INCREMENTAL HASH FUNCTIONS

A THESIS

SUBMITTED TO THE DEPARTMENT OF MATHEMATICS

AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Emrah Karagöz

June 2014

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Hamza YEŞİLYURT(Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Ahmet Muhtar GÜLOĞLU

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Ali Aydın SELÇUK

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

INCREMENTAL HASH FUNCTIONS

Emrah Karagöz
M.S. in Mathematics
Supervisor: Asst. Prof. Dr. Hamza YEŞİLYURT
June 2014

Hash functions are one of the most important cryptographic primitives. They map an input of arbitrary finite length to a value of fixed length by compressing the input, that is why, they are called *hash*. They must run efficiently and satisfy some cryptographic security arguments. They are mostly used for data integrity and authentication such as digital signatures.

Some hash functions such as SHA family (`SHA1-SHA2`) and MD family (`MD2-MD4-MD5`) are standardized to be used in cryptographic schemes. A common property about their construction is that they are all iterative. This property may cause an efficiency problem on big size data, because they have to run on the entire input even it is slightly changed. So the question is "Is it possible to reduce the computational costs of hash functions when small modifications are done on data?"

In 1995, Bellare, Goldreich and Goldwasser proposed a new concept called **incrementality**: a function f is said to be incremental if $f(x)$ can be updated in time proportional to the amount of modification on the input x. It brings out two main advantages on efficiency: incrementality and parallelizability. Moreover, it gives a provable security depending on hard problems such as discrete logarithm problem (DLP). The hash functions using incrementality are called *Incremental Hash Functions*. Moreover, in 2008, Dan Brown proposed an incremental hash function called ECOH by using elliptic curves, where DLP is especially harder on elliptic curves, and which are therefore quite popular mathematical objects in cryptography.

We state incremental hash functions with some examples, especially ECOH , and give their security proofs depending on hard problems.

Keywords: Incremental Hash Functions, MuHASH, AdHASH , ECOH .

ÖZET

ARTIMLI ÖZET FONKSİYONLARI

Emrah Karagöz
Matematik, Yüksek Lisans
Tez Yöneticisi: Yrd. Doç. Dr. Hamza YEŞİLYURT
Haziran 2014

Özet Fonksiyonları daha çok veri bütünlüğünde ve elektronik imza gibi kimlik doğrulamada kullanılan kriptolojinin en önemli araçlardan biridir. Bu fonksiyonlar, herhangi bir uzunluktaki girdiyi belli özelliklerle sıkıştırarak sabit bir uzunluktaki çıktıya götürler. Bu fonksiyonlar ayrıca hızlı hesaplanabilir olmalı ve bazı kriptografik güvenlik gereksinimlerini sağlamalıdırlar.

SHA ve MD gibi özet fonksiyonları aileleri, kriptolojik uygulamalarda kullanılmak üzere standartlaştırılmış özet fonksiyonlarıdır. Bunların en önemli yapısal özelliği yinelemeli (*iterative*) olmalarıdır. Bu özellik büyük boyutlu verilerde verimlilik problemine neden olur, çünkü girdide ufak bir değişiklik olsa bile özet fonksiyonu bütün girdi üzerinde tekrar çalışmalıdır. Bu yüzden sormamız gereken soru "Veri üzerindeki ufak bir değişiklik olduğunda özet fonksiyonun hesaplama maliyetini düşürmek mümkün müdür?" olacaktır.

1995 yılında Bellare, Goldriech ve Goldwasser tarafından *artımlılık* isimli yeni bir konsept sunulmuştur: eğer x girdisi üzerinde yapılan küçük bir değişiklik sonucu $f(x)$ değeri değişiklikle doğru orantılı olacak bir zamanda güncellenebiliyorsa f fonksiyonuna artımlı denir. Bu konsept verimlilik açısından çok önemli iki tane avantaj sağlamıştır: artımlılık ve paralelleştirilebilirlik. Ayrıca ayrık logaritma problemi gibi çözülmesi zor olan problemlere dayalı bir güvenlik sağlamıştır. Bu özelliği kullanan özet fonksiyonlarına *Artımlı Özet Fonksiyonları* diyoruz. Ayrıca, Dan Brown 2008 yılında artımlı özet fonksiyonlarına örnek olacak Eliptik Eğrilerde Özet Fonksiyonu (ECOH) adlı özet fonksiyonunu önermiştir.

Bu tezde, artımlı özet fonksiyonları örnekleriyle (özellikle de ECOH örneğiyle) birlikte incelenmiş ve bunların güvenlik ispatları çözülebilirliği zor problemlerle karşılaştırılırak gösterilmiştir.

Anahtar sözcükler: Artımlı Özet Fonksiyonları, MuHASH, AdHASH , ECOH .

Acknowledgement

Although all the work put out for this thesis is presented in following chapters, the soul of this thesis and the people created this soul are presented here. Therefore, writing a good acknowledgement for these people deserves more attention, but, it is more enjoyable than writing all the thesis.

I start by thanking my former advisor Asst. Prof. Dr. Koray Karabina, I am deeply indebted to him. He inspired the idea of this thesis, encouraged me and supported my studies. I will always be remembering our meetings such as our after-midnight study when we came together in the end of an exhausted day in CryptoDays conference in Gebze. I will always admire his attention to his students, and his fidgety moves when he gives a lecture or a presentation.

I secondly thank my advisor Asst. Prof. Dr. Hamza Yeşilyurt. He accepted being my advisor after Koray Karabina had left Bilkent University. He read and checked the errors in my thesis again and again. He spent his plenty time for me, in spite of the fact that, he could spend it to his newborn baby.

I also thank Asst. Dr. Ahmet Muhtar Güloğlu and Prof. Dr. Ali Aydın Selçuk. They accepted to be in the jury of my thesis defense. Moreover, Prof. Dr. Ali Aydın Selçuk lectured the cryptography course in Bilkent University and taught me the practical cryptography by giving a lot of projects and homeworks.

I would like to acknowledge to people who involves in my cryptography career by starting from Boğaziçi University where I got my BS degree in Maths. I start by thanking Prof. Dr. Yılmaz Akyıldız, who lectured my first cryptography course in the university. After this lecture, I found this area interesting and I decided to study in cryptography. Moreover, he wrote many reference letters for me, even after he had retired from the university. I also thank my ex-advisor in the university, Ferit Öztürk, who suggested me to stop studying mathematics, to forget an academic life, and to find a job and work in a different area. I am sorry, I could not keep this advice in mind, and I continued to pursue my dreams. It

seems that sometimes miracles can happen. I finally thank Müge Taşkın Aydın, whom I studied with in my last year in the university, and who did not write a reference letter by telling that she did not get to know me in this period, however, she had told before that she wanted to. After a year I graduated from the university, she saw me in a conference and asked whether I was still angry or not, but my answer was no, I was not, because I knew that it was not own her decision.

I contiue to acknowledge to Prof. Dr. İsmail Güloğlu and Prof. Dr. Mehpare Bilhan, who are doyen of Turkish mathematicians in this life, as far as I see. I had a chance to know Prof. Dr. İsmail Güloğlu in Doğuş University when I was working there as a teaching assistant. He encouraged me to go to Ankara and to learn cryptography there as fast as I can. He also taught algebra courses and made this field lovely to me. I know him as a person who is still desirous to learn and ambitious to teach like a young researcher, even if his age gets older. I met Prof. Dr. Mehpare Bilhan in METU IAM and she taught the course of finite fields and its applications. I know her as a person who standed out against her illness with her ambition to teach and her love to students. I believe she will recover soon. I will always respect to these two admirable people forever.

I would like to thank my dear teachers in Bilkent University and METU IAM, who taught many things in Maths and Cryptography: Mefharet Kocatepe, Ergün Yalçın, Müfit Sezer, Alexander Goncharov, Metin Gürses, Laurance Barker, Hakkı Turgay Kaptanoğlu, Salih Karadağ and especially Meltem Sağtürk from Bilkent University; Muhiddin Uğuz, Ersan Akyıldız and Ferruh Özbudak from METU IAM. Salih Karadağ, as we call him *Salih Başkan*, usually calls me "*Neşeli Çocuk*" (happy kid). I promise I will continue to look happy forever.

I will not forget to thank my dear friends : Erion Dula, Fatih Çiğci, Can Türkün, Hubeyb Gürdoğan, İsmail Özkaraca, Zeliha Ural, Merve Demirel, Yasemin Türedi, Emre Şen and Mehmet Kişioğlu from Bilkent University; Abdullah Öner, Bekir Danış, Recep Özkan, Elif Doğan, İsmail Alperen Öğüt, Oğuz Gezmiş, Burak Hatinoğlu again from Bilkent University, but we call them as "*Genç Subaylar*" (young soldiers) because their entrance to the university was

the next semester of ours; Mehmet Toker, Halil Kemal Taşkın, Murat Demircioğlu, Mustafa Şaylı, Sabahattin Çağ, Ahmet Sınak, Rumi Melih Pelen, Kamil Otal, and Pınar Çomak from METU. Erion and Fatih have also been my friends from Boğaziçi, and they shared the boring life of Ankara by making it full of action. Erion, Can and I also shared the same office room in Bilkent, and I thank them for keeping our castle from the known person. In addition, I thank very much to Abdullah, Oğuz and Burak who were my only supporters in thesis defense, while it was occurred in time of vacation.

I will also acknowledge to my dear colleagues from TÜBİTAK BİLGEM UEKAE: Hüseyin Demirci, Fatih Birinci, Şükran Külekçi, Mehmet Sabır Kiraz, Mehmet Karahan, Ziynet Nesibe Dayıoğlu, Dilek Çelik, Mehmet Emin Gönen, Oğuzhan Ersoy, and especially İsa Sertkaya and Birnur Ocaklı. İsa has given many many advices and shared his experiences about work, life and even in writing thesis, however, I could not take care about them enough, and this is why, he always gets angry to me. On the other hand, Birnur, the adorable lady, has been the real heroin behind the scene with her supportive words, her patience, and her struggles for motivating me to finish my thesis at once. Therefore, she deserves this special compliment, which is only for her: "*Thanks to Birnur*".

Finally, I would like to thank my parents who show an enormous patience with a great love to a son like me. They think I am studying for my PhD degree in Ankara, and they will continue to think like this for a while, but do not worry, one day your son will have his PhD degree and will thank you again in acknowledgment part in his PhD thesis. I also thank to my little sister with these words: *you always love me more in a moment than I could in a lifetime*.

By the way, I can not forget to thank TÜBİTAK who financially supported by through the graduate fellowship, namely "TÜBİTAK BİDEB 2210-Yurt İçi Yüksek Lisans Doğrudan Burs Programı". I am grateful to council for their kind support, and I believe that they will always continue to support young researchers.

This thesis is not a big success in my career, however, it is an award of small steps for my pursued big studies. Thus, I present this award to *my lonely and beautiful country, which I love passionately*, as Nuri Bilge Ceylan did in Cannes.

To all my friends,
who really wants my thesis to be finished at once.
I hope all you are happy now ☺

Contents

List of Figures

List of Tables

List of Symbols

$a\|\|b$	Concatenation of bitstrings a and b
$\{0,1\}^n$	Set of bitstrings of length n
$\{0,1\}^*$	Set of all bitstrings
$len(M)$	Bitlength of a bitstring M
$\mathtt{0}^k$	The bitstring $\mathtt{00\ldots0}$ of length k
$\wedge$	AND operation
$\vee$	OR operation
$\neg x$	Negation of x
$\lll n$	Cyclic left rotation by n
$\ggg n$	Cyclic right rotation by n
$\oplus$	Exclusive OR operation
$\boxplus$	Modular Addition Operation
$\mathbb{Z}$	The set of integers $\{\ldots,-2,-1,0,1,2,\ldots\}$
$\mathbb{Q}$	The set of rational numbers $\{\frac{a}{b} : a,b \in \mathbb{Z}, b \neq 0\}$
$\mathbb{R}$	The set of real numbers
$\mathbb{F}_q$	Finite field of q elements
$\lceil x \rceil$	The smallest integer greater than or equal to x
$\lfloor x \rfloor$	The largest integer less than or equal to x
$\bigcup_i A_i$	The union of the sets A_i
$\bigcap_i A_i$	The intersection of the sets A_i
$A - B$	The difference of the set A from the set B, i.e. $\{a : a \in A \text{ and } a \notin B\}$
$a \mid b$	The integer a divides the integer b
$a \nmid b$	The integer a divides the integer b

Chapter 1

Introduction

Information is an understood quantity. An identity of a person, a letter, a sentence, even a mathematical formula is an information. It is expressed in letters, numbers or symbols of a certain language. In daily lives, we use words to express the information, on the other hand, a mathematical formula is expressed with numbers and mathematical symbols. But in computer science, every information is seen as a combination of zeros and ones where these two numbers 0 and 1 are called as *bit* and their any combination is called as *bitstring*.

Not every information is public, some is intended to be known by only the people who have permission to know it. For example, the PIN code of someone's cell phone has to be known by the owner of the cell phone. The secret letters or messages sent among the allied countries has to be not seen by their enemies. Many protocols and mechanisms have been created to satisfy the security of the information. In formal definition according to [1], cryptography is the study of mathematical techniques related to aspects of information security. The four goals of cryptography are *confidentiality* (keeping the content of information from all but those authorized to have it), *data integrity* (addressing the unauthorized alteration of data), *authentication* (identification of entities and information itself), and *non-repudiation* (preventing an entity from denying previous commitments or actions). There are several tools in cryptography such as block/stream ciphers and digital signatures. One of the most important cryptographic primitives are

hash functions. They are mainly used to satisfy the goals of data integrity and authentication in cryptography.

A *cryptographic hash function* maps a bitstring of arbitrary length to a bitstring of fixed length. Hash functions takes a data of big size and gives its hash value of short fixed size. The main idea of hash functions is to represent the data in a compact form and use it as the identification of the data. The output of a hash function is called **hash value** or simply **hash**. The term *hash* is originated from compressing a message of big size to a small value of fixed length.

Hash functions are mainly used for data integrity: when an authorized entity receive a data whose hash value is known by himself, if an unauthorized entity has altered the original data, then the receiver can easily recognize whether it is changed or not by comparing its original hash value and the hash value he computed. In that sense, hash functions are used in digital signature schemes, where a message is hashed first, and then the hash value as a representative of the message, is signed instead of the original message using a public key encryption.

They are also used for authentication such as message authentication codes (MACs), passwords and passphrases. For example, mail services do not save the database of the passwords of their users, instead of this, they save only the hash values of passwords: when the user enters his password, the hash value is computed and sent to the mail server, then the mail server checks whether the value is equal to the saved value. In that sense, hash functions are used for comparing two values without revealing or storing them in clear. Other uses of hash functions are checksums of files, key generation procedures, and random number generators.

Hash functions are designed to be efficient in sense of running time of computation, and to satisfy some security properties in sense of cryptography: 1) *preimage resistance* - it is difficult to find a message for a given hash value , 2) *second preimage resistance* - it is difficult to modify a message without changing the hash value 3) *collision resistance* - it is difficult to find two different messages whose hash values are same. The security level of a hash function is determined by the computational difficulty of these properties.

Some hash functions are standardized to be used commonly in practice because of their security level. For example, National Standards Institution of USA (NIST) proposes hash functions SHA-1 and SHA-2 in the document FIPS 180-4 [2] to be supported by most of the cryptographical tools. Also the MD family (MD2[3], MD4[4] and MD5[5]) and RIPEMD family [6] of hash functions are mostly supported. In standard hash functions, the construction is mainly based on a compression function f. It starts with an initial hash value H_0, then computes the latter hash value H_{i+1} by using the message blocks M_i and the former hash values H_i until the last message block is processed, i.e. $f(H_i, M_i) = H_{i+1}$ for $i = 0, 1, \ldots, n$ where n is the number of blocks in the message. In that sense, it runs efficiently since it uses the same function, but it runs *iteratively*, in other words, the hash function has to process all message blocks again even if there is a small change on the message. This will result a main problem on big data.

In 1995, Bellare, Goldreich and Goldwasser proposed a new construction for hash functions called *Incremental Hash Functions* [1] in [8] where it is named by the concept of *incrementality*:

Definition 1. *Given a map f and inputs x and x' where x' is a small modification of x. Then f is said to be* ***incremental*** *if one can update $f(x)$ in time proportional to the amount of modification between x and x' rather than having to recompute $f(x')$ from scratch.*

Incremental hash functions can be efficiently used in practice where incrementality makes a big difference in sense of time of computation. This difference can be seen easily on some examples such as storing files online (today it is called as *cloud servers*), software updates and virus protection. These examples will be detailed in Chapter 3.

In 2008, Dan Brown proposed a practical example for incremental hash functions called ECOH (Elliptic Curve Only Hash) [9] to SHA-3 contest of NIST. His submission ECOH was constructed on elliptic curves which are quite popular in modern cryptography. This popularity of elliptic curves comes from the difficulty

[1] Their first incremental hash function is based on exponentiation in a group of prime order. Then their construction is improved by Bellare and Micciancio in [7].

level of discrete logarith problem in this mathematical object.

This thesis is organized as follows: in Chapter 3 , we state the construction of incremental hash functions with some examples. In Chapter 4, we then discuss their security proofs with comparing computationally hard problems. In Chapter 5, we give the practical example ECOH of incremental hash functions. Finally, in Chapter 6, the thesis is concluded with some remarks, open problems and future work.

Chapter 2

Preliminaries

We start with *Groups and Fields* to define mathematical objects used in incremental hash functions. We, then, continue with *Message Encoding and Parsing into Blocks* to represent messages in numerical, binary or hex representation via standard character tables, and to parse them equally into the blocks of fixed length. Finally, *Is it Easy or Hard?* is about the complexity theory to mention the levels of security arguments.

2.1 Groups and Fields

Some algebraic structures such as groups and finite fields especially are used for the construction of incremental hash functions. Therefore they are defined with some examples. Here we follow [10].

Definition 2 (Binary Operation)**.** *A* ***binary operation*** $\star$ *on a non-empty set* G *is a function* $\star : G \times G \to G$*. For any* $a, b \in G$*, this function is denoted by* $a \star b$*. The binary operation* $\star$ *is* ***associative*** *if the equality* $(a \star b) \star c = a \star (b \star c)$ *holds, and is* ***commutative*** *if the equality* $a \star b = b \star a$ *holds for any* $a, b, c \in G$*.*

Example 1. $+, \times$ *and* $-$ *(usual addition, multiplication and subtraction, respectively) are commutative binary operations on* $\mathbb{Z}$ *(or on* $\mathbb{Q}$*,* $\mathbb{R}$*,* $\mathbb{C}$ *respectively).*

However, $-$ is not a binary operation on $\mathbb{Z}^+$ since $2-5=-3\notin\mathbb{Z}^+$.

Definition 3. *Let G be a non-empty set and $\star$ be a binary operation on G. Then $(G,\star)$ is called a **group** if the following properties are satisfied:*

1. *$\star$ is associative, in other words, $(a\star b)\star c = a\star(b\star c)$ holds for every $a,b,c\in G$.*
2. *There exists an element $e\in G$, called an **identity** of G, such that $a\star e = e\star a = a$ for all $a\in G$.*
3. *For each $a\in G$, there is an element $a^{-1}\in G$, called **inverse** of a, such that $a\star a^{-1} = a^{-1}\star a = e$.*

*The group $(G,\star)$ is called **abelian** if $a\star b = b\star a$ for all $a,b\in G$, and is called **finite group** if it contains finitely many elements.*

Example 2. *$\mathbb{Z}$ $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ are groups under $+$ with $e=0$ and $a^{-1}=-a$. Also $\mathbb{Q}-\{0\}$, $\mathbb{R}-\{0\}$ and $\mathbb{C}-\{0\}$ are groups under $\times$ with $e=1$ and $a^{-1}=\dfrac{1}{a}$. However, $\mathbb{Z}-\{0\}$ is not a group since $\dfrac{1}{2}$, the inverse of $2\in\mathbb{Z}$, is not an integer.*

Definition 4. *Let K be a nonempty set and $+$ and $\times$ be two binary operations on K. Then K is called as **field** if the followings are satisfied:*

1. *$(K,+)$ is an abelian group,*
2. *$(K^\times,\times)$ is an abelian group where $K^\times = K-\{0\}$ and 0 is the identity element of $(K,+)$,*
3. *The distributive law of $\times$ over $+$ exists: for all $a,b,c\in K$,*

   $$a\times(b+c) = (a\times b)+(a\times c) \text{ and } (a+b)\times c = (a\times c)+(b\times c)$$

   holds.

In a field K, the identity element of $(K,+)$ is denoted by 0 and the identity element of $(K^\times,\times)$ is denoted by 1, where $1\neq 0$. The additive inverse of $a\in K$ is denoted by $-a$ and the multiplicative inverse of $a\in K^\times$ is denoted by a^{-1}, $1/a$ or $\frac{1}{a}$.

Example 3. *$\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, and $\mathbb{Z}_p$ for prime p are fields. However $\mathbb{Z}$ is not a field since $2 \in \mathbb{Z}^\times$ has no multiplicative inverse in $\mathbb{Z}^\times$. Also $\mathbb{Z}_6$ is not a field since $3 \in \mathbb{Z}_6$ has no multiplicative inverse in $\mathbb{Z}_6^\times$.*

Definition 5. *The **characteristic** of a field K, denoted by $char(K)$, is defined to be the smallest positive integer n such that*

$$\underbrace{1+1+\ldots+1}_{n} = 0$$

if it exists. Otherwise, it is defined to be 0.

Example 4. *The characteristic of $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$ is 0. The characteristic of $\mathbb{Z}_p$ for prime p is p.*

It is easy to show that the characteristic of a field K is always 0 or a prime p.

Definition 6. *A field that contains finitely many elements is called **finite field**. A finite field is denoted by $\mathbb{F}_q$ where q is the number of elements in the field.*

It can be shown that the characteristic of a finite field $\mathbb{F}_q$ is always a prime number p and the number of elements in $\mathbb{F}_q$ is a power of p, i.e. $q = p^n$ for some positive integer n. If $n = 1$, then $\mathbb{F}_p = \mathbb{Z}_p$

Definition 7. *Let $p(x)$ be a polynomial of degree $n \in \mathbb{Z}$ over a field K, i.e. $p(x) = a_0 + a_1x + \ldots + a_nx^n$ where $a_i \in K$ for $i = 1 \ldots n$. Then the polynomial $p(x)$ is **irreducible** over K if there exists no polynomials $q(x)$ and $r(x)$ over K of degrees greater than or equal to 1 such that $p(x) = q(x)r(x)$.*

Example 5. *The polynomial $p(x) = x^2 + 1$ is irreducible over $\mathbb{R}$ and $\mathbb{Z}_3$, however is not irreducible over $\mathbb{Z}_2$ since $x^2 + 1 = (x + 1)(x + 1)$.*

Now we conclude this section by giving the construction of finite fields. If $p(x)$ is an irreducible polynomial of degree $n \geq 2$ over finite field $\mathbb{F}_p$ and α be a root of $p(x)$, i.e. $p(\alpha) = 0$, then the finite field $\mathbb{F}_q$ with $q = p^n$ can be regarded as the set

$$\{a_0 + a_1\alpha + \ldots + a_{n-1}\alpha^{n-1} : a_i \in \mathbb{F}_p \text{ for } i = 0, \ldots, n-1\}.$$

The addition of the elements $a_0+a_1\alpha+\ldots+a_{n-1}\alpha^{n-1}$ and $b_0+b_1\alpha+\ldots+b_{n-1}\alpha^{n-1}$ in $\mathbb{F}_q$ is

$$(a_0 + b_0 \mod p) + (a_1 + b_1 \mod p)\alpha + \ldots + (a_{n-1} + b_{n-1} \mod p)\alpha^{n-1}$$

and the multiplication of these two elements is

$$(a_0 + a_1\alpha + \ldots + a_{n-1}\alpha^{n-1})(b_0 + b_1\alpha + \ldots + b_{n-1}\alpha^{n-1}) \mod p(\alpha).$$

Example 6. *For the irreducible polynomial* $p(x) = x^2 + 1$ *over* $\mathbb{F}_3$*,*

$$\mathbb{F}_9 = \{0, 1, 2, \alpha, 1 + \alpha, 2 + \alpha, 2\alpha, 1 + 2\alpha, 2 + 2\alpha\}$$

where $1 + \alpha^2 = 0$ *over* $\mathbb{F}_3$*, i.e.* $\alpha^2 = 2$*. In that case, the multiplication of* $1 + \alpha$ *by* $1 + 2\alpha$ *is*

$$(1 + \alpha)(1 + 2\alpha) = 1 + 2\alpha + \alpha + 2\alpha^2 = 1 + 0 + 4 = 2$$

and the inverse of $1 + \alpha$*, i.e.* $(1 + \alpha)^{-1}$*, is* $2 + \alpha$ *since*

$$(1 + \alpha)(2 + \alpha) = 2 + \alpha + 2\alpha + \alpha^2 = 2 + 0 + 2 = 1.$$

2.2 Message Encoding and Parsing into Blocks

The main input of hash functions are messages which consists of characters such as letters, numbers, or symbols. Each character can be represented by a number using a table called "character table" or "character set" [11]. Therefore, any message of an arbitrary length can be represented by these numbers. All this numerical representation can be parsed into the blocks of fixed length so that the hash function processes each block to calculate the final hash value.

2.2.1 Message Encoding

In computer science, every character has to be encoded to a numerical representation to be understood by computers. The commonly used numerical representation is *binary representation* which is a sequence consisting of numbers 0 and

1 called as *bits*. In that sense, this representation is also called as *bit representation*. The mapping from characters to numbers are done by using *character tables*. Some standard character tables such as *ASCII* [1] [12, 13] and *base64* [14] are given in Table 2.1 and in Table 2.2, respectively. Characters are represented in 8-bits by using ASCII table and in 6-bits by using base64 table.

Val	Char	Val	Char	Val	Char	Val	Char	Val	Char	Val	Char	Val	Char
0-32	non-printable	64	@	96	‘	128	Ă	160	ă	192	À	224	à
33	!	65	A	97	a	129	Ą	161	ą	193	Á	225	á
34	"	66	B	98	b	130	Ć	162	ć	194	Â	226	â
35	#	67	C	99	c	131	Č	163	č	195	Ã	227	ã
36	$	68	D	100	d	132	Ď	164	ď	196	Ä	228	ä
37	%	69	E	101	e	133	Ě	165	ě	197	Å	229	å
38	&	70	F	102	f	134	Ę	166	ę	198	Æ	230	æ
39	’	71	G	103	g	135	Ğ	167	ğ	199	Ç	231	ç
40	(	72	H	104	h	136	Ĺ	168	Í	200	È	232	è
41	)	73	I	105	i	137	Ľ	169	ľ	201	É	233	é
42	*	74	J	106	j	138	Ł	170	ł	202	Ê	234	ê
43	+	75	K	107	k	139	Ń	171	ń	203	Ë	235	ë
44	,	76	L	108	l	140	Ň	172	ň	204	Ì	236	ì
45	-	77	M	109	m	141	Ŋ	173	ŋ	205	Í	237	í
46	.	78	N	110	n	142	Ő	174	ő	206	Î	238	î
47	/	79	O	111	o	143	Ŕ	175	ŕ	207	Ï	239	ï
48	0	80	P	112	p	144	Ř	176	ř	208	Đ	240	ð
49	1	81	Q	113	q	145	Ś	177	ś	209	Ñ	241	ñ
50	2	82	R	114	r	146	Š	178	š	210	Ò	242	ò
51	3	83	S	115	s	147	Ş	179	ş	211	Ó	243	ó
52	4	84	T	116	t	148	Ť	180	ť	212	Ô	244	ô
53	5	85	U	117	u	149	Ţ	181	ţ	213	Õ	245	õ
54	6	86	V	118	v	150	Ű	182	ű	214	Ö	246	ö
55	7	87	W	119	w	151	Ů	183	ů	215	Œ	247	œ
56	8	88	X	120	x	152	Ÿ	184	ÿ	216	Ø	248	ø
57	9	89	Y	121	y	153	Ź	185	ź	217	Ù	249	ù
58	:	90	Z	122	z	154	Ž	186	ž	218	Ú	250	ú
59	;	91	[	123	{	155	Ż	187	ż	219	Û	251	û
60	<	92	\	124	|	156	Ĳ	188	ĳ	220	Ü	252	ü
61	=	93	]	125	}	157	İ	189	¡	221	Ý	253	ý
62	>	94	^	126	~	158	đ	190	¿	222	Þ	254	þ
63	?	95	_	127	-	159	§	191	£	223	ß	255	ß

Table 2.1: ASCII Encoding Table

Example 7. *The word* `crypto` *can be encoded by using the ASCII table as follows:*

[1] American Standard Code for Information Interchange

Val	Char	Val	Char	Val	Char	Val	Char
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

Table 2.2: base64 Encoding Table

Character	*Value on ASCII Table*	*Binary Representation (in 8-bits)*
c	99	01100011
r	114	01110010
y	121	01111001
p	112	01110000
t	116	01110100
o	111	01101111

So the word crypto *can be represented in ASCII encoding as the concatenation of these* $6 \times 8 = 48$ *bits:*

$$\texttt{crypto} \rightarrow \texttt{011000110111001001111001011100000111010001101111}.$$

Example 8. *The same word* crypto *can be encoded using base64 table as follows:*

Character	*Value on base64 Table*	*Binary Representation (in 6-bits)*
c	28	011100
r	43	101011
y	47	101111
p	41	101001
t	45	101101
o	40	101000

So the word crypto *can be represented in base64 encoding as the concatenation of these* $6 \times 6 = 36$ *bits:*

$$\texttt{crypto} \rightarrow \texttt{011000110111001001111001011100000111010001101111}.$$

As it is explained in the examples, any message of consisting characters can be encoded to its binary representation by using a standard character table. Then these binary representations are used in cryptographic operations.

For simplicity, binary representation can be expressed in hex representation shortly by using 16 characters 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f instead of 4-bits, see Table 2.3.

4-bit Value	Hex Value	4-bit Value	Hex Value	4-bit Value	Hex Value	4-bit Value	Hex Value
0000	0	0100	4	1000	8	1100	c
0001	1	0101	5	1001	9	1101	d
0010	2	0110	6	1010	a	1110	e
0011	3	0111	7	1011	b	1111	f

Table 2.3: Hex table

Example 9. *The encoding of the word* crypto *via ASCII*

$$\texttt{011000110111001001111001011100000111010001101111}$$

can be represented via hex table as

$$\texttt{63727970746f}$$

and the encoding of the word `crypto` *via base64*

$$\texttt{01110010101110111110100110110110100 0}$$

can be represented via hex table as

$$\texttt{72bbe9b68}.$$

Definition 8. *For a positive integer n, a* ***bitstring of length*** **n** *is a sequence of bits*

$$a_1 a_2 \dots a_n$$

where $a_i \in \{0,1\}$ for $i = 1 \dots n$. The set of all bitstrings of length n is denoted by $\{0,1\}^n$. The number of bitstrings in the set $\{0,1\}^n$ is 2^n.

Example 10. *The encoding of the word* `crypto` *via ASCII is a bitstring of length 48. On the other hand, the encoding of the word* `crypto` *via base64 is a bitstring of length* 36*.*

Example 11. *For $n = 3$, the set of bitstrings of length 3 is*

$$\{0,1\}^3 = \{\texttt{000}, \texttt{001}, \texttt{010}, \texttt{011}, \texttt{100}, \texttt{101}, \texttt{110}, \texttt{111}\}$$

and this set contains $2^3 = 8$ elements. For $n = 4$, the set of bitstrings of length 4 is

$$\{0,1\}^4 = \left\{ \begin{array}{c} \texttt{0000}, \texttt{0001}, \texttt{0010}, \texttt{0011}, \texttt{0100}, \texttt{0101}, \texttt{0110}, \texttt{0111}, \\ \texttt{1000}, \texttt{1001}, \texttt{1010}, \texttt{1011}, \texttt{1100}, \texttt{1101}, \texttt{1110}, \texttt{1111} \end{array} \right\}$$

and this set contains $2^4 = 16$ elements.

When the length of a bitstring is a fixed positive integer n, then we say that this bitstring belongs to the set $\{0,1\}^n$. However, the length of messages, and so its length of binary representations, may be different and it can not be fixed. So a big set can be defined containing all bitstrings as follows:

Definition 9. *The set of all bitstrings of arbitrary length is denoted by $\{0,1\}^*$ and is defined as*

$$\{0,1\}^* = \bigcup_{n \in \mathbb{N}} \{0,1\}^n.$$

2.2.2 Parsing Messages into Blocks

The bitstrings can be parsed into blocks of equal lengths before cryptographic operations are applied.

Definition 10. *Consider a bitstring M of length n, i.e. $M \in \{0,1\}^n$, and let b be a positive integer divisor of n. Then the bitstring M can be parsed into k parts $M_1, M_2, \ldots, M_k$ where $k = n/b$ and each M_i is a bitstring of length b. The message M can be written as*

$$M = M_1 M_2 \ldots M_k.$$

*Then each M_i is called as a **block** of M. In other words, M is parsed into k blocks $M_1 M_2 \ldots M_k$. Also it is said that the **block length** is b.*

Example 12. *Take the bitstring of length 48 which corresponds to the ASCII encoding of the word* `crypto`*:*

$$M = \texttt{011000110111001001111001011100000111010001101111}.$$

Then the bitstring M can be parsed into 8 blocks of length 6, $M_1, M_2 \ldots M_8$:

$$M = \texttt{011000}\ \ \texttt{110111}\ \ \texttt{001001}\ \ \texttt{111001}\ \ \texttt{011100}\ \ \texttt{000111}\ \ \texttt{010001}\ \ \texttt{101111}$$

where $M_1 = \texttt{011000}$, $M_2 = \texttt{110111}$, $M_3 = \texttt{001001}$, $M_4 = \texttt{111001}$, $M_5 = \texttt{011100}$, $M_6 = \texttt{000111}$, $M_7 = \texttt{010001}$ and $M_8 = \texttt{101111}$.

A bitstring can be parsed into the blocks when the block length divides the length of the bitstring. However there is a way, called as **padding**, to parse a bitstring whose length is not divided by block length:

Definition 11. *Consider a bitstring M of length n and let b be a block length. If b does not divide n, then append a bitstring P of length k to M, where k is the smallest positive integer so that b divides $n+k$. Then the new bitstring $M' = MP$ can be parsed into the blocks of length b. This operation is called as **padding** and M' is called as **padded bitstring**. In general $P = \texttt{1}||\texttt{0}^{k-1}$ for $k \geq 2$ and $P = \texttt{1}$ when $k = 1$.*

Example 13. *Take the bitstring:*

$$M = \texttt{01100011011100100111100101110000011101000110111}$$

of length 48. Then M can be parsed into 5 blocks of length 10 with padding the bitstring `10`*:*

$$M' = M||\texttt{10} = \texttt{0110001101}\quad\texttt{1100100111}\quad\texttt{1001011100}\quad\texttt{0001110100}\quad\texttt{0110111110}.$$

In hash functions, for the added security, the padding procedure is applied even the block length divides the length of the bitstring. For instance in the hash function `SHA-1`, the length or the checksum of the message in a fixed-length bitstring can be sometimes appended to the bitstring $\texttt{10}\ldots\texttt{0}$.

2.3 Is it Easy or Hard?

Information can be protected by cryptographic tools and it is assumed to be secure if it is not possible for the adversary to defeat the information security. So the question "How is a cryptographic tool secure?" is answered in this section by using the complexity theory in [1].

2.3.1 Complexity Theory

The complexity of the computations in cryptography has two main parameters called *space* and *time*. The space parameter is the amount of storage of the information you need, and the time parameter is the amount of time to do the computations by using the information in the space. The time parameter come first in the complexity assuming that you have enough space to do your computation.

An **algorithm** is a well-defined computational procedure that takes a variable input and halts with an output. The **running time** of an algorithm on a particular input is the number of primitive operations or steps executed. The

worst-case running time of an algorithm is an upper bound on the running time for any input, expressed as a function of the input size. In complexity theory, the running time is approximately evaluated by Big-O notation $\mathcal{O}$ and it is classified in three classes: polynomial-time, exponential-time and sub-exponential time.

Definition 12. *Let f and g be functions on $\mathbb{Z}^+$. Then $f(n) = \mathcal{O}(g(n))$ if there exists a positive constant c and a positive integer n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.*

Definition 13. *Let n be the input size of the algorithm and k be a constant. A **polynomial-time algorithm** is an algorithm whose worst-case running time function is of the form $\mathcal{O}(n^k)$. Any algorithm whose running time cannot be so bounded is called an **exponential-time algorithm**. A **subexponential-time algorithm** is an algorithm whose worst-case running time function is of the form $e^{\mathcal{O}(n)}$.*

Polynomial-time algorithms are regarded as *efficient* algorithms, while exponential-time algorithms are considered *inefficient*. A subexponential-time algorithm is asymptotically faster than an algorithm whose running time is fully exponential in the input size, while it is asymptotically slower than a polynomial-time algorithm.

The complexity theory restricts its attention to *decision problems* which have either YES or NO as an answer.

Definition 14. *The complexity class **P** is the set of all decision problems that are solvable in polynomial time. The complexity class **NP** is the set of all decision problems for which a YES answer can be verified in polynomial time given some extra information, called a certificate. The complexity class **co-NP** is the set of all decision problems for which a NO answer can be verified in polynomial time using an appropriate certificate.*

2.3.2 Models for evulating security

After defining the terms in complexity theory, the security of the cryptographic tools can be evaluated under some security models:

- **Unconditional security.** The question here is whether or not there is enough information available to defeat the system when the adversary is assumed to have unlimited computational resource. This model is also called as *perfect secrecy*.
- **Complexity-theoretic security.** The adversary has polynomial computational power to defeat the information security. Usually the worst-case analysis is used. Polynomial attacks may be feasible under the model but still be computationally infeasible in practice.
- **Provable Security.** A cryptographic tool is said to be *provably secure* if the adversary defeats the system when he solves a well-known and supposedly difficult problem. This problem is typically number-theoretic such as integer factorization or the computation of discrete logarithms.
- **Computational Security.** The system is said to be computationally secure if the perceived level of computation required to defeat it, even using the best attack known, exceed by a comfortable margin, the computational resources of the hypothesized adversary. This is sometimes called *practical security*.
- **Ad-hoc security.** This approach consists of any variety of convincing arguments that every successful attack requires a resource level such as time and space greater than the fixed resources of a perceived adversary. It is also called as *heuristic security*, with security here typically in the computational sense.

In this thesis, we mostly used the models of complexity-theoretic security, provable security and ad-hoc security.

2.3.3 Some perspective for computational security

Some certain quantities are often considered to evaluate the security of cryptographic tools.

Definition 15. *The work factor W is the minimum amount of work required to defeat the information security. It is measured in appropriate units such as elementary operations or clock cycles in computers.*

In that sense, if W is t years for sufficiently large t, the cryptographic tool is a secure system. For comparing the sufficiency for large t, Table 2.4 can be used.

Reference	Magnitude (as power of 10)	Magnitude (as power of 2)
Seconds in a year	$\approx 3 \times 10^7$	$\approx 2^{25}$
Age of our solar system (years)	$\approx 6 \times 10^9$	$\approx 2^{32}$
Seconds since creation of solar system	$\approx 2 \times 10^{17}$	$\approx 2^{57}$
Electrons in the universe	$\approx 8.37 \times 10^{77}$	$\approx 2^{259}$
Number of 75-digit prime numbers	$\approx 5.2 \times 10^{72}$	$\approx 2^{241}$
Binary strings of length 64	$\approx 1.8 \times 10^{19}$	2^{64}
Binary strings of length 128	$\approx 3.4 \times 10^{38}$	2^{128}
Clock cycles per year, 50 MHz computer	$\approx 1.6 \times 10^{15}$	$\approx 2^{50}$
Clock cycles per year, 1 GHz computer	$\approx 3 \times 10^{16}$	$\approx 2^{54}$
In the fastest super-computer (as of Nov 2013),		
Float operations per second	$\approx 33.86 \times 10^{15}$	$\approx 2^{55}$
Float operations per year	$\approx 1.01 \times 10^{24}$	$\approx 2^{80}$

Table 2.4: Reference numbers comparing relative magnitudes

Chapter 3

Incremental Hash Functions

We start by recalling some basic properties of hash functions[1]. In section 3.1, we also give an example of a standard hash function. In section 3.2, we state incremental hash functions and the paradigm standing behind incremental hashing, called *Randomizer-then-Combine Paradigm*. In section 3.3, some examples of incremental hash functions are given with their incrementality properties are given.

3.1 Hash Functions

One of the fundamental cryptographic tools is hash functions. They map the bitstrings of arbitrary length to a bitstring of fixed length. In that sense, they actually map the large domains to smaller ranges. However, they also satisfy some security arguments of cryptographic schemes where they are used. They are mainly used for data integrity and message authentication.

A hash function is defined as follows:

Definition 16. *A function* $H : \{0,1\}^* \to \{0,1\}^n$ *which takes a bitstring* M *of arbitrary finite length, called* ***message****, and outputs a bitstring* $H(M)$ *of fixed*

[1]For the detailed information about hash functions, one can see Chapter 9 of the book [1].

*length n, called **hash** of M, is a **hash function** if it satisfies the following four properties:*

1. ***Ease of Computation:*** *For a given message $M \in \{0,1\}^*$, it is easy to compute its hash $H(M)$.*
2. ***Preimage Resistance:*** *For a given hash $h \in \{0,1\}^n$, it is infeasible to generate a message $M \in \{0,1\}^*$ such that $H(M) = h$.*
3. ***Second Preimage Resistance.*** *For a given message M and its hash $H(M)$, it is infeasible to find a message M' such that $M' \neq M$ but $H(M) = H(M')$.*
4. ***Collision Resistance.*** *It is infeasible to find two messages M, M' with $M \neq M'$ so that they have the same hash, i.e. $H(M) = H(M')$.*

The first property is about the efficiency while the others are about the security of hash functions. The third and the fourth properties may seem to have same meaning, because it is aimed to find two different messages with same hash in both of them. However, they are different: in the third property it is restricted to find a second preimage for a fixed hash, while there is no restriction on the hash value in the fourth property. The expected complexities of security properties of hash functions are given in Table 3.1.

Pre-image resistance	2^n
Second pre-image resistance	2^n
Collision resistance	$2^{n/2}$

Table 3.1: Expected complexities of the security of hash functions for an n-bit output

The hash functions are *many-to-one* functions since the size of the domain $\{0,1\}^*$ is larger than the range $\{0,1\}^n$ for any positive integer n, and this results in collisions. For this reason, a hash function must be constructed so that two randomly chosen inputs are mapped to the same output with probability 2^{-n}.

There are two classes of hash functions, namely Modification Detection Codes (MDCs) and Message Authentication Codes (MACs). The difference between these two classes is that secret keys are not used in MDCs while they are used in MACs. For this reason, MDCs are used in data integrity and MACs are used in authentication. Moreover, MACs can be constructed by using MDCs.

MDCs can be splitted into two groups called *One-way hash functions* (OWHFs) and *Collision-resistance hash functions* (CRHFs). For the OWHFs, preimage resistance and second preimage resistance are required, on the other hand, for the CRHFs, second preimage resistance and collision resistance are required .

3.1.1 Merkle-Damgard Construction

Standard hash functions such as SHA and MD family are constructed based on *Merkle-Damgard* model. In this model, a compression function is used and runs iteratively.

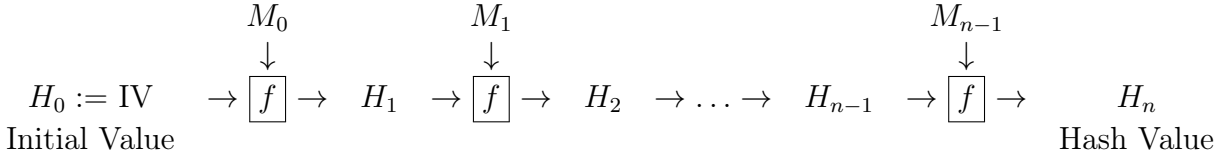

Figure 3.1: Merkle-Damgard Construction

Let $H : \{0,1\}^* \rightarrow \{0,1\}^n$ be a hash function constructed on Merkle-Damgard construction model. H takes a message M parsed into blocks of length b as $M_0 M_1 \ldots M_{n-1}$ and gives out the hash value $H(M)$ by using a compress function f iteratively. In the $i-$th step, this compression function f takes the n-bit bitstring H_{i-1} and the b-bit message block M_{i-1} and gives the next n-bit bitstring H_i (see figure 3.1). Here, the first value H_0 is set to an initial value called IV. This construction can be expressed as follows and it is clearly iterative:

$$H_i := \begin{cases} IV & \text{for } i = 0, \\ f(H_{i-1}, M_{i-1}) & \text{for } i = 1 \ldots n-1 \end{cases}$$

where $f : \{0,1\}^n \times \{0,1\}^b \to \{0,1\}^n$ is the compression function.

In general, the function f compress the message blocks in substeps called *rounds* by using subfunctions called *round function*. In each round t, the round function f_t uses linear structures such as XOR operations, bit rotations, permutations or specific matrices; and nonlinear structures such as nonlinear functions using AND operations or S-boxes. The bits of the block M_i and the value H_i are so mixed by these round functions.

3.1.2 A Standard Hash Function: `SHA-1`

The hash function name `SHA-1` is designed by National Security Agency (NSA) of United States in 1995. It is published in the document FIPS PUB 180-4 [2] by NIST. At present, most of the cryptographic applications and protocols employs `SHA-1`. Its name `SHA` stands for *secure hash algorithm*.

`SHA-1` takes a bitstring of size at most $2^{64} - 1$ (not arbitrary length[2]), and outputs a 160-bit hash value. Its block size is 512-bit and it is designed on Merkle-Damgard construction.

`SHA-1` produces the hash value of a message M in three main steps: In the first step, the padding bitstring is appended to the message M to get padded message M' and then the padded message M' is parsed into n blocks of size 512-bit, i.e. $M' = M_0 M_1 \dots M_{n-1}$. In the second step, the initial state H_0 is set to a constant 160-bit bitstring. In the third step, the compression function f gets the 160-bit state H_i and the 512-bit message block M_i, and outputs the latter 160-bit state H_{i+1} for $i = 0, 1, \dots, n-1$. Here the compression function f runs in 80 subfunctions called also as *rounds*. The final state H_n is output as the final hash value of M.

In the following paragraphs, the mains three steps of `SHA-1` are explained in detail for an l-bit message M.

[2]It is assumed in practice that the size of a message can not be bigger than 2^{64}

1. Step: Padding and Parsing The message is padded by a padding bitstring to make the length of the padded message a multiple of 512 since the block size of SHA-1 is 512. The padding bitstring is specified as follows: it is the concatenation of the bitstring $10\ldots0$ of size k where k is the smallest positive integer satisfying the equation $k+l \equiv 448 \mod 512$, and the 64-bit bitstring representation of the length l:

$$M' := M||\underbrace{\texttt{100}\ldots\texttt{0}}_{k\text{-bit}}||\underbrace{l_1 l_2 \ldots l_{64}}_{\text{64-bit length } l}.$$

Then the padded message M' is parsed into the n blocks of size 512-bit:

$$M_0 M_1 \ldots M_{n-1}.$$

2. Step: Initialization. In the second step, the inital hash value H_0 is set by 160-bit bitsting

$$H_0 = \texttt{67452301efcdab8998badcfe10325476c3d2e1f0}.$$

3. Step: Compression Function. For a given 160-bit state H_i and the 512-bit message block M_i, the compression function f outputs recursively the latter 160-bit state H_{i+1}, i.e. $H_{i+1} := f(H_i, M_i)$ for $i = 0, 1, \ldots, n-1$. Here H_0 is the initialized in the second step and the message blocks M_i are determined in the first step. The final state H_n is the hash value of the message M.

The compression function f has 80 rounds with round functions f_t for $0 \leq t \leq 79$. In each round t, the round function f_t takes 160-bit bitstring in 5 words[3] as $A_t B_t C_t D_t E_t$ and outputs the next 160-bit bitstring in 5 words again $A_{t+1} B_{t+1} C_{t+1} D_{t+1} E_{t+1}$, in other words,

$$A_{t+1} B_{t+1} C_{t+1} D_{t+1} E_{t+1} := f_t(A_t B_t C_t D_t E_t)$$

[3] 32-bit bitstring

where the words $A_{t+1}, B_{t+1}, C_{t+1}, D_{t+1}$ and E_{t+1} are computed as

$$\begin{aligned} A_{t+1} &:= E_t \boxplus g_t(B_t, C_t, D_t) \boxplus (A_t \lll 3) \boxplus W_t \boxplus K_t \\ B_{t+1} &:= A_t \\ C_{t+1} &:= B_t \lll 30 \\ D_{t+1} &:= C_t \\ E_{t+1} &:= D_t \end{aligned}$$

where the nonlinear function g_t is defined as

$$g_t(x, y, z) = \begin{cases} (x \wedge y) \oplus (\neg x \wedge z) & 0 \leq t \leq 19 \\ x \oplus y \oplus z & 20 \leq t \leq 39 \\ (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) & 40 \leq t \leq 59 \\ x \oplus y \oplus z & 60 \leq t \leq 79 \end{cases}$$

Here the words W_t are computed from the words $M_0^{(i)}, M_1^{(i)}, \ldots M_{15}^{(i)}$ of the message block M_i as follows

$$W_t = \begin{cases} M_t^{(i)} & 0 \leq t \leq 15 \\ (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \lll 1 & 16 \leq t \leq 79 \end{cases}$$

and the constants K_t is set to specific words as follows

$$K_t = \begin{cases} \texttt{5a827999} & 0 \leq t \leq 19 \\ \texttt{6ed9eba1} & 20 \leq t \leq 39 \\ \texttt{8f1bbcdc} & 40 \leq t \leq 59 \\ \texttt{ca62c1d6} & 60 \leq t \leq 79 \end{cases}$$

The round function f_t is simply illustrated in Figure 3.2.

Hash Value of M. The compression function f runs on states $H_0, H_1, \ldots, H_{n-1}$ and the message blocks $M_0, M_1, \ldots, M_{n-1}$ until the last message block M_{n-1} is used. The last f function outputs the final state H_n and this H_n is used as the hash value of M.

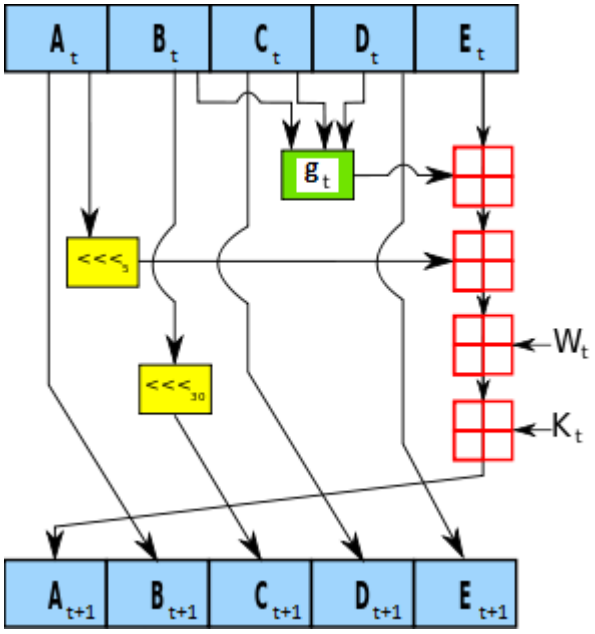

Figure 3.2: The *round* function f_t of f in SHA-1

3.2 Incremental Hash Functions

Most of the hash functions including standard ones such as SHA and MD family have the iterative construction based on Merkle-Damgard model. Such functions runs on the entire data even a small part of the data is changed, because they are iterative. This can be a big problem for the data of big size on the aspect of efficiency.

Bellare, Goldreich and Goldwasser [8] proposed a new construction in 1995 to solve this problem and called it as *incrementality*:

Definition 17. *Given a map f and inputs x and x' where x' is a small modification of x. Then f is said to be* ***incremental*** *if one can update $f(x)$ in time proportional to the amount of modification between x and x' rather than having to recompute $f(x')$ from scratch.*

The incremental hash functions are constructed on this property. Bellare, Goldreich and Goldwasser proposed their first incremental hash function based on exponentiation in a group of prime order using the fact that this group is

abelian. The main feature of the construction in incremental hash functions is mapping the bitstrings blocks to group elements and then multiply them in the group.

Incremental hash functions can be efficiently used in practice where incrementality makes a big difference. Some examples are given below to explain where it makes a big difference, in other words, why we need incremental hash functions:

Example 1: Software Updates. Imagine that a software company produces a software and continuously updates it (slight updates such as fixing a bug etc.) Every time the software is updated, the company should sign it to convince customers that all the changes are made by the company.

Example 2: Big Databases. A state keeps all information of its citizens and wants to be sure that an unauthorized person can not able to change the data. Therefore the state will take its hash to satisfy the integrity of this data. However this data is very big and changes a lot, therefore one wants to take the hash of this changing data in a small time if a slight change is done on an information of a citizen.

Example 3: Virus Protection. An anti-virus program may want to take a hash of the hard drive of the computer to be aware of viruses. However the user do many changes while he is using the computer, so computing the hash value continuously may be difficult. See also [15].

Example 4: Storing Files Online. Many of computer users keep their files such as documents, notes, music or photos on their online storages provided by Dropbox or Google Drive. This storage changes rapidly by uploading new files or deleting old ones so that the user can not absolutely know which files are added, deleted or updated. There may be an unauthorized access to his account and a file is added, deleted or changed without his permission. Therefore he may want to trace this traffic by taking the hash of all his storage.

3.2.1 Randomize-then-Combine Paradigm

Bellare and Micciancio suggested a new paradigm called *The Randomize-then-Combine Paradigm* for collision-free hash functions in [7]. This can be actually seen as the underlying paradigm for the construction of the incremental hash functions. Therefore this concept can be extended to a general view and be re-defined with some small differences without changing the name of the paradigm.

There are two main parts of this paradigm: a *randomizer* function h that maps the bit strings to elements of a group, and the *combine* operation that gives the product of these group elements in the group.

Definition 18. *A function h that maps the blocks of length b to an abelian group G, i.e. $h : \{0,1\}^b \to G$ is called* ***randomizer function****.*

For a padded message M that is parsed into its blocks $M_1 M_2 \dots M_k$ of each length b, the randomizer function h maps these blocks to the group elements as $h(M_i) = g_i \in G$ for $i = 1, \dots, k$.

It is applied to inputs of fixed length. It can be seen as a compression function, however, it can run in parallel since it is not iterative.

Definition 19. *For a message $M = M_1 M_2 \dots M_k$ and a randomizer function $h : \{0,1\}^b \to (G, \odot)$, the group operation $\odot$ is called the* ***combining operation****.*

Incremental hash functions can now be defined using these two definitions:

Definition 20. *Let $(G, \odot)$ be an abelian group, b be the block size, and $h : \{0,1\}^b \to G$ be a randomizer function. Then the function $\texttt{IncHash}_h^G : \{0,1\}^* \to G$ is called* ***incremental hash function****. For a message $M = M_1 M_2 \dots M_k$, the hash value of M is*

$$\texttt{IncHash}_h^G(M) = \bigodot_{i=1}^{k} h(M_i).$$

As randomizer functions and groups vary, incremental hash functions having different security parameters[4] can be defined.

[4]Bellare and Micciancio did these variations in [7]

Incrementality and paralellizability. According to the definition of incremental hash functions, it is clearly seen that the computation via randomizer function h can be parallelizable since h is applied to each block of a message independently. Also the incrementality property holds because the chosen group is abelian and the randomizer function runs on eeach block independently.

The incrementality can be detailed as follows: if a block M_i of the message M is changed to M_i', then the hash value of the new message M' can be easily re-computed from old hash value of the message M by

$$\texttt{IncHash}_h^G(M') = \texttt{IncHash}_h^G(M) \odot h(M_i)^{-1} \odot h(M_i')$$

where $h(M_i)^{-1} \in G$ is the inverse of the group element $h(M_i) \in G$.

Security requirements. The randomizer function h is described as a random oracle [16] and its security is accepted *"ideal"* in [7]. However in practice h can be derived from a standard hash function like `SHA-1`, or from additional parameters such as a set of elements of a group G. Therefore the randomizer function h must be chosen carefully and its security requirements must be taken into consideration. h needs to be collision-free [7], and sometimes one-way if it is required.

The security of combine operation depends on computationally hard problems defined on the group, and so on the choice of the group. These problems and their security reductions is given in Chapter 4.

This paradigm has two parts linked to each other, so the security relies on the weaker part. However assuming that the randomizer function h is *ideal*, the security of the paradigm relies only on the security of the combining operation, in other words, the computationally hard problem in chosen group. In that sense, incremental hash functions give provable security.

In [7], it is stated that the randomizer function is chosen to be ideal and so the security of this paradigm depends only the choice of the group. However it may not be enough in practice since two parts of the paradigm performs independently. Therefore the weaker one of these two parts may result a security gap and so an

attack can be found by the adversary. In other words, if the adversary finds an attack on the h function, then this attack can be applied on the whole hash function without solving the computationally hard problem on the chosen group.

Output truncation. The output of the hash function based on this paradigm is an element of a group G. This element can be expressed in binary representation, and then it can be truncated to a shorter length, for example via a standard hash function like `SHA-1`. It still remains collision-free for the security requirements and parallelizable, but no longer incremental. Therefore, this must be an option when one does not need incrementality.

3.2.2 Standard Hash Functions vs. Incremental Hash Functions

Standard hash functions are iterative because they are based on Merkle-Damgard construction, on the other hand, incremental hash functions use property of incrementality. Moreover, the randomizer function in incremental hash functions can be parallelizable, while the compression function in standard hash functions are not. Also incremental hash functions gives security depending on a computationally hard problem and the standard hash functions are secure when their compression functions are secure. Table 3.2.2 summaries these differences.

	Standard HFs	Incremental HFs
Construction	Iterative	Incremental
Compression Functions	Not Parallelizable	Parallelizable
When some changes applied on data	Re-hash entire data	Apply on the the changed part only
Security argument	Cryptanalysis of the compression function	Provable Security

Table 3.2: Standard hash functions versus Incremental hash functions

3.3 Some Examples of Incremental Hash Functions

In aspect of randomize-then-combine paradigm, some examples of incremental hash functions are given in this section.

3.3.1 Impagliazzo and Naor's hash function

Impagliazzo and Naor defined a hash function in [17] in 1990, which hashes a message bitwise instead of hashing block by block.

Definition 21 (IN's Hash Function)**.** *Let $(G, \odot)$ be a finite abelian group and $g_1, g_2, \ldots, g_n$ be elements in G. Then the hash function $\texttt{INHash}^G_{g_1,\ldots,g_n}$ takes a message $M = M_1 M_2 \ldots M_n$ in its bit representation and computes the hash of M as*

$$\texttt{INHash}^G_{g_1,\ldots,g_n}(M) = \bigodot_{i=1}^{n} (M_i g_i)$$

where $(M_i g_i) = g_i$ if $M_i = 1$ and $(M_i g_i) = e$ (the identity element of G) otherwise.

From the definition, it is clear to see that the number of group elements required is equal to the bitlength of the message. Therefore, it is not so efficient for long messages.

The randomizer function in $\texttt{INHash}^G_{g_1,\ldots,g_n}$ is $h : \{0,1\} \to G$ where $h(M_i) = g_i$ or $h(M_i) = e$ determined by the value of the bit M_i. Moreover, if the message $M = M_1 \ldots M_j \ldots M_n$ is changed to $M' = M_1 \ldots M'_j \ldots M_n$, then the new hash value is

$$\texttt{INHash}^G_{g_1,\ldots,g_n} = \texttt{INHash}^G_{g_1,\ldots,g_n}(M) \odot (M_j g_j) \odot (M'_j g_j).$$

3.3.2 Chaum, van Heijst and Pfitzmann's Hash Function

Chaum, van Heijst and Pfitzmann defined a hash function in [18] in 1991. Their hash function uses modular exponentiation and multiplication. The message blocks in this function can be seen as their integer representation.

Definition 22 (CvHP's Hash Function)**.** *Let $p = 2q + 1$ be a prime for some large prime q, and $a, b \in \mathbb{Z}_p - \{0\}$ be random elements. For any given message $M \in \mathbb{Z}_{q^2}$, M can be written uniquely as $M = M_1 + qM_2$ with $0 \leq M_1, M_2 \leq q-1$. Then, the $\texttt{CvHPHash}_p^{a,b}$ hash of the message M is defined by*

$$\texttt{CvHPHash}_p^{a,b}(M) = a^{M_1} b^{M_2} \mod p.$$

This function is not a hash function in proper sense because it can be applied only to messages whose bit length are $\leq 2 \log_2 q$ whereas a hash function has to be defined for arbitrarily long messages.

The randomizer functions in $\texttt{CvHPHash}_p^{a,b}$ are $h_a, h_b : \mathbb{Z}_q \to \mathbb{Z}_p$ where $h_a(M_1) = a^{M_1}$ and $h_b(M_2) = b^{M_2}$ determined by the parameters a and b. Moreover, if the message $M = (M_1, M_2)$ is changed to $M' = (M_1', M_2)$, then the new hash value is

$$\texttt{CvHPHash}_p^{a,b}(M') = a^{M_1' - M_1} \texttt{CvHPHash}_p^{a,b}(M) \mod p.$$

Clearly, $\texttt{CvHPHash}_p^{a,b}$ is incremental but its incrementality is not so efficient, because the possible number of changes is 1 or 2 since there are two blocks in the message. However, it formed the base work for Bellare, Goldreich and Goldwasser [8].

3.3.3 Bellare, Goldreich and Goldwasser's Hash Function

Bellare, Goldreich and Goldwasser proposed a hash function in [8] in 1995. It is based on the idea which is a combination of ideas standing behind IN's hash function and CvHP's hash function. It uses one modular exponentiation per message block to hash the message.

Definition 23 (BGG's Hash Function)**.** *Let $(G, \odot)$ be an abelian group of prime order p and $g_1, g_2, \ldots, g_n$ be elements in G. For a message $M = M_1 M_2 \ldots M_n$ parsed into blocks of length b, the $\texttt{BGGHash}^G_{g_1,\ldots,g_n}$ of M is*

$$\texttt{BGGHash}^G_{g_1,\ldots,g_n}(M) = \bigodot_{i=1}^{n} g_i^{\langle M_i \rangle}$$

where $\langle M_i \rangle$ is the integer representation of the block M_i.

The randomizer functions in $\texttt{BGGHash}^G_{g_1,\ldots,g_n}$ is $h : \{0,1\} \to G$ where $h(M_i) = g_i^{\langle M_i \rangle}$. Moreover, if the message $M = M_1 \ldots M_j \ldots M_n$ is changed to $M' = M_1 \ldots M'_j \ldots M_n$, then the new hash value is

$$\texttt{BGGHash}^G_{g_1,\ldots,g_n}(M') = \texttt{BGGHash}^G_{g_1,\ldots,g_n}(M) \odot g_j^{-\langle M_j \rangle} \odot g_j^{\langle M'_j \rangle}.$$

$\texttt{BGGHash}^G_{g_1,\ldots,g_n}$ is more efficient then $\texttt{INHash}^G_{g_1,\ldots,g_n}$ since it uses the integer representation of blocks instead of bits, i.e. its block size is bigger. However, the group order restricts this efficiency because it is expected that the block size is $\leq \log p$. Moreover, the storage of parameters $g_1, \ldots, g_n$ may be difficult if the number n is so large.

3.3.4 Bellare and Micciancio's Hash Functions

Bellare and Micciancio suggested a new paradigm called *Randomize-then-Combine Paradigm* in [7] in 1997. This method is based on their previous works [8] and [15], and gives a reduced cost in sense of computation and incrementality. Using this paradigm, they derived three specific hash functions, namely MuHASH, AdHASH and LtHASH.

The randomizer function in their paradigm takes the message blocks with their indices and then maps to the group elements. This function may be a random oracle or a standard hash function like `SHA-1`. By this construction of randomizer function, we get out of using such parameters $g_1, g_2, \ldots, g_n$ as we do in BGG's hash function.

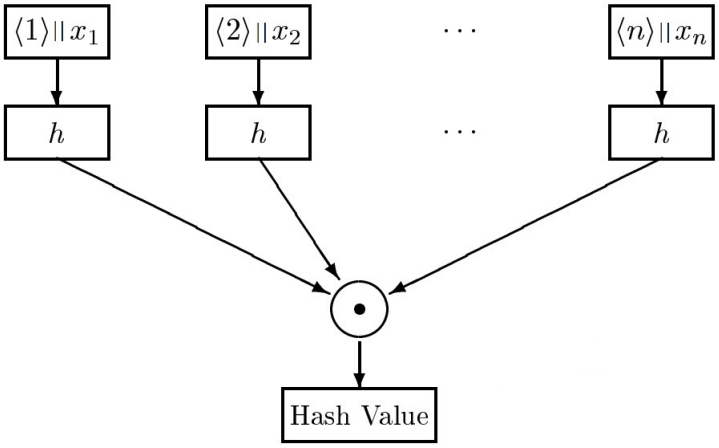

Figure 3.3: Randomizer-then-combine paradigm in BM's hash functions.

Let b be the block length and assume that indices can be represented in l-bit bitstrings. For a message block M_i in a message M, the randomizer function h maps the blocks of length $b+l$ to an element of a group G, i.e. $h : \{0,1\}^{b+l} \to G$, as follows:

$$h(I_i||M_i) = g_i$$

where I_i is the l-bit representation of the index i. It can be clearly seen that the number of blocks is bounded above by $2^l - 1$. The combining operation is the group operation in G which gives the hash value.

Definition 24 (BM's Hash Function)**.** *Let $(G, \odot)$ be an abelian group, b be the block size, l is the upper bound parameter for the number of blocks, and h be the randomizer function $h : \{0,1\}^{b+l} \to G$. Then for a message $M = M_1 M_2 \dots M_n$ parsed into blocks of length b, the $\mathtt{BMHash}_h^G$ of M is*

$$\mathtt{BMHash}(M) = \bigodot_{i=1}^{n} h(I_i||M_i)$$

where I_i is the l-bit representation of the index i.

If a message $M = M_1 \dots M_j \dots M_n$ is changed to $M' = M_1 \dots M'_j \dots M_n$, then the new hash value is

$$\mathtt{BMHash}_h^G(M') = \mathtt{BMHash}_h^G(M) \odot h(I_j||M_j)^{-1} \odot h(I_j||M'_j).$$

The upper bound parameter l for the number of blocks can be set by $l = 80$ since a message with more than 2^{80} blocks is never needed to hash in practice. Moreover, the block length b can be set so that $2^b < |G|$.

Bellare and Micciancio give four types of $\texttt{BMHash}_h^G$ hash function: MuHASHfor multiplicative group G, AdHASH for modular addition, LtHASH for lattices, XHASH for XOR operation. The list of $\texttt{BMHash}_h^G$ is given in Table 3.3.

Type of BM's hash	The based group	Hash value of $M = M_1 M_2 \dots M_n$
Multiplicative Hash MuHASH	Multiplicative group G	$\mathsf{MuHASH}_h^G\ (M) = \prod_{i=1}^{n} h(I_i \| \| M_i)$
Additive Hash AdHASH	$\mathbb{Z}_N$ for large $N \in \mathbb{Z}^+$	$\mathsf{AdHASH}_h^N\ (M) = \sum_{i=1}^{n} h(I_i \| \| M_i) \mod N$
Lattice-based Hash LtHASH	$\mathbb{Z}_p^k$ for prime p and $k \in \mathbb{Z}^+$	$\mathsf{LtHASH}_h^{p,k}\ (M) = \sum_{i=1}^{n} h(I_i \| \| M_i)$ (vector addition in $\mathbb{Z}_p^k$)
XOR Hash XHASH	$\{0,1\}^k$ with XOR addition	$\texttt{XHASH}_h^k(M) = \bigoplus_{i=1}^{n} h(I_i \| \| M_i)$

Table 3.3: Types of $\texttt{BMHash}_h^G$ functions

MuHASHfunction. The name MuHASHcomes from the fact that the combining operation is set to multiplication in a multiplicative group G. For example one can take $G = \mathbb{Z}_p^*$ where p is a prime. In this case the randomizer function h maps the blocks to the elements of $\mathbb{Z}_p^*$ and the combine operation is the multiplication in modulo p.

In MuHASH, the cost for a b-bit block is the sum of computation of h and one modular multiplication per block. However one can see that this cost equals

only to one modular multiplication when the computation of h is comparatively small, for example if SHA is chosen for h. At first look this cost can be seen too much but there are two points to consider: the first, it is multiplication, not exponentiation and the second, total cost of modular multiplications can be reduced by making the block size b larger. In this sense, MuHASHis much faster than any number theory based hash function. Moreover, if hardware for modular multiplication is present then MuHASHbecomes even more efficient to compute.

In MuHASH, incremental operation on a block takes one multiplication and one division, which shows that MuHASHis fast when it updates changes on the message.

AdHASH function. This hash function is called AdHASH since it uses modular addition. It differs from MuHASHbecause it is quite attractive both on the efficiency and on the security fronts. It is a significant improvement to replace the multiplication operation by addition so that AdHASH becomes much faster than MuHASH. Now the cost for hashing of a message of n blocks is n modular addition and the cost for increment operation for a block is two modular additions. By this efficiency and cost, AdHASH can compete with standard hash functions in speed.

LtHASH function. Its name is LtHASH because lattices are used here. The combining operation is set to componentwise addition. In LtHASH, the cost for a b-bit block is the sum of computation of h and one vector addition per block. Incremental operation on a block takes one vector addition and one vector subtraction which can be actually seen as two vector additions in $\mathbb{Z}_p^k$.

XHASH function. They present XHASH using bitwise XOR as combining operation. It works in conventional sense, i.e. its security does not depend on any number theoretical problem. However, setting the combining operation to bitwise XOR makes XHASH insecure because of an attack which uses Gaussian elimination and pairwise independence. Incremental operation on a block takes

two bitstring addition.

Chapter 4

Security of Incremental Hash Functions

In this chapter, computationally hard problems are first defined for provable security. Then security proofs of CvHP's and BGG's hash functions are given in Sections 4.2 and 4.3 respectively. For security proofs of BM's hash functions, Balance Lemma is introduced in relation with DLP in Section 4.4, and finally security of MuHASH, AdHASH, LtHASH and XHASH of BM's hash functions are given in Section 4.5.

4.1 Computationally Hard Problems

We define some computationally hard problems such that security of the incremental hash functions relies on hardness of those problems.

Definition 25 (**Hardness of a Computational Problem**). *A problem P is a (t, ϵ)-hard if no algorithm, limited to run in time t, can find a solution of the problem with probability more than ϵ.*

Definition 26 (**Balance Problem - BP**). *For a group $(G, \odot)$, a positive integer q and random elements $a_1, a_2, \ldots, a_q \in G$, find the weights $\omega_1, \omega_2, \ldots \omega_q \in$*

$\{-1, 0, 1\}$, *not all zero, such that*

$$a_1^{\omega_1} \odot a_2^{\omega_2} \odot \ldots \odot a_q^{\omega_q} = e$$

where e is the identity element of G. This problem is called (G, q)-balance problem.

In (G, q)-balance problem, the elements $a_1, a_2, \ldots, a_q$ are not determined parameters for the problem, because they are randomly given to a (G, q)-balance problem solving algorithm. Therefore, the algorithm must run for any q elements of G.

Using the Definition 25, we say (G, q)-balance problem is (t, ϵ)-hard if no algorithm can find a solution to an instance $a_1, a_2, \ldots, a_q$ of the problem with probability more than ϵ in limited time t.

Definition 27 (**Discrete Logarithm Problem - DLP**). *For a given group G and two elements $g, h \in G$, find the discrete logarithm to the base g of h in group G, that is denoted by $log_g h$. In other words find a non-negative integer x, if it exists, such that $g^x = h$. This problem is called discrete logarithm problem (DLP) in group G.*

For DLP in a group G, the elements g, h are randomly given to an algorithm that solves the problem. Therefore, the algorithm must run for any $g, h \in G$.

If G is a cyclic group, i.e. it is generated by an element $g \in G$, then there is always a non-negative integer x such that $g^x = h$ for any given $h \in G$. Therefore, $g \in G$ is usually given as a generator and h is randomly taken from G.

Example 14. *Take the multiplicative group $\mathbb{Z}_p^*$ for prime $p = 3323$ where $g = 2$ is one of its generators. For $h = 15$, the discrete logarithm to the base g of h in $\mathbb{Z}_{3323}^*$ is*

$$\log_g h = \log_2 15 = 439,$$

in other words $2^{439} = 15 \mod 3323$.

Now we define two more problems which are specified versions of (G, q)-balance problem by choosing $G = \mathbb{Z}_N$ for a large positive integer N.

Definition 28 (**Weighted Knapsack Problem -WKP**). *For a k-bit positive integer N and q numbers $a_1, a_2, \ldots, a_q \in \mathbb{Z}_N$, find weights $w_1, \ldots, w_q \in \{-1, 0, 1\}$, not all zero, such that*

$$\sum_{i=1}^{q} w_i a_i = 0 \mod N.$$

This problem is called (k, q)-weighted-knapsack problem.

In this problem, the elements $a_1, a_2, \ldots, a_q \in \mathbb{Z}_N$ are chosen randomly, therefore they are not actual parameters of the problem, as they were not in balance problem.

This problem can be re-defined by allowing weights to take the only values 0 and 1, because the inverse of an element $a \in \mathbb{Z}_N$ is directly computed as $-a = N - a$ in $\mathbb{Z}_N$. However, the hardness of the new problem does not remain the same, actually the old one is harder than new one.

Definition 29 (**Standard Modular Knapsack Problem -MKP**). *For a k-bit positive integer N and q numbers $a_1, a_2, \ldots, a_q \in \mathbb{Z}_N$, find weights $w_1, \ldots, w_q \in \{0, 1\}$, not all zero, such that*

$$\sum_{i=1}^{q} w_i a_i = 0 \mod N.$$

This problem is called (k, q)-knapsack problem.

It is showed in [19] that the weighted knapsack problem is hard as long as there is no polynomial time approximation algorithm for the shortest vector problem in a lattice.

One more problem as a specified version of (G, q)-balance problem can be defined by choosing $G = \mathbb{Z}_p^k$, i.e. lattices, for positive integers p, k.

Definition 30 (**The Matrix Kernel Problem**). *For a s-bit positive integer p and a matrix $M_{k \times n}$ over $\mathbb{Z}_p$, find a nonzero vector w whose entries can be -1, 0 or 1 such that*

$$Mw = 0$$

where 0 *is the zero vector of* $\mathbb{Z}_p^k$*. This problem is called as* (k, s, n)*-matrix-kernel problem. The operation here is matrix-vector multiplication modulo* p*.*

In matrix kernel problem, the matrix $M_{k\times n}$ over $\mathbb{Z}_p$ is chosen randomly. The columns of the matrix can be seen as the elements $a_1, a_2, \ldots, a_n \in \mathbb{Z}_p^k$ in balance problem. The parameter s gives the size of p, as k gives the size of N in knapsack problems.

It is showed in [19] that the matrix-kernel problem whose parameters satisfy the inequality $ks < n < \dfrac{2^s}{2k^4}$ is hard under the assumption that there is no polynomial time algorithm to approximate the length of shortest vector in a lattice within a polynomial factor.

4.2 Security of CvHP's Hash Function

For given prime number p (such that $p = 2q + 1$ for some large prime q) and elements $a, b \in \mathbb{Z}_p^*$, the $\texttt{CvHPHash}_p^{a,b}$ of a message $M \in \mathbb{Z}_{q^2}$ is defined as

$$\texttt{CvHPHash}_p^{a,b}(M) = a^{M_1} b^{M_2} \mod p$$

where M_1, M_2 is uniquely determined from the equation $M = M_1 + qM_2$ when $0 \leq M_1, M_2 \leq q - 1$. The following theorem shows that the security of CvHP's hash function relies on DLP in multiplicative group $\mathbb{Z}_p^*$:

Theorem 1 ([18])**.** $\texttt{CvHPHash}_p^{a,b}$ *is not collision resistant if and only if DLP is solvable in the multiplicative group* $\mathbb{Z}_p^*$*.*

Proof. Let $M, M' \in \mathbb{Z}_{q^2}$ be two messages such that $M \neq M'$ but $\texttt{CvHPHash}_p^{a,b}(M) = \texttt{CvHPHash}_p^{a,b}(M')$. Then $a^{M_1} b^{M_2} = a^{M_1'} b^{M_2'} \mod p$, or equivalently,

$$a^{M_1 - M_1'} = b^{M_2' - M_2} \mod p$$

holds for uniquely determined $0 \leq M_1, M_1', M_2, M_2' \leq q-1$ where $M = M_1 + M_2 q$ and $M' = M_1' + M_2' q$.

Let $d = \gcd(M_2' - M_2, p-1)$. Since $p-1 = 2q$ and q is prime, it must be the case that $d \in \{1, 2, q, p-1\}$. Therefore we examine these four cases:

Case 1: If $d = 1$, let $y = (M_2' - M_2)^{-1} \mod p-1$. Then we have

$$\begin{aligned} b &= b^{(M_2'-M_2)y} \mod p \\ &= a^{(M_1-M_1')y} \mod p \end{aligned}$$

and so

$$\log_a b = (M_1 - M_1')(M_2' - M_2)^{-1} \mod p-1.$$

Case 2: Now suppose that $d = 2$. Since $p-1 = 2q$ and q is odd, we must have $\gcd(M_2' - M_2, q) = 1$. Let $y = (M_2' - M_2)^{-1} \mod q$. Now $(M_2' - M_2)y = kq + 1$ for some integer k and

$$\begin{aligned} b^{(M_2'-M_2)y} &= b^{kq+1} \mod p \\ &= (-1)^k b \mod p \\ &= \pm b \mod p \end{aligned}$$

since $b^q = -1 \mod p$. Then

$$\begin{aligned} b^{(M_2'-M_2)y} &= a^{(M_1-M_1')y} \mod p \\ &= \pm b \mod p. \end{aligned}$$

It follows that

$$\log_a b = (M_1 - M_1')y \mod p-1$$

or

$$\log_a b = (M_1 - M_1')y + q \mod p-1$$

and it can be easily tested which of these two possibilities is correct.

Case 3: Now suppose $d = q$. But we have $0 \le M_2, M_2' \le q-1$ and so $-(q-1) \le M_2' - M_2 \le (q-1)$. Therefore it is impossible to have $\gcd(M_2' - M_2, p-1) = q$, in other words this case does not arise.

Case 4: The final possibility $d = p-1$ holds only if $M_2 = M_2'$. But we have

$$a^{M_1} b^{M_2} = a^{M_1'} b^{M_2'} \mod p$$

so

$$a^{M_1} = a^{M_1'} \mod p$$

and $M_1 = M_1'$. Thus $M = M'$ which is a contradiction. So this case is not possible either.

Therefore, $\log_a b$ is computable in each case.

Now conversely assume that the DLP is solvable in multiplicative group $\mathbb{Z}_p^*$. Then compute $\log_a b$ and let $l = \log_a b$ for simplicity. We want to find two messages $M, M' \in \mathbb{Z}_{q^2}$ such that $\texttt{CvHPHash}_p^{a,b}(M) = \texttt{CvHPHash}_p^{a,b}(M')$ but $M \neq M'$.

Take an arbitrary message $M \in \mathbb{Z}_{q^2}$ and compute $M_1, M_2 \in \{0, 1, \ldots q-1\}$ so that $M = M_1 + M_2 q$. Then the $\texttt{CvHPHash}_p^{a,b}$ of M is

$$\begin{aligned} \texttt{CvHPHash}_p^{a,b}(M) &= a^{M_1} b^{M_2} \mod p \\ &= a^{M_1} (a^l)^{M_2} \mod p \\ &= a^{M_1 + l M_2} \mod p. \end{aligned}$$

Now we must find M_1' and $M_2' \in \{0, 1, \ldots q-1\}$ so that we can define $M' = M_1' + M_2' q$. Let M_2' be taken arbitrarily in $\{0, 1, \ldots q-1\}$ but $M_2' \neq M_2$, and define M_1' as

$$M_1' = M_1 - l(M_2' - M_2) \mod q.$$

Then the $\texttt{CvHPHash}_p^{a,b}$ of M' is

$$\begin{aligned} \texttt{CvHPHash}_p^{a,b}(M') &= a^{M_1'} b^{M_2'} \mod p \\ &= a^{M_1'} (a^l)^{M_2'} \mod p \\ &= a^{M_1' + l M_2'} \mod p \\ &= a^{M_1 - l M_2' + l M_2 + l M_2'} \mod p \\ &= a^{M_1 + l M_2} \mod p \\ &= \texttt{CvHPHash}_p^{a,b}(M). \end{aligned}$$

So we find a collision for the hash function $\texttt{CvHPHash}_p^{a,b}$. □

4.3 Security of BGG's Hash Function

For an abelian group $(G, \odot)$ of prime order p and elements $g_1, g_2, \ldots, g_n \in G$, the $\texttt{BGGHash}^G_{g_1,\ldots,g_n}$ of a message $M = M_1 M_2 \ldots M_n$ parsed into blocks of length b is

$$\texttt{BGGHash}^G_{g_1,\ldots,g_n}(M) = \bigodot_{i=1}^{n} g_i^{\langle M_i \rangle}$$

where $\langle M_i \rangle$ is the integer representation of the block M_i.

The security of the hash function $\texttt{BGGHash}^G_{g_1,\ldots,g_n}$ depends on DLP in G. For the proof, two algorithms namely collision finder algorithm A and discrete log finder algorithm B are used.

The collision finder algorithm A runs on hash function $\texttt{BGGHash}^G_{g_1,\ldots,g_n}$: it takes a hash value h generated by $\texttt{BGGHash}^G_{g_1,\ldots,g_n}$, and gives out two distinct messages M_1 and M_2 such that $\texttt{BGGHash}^G_{g_1,\ldots,g_n}(M_1) = \texttt{BGGHash}^G_{g_1,\ldots,g_n}(M_2) = h$.

On the other hand, the discrete log finder algorithm B runs on a group G of prime order p. Since it is of prime order, it is generated by an element, say g. We want to find the discrete logarithm of an element $h \in G$, i.e. a positive integer k such that $g^k = h$. The algorithm B takes the inputs G, g, h and outputs the value k.

Bellare et al. gives the security proof for $\texttt{BGGHash}^G_{g_1,\ldots,g_n}$:

Theorem 2 ([8]). *Suppose a collision finder algorithm A succeeds in (t, ϵ) for the hash function $\texttt{BGGHash}^G_{g_1,\ldots,g_n}$. Then the discrete log finder algorithm B succeeds in (t', ϵ') for the group G where $t' = t + \mathcal{O}(n \log^3 p)$ and $\epsilon' = \epsilon/2$.*

Proof. Assume that we have a collision finder algorithm A. Then we can costruct a discrete log finder algorithm B.

The discrete log finder algorithm B takes the inputs G, g, h where the order of G is prime p. It selects $r_1, \ldots, r_n \in \{0, 1\}$ and $u_1, \ldots, u_n \in \{0, 1, \ldots, p-1\}$ at

random. Then it sets

$$g_i = \begin{cases} g^{u_i} & \text{if } r_i = 0 \\ h^{u_i} & \text{if } r_i = 1 \end{cases}$$

for $i = 1, \ldots, n$. The values $g_1, g_2, \ldots, g_n$ are also provided to the algorithm A to specify the hash function $\texttt{BGGHash}^G_{g_1,\ldots,g_n}$. Then the algorithm B calls the collision finder algorithm A and gets two distinct messages $M = M_1 \ldots M_n$ and $M' = M'_1 \ldots M'_n$ such that $\texttt{BGGHash}^G_{g_1,\ldots,g_n}(M) = \texttt{BGGHash}^G_{g_1,\ldots,g_n}(M')$.

Now the algorithm B sets $t_i := \langle M_i \rangle$ and $t'_i := \langle M'_i \rangle$, the integers corresponding to bit representation of message blocks. Then it lets $a = \sum_{r_i=1} u_i(t_i - t'_i) \mod p$. If $a = 0$ then B halts with no input, otherwise B computes the inverse b of a modulo p. Finally it outputs the discrete logarithm of h over g in the group G:

$$\mathsf{index}^G_p(h) = b \sum_{r_i=0} u_i(t'_i - t_i) \mod p.$$

Note that the algorithm B calls the algorithm A once. It also performs some arithmetic modulo p of which the dominant part is $O(n)$ exponentiations that accounts for the claimed running time. Therefore $t' = t + \mathcal{O}(n \log^3 p)$.

The collision finder algorithm A succeeds with probability ϵ. After finding collisions M and M' then it gives

$$\bigodot_{i=1}^{n} g_i^{t_i} = \bigodot_{i=1}^{l} g_i^{t'_i}$$

or equivalently

$$\bigodot_{r_i=1} h^{u_i(t_i - t'_i)} = \bigodot_{r_i=0} g^{u_i(t'_i - t_i)}.$$

Note that the left hand side is h^a. We have that $a \neq 0$ with probability at least $1/2$. Then we get the following equation

$$h = h^{ab} = \bigodot_{r_i=0} g^{bu_i(t'_i - t_i)} = g^{\mathsf{index}^{G_p}_p(h)}.$$

Hence, $\epsilon' = \epsilon/2$. □

4.4 Balance Lemma

Balance problem is the main security argument in incremental hash functions. It is defined in different versions by specifying the underlying group. In this section, the relation between balance problem and collision resistance, which is a security requirement for all hash functions, is defined. Moreover, the reduction of balance problem and discrete logarithm problem is given so that the hardness of balance problem can be computed by the hardness of discrete logarithm problem for a given group.

4.4.1 Balance Problem & Collision Resistance

(G, q)-balance problem is given in Definition 26: for a group $(G, \odot)$, a positive integer q and randomly chosen elements $a_1, a_2, \ldots, a_q \in G$, find the weights $\omega_1, \omega_2, \ldots \omega_q \in \{-1, 0, 1\}$, not all zero, such that

$$a_1^{\omega_1} \odot a_2^{\omega_2} \odot \ldots \odot a_q^{\omega_q} = e$$

where e is the identity element of G. This problem is said to be (t, ϵ)-hard if no algorithm that is limited to run in time t can find a solution $\omega_1, \omega_2, \ldots \omega_q \in \{-1, 0, 1\}$ for instance $a_1, a_2, \ldots, a_q \in G$ with probability more than ϵ.

Let $\mathsf{IncHash}[G, b, h]$ be an incremental hash function where G is the underlying abelian group, b is the block length and $h : \{0,1\}^b \to G$ is a randomizer function. Now the collision resistance for $\mathsf{IncHash}[G, b, h]$ is defined:

Definition 31 (**Collision-resistance**). *Let C be a collision finder algorithm that finds two messages M, M' with $M \neq M'$ so that $\mathsf{IncHash}(M) = \mathsf{IncHash}(M')$. Then the incremental hash function $\mathsf{IncHash}[G, b, h]$ is said to be (t, q, ϵ)-**collision-free** if no collision finder algorithm C which is limited to run in time t and makes at most q oracle queries via h succeeds with probability at least ϵ.*

It is easily seen that finding the weights in (G, q)-balance problem is equivalent to finding two disjoint subsets $I, J \subset \{1, 2, \ldots, q\}$ so that $\bigodot_{i \in I} a_i = \bigodot_{j \in J} a_j$.

By this equality, it seems to find finding collisions in $\mathsf{IncHash}[G, b, h]$ is same as solving the balance problem in G. Remember that the integer q is the number of computations of h and it is assumed that h is ideal, i.e. collision-free. Moreover, $c > 1$ is a small constant depending on the model of computation which can be derived from the proof.

Lemma 1 (**The Balance Lemma** [7])**.** *For an abelian group $(G, \odot)$ and a positive integer q, assume that (G, q)-balance problem is (t', ϵ')-hard. Then the hash function $\mathsf{IncHash}[G, b, h]$ is a (t, q, ϵ)-collision-free where $\epsilon = \epsilon'$ and $t = t'/c - qb$.*

Proof. Let C be a given collision-finder algorithm for $\mathsf{IncHash}[G, b, h]$, which takes the group G and the randomizer function h, and outputs two different messages M and M' with $\mathsf{IncHash}(M) = \mathsf{IncHash}(M')$. Now an algorithm K that solves the (G, q)-balance problem can be constructed as follows.

The algorithm K takes the group G and a list of values $\{a_1, a_2 \ldots, a_q\}$ selected uniformly at random in G. K runs on randomizer function h of C, answering its oracle queries with the values $a_1, a_2 \ldots, a_q$ in order without repeating. Let $Q_i \in \{0,1\}^b$ denote the i-th oracle query of h so that $h(Q_i) = a_i$, and let R be the set of all Q_i's, i.e. $R = \{Q_1, Q_2, \ldots, Q_q\}$.

The algorithm K runs C once and takes the outputs of C, i.e. two messages $M = M_1 M_2 \ldots M_n$ and $M' = M'_1 M'_2 \ldots M'_m$ with $\mathsf{IncHash}(M) = \mathsf{IncHash}(M')$ but $M \neq M'$. This means that $\bigodot_{i=1}^{n} h(M_i) = \bigodot_{i=1}^{m} h(M'_i)$.

Without loss of generality assume that all the blocks appear in the query set R, i.e. $M_1, M_2, \ldots, M_n, M'_1, M'_2, \ldots, M'_m \in R$. We can index these blocks with respect to queries the Q_i: let $f_M(i)$ be the unique value $j \in \{1, 2, \ldots, q\}$ such that $M_i = Q_j$ and let $f_{M'}(i)$ be the unique value $j \in \{1, 2, \ldots, q\}$ such that $M'_i = Q_j$. Then define the sets $I = \{f_M(i) : i = 1, \ldots, n\}$ and $J = \{f_{M'}(i) : i = 1, \ldots, m\}$. and re-write the equation $\bigodot_{i=1}^{n} h(M_i) = \bigodot_{i=1}^{m} h(M'_i)$ as follows:

$$\bigodot_{i \in I} a_i = \bigodot_{j \in J} a_j.$$

Since $M \neq M'$, it is clear that $I \neq J$. Now define

$$w_i = \begin{cases} -1 & \text{, if } i \in J - I \\ 0 & \text{, if } i \in I \cap J \text{ or } i \notin I \cup J \\ +1 & \text{, if } i \in I - J \end{cases}$$

for $i = 1, \ldots, q$. All $w_1, \ldots, w_q$ can not be 0 since $I \neq J$. Finally, it implies that

$$a_1^{w_1} \odot \ldots \odot a_q^{w_q} = e$$

where e is the identity element of G.

The algorithm K succeeds if the algorithm C does, in other words $\epsilon = \epsilon'$. The time for computation is $t'/c = t + qb$ since K runs C for once and makes q queries via h function. □

According to Balance Lemmaa we can say that if (G, q)-balance problem is hard in the group G then the incremental hash function $\mathsf{IncHash}(G, b, h)$ is collision-free.

4.4.2 Balance Problem & Discrete Logarithm Problem

Balance Lemma shows that the collision resistance of incremental hash functions $\mathsf{IncHash}(G, b, h)$ relies on the hardness of (G, q)-balance problem. So the important question must be answered: "How can the hardness of (G, q)-balance problem be determined?".

In this section, three theorems are given to answer this question by showing the relation between balance problem (BP) and discrete logarithm problem (DLP) for different groups G.

The following theorem proves this relation for any finite group.

Theorem 3 (**BP & DLP for General Groups** [7])**.** *Let G be a group of order L-bit integer. Assume that the discrete logarithm problem in G is (t', ϵ')-hard. Then for any positive integer q, the (G, q)-Balance Problem is (t, ϵ)-hard, where*

$$\epsilon = q\epsilon'$$

and

$$t = t'/c - q \cdot [T_{rand}(G) + T_{exp}(G) + L]$$

where $T_{rand}(G)$ and $T_{exp}(G)$ are required time for choosing a random element in G and exponentiation operation in G, respectively.

Proof. Let A be a given algorithm to solve (G, q)-balance problem: it takes G and a sequence of q random elements $a_1, a_2, \ldots, a_q \in G$ and outputs $w_1, w_2, \ldots, w_q \in \{-1, 0, 1\}$, not all zero, such that $\bigodot_{i=1}^{q} a_i^{w_i} = e$. Now an algorithm I which solves discrete logarithm problem in G can be constructed: for $g, h \in G$, a non-negative integer x can be found so that $h = g^x$ via the algorithm A.

Let $\rho = |G|$. The algorithm I first picks a random integer $q^* \in \{1 \ldots q\}$. Then I computes the elements a_i for $i = 1, \ldots, q$ as follows: If $i = q^*$ then $a_i = h$, otherwise it chooses a random $r_i \in \mathbb{Z}_\rho$ and sets $a_i = g^{r_i}$.

I runs the algorithm A for the group G and elements $a_1, \ldots, a_q \in G$, and gets the weights $w_1, \ldots, w_q$, not all zero, such that

$$a_1^{w_1} \odot \ldots \odot a_q^{w_q} = e.$$

Let i^* be such that $w_{i^*} \neq 0$. Since q^* is chosen randomly and unknown to A, the case $q^* = i^*$ with probability $1/q$. Without loss of generality take $q^* = i^* = 1$ and substitute the values in the previous equation

$$h^{w_1} \odot g^{w_2 r_2} \odot \ldots \odot g^{w_q r_q} = e$$

and re-arrange the terms by noticing $w_1^{-1} = -w_1$ in $\mathbb{Z}_\rho$

$$h = g^{-w_1(w_2 r_2 + \ldots + w_q r_q)}.$$

Thus the discrete logarithm of h is

$$\log_g h = -w_1(w_2 r_2 + \ldots + w_q r_q) \mod \rho.$$

The algorithm I succeeds when A is successful and $w_{q^*} \neq 0$. Therefore, $\epsilon' = \epsilon/q$.

On the other hand, the algorithm I runs A once. Computing each element a_i takes one random choice and one exponentiation in G, that is, $T_{rand}(G) + T_{exp}(G)$.

In addition, the final modular additions takes qL time. Therefore, the total time for I is $t' = t + q\,[T_{rand}(G) + T_{exp}(G) + L]$. □

For prime order groups, the following theorem proves this relation by improving the probability.

Theorem 4 (**BP & DLP for Groups of Prime Order** [7])**.** *Let G be a group of L-bit prime order. Assume the discrete logarithm problem in G is (t', ϵ')-hard. Then for any positive integer q, the (G, q)-Balance Problem is (t, ϵ)-hard, where*

$$\epsilon = 2\epsilon'$$

and

$$t = t'/c - q \cdot [T_{rand}(G) + T_{mult}(G) + T_{exp}(G) + L] - L^2$$

where $T_{rand}(G)$, $T_{mult}(G)$ and $T_{exp}(G)$ are required time for choosing a random element in G, group operation in G and exponentiation operation in G, respectively.

Proof. Assume G is a group of prime order. Let $|G| = \rho$ and $g \in G$ be a generator of G. Let A be a given algorithm to solve (G, q)-balance problem. An algorithm I which solves discrete logarithm problem in G can be constructed by using A.

Let h be a given element in G. The algorithm I needs to find a positive integer x so that $h = g^x$.

The algorithm I first chooses random $r_i \in \mathbb{Z}_\rho$ and $d_i \in \{0, 1\}$ and sets $a_i = g^{d_i} h^{r_i}$ for each $i = 1 \ldots q$. I runs the algorithm A with the group G and the elements $a_1, \ldots, a_q$ and gets the weights $w_1, \ldots, w_q$, not all zero, such that $\bigodot_{i=1}^{q} a_i^{w_q} = e$. When the values are substituted, it results

$$h^{w_1 r_1} g^{w_1 d_1} \ldots h^{w_q r_q} g^{w_q d_q} = e.$$

Re-arranging terms gives

$$h^{w_1 r_1 + \ldots + w_q r_q} = g^{-(w_1 r_1 + \ldots + w_q r_q)}.$$

By letting

$$\begin{aligned} r &= w_1 r_1 + \ldots + w_q r_q \mod \rho \\ d &= -(w_1 r_1 + \ldots + w_q r_q) \mod \rho \end{aligned}$$

we get $h^r = g^d$. If $r \neq 0$, the algorithm I computes $r^{-1} \mod \rho$ and gives the discrete logarithm of h:

$$log_g h = r^{-1} d.$$

The algorithm I succeeds when A is successful and $r \neq 0$. It can be observed that $r \neq 0$ with probability at least $1/2$ since the value of d_1 remains equi-probably 0 or 1 from the point of view of A, and is independent of other d_i values. At most one of the two possible values of d_1 can make $d = 0$ and hence $r = 0$. Thus, $\epsilon = 2\epsilon'$.

The algorithm I runs A once. Computing each element a_i takes one random choice, one exponentiation and one multiplication (by g since $g^{d_i} = g$ or 1) in G, that is, $T_{rand}(G) + T_{exp}(G) + T_{mult}(G)$. In addition, computing r and d takes qL time. Also computing $r^{-1}d$ in the final takes L^2 time. Therefore, the total time for I is $t' = t + q\,[T_{rand}(G) + T_{exp}(G) + T_{mult}(G) + L] + L^2$. □

The order of the group $\mathbb{Z}_p^*$ for prime p is not prime, therefore Theorem 4 does not work for this group. However, the following theorem gives a better relation rather than Theorem 3.

Theorem 5 (BP & DLP for $\mathbb{Z}_p^*$ [7]). *Let p be a k-bit prime number with $k \geq 6$. Assume the discrete logarithm problem in $\mathbb{Z}_p^*$ is (t', ϵ')-hard. Then for any positive integer q, the $(\mathbb{Z}_p^*, q)$-balance problem is (t, ϵ)-hard, where*

$$\epsilon = 4\epsilon' \ln(0.694k)$$

and

$$t = t'/c - qk^3 - k^2.$$

Proof. Let an algorithm A be given for $(\mathbb{Z}_p^*, q)$-balance problem. We construct an algorithm I that solves discrete logarithm problem in $\mathbb{Z}_p^*$ using A. Let $h \in \mathbb{Z}_p^*$ be given to the algorithm I.

The order of $\mathbb{Z}_p^*$ is $\rho = p - 1$. Let g be a generator of $\mathbb{Z}_p^*$.

The algorithm I first chooses random $r_i, d_i \in \mathbb{Z}_\rho$ and sets $a_i = g^{d_i} h^{r_i}$. I runs the algorithm A for $\mathbb{Z}_p^*$ and the elements $a_1, \ldots, a_q \in \mathbb{Z}_p^*$, and gets the weights $w_1, \ldots, w_q$, not all zero, such that $\prod_{i=1}^{q} a_i^{w_q} = 1 \mod p$. When the values are substituted, it results

$$h^{w_1 r_1} g^{w_1 d_1} \ldots h^{w_q r_q} g^{w_q d_q} = 1 \mod p.$$

Re-arranging terms gives

$$y^{w_1 r_1 + \ldots + w_q r_q} = g^{-(w_1 r_1 + \ldots + w_q r_q)} \mod p.$$

By letting

$$\begin{aligned} r &= w_1 r_1 + \ldots + w_q r_q \mod \rho \\ d &= -(w_1 r_1 + \ldots + w_q r_q) \mod \rho \end{aligned}$$

we get $h^r = g^d \mod p$. If $gcd(r, \rho) = 1$, the algorithm I computes $r^{-1} \mod \rho$ and gives the discrete logarithm of h:

$$log_g h = r^{-1} d \mod \rho.$$

The algorithm I succeeds when A is successful and $gcd(r, \rho) = 1$. It can be observed that $gcd(r, \rho) = 1$ with probability at least $\frac{1}{4 \ln \ln \rho}$ since the inequality

$$\frac{\phi(\rho)}{\rho} \geq \frac{1}{4 \ln \ln(\rho)} \geq \frac{1}{4 \ln \ln(2^k)} \geq \frac{1}{4 \ln k \ln 2} \geq \frac{1}{4 \ln(0.694k)}$$

holds for $k \geq 6$ ([20]). Therefore, $\epsilon = 4\epsilon' \ln(0.694k)$.

The algorithm I runs A once. Computing each element a_i takes two random choices, two exponentiations and one multiplication in $\mathbb{Z}_p^*$, that is, k^3 time. In addition, computing r and d takes qk time. Also computing $r^{-1}d$ in the final takes k^2 time. Therefore, the total time for I is $t' = t + q(k^3) + k^2$. □

4.5 Security of Bellare and Micciancio's Hash Functions

Bellare and Micciancio (BM) suggested three specific incremental hash functions, namely MuHASH, AdHASH and LtHASH, named with respect to chosen underlying group. They all give provable security depending on the hardness of discrete logarithm problem, the weighted knapsack problem and matrix kernel problem, respectively. However the main problem where the security of these incremental hash functions depends on is the balance problem.

BM's incremental hash functions are defined with a randomizer function h and a group G. However, the randomizer function is not defined specially as it is defined on other incremental hash functions. As it is stated in [7], one can use a standard hash function like `SHA-1` in practice, or a random oracle [16]. In that sense, Bellare and Micciancio assumes that this randomizer function is *ideal* hash function, in other words, it is totally secure. Therefore they analyze only the security of the combining operation which relies on balance problem in chosen group.

Choosing the combining operation in BM's hash functions is actually equivalent to choosing group or class of groups. Bellare and Micciancio states that one must be careful about choosing the group so that a computationall hard problem such as DLP or weighted knapsack problem is hard to solve in the group, because the right choice is crucial for security and efficiency [7]. For example, choosing XOR operation does not work because an attack (see 4.5.4) is proposed for this hash function. On the other hand, they proposed multiplication in a group where DLP is hard, resulting MuHASH; and addition modulo an integer of appropriate size, resulting AdHASH . LtHASH is an advanced version of AdHASH over lattices.

4.5.1 Security of MuHASH

Bellare and Micciancio claim that MuHASHis collision-free as long as DLP in group G is hard and the randomizer function h is ideal. Furthermore they claim that MuHASHmay be secure even if DLP in G is easy since there is no attack that finds collisions even if it is easy to compute discrete logarithms.

The security of MuHASHis satisfied with DLP by constructing the relations between balance problem and collision resistance and between balance problem and DLP. They give the main theorem and improve it for groups of prime orders and $\mathbb{Z}_p^*$.

Theorem 6 (**Security in General Groups** [7]). *Let G be a group of L-bit order and assume DLP in G is (t', ϵ')-hard. Then for any q, MuHASH_h^G is a (t, q, ϵ)-collision-free hash function where*

$$\epsilon = q\epsilon' \quad \textit{and} \quad t = t'/c - q \cdot [T_{rand}(G) + T_{exp}(G) + L + b]$$

where $T_{rand}(G)$ and $T_{exp}(G)$ are required time for choosing a random element in G and exponentiation in G, respectively.

Proof. Lemma 1 and Theorem 3. □

In Theorem 6, the probability relation between finding collisions and solving DLP is $\epsilon = q\epsilon'$. Since a typical choice of q is about 2^{50} in practice, the DLP in G must be very hard in order to make finding collisions in the hash function quite hard.

Theorem 7 (**Security in Groups of Prime Order** [7]). *Let G be a group of L-bit prime order and assume DLP in G is (t', ϵ')-hard. Then for any q, MuHASH_h^G is a (t, q, ϵ)-collision-free hash function where*

$$\epsilon = 2\epsilon' \quad \textit{and} \quad t = t'/c - q \cdot [T_{rand}(G) + T_{mult}(G) + T_{exp}(G) + L + b] - L^2$$

where $T_{rand}(G)$, $T_{mult}(G)$ and $T_{exp}(G)$ are required time for choosing a random element in G, group operation in G and exponentiation operation in G, respectively.

Proof. Lemma 1, Theorem 4. □

Theorem 8 (**Security in $\mathbb{Z}_p^*$** [7])**.** *Let p be a k-bit prime number with $k \geq 6$. Suppose DLP in $\mathbb{Z}_p^*$ is (t', ϵ')-hard. Then for any q, MuHASH_h^G is a (t, q, ϵ)-collision-free hash function where*

$$\epsilon = 4\epsilon' \ln(0.694k) \quad \textit{and} \quad t = t'/c - qk^3 - qb.$$

Proof. Lemma 1 and Theorem 5. □

4.5.2 Security of AdHASH

Collision-freeness of AdHASH is related to the *Weighted Knapsack Problem*: Bellare and Micciancio show that AdHASH is collision-free as long as the weighted knapsack problem is hard and the randomizer function h is ideal. According to [19], the weighted knapsack problem is hard as long as there is no polynomial time approximation algorithm for the shortest vector problem in a lattice.

The theorem below uses the fact that this problem is a special case of balance problem, and then it applies Lemma 1 in the proof. Below $c > 1$ is a small constant, depending on the computation:

Theorem 9 ([7])**.** *Let N be a k-bit integer and q be a positive integer such that the (k, q)-weighted-knapsack problem is (t', ϵ')-hard. Then AdHASH_h^N is a (t, q, ϵ)-collision-free hash function where*

$$\epsilon = \epsilon' \quad \textit{and} \quad t = t'/c - qN.$$

Proof. When we set the group G to $\mathbb{Z}_N$ in Lemma 1, the incremental function becomes AdHASH_h^N . On the other hand, $(\mathbb{Z}_N, q)$-balance problem is actually (N, q)-weighted-knapsack problem. Therefore, $\epsilon = \epsilon'$ and $t = t'/c - qN$ by Lemma 1. □

4.5.3 Security of LtHASH

Collision-freeness of LtHASH is related to the *Matrix-Kernel Problem*: Bellare and Micciancio claims that LtHASH is collision-free as long as the matrix-kernel problem is hard and the randomizer function h is ideal.

The proof of the following theorem is similar to the proof of Theorem 9. Below $c > 1$ is a small constant, depending on the computation:

Theorem 10 ([7]). *Let k, q, s be integers such that the (k, q, s)-matrix-kernel problem is (t', ϵ')-hard. Then $\mathsf{LtHASH}_h^{p,k}$ (where p is a s-bit prime) is a (t, q, ϵ)-collision-free hash function where*

$$\epsilon = \epsilon' \quad \text{and} \quad t = t'/c - qks.$$

Proof. Setting the group G to $\mathbb{Z}_p^k$ in Lemma 1 makes the incremental function $\mathsf{LtHASH}_h^{p,k}$. Moreover, $(\mathbb{Z}_p^k, q)$-balance problem is equivalent to (k, q, s)-matrix-kernel problem. Therefore, $\epsilon = \epsilon'$ and $t = t'/c - qk \log p$ by Lemma 1. □

4.5.4 Security of XHASH

Bellare and Micciancio set the combining operation to bitwise XOR, however XHASH becomes insecure in this case. They present an attack which uses Gaussian elimination and pairwise independence.

For a fixed k, the underlying group G is $\{0,1\}^k$. Let $z \in \{0,1\}^k$ be given. In this attack, we want to find a message M so that $\mathsf{XHASH}_h^k(M) = h$.

Fix two messages of length of n blocks, $M^0 = M_1^0 \dots M_n^0$ and $M^1 = M_1^1 \dots M_n^1$ such that $M_i^0 \neq M_i^1$ for $i = 1, 2, \dots, n$, i.e. the i-th blocks are different. For any n-bit bitstring $y = y_1 y_2 \dots y_n$, let

$$M^y = M_1^{y_1} M_2^{y_2} \dots M_n^{y_n}$$

so that the i-th block of M^y is either M_i^0 or M_i^1.

Now, compute $2n$ values $\alpha_i^j := h(I_i||M_i^j)$ for $i = 1 \ldots n$ and $j = 0, 1$ where I_i is the bitstring representation of index i. We want to find a y such that

$$\mathsf{XHASH}_h^k\ (M^y) = \alpha_1^{y_1} \oplus \alpha_2^{y_2} \oplus \ldots \oplus \alpha_n^{y_n} = h.$$

This equation can be rewritten as

$$\bigoplus_{i=1}^{n} \left(\alpha_i^0 y_i \oplus \alpha_i^1 (1 - y_i)\right) = h$$

since

$$\alpha_i^0 y_i \oplus \alpha_i^1 (1 - y_i) = \begin{cases} \alpha_i^0 & \text{if } y_i = 1 \\ \alpha_i^1 & \text{if } y_i = 0 \end{cases}$$

for $i = 1, 2, \ldots, n$.

Regard $y_1, y_2, \ldots, y_n$ as variables and define new variables $\overline{y_1}, \overline{y_2}, \ldots, \overline{y_n}$ so that $\overline{y_i} = 1 - y_i$. Then we have $n + k$ equations in $2n$ unknowns over $\mathbb{Z}_2$:

$$\begin{aligned} y_i \oplus \overline{y_i} &= 1 \quad \text{for } i = 1, 2, \ldots, n \\ \bigoplus_{i=1}^{n} \left(\alpha_i^0[j] y_i \oplus \alpha_i^1[j](1 - y_i)\right) &= h[j] \quad \text{for } j = 1, 2, \ldots, k \end{aligned}$$

where $\alpha[j]$ and $h[j]$ denotes the j-th bit of bitstrings α and h.

There exists a solution[1] y with probability at least $1 - \dfrac{2^k}{2^n}$. This probability is $\dfrac{1}{2}$ when $n = k + 1$ and the equation system can be solved via Gauss elimination by setting one unknown arbitrarily since there are $n + k = 2k + 1$ equations and $2n = 2k + 2$ unknowns.

This attack makes $2n$ many h-computations, sets up a certain linear system, and then uses Gauss elimination to solve it.

[1] see Lemma A.1 in [7]

Chapter 5

Elliptic Curve Only Hash ECOH

The incremental hash function ECOH, Elliptic Curve Only Hash, is proposed by Dan Brown in 2008. ECOH is based on Bellare and Micciancio's hash function MuHASHand uses elliptic curves on finite fields as a DLP-hard group. It does not use a specified randomizer function, the blocks of a message represented directly by the corresponding points on the given elliptic curve.

5.1 Elliptic Curves in Cryptography

Let K be a field. An elliptic curve E over K is the set of points $(x, y) \in K \times K$ satisfying the equation

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

over K for some constants a_1, a_2, a_3, a_4, a_6, and a point $\mathcal{O}$ called **point at infinity**. This equation is called **generalized Weierstrass equation**. If $char(K) \neq 2, 3$ then it can be transformed to

$$y^2 = x^3 + Ax + B$$

where A, B are constants in K. This equation is called **Weierstrass equation**.

An addition operation $+_E$ can be defined on elliptic curves as follows: Let E be an elliptic curve over a field K with $char(K) \neq 2, 3$ and defined by the equation $y^2 = x^3 + Ax + B$ for some constants $A, B \in K$. For given two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ on E such that $P_1, P_2 \neq \mathcal{O}$, define the addition of two points as

$$P_3 = P_1 +_E P_2$$

where $P_3 = (x_3, y_3)$. Then the the coordinates of the point P_3 can be calculated as follows:

1. If $x_1 \neq x_2$, then

$$x_3 = m^2 - x_1 - x_2, \quad y_3 = m(x_1 - x_3) - y_1, \quad \text{where } m = \frac{y_2 - y_1}{x_2 - x_1}.$$

2. If $x_1 = x_2$ but $y_1 \neq y_2$, then $P_1 +_E P_2 = \mathcal{O}$.

3. If $P_1 = P_2$ and $y_1 \neq 0$, then

$$x_3 = m^2 - 2x_1, \quad y_3 = m(x_1 - x_3) - y_1, \quad \text{where } m = \frac{3x_1^2 + A}{2y_1}.$$

4. If $P_1 = P_2$ and $y_1 = 0$, then $P_1 +_E P_2 = \mathcal{O}$.

For elliptic curves over a field K of characteristic 2 or 3, the addition of points is similarly formulated but in more sophisticated way.

Under this adding operation, the elliptic curve E forms an abelian group:

Theorem 11 ([21]). *The addition of points on an elliptic curve E satisfies the following properties:*

1. *$P_1 +_E P_2 = P_2 +_E P_1$ for all P_1, P_2 on E,*
2. *$P +_E \mathcal{O} = P$ for all P on E,*
3. *Given a point P on E, there exists P' on E with $P +_E P' = \mathcal{O}$. This point P' is usually denoted by $-P$,*
4. *$(P_1 +_E P_2) +_E P_3 = P_1 +_E (P_2 +_E P_3)$ for all P_1, P_2, P_3 on E.*

In other words, the points on $(E, +_E)$ form an additive abelian group where $\mathcal{O}$ is the identity element of E.

Proof. See the proof of Theorem 2.1 in [21] □

There are some standard elliptic curves proposed by NIST in FIPS 186-3 [22]. The principal parameters are the elliptic curve E defined with an equation over a finite field and a designated point $G = (G_x, G_y) \in E$ called *the base point*. The base point has order r which is a large prime. The number of points on the curve is $n = fr$ for some integer f (the cofactor) not divisible by r. For efficiency reasons, it is desirable to take the cofactor to be as small as possible. All of the NIST curves have cofactors 1, 2, or 4.

There are two types of NIST curves: 1) Curves over prime fields $\mathbb{F}_p$ for prime number p and 2) Curves over binary fields $\mathbb{F}_{2^m}$ for some positive integer m. The curves over binary fields has also two subtypes because of their different equations.

For curves over prime fields $\mathbb{F}_p$, NIST determines five prime numbers for p where the length of primes are 192-bit, 224-bit, 256-bit, 384-bit, 521-bit. The curves are labeled as **P-192**, **P-224**, **P-256**, **P-384** and **P-521**. Each curve satisfies the equation

$$E : y^2 = x^3 - 3x + b.$$

Here the coefficient A is set to -3 for efficiency reasons. The coefficients b and p are chosen carefully to be sure that the group $(E, +_E)$ is of prime order r. In that case, the base point G generates all the group and therefore the cofactor is $f = 1$. The parameters of the curve **P-256** is given in Table 5.1.

For curves over binary fields $\mathbb{F}_{2^m}$, NIST determines five positive prime integers for $m : 163, 233, 283, 409, 571$. Two subtypes of binary curves, called pseudorandom curves and Koblitz curves, are given by the equations

$$E_b : y^2 + xy = x^3 + x^2 + b$$

for pseudo-random curves, and

$$E_a : y^2 + xy = x^3 + ax^2 + 1$$

p	1157920892103562487626974469494075735300861434152903141955336 31308867097853951
r	1157920892103562487626974469494075735299969552241357603424222 59061068512044369
b (in hex)	5ac635d8 aa3a93e7 b3ebbd55 769886bc 651d06b0 cc53b0f6 3bce3c3e 27d2604b
G_x (in hex)	6b17d1f2 e12c4247 f8bce6e5 63a440f2 77037d81 2deb33a0 f4a13945 d898c296
G_y (in hex)	4fe342e2 fe1a7f9b 8ee7eb4a 7c0f9e16 2bce3357 6b315ece cbb64068 37bf51f5

Table 5.1: The parameters of NIST Curve **P-256**

where $a = 0$ or 1 for Koblitz curves. The cofactor is $f = 2$ for pseudorandom curves, and is $f = 2$ if $a = 1$ and $f = 4$ if $a = 0$ for Koblitz curves. Pseudorandom curves are labeled by **B-163**, **B-233**, **B-283**, **B-409** and **B-571** for pseudorandom curves, and **K-163**, **K-233**, **K-283**, **K-409** and **K-571** for Koblitz curves. The parameters of the Koblitz curves **K-283** are given in Table 5.2.

The Curve	**K-283** - $f(t) = t^{283} + t^{12} + t^7 + t^5 + 1$
a	0
r	3885337784451458141838923813647037813284811733793061324295874997529815829704422603873
	Polynomial Basis
G_x (in hex)	503213f 78ca4488 3f1a3b81 62f188e5 53cd265f 23c1567a 16876913 b0c2ac24 58492836
G_y (in hex)	1ccda38 0f1c9e31 8d90f95d 07e5426f e87e45c0 e8184698 e4596236 4e341161 77dd2259
	Normal Basis
G_x (in hex)	3ab9593 f8db09fc 188f1d7c 4ac9fcc3 e57fcd3b db15024b 212c7022 9de5fcd9 2eb0ea60
G_y (in hex)	2118c47 55e7345c d8f603ef 93b98b10 6fe8854f feb9a3b3 04634cc8 3a0e759f 0c2686b1

Table 5.2: The parameters of NIST Curve **K-283**

The parameters of all NIST curves can be found in [22].

5.2 Elliptic Curve Only Hash: ECOH

Elliptic Curve Only Hash [9], namely ECOH , is proposed to SHA-3 contest of NIST by Dan R. L. Brown in 2008. It is based on BM's hash function MuHASHand it uses elliptic curve groups as DLP-hard groups. However the function which

maps the message blocks to the group elements is not ideal in sense of MuHASH.

In ECOH, the message is first padded to N and then parsed into the blocks $N_0, N_1, \ldots, N_{k-1}$. Then the blocks are mapped to the elliptic curve points with their indices and some extra bitstrings called *counters*. Finally the points are added to get a pre-hash point Q and then it is mapped to a resulting hash value which has a standard size. ECOH uses the curves over binary fields.

There are four ECOH hash functions called ECOH-224, ECOH-256, ECOH-384 and ECOH-512 where the length of each hash value is 224-bit, 256-bit, 384-bit and 512-bit, respectively. The parameters of ECOH hash functions are listed in Table 5.3.

ECOH algorithms	NIST-recommended elliptic curve E	Block Length *blen*	Index Length *ilen*	Counter Length *clen*
ECOH-224	B-283	128-bit	64-bit	64-bit
ECOH-256	B-283	128-bit	64-bit	64-bit
ECOH-384	B-409	192-bit	64-bit	64-bit
ECOH-512	B-571	256-bit	128-bit	128-bit

Table 5.3: Parameters of ECOH hash functions

Remember that the curves B-283, B-409 and B-571 use the binary fields $\mathbb{F}_{2^{283}}$, $\mathbb{F}_{2^{409}}$ and $\mathbb{F}_{2^{571}}$ respectively. All the points corresponding to blocks are in the subgroup generated by the base point G which is determined by NIST.

All ECOH hash functions have the same generic algorithm:

1. **Padding and Parsing.** Let $N = M||1||0^j$ where j is the smallest non-negative integer such that $blen \mid len(N)$. Then parse N into k blocks $N = N_0 N_1 \ldots N_{k-1}$ where $k = \frac{len(N)}{blen}$.

2. **Concatenating with Indices.** Append the bitstrings I_i to the blocks N_i, where I_i is the *ilen*-bit representation of the integer i for $i = 0 \ldots k-1$. Denote them by O_i, i.e. $O_i = N_i||I_i$.

3. Let $O_k = \left(\bigoplus_{i=0}^{k-1} N_i\right) || I_{len(M)}$ where $I_{len(M)}$ is the $ilen$-bit representation of the length $len(M)$ of the message M.

4. **Mapping blocks to points** Let $P_i = (x_i, y_i)$ be the point in the subgroup generated by G so that $X_i = 0^{m-(blen+ilen+clen)} || O_i || C_i$ is the m-bit bitstring representation of x_i for the smallest integer represented by $clen$-bit bitstring C_i, and the rightmost bit of y_i/x_i equals the leftmost bit of N_i.

5. **Combining Operation.** Let $Q = \sum_{i=0}^{k} P_i$ be the pre-hash value.

6. **Hash Output.** Output the n-bit representation of $\lfloor x(Q + \lfloor x(Q)/2 \rfloor G)/2 \rfloor \bmod 2^n$ where $x(P)$ is the integer value of x-coordinate of a point P.

The randomizer function in ECOH works as try-and-increment method: it starts with the bitstring $O_i||I_i||00\ldots0$ and increments it by one in each step until it finds a point in the subgroup of the curve. One can notice that the message blocks can be easily seen in the bitstring representations of x-coordinates of the corresponding points. Also it is noticed that the leftmost bit of N_i is used to determine which y_i is chosen since there are two choices for each x_i.

ECOH loses its incrementality property at the end of hash output operation. Therefore it can remain incremental if this step is ignored.

5.3 The Ferguson-Halcrow Second Preimage Attack on ECOH

In second preimage attacks, a message and its hash value is given and it is aimed to find another message with same hash value. In this attack, assume that the prehash value Q for a message is given.

This attack works for messages M so that $blen \mid len(M)$. The main idea in the attack is setting the blocks of the message so that their checksum is equal to zero bitstring $00\ldots0$. Then a collision is searched in two lists consisting of elliptic curve points corresponding to such messages.

For a message M where $blen \mid mlen$, M can be parsed into blocks $M = M_0 M_1 \dots M_{k-1}$. Then the padded message N can be parsed into blocks $N = N_0 \dots N_{k-1} N_k$ where $N_i = M_i$ for $i = 0, 1, \dots, k-1$ and $N_k = \mathsf{10}^{blen-1}$. The bitstrings O_i is the concatination of blocks and indices, i.e. $O_i = M_i || I_i$ for $i = 0, 1, \dots, k-1$ and $O_k = \mathsf{10}^{blen-1} || I_k$. The extra bitstring O_{k+1} determined by the checksum and the length of the message is $O_{k+1} = \mathsf{0}^{blen} || I_{kblen}$ if the checksum of the message blocks is set to the bitstring $\mathsf{0}^{blen}$, in other words

$$\bigoplus_{i=0}^{k-1} N_k = \bigoplus_{i=0}^{k-1} M_k = \mathsf{0}^{blen}.$$

Let P_i be the point corresponding to the message block M_i for $i = 0, 1, \dots, k-1$ via O_i; $P_{padding}$ be the point corresponding the padding block via O_k; and P_{cl} be the point corresponding to O_{k+1} determined by checksum and the message length. Then the prehash value of a message M is

$$\sum_{i=0}^{k-1} P_i + P_{padding} + P_{cl}.$$

In that sense, the last two points $P_{padding}$ and P_{cl} are always fixed if the checksum and the messege length is fixed.

In Ferguson and Hallcrow's attack, the number of message blocks is fixed to 6, i.e. $M = M_0 M_1 M_2 M_3 M_4 M_5$. Then two lists L_1 and L_2 are prepared as follows: Choose K different random values for (M_0, M_1) and define $M_2 := M_0 \oplus M_1$. Then compute the sum of corresponding points P_0, P_1 and P_2 and store it in list L_1, where P_0, P_1 and P_2 are the corresponding points to blocks M_3, M_4 and M_5, respectively. On the other hand, choose again K different random values for (M_3, M_4) and define $M_5 := M_3 \oplus M_4$. Then compute the point $Q - P_3 - P_4 - P_5 - P_{padding} - P_{cl}$ and store them in list L_2, where P_3, P_4 and P_5 are the corresponding points to blocks M_3, M_4 and M_5, respectively. Note that both of $P_{padding}$ and P_{cl} are fixed since padding occurs as a block itself in the padded message and checksum of the message blocks is 0 bitstring, i.e.

$$\bigoplus_{i=0}^{5} M_k = \mathsf{0}^{blen}.$$

If there is a match between these two lists L_1 and L_2, namely $(M_0^{(j)}, M_1^{(j)})$ from list L_1 and $(M_2^{(k)}, M_3^{(k)})$ from list L_2 , then the message

$$M = M_0^{(j)} M_1^{(j)} M_2^{(j)} M_3^{(k)} M_4^{(k)} M_5^{(k)}$$

is a second preimage for the hash value Q, because

$$P_1^{(j)} + P_2^{(j)} + P_3^{(j)} = Q - P_4^{(k)} - P_5^{(k)} - P_6^{(k)} - P_{padding} - P_{cl}$$

implies that

$$\mathsf{ECOH}\ (M) = P_1^{(j)} + P_2^{(j)} + P_3^{(j)} + P_4^{(k)} + P_5^{(k)} + P_6^{(k)} + P_{padding} + P_{cl} = Q.$$

The attack complexity is $2K$ computations where

$$K \approx \sqrt{\text{number of points on the elliptic curve}}$$

For this reason, it has complexity 2^{143} for ECOH -224 and ECOH -256, 2^{206} for ECOH -384 and 2^{287} for ECOH -512. However, the expected security level is 2^{224} for ECOH -224, 2^{256} for ECOH -256, 2^{384} for ECOH -384, and 2^{512} for ECOH -512. Therefore, ECOH is not a second preimage resistant hash function.

5.4 ECOH2

Ferguson-Halcrow attack is successful because the number of points on the elliptic curve where ECOH is defined makes the attack complexity less than expected security level. For this reason, ECOH is directly updated to ECOH2 by doubling the elliptic curve size, in other words message blocks are mapped to the points whose coordinates are in $\mathbb{F}_{2^{2m}}$. In that sense, efficiency is not too adversely affected, and is indeed potentially improved for two reasons: more message bits are used per point, and elliptic curve twists are used to lessen the number of attempted points per message.

In ECOH2 , the parameter m is replaced by setting d as $d = 4m$. The parameters of ECOH2 hash functions are listed in Table 5.4.

ECOH2 algorithms	NIST-recommended elliptic curve E	Block Length *blen*	Index Length *ilen*	Counter Length *clen*
ECOH2-224	B-283	384-bit	64-bit	64-bit
ECOH2-256	B-283	384-bit	64-bit	64-bit
ECOH2-384	B-409	640-bit	64-bit	64-bit
ECOH2s-512	B-571	768-bit	128-bit	128-bit

Table 5.4: Parameters of ECOH2 hash functions

The generic algorithm of ECOH2 is same as ECOH 's algorithm except the step mapping to points: Let $P_i = (x_i, y_i)$ be the point in the subgroup generated by G so that $X_i = 0^{d-(blen+ilen+clen)}||O_i||C_i$ is the m-bit bitstring representation of x_i for the smallest integer represented by *clen*-bit bitstring C_i, and the rightmost bit of y_i/x_i equals the leftmost bit of N_i.

5.5 Security of ECOH and ECOH 2

The main security argument of ECOH and ECOH2 is the problem of finding points $P_1, P_2, \ldots, P_k$ such that

$$P_1 + P_2 + \ldots + P_k = Q$$

for a given point Q. However, both of ECOH and ECOH2 works on the subgroup of a standard elliptic curve generated by a base point G. Therefore, each point P_i can be written as a_iG and the point Q can be written as aG for some integers $a_1, a_2, \ldots, a_k, a$. In that sense the equation transforms to solving

$$a_1 + a_2 + \ldots + a_k = a \mod n$$

where n is the order of the point G. However, the discrete logarithm problem must be easy to find such an equation and, in elliptic curves, DLP is especially harder.

Semaev [23] reduce he problem of finding a number of points $P_1, P_2, \ldots, P_n$ such that

$$P_1 + P_2 + \ldots + P_k = Q$$

to a new one: the problem of finding bounded solutions to some explicit modular multivariate polynomial equations which arise from the elliptic curve summation polynomials. He introduces polynomials called **Semaev's summation polynomials**:

Definition 32 ([23])**.** *Let E be the elliptic curve of points (x, y) satisfying the equation $y^2 = x^3 + ax + b$ over a field field K with $char(K) \neq 2, 3$ for some $a, b \in K$. For any integer $n \geq 2$, the polynomial*

$$f_n(x_1, x_2, \ldots, x_n)$$

*is a **summation polynomial** if for any $x_1, x_2, \ldots, x_n \in \overline{K}$*

$$f_n(x_1, x_2, \ldots, x_n) = 0$$

if and only if there exist $y_1, y_2, \ldots, y_n \in \overline{K}$ such that the points (x_i, y_i) are on E and

$$(x_1, y_1) + (x_2, y_2) + \ldots + (x_n, y_n) = \mathcal{O}$$

in $E(\overline{K})$.

Theorem 12 ([23])**.** *Let $f_n(x_1, x_2, \ldots, x_n)$ be a summation polynomial defined as above. Then*

$$f_2(x_1, x_2) = x_1 - x_2$$

for $n = 2$,

$$\begin{aligned} f_3(x_1, x_2, x_3) &= (x_1 - x_2)^2 x_3^2 - 2[(x_1 + x_2)(x_1 x_2 + a) + 2b)x_3] \\ &\quad + [(x_1 x_2 - a)^2 - 4b(x_1 + x_2)] \end{aligned}$$

for $n = 3$, and

$$f_n(x_1, x_2, \ldots, x_n) = Res_x\left(f_{n-k}(x_1, \ldots, x_{n-k-1}, x), f_{k+2}(x_{n-k}, \ldots, x_n, x)\right)$$

for $n \geq 4$ and $n - 3 \geq k \geq 1$.

Proof. The proof for $n = 2$ is done here. For the proof for and $n = 3$ and $n \geq 4$, see [23].

For $n = 2$, we have

$$\begin{aligned}(x_1, y_1) + (x_2, y_2) = \mathcal{O} &\iff (x_1, y_1) = (x_2, -y_2) \\ &\iff x_1 = x_2, y_1 = -y_2\end{aligned}$$

Therefore, for any $x_1, x_2 \in \overline{K}$, one can define $f_2(x_1, x_2) = x_1 - x_2$. When $f_2(x_1, x_2) = 0$, the roots y_1, y_2 of the polynomial $y^2 = x_1^3 + ax_1 + b$ satisfy $y_1 = -y_2$. □

We further have the following theorem.

Theorem 13 ([23])**.** *Let $f_n(x_1, x_2, \ldots, x_n)$ be a summation polynomial defined as above. Then the polynomial f_n have the following properties:*

i) The polynomial f_n is symmetric and of degree 2^{n-2} in each variable x_i for any $n \geq 3$.

ii) The polynomial f_n is absolutely irreducible and

$$f_n(X_1, \ldots, X_n) = f_{n-1}^2(X_1, \ldots, X_{n-1})X_n^{2^{n-2}} + \ldots$$

for any $n \geq 3$

Proof. See [23]. □

Semaev finds a relation between solving the discrete logarithm problem in elliptic curves and finding solutions for the Semaev's summation polynomials. He gives an algorithm to find bounded solutions for summation polynomial. However, it remains computationally hard to find such solutions.

Chapter 6

Conclusion

In this thesis, we surveyed the new concept of Incremental Hash Functions in cryptography. We gave the construction idea, the efficiency benefits such as incrementality and paralllelizability.

Morover, some examples of incremental hash functions are introduced, especially ECOH . A hash function on elliptic curves is not a standard construction and this makes ECOH interesting. It becomes more interesting because it uses incrementality property with MuHASHstructure.

In incremental hash functions, we only change the blocks, but we do not insert or delete blocks or bitstrings. This is not studied well in cryptography and left as an open problem.

In all the constructions of the incremental hash functions, the emphasis is given to the combining operation. One may seek to find efficient randomizer functions satisfying security levels so that the security of the incremental hash functions depends only to the computationally hard problems of the underlying group. On the other hand, the complexity of algorithms solving computationally hard problems can be reduced in the future. It is also desirable to search for new group structures for added security and efficiency.

Bibliography

[1] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*. CRC Press, Inc., 1st ed., 1996.

[2] National Institute of Standards and Technology, *FIPS PUB 180-4: Secure Hash Standard*. Mar. 2012.

[3] B. Kaliski, "The MD2 Message-Digest Algorithm." RFC 1319 (Historic), Apr. 1992. Obsoleted by RFC 6149.

[4] R. L. Rivest, "The MD4 Message Digest Algorithm," in *CRYPTO*, pp. 303–311, 1990.

[5] R. Rivest, "The MD5 Message-Digest Algorithm." RFC 1321 (Informational), Apr. 1992. Updated by RFC 6151.

[6] H. Dobbertin, A. Bosselaers, and B. Preneel, "RIPEMD-160: A Strengthened Version of RIPEMD," in *FSE*, pp. 71–82, 1996.

[7] M. Bellare and D. Micciancio, "A new paradigm for collision-free hashing: Incrementality at reduced cost," in *Advances in Cryptology—EUROCRYPT 97* (W. Fumy, ed.), vol. 1233 of *Lecture Notes in Computer Science*, pp. 163–192, Springer-Verlag, 11–15 May 1997.

[8] M. Bellare, O. Goldreich, and S. Goldwasser, "Incremental cryptography: The case of hashing and signing," in *CRYPTO*, pp. 216–233, 1994.

[9] D. R. L. Brown, A. Antipa, M. Campagna, and R. Struik, "ECOH: the Elliptic Curve Only Hash." Submission to NIST, 2008.

[10] D. S. Dummit and R. M. Foote, *Abstract Algebra (3rd Edition)*. John Wiley and Sons, 2004.

[11] J. Jenkins, "Character sets," in *Encyclopedia of Language & Linguistics (2nd Edition)* (K. Brown, ed.), pp. 296–299, Oxford: Elsevier, second edition ed., 2006.

[12] American Standards Association, "American Standard Code for Information Interchange," *ASA X3.4*, 1963.

[13] V. Cerf, "ASCII format for network interchange." RFC 20, Oct. 1969.

[14] S. Josefsson, "The Base16, Base32, and Base64 Data Encodings." RFC 4648 (Proposed Standard), Oct. 2006.

[15] M. Bellare, O. Goldreich, and S. Goldwasser, "Incremental cryptography and application to virus protection," in *STOC*, pp. 45–56, 1995.

[16] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *ACM Conference on Computer and Communications Security*, pp. 62–73, 1993.

[17] R. Impagliazzo and M. Naor, "Efficient cryptographic schemes provably as secure as subset sum," *J. Cryptology*, vol. 9, no. 4, pp. 199–216, 1996.

[18] D. Chaum, E. van Heijst, and B. Pfitzmann, "Cryptographically strong undeniable signatures, unconditionally secure for the signer," in *CRYPTO*, pp. 470–484, 1991.

[19] M. Ajtai, "Generating hard instances of lattice problems (extended abstract)," in *STOC*, pp. 99–108, 1996.

[20] J. Rosser and L. Schoenfeld, "Approximate formulas for some functions of prime numbers," in *Illinois Journal of Math Vol. 6*, 1962.

[21] L. C. Washington, *Elliptic Curves: Number Theory and Cryptography, Second Edition*. Chapman & Hall/CRC, 2 ed., 2008.

[22] National Institute of Standards and Technology, *FIPS PUB 186-3: Digital Signature Standard (DSS)*. June 2009.

[23] I. Semaev, "Summation polynomials and the discrete logarithm problem on elliptic curves," *IACR Cryptology ePrint Archive*, vol. 2004, p. 31, 2004.

Appendix A

Elliptic Curves proposed by NIST

The principal parameters are the elliptic curve E defined with an equation over a finite field and a designated point $G = (G_x, G_y) \in E$ called *the base point*. The base point has order r which is a large prime. The number of points on the curve is $n = fr$ for some integer f (the cofactor) not divisible by r. For efficiency reasons, it is desirable to take the cofactor to be as small as possible. All of the NIST curves have cofactors 1, 2, or 4.

A.1 Elliptic Curves over Prime Fields

The underlying finite fields are prime field $\mathbb{F}_p$ for prime number p and binary fields $\mathbb{F}_{2^m}$ for some positive integer m. NIST choose the primes of length 192-bit, 224-bit, 256-bit, 384-bit, 521-bit for prime fields, and $m = 163, 233, 283, 409, 571$ for binary fields.

Over the prime fields, the curve satisfying the equation

$$E : y^2 = x^3 - 3x + b$$

is chosen, i.e. $a = -3$. The coefficient b and prime p is chosen carefully to satisfy that E is of prime order r and so the cofactor $f = 1$. The parameters of the NIST curves given below.

The Curve	**P-192**
p	6277101735386680763835789423207666416083908700390324961279
r	6277101735386680763835789423176059013767194773182842284081
b	`64210519 e59c80e7 0fa7e9ab 72243049 feb8deec c146b9b1`
G_x	`188da80e b03090f6 7cbf20eb 43a18800 f4ff0afd 82ff1012`
G_y	`07192b95 ffc8da78 631011ed 6b24cdd5 73f977a1 1e794811`
The Curve	**P-224**
p	6277101735386680763835789423207666416083908700390324961279
r	6277101735386680763835789423176059013767194773182842284081
b	`64210519 e59c80e7 0fa7e9ab 72243049 feb8deec c146b9b1`
G_x	`188da80e b03090f6 7cbf20eb 43a18800 f4ff0afd 82ff1012`
G_y	`07192b95 ffc8da78 631011ed 6b24cdd5 73f977a1 1e794811`

Table A.1: Parameters of **P-192** and **P-224** Curves

The Curve	**P-256**
p	1157920892103562487626974469494075735300861434152903141955336 31308867097853951
r	1157920892103562487626974469494075735299969552241357603424222 59061068512044369
b	`5ac635d8 aa3a93e7 b3ebbd55 769886bc 651d06b0 cc53b0f6 3bce3c3e 27d2604b`
G_x	`6b17d1f2 e12c4247 f8bce6e5 63a440f2 77037d81 2deb33a0 f4a13945 d898c296`
G_y	`4fe342e2 fe1a7f9b 8ee7eb4a 7c0f9e16 2bce3357 6b315ece cbb64068 37bf51f5`
The Curve	**P-384**
p	3940200619639447921227904010014361380507973927046544666794829340424572177149687032904726608825893800186160697311231 9
r	3940200619639447921227904010014361380507973927046544666794690527962765939911326356939895630815229491355443365394264 3
b	`b3312fa7 e23ee7e4 988e056b e3f82d19 181d9c6e fe814112 0314088f 5013875a c656398d 8a2ed19d 2a85c8ed d3ec2aef`
G_x	`aa87ca22 be8b0537 8eb1c71e f320ad74 6e1d3b62 8ba79b98 59f741e0 82542a38 5502f25d bf55296c 3a545e38 72760ab7`
G_y	`3617de4a 96262c6f 5d9e98bf 9292dc29 f8f41dbd 289a147c e9da3113 b5f0b8c0 0a60b1ce 1d7e819d 7a431d7c 90ea0e5f`

Table A.2: Parameters of **P-256** and **P-384** Curves

The Curve	**P-521**
p	6864797660130609714981900799081393217269435300143305409394463459185543183397656052122559640661454554977296311391480858037121987999716643812574028291115057151
r	6864797660130609714981900799081393217269435300143305409394463459185543183397655394245057746333217197532963996371363321113864768612440380340372808892707005449
b	051 953eb961 8e1c9a1f 929a21a0 b68540ee a2da725b 99b315f3 b8b48991 8ef109e1 56193951 ec7e937b 1652c0bd 3bb1bf07 3573df88 3d2c34f1 ef451fd4 6b503f00
G_x	c6 858e06b7 0404e9cd 9e3ecb66 2395b442 9c648139 053fb521 f828af60 6b4d3dba a14b5e77 efe75928 fe1dc127 a2ffa8de 3348b3c1 856a429b f97e7e31 c2e5bd66
G_y	118 39296a78 9a3bc004 5c8a5fb4 2c7d1bd9 98f54449 579b4468 17afbd17 273e662c 97ee7299 5ef42640 c550b901 3fad0761 353c7086 a272c240 88be9476 9fd16650

Table A.3: Parameters of **P-521** Curve

A.2 Elliptic Curves over Binary Fields

Over the binary fields, two types of curves called pseudo-random curve and Koblitz curve. Pseudo-random curve satisfies the equation

$$E_b : y^2 + xy = x^3 + x^2 + b$$

and Koblitz curve satisfies the equation

$$E_a : y^2 + xy = x^3 + ax^2 + 1$$

for $a = 0$ or 1. The cofactor is $f = 2$ for pseudorandom curves, and it is $f = 2$ if $a = 1$ and $f = 4$ if $a = 0$ for Koblitz curves. The parameters of the binary curves are given below.

The Curve	**K-163** - $f(t) = t^{163} + t^7 + t^6 + t^3 + 1$
a	1
r	5846006549323611672814741753598448348329118574063
	Polynomial Basis
G_x	`2 fe13c053 7bbc11ac aa07d793 de4e6d5e 5c94eee8`
G_y	`2 89070fb0 5d38ff58 321f2e80 0536d538 ccdaa3d9`
	Normal Basis
G_x	`0 5679b353 caa46825 fea2d371 3ba450da 0c2a4541`
G_y	`2 35b7c671 00506899 06bac3d9 dec76a83 5591edb2`

Table A.4: Parameters of **K-163** Curve

The Curve	**B-163** - $f(t) = t^{163} + t^7 + t^6 + t^3 + 1$
r	5846006549323611672814742442876390689256843201587
	Polynomial Basis
b	`2 0a601907 b8c953ca 1481eb10 512f7874 4a3205fd`
G_x	`3 f0eba162 86a2d57e a0991168 d4994637 e8343e36`
G_y	`0 d51fbc6c 71a0094f a2cdd545 b11c5c0c 797324f1`
	Normal Basis
b	`6 645f3cac f1638e13 9c6cd13e f61734fb c9e3d9fb`
G_x	`0 311103c1 7167564a ce77ccb0 9c681f88 6ba54ee8`
G_y	`3 33ac13c6 447f2e67 613bf700 9daf98c8 7bb50c7f`

Table A.5: Parameters of **B-163** Curve

The Curve	**K-233** - $f(t) = t^{233} + t^{74} + 1$
a	0
r	3450873173395281893717377931138512760570940988862252126328087024741343
	Polynomial Basis
G_x	`172 32ba853a 7e731af1 29f22ff4 149563a4 19c26bf5 0a4c9d6e efad6126`
G_y	`1db 537dece8 19b7f70f 555a67c4 27a8cd9b f18aeb9b 56e0c110 56fae6a3`
	Normal Basis
G_x	`0fd e76d9dcd 26e643ac 26f1aa90 1aa12978 4b71fc07 22b2d056 14d650b3`
G_y	`064 3e317633 155c9e04 47ba8020 a3c43177 450ee036 d6335014 34cac978`

Table A.6: Parameters of **K-233** Curve

The Curve	**B-233** - $f(t) = t^{233} + t^{74} + 1$
r	6901746346790563787434755862277025555839812737345013555379383634485463
	Polynomial Basis
b	`066 647ede6c 332c7f8c 0923bb58 213b333b 20e9ce42 81fe115f 7d8f90ad`
G_x	`0fa c9dfcbac 8313bb21 39f1bb75 5fef65bc 391f8b36 f8f8eb73 71fd558b`
G_y	`100 6a08a419 03350678 e58528be bf8a0bef f867a7ca 36716f7e 01f81052`
	Normal Basis
b	`1a0 03e0962d 4f9a8e40 7c904a95 38163adb 82521260 0c7752ad 52233279`
G_x	`18b 863524b3 cdfefb94 f2784e0b 116faac5 4404bc91 62a363ba b84a14c5`
G_y	`049 25df77bd 8b8ff1a5 ff519417 822bfedf 2bbd7526 44292c98 c7af6e02`

Table A.7: Parameters of **B-233** Curve

The Curve	**K-283** - $f(t) = t^{283} + t^{12} + t^7 + t^5 + 1$
a	0
r	3885337784451458141838923813647037813284811733793061324295874997529815829704422603873
	Polynomial Basis
G_x	503213f 78ca4488 3f1a3b81 62f188e5 53cd265f 23c1567a 16876913 b0c2ac24 58492836
G_y	1ccda38 0f1c9e31 8d90f95d 07e5426f e87e45c0 e8184698 e4596236 4e341161 77dd2259
	Normal Basis
G_x	3ab9593 f8db09fc 188f1d7c 4ac9fcc3 e57fcd3b db15024b 212c7022 9de5fcd9 2eb0ea60
G_y	2118c47 55e7345c d8f603ef 93b98b10 6fe8854f feb9a3b3 04634cc8 3a0e759f 0c2686b1

Table A.8: Parameters of **K-283** Curve

The Curve	**B-283** - $f(t) = t^{283} + t^{12} + t^7 + t^5 + 1$
r	7770675568902916283677847627294075626569625976904889109196526770044277787378692871
	Polynomial Basis
b	27b680a c8b8596d a5a4af8a 19a0303f ca97fd76 45309fa2 a581485a f6263e31 3b79a2f5
G_x	5f93925 8db7dd90 e1934f8c 70b0dfec 2eed25b8 557eac9c 80e2e198 f8cdbecd 86b12053
G_y	3676854 fe24141c b98fe6d4 b20d02b4 516ff702 350eddb0 826779c8 13f0df45 be8112f4
	Normal Basis
b	157261b 894739fb 5a13503f 55f0b3f1 0c560116 66331022 01138cc1 80c0206b dafbc951
G_x	749468e 464ee468 634b21f7 f61cb700 701817e6 bc36a236 4cb8906e 940948ea a463c35d
G_y	62968bd 3b489ac5 c9b859da 68475c31 5bafcdc4 ccd0dc90 5b70f624 46f49c05 2f49c08c

Table A.9: Parameters of **B-283** Curve

The Curve	**K-409** - $f(t) = t^{409} + t^{87} + 1$
a	0
r	3305279843951242994759576540163855199142023414821406096423243950228807112892491910506732584577774580140963665906177313586 71
	Polynomial Basis
G_x	060f05f 658f49c1 ad3ab189 0f718421 0efd0987 e307c84c 27accfb8 f9f67cc2 c460189e b5aaaa62 ee222eb1 b35540cf e9023746
G_y	1e36905 0b7c4e42 acba1dac bf04299c 3460782f 918ea427 e6325165 e9ea10e3 da5f6c42 e9c55215 aa9ca27a 5863ec48 d8e0286b
	Normal Basis
G_x	1b559c7 cba2422e 3affe133 43e808b5 5e012d72 6ca0b7e6 a63aeafb c1e3a98e 10ca0fcf 98350c3b 7f89a975 4a8e1dc0 713cec4a
G_y	16d8c42 052f07e7 713e7490 eff318ba 1abd6fef 8a5433c8 94b24f5c 817aeb79 852496fb ee803a47 bc8a2038 78ebf1c4 99afd7d6

Table A.10: Parameters of **K-409** Curve

The Curve	**B-409** - $f(t) = t^{409} + t^{87} + 1$
r	661055968790248598951915308032771039828404682964281219284648798304157774827374805208143723762179110965979867288366567526771
	Polynomial Basis
b	021a5c2 c8ee9feb 5c4b9a75 3b7b476b 7fd6422e f1f3dd67 4761fa99 d6ac27c8 a9a197b2 72822f6c d57a55aa 4f50ae31 7b13545f
G_x	15d4860 d088ddb3 496b0c60 64756260 441cde4a f1771d4d b01ffe5b 34e59703 dc255a86 8a118051 5603aeab 60794e54 bb7996a7
G_y	061b1cf ab6be5f3 2bbfa783 24ed106a 7636b9c5 a7bd198d 0158aa4f 5488d08f 38514f1f df4b4f40 d2181b36 81c364ba 0273c706
	Normal Basis
b	124d065 1c3d3772 f7f5a1fe 6e715559 e2129bdf a04d52f7 b6ac7c53 2cf0ed06 f610072d 88ad2fdc c50c6fde 72843670 f8b3742a
G_x	0ceacbc 9f475767 d8e69f3b 5dfab398 13685262 bcacf22b 84c7b6dd 981899e7 318c96f0 761f77c6 02c016ce d7c548de 830d708f
G_y	199d64b a8f089c6 db0e0b61 e80bb959 34afd0ca f2e8be76 d1c5e9af fc7476df 49142691 ad303902 88aa09bc c59c1573 aa3c009a

Table A.11: Parameters of **B-409** Curve

The Curve	**K-571** - $f(t) = t^{571} + t^{10} + t^5 + t^2 + 1$
a	0
r	1932268761508629172347675945465993672149463664853217499328617625725759571144780212268133978522706711834706712800825351461273674974066617311929682421617092503555733685276673
	Polynomial Basis
G_x	26eb7a8 59923fbc 82189631 f8103fe4 ac9ca297 0012d5d4 60248048 01841ca4 43709584 93b205e6 47da304d b4ceb08c
G_y	bbd1ba39 494776fb 988b4717 4dca88c7 e2945283 a01c8972 349dc80 7f4fbf37 4f4aeade 3bca9531 4dd58cec 9f307a54 ffc61efc 006d8a2c 9d4979c0 ac44aea7 4fbebbb9 f772aedc b620b01a 7ba7af1b 320430c8 591984f6 01cd4c14 3ef1c7a3
	Normal Basis
G_x	04bb2db a418d0db 107adae0 03427e5d 7cc139ac b465e593 4f0bea2a b2f3622b c29b3d5b 9aa7a1fd fd5d8be6 6057c100
G_y	8e71e484 bcd98f22 bf847642 37673674 29ef2ec5 bc3ebcf7 44cbb57 de20788d 2c952d7b 56cf39bd 3e89b189 84bd124e 751ceff4 369dd8da c6a59e6e 745df44d 8220ce22 aa2c852c fcbbef49 ebaa98bd 2483e331 80e04286 feaa2530 50caff60

Table A.12: Parameters of **K-571** Curve

The Curve	**B-571** - $f(t) = t^{571} + t^{10} + t^5 + t^2 + 1$
r	3864537523017258344695351890931987344298927329706434998657235251451519142289560424536143999389415773083133881121926944486246872462816813070234528288303332411393191105285703
	Polynomial Basis
b	2f40e7e 2221f295 de297117 b7f3d62f 5c6a97ff cb8ceff1 cd6ba8ce 4a9a18ad 84ffabbd 8efa5933 2be7ad67 56a66e29 4afd185a 78ff12aa 520e4de7 39baca0c 7ffeff7f 2955727a
G_x	303001d 34b85629 6c16c0d4 0d3cd775 0a93d1d2 955fa80a a5f40fc8 db7b2abd bde53950 f4c0d293 cdd711a3 5b67fb14
G_y	99ae6003 8614f139 4abfa3b4 c850d927 e1e7769c 8eec2d19 37bf273 42da639b 6dccfffe b73d69d7 8c6c27a6 009cbbca 1980f853 3921e8a6 84423e43 bab08a57 6291af8f 461bb2a8 b3531d2f 0485c19b 16e2f151 6e23dd3c 1a4827af 1b8ac15b
	Normal Basis
b	3762d0d 47116006 179da356 88eeaccf 591a5cde a7500011 8d9608c5 9132d434 26101a1d fb377411 5f586623 f75f0000 1ce61198 3c1275fa 31f5bc9f 4be1a0f4 67f01ca8 85c74777
G_x	0735e03 5def5925 cc33173e b2a8ce77 67522b46 6d278b65 0a291612 7dfea9d2 d361089f 0a7a0247 a184e1c7 0d417866 e0fe0feb 0ff8f2f3 f9176418 f97d117e 624e2015 df1662a8
G_y	04a3642 0572616c df7e606f ccadaecf c3b76dab 0eb1248d d03fbdfc 9cd3242c 4726be57 9855e812 de7ec5c5 00b4576a 24628048 b6a72d88 0062eed0 dd34b109 6d3acbb6 b01a4a97

Table A.13: Parameters of **B-571** Curve