important to be able to understand trapdoors as thoroughly as possible.

*Previous work.*

- In early 2016, Microsoft Vulnerability Research warned that the OpenSSL address implementation in Socat contains a hard-coded Diffie-Hellman 1024-bit prime number that actually was not prime at all. The security advisory itself states that "...since there is no indication of how these parameters were chosen, the existence of a trapdoor that makes possible for an eavesdropper to recover the shared secret from a key exchange that uses them cannot be ruled out." [54]

- For purely academic motivations, or perhaps just for existential demonstration, a malicious variant of **sha1** was detailed and published in 2014 by Albertini, Aumasson, Eichlseder, Mendel, and Schläffer [1]. This variant could only produce a single collision, however.

- In 2012, Aumasson published a preliminary paper on trapdoor hash functions [3] which detailed a malicious version of the SHA3 candidate BLAKE. This construction, while simple and not easily detectable, must allow for the collision inputs to be known before the finalization function is derived and built into the hash function.

- In 2007, Shumow and Ferguson openly discussed the potential for a backdoor in the pseudo-random number generator DUAL_EC_DRBG [46].

- If an adversary Eve can break an existing hash function (generate collisions, pre-images, and second pre-images), and no one else knows either how to do so or that Eve can do it at all, then this is equivalent to having a trapdoor hash function. In 2005, Wang, Yin, and Yu published a groundbreaking paper which detailed how to find collisions in **sha1** [51]. We also refer the reader to Cochran's paper examining the finer details of this construction, which is a little easier to digest [15].

- Earlier that year, Wang and Yu had already published a paper breaking MD5 and other hash functions [52] (these techniques were applied in the SHA1 paper).

## 3.4 Goal

We give properties of an ideal trapdoor hash function, then gradually build up more and more useful examples to illustrate our goals.

As a final proof of concept, we construct a trapdoor hash function using elliptic curves based on a trapdoor system by Edlyn Teske [50]. Our function can be used to produce arbitrary collisions, pre-images, and second pre-images. Furthermore, we claim that the algorithm leaks no information about the trapdoor.

The trapdoor uses elliptic curves, and is dependent on the so-called Weil descent attack on the elliptic curve discrete logarithm problem. We give a brief background on this and Teske's paper.

Given the limited body of research on trapdoor hash functions, we hope that this chapter would serve two purposes:

- To provide a concrete link between the study of elliptic/hyperelliptic curves and the ever-growing work on hash functions.
- To serve as a stepping stone to more and better examples of trapdoors in hash functions.

## 3.5 Cryptographic Hash Functions

For the sake of readability, we will state again the resistance properties that are desired for a cryptographic hash function (Table 3.1).

Furthermore, it is specifically important that a hash function have an output length of at least 160 (See Table 2.1 and Equation 2.8.2).

- Collision resistance: It should be computationally infeasible to find distinct $x, x'$ with $\mathbf{hash}(x) = \mathbf{hash}(x')$.
- Pre-image Resistance: For a given digest $y$, it should be computationally infeasible to find an $x$ with $\mathbf{hash}(x) = y$. We also refer to this as the "one-way property."
- Second Pre-image Resistance: For a given $x$, it should be computationally infeasible to find $x' \neq x$ with $\mathbf{hash}(x) = \mathbf{hash}(x')$.

Table 3.1: Cryptographic Hash Function Properties, Revisited

## 3.6 Trapdoor Hash Functions

In 1997, H. Krawcyzk and T. Rabin [31] presented *chameleon hash functions*, which have trapdoors. The existence of the trapdoor, however, is known and agreed upon beforehand. As presented, the main utility of chameleon hash functions is in signature schemes. Suppose Alice signs a message $m$ by first hashing it with, say, $H$ (a chameleon hash) and then using her private RSA exponent on the hash. If Bob, the recipient of $m$ and the signature, is the only one with access to the trapdoor for $H$, then he cannot prove to any third party that the signature corresponds to a given message because he can reverse-engineer the hash of the message from Alice any way he wants. Alice, as in a normal signature scheme, cannot repudiate her own signature because she does not have access to the trapdoor. We refer to [31] and [37] for more details, but would like to summarize by saying that a chameleon hash function is a bit too transparent for what we want.

We are looking for something more subtle. The hash function we seek should seem innocuous to someone who is not aware of the existence of the trapdoor. We offer the following as a definition:

**Definition 3.6.1.** *A **trapdoor hash function** with secret key $k$, has the properties listed in Table 3.2. The notion of "undectability" was introduced by Aumasson in [3] and revisited in [1]. He also introduced the term "undiscoverability" to refer to the fact that an adversary cannot discover the key even by comparing select messages and their corresponding digests.*

*To rephrase this, the owner of the trapdoor key should be able to use the key to produce a pre-image, but an adversary should not be able to discover the key (or its existence) by comparing the pre-image to other messages (chosen by the adversary or not).*

- HASH FUNCTION: It satisfies Definition 2.8.1 without knowledge of $k$.
- TRAPDOOR: With knowledge of $k$, all three hash function resistances can be broken.
- UNDETECTABILITY: The function algorithm does not reveal either the key or even the existence of a trapdoor.

Table 3.2: Trapdoor Hash Function Properties

We will first see an example that nicely illustrates the TRAPDOOR property. It has been constructed to look contrived: we are merely trying to demonstrate the simplicity of implementing a trapdoor, but the difficulty of making it undetectable.

The reason for this is that the last property, UNDETECTABILITY, is the hardest to quantify, and we will deal with it on a case-by-case basis without giving its definition too much rigor.

After all, since humans are the ones that recognize trapdoors (and not computers), any number of steps (or lack of steps) in a hash function could raise suspicion. Indeed, creating an algorithm that recognizes trapdoors could be a new area of interest altogether.

Regardless, we will hopefully make the point of UNDETECTABILITY clear with the following examples.

### 3.6.1 Simple Example

Here we will give a simple example of a hash function with a trapdoor. This example is for illustrative purposes only, and should not be considered as a feasible hash function.

Let $k$ be a fixed secret 160 bit string. Suppose $m$ is a message with length at least 320. Let $x$ be the first 160 bits, $y$ the next 160 bits, and $z$ the remaining bits. The function $H$ is defined as follows: if $x = k$, then $H(m) = y$, and otherwise $H(m) = \mathbf{sha1}(m)$.

Clearly, we can produce messages (as many as we want) that map to the same digest, or in fact any digest we want. If $d$ is the desired digest, then simply set $m = k||d$. Unfortunately, this function is trivially transparent, as it reveals $k$ in its own algorithm.

Perhaps a very slight improvement would be the following $H'$: if $\mathbf{sha1}(x) = k$, then $H'(m) = y$, and otherwise $H'(m) = \mathbf{sha1}(m)$. In this variation, it is still very clear that the function contains a trapdoor, but at least the algorithm does not give away the secret key (in this case, the key would be a pre-image of $k$ rather than $k$ itself).

This idea is actually the stepping stone to a slightly better variation involving both **sha1** and RSA.

### 3.6.2 Trapdoor Hash Function Based on RSA

In this section, we will describe a nontrivial hash function with trapdoor that allows a party with the secret information to generate collisions and find first and second pre-images.

Let $m$ be a given message, and define $f(x) = x^e \pmod N$ (and furthermore, we assert that the output of $f$ must be expressed in the same number of bits as $x$). Here, we take $N$ to be an RSA modulus (of the form $N = pq$), and $e$ to be the public key. Assume further that $N < 2^{160}$. Suppose there is also a private key $d$, such that $ed = 1 \pmod{\phi(n)}$, where $\phi$ is Euler's totient function.

Then express $m$ as $m = m_1||m_2$, where $|m_2| = 160$. (If $|m| < 160$, then $H = SHA1$). Define our trapdoor hash function $H$ as

$$H(m) = \mathbf{sha1}(m_1) \oplus f(m_2) \tag{3.1}$$

**Theorem 3.6.1.1.** *The function $H(m)$ defined in* (3.1) *has a trapdoor (although it is not undetectable).*

*Proof.* Pre-images. Given a digest $y$, we can produce a message $m$ such that $H(m) = y$. Let $m_1$ be any string. Let $m_2 = (\mathbf{sha1}(m_1) \oplus y)^d \pmod N$. Define $m = m_1||m_2$. From the

definition of $H$, we have $H(m) = \mathbf{sha1}(m_1) \oplus (\mathbf{sha1}(m_1) \oplus y)^{ed} \pmod{N} = y$.

Second pre-images. Observe that since we are not restricted to a choice of $m_1$, we can generate as many pre-images as we want.

Collisions. Certainly if we can find second pre-images, then we can also find collisions.

Furthermore, usage of the key does not give away the key (the key being the private exponent or the factorization of $N$). And if we consider this function as a black box, then usage of the key does not reveal the existence of a trapdoor, since we have such broad control over $m_1$, the pre-images we generate give away no information about the key.

□

The problem is that this function is highly suspicious, and with a little thought, an observer can guess that a trapdoor is being used. This function is nice, however, in the sense that the fabricated collisions reveal no information about the trapdoor. Since we have complete control over $m_1$, we can make pre-images that are, for example, arbitrarily long, or contain any string we want.

### 3.6.3 Trapdoor Hash Function Based on VSH

The last two examples, while not being particularly useful in practice, at least provide us with some direction regarding what we want in an ideal trapdoor hash function. Ideally, we want something that obfuscates the trapdoor throughout the body of the function, and not just tacked on at the end. Furthermore, using something less conspicuous than RSA would be nice, because the relationship between RSA private and public keys is common knowledge.

VSH stands for Very Smooth Hash, an algebraic hash function created by S. Contini, A. Lenstra, and R. Steinfeld [16]. The security of VSH is based on the hardness of finding modular square roots of very smooth numbers (numbers whose prime factors do not exceed a specified threshold). This problem is known as VSSR (Very Smooth Nontrivial Modular Square Root).

**VSH Algorithm.** Let $p_i$ be the $i$th prime. Let $k$, the block length, be the largest integer such that $\prod_{i=1}^{k} p_i < n$. Let $m$ be an $\ell$-bit message to be hashed, consisting of bits $m_1, \ldots, m_\ell$, and assume $\ell < 2^k$. To compute the hash of $m$ perform steps 1 through 5:

1. Let $x_0 = 1$.
2. Let $L = \lceil \frac{\ell}{k} \rceil$ (the number of blocks). Let $m_i = 0$ for $\ell < i \leq Lk$ (padding).
3. Let $\ell = \sum_{i=1}^{k} \ell_i 2^{i-1}$ with $\ell_i \in \{0, 1\}$ be the binary representation of the message length $\ell$ and define $m_{Lk+i} = \ell_i$ for $1 \leq i \leq k$.
4. For $j = 0, 1, \ldots, L$ in succession compute
$$x_{j+1} = x_j^2 \times \prod_{i=1}^{k} p_i^{m_{jk+i}} \pmod{n}.$$
5. Return $x_{L+1}$.

Table 3.3: Unified Pseudocode for VSH

For clarity's sake, if we set $e_i = \sum_{j=0}^{L} m_{jk+i} 2^{L-i}$ for $1 \leq i \leq k$, we can see that the value calculated by the VSH algorithm is equal to $\prod_{i=1}^{k} p_i^{e_i}$.

VSH uses an RSA modulus $n$ and requires modular exponentiation (which makes it much slower than SHA1) and it actually contains a trapdoor. The trapdoor (as outlined in [16]) is based on knowing $\phi(n)$, but using this information to produce collisions reveals the existence of the trapdoor, as well as (a multiple of) $\phi(n)$, which would compromise the whole hash function. Furthermore, the trapdoor does not allow the user to generate pre-images, merely collisions and second pre-images.

## 3.7 Contribution: Elliptic Curve Trapdoor Variant

The VSH authors also published variants: one used the hardness of the discrete log problem (DLP) instead of VSSR [32]. One used elliptic curves as the underlying group, and one used cyclotomic polynomials.

We would like to expand on the idea of using elliptic curves to create a trapdoor hash

function.

Although we concede that the DLP is intractable over an elliptic curve group of sufficient size, in 2006, Edlyn Teske published a paper [50] detailing a method for converting a DLP over one elliptic curve to a DLP over the Jacobian of a hyperelliptic curve where it may be easier to solve. She worked exclusively in the case of an elliptic curve over the finite field of order $2^{161}$, so in practice any cryptographic algorithm that used $\mathbb{F}_{2^{161}}$ would be highly suspect, but it seems reasonable to assume that there may and will be other cases that accomplish the same task. Furthermore, as study on hyperelliptic curves continues to move forward, we may eventually find that underlying structures of the Jacobian could impact cryptographic security.

To summarize Teske's construction:

- Alice creates a secret elliptic curve $E_s$ over $\mathbb{F}_{2^{161}}$.
- Alice then constructs a secret (and sufficiently long) chain of isogenies staring with $E_s$ and ending with $E_p$ (the public curve). The parameters are chosen so that the most efficient way of solving the DLP over $E_p$ is Pollard's rho method.
- The Gaudry-Hess-Smart (GHS) Weil descent attack (which we explain in more detail later) reduces the elliptic curve DLP over $E_s$ to a hyperelliptic curve DLP in the Jacobian of a curve of genus 7 or 8, where the index calculus method will be much more efficient.
- The upshot is that Alice can release a public elliptic curve $E_p$, over which the DLP is intractable, but then solve the DLP over an isogenous elliptic curve $E_s$ using the GHS attack.

Furthermore, Teske details the importance of constructing isogenies that leak no information about the secret elliptic curve given the public one.

Knowing this, it is a fairly significant step (over the previous attempts in this chapter) to adopt Teske's construction and incorporate it into a VSH variant to create a trapdoor hash function.

**Elliptic Curve Trapdoor Algorithm**. Set up secret and public elliptic curves over $\mathbb{F}_{2^{161}}$, respectively $E_s$ and $E_p$, a la Teske. The isogeny chain and $E_s$ are the secret key $k$. Randomly select $n$ elliptic points $P_1, \ldots, P_n$ in $E_p$. These should be chosen as nothing-up-my-sleeve constants. Since there is an isogeny from $E_s$ to $E_p$, points in $E_p$ correspond (possibly not in a one-to-one relationship) with points in $E_s$.

1. Let $m$ be the message that we want to hash. Pad $m$ with 0s until the message has length a multiple of $n$.
2. Break $m$ into $n$ blocks: $m_1, \ldots, m_n$.
3. Set $Q = \sum_{i=1}^{n} m_i P_i$.
4. Output $Q$, or rather some bit representation thereof.

Table 3.4: Pseudocode for ECTA

**Theorem 3.7.0.1.** *The algorithm detailed in Table 3.4 is a trapdoor hash function.*

*Proof.* Suppose we know $k$. Can we break all three resistances?

PRE-IMAGES. Suppose we are given an output $Q$. Generate any elliptic points $Q_1, \ldots, Q_{n-1}$ in $E_p$. Set $Q_n = Q - \sum_{i=1}^{n-1} Q_i$. For $1 \leq i \leq n$, use the GHS attack on $E_s$ to solve the DLP corresponding to:

$$m_i P_i = Q_i.$$

Note: this is a nontrivial step. While it is easier to solve the DLP in $E_s$ than in $E_p$, this step still requires $n$ instances of the GHS attack, and this may be difficult if $n$ is large. Teske mentions that it would still take an estimated 25,000 days on a 1GHz PIII workstation. The point is, this is not a computational step that will only take a few minutes.

Regardless, we have produced $m_1, \ldots, m_n$ and therefore the pre-image is $m = m_1||m_2||\cdots||m_n$.

SECOND PRE-IMAGES. We can find second pre-images with the same algorithm as finding pre-images. The difference is, we have to make sure that the second pre-image is different

from the given message. Hence, simply repeat the algorithm with a different choice of elliptic points, until a distinct pre-image is obtained.

Collisions. Certainly if we can find second pre-images, then we can also find collisions. Furthermore, this hash algorithm is nearly identical to that of VSH, except that it uses points on an elliptic curve rather than primes modulo $N$. The point is, without knowledge of $k$, a user can treat this function like any other cryptographic hash function. Without the key, the most efficient way of producing collisions would be Pollard's rho method. Quite unlike the RSA example we gave earlier on, this function does not give away the existence of a trapdoor in the algorithm.

Finally, even if Alice has witnessed Eve use the trapdoor to produce a pre-image, there is no known way for Alice to discern any information about the trapdoor. □

## 3.8 GHS Weil Descent Attack

In their paper [19], Gaudry, Hess, and Smart used the well-known technique of Weil descent to transform an elliptic curve DLP into a DLP over the Jacobian of a hyperelliptic curve.

A brief summary of the attack:

- An elliptic curve $E$ over a large finite field is given.
- $E$ is then reduced to a variety over a smaller field.
- The variety is intersected with enough hyperplanes to form a curve in such a way to keep the degree low and therefore the genus of the resulting hyperelliptic curve as welly.
- A hyperelliptic curve $C$ is derived, and an explicit homomorphism from $E$ to the Jacobian of $C$ is constructed.
- Now, solving an instance of the DLP in $\mathrm{Jac}(C)$ gives a solution to the corresponding DLP in $E$.

To begin with, the general form of the elliptic curves they study is:

$$Y^2 + XY = X^3 + \alpha X^2 + \beta \tag{3.2}$$

The variables $X, Y$ and the constants $\alpha$ and $\beta$ are from the field $\mathbb{F}_{q^n}$, where $q$ is a power of 2.

If we view $\mathbb{F}_{q^n}$ as an $n$-dimensional vector space over $\mathbb{F}_q$ with basis $\{\psi_0, \psi_1, \ldots, \psi_{n-1}\}$, then we can write

$$\begin{aligned} Y &= y_0\psi_0 + \cdots + y_{n-1}\psi_{n-1} \\ X &= x_0\psi_0 + \cdots + x_{n-1}\psi_{n-1} \\ \alpha &= a_0\psi_0 + \cdots + a_{n-1}\psi_{n-1} \\ \beta &= b_0\psi_0 + \cdots + b_{n-1}\psi_{n-1}, \end{aligned}$$

where $x_i, y_i, a_i, b_i \in \mathbb{F}_q$.

Now, we replace $X, Y, \alpha, \beta$ in 3.2 with their corresponding expansions over $\mathbb{F}_q$, equate coefficients of the $\psi_i$, and we obtain an abelian variety $A$ over $\mathbb{F}_q$, where the group law is inherited from the elliptic curve group law. The variety $A$ is called the Weil restriction, and this process is called Weil Descent.

The authors of [19] intersect the variety $A$ with enough hyperplanes (specifically, $x_0 = \cdots = x_{n-1}$) to reduce the degree of the curve. With $n$ equations and $n+1$ variables $(y_0, \ldots, y_{n-1}, x)$, it is possible to eliminate variables and produce a curve in just two variables. They then extract from this a hyperelliptic curve $C$ and explicitly construct a homomorphism from the original elliptic curve $E$ to the Jacobian of $C$.

They provide numerous examples of these derived hyperelliptic curves and even run an example computation using an elliptic curve over $\mathbb{F}_{2^{84}}$ and a corresponding hyperelliptic curve over $\mathbb{F}_{2^{21}}$. They ran an algorithm to solve an instance of a DLP over the Jacobian of the

hyperelliptic curve. Though the computation took about two weeks of calendar time, they estimate that the timing on a single Pentium II 450 MHz machine would have been about 31 weeks. For comparison, an equivalent calculation on the 84-bit curve, using Pollard's rho method, would have taken about 44 weeks on the same machine.

It has been shown [35] that the Weil descent attack is not useful for all curves over $\mathbb{F}_{2^n}$ (where $n$ is prime between 160 and 600). Certainly, when $n < 160$, the case is not of cryptographic interest. As it is, Teske's trapdoor paper studies only curves over $\mathbb{F}_{2^{161}}$ (note that 161 is not prime). In Teske's case, the Weil descent attack is useful and significantly reduces the computational complexity of the ECDLP. Standard NIST curves use finite fields of prime degree over $\mathbb{F}_2$ (*binary fields*). We refer to [40] for more information regarding NIST-approved elliptic curves.

Even though Weil descent has not proven to be useful in practical cases, its impact cannot go unnoted. The study of elliptic curves (and especially hyperelliptic curves) is still active, and further research may reveal more ways to both strengthen and compromise cryptographic security based on these curves. As an aside, it is still an open problem whether Weil Descent can be applied over non-binary fields (where the base field is not $\mathbb{F}_2$). For more details, we refer the reader to the Weil descent attack paper [19], and followup papers [35] and [26].

## 3.9 Summary

As we have mentioned, our ideal construction would be an ARX-based trapdoor hash function, i.e., one that used only addition, rotation, and XOR operations like **sha1**. So far, we have only been able to give examples that rely on problems that are computationally infeasible without some secret information (like RSA). We suggest that it may also be possible to make a trapdoor hash function that does *not* rely on such computational restraints. Hopefully, this would mean the function could be much faster than algebraic hash functions (like VSH).

Our goal in the future is to continue developing methods for building trapdoor hash functions. Our current construction seems to be only on the edge of feasibility, and furthermore, it is only viable over the right fields. However, we hope that this serves as a useful step to building more robust trapdoor cryptosystems. It seems likely that intractable problems like the DLP will remain useful in building these.

Another goal is to further address the question of innocuity. What does it mean for a hash function to be above human suspicion, as far as trapdoors are concerned? Could a program be written to detect trapdoors?

Building a trapdoor into an ARX-based hash function is out of the scope of this thesis, and the question remains worthy of research.

# Chapter 4

# Time-lock

## 4.1 Overview

The essence of time-lock encryption: Alice would like to send an encrypted message to Bob with the assurance that he will not be able to decrypt it until after a given date.

Practically, enacting this scheme is very difficult. The trouble can be boiled down to two main problems, which can be thought of as the theme of this chapter:

- If Alice bases the time-lock on CPU time, then she cannot control Bob's computational power
- If she bases the time-lock on realtime, then she cannot predict the future

The paradoxical nature of this problem has led to many papers and solutions, each of which tries to mitigate one of the above two problems. We start with the assumption that a central authority emits secure values from which encryption keys can be derived, and the problem we answer is this: how can that central authority endow intermediate authorities with the same power, while commutatively and hierarchically linking corresponding values?

It may seem that this problem is unrelated to the central theme of this dissertation (namely, cryptographic hash functions), but our contribution is a chaining construction making use of both hash chains and elliptic curve addition.

## 4.2 Definitions and Background

We have talked extensively about hash functions, but we have not introduced the concept of *hash chains*. We will talk more about hash chains in the next section, and in particular we will see how we can apply it to the problem of **time-lock encryption.** Encrypted data that cannot be decrypted before a given date is considered "time-locked," and we will discuss previous work that has tried to satisfactorily answer this problem.

Finally, we will discuss in some detail the mathematics behind Bitcoin wallets, and how a structure designed specifically for wallets could theoretically carry over to a solution for a variant of time-lock encryption.

### 4.2.1 Hash Chains

What is a hash chain?

**Definition 4.2.1.** *Let $h$ be a hash function. A* ***hash chain*** *is a sequence $\{s_0, s_1, \ldots, s_N\}$ with the property that*

$$h(s_i) = s_{i+1},$$

*for all $i$ in $[0, \ldots, N-1]$.*

Hash chains of course have the same one-way properties as the hash function itself. But they also are nice in that only a single seed is needed to uniquely compute a whole chain.

Hash chains are used in a variety of places, for example:

- Sequence of one-time passwords
- Chain of non-repudiable trust
- Time-lock encryption (addressed in next section)
- Generating arbitrary non-secure pseudorandom bits with very small seed

### 4.2.2 Time-lock Encryption

The essential problem of time-lock encryption is the ability to encrypt data with the following properties:

- The cipher text cannot be decrypted before a given date.
- After the date, without any more action from the encrypter, the cipher text can be decrypted by anyone (or just previously determined parties).

**Example 4.2.1.1.** *Suppose Alice wants to encrypt her message $m$ for a week. She picks a random seed $s_0$ and hashes it continuously for a week to produce $s_N$ ($N$ being the number of iterations of the hash function). She uses $s_N$ to encrypt $m$.*

*Alice wants Bob to be able to decrypt $m$ after one week. She releases to Bob her cipher text $c$, $s_0$, and $N$. Now, if Bob (has identical hardware and) hashes $s_0$ exactly $N$ times, he will reproduce $s_N$ (after a week, supposedly) and decrypt $c$.*

Obviously, there are problems with this example. The first problem is that Alice cannot guarantee that Bob does not have faster hardware than Alice. He may be able to decrypt in less than a week. The second problem is that Alice cannot guarantee that Bob will start hashing immediately, or that he will hash continuously for a whole week. That is, she cannot guarantee that her message will be available *after* the deadline, either. This would particularly be a problem when the time involved in the encryption is quite long (say, years or even decades).

Time-lock of varying lengths of time could be used for:

- Sealed bidding in an auction
- Key escrow schemes
- Wills

As an aside, it is possible for Alice to set up this encryption step *without* hashing for a full week. Instead, Alice starts with $m$ random seeds, $s_1, \ldots, s_m$. Now, for just this example, let a superscript denote the number of iterations of a hash function. Alice computes (in parallel, to mitigate the sequential nature of the computation from the previous example):

$$s_1^N, \ldots, s_m^N.$$

She then uses $s_i^N$ to encrypt $s_{i+1}$ for $i$ in $[1, \ldots, m-1]$. Let us denote the encrypted seeds by $c_2, \ldots, c_m$. As before, she uses the last hash, $s_m^N$ to encrypt $m$ (to produce $c$, the cipher text). Alice then releases $c$, $s_1$, $c_2, \ldots, c_m$, and $N$, of course.

For Bob to decrypt $c$, he must hash $s_1$ until he reproduces $s_1^N$, then use $c_2$ and $s_1^N$ to recover $s_2$, and so on. In this way, Alice can exploit the parallelization to reduce the computation time to $\frac{1}{m}$, but the decryption time remains unchanged. The drawback is that Alice must release $m$ times as much information (space/time trade-off).

Table 4.1: Alternate Approach to Hash Chaining

Previously, there have been two natural approaches to time-lock encryption.

**Time-lock puzzles**. Create a difficult (and non-parallelizable) problem that cannot be solved without running a sequence of computations. Furthermore, these puzzles should be comparatively easy to generate. Some well-known examples were introduced by Rivest, Shamir, and Wagner in [43]. This is similar to hash-chaining in the sense that it relies on a predictable connection between computational time and realtime. The problem is that there is no way to control the computational power of the other parties, especially when the time-lock involved is long enough to take Moore's Law into account.

**Trusted third party**. Alice encrypts $m$ with a key $k$. She gives Charles (the trusted third-party) $k$, and releases the cipher text $c$ to Bob. Alice *trusts* Charlie not to give away the key, and to give Bob the key at the appointed time. Now, admittedly, there is action that needs to be done for Bob to be able to decrypt (namely, Charles must give $k$ to Bob after the agreed-upon amount of time has passed). Alice, however, need not contribute anything more. It bears noting that this solution does not imply that Charles can decrypt just because he has knowledge of $k$. Alice can give Bob "half" of the key $k_1$ and Charlie the other half $k_2$, and use the XOR $k = k_1 \oplus k_2$ as the key to encrypt $m$. Now Bob still cannot

decrypt without Charles' key, and the only way Charles could decrypt is by convincing Bob to release his key.

This is fine, but there is an issue of scalability. If millions of users are trying to invoke time-lock encryption, Charlie must now hold millions of different secret keys, as well as correct release times. It seems as if there is too much room for error or even security breaches. This leads us to our next section detailing a halfway solution using hash chains.

### 4.2.3 Previous Research

Here is a history of previous and related work:

- The question of Timed-release Crypto was first posed in 1993 by Timothy C. May on the Cypherpunks mailing list. At the time, the only solution he gave was that of a trusted third party.

- The largest step forward after that was by Rivest, Shamir, and Wagner in 1996 [43]. Their "Time-lock Puzzles" approach was to ensure decryption relied on solving difficult and "intrinsically sequential" puzzles (the example they gave was repeated squaring). The idea was to eliminate the possibility of parallelization. The solution is considerably better than the "hash-for-a-week" toy example we gave earlier because encryption can be done efficiently, while decryption is inefficient. Also, the difficulty of the puzzle can be easily adjusted.

- Time-lock puzzles were discussed again by Rangasamy et al. in 2011 [42]. Specifically, they introduced *client puzzles*, which are time-lock puzzles posed by a server to a user in order to prove its legitimacy in accessing the server. That is, the puzzles are used to provide protection against Denial-of-Service (DoS) attacks.

- In 2011, Mahmoody, Moran, and Vadhan published a paper [34] tackling the problem of time-lock puzzles using a random oracle. Their main result is that within the

random oracle model (i.e., unrelated to the 1996 solution) "Time-lock puzzles with large difficulty gap are impossible." That is, constructing time-lock puzzles with a gap between the work of the puzzle generator and the puzzle solver is impossible utilizing only a random oracle.

- In 2014, Katz, Miller, and Shi take inspiration from Bitcoin proof-of-work (which is itself built to be a time-lock puzzle) to produce (some) assurance of security [30].

- In 2015, Liu, Garcia, and Ryan released a paper [33] using the method of "witness encryption" and the proof-of-work problems inherent in the Bitcoin block chain to detail a method of time-lock encryption. As it happens, in 2015 another paper [27] by Tibor Jager was published also relying on the Bitcoin block chain and witness encryption to enact time-lock. Witness encryption is described in detail by Garg, Gentry, Sahai, and Waters in a 2013 paper [18]. Witness encryption is currently not considered feasible in practice.

- Also in 2015, Bitansky, Goldwasser, Jain, Paneth, Vaikuntanathan, and Waters published a paper [9] using randomized encodings to construct time-lock puzzles.

We would like to point out that this chapter, while it does make reference to Bitcoin, does *not* use the block chain, the Bitcoin network, or witness encryption. Furthermore, it seems like there has been significant progress made on time-lock puzzles, so our work is unique in that we are trying to approach the problem of time-lock encryption from a different angle. Our beacon method is a halfway approach between a trusted third-party and hash chains.

Though we will address the topic of Bitcoin in detail in the next section, it seems relevant to point out right now that although the Bitcoin proof-of-work functions as a time-lock puzzle, since there is a range of solutions to the puzzle, it is not easy to use the proof as encryption since it cannot be known in advance. This is the beginning of witness encryption, and we encourage the reader to consult the relevant articles [18] [27] [33].

## 4.3 The Beacon

In this section, we introduce the concept of a *beacon*. The idea is that a trusted authority releases values on a timed basis, with the property that they cannot be forged or predicted.

For example, a current work in progress is the "NIST Randomness Beacon," which releases 512-bit blocks every sixty seconds (available at `https://beacon.nist.gov/home`). According to NIST:

> The NIST Randomness Beacon expands the use of randomness to multiple scenarios in which the latter methods cannot be used. The extra functionalities stem mainly from three features. First, the Beacon-generated numbers cannot be predicted before they are published. Second, the public, time-bound, and authenticated nature of the Beacon allows a user application to prove to anybody that it used truly random numbers not known before a certain point in time. Third, this proof can be presented offline and at any point in the future. For example, the proof could be mailed to a trusted third party, encrypted and signed by an application, only to be opened if needed and authorized.
>
> NIST encourages the community at large to research and publish novel ways in which this tool can be used.

In [14], Clark and Hengartner describe methods to extract randomness beacons from financial data.

For our purposes, the beacon will be deterministic and pre-computed, although only for the trusted third-party.

**Example 4.3.0.1.** *Suppose Company X (CX) would like to provide a trusted, third-party, time-lock encryption protocol for its clients. CX generates a random seed $s_0$ and computes the hash chain $s_0, \ldots, s_N$ (recall, with the property that $h(s_i) = s_{i+1}$). For this example, let $N = 1000$, for ease of notation. The beacon that this section is named for is this: CX*

*releases* $s_{1000}, s_{999}, s_{998}, \ldots$, *(that is, in reverse order) once every hour. The point is that any user on the network (perhaps joining for the first time) can check* $h(s_{998}) = s_{999}$ *and therefore trust the source releasing the hashes.*

*Now, Alice approaches CX and asks for a key that will provide a week (168 hours) of time-lock encryption. Obviously, CX does* ***not*** *give Alice* $s_{832}$*, as Alice could compute intermediate hashes that would compromise CX's integrity as a trusted third-party. Instead, CX gives Alice (for example)*

$$K = h(s_{832}||\texttt{alice}),$$

*(where* `alice` *is some bitstring representation unique to Alice, or even just her name). CX also gives her the steps for reproducing* $K$ *once* $s_{832}$ *is released (note that 1000-168 = 832). Alice uses* $K$ *to encrypt* $m$ *to produce* $c$*. She publicly releases* $c$ *and the steps for reproducing* $K$ *once* $s_{832}$ *is released (i.e., "take* $s_{832}$*, concatenate it with* `alice`*, then hash it").*

*At this point, Alice need not take any further action to ensure her message can be decrypted after one week. After 168 hours, CX's beacon releases* $s_{832}$ *and anyone can recover* $K$ *and decrypt* $c$*. In fact, Alice can narrow down who will be able to decrypt by only providing select people with the concatenation string.*

**Remark 3.** *This method raises the problem of storage. How many hashes should CX store? If they store a million hashes, then they can release quickly with no extra computation. But this may be expensive. If they store one in every thousand hashes, then they can store fewer values, but will need to recompute hashes after every thousand releases. A question for further inquiry: how best to address this problem?*

**Remark 4.** *What happens when CX runs out of hashes? Do they regenerate another million? How can trust be maintained or re-established? Goyal in 2004 [23] and Zhao and Li in 2005 [55] discuss methods of re-initializing hash chains.*

### 4.3.1 The Extended Beacon

We would like to extend the above method to allow for more versatility. Suppose Company A (CA) approaches CX and requests some sort of authentication to release hashes in parallel. Specifically, we would like the following properties:

- CA should be able to release hashes, similarly to CX (in reverse order).
- This "side chain" should be linked in some way to the main chain, in the sense that users that trust CX can confirm the validity of CA.
- Users (still) cannot compute unreleased hashes from either company.
- CA cannot compute unreleased hashes from CX.

**Example 4.3.0.2.** *Suppose CX is releasing (as before)* $s_{1000}, s_{999}, s_{998}, ...,$ *and so on, every hour. If CA requests a 500 hour license to release hashes, CX can give CA* $t_{500} = h(s_{500}||\texttt{somekey})$ *(or some other derivative of* $s_{500}$*), and CA can now iteratively hash* $t_{500}$ *and release the hashes in reverse order. To illustrate, consider Figure 4.1. Arrows indicate hashing.*

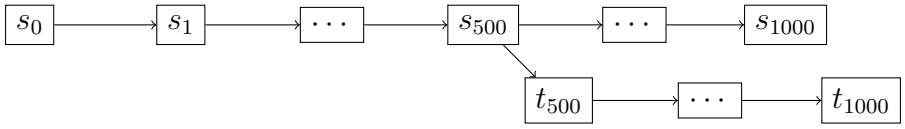


Figure 4.1: Parallel Hash Chains

The problem in this example is that as $(s_i, t_i)$ are released from the respective companies $(i = 1000, 999, \ldots)$, there is no way for users to check that the $t_i$ are derived from the $s_i$.

### 4.3.2 The Ideal Solution

Ideally, the sort of solution we desire would be one where on each release, users could verify that $t_i$ was derived from $s_i$. Consider Figure 4.2.

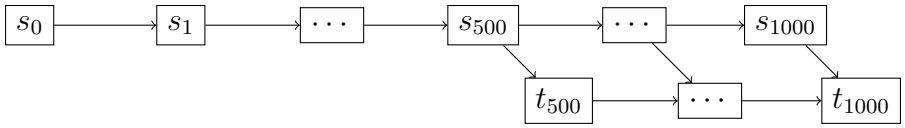


Figure 4.2: Commutative Parallel Chains

It would certainly be nice if we had a pair of commutative hash functions, but that is an entire problem for research in itself. Instead, we propose a solution using RSA private keys.

**Example 4.3.0.3.** *Let $N$ be an RSA modulus. Let $e$ and $d$ be public and private RSA exponents (respectively) for CX. Let $\bar{e}$ and $\bar{d}$ be public and private RSA exponents corresponding to CA (note later that CA will* ***not*** *have knowledge of $\bar{d}$). Let $x$ be some private bitstring known only to CX.*

*Instead of hashing, CX should compute*

$$x, x^e, x^{e^2}, \ldots, x^{e^{1000}}.$$

*These are the values that will be released in reverse order. As a side note, it may actually be more convenient to think of these as:*

$$y^{d^{1000}}, y^{d^{999}}, y^{d^{998}}, \ldots, y,$$

where $y = x^{e^{1000}}$.

The benefit of this method is that CX is not limited to 1000 values and can continue computing them (exponentiation to the $d$th power).

Now, again CA would like 500 hours of authenticated time-lock encryption power. CX releases to CA $x^{\bar{e}e^{500}}$ (or $y^{\bar{e}d^{500}}$). CA can now compute

$$x^{\bar{e}e^{500}}, x^{\bar{e}e^{501}}, \ldots, x^{\bar{e}e^{1000}}.$$

These ideas are illustrated in Figure 4.3. It has been designed to mimic our ideal con-

struction in Figure 4.2. Horizontal arrows indicate exponentiation to the $e$ power. Diagonal arrows indicate exponentiation to the $\bar{e}$ power.

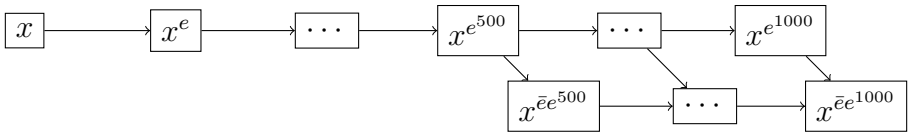


Figure 4.3: RSA Parallel Chains

Now, as $(x^{e^i}, x^{\bar{e}e^i})$ gets released, users can verify CA's value by raising CX's value to the power of $\bar{e}$. This is trusted because the only entity with knowledge of $\bar{d}$ is CX itself. Furthermore, exponentiation also takes the place of hashing as the values are linked horizontally by exponentiation to the power of $e$. CA cannot compute any of CX's unreleased values because it lacks $\bar{d}$. Furthermore, CA cannot compute values to the "left" of $x^{\bar{e}e^{500}}$ because it lacks $\bar{d}$. Finally, CX need not precompute any values. It can compute to the left using exponentiation to the power of $d$ (which nobody else has).

Some further questions:

- Does this construction work with elliptic curves?
- Can we construct an example without such reliance on exponentiation, which can become computationally expensive?

The answer to the first question is no, at least, not exactly. There is no analog of RSA in an elliptic curve group because the size of the group gives away inverses. Typically, the problem invoked in an elliptic curve group is the Discrete Log Problem (DLP). Could the DLP be exploited to set up a similar construction?

The rest of this chapter is dedicated to answering the second question.

### 4.3.3 Hierarchical Deterministic Wallets

Bitcoin users send and receive money using addresses. These addresses are public keys (or hashes thereof), which correspond to private keys that only the corresponding user knows.

- An integer $s_0$ is pseudorandomly generated. This is the master parent private key.
- Signing in Bitcoin is done with ECDSA (Elliptic Curve Digital Signing Algorithm). There is a fixed, global elliptic point, called $g$. The master parent public key is $p_0 = g \cdot s_0$. The public key (or rather, the hash thereof) is the Bitcoin address the user broadcasts to the network.
- An integer $c_0$ (called the *chain code*) is generated pseudorandomly.
- To derive a child private key (say, child with index $i$), take a keyed hash function $h$ (Bitcoin uses HMAC, with SHA-512), and compute: $h(c_0, p_0||i)$ (where $c_0$ is the key and $p_0||i$ is the message).
- Take the left 256 bits of the output ($L256$), and $s_1 = s_0 + L256$.
- The right 256 bits become the new chain code, $c_1$.
- As with the parent keys, the new public key can be computed as $p_1 = g \cdot s_1$. But observe that $p_1$ can be computed without knowledge of $s_0$ or $s_1$:

$$\begin{aligned} p_1 &= g \cdot s_1 \\ p_1 &= g \cdot (s_0 + L256) \\ p_1 &= g \cdot s_0 + g \cdot L256 \\ p_1 &= p_0 + g \cdot L256 \end{aligned}$$

Table 4.2: HD Wallet Key Derivation: Algorithm

Bitcoin uses ECDSA for signatures and keys. For privacy and tracking reasons, it is desirable to use a different Bitcoin address for every transaction. A user's collection of addresses is called a "wallet." Because a single user may end up making arbitrarily many transactions, if the addresses are generated pseudo-randomly, it quickly becomes cumbersome to hold all of them in memory.

Instead, Bitcoin clients implement *hierarchical deterministic* wallets (or HD wallets). Starting with a random seed, addresses are derived hierarchically (as in a tree, starting with the root) and are indistinguishable from pseudorandomly generated addresses, using a keyed hash function (hence, deterministically).

The method for deriving addresses is outlined in Table 4.2.

A desirable property of this construction is that public keys can be computed without knowing the private keys. As an application, the CEO of Company X would like to endow his subordinates with the ability to set up accounts to receive funds, but not to spend any of the funds. The CEO, who holds the master secret key, can compute all of the public keys and corresponding secret keys.

Side note: knowledge of the master public key and *any* private key is sufficient to compute the master private key. As we will discuss later, it is possible to derive child private keys using the parent private key hash (instead of the public key hash), although this does break the public key property (that all public keys can be computed without knowledge of the private keys). Gutoski and Stebila [25] provide a layer of protection against this problem (key leakage) while maintaining the public key property. See also [20] for another scheme improving the security of Bitcoin wallets.

In Figure 4.4, we have depicted the algorithm for deriving child private and public keys.

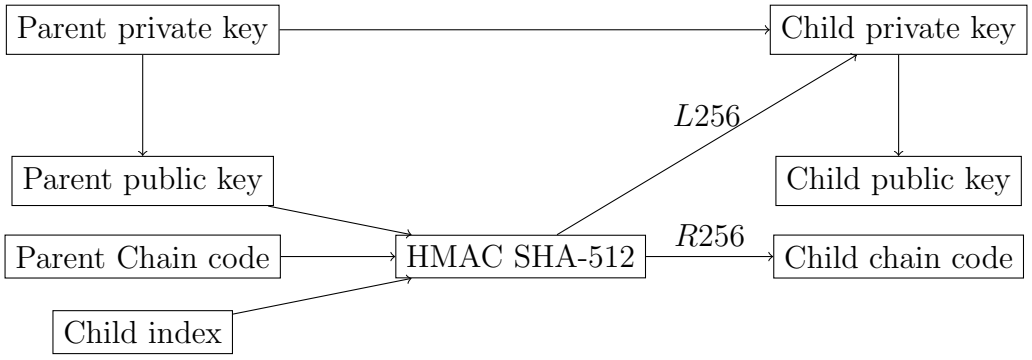


Figure 4.4: HD Wallet Key Derivation: Illustration

## 4.4 Main Result

### 4.4.1 Contribution

In this section, we propose an alternate solution to the Extended Beacon problem described in Section 4.3.1. Our method does not rely on exponentiation and the invocation of RSA exponents. Instead, our construction is a mix of elliptic curve multiplication and hashing. We make use of the HD Wallet structure described in Section 4.3.3.

We will discuss two constructions, each with its advantages and disadvantages. The first construction exhibits the following properties:

- Arbitrarily many side beacons.
- Slow verification.

The second construction has the following properties:

- Limited amount of side beacons.
- Fast verification.
- Larger signatures.

We will spend the next two sections detailing our two constructions.

### 4.4.2 Construction 1

We are going to build this up step-by-step with detailed pictures.

**Step 1.** Company X (as before) is the master company, and (as in the Bitcoin protocol) there is an agreed upon elliptic point $g$, a generator. CX generates a pseudorandom seed $S_0$ (an integer). This is the master private key. Compute $P_0 = g \cdot S_0$, the master public key. Hash $P_0$ (using the chain code and child index, a la HD Wallets, although we are going to

leave the chain code out of the picture, for the sake of simplicity) to obtain $L256$ and $R256$. Then, compute:

$$S_1 = S_0 + L256$$

$$P_1 = g \cdot S_1.$$

But as before, we have the nice property that $P_1 = P_0 + g \cdot L256$.

We can continue generate child private keys and the corresponding child public keys in this manner (Figure 4.5). The dashed line indicates multiplication by $g$.

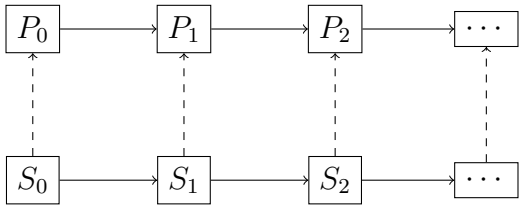


Figure 4.5: The Extended Beacon: Attempt 1

Note that in Figure 4.5, horizontal arrows imply an addition of $L256$ (or $g \cdot L256$ in the case of the public keys). Now, if we were to naively compare this to our previous examples of hash chains, we might assume that CX would release the master public key (and therefore sufficient information to compute all the public keys), and then the private keys in reverse order. Thus, users could check that each released private key matches with the corresponding public key (by multiplying by $g$). Unfortunately, because of the way the private keys are derived, namely, with the hash of the public keys, once a single private key is released, *all* the private keys can be recovered. This is highly undesirable. Instead, CX should release signatures using the private keys in reverse order. At this point, it is not important what the message being signed is, simply that CX is proving knowledge of the private keys *without* releasing them.

For example, assume $P_0$ through $P_{1000}$ are known. Instead of releasing $S_{1000}$, CX uses $S_{1000}$ to sign the message "0." Users can then use $P_{1000}$ to verify the signature.

To sell a time-lock encryption key, CX would release a derivation of the *signature* using the appropriate private key. For analogy to our hash chain example from before, suppose Alice, a customer, requests a week (168 hours) of time-lock encryption. CX would give to Alice $h(\sigma_{832}(0)||\texttt{alice})$, where $\sigma_i$ indicates a signature using $S_i$. Alice can use this to encrypt her message, and after 168 hours, $\sigma_{832}(0)$ would be released, at which point anybody with Alice's ciphertext could decrypt.

Of course, we are trying to find a way to answer the Extended Beacon problem of Section 4.3.1. With that in mind, our next goal is to try to create a tree structure of private and public keys where CX controls the main chain and sells information to a child company to create its own beacon. We will refer to this child company as Company A (CA).

The first thing that springs to mind is to, at some point in the chain, hash the public key using a different child index (as is the method for deriving the tree structure in HD wallets).

Consult Figure 4.6 for a suitable picture illustrating the process so far. Note that $T_{99}$ is derived from $P_{98}$ in the same way as $S_{99}$, although with a different child index.

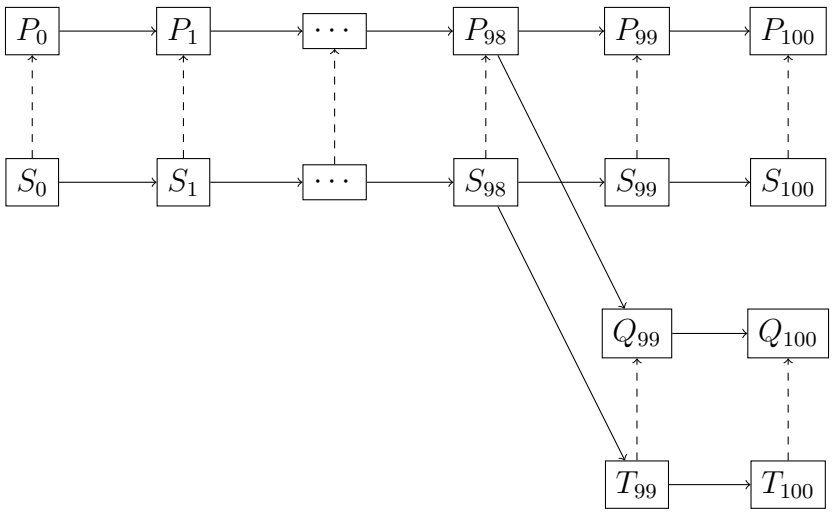


Figure 4.6: The Extended Beacon: Attempt 2

Aesthetically, the method of Figure 4.6 is pleasing. It does not work, however. As it currently stands, when the child company receives $T_{99}$, they can compute $S_{98}$ (since $P_{98}$ is

public) and then compute all of the $S_i$.

We can modify it by requiring that $T_{99}$ be derived from hashing $S_{98}$ instead of $P_{98}$. In the context of HD Wallets, this is called "hardened key derivation." The idea is that it creates a firewall between $S_{98}$ and $T_{99}$ wherein knowledge of all the public keys and $T_{99}$ does not reveal $S_{98}$, which is desirable. Specifically, we hash $S_{98}$ and take the left 256 bits (call it $L256$ as before, but with the understanding that this cannot be publicly computed), and $T_{99} = S_{98} + L256$. CX is the only entity with access to $S_{98}$ so the child company CA cannot reverse this process. Since there is no link from $P_{98}$ and $Q_{99}$, we simply compute $Q_{99} = g \cdot T_{99}$.

Under this paradigm, we still need some way for the public to verify that $T_{99}$ is linked to the main chain (since now there is no relation between $P_{98}$ and $Q_{99}$). When CX issues $T_{99}$ to the child company CA, it should also release a signature (using $S_{98}$) confirming $Q_{99}$. At this point, the only reason that $T_{99}$ is generated deterministically and not pseudorandomly is for storage reasons. CX need not store $T_{99}$ because they know exactly how to rederive it using $S_{98}$. This is important if CX generates enough child companies for storage to be an issue.

Figure 4.7 illustrates the final construction. The thick line indicates a hardened key derivation.

Let us look at an overview of the entire process.

- Company X (CX) generates a sequence of private and public keys with the property that all the public keys can be computed sequentially.
- CX releases the master public key and uses the private keys in reverse order to sign a simple message such as "0." This is the main beacon.
- CX sells a derived private key to a child company (CA) such that this child can act independently as its own beacon but cannot reverse compute CX's private keys. CX must also issue a signature (a certificate of sorts) using the private key at the point of branching.

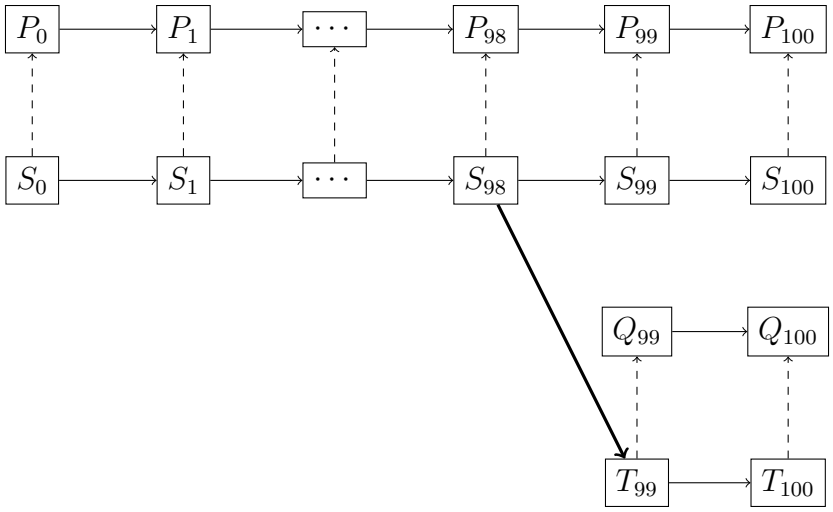


Figure 4.7: The Extended Beacon: Attempt 3

- Users can use the public keys to verify signatures from both CX and CA. They can also use the certificate mentioned in the previous step to verify that CA is linked to CX (and furthermore, linked at a very specific point in "time").

- Note that, from the above step, a user must follow the public keys from the branching point down to the currently released signature (there should be an unbroken path from the child branching point to the currently released value). This is what causes the slow verification.

- CX can have arbitrarily many child companies.

- The parent and child companies actually do not need to release their beacons in synch. As long as the child is not exceeding time-lock issuances longer than the initial length given by the parent, they can release their beacons as frequently as they desire.

### 4.4.3 Construction 2

In this section, we are going to slightly tweak the HD Wallet structure to allow for faster verification at the expense of storage and limited children.

As before, Company X (CX) should pseudorandomly generate a master private key, $S_0$.

Instead of having just one elliptic point, let $g_1, \ldots, g_n$ be generators in an elliptic curve group. Furthermore, let $P_{0,1}, \ldots, P_{0,n}$ be defined by

$$P_{0,i} = g_i \cdot S_0.$$

Let $S_1$ be defined as follows:

$$S_1 = h(P_{0,1}|| \cdots ||P_{0,n}) + S_0.$$

Then it is still justifiable to define $P_{1,i} = g_i \cdot S_1$, because

$$\begin{aligned} P_{1,i} &= g_i \cdot S_1 \\ &= g_i \cdot h(P_{0,1}|| \cdots ||P_{0,n}) + g_i \cdot S_0 \\ &= g_i \cdot h(P_{0,1}|| \cdots ||P_{0,n}) + P_{0,i}. \end{aligned}$$

The point is that the public keys $P_{k,i}$ are computable without knowing any of the private keys.

Now, the sake of an actual example, let's suppose that Company X has three child companies, denoted C1, C2, C3. Correspondingly, we have three elliptic points $g_1, g_2, g_3$, and three master public keys: $P_{0,1}, P_{0,2}, P_{0,3}$. The beacon in this example is similar: CX releases three signatures, using $S_k$, each corresponding to one of $P_{k,i}$. The child companies each release $P_{k,i}$. Users can verify the signatures using the published public keys. Some things to note:

- Though we still call them "public" keys, the child companies must keep them secret (until the scheduled beacon release).
- Regardless of this secrecy of the public keys, we still cannot allow the parent company

to release the secret keys. Otherwise the child companies would be able to reverse engineer the secret keys (although only up to the highest level public key they have access to, and even then they need the *three* public keys across the child companies to do this reverse computation). We can certainly expect that the three child companies will cooperate.

- The parent company sells time-lock encryption keys derived from their secret keys, while the child companies sell time-lock encryption keys derived from their public keys (but keep in mind that these keys are secret from the users).

Consult Figure 4.8 for a visual illustration of the construction. Recall that $P_{k,i} = g_i \cdot S_k$.

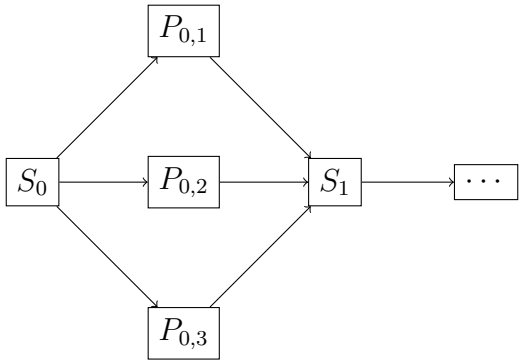


Figure 4.8: The Extended Beacon: Variation

## 4.5 Summary

*Perfect* time-lock encryption would enable the encrypter to publish ciphertext, and ensure that without any further action from themselves *or even a trusted third party (TTP)*, the ciphertext could be decrypted but only after a given date.

The best known methods of tying computational time to realtime involve generating so-called time-lock puzzles. The only guaranteed method of having an exact release date is entrusting the decryption key to a TTP. Since there is a decent body of research attacking time-lock encryption itself, we opted to approach a slightly different problem. The problem:

if a single entity can issue time-lock encryption keys based around a trusted beacon, how can that entity authorize an intermediate company to do issue time-lock encryption keys, while maintaining a verifiable relationship from parent to child company?

Our solution is a compromise between the encrypter and the TTP, wherein the TTP releases a beacon at predictable intervals, maintaining trust via hash chains (or with a more complicated system). The TTP can then sell time-lock encryption keys whose derivation is publicly known but whose source is only released at a given time.

We further want to provide the TTP with the ability to certify child companies to sell time-lock encryption, transferring trust from the parent company. Of course this is possible using an RSA modulus and public and private key pairs. In an attempt to get away from the reliance on RSA and exponentiation, we use the hierarchical structure built into Bitcoin's HD Wallets to provide a practical workaround. Elliptic curve multiplication and hashing take the place of exponentiation. We are excited about this intersection of cryptographic problems and Bitcoin structures and hope that it inspires further work using and improving Bitcoin.

In closing, here are some questions and open problems for the future:

- Remark 4.3 from Section 4.3 still stands. To optimize space and minimize re-computation, how many hashes (or public keys) should Company X keep in temporary storage as they are releasing values?
- Can we create an Extended Beacon paradigm that only depends on hashes (and not exponentiation or elliptic curve multiplication?)
- How can we continue to improve Bitcoin?
- How can Bitcoin structures be subverted for other cryptographic uses?

# Chapter 5

# Bitcoin Mixing with Contracts

## 5.1 Overview

Since its implementation in 2009, Bitcoin has seen explosive rises in popularity. Until early 2013, the value of 1 BTC had not exceeded 15 USD. By October later that year, the value had risen to 200 USD. It reached an all-time high of nearly 1200 USD in mid-November. After falling quickly, the value of 1 BTC has fluctuated between 200 and 500 USD.

There are currently in excess of fifteen million bitcoins in existence. At a value of around 400 USD per bitcoin, the cryptocurrency boasts a market capitalization of more than six million USD in circulation. By comparison, Ethereum—the next largest cryptocurrency by market capitalization—is worth just under one million USD (taken from `https://coinmarketcap.com/`).

Each block on the block chain can contain up to 1500 transactions. The total value of bitcoins exchanged in each block varies from between one to ten million dollars. Since blocks are mined on average at a rate of six per hour, these blocks represent the exchange of six to sixty million dollars every hour. (All of this information and more is available on `https://blockchain.info/`).

All of these transactions are publicly and indefinitely available on the block chain. There

are pros and cons to this. Obviously, the public nature of the block chain is what makes distributed consensus (and thus, the whole cryptocurrency) possible at all. However, it severely limits privacy for its users.

We have already discussed the importance of using multiple addresses for transactions, though a natural issue comes up that is not simply solved with multiple addresses (or HD wallets, which the solve the problem of scalability, Section 4.3.3). If Alice receives 10 BTC in address $A_1$, then in order to spend it, she must sign with the private key corresponding to $A_1$. If Alice sends Bob the 10 BTC, now Bob can see where Alice received the money in the first place. Even sending the money through dummy addresses does not accomplish anything, as Bob can always trace the transactions back their source.

So, this is the privacy problem. How can we guarantee for Bitcoin users that their transactions cannot be traced?

## 5.2 Transactions

Please see Sections 2.9 and 2.9.5 for the basics of Bitcoin and Miners.

In this section, we will thoroughly discuss transactions, the foundational data structure for exchanging money on the Bitcoin network.

### 5.2.1 Structure

A *transaction* is a structure that encodes information about the exchange of funds.

See Table 5.1, adopted from Antonopoulos [2].

The *locktime* field allows a user to specify the *earliest block or time that may include a transaction.* If locktime is nonzero and below 500 million, it is interpreted as a block height (meaning the transaction cannot be included in the block chain prior to the specified block height). If it is above 500 million, it is interpreted as a Unix Epoch timestamp (seconds since January 1, 1970), and the transaction is not included in the block chain prior to the

| Field | Description |
|---|---|
| Version | Specifies which rules this transaction follows |
| Input Counter | How many inputs are included |
| Inputs | One or more transaction inputs |
| Output Counter | How many outputs are included |
| Locktime | A Unix timestamp or block number |

Table 5.1: Transaction Structure

specified time.

### 5.2.2 Inputs and Outputs

It is not correct to think of bitcoins residing in any given account. Rather, it is better to view an Unspent Transaction Output (UTXO) as claimable money (and usually, the encumbrance can only be lifted by a single user). An account, then, is simply a scattered list of UTXOs on the block chain that can be claimed by a given private key.

The smallest indivisible unit of bitcoins is called a "satoshi," in honor of Satoshi Nakamoto, and one satoshi $10^{-8}$ BTC. A UTXO is recorded as a multiple of satoshis. When spending a UTXO, the entire value must be consumed in the transaction. The standard way for a user to make change is to list multiple outputs, sending some back to himself. That is, if Alice is sending 1 BTC to Bob, but her only UTXO is in the value of 1.7 BTC, then she would make a transaction with 1.7 BTC as the total input, 1 BTC to Bob as one output, and 0.7 BTC to herself as another output.

### 5.2.3 Transaction Fees

For a given transaction, the sum of the inputs certainly must be at least as large as the sum of the outputs. However, the difference (inputs - outputs) is interpreted as the *transaction fee*, and the miner that includes the transaction in a block awards himself this fee (as well as the block reward).

## 5.3 Scripts and Contracts

Transactions are quite a bit more complicated than we have discussed so far. It is not as simple as "Alice pays Bob 100 USD."

What actually happens is that Alice places an *encumbrance* or *locking script* on 100 USD, such that the only way to satisfy the encumbrance is with a corresponding *unlocking script* containing Bob's signature. Alice can use Bob's public key to make the locking script, but only Bob has the private key to make the signature.

In a high level description, when Bitcoins are being spent, there are two transactions to consider. The output of Tx1 by Alice, which locks the money to Bob's address, and the input of Tx2 by Bob which proves ownership of the public key.

In technical details, when evaluating validity of transactions, the unlocking script of Tx2 is placed on a stack. When the stack is finished evaluating, the output is sent to the top of the stack containing the locking script of Tx1. If it evaluates to `True` then Tx2 is considered valid. In the original Bitcoin client, the unlocking and locking scripts were concatenated and executed in sequence. For security reasons, this was changed in 2010, because of a vulnerability that allowed a malformed unlocking script to push data onto the stack and corrupt the locking script. In the current implementation, the scripts are executed separately with the stack transferred between the two executions [2].

The Bitcoin transaction script language, called *Script*, is very simple, designed to be limited in scope and executable on a range of hardware [2]. The language used in the scripts is simple, allowing for basic mathematical operations, hash functions, and boolean logic.

There are some keywords baked into the script that the Bitcoin core developers have agreed on. Occasionally, new keywords are added to (or redacted from) accepted usage.

This means that locking scripts could be as simple as $n-1=0$, and anyone could "claim" this transaction with an unlocking script of $n=1$.

The majority of Bitcoin transactions use "pay-to-public-key-hash" transactions: Alice locks money with (the hash of) Bob's public key. The unlocking script will contain Bob's

signature. A detailed description of this execution is shown in Figures 5.1 and 5.2 (Note: figures adopted from [2]). In this example, we consider the locking and unlocking script given by:

$$\overbrace{\texttt{<sig> <PubK>}}^{\text{Unlocking Script}}\overbrace{\texttt{DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG}}^{\text{Locking Script}}$$

The value `PubK` is Bob's public key, and `PubKHash` the hash. The value `<sig>` is the (ECDSA) signature corresponding to Bob's public key. Alice creates the locking script with only knowledge of Bob's public key, but Bob must know the private key corresponding to the public key to create the unlocking script.

One might ask why the above script cannot be written as follows:

$$\overbrace{\texttt{<sig>}}^{\text{Unlocking Script}}\quad\overbrace{\texttt{<PubK> CHECKSIG}}^{\text{Locking Script}}$$

The answer is that Bitcoin addresses are broadcast as hashes, so the payee cannot provide a full public key in the locking script. For readability, however, we will often shorten scripts to an abbreviated form like this.

**Example 5.3.0.1** (An Actual Transaction)**.** *Block 272667 contains a transaction which employs the following as a locking script:*

$$\texttt{OP\_HASH256 } s \texttt{ OP\_EQUAL},$$

*where*

$$s = 6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000.$$

*What this means is that a valid unlocking script will simply be a value whose double hash*

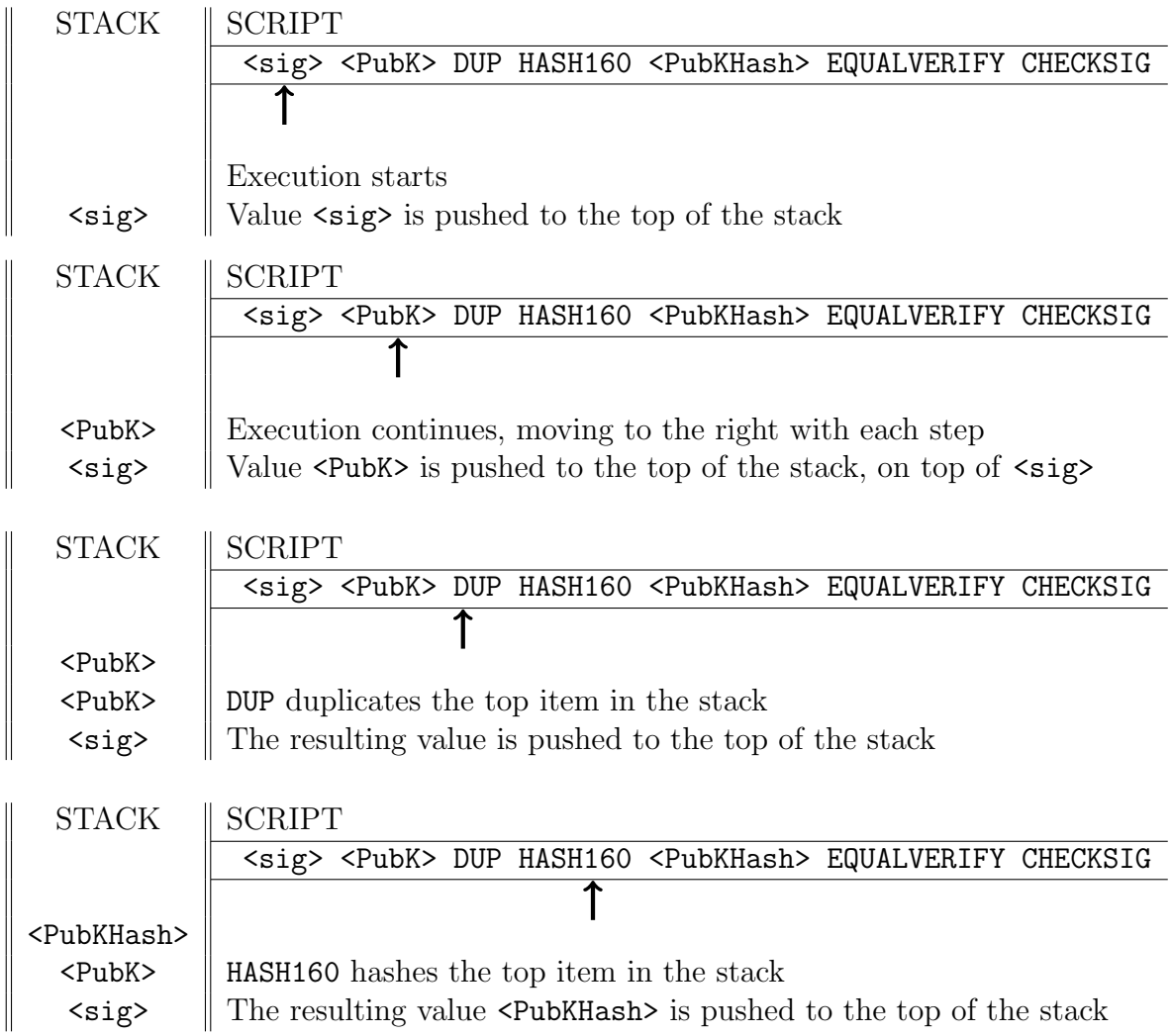


Figure 5.1: Evaluating a Script

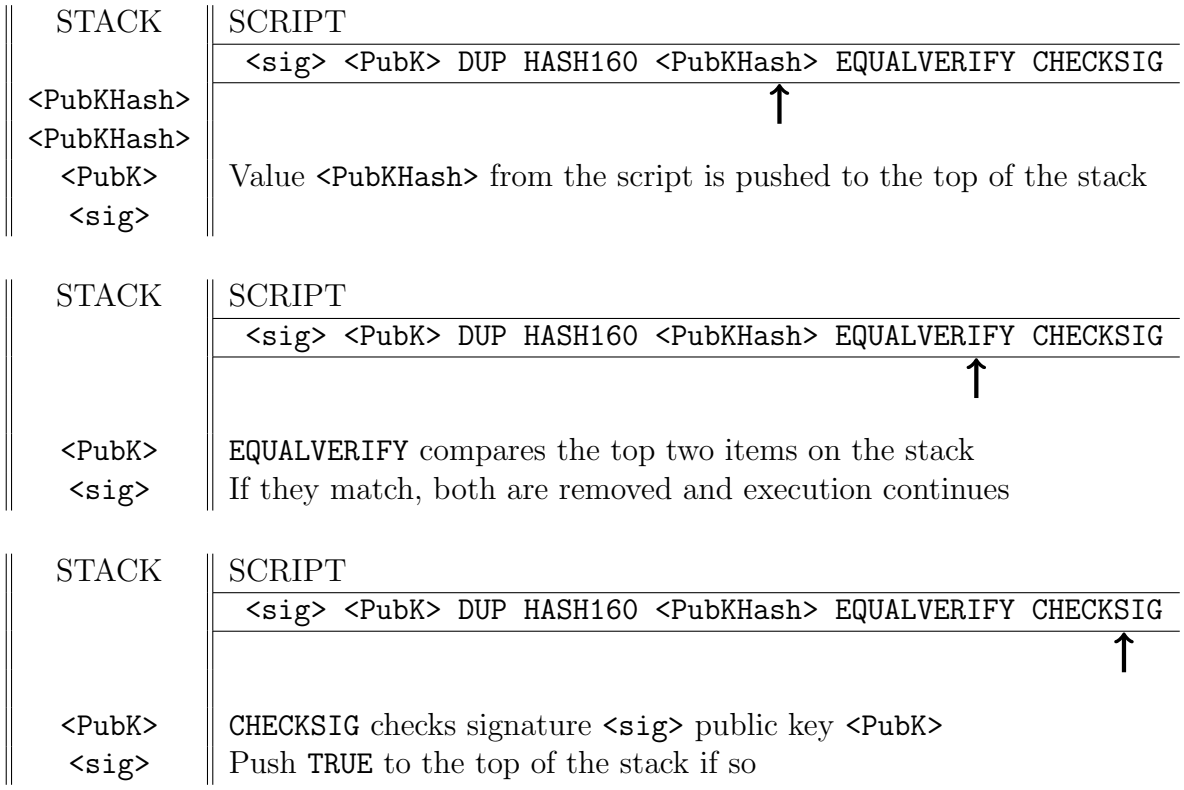


Figure 5.2: Evaluating a Script (cont.)

*(using **SHA**256) is equal to s. This transaction is viewable at*

*https://blockchain.info/block-index/272667*,

*hash starting with* `a4bfa8ab`. *The output is 1 BTC.*

Of course, scripts can be made quite complicated, and we call these contracts. Contracts do not make anything possible that was not already possible, but they have the potential to automate many types of monetary transactions, effectively reducing trust and speeding up the processes.

**Example 5.3.0.2** (CHECKLOCKTIMEVERIFY)**.** *The CHECKLOCKTIMEVERIFY operator allows transactions to be included in the block chain with an encumbrance specifying that the transaction that spends it must have locktime at least a given threshold.*

*Formally, CHECKLOCKTIMEVERIFY compares the item on the top of the stack to the locktime field of the spending transaction. If the stack item is greater than the locktime field, then the script terminates with an error. (That is, the spending transaction will not be considered valid).*

**Problem 5.3.0.1** (Coin Freeze)**.** *Because we will be using the idea extensively, we would like to introduce the idea of coin freeze, a method of locking up or freezing coins. Examples where this will be useful are bountiful.*

*Alice broadcasts a transaction* `Tx1` *in the amount of 1 BTC. The encumbrance placed on the output is this: only Alice can sign for the money, and using CHECKLOCKTIMEVERIFY, Alice specifies that the transaction that spends* `Tx1` *must have locktime* $> 500,000$. *Since* `Tx1` *itself has no locktime restriction, it is valid for immediate inclusion in a block. After it has been embedded into the block chain, Alice now has 1 BTC "locked up" in the sense that she possesses the private key to sign for it, but she cannot spend it until block 500,000.*

*The locking script is shown in Table 5.2.*

```
//CoinFreeze (1 output)
500000 CHECKLOCKTIMEVERIFY <PubK>
```

Table 5.2: CoinFreeze Locking Script

*The corresponding transaction must have locktime* $> 500000$*, and the unlocking script would still simply be* `<sig>`*.*

*We will reference the coin freeze protocol as necessary. Intuitively, freezing coin can be used as a pledge or a sign of limited trust.*

Next, we will provide several scenarios where scripts, contracts, and coin freezing would be useful.

**Example 5.3.0.3** (Kickstarter Account)**.** *Alice announces to the community a bid for funds to create an independent film. Users can sign money to her in the form of an unfinished transaction, with the explicit condition that the output must be (for example) 10,000 USD. If Alice does not receive enough support, then the transaction will never be considered valid and thus will never be included in the block chain. Thus, there is no risk for the supporters in losing an investment.*

**Example 5.3.0.4** (Transaction Mediation)**.** *Alice, Bob, and Marty the mediator are arranging a large transaction: Bob wants to buy a car from Alice for 50 BTC. Since this is a large sum of money, they are being very careful regarding trust. Let us also assume that Marty is an honest mediator, and cannot be corrupted or bought out.*

*First, Bob signs the money to a script that says two out of three signatures (from Alice, Bob, and Marty) must be present in order for the money to be spent. Now, he waits for the transaction to be safely embedded (six blocks deep) in the block chain. At this point, he anticipates that Alice will give him the car soon.*

*If he does not receive the car, the mediator sides with Bob, and the two of them sign the money back to Bob. If he does receive the car, but refuses to sign the transaction, then the*

*mediator sides with Alice and the two of them sign the money to her.*

**Example 5.3.0.5** (Wills)**.** *An older gentleman would like a sum of money to be given to his grandson on his eighteenth birthday or after his (the grandfather's) death, whichever comes first. On the first day of the year, the grandfather signs the money to a script that does not allow for the money to be signed for by anyone besides himself, but after a year has passed, the money can be signed for by the grandson. As it stands, the grandson could claim the money after a year. However, just before the year is up, the grandfather spends the money by signing it to a new "one-year holding account." If the grandfather dies in the intervening time, then the grandson will have to wait at most a year before he can claim the money.*

**Example 5.3.0.6** (Donation to the Homeless)**.** *Alice would like to donate to the homeless man outside Restaurant Chain X (RCX), but she is concerned he will simply buy alcohol or drugs. However, what Alice can do is tip the homeless man a Bitcoin transaction that signs money to a script such that he can only spend it if it is cosigned by RCX. Thus, the money is useless to the man unless he spends it at RCX. Furthermore, Alice can employ a coin freeze (Problem 5.3): the encumbrance is set so that the homeless man can spend it (with the signature of RCX) within one day, and after one day, Alice can sign for it again. Thus, the money is only "frozen" for Alice.*

**Example 5.3.0.7** (An Alternative to DRM)**.** *As digital content becomes ubiquitous, and lawmakers struggle to keep up, it is of the utmost importance to find ways to satisfy both the content-creators and the consumers. Digital Rights Management (DRM) is a temporary solution; in fact, it is not really a solution at all. Pirates still pirate content (music, movies, or video games), and honest consumers are punished (playing games, music, or movies must be linked to a physical machine or an online account).*

*Here, we suggest a primitive method to potentially eliminate the need for DRM-embedded video games. It is similar to Example 5.3. Jon, a video game creator, has finished a game but has not released it yet. He knows that as soon as a digital copy is available, there is*

*essentially no way to force pirates (or anyone, really) to pay for the game. So, he crunches some numbers based on his last successful video game: some people are willing (and able) to pay 40, some 20, some 10, or 5 USD. Even the stubbornest pirate is probably more willing to honestly pay 1 dollar then illegally download a video game. Based on these numbers, he comes up with a grand total of (say) 500,000 USD.*

*After generating a fresh Bitcoin account, Jon announces his account and his goal of 500,000 USD. Alice, a user who is willing and able to buy the game at 40 USD, submits a transaction with the following properties:*

- *Input = 40 USD*
- *Output = 500,000 USD*
- *CHECKLOCKTIMEVERIFY: one month*
- *Within one month, only Jon can sign*
- *After one month, only Alice can sign*

*For the duration of a month, Alice has frozen an amount equal to 40 USD such that only Jon can spend it. This is a "pledge": Alice is saying that she is willing to forfeit the opportunity cost of having the 40 USD in liquid assets with the hopes of getting a DRM-free game.*

*If Bob receives enough transactions to total 500,000 USD, he releases the game DRM-free. If he does not receive enough money in a month, all the money is unfrozen and goes back to its users, and Jon releases a game with embedded DRM. At least we are no worse off than where we were before.*

Contracts are not widely implemented, and the transactions must be hard coded (i.e., cannot be done with the Bitcoin core frontend)

## 5.4 Description of the Problem

True anonymity is difficult to achieve in Bitcoin precisely because of the public nature of the block chain.

Simply stated, if a user receives money in account $A$, he must eventually spend it from account $A$ (and therefore everybody would know where the money originated). If this is undesirable, then the question is: how can bitcoins be laundered?

## 5.5 Previous Work

### 5.5.1 Bitcoin Exchanges

A *Bitcoin exchange* is a service that allows its customers to trade USD (or other cryptocurrencies) for Bitcoin. There are a number of these businesses, mostly centralized, but since their main function is to exchange currency, the best way to achieve privacy would be to trade in bitcoins for USD (for example), then trade back USD for bitcoins. This would be costly, and essentially equivalent to the service offered by mixes, as we will see in the next section.

### 5.5.2 Mixing Services

There are a number of mixing servers, also called *mixes*, available already. The principle behind server-based mixing is simple: users transfer money to the mix via a Bitcoin transaction, and the mix uses previously received funds to transfer an equal amount of money back to the user. In Figure 5.3, Alice transfers 10 BTC to the mix. The mix has 10 BTC (in total) from previous clients and it transfers these back to Alice. Clearly, Alice is placing trust in the mixing service not to simply steal her coins. To quote a line from BITMIXER, "Our business model relies upon having an outstanding reputation in the Bitcoin community." (`https://bitmixer.io/`). Furthermore, a mix must start with a fairly large reserve

of bitcoins.

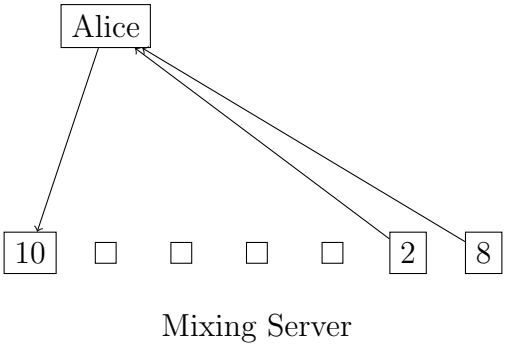


Figure 5.3: Mixing

Some downsides to mixing services:

- A centralized system runs counter to the whole point of Bitcoin
- Trust is located in a single point (of possible failure or corruption)
- Users receive other users' "dirty" money
- The server knows the user's input and output addresses

A list of mixing services:

- BITMIXER
- Bitcoin Fog
- Bitlaunder
- Coinmixer
- Bitlaundry
- Bitcoin Laundry
- SharedCoin

For further reading, in 2013, Möser, Böhme, and Breuker ran experiments on Bitcoin Fog, BitLaundry, and SharedCoin (formerly Send Shared) to see whether they could successfully link input and output addresses after mixing [38]. They discovered that BitLaundry did not successfully anonymize transactions.

### 5.5.3 CoinJoin

CoinJoin is an unimplemented method of bitcoin transaction obfuscation, which improves privacy by grouping $n$ users into a single transaction. The transaction would have $n$ inputs and $n$ outputs, all agreed upon by the users. The transaction thus acts as a "hub" by which an output cannot be traced back to a single input. CoinJoin was first introduced in 2013 on a Bitcoin forum (`https://bitcointalk.org/?topic=279249`).

In Figure 5.4, $n$ users have (say) 1 BTC each ready to spend from addresses $A_1, A_2, \ldots, A_n$. Each generates a "fresh" address $B_i$ (as well as the corresponding private key). They must agree on $n$ output addresses, $B_1, B_2, \ldots, B_n$. There are ways of communicating these output addresses anonymously, but more on that later. They each sign the input and combine the signatures into one transaction. This transaction is broadcast to the entire network. After it is recorded into the block chain, the $n$ users can use their fresh private keys to spend the money sent to the $B_i$ addresses.

In theory, CoinJoin seems like an elegant solution. The single biggest problem is the threat of Denial-of-Service (DoS) attacks. If Alice, Bob, and Eve have met and exchanged information to create a CoinJoin transaction, then Eve can simply refuse to sign the transaction and it will never be broadcast. Furthermore, a single user with a fixed amount of money can enact DoS attacks on the whole network of CoinJoin users, effectively ensuring that nobody would be able to mix their coins.

Note: it is not good enough for honest CoinJoiners to "flag" someone like Eve, because she could simply transfer the money to a different account. Even if it was feasible to trace back every party's coins to make sure none of them traced back to a flagged account, Eve

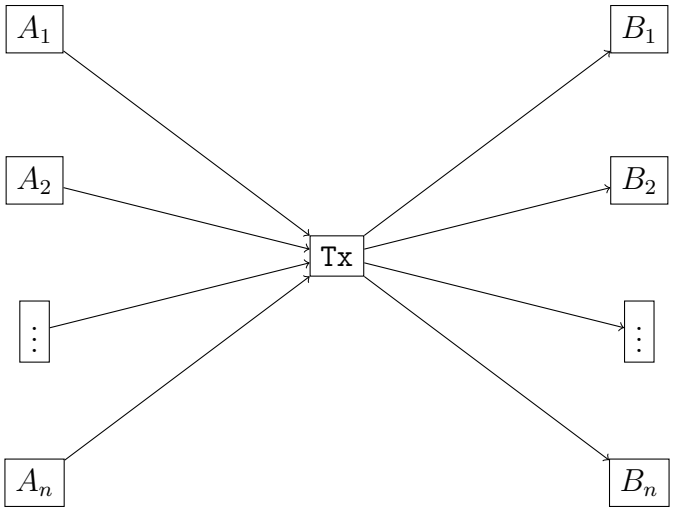


Figure 5.4: CoinJoin

could simply use a central mixing service or a Bitcoin exchange. This is decidedly off-topic, but if Eve works for a mixing service (which makes money off its clients), it would be in her benefit to persistently cripple a decentralized mixing protocol like CoinJoin.

### 5.5.4 Other

There are many proposed solutions to the problem of Bitcoin mixing, and here we summarize the current state of research on this topic:

- As early as 2012, the problem of anonymity was already apparent, and Barber et al. suggested an idea called a "fair-exchange protocol" between two users who wish to mix coins [5]. The problem is the same as that of CoinJoin itself: denial-of-service attacks. This paper also serves as a decent introduction to the fuzzy edges of Bitcoin: it addresses the financial problems, the crypto problems, the consensus problems, and the anonymity problems.

- In 2014, Ben-Sasson et al. suggested Zerocash [7], which is probably the best solution to cryptocurrency privacy not compatible with Bitcoin. The privacy method described with Zerocash is based on users offering zero-knowledge proofs to spend coins instead

of actual signatures. The drawback is that Zerocash drastically changes the way transactions, blocks, and verifications are built and handled, meaning that a brand new cryptocurrency would have to be implemented.

- In 2014, Bonneau et al. proposed Mixcoin [10], which features a central mix server with strong incentive *not* to steal the users' coins. This is an improvement in the sense that current mixes only have their reputation to lose.

- In 2014, Kate et al. introduced CoinShuffle [29], a new approach to shuffling output addresses which is still subject to similar low-cost DoS attacks as CoinJoin.

- In 2014, Bissias et al. published a paper describing Xim [8], a method which specifically targeted the DoS-attack problem of CoinJoin. Using Xim, users mix coins two at a time, finding mix partners based on ads placed in the block chain.

- In 2015, Grossman et al. developed Coinparty [24], a novel method that transfers trust away from a central service to a $t$-out-of-$n$ style account. Thus, malicious parties must represent at least a given threshold stake in the mix in order to subvert the transaction.

- In 2015, Rowan and Valenta published Blindcoin [44], which addressed the problem of mix servers seeing the link from a user's input address to their output address. The protocol use a blind signature scheme, as well as an append-only public log. The same year, ShenTu and Yu also published a blind signature mixing scheme, based on ECDSA [45].

## 5.6 Contribution

Our first contribution is a meet-up method using coin freeze. The goal: eliminate cheap DoS attacks.

Our second contribution is a slight alteration to the transaction structure to allow for a persistent public and decentralized mix.

### 5.6.1 Phase 1: Meet-and-Freeze

Alice and Bob are looking for partners to mix their coins. For the sake of example, suppose they both want to mix exactly 1 BTC. Alice sends to Bob a half-signed transaction with the following properties:

- Two inputs: 1 BTC from Alice and 1 BTC from Bob (plus transaction fees)
- Two outputs: 1 BTC each with the following locking script
- Encumbrance: for a given period of time, output $A$ must be signed by both Alice and Bob, and thereafter can only be signed for by Alice. Similarly, for the same amount of time, output $B$ must be signed by both Alice and Bob, and thereafter can only be signed for by Bob.

Table 5.3 shows the actual locking script. Note about the `IF` operator: if the value on the top of the stack is 1, the statements following `IF` are executed, and the top value is removed. For a full list of operators available in the script language, see Appendix A.

So, the locking script in Table 5.3 can be unlocked with the following unlocking script:

```
<sig1> <sig2> 0,
```

or after the expiry time:

```
<sig1/sig2> 1.
```

```
IF
    <expiry time> CHECKLOCKTIMEVERIFY DROP <PubK1/PubK2> CHECKSIG
ELSE
    2 <PubK1> <PubK2> 2 CHECKMULTISIG
```

Table 5.3: Two Person Coin-Freeze Locking Script

The idea is that pairs of two meet and becomes groups of four. Groups can grow larger

and asymmetrically (four meets two and so on), until an agreed-upon threshold is reached, and they proceed to Phase 2.

### 5.6.2 Phase 2: Coin Melt

At this point, there are $n$ users in contact, who have each locked up 1 BTC in a mutually shared encumbrance. Phase 2 is the mixing phase. Though since all users are agreeing to lift the freeze encumbrance, perhaps it would be better described as the "melting" phase.

The users generate fresh public keys, and exchange these keys anonymously with each other (See Section 5.6.4). They must then agree to sign a transaction with the following properties:

- $n$ inputs: 1 BTC from each user
- One output: $n$ BTC in total
- Encumbrance (two conditions)
  - Condition 1. To spend the money freely, everyone must sign (with their freshly generated private keys)
  - Condition 2 - a single user may sign for the money, but the output is limited to 1 BTC per public key.

This "Coin Melt" transaction is our second contribution.

We will first show the script for the Coin Melt transaction, discuss the new language operators, and then discuss some alternatives that illustrate the advantages of our construction.

The Coin Melt transaction locking script is shown in Table 5.4.

```
//Coin Melt (1 output)
IF
    1 <PubK1> <PubK2> ... <PubKn> n CHECKMULTISIG
    2 CHECKOUTCOUNTER
    <hash> CHECKOUTHASH
    <value> CHECKOUTVALUE
ELSE
    n <PubK1> <PubK2> ... <PubKn> n CHECKMULTISIG
ENDIF
```

Table 5.4: Coin Melt Locking Script

**CHECKOUTCOUNTER**

The CHECKOUTCOUNTER compares the item on the top of the stack to the number of outputs in the spending transaction. If they are equal, remove the top stack item and continue. Otherwise, mark the transaction as invalid.

**CHECKOUTHASH**

The CHECKOUTHASH compares the item on the top of the stack to the hash of the XOR of the output addresses (in all of the outputs). Keep in mind that addresses are hashes of public keys. If they are equal, remove the top stack item and continue. Otherwise, mark the transaction as invalid.

**CHECKOUTVALUE**

The CHECKOUTVALUE compares the item on the top of the stack $n$ to the output values. If the output values are each equal to (# of public keys) $\times n$, remove the top stack item and continue. Otherwise, mark the transaction as invalid.

A simple CoinJoin locking script is shown in Table 5.5. As we mentioned, the major disadvantage to CoinJoin is DoS attacks. However, our Muskteer transaction has another advantage: there remains the possibility for the users to assimilate more users and continue

mixing. In Table 5.5, mixing is over. To mix again, users must step through the meet process all over again.

```
//CoinJoin (n outputs)
<PubK1/PubK2/.../PubKn> CHECKSIG
```

Table 5.5: CoinJoin Locking Script

So, we might ask: can we make a simple adjustment to the CoinJoin locking script to allow for persistent mixing? One option would be simply to extend the conditions to allow for multi-signatures. This is shown in Table 5.6 (for the sake of example, we name it CoinJoin++).

For each output, either one user can sign, or all users must sign. However, when one user, say the user with `PubK1` spends one output, we would like to guarantee that he is no longer needed to sign for the persistent mix (using the alternate multi-signature condition). Unfortunately, his name is still on the contract, so this will not work either.

```
//CoinJoin++ (n outputs)
IF
    1 <PubK1/PubK2/.../PubKn> n CHECKMULTISIG
ELSE
    n <PubK1> <PubK2> ... <PubKn> n CHECKMULTISIG
ENDIF
```

Table 5.6: CoinJoin++ Locking Script

These problems are what gave way to the Coin Melt transaction. Using a restraint on the output, we guarantee that users leaving the mix have no further affect on the continued mixing.

### 5.6.3 Persistence

Each of the users that remain attached to the Coin Melt transaction can either spend their mixed coins individually, or elect altogether to continue mixing. The advantage is that there is no need to go through Phase 1 again. The remaining users meet with new users who have completed Phase 1, and together they can increase the mixing pool by executing Phase 2 again. Note that *all* users should again generate fresh public keys for this phase.

Users can spend their share of the transaction at any time, but the locking script ensures that they will pay the correct amount back to the other users.

### 5.6.4 Key Exchange

We give three simple options for $n$ honest users to exchange information anonymously. Keep in mind that the goal here is for $n$ users to meet up with known public addresses $A_1, \ldots, A_n$, then exchange $n$ freshly generated public addresses $B_1, \ldots, B_n$, such that no one can link the $A_i$ to any of the $B_i$.

#### Mix Network

In this context, a *mix* is a server that collects encrypted messages from multiple senders, permutes the order, and forwards them on to their next destination. A *mix network* simply describes a web of mixes. Chaining messages from mix node to mix node provides better and better security in obscuring the link between the sender and the final destination.

Suppose Alice wants to send an anonymous message to Bob using a mix. Bob has a public key $K_B$, and the mix has a public key $K_M$. Alice encrypts her message $m$ using $K_B$ (inner envelope), then appends Bob's e-mail address, and encrypts again using $K_M$ (outer envelope). Alice then sends the ciphertext to the mix. The mix server decrypts the outer envelope to find Bob's address, then forwards the inner envelope to Bob, who can then decrypt to find the original message. If she wishes, Alice can include a return address in the message so that Bob can reply, with the added benefit that the mix cannot see the return

address. This method was first described by David Chaum in [13].

There *are* mix networks known as *anonymous remailers* in existence, but they are not used as much as Tor (next section).

### Onion Routing

*Onion routing* is similar to a mix network. Here, a user sends a message through a chain of nodes, each of which can only see the location immediately previous and the destination. So, if one entity controls the entire network, onion routing is useless.

For Alice to send a message $m$ to Bob (public key $B$) through nodes $X$, $Y$, and $Z$, she encrypts as follows:

$$K_X(K_Y(K_Z(K_B(m), B), Z), Y)$$

Node $X$ can "peel" away the outer layer, discover the next address $Y$, and send the remaining unpeeled data on to the next destination.

Tor (The Onion Router), is a popular network for anonymous browsing on the internet that employs onion routing. In comparison to mix networks or anonymous remailers, Tor is probably the best practical option for Bitcoin users.

### Chaum Blind Signature

Blind signature schemes were introduced by David Chaum in [12]. Suppose Alice wants to obtain a signature from Bob for a message $m$, without letting him know the content of the message. Alice blinds the message to form $m'$, and sends $m'$ to Bob. Bob signs $m'$, and sends the signature back to Alice. Alice unblinds the message, and (provided the signature scheme is correctly constructed) the signature is valid for $m$. We leave out the details of specific blind signature schemes, but refer the reader to [12] and [6].

For our purposes of exchanging output addresses secretly, we could have a server play the role of signatory. Users connect, submit blinded output addresses for signing, and then

disconnect. They reconnect anonymously and provide the unblinded (signed) output address which the server accepts because it can verify its own signature, but cannot link the output address with any one of the users. This is an important part of BlindCoin [44], although their method specifically involves using the server to mix the coins. Here, we just need a method to establish private communication, but ultimately the users take care of mixing their own coins.

On the other hand, we could also have each of the users function as a signatory. Each user collects blinded signatures on his own output address from every other user. They then reconnect anonymously and exchange unblinded output addresses. Each can privately verify their own signature among the $n^2$ signatures, and so they have successfully exchanged output addresses anonymously.

### 5.6.5 Analysis of DoS Attacks

There are approximately 500,000 users on the Bitcoin network at any given time. Suppose, during Phase 1, users try to meet up with a group of around 20. Accordingly, there could be 250,000 groups preparing to mix at the same time. If each user wants to mix 1 BTC, then in order to enact a DoS attack, a single user would need to have a presence in all 250,000 groups and have 1 BTC per group as a false pledge. This would mean a single user would need 250,000 BTC (or 100 million USD) at immediate disposal to deny mixing to all users.

Furthermore, since interrupting a mix attempt locks up the attacker's money as well, to persistently block users during the "frozen" period, they would need even more money (one could imagine a single honest user trying multiple times to mix different bitcoins during the same period).

Certainly, an entity with enough time and money would be able to do this, but it is a sizable difference to the simplistic CoinJoin DoS attack. Recall: a single user with a fixed amount of money could simultaneously and persistently enact DoS attacks on all users attempting to employ CoinJoin.

## 5.7 Summary

It currently seems like the best way to allow for mixing of digital money would be a transition to brand new cryptocurrency, like Zerocash. The drawbacks are the increased overhead and the colossal transition to the new system.

Working within the confines of the current Bitcoin system, we have developed a method for enhancing a simple protocol like CoinJoin to reduce DoS attacks, with the assumption that an adversary has limited funds. We have also proposed new logic for the Bitcoin script to simplify this process, and promote easier and persistent mixing transactions.

# Chapter 6

# Conclusion and Final Remarks

The goal of this thesis was to approach three problems related to cryptographic hash functions in unique and innovative ways.

For the problem of creating a trapdoor hash function, our solution used elliptic curves, and a major result by Edlyn Teske.

We did not directly attack time-lock encryption. Instead, the problem we addressed was: how to allow a single trusted entity to authorize intermediaries the power of issuing time-lock encryption keys, while retaining a trusted and persistently verifiable link between them. One solution is with the use of RSA exponents, but we wanted to find a way to avoid using exponentiation.

Finally, after thoroughly researching the internal structure of the Bitcoin protocol, we found one area that merited improvement, namely: transaction privacy. We provide a method for mixing bitcoins using contracts.

# Bibliography


[1] Ange Albertini, Jean-Philippe Aumasson, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Malicious Hashing: Eve's Variant of SHA-1, 2014.

[2] Andreas M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Crypto-Currencies*. O'Reilly Media, Inc., 1st edition, 2014.

[3] Jean-Philippe Aumasson. Eve's SHA3 Candidate: Malicious Hashing, 2012.

[4] Lear Bahack. Theoretical bitcoin attacks with less than half of the computational power (draft). Cryptology ePrint Archive, Report 2013/868, 2013. `http://eprint.iacr.org/`.

[5] Simon Barber, Xavier Boyen, Elaine Shi, and Ersin Uzun. Bitter to Better: How to Make Bitcoin a Better Currency. In *Financial Cryptography and Data Security, Lecture Notes in Computer Science*, pages 399–414, 2012.

[6] Mihir Bellare, Chanathip Namprempre, David Pointcheval, and M. Semanko. The One-More-RSA-Inversion Problems and the Security of Chaum's Blind Signature Scheme. Cryptology ePrint Archive, Report 2001/002, 2001. `http://eprint.iacr.org/`.

[7] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459 – 474. IEEE, 2014.

[8] George Bissias, Brian N. Levine, Marc Liberatore, and A. Pinar Ozisik. Sybil-Resistant Mixing for Bitcoin. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 149–158. ACM, 2014.

[9] Nir Bitansky, Shafi Goldwasser, Abhishek Jain, Omer Paneth, Vinod Vaikuntanathan, and Brent Waters. Time-Lock Puzzles from Randomized Encodings. Cryptology ePrint Archive, Report 2015/514, 2015. `http://eprint.iacr.org/`.

[10] Joseph Bonneau, Arvind Narayanan, Andrew Miller, Jeremy Clark, Joshua A. Kroll, and Edward W. Felten. Mixcoin: Anonymity for Bitcoin with Accountable Mixes. Cryptology ePrint Archive, Report 2014/077, 2014. `http://eprint.iacr.org/`.

[11] Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, Arjen K. Lenstra, and Peter L. Montgomery. On the security of 1024-bit rsa and 160-bit elliptic curve cryptography. Cryptology ePrint Archive, Report 2009/389, 2009. `http://eprint.iacr.org/`.

[12] David Chaum. Blind Signatures for Untraceable Payments. In *Advances in Cryptology Proceedings of Crypto 82*, pages 199–203, 1983.

[13] David L. Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Commun. ACM*, 24(2):84–90, February 1981.

[14] Jeremy Clark and Urs Hengartner. On the Use of Financial Data as a Random Beacon. Cryptology ePrint Archive, Report 2010/361, 2010. `http://eprint.iacr.org/`.

[15] Martin Cochran. Notes on the Wang et al. $2^{63}$ SHA-1 Differential Path. Cryptology ePrint Archive, Report 2007/474, 2007.

[16] Scott Contini, Arjen K. Lenstra, and Ron Steinfeld. VSH, an Efficient and Provable Collision Resistant Hash Function, 2005.

[17] Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics and Chip Design*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

[18] Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. Witness Encryption and its Applications. Cryptology ePrint Archive, Report 2013/258, 2013. `http://eprint.iacr.org/`.

[19] Pierrick Gaudry, Florian Hess, and Nigel Smart. Constructive and Destructive Facets of Weil Descent on Elliptic Curves, 2000.

[20] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security. Cryptology ePrint Archive, Report 2016/013, 2016, 2016. `http:\\eprint.iacr.org/`.

[21] Sharon Goldberg, Ethan Heilman, Alison Kendler, and Aviv Zohar. Eclipse Attacks on Bitcoin's Peer-to-Peer Network. Cryptology ePrint Archive, Report 2015/263, 2015. `http://eprint.iacr.org/`.

[22] Dan Goodin. Anatomy of a hack: How crackers ransack passwords like qeadzcwrsfxv1331. `http://arstechnica.com/security/2013/05/how-crackers-make-minced-meat-out-of-your-passwords/`, 2013.

[23] Vipul Goyal. How to Re-initialize a Hash Chain. Cryptology ePrint Archive, Report 2004/097, 2004. `http://eprint.iacr.org/`.

[24] Fred Grossman, Martin Henze, Nicolas Inden, Klaus Wehrle, and Jan Henrik Ziegeldorf. CoinParty: Secure Multi-Party Mixing of Bitcoins. In *CODASPY 2015 Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 75–86, 2015.

[25] Gus Gutoski and Douglas Stebila. Hierarchical Deterministic Bitcoin Wallets that Tolerate Key Leakage. Cryptology ePrint Archive, Report 2014/998, 2014. `http://eprint.iacr.org/`.

[26] Michael Jacobson, Alfred Menezes, and Andreas Stein. Solving Elliptic Curve Discrete Logarithm Problems Using Weil Descent, 2001.

[27] Tibor Jager. How to Build Time-Lock Encryption. Cryptology ePrint Archive, Report 2015/478, 2015. `http://eprint.iacr.org/`.

[28] Ghassan O. Karame, Elli Androulaki, and Srdjan Capkun. Two bitcoins at the price of one? double-spending attacks on fast payments in bitcoin. Cryptology ePrint Archive, Report 2012/248, 2012. `http://eprint.iacr.org/`.

[29] Aniket Kate, Pedro Moreno-Sanchez, and Tim Ruffing. CoinShuffle: Practical Decentralized Coin Mixing for Bitcoin. *Computer Security - ESORICS*, 8713:345 – 364, 2014.

[30] Jonathan Katz, Andrew Miller, and Elaine Shi. Pseudonymous Secure Computation from Time-Lock Puzzles, 2014.

[31] Hugo Krawczyk and Tal Rabin. Chameleon Hashing and Signatures, 1997.

[32] Arjen K. Lenstra, Daniel Page, and Martijn Stam. Discrete Logarithm Variants of VSH, 2006.

[33] Jia Liu, Flavio Garcia, and Mark Ryan. Time-release Protocol from Bitcoin and Witness Encryption for SAT. Cryptology ePrint Archive, Report 2015/482, 2015. `http://eprint.iacr.org/`.

[34] Mohammad Mahmoody, Tal Moran, and Salil Vadhan. Time-Lock Puzzles in the Random Oracle Model. In *Of Lecture Notes in Computer Science*, pages 39–50. Springer, 2011.

[35] Alfred Menezes and Minghua Qu. Analysis of the Weil Descent Attack of Gaudry, Hess and Smart, 2000.

[36] Michael Mitzenmacher and Salil Vadhan. Why Simple Hash Functions Work: Exploiting the Entropy in a Data Stream. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '08, pages 746–755, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

[37] Payman Mohassel. One-time Signatures and Chameleon Hash Functions, 2010.

[38] Malte Möser, Rainer Möhme, and Dominic Breuker. An Inquiry into Money Laundering Tools in the Bitcoin Ecosystem, 2013.

[39] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. `https://bitcoin.org/bitcoin.pdf/`.

[40] NIST. Recommended Elliptic Curves For Federal Government Use. `http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf`, 1999.

[41] Nicole Perlroth, Jeff Larson, and Scott Shane. N.S.A. Able to Foil Basic Safeguards of Privacy on Web. *The New York Times*, 2013.

[42] Jothi Rangasamy, Douglas Stebila, Lakshmi Kuppusamy, Colin Boyd, and Juan Gonzalez Nieto. Efficient Modular Exponentiation-based Puzzles for Denial-of-Service Protection. Cryptology ePrint Archive, Report 2011/665, 2011. http://eprint.iacr.org/.

[43] Ronald L. Rivest, Adi Shamir, and David A. Wagner. Time-Lock Puzzles and Timed-Release Crypto, 1996.

[44] Brendan Rowan and Luke Valenta. BlindCoin: Blinded, Accountable Mixes for Bitcoin. In *Financial Cryptography and Data Security, Lecture Notes in Computer Science*, pages 112–126, 2015.

[45] QingChun ShenTu and Jianping Yu. A Blind-Mixing Scheme for Bitcoin based on an Elliptic Curve Cryptography Blind Digital Signature Algorithm. *CoRR*, abs/1510.05833, 2015.

[46] Dan Shumow and Niels Ferguson. On the Possibility of a Back Door in the NIST SP800-90 Dual EC PRNG. http://rump2007.cr.yp.to/15-shumow.pdf, 2007.

[47] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*. Graduate texts in mathematics. Springer, 1986.

[48] Joseph H. Silverman. *Advanced Topics in the Arithmetic of Elliptic Curves*. Graduate texts in mathematics. Springer, 1994.

[49] Douglas Stinson. *Cryptography: Theory and Practice*. CRC/C&H, 3rd edition, 2006.

[50] Edlyn Teske. An Elliptic Curve Trapdoor System, 2006.

[51] Xiaoyun Wang, YiqunLisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1, 2005.

[52] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions, 2005.

[53] Lawrence C. Washington. *Elliptic Curves: Number Theory and Cryptography*. Chapman & Hall/CRC, 2003.

[54] Santiago Zanella-Beguelin and Microsoft Vulnerability Research. Socat Security Advisory 7. http://www.dest-unreach.org/socat/contrib/socat-secadv7.html, 2016.

[55] Yuanchao Zhao and Daoben Li. An Improved Elegant Method to Re-initialize Hash Chains. Cryptology ePrint Archive, Report 2005/011, 2005. http://eprint.iacr.org/.

# Appendix A

# Script

The Bitcoin transaction script language, *Script*, is a simple, stack-based language.

Script contains many operators, but is deliberately limited in one important way: there are no loops or complex flow control capabilities other than conditional flow control. This ensures that the language is not *Turing Complete*, meaning that scripts have limited complexity and predictable execution times. Script is not a general-purpose language. These limitations ensure that the language cannot be used to create an infinite loop or other form of "logic bomb" that could be embedded in a transaction in a way that causes a denial-of-service attack against the Bitcoin network. A limited language prevents the transaction validation mechanism from being used as a vulnerability [2].

Figures A.1 through A.7 contain lists of operations for Script. Since Script is a stack-based language, many operations involve pushing, popping, or moving items around in the stack data structure. Some of the arithmetic operations require one (or more inputs), and we refer to those as $a$ and $b$ when necessary.

| Word | Description |
|---|---|
| OP_0 | Empty array |
| OP_PUSHDATA1 | The next one byte contains the number of bytes to be pushed onto the stack (OP_PUSHDATA2 and OP_PUSHDATA4 are defined similarly for two and four bytes) |
| OP_1NEGATE | -1 |
| OP_1 | 1 |
| OP_N | N (between 2 and 16) |

Figure A.1: Constants

| Word | Description |
|---|---|
| OP_NOP | Do nothing |
| OP_IF | Execute the following statements if the top stack value is not 0 |
| OP_NOTIF | Execute the following statements if the top stack value is 0 |
| OP_ELSE | The following statements are executed if and only if the proceeding IF, NOTIF, or ELSE was not executed |
| OP_ENDIF | Ends an IF/ELSE block. All blocks must end, or the transaction is invalid |
| OP_VERIFY | Marks the transaction as invalid if the top stack value is not true |
| OP_RETURN | Marks transaction as invalid |

Figure A.2: Flow Control Operators

| Word | Description |
|---|---|
| OP_TOALTSTACK | Puts the input onto the top of the alt stack. Removes it from the main stack |
| OP_FROMALTSTACK | Puts the input onto the top of the main stack. Removes it from the alt stack |
| OP_IFDUP | If the top stack value is not 0, duplicate it |
| OP_DEPTH | Puts the number of stack items onto the stack |
| OP_DROP | Removes the top stack item |
| OP_DUP | Duplicates the top stack item |
| OP_NIP | Removes the second-to-top stack item |
| OP_OVER | Copies the second-to-top stack item to the top |
| OP_PICK | The item $n$ back in the stack is copied to the top |
| OP_ROLL | The item $n$ back in the stack is moved to the top |
| OP_ROT | The top three items on the stack are rotated to the left |
| OP_SWAP | The top two items on the stack are swapped |
| OP_TUCK | The item at the top of the stack is copied and inserted before the second-to-top item |
| OP_2DROP | Removes the top two stack items |
| OP_2DUP | Duplicates the top two stack items |
| OP_3DUP | Duplicates the top three stack items |
| OP_2OVER | Copies the pair of items two spaces back in the stack to the front |
| OP_2ROT | The fifth and sixth items back are moved to the top of the stack |
| OP_2SWAP | The top two pairs of items are swapped |

Figure A.3: Stack Operators

| Word | Description |
|---|---|
| OP_EQUAL | Returns 1 if the inputs are exactly equal, 0 otherwise |
| OP_EQUALVERIFY | Same as OP_EQUAL, but execute OP_VERIFY afterward |

Figure A.4: Bit Logic Operators

| Word | Description |
|---|---|
| OP_1ADD | 1 is added to the input |
| OP_1SUB | 1 is subtracted from the input |
| OP_NEGATE | The sign of the input is flipped |
| OP_ABS | The input is made positive |
| OP_NOT | If the input is 0 or 1, it is flipped. Otherwise, the output is 0 |
| OP_0NOTEQUAL | Returns 0 if the input is 0, and 1 otherwise |
| OP_ADD | $b$ is added to $a$ |
| OP_SUB | $b$ is subtracted from $a$ |
| OP_BOOLAND | If $a$ and $b$ are not 0, the output is 1, and otherwise 0 |
| OP_BOOLOR | If $a$ or $b$ is not 0, the output is 1, and otherwise 0 |
| OP_NUMEQUAL | Returns 1 if $a = b$, and otherwise 0 |
| OP_NUMEQUALVERIFY | Same as OP_NUMEQUAL, but executes OP_VERIFY afterward |
| OP_NUMNOTEQUAL | Returns 1 if $a \neq b$, and otherwise 0 |
| OP_LESSTHAN | Returns 1 if $a < b$, and otherwise 0 |
| OP_GREATERTHAN | Returns 1 if $a > b$, and otherwise 0 |
| OP_LESSTHANOREQUAL | Self explanatory |
| OP_GREATERTHANOREQUAL | Self explanatory |
| OP_MIN | Returns the smaller of $a$ and $b$ |
| OP_MAX | Returns the larger of $a$ and $b$ |
| OP_WITHIN | Returns 1 if an input is within a specified range (left-inclusive), and 0 otherwise |

Figure A.5: Arithmetic Operators

| Word | Description |
|---|---|
| OP_RIPEMD160 | The input is hashed using RIPEMD-160 |
| OP_SHA1 | The input is hashed using SHA1 |
| OP_SHA256 | The input is hashed using SHA256 |
| OP_HASH160 | The input is hashed twice: first with SHA256, then with RIPEMD160 |
| OP_HASH256 | The input is hashed twice with SHA256 |
| OP_CHECKSIG | The entire transaction's inputs, outputs, and script are hashed. The signature used must be a valid signature for the hash and public key. If it is, 1 is returned, and otherwise 0. |
| OP_CHECKSIGVERIFY | Same as OP_CHECKSIG, but OP_VERIFY is executed afterward |
| OP_CHECKMULTISIG | Checks m signatures against n public keys |
| OP_CHECKMULTISIGVERIFY | Same as OP_CHECKMULTISIG, but OP_VERIFY is executed afterward |

Figure A.6: Crypto Operators

| Word | Description |
|---|---|
| OP_CHECKLOCKTIMEVERIFY | Marks transaction as invalid if the top stack item is greater than the transaction's LockTime field, otherwise script evaluation continues as though an OP_NOP was executed. Transaction is also invalid if 1. the stack is empty; or 2. the top stack item is negative; or 3. the top stack item is greater than or equal to 500000000 while the transaction's nLockTime field is less than 500000000, or vice versa |

Figure A.7: Locktime Operators