Provably Secure Cryptographic Hash Functions

Maike Massierer

Supervisors: James Franklin and Richard Buckland

School of Mathematics,
The University of New South Wales.

December 2006

Submitted in partial fulfillment of the requirements of the degree of
Bachelor of Science with Honours

Acknowledgements

The work on this thesis was part of a Study Abroad year at the University of New South Wales, Sydney. It has been a very valuable academic and personal experience, and thanks go to the many people who have supported and inspired me throughout this year.

For the support with the work on this thesis, the greatest thanks go to my supervisors Jim Franklin and Richard Buckland. To Jim, for his never-ending supply of time and patience, and for his interest in every aspect of my studies. And to Richard, for his brilliant ideas and guidance. Also to Roland, my inofficial third supervisor, who spent enormous amounts of time helping me with all kinds of smaller and bigger problems.

I would also like to thank Catherine Greenhill for her friendly advice regarding several graph theory questions.

Thank you to Jim, James, Roland and Tara, who willingly offered to help with proof-reading the entire document to fix my English, layout, logic and notation.

Lastly, for their loving support and faithful prayers, thanks must go to my family and friends, and especially to Bec.

Maike Massierer, 13 December 2006

Introduction

A hash function is simply a mapping

$$h : \{0,1\}^* \to \{0,1\}^m$$

from the set of all binary strings to the set of binary strings of a fixed size. Every good hash function has the property that two different inputs are very unlikely to be mapped to the same value.

Hash functions have many applications in computer science, and a particularly interesting area is cryptography. The main goal of cryptography is to provide three basic security characteristics of information: confidentiality, integrity and authentication. Confidentiality is the assurance of data privacy. Integrity is the assurance that data has not been altered. Authentication is the process of verifying an identity. The security of much of our communication today relies on cryptographic protocols that ensure these attributes, and many such protocols use hash functions as building blocks. However, not every hash function is good enough to be used in cryptography. In fact, only so-called *cryptographic hash functions* that fulfil certain security properties may be used in cryptographic applications such as digital signatures and pseudo-random number generators. Cryptographic hash functions must not only have good statistical properties. They must also withstand serious attack by malicious and powerful attackers who are trying to invade our privacy.

The design of such cryptographic hash functions is an important but extremely difficult task. Many have been proposed, but most of them soon turned out to be too weak to resist attacks. Only two families of hash functions came to be widely used (namely the MD and SHA families, the most well-known members of which are MD5 and SHA-1, respectively). Unfortunately, their security relies on heuristic arguments rather than mathematical proofs. As might be expected, weaknesses have recently been found in both of them and as a result, there currently exist no secure and practical cryptographic hash functions. Hence there is little basis for trust in the applications that use them, and a great need for research into good cryptographic hash functions.

These recent developments in cryptanalysis have clearly shown that currently used cryptographic hash functions are not good enough. But research in this area is not only very important because most existing hashes have been broken. The problem is not so much that flaws have been found in current designs, but that their construction often seems ad-hoc and their security cannot be proven.

Information security is too important to be left up to assumptions and luck. What we really need are hash functions the security of which can be trusted.

Much research is being done in the area of provably secure hash functions and many promising designs have been proposed, but none of them are practical enough to be used yet. Speed is especially a problem. Provably secure designs are not nearly as fast as conventional hash functions, which makes them unpopular. Also, proofs of security are extremely difficult to create. They rely on assumptions about the resources and power of possible attackers, and the ideas and definitions of security are anything but consistent. Many different attack models are in use and there is no general understanding of how secure is good enough. Maybe there never can be, since the answer to this question depends on the level of concern. Still, much research is to be done in finding usable definitions of security properties and manageable methods to prove them.

The direction in current research is to relate security properties to provably difficult problems, that is, members of the class of NP-complete problems. If it can be proven that a certain attack on a given hash function is at least as hard as solving a well-known NP-complete problem, then that gives a clear indication of how secure a hash function is against this attack. This might not be the most satisfactory method of proving security, but it is one that actually works in practice. In the past twenty years, many hash functions that are secure in this sense have been constructed and continually improved, and there is hope that they may soon be able to replace the currently used hash algorithms with questionable security.

This thesis looks into concepts used to design provably secure cryptographic hash functions.

Chapter 1 gives a general introduction to cryptographic hash functions by discussing different security properties and possible attacks on them. It also gives some motivation for the need of good hash functions by presenting various cryptographic applications, and it analyses why the design of cryptographic hash functions is so difficult.

Chapter 2 discusses the two most commonly used hash functions MD5 and SHA-1. It presents their algorithms, their security arguments and the recent ground-breaking attacks on them, followed by a brief discussion of why current hash functions are not good enough.

Chapter 3 then looks into methods of designing provably secure hash functions, and Chapter 4 presents a very natural design of a provably secure hash that illustrates these methods.

Finally, Chapter 5 proposes a new type of provably secure hash function with a security assumption that makes it easier to design than other provably secure hash functions. It presents a pioneer design of this kind, based on the graph theoretical Hamiltonian cycle problem, and a proof of its security.

Contents

Chapter 1

Introduction to Hash Functions

The concept of hashing was apparently first used by Hans Peter Luhn of IBM in a memo dated 1953 [24]. The term *hash*, which normally means "to chop and mix", came into use about ten years later and seems to be a very good informal description of what a hash function does: it mixes up the bits of the input and chops some off to produce a shorter and random-looking output.

More formally speaking, a hash function is simply a function that examines input data of arbitrary length and produces an output of fixed length called the *hash value*. Hash functions may have different properties, depending on their use. For all applications, it is important that given two different inputs, they are unlikely to hash to the same value.

There are various applications of hash functions, many of them in computer science, and a very significant one is cryptography. This thesis focuses on *cryptographic hash functions*, which are particularly interesting but also very hard to design, because they must have certain security properties.

This chapter starts by defining (cryptographic) hash functions and their key properties, and then explains the most common methods of attacking these properties. It also gives some motivation by discussing the most important applications, followed by a brief discussion of why it is hard to design such functions.

1.1 Definition of a Hash Function

Notation 1.1.1
For $m \in \mathbb{N}$ let $\{0,1\}^m$ denote the set of all bit strings of length m, and $\{0,1\}^* = \bigcup_{m\in\mathbb{N}}\{0,1\}^m$ the set of all bit strings.

Definition 1.1.2 (hash function)
A **hash function** *h takes an input of arbitrary length and computes an output of a fixed length m, called the* **hash value**. *In cryptography, the input is a* **message** *in the form of a bit string and the output is called a* **message digest**. *Hence h can be written*

$$h : \{0,1\}^* \to \{0,1\}^m.$$

Since a hash function is usually expected to make its input smaller, not bigger, the size of the input is often required to be greater than or equal to m. Typical values for m are 128, 160 and 256.

1.2 Desired Properties of Cryptographic Hash Functions

Although the official definition of a hash function only requires it to map its arbitrary-sized input to a fixed-size output, every good hash function is expected to produce few hash collisions. Given two inputs, it should be very unlikely (though obviously not impossible) that they are mapped to the same value. This property suffices for many applications. However, there are also many applications that require further properties of the hash functions they use (for applications, see Section 1.4).

A very interesting area where hash functions find many applications is cryptography. Cryptography generally ensures the "security" (meaning confidentiality, authentication, integrity) of certain transactions of communication, and it always assumes that some malicious entity exists that is trying to "break" the security. Any method used in cryptography must therefore still achieve the desired security under attack. That is also true for the hash functions used in cryptographic protocols, they must withstand serious attack. To ensure this, it is required that they satisfy a variety of security properties.

Hash functions used for cryptographic purposes are called **cryptographic hash functions**. Since this thesis is only about cryptographic hash functions, they will from now on simply be referred to as hash functions. Ordinary hash functions that are used for purposes other than cryptography will be called **non-cryptographic hash functions**.

There are three basic properties that cryptographic hash functions may fulfil, but not every cryptographic hash function must satisfy all of them. While the first property (preimage resistance) is always required, the other two are only necessary for some applications in cryptography. These understandings and definitions are not consistent throughout the literature. The ones chosen here seem to be the be most common and most useful for the purpose of this thesis.

Definition 1.2.1 (preimage resistance)
A hash function $h : \{0,1\}^* \to \{0,1\}^m$ is called **preimage resistant** *if given a digest $d \in \{0,1\}^m$ it is* hard *to find a message (a preimage) $M \in \{0,1\}^*$ such that $h(M) = d$. This is often also referred to as h is* **one-way**.

Definition 1.2.2 (second preimage resistance)
A hash function $h : \{0,1\}^* \to \{0,1\}^m$ is called **second preimage resistant** *if given a message $M_1 \in \{0,1\}^*$ it is* hard *to find another message $M_2 \in \{0,1\}^*$ (a second preimage) such that $h(M_1) = h(M_2)$.*

Definition 1.2.3 (collision resistance)
A hash function $h : \{0,1\}^* \rightarrow \{0,1\}^m$ is called **collision resistant** *if it is* hard *to find two messages M_1 and $M_2 \in \{0,1\}^*$ (with $M_1 \neq M_2$) such that $h(M_1) = h(M_2)$. Say M_1 and M_2* **collide**.

Notice that collision resistance is a stronger requirement than second preimage resistance. In the definition of second preimage resistance, one of the messages that collide is *fixed*, whereas with collision resistance, both messages can be chosen *arbitrarily*. Therefore if a hash function is collision resistant it is automatically second preimage resistant, but the converse of this statement is not true.

The big question is what is meant by "hard". Obviously, preimages and collisions always exist. But it should be *hard* to produce them on purpose, even by someone who puts in a great deal of effort. Informally speaking, it should be impossible to write an algorithm that performs a *hard* operation efficiently. For example, even if all the computational power in the world and that of the foreseeable future and a huge amount of time (e.g. the age of the universe) were available for computation, there should still be no algorithm that calculates a preimage for a given hash value. There may exist (and in our case there will always exist) an algorithm that solves the problem in theory (whereas there are problems for which it is *impossible* to write an algorithm which finds a solution, e.g. the Halting Problem). For example, a preimage to a given hash can always be found by trying enough different possibilities. But this must not be feasible in practice, for example because the computational power and time required to try enough possible preimages to find the right one would exceed the available resources.

How something like this can be defined more precisely and how it can be achieved is the most important question in cryptographic theory and the topic of this thesis. How do we know how hard something is and how hard is hard enough? How can we be sure something is really as hard as we think it is? Some precise mathematical definitions are necessary to quantify "hardness", and mathematical proofs are highly desirable to give assurance. Chapter 3 makes an effort to shed some light on these questions.

In addition to the three key properties defined above, there are other properties that cryptographic hash functions may or should fulfil. These are not as precisely defined and it is often sufficient to do some analysis or experiments to ensure that they are satisfied, rather than looking for proofs.

Hash functions are always expected to be **efficiently computable**. Most applications require the computation of hash values to be very fast, because many evaluations of the function must be performed. For cryptographic hash functions there is often a trade-off between security and speed, and unfortunately many are willing to sacrifice security for speed.

Hash functions are also **deterministic**, which means that given a particular input, the function always computes the same output.

In spite of being deterministic and efficiently computable, cryptographic hash functions are usually expected to behave like a **random** function, in that it should be impossible to predict any output bits given a particular input without actually applying the function. In fact, it is even desired that adjacent bit strings have completely different hashes. This is called the **avalanche effect** and more precisely stated as follows. The avalanche effect is evident if, when an input to a hash function is changed slightly, the output changes significantly. In other words, even when only one bit of the input is flipped, each output bit should flip with probability one-half. Also, hash functions should always produce a "random-looking" output.

Two types of cryptographic hash functions are usually distinguished. A hash function that only has one input is what is often just referred to as **hash function** and sometimes called **Manipulation Detection Code (MDC)**. This is the type of hash function that this thesis is about, so when talking about hash functions we mean single-input hash functions. But for completeness we mention that there are also hash functions which take two inputs, one of them a message as before and the other one a secret key. These functions are called **keyed hash functions**, an important subclass of which are **Message Authentication Codes (MACs)** (see Section 1.4). Keyed hash functions are sometimes constructed from ordinary hash functions by concatenating the message with the key to form the input to an ordinary hash function.

Note that there are also constructions of hash functions where the second parameter is public. An example of this class are Universal One-Way Hash Functions, which will be discussed in Chapter 4. These are not keyed hash functions in the sense discussed above.

1.3 Attacks on Cryptographic Hash Functions

The main difference between non-cryptographic hash functions and cryptographic hash functions is that cryptographic hash functions must withstand serious attack. For example, non-cryptographic hash functions are expected to produce few hash collisions in normal use. In contrast to that, cryptographic hash functions are expected to produce no collisions even when a malicious attacker deliberately tries to create them using all his/her mathematical knowledge, computational power and any other resources available to him/her. This would be called a *collision attack*. Similarly, an attack that tries to produce a preimage of a hash value (under a certain hash function) is called a *preimage attack* on that hash function, and an attack that attempts to create a second preimage is called a *second preimage attack*. These are the three properties of hash functions which may be attacked (and therefore it is desirable to somehow prove that such attacks are hard). The other properties mentioned in Section 1.2 such as efficiency, determinism, randomness and avalanche effect are desirable properties of hash functions, but they are not attacked directly. However, poor avalanche effect or missing randomness may allow attacks on one or several of the security properties (in this case, for example, through statistical analysis).

1.3.1 Brute Force Attacks

The conceptually simplest type of attack is a *brute force attack*. Such an attack performs an exhaustive search, that is, it tries a large number of possibilities until it is successful. For example, a brute force preimage attack which tries to find a preimage for a given digest d would try many different messages, hash them, and compare the results to d until it finds a match. The more efficient a hash function is, the more efficient this becomes.

If a hash function has 2^m possible output values (say an m-bit hash and every m-bit string is a possible output, i.e. the hash function is surjective), then one would have to try $\frac{1}{2}\cdot 2^m = 2^{m-1}$ messages on average to find a preimage. One would say the hash function has $m-1$ **bits of security**, or often also m bits of security, since this is only a rough measure anyway. A second preimage attack works similarly. One has to try approximately 2^{m-1} messages to find one that hashes to the same value as a given message. Note that in terms of a brute force attack, there is no difference between preimage and second preimage attacks. In more intelligent attacks, however, the extra knowledge of the first message (for which a collision should be produced) might leave an attacker at a slight advantage.

There is a more clever and efficient way to perform a brute force collision attack. It is based on the *birthday paradox*, a standard statistics problem:

How many people must be in a room for the probability that one of them shares my birthday to be greater than 50%? To answer this question, let $p(m)$ be the probability of at least one out of m people sharing my birthday. Also assume every year has 365 days and all birthdays are equally likely. Then

$$p(m) = 1 - \left(\frac{364}{365}\right)^m .$$

Solving $p(m) > 0.5$ gives that there must be at least 253 people in the room.

How many people must be in a room for the probability that two of them share the same birthday to be greater than 50%? If the probability that this is the case for m people in a room is $q(m)$ then

$$q(m) = 1 - 1 \cdot \left(\frac{364}{365}\right) \cdot \left(\frac{363}{365}\right) \cdot \left(\frac{362}{365}\right) \cdots \left(\frac{365-m+1}{365}\right) .$$

Now solving $q(m) > 0.5$ gives $m \geq 23$. So there need only be 23 people in a room for the probability that two of them have the same birthday to be greater than

50%. Most people intuitively expect this number to be much larger, which is why this is called the birthday *paradox*.

Still, it becomes clear why this is so when considering that there are exactly $\binom{23}{2} = 253$ possible pairings of people in a room of 23 people. Considering this it is not surprising that the probability that the birthdays of at least two of these people coincide is equal to that of the first scenario, where one of 253 people must share my birthday (hence there are 253 I/someone-else pairs).

When people with the same birthday are replaced by messages with the same hash value, this theory can be applied to attacks on hash functions. The first scenario is equivalent to performing a second preimage attack as explained above. One message is fixed, and to find another message with the same hash, a large number of messages have to be tried. The second scenario can be used to perform a collision attack. This is often called a *birthday attack*, since its success is based on the birthday paradox. To find two arbitrary messages that collide, a much smaller amount of work needs to be done. In fact, by using the Taylor series approximation of $q(m)$ and solving the resulting equation, it can be seen that for a surjective m-bit hash function (where every output has the same probability and m is sufficiently large) one is expected to obtain a collision by evaluating the hash function for approximately $1.2 \cdot \sqrt{2^m} = 1.2 \cdot 2^{\frac{m}{2}}$ messages. For example, a machine that hashes a million messages per second would take 600,000 years to find a second preimage for a 64-bit hash function, but it could find an arbitrary collision in about one hour [56]. Hence the complexity of a birthday attack is only a square root of the complexity of a regular brute force attack, and if a hash is open to birthday attacks (i.e. it is meant to be collision resistant), it must have double the number of bits of security.

A brute force attack is possible on any hash function, regardless of its structure. Hence one needs to make sure that the hash value is big enough, so that brute force attacks become too complex for even the fastest computers available. So how big is "big enough"? That depends very much on the available computing power, and hence it changes with time. Currently, NIST (the American National Institute of Standards and Technology) recommends replacing all hash functions that are open to birthday attacks by ones which achieve at least 256 bits of security [59]. This means that 256 bits of security are considered secure against brute force birthday attacks (and 128 bits for hashes that are not open to birthday attacks). This number comes from calculations which take into account that the current protocol will be used for a few years, that messages in the protocol will be stored for a few years after that, and that it should then still take a few years brute force to finally crack them. It probably assumes that Moore's Law, the empirical observation that the computing power per unit cost doubles approximately every 18 months, continues to hold.

1.3.2 Cryptanalytic Attacks

There are “smarter” attacks than brute force. Careful mathematical analysis of the functioning of a hash algorithm often allows attacks that require less complexity than brute force, but which are specific to that one hash algorithm or type of hash algorithm they are made for. The study of methods to perform these kinds of attacks is called *cryptanalysis*, and the attacks are called *cryptanalytic attacks*.

Of course, the power of an attacker depends on the resources available to him. In cryptanalysis, *Kerckhoffs' principle* is generally assumed: that the algorithm itself is public knowledge, that is, known by the attacker. This is true for all standardised and other widely used hash functions. Security should not rely on the algorithm being hidden from an attacker (this is often called *security through obscurity*), but it should rather have its foundation in the mathematical properties of the hash.

Note also that many successful cryptanalytic attacks are *side channel attacks*, which exploit a weakness in the implementation of an algorithm rather than its mathematical structure. Such attacks are dangerous and defence against them is important, but this is not the subject of this thesis.

A hash function is considered *broken* if an attack has been found that can produce preimages, second preimages, or collisions with less computational complexity than a brute force attack would take. For example, a hash with a 160-bit hash value is considered broken if collisions can be found in 2^{77} operations. This is regardless of whether 2^{77} is actually computationally feasible or not, so it does not necessarily mean that the attack can be exploited in practice.

1.3.3 The Adversary

In the theory of cryptography, an attacker is often represented as an *adversary algorithm*. A successful attacker is an algorithm that can produce (second) preimages or collisions within feasible complexity. For a hash function to be secure, we need to know that such an algorithm does not exist. This is the approach usually taken when proving security properties of hash functions. It is assumed that there is an algorithm which can efficiently produce (second) preimages or collisions, and this algorithm is then used to show that something else can be produced efficiently with it when we know that it cannot, giving a contradiction.

For example, to prove that a hash is preimage resistant, it needs to be shown that it is “hard” to produce a preimage for a given hash value, that is, that there is no polynomial-time (i.e. efficient) algorithm which can do this. A typical proof would assume that such an algorithm exists and derive a contradiction.

1.4 Applications of Hash Functions

Hash functions are used widely for many different applications in computer science. As an illustration of how cryptographic hash functions are special, we will start with a brief discussion of non-cryptographic uses of hash functions.

1.4.1 Applications of Non-Cryptographic Hash Functions

Here the fundamental property that two different inputs are likely to hash to different values is assumed, but beyond that specific properties are not required.

The most well-known example is the *hash table*, which allows efficient lookup of data records. A hash function maps each key (e.g. a person's name) to a (probably unique) index, the hash digest, which is used to store and locate the desired value. It is important that the indexes are mostly distinct, because the procedure for collision resolution adds undesired complexity and therefore degrades the performance of a hash table implementation.

Another important application is *error detection*. When data is transmitted electronically, for example through the Internet, bit errors are likely to occur. To detect them, a hash value of the data can be added and upon receipt, the hash value of the data can be re-calculated and compared to the received hash value. If the values match, errors are highly unlikely because the original message and a received incorrect message are very unlikely to hash to the same value.

Hash functions are also used for *quick comparison of data*. Where it would be too complex to compare two data sets, the hash values can be compared. If they are the same then it is very probable that the original data sets are the same. This method is used for audio identification, for example to find out if an MP3 file matches one of a list of given files.

1.4.2 Applications of Cryptographic Hash Functions

A hash function that has some or all of the security properties defined in Section 1.2 can be used for even more interesting purposes: for applications in cryptography.

Most people immediately associate cryptography with encryption and decryption of data. However this is only a small part of what is actually involved in cryptography. Besides encryption, the discipline also includes authentication, access control, digital signatures and much more. During the last few years each of these techniques have become central to computer network security, which now concerns anyone using a computer network such as the Internet; and many of these techniques utilise cryptographic hash functions.

A standard example that well illustrates the use of cryptographic hash functions is *password storage* in a multiple-user system, where each user has to authenticate themselves by entering a password. The system has to check whether an entered password is correct, but simply storing all username-password matches in a file is not a good idea because such a file could easily be compromised by an attacker, who would then have access to all the passwords. What is done in practice (e.g. in all Linux systems) is that the the hash values of passwords are stored. That way an attacker who gains access to the file still does not know a single password. He/she cannot calculate them because the cryptographic hash function is one-way. But when a password is entered, its hash can be re-calculated and compared to the stored value, thus authenticating a user who knows the correct password.

Hash functions are also a vital part of many *digital signature* schemes, which are designed to have all the properties of a physical signature but can be computed and transmitted electronically. Since digital signatures often use public key cryptography and are therefore expensive to compute, it is useful to hash a message and only sign the (usually much shorter) hash value. Since hashing is fast, this saves computation time. This does mean, however, that the signature scheme is only as good as the hash function used and that flaws in the hash function will weaken the security of the signature. In other words, the security of the signature scheme depends on the strength of the hash function. For example, if two documents have the same hash, their signature is also identical. If it is easy to produce collisions then signatures can be forged for certain documents. If it is known that a signature can easily be forged then the signer can credibly deny that he/she signed the document, leaving the signature scheme useless.

Similarly to the error checking functionality of non-cryptographic hashes, cryptographic hash functions are used to check the integrity of files. This is often referred to as *malicious code recognition* and finds applications in intrusion detection systems and file servers. Some types of intrusion detection systems store cryptographic hashes of important system files and are able to check that a file has not been altered by re-computing the hash and comparing it to the stored value. File servers often provide so-called *cryptographic checksums* that can be used to affirm the integrity of downloaded files, as long as the server can be trusted. In contrast to error checking, where only randomly occurring errors must be found (which makes a non-cryptographic hash sufficient), malicious code detection requires detecting deliberate alterations made to code. This includes attackers who are aware that hashes have been stored and may thus try to produce malicious code with the same hash as the original code. If the hash function is second preimage resistant, however, such code becomes impossible to produce. This is a good way of preventing a computer from running maliciously altered code, which might otherwise cause much harm to the system.

Hash functions can also be used to build other cryptographic primitives, such as *message authentication codes (MACs)*, which are used to authenticate messages.

A MAC algorithm can be a *keyed hash function*, that is, a hash function that takes as input an arbitrarily long message and a secret key. For a collision resistant hash function, only someone who knows the secret key is able to compute the particular hash value or MAC. The message authentication code also protects the integrity of the message, since it allows the verifier (who also possesses the secret key) to detect any changes to the message content.

Cryptographic hash functions can also be used to build *commitment schemes*, *block ciphers*, *stream ciphers* and *pseudo-random number generators*.

1.5 Design of Cryptographic Hash Functions

Looking at the different requirements a cryptographic hash function should fulfil and the attacks it must withstand, it is already obvious that such functions are hard to design. It becomes even more obvious when considering the following arguments.

The space of inputs $\{0,1\}^*$ has infinitely many elements, whereas the number of possible outputs $|\{0,1\}^m| = 2^m$ is finite. Therefore a hash function is never injective, that is, collisions always exist. For a cryptographic hash function to be collision resistant it does not only have to be unlikely that two given messages will hash to the same value, but it has to be hard for an adversary to find a collision who deliberately tries to produce one. The difficulty lies in designing a hash function such that these collisions are hard to produce.

Another challenge lies in the requirement that hash functions be fast. They must be efficiently computable in one direction and intractable in the other direction. Most operations that have this property (e.g. factoring or discrete logarithms) require multiplications or even exponentiations, which are expensive to compute. Operations that are cheap on a computer, such as XOR or addition, are easy to reverse.

The greatest difficulty in hash function design, however, lies in *proving* properties. It is not good enough to assume or argue that security properties are satisfied, but it is extremely hard to be sure that they are. Almost all hash functions that were thought to have wonderful cryptographic properties but have no proofs of security were broken eventually. But at the same time, it is very difficult to come up with good proofs. This is partly due to the fact that proofs have to make many simplifying assumptions, which are often not realistic enough. A human attacker has to be impersonated by an algorithm, and computing power has to be quantified precisely. Also, the ideas of "hardness" vary greatly among cryptographers, and many definitions are so complicated that they become very tricky to handle mathematically.

Because it is so difficult to design good cryptographic hash functions, the security of most hash algorithms used today is questionable. Many of them are well thought out. Much time and energy has gone into their construction and they were tested thoroughly. Yet these designs seem ad-hoc and it is not immediately clear why they achieve the properties they claim. Often it is more hope than certainty. Maybe not surprisingly, weaknesses are frequently found and it has become a race to keep up with the most recent attacks and replace hash functions with newer ones as weaknesses are found, hoping that the new algorithm might last for a while and that someday someone will finally find a hash function that can really be trusted.

Chapter 2
Custom-Designed Hash Functions

The most commonly used hash functions today are MD5 and SHA-1. These are custom-designed hash functions, that is, algorithms that were especially designed for hashing operations. Other examples of custom-designed hash algorithms are MD2, MD4 and MD5 (the MDx-family), SHA-0, SHA-1, SHA-256/224 and SHA-512/384 (the SHA-family), RIPEMD-160, HAVAL and N-hash.

Custom-designed hash algorithms are designed to be very efficient on 32-bit machines, which makes them very popular, even though their security is only based on heuristic arguments. None of the desired properties of cryptographic hash functions can actually be proven for them. However recent advances in cryptanalysis have shown that this is not good enough. In fact, all of the hash functions mentioned above apart from the SHA-2 algorithms (i.e. SHA-224, SHA-256, SHA-384, SHA-512) are currently considered broken. Although not all of the theoretical attacks are practical yet, they are rapidly being improved and put into practice. Trust in custom-designed hashing has long been undermined, leaving hardly any cryptographic hash functions that can still be used without concern.

This chapter discusses the two most widely used custom-designed hash algorithms MD5 and SHA-1, how they work, how their security is argued and how they have been attacked. This motivates the research of hash functions with provable security, which will be investigated in the following chapters.

2.1 MD5

The Message Digest Algorithm 5, known as MD5, was designed by Ronald Rivest of MIT in 1991 and is specified in the MD5 RFC [53]. It takes a message of arbitrary length as input and produces a 128-bit message digest.

MD5 is used widely in the software world to compute cryptographic checksums and to store passwords. It is part of applications such as GPG (public key encryption), Kerberos (network authentication), TLS (secure client-server connections), SSL (client-server authentication), Cisco type 5 enable passwords (password storage system) and RADIUS (remote user authentification) [67].

2.1.1 Terminology and Notation

A “byte” is an 8-bit quantity and a “word” is a 32-bit quantity. A sequence of 8 bits is interpreted as a byte with the most significant bit listed first, and a sequence of 4 bytes is interpreted as a word with the least significant byte listed first.

Let “+” denote addition mod 2^{32} and let “$\lll s$” denote circular left shift (rotation) by s bit positions. Let “$X \wedge Y$” denote bit-wise AND of X and Y, “$X \vee Y$” bit-wise OR of X and Y, “$X \oplus Y$” bit-wise XOR of X and Y, and “$\neg X$” the bit-wise complement of X.

2.1.2 The MD5 Algorithm

Let M be the input message of length b bits. M is first padded to a multiple of 512 bits and then divided into 512-bit blocks $M_0, \ldots, M_{n-1}$, each consisting of 16 words. Each block is then processed in 4 rounds, each consisting of 16 operations, using a 4-word buffer denoted A, B, C, D. After all blocks have been processed, the buffer contains the message digest.

More specifically, the steps in MD5 are:

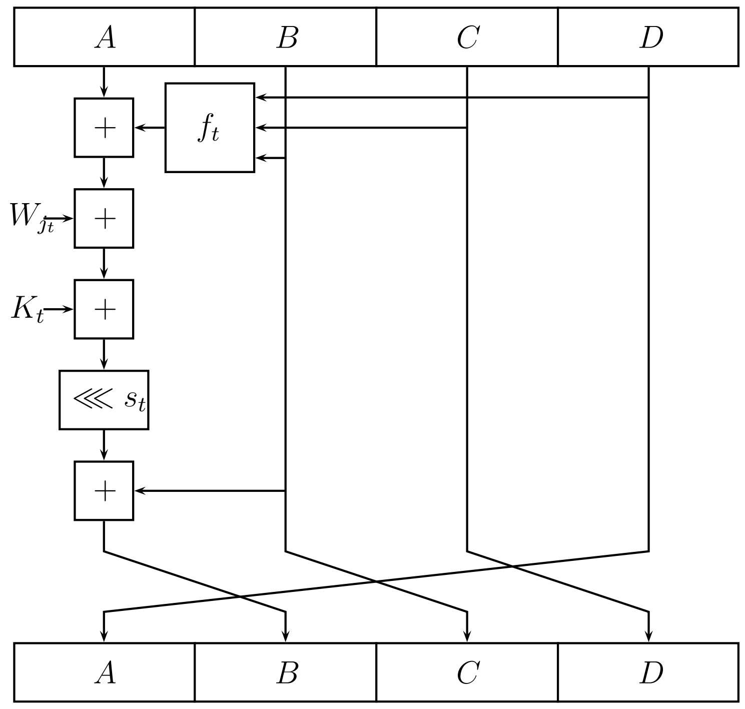
1. **Padding.** A single bit “1” is appended at the end of the message. Then “0” bits are appended until the length of the new message is congruent to 448 modulo 512. Finally a 64-bit representation of b (the length of the original message) is appended. The resulting message is an exact multiple of 512 bits long.
2. **Initialise buffer.** The buffer is initialised to the hex values
   $A = 01234567$
   $B = 89\mathrm{ABCDEF}$
   $C = \mathrm{FEDCBA98}$
   $D = 76543210$
   (with the least significant bit listed first).
3. **Compute constants.** A 64-element table is computed from the sine function according to the formula $K_t = \lfloor 2^{32} \cdot |sin(t+1)| \rfloor$ for $t = 0, \ldots, 63$, where t is in radians.
4. **Auxiliary functions.** Four auxiliary functions, which each take as input three 32-bit words and produce as output one 32-bit word, are defined as
   $f_t(B, C, D) = (B \wedge C) \vee (\neg B \wedge D)$ for $t = 0, \ldots, 15$
   $f_t(B, C, D) = (B \wedge D) \vee (C \wedge \neg D)$ for $t = 16, \ldots, 31$
   $f_t(B, C, D) = B \oplus C \oplus D$ for $t = 32, \ldots, 47$
   $f_t(B, C, D) = C \oplus (B \vee \neg D)$ for $t = 48, \ldots, 63$.

5. **Word order.** Define the following vector that determines in which order the words of a block will be processed in each round:
   Round 1: $(j_0, \ldots, j_{15}) = (0, 1, \ldots, 15)$
   Round 2: $(j_{16}, \ldots, j_{31}) = (1, 6, 11, 0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12)$
   Round 3: $(j_{32}, \ldots, j_{47}) = (5, 8, 11, 14, 1, 4, 7, 10, 13, 0, 3, 6, 9, 12, 15, 2)$
   Round 4: $(j_{48}, \ldots, j_{63}) = (0, 7, 14, 5, 12, 3, 10, 1, 8, 15, 6, 13, 4, 11, 2, 9)$
6. **Shift amounts.** Define the following shift amounts:
   Round 1: $(s_0, \ldots, s_{15}) = (7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22)$
   Round 2: $(s_{16}, \ldots, s_{31}) = (5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20)$
   Round 3: $(s_{32}, \ldots, s_{47}) = (4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23)$
   Round 4: $(s_{48}, \ldots, s_{63}) = (6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21)$
7. **Process message in 16-word blocks.**

```
/* Process each 16-word block. */
for $i = 0, \ldots, n-1$ do
    (a) Divide $M_i$ into words $W_0, \ldots, W_{15}$, where $W_0$ is the left-most word.
    (b) Save $A$ as $\bar{A}$, $B$ as $\bar{B}$, $C$ as $\bar{C}$ and $D$ as $\bar{D}$:
        $\bar{A} = A$
        $\bar{B} = B$
        $\bar{C} = C$
        $\bar{D} = D$
    (c) for $t = 0, \ldots, 63$ do
        $X = B + ((A + f_t(B, C, D) + W_{j_t} + K_t) \lll s_t)$
        $A = D$
        $D = C$
        $C = B$
        $B = X$
        end /* of loop on $t$ */
    (d) Then increment each of the four registers by the value it had before
        this block was started:
        $A = \bar{A} + A$
        $B = \bar{B} + B$
        $C = \bar{C} + C$
        $D = \bar{D} + D$
end /* of loop on $i$ */
```

8. **Output.** The message digest is A, B, C, D.

One MD5 operation at step 7c can be described in the following diagram:



2.1.3 Security of MD5

According to his own statements in the specification [53], Rivest designed MD5 so that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given prespecified target message digest, that is, to be collision resistant and preimage resistant. In addition, MD5 was made to be fast on 32-bit machines and to operate without large substitution tables, hence it can be coded compactly.

While the second and third attributes can be easily verified and are definitely true, the security of MD5 is based on a number of heuristic arguments and no proofs of security exist. Heuristic arguments include that

- the auxiliary functions are non-invertible, non-linear and asymmetric,
- if bits in B, C and D are independent and unbiased, then each bit of $f_t(B, C, D)$ will be independent and unbiased,
- each step adds in the result of the previous step,
- each step has a unique additive constant,
- input words are accessed in a different order in each round, and
- shift amounts in different rounds are distinct.

All of these attributes are said to increase the avalanche effect, meaning that if an input is changed slightly (for example, changing a single input bit), then the output changes significantly (for example, half the output bits flip).

While its speed and the fact that the algorithm is fairly simple and publicly available have made MD5 very popular, it seems rather alarming that the algorithm is used in many cryptographic applications to this day, considering its security is not supported by any proof at all.

2.1.4 Attacks on MD5

MD5 was designed in 1991 to replace an earlier hash function MD4 in which flaws had been found. However, it was soon discovered that MD5 also has its problems. Starting in 1993 the use of MD5 was more and more questioned by several successful collision attacks, and recent results have completely destroyed confidence in the algorithm.

In 1993, den Boer and Bosselaers [5] were able to find a so-called pseudo-collision for the compression function of MD5, that is, two different initialisation vectors that produce a collision when the MD5 compression function is applied to the same message. Although this is an attack that has no practical significance, it exposed the first weakness in MD5.

Dobbertin [16] announced a collision of the MD5 compression function in 1996. While this was not an attack on the full version of MD5, it worried cryptographers enough to recommend switching to a replacement, such as SHA-1, WHIRLPOOL, or RIPEMD-160.

Also, a hash of 128 bits is small enough to allow birthday attacks. Cooke and his company launched a distributed search project in 2004 with the aim of finding collisions for MD5 by a brute force search using Pollard's rho method [39, 40]. The project was abandoned a few months later, when it was announced that collisions had actually been found by analytical methods. This announcement was one of the greatest moments in cryptanalysis for many, and it is said that Wang and her team [66] received a standing ovation when they reported that they had found collisions for the full MD5 at the CRYPTO conference in August 2004. Their attack took about one hour on an IBM p690 cluster, a very powerful Unix server.

Less than a year later, this attack was further improved by Klima [31], who presented an improved algorithm that is able to construct collisions within only a few hours on a single notebook computer. Lenstra, Wang and de Weger [36] showed how this could become an attack of practical importance by constructing two X.509 certificates with different public keys and the same MD5 hash. X.509 is a standard for a public key infrastructure which is widely used, and this attack allows the construction of false certificates. It is the first attack on MD5 of practical significance.

In March 2006 Klima [32] presented a further improved algorithm that can find collisions within one minute on an ordinary notebook computer. It uses a method called tunneling.

This completes the history of MD5 cryptanalysis as of today (7/12/06), but one can never be sure which attack is being worked on behind the scenes right now. Although only collision attacks (and no preimage or second preimage attacks) have been announced so far, it has also been shown how even collision attacks can be of practical importance, leaving MD5 an untrustworthy hash algorithm. Its replacement has long been recommended by anyone who understands the significance of these attacks.

Apart from cryptanalytic attacks, numerous projects have recently created MD5 reverse lookup databases. These are easily accessible online and may be used to look up preimages of a large number of MD5 digests. Such databases can be found at

- `http://md5.crysm.net/`
- `http://md5.benramsey.com/`
- `http://md5.rednoize.com/`,

just to give some examples. When trying to crack passwords, which are frequently just dictionary words, consulting such a database is often a successful approach.

2.2 SHA-1

SHA-1 is the most commonly used member of the Secure Hash Algorithm (SHA) family. It was published by the National Security Agency (NSA) in 1995 as a US government standard [17] and to replace the SHA-0 algorithm from 1993, in which a flaw had been found. SHA-1 takes an input message of at most $2^{64} - 1$ bits and produces a message digest of length 160 bits.

Since MD5 became untrustworthy, SHA-1 has become the most commonly used hash function. It is employed in security applications and protocols such as OpenPGP (encryption of data), S/MIME (public key encryption and signing of e-mail), IPSec (encryption and/or authentification of IP packets) and SSH (secure remote login). The copy prevention of Microsoft's Xbox game console also relies on SHA-1 [60].

2.2.1 Terminology and Notation

As for MD5, 8 bits make up a "byte" with the most significant bit listed first, 4 bytes make up a "word" with the most significant byte listed first, and 16 words make up a block.

As before, let "$+$" denote addition mod 2^{32} and "$\lll s$" circular left shift (rotation) by s bits. Let "$\wedge$", "$\vee$", "$\neg$" and "$\oplus$" denote bit-wise AND, OR, NOT and XOR, respectively.

2.2.2 The SHA-1 Algorithm

SHA-1 is often considered a successor of MD5 because its design is very similar. Padding is performed in the same way, then a message M of length b bits is split into 16-word blocks $M_0, \dots, M_{n-1}$ and each block is processed in 4 rounds, consisting of 20 operations each, and using a 5-word buffer A, B, C, D, E. After all blocks have been processed, the buffer contains the message digest.

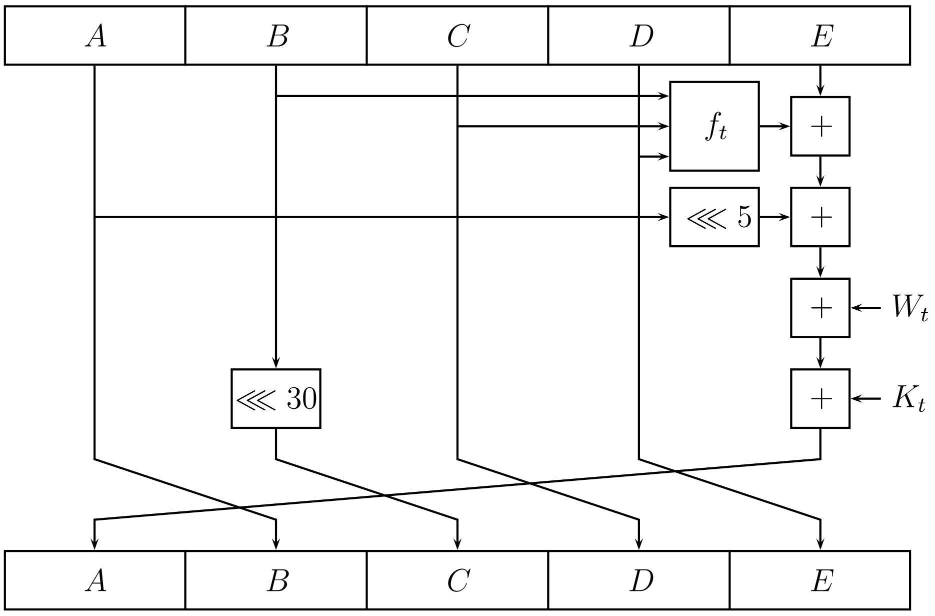
More specifically, the steps in SHA-1 are:

1. **Padding.** M is considered as a bit string and a single bit "1" is appended at the end of the message. Then "0" bits are appended until the length of the new message is congruent to 448 modulo 512. Finally a 64-bit representation of b is appended, resulting in a message which is an exact multiple of 512 bits long.
2. **Initialise buffer.** The buffer is initialised to the hex values
   $A = 67452301$
   $B = \text{EFCDAB89}$
   $C = \text{98BADCFE}$
   $D = 10325476$
   $E = \text{C3D2E1F0}.$
3. **Constants.** The following constants are used (in hex):
   $K_t = \text{5A827999}$ for $t = 0, \dots, 19$
   $K_t = \text{6ED9EBA1}$ for $t = 20, \dots, 39$
   $K_t = \text{8F1BBCDC}$ for $t = 40, \dots, 59$
   $K_t = \text{CA62C1D6}$ for $t = 60, \dots, 79$
4. **Auxiliary functions.** A sequence of logical functions is used, each operating on three words and producing one word as output. They are defined as follows:
   $f_t(B, C, D) = (B \wedge C) \vee (\neg B \wedge D)$ for $t = 0, \dots, 19$
   $f_t(B, C, D) = B \oplus C \oplus D$ for $t = 20, \dots, 39$
   $f_t(B, C, D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$ for $t = 40, \dots, 59$
   $f_t(B, C, D) = B \oplus C \oplus D$ for $t = 60, \dots, 79$
5. **Process message in blocks.**

```
/* Process each 16-word block. */
for $i = 0, \dots, n-1$ do
  (a) Divide $M_i$ into 16 words $W_0, \dots, W_{15}$, where $W_0$ is the left-most word.
  (b) For $t = 16, \dots, 79$ let $W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \lll 1$.
  (c) Save $A$ as $\bar{A}$, $B$ as $\bar{B}$, $C$ as $\bar{C}$, $D$ as $\bar{D}$ and $E$ as $\bar{E}$:
      $\bar{A} = A$
      $\bar{B} = B$
      $\bar{C} = C$
      $\bar{D} = D$
      $\bar{E} = E$
  (d) for $t = 0, \dots, 79$ do
      $X = (A \lll 5) + f_t(B, C, D) + E + W_t + K_t$
```

```
        $E = D$
        $D = C$
        $C = B \lll 30$
        $B = A$
        $A = X$
        end /* of loop on $t$ */
    (e) Then increment each of the four registers by the value it had before
        this block was started:
        $A = \bar{A} + A$
        $B = \bar{B} + B$
        $C = \bar{C} + C$
        $D = \bar{D} + D$
        $E = \bar{E} + E$
    end /* of loop on i */
6. **Output.** The message digest is $A, B, C, D, E$.
```

One SHA-1 operation at step 5d can be described by the following diagram:



2.2.3 Security of SHA-1

The authors of the SHA-1 RFC [17] claim that it is computationally infeasible to find a message which corresponds to a given message digest, or to find two different messages which produce the same message digest for SHA-1. As with MD5, however, no proofs of security exist and all there is to support this statement are heuristic arguments like those mentioned for MD5 in Section 2.1.3.

SHA-1 is also very fast on 32-bit machines and can be coded quite compactly, and it is thus used very widely. In fact, as flaws were found in MD5, cryptographers recommended replacing MD5 by SHA-1, which was done in many applications. Since SHA-1 has been broken as well (meaning that collisions can be produced with less computational complexity than that of a brute force attack), NIST now plans to replace SHA-1 by members of the SHA-2 family (SHA-224, SHA-256, SHA-384, SHA-512, named after their digest lengths), for which no attacks have been reported, by 2010.

2.2.4 Attacks on SHA-1

The members of the SHA-family were designed as successors of MD4, just as MD5 was, but they lasted a bit longer. SHA-1 is very similar to its predecessor SHA-0, and so the first reason to doubt the security of SHA-1 was the announcement that SHA-0 had been broken by Chabaud and Joux [8] at CRYPTO ’98. The next milestone in the cryptanalysis of SHA-0 was when Wang and her team [66] announced their collision attack in 2004, which also works for SHA-0. That was when cryptographers first started to recommend finding alternatives to SHA-1, especially in the design of new cryptosystems. Also as a result of that, NIST announced it would phase out the use of SHA-1 by 2010 and replace it by SHA-2 variants [44].

The first successful attack on SHA-1 itself was performed by Rijmen and Oswald [52] in early 2005. They were able to break a reduced version of SHA-1: 53 out of 80 rounds. Only a month later a break of the full version of SHA-1 was announced by Wang, Yin and Yu [58]. This was another famous day for Wang and her team, who based their attack on several different methods used in earlier attacks on SHA-0 and MD5. This attack required 2^{69} operations, but was soon improved to take only 2^{63} [57]. Such collision attacks generally work by starting off with two messages and continually modifying them throughout the attack. That means that the structure of the colliding messages is determined by the attack, and they will almost certainly turn out to be complete gibberish. Although this is of theoretical importance, it is hard to turn it into a practical attack.

At CRYPTO ’06, however, Recheberger and de Cannière announced the first collision attack on SHA-1 where the attacker can influence the colliding messages. According to Recheberger [61], the new attack allows up to 25% of the colliding messages to be freely selected, as straight text for instance. The remaining 75% are again determined by the attack, but it is suspected that the amount to be freely selected can be further increased by optimising the attack. This is now a quite practical attack itself, considering that HTML documents, for example, may have complete nonsense after the `</html>` tag that will never be printed. So it is now possible to produce two seemingly identical html documents with the same SHA-1 hash. This leaves SHA-1 no better off than MD5.

Just as with MD5, (second) preimage attacks on SHA-1 have not been accomplished, but the collision attacks have reached a level that causes serious concern and makes urgent a quick replacement of the algorithm.

2.3 Security of Custom-Designed Hash Functions

As for MD5 and SHA-1, successful collision attacks have been performed for all custom-designed hash functions except RIPEMD-160 and the SHA-2 family. Admittedly, some collision attacks are of limited practical significance and preimage attacks have not been found. Still, the reasons to distrust custom-designed hashing are many. More important than the actual attacks is the fact that security properties cannot be proven, leaving users in permanent distress about how secure they are. Hash functions have too many important applications to leave their security up to assumptions and luck. Hash functions of which security properties are certain are highly desirable, making research into provably secure hash functions absolutely necessary.

Chapter 3
Provably Secure Hashing

Recent developments in the cryptanalysis of hash functions have clearly shown that heuristic security arguments are not good enough. Something provable is needed. Hashing is a vital concept in cryptography and at the moment there are no hash functions which are efficient and whose security can be trusted at the same time. Nobody knows exactly how far away we are from such designs, but research in the area of provably secure hash functions has been going on for many years. There are some promising designs that seem to need only a little bit more work to make them practical.

One major issue is always the speed-security trade-off. Security typically relates to the difficulty of a well-known hard problem, but most of these problems involve operations which are slow on a computer, such as exponentiation or multiplication. Provably secure hash functions will probably never be as fast as custom-designed hash functions, but that might just be a sacrifice one has to make to achieve security.

First, however, a few basic questions have to be answered. What properties exactly does a provably secure hash function have, what is considered secure, how is security measured, how is security proven and how are secure hash functions designed? These basic ideas are introduced in this chapter before the following chapters go on to show some concrete examples of such functions.

3.1 Types of Provably Secure Hash Functions

There are different types of provably secure hash functions, depending on which of the security properties have to be fulfilled.

Definition 3.1.1 (collision resistant hash function (CRHF))
A **collision resistant hash function (CRHF)** *is a hash function that is*

- *preimage resistant and*
- *collision resistant.*

The concept of collision resistant hash functions was first introduced by Merkle [41], and the first formal definition was given by Damgård in 1989 [13].

Since collision resistance implies second preimage resistance, collision resistant hash functions have all three desired security properties and can therefore be considered the "best" hash functions. However, they are also the hardest to design, since collision attacks are very difficult to defend against. In fact, the effort to find a collision is only the square root of the effort to find a second preimage. This motivates the definition of another primitive.

Definition 3.1.2 (one-way hash function (OWHF))
A **one-way hash function (OWHF)** *is a hash function that is*

- *preimage resistant and*
- *second preimage resistant.*

One-way hash functions were first introduced by Diffie and Hellman in 1976 [15]. Since second preimage resistance is a weaker requirement than collision resistance, they are easier to design than CRHFs.

A slightly different primitive was introduced by Naor and Yung in [43] in 1989.

Definition 3.1.3 (universal one-way hash function (UOWHF))
A **universal one-way hash function (UOWHF)** $\mathcal{H}$ *is a family of one-way hash functions indexed by a key*

$$\mathcal{H} = \{h_K : \{0,1\}^* \rightarrow \{0,1\}^n \mid K \in \mathcal{K}\}$$

where $\mathcal{K} = \{0,1\}^k$ *is the key space (hence* k *is the key size in bits).*

Note that the keys here are not secret elements. They are public and more parameters than keys, but traditionally called keys. Their purpose is not obvious without the more formal definition of UOWHFs using the model of an attacker algorithm, which will be given in Section 4.1.

Naor and Yung [43] prove constructively that UOWHFs exist and also show that their security assumption, although not as strong as that of CRHFs, is strong enough for some applications. For example, they propose a one-way based secure digital signature scheme, which is based on UOWHFs and still secure against the most general attack known (it is existentially unforgeable under an adaptive chosen plaintext attack).

After having weakened the assumption in going from a CRHF to a OWHF, it seems natural to take this idea one step further, pull out yet another property, and define yet another type of provably secure hash function.

Definition 3.1.4 (preimage resistant hash function (PRHF))
A **preimage resistant hash function (PRHF)** *is a hash function that is*

- *preimage resistant.*

This is the “weakest” type of hash function, requiring neither second preimage resistance nor collision resistance, but only preimage resistance. Therefore it is also the easiest to design. And yet it is still good enough for many authentication applications such as password storage, as shown in Chapter 5.

3.2 NP-Complete Problems

All types of provably secure hash functions require certain operations (such as finding preimages, second preimages, or collisions) to be "hard" or "computationally infeasible", even though they are theoretically possible. A preimage always exists, but no one should be able to find it, even if they possess all the computational power and resources in the world and are given a huge amount of time (e.g. the age of the universe). How can something like that be defined more precisely, and how can it be proven?

In theoretical computer science there is a theory about a class of problems, called *NP-complete problems*, which are believed to be very hard to solve. Although the proof of this remains one of the biggest open problems in computer science today, scientists have very good reasons to believe that no efficient algorithms exist to solve such problems. They are considered the hardest problems with solutions that can be verified efficiently.

Thus far we have not rigorously defined what it means for a problem to be "hard". A popular and probably the most practicable approach in cryptography is calling a problem "hard" precisely when it is NP-complete. The hardness of operations is then proven by relating them to well-known NP-complete (i.e. believed intractable) problems. If it can be proven that finding a preimage to a hash is at least as hard as solving a given NP-complete problem, for instance, then that gives an excellent indication that such a computation is among the most difficult. This is different to the concept of proving things in mathematics and might seem imprecise and unsatisfactory at first, but it is the best we can do at the moment, and it has turned out to work well in practice. It is clearly much better than what can be done for custom-designed hashes.

The machinery to do this has been formalised over the years (e.g. in [11]) and allows proving concrete statements about how secure certain operations are. This section introduces some of the most important concepts of the theory of NP-completeness and explains how NP-complete problems can be used to prove cryptographic properties of hash functions.

3.2.1 Terminology and Notation

First, an important distinction between the terms "problem", "instance" and "algorithm" must be made. A *problem* is a general question to be answered and has many *instances*; each instance has a *solution*. There may be many *algorithms* which solve a problem. A problem is formally defined as follows:

Definition 3.2.1 (abstract problem)
An **abstract problem** *Q is a function $Q : I \to S$ that maps an instance from the set I of problem* **instances** *to a solution from the set S of problem* **solutions**.

For example, the problem ShortestPath can be formulated as "given a graph G and two vertices v and w, find the shortest path between v and w". (For graph theory definitions see Section 5.1.) An instance of this problem would be a triple of a specific graph $G = (V, E)$ and two vertices v, w. A solution would be a sequence of vertices $v, v_1, \ldots, v_n, w$ describing a path from v to w.

A *decision problem* is one that has a yes/no solution. More formally:

Definition 3.2.2 (decision problem)
A **decision problem** *Q is a function $Q : I \to \{0, 1\}$ that maps an instance from the set I of problem instances to a solution from the set $\{0,1\}$.*

Formally the theory of NP-completeness only applies to decision problems. Although many natural problems are not decision problems, every problem can be turned into a decision problem in a natural way. ShortestPath is an *optimisation problem*, but it can be stated as a decision problem in the following way: "given a graph G and two vertices v and w, does there exist a path from v to w of length at most k?" This problem is referred to as Path(k). The solution 1 is usually taken to mean "yes" and 0 is taken to mean "no". It is obvious that if the optimisation problem can be solved, the decision problem can also be solved, simply by comparing the value obtained from the optimisation problem solution (e.g. the length of the shortest path) to the bound in the decision problem (e.g. k). This means that the optimisation problem is at least as hard as the decision problem. Therefore optimisation problems may also be referred to as NP-complete (as defined in Section 3.2.5), although the theory of NP-completeness formally only applies to decision problems.

An **algorithm** is a general step-by-step procedure (a computer program) that solves a given problem. That is, given any instance of a problem, the algorithm attempts to find the corresponding solution. Being a computer program, an algorithm can only take inputs in the form of binary strings (e.g. a binary representation of natural numbers). Hence any problem instance must be encoded in this form.

Definition 3.2.3 (encoding)
An **encoding** *is a mapping from a set of abstract objects to the set of binary strings $\{0, 1\}^*$.*

For example, a graph can be encoded to a binary string by listing the rows of its adjacency matrix, by listing all vertices and edges in a specified way, or by listing each vertex with all its neighbours.

Using an encoding, every abstract problem can be converted to a *concrete problem.*

Definition 3.2.4 (concrete problem)
A **concrete problem** *is an abstract problem with instance set $\{0, 1\}^*$.*

Notation 3.2.5
Let f and g be functions from $\mathbb{N}$ to $\mathbb{R}$. Then $f(n) = O(g(n))$ if there exists a constant c such that $|f(n)| \leq c \cdot |g(n)|$ for all $n \geq 0$.

An algorithm solves a problem in time $O(g(n))$ if, when it is provided with a problem instance of length n (i.e. the binary string of the encoding has length n), the algorithm can produce the solution in at most $O(g(n))$ time.

3.2.2 Polynomial Time Solvable Problems

Definition 3.2.6 (polynomial time solvable problem)
A problem is **polynomial time solvable** *if there exists an algorithm A that solves it in time $O(n^k)$ for some constant k, where n is the length of the input. The algorithm A is called a* **polynomial time algorithm**.

The set of all polynomial time solvable problems is usually denoted as P. They are generally considered tractable.

Definition 3.2.7 (complexity class P)
The **complexity class P** *is defined as the set of concrete decision problems that are solvable in polynomial time.*

An example of a problem in P is $\mathsf{Path}(k)$. An algorithm has been specified that can do this in polynomial-time.

Since polynomials are closed under addition, multiplication and composition, P also has nice closure properties. For example, if an algorithm makes a fixed number of calls to a polynomial-time subroutine, it is still polynomial. If the output of one polynomial-time algorithm is fed into another polynomial-time algorithm, the composite algorithm is still polynomial-time. These properties are essential for most proofs involving polynomial-time algorithms.

3.2.3 Polynomial Time Verifiable Problems

Definition 3.2.8 (polynomial time verifiable problem)
A problem is **polynomial time verifiable** *if there exists a two-input polynomial-time algorithm A which, given a problem instance and a solution, verifies that the solution is correct. The algorithm A is called a* **verification algorithm**. *The solution is often referred to as a* **certificate** *which certifies that the instance is indeed an instance of the given problem.*

For example, consider the problem $\mathsf{Path}(k)$. Given a specific instance (i.e. a graph and two vertices) and a solution (i.e. a path p between the two vertices), it can easily be checked whether the length of the path is at most k. If so, p can be viewed as a certificate that the instance indeed belongs to $\mathsf{Path}(k)$.

The set of all polynomial time verifiable problems is usually denoted as NP.

Definition 3.2.9 (complexity class NP)
The **complexity class NP** *is the set of concrete decision problems that are verifiable in polynomial time.*

Path(k) is in NP. However, since it is also in P, nothing is gained. Verifying a solution takes about as long as finding one from scratch. In fact, any problem in P is automatically also in NP, since given a problem instance and a solution, the solution can be found in polynomial time (since the problem is in P) and then be compared to the given solution. If they match, then the solution is correct and has been verified in polynomial time. Hence P $\subseteq$ NP. The question whether NP $\subseteq$ P, that is, whether any polynomial time verifiable problem can also be solved in polynomial time, remains unanswered to this day. The class of problems for which no polynomial-time algorithm is known but which can be verified in polynomial time is called NPC. For a more precise definition more machinery is needed.

3.2.4 Formal-Language Theory

Definition 3.2.10 (alphabet)
An **alphabet** Σ *is a finite set of symbols.*

Definition 3.2.11 (language)
A **language** L *over an alphabet* Σ *is a set of strings made up of symbols from* Σ*. The language of all strings over* Σ *is* Σ^**.*

The most commonly used alphabet is $\Sigma = \{0,1\}$ and a language over this alphabet is $\{0,1\}^*$.

The set of instances I of any decision problem Q can be represented as a language. When every instance is encoded as a binary string, I can be taken to be $\{0,1\}^*$. Note that not every string in $I = \{0,1\}^*$ may “make sense” as a problem instance, but this does not matter because we can simply define the problem solution for these strings to be 0.

Since Q is entirely characterised by those problem instances that produce a 1, we can view Q as a language L over $\{0,1\}$ where $L = \{x \in \{0,1\} | Q(x) = 1\}$. In other words, $L \subseteq I$ is the set of problem instances which produce a 1. This way, every decision problem can be described as a language $L \subseteq \{0,1\}^*$.

This formal-language framework now allows a more concise relation between decision problems and the algorithms that solve them.

Definition 3.2.12 (accept/reject)
An algorithm A **accepts** *a string* $x \in \{0,1\}^*$ *if* $A(x) = 1$ *and* **rejects** x *if* $A(x) = 0$*. The language accepted by* A *is* $L = \{x \in \{0,1\}^* | A(x) = 1\}$*.*

Definition 3.2.13 (decide)
A language L *is* **decided** *by an algorithm* A *if every binary string is either accepted or rejected by* A*.*

Notice that there is a subtle difference between accepting and deciding a language. An algorithm which accepts a language L (i.e. which accepts all $x \in L$) need not necessarily reject all strings $y \notin L$. It may also produce no output for these strings (e.g. loop forever). An algorithm which decides a language L must either accept or reject *every* element $x \in L$.

Using this terminology, a **complexity class** can now be defined as a set of languages, membership in which is determined by a *complexity measure* (such as running time) on an algorithm that determines whether a given string belongs to a certain language.

The definition of P can be stated as

$$\mathrm{P} = \{\ L \subseteq \{0,1\}^* |\ \exists \text{ algorithm } A \text{ that decides } L \text{ in polynomial time }\}$$

and

$$\mathrm{NP} = \{\ L \subseteq \{0,1\}^* |\ \exists \text{ algorithm } A \text{ that verifies } L \text{ in polynomial time }\}$$

where a **verification algorithm** A is now more precisely defined as a two-argument algorithm, where one argument is an ordinary input string $x \in \{0,1\}^*$ and the other is a binary string y (called certificate), which verifies x if there exists a certificate y such that $A(x,y) = 1$. The language verified by A is then

$$L = \{x \in \{0,1\}^* |\ \text{there exists a } y \text{ such that } A(x,y) = 1\ \}.$$

3.2.5 NP-complete Problems

Finally we can give a definition of the class NPC of NP-complete problems. It is the set of problems which are polynomial time verifiable but not known to be polynomial time solvable. All problems in NPC are “essentially equivalent” in the sense that if any one NP-complete problem can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time. This “equivalence” is called *reducibility.*

A problem Q_1 can be reduced to another problem Q_2 if any instance of Q_1 can be “easily rephrased” as an instance of Q_2, and a solution to the instance of Q_2 provides a solution to the instance of Q_1. That means that if Q_2 can be solved, Q_1 can also be solved, hence Q_1 is at most as hard as Q_2. Describing Q as a language L, this can be stated as

Definition 3.2.14 (polynomial-time reducible)
A language L_1 is **polynomial-time reducible** *to the language L_2 (write $L_1 \leq L_2$) if there exists a polynomial-time computable* **reduction function** *(meaning there exists a polynomial-time algorithm that can compute this function)*

$$f : \{0,1\}^* \to \{0,1\}^*$$

such that for all $x \in \{0,1\}^* : x \in L_1$ if and only if $f(x) \in L_2$.

If $L_1 \leq L_2$ then L_1 is no more than a polynomial factor harder than L_2. So in this case $L_2 \in P$ implies $L_1 \in P$.

It can also be seen quite easily that this relation is symmetric. If $L_1 \leq L_2$ then $L_2 \leq L_1$ is also true.

Definition 3.2.15 (NP-complete)
A language L is **NP-complete** *if*

1. *$L \in NP$ and*
2. *$L' \leq L$ for every $L' \in NP$.*

Let **NPC** *denote the set of NP-complete problems.*

An example of an NP-complete problem is HamCycle: "does a given graph G have a Hamiltonian cycle (i.e. a cycle that visits each vertex of G)?" HamCycle is clearly in NP, because given a graph and a cycle, it can be easily verified that it is in fact a Hamiltonian cycle. But given a (sufficiently large) graph, it is "very hard" to find a Hamiltonian cycle or even to decide if one exists.

Definition 3.2.16 (NP-hard)
A language L is **NP-hard** *if it satisfies property 2 in the above definition.*

Property 2 requires that any language $L' \in NP$ is reducible to an NP-complete language L, hence L is at least as hard as any other language $L' \in NP$. That is why the languages in NPC are often referred to as the "hardest" languages in NP.

Since the relation "$\leq$" is symmetric, any language in NPC is no more than a polynomial factor harder than any other language in NPC, so all languages in NPC can be considered "equivalent" and they can all be reduced to each other. Hence to prove a language NP-complete, it only has to be reduced to one language which is already known to be NP-complete. This is stated in the following

Lemma 3.2.17
If L is a language such that $L' \leq L$ for some $L' \in NPC$, then L is NP-hard. If $L \in NP$, then $L \in NPC$.

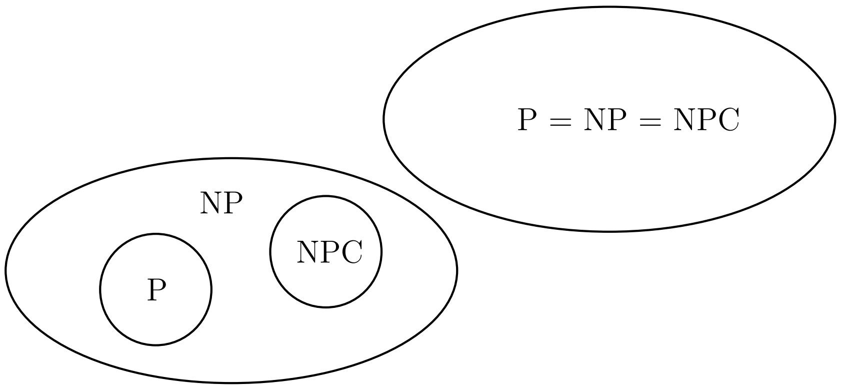
Hence to show that a language L is in NPC, one has to do the following:

1. Show that $L \in NP$.
2. Select a known language $L' \in NPC$.
3. Describe an algorithm A that computes a function f mapping every instance of L' to an instance of L.
4. Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0,1\}^*$.
5. Prove that the algorithm A runs in polynomial time.

This method is usually used to prove that a problem in cryptography is NP-complete, that is, sufficiently hard. Hundreds of proven NP-complete problems exist, so it is often easy to select an appropriate one for a proof. It remains to answer the question why NP-complete problems are thought to be so hard that cryptographers are willing to put their trust in the intractability of these problems. It has been proven that

Theorem 3.2.18 (P = NP?)
If any NP-complete problem is in P, then P = NP. If any NP problem is not polynomial-time solvable, then NPC and P do not intersect.

This essentially says that if one NP-complete problem is polynomial-time solvable, then any NP-complete problem is polynomial-time solvable; and if one NP-complete problem is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable. In diagrams, the two possible scenarios are:



The so-called "P $\neq$ NP question" was posed in 1971 by Stephen A. Cook [10] and has not been answered despite years of extensive research. No one has been able to prove a superpolynomial-time lower bound for any NP-complete problem. Still, most scientists believe that the scenario on the left is true, since no polynomial-time algorithm has been found for any NP-complete problem despite years of study. Given the wide range of NP-complete problems that have been studied so far, without any progress toward a polynomial-time solution, "it would be truly astounding if all of them could be solved in polynomial time", says Cormen [11].

3.2.6 NP-complete Problems in Cryptography

Hundreds of problems from different areas (graph theory, network design, sets and partitions, sequencing and scheduling, mathematical programming, algebra, number theory, logic, program optimisation and others) have been proven to be NP-complete to this date (for an elaborate list see [20]), and many of them can be related to cryptographic problems, some even in very natural ways (for example, the subset sum problem, see Section 4.2, and the Hamiltonian cycle problem, see Chapter 5). However, there are some problems with this approach. The obvious problem is that it is unclear whether NP-complete problems are really hard, and a polynomial-time algorithm for any of them would make them all tractable, thus destroying any cryptographic scheme based on them.

The other major issue is that an NP-complete problem is only hard in the general case, not in every case. NP-completeness only guarantees that there is no polynomial-time algorithm which solves every instance of a problem, but there may always be special cases which are easy to solve. For some NP-complete problems algorithms exist that solve a large percentage of instances efficiently, which would be disastrous for a cryptographic application. It takes some analysis to determine how many such special cases exist, and it is desirable to show that at least most cases are intractable (or the average case is intractable), if not every case.

Some NP-complete problems can be defeated with the help of *approximation algorithms*. These are algorithms that find near-optimal solutions to optimisation versions of NP-complete problems in polynomial running time. In practice, near-optimality is sometimes good enough. It must be ensured that NP-complete problems on which cryptographic security relies do not have efficient approximation algorithms. And still it is always possible that one will be developed in the future. There are, however, NP-complete problems for which it has been proven that they have a version which is "absurdly hard to approximate" [69], that is, that no efficient approximation algorithms exist. In fact, there is an interesting theory of inapproximable NP-hard problems. Arora [1] shows that for certain NP-hard problems, achieving a reasonable approximation is no easier than computing optimal solutions. In other words, approximation of these problems is NP-hard.

Another approach is to consider algorithms that solve "typical" or "average" instances instead of worst-case instances of NP-complete problems. In practice, however, identifying "typical" instances is not easy [1].

Lastly, we should comment on the rather philosophical question why polynomial-time solvable problems are considered tractable or even efficient. One could argue that a polynomial-time solvable problem which can only be solved in $O(n^{100})$ surely is computationally infeasible. This is true, but in practice, there are very few practical problems that require time on the order of such a high-degree polynomial. The polynomial-time computable problems encountered in practice typically require much less time. Therefore there is good reason to think that a polynomial-time problem will be feasible in practice. The other reason why all polynomial-time solvable problems are put together in one class is the useful closure properties mentioned earlier (see Section 3.2.2).

Altogether, it does not matter whether there are polynomial-time solvable problems which are really not feasible in practice. What is important in cryptography is that anything which is not polynomial-time solvable is computationally infeasible. Algorithms whose time complexity cannot be bounded by a polynomial, but only by an exponential function, are referred to as *exponential time* *algorithms* and generally considered inefficient, because most of them are merely variations on exhaustive search, whereas polynomial-time algorithms are generally made possible only through gain of some deeper insight into the structure of a problem.

3.3 Domain Extender Algorithms

A common approach to designing hash functions is to perform two steps. In the first step, an (efficient) *compression function* h is designed, which hashes a (relatively short) n-bit string to a shorter m-bit string, that is, takes an input of fixed length and compresses it by $n-m$ bits to an output of a shorter fixed length. Then a *domain extender algorithm* is used to build a hash function h' from the compression function h. The function h' hashes inputs of arbitrary length l (for $l > n$) to produce an m-bit hash value.

The compression function h is usually constructed to achieve some specified security property, and the domain extender is designed to preserve this property. For example, a good domain extender algorithm is one that produces a hash function h' which is collision resistant if h is collision resistant. This two-step process makes it easier to design hash functions as well as to prove specific security properties.

A very efficient and also simple, natural and well-known extender algorithm is the one proposed independently by Merkle [41] and Damgård [14] at CRYPTO '89. It is widely used and has become known as the *Merkle-Damgård extender*. It uses a compression function $h : \{0,1\}^n \to \{0,1\}^m$ to build a hash function h'. The input message M is split into a first block M_0 of length n and $\mathcal{L}$ blocks $M_1, \ldots, M_{\mathcal{L}}$ of length $n-m$. Then the blocks are processed consecutively. The output of one step (m bits) is concatenated with a new message block ($n-m$ bits) to form the n-bit input of the next iteration.

Notation 3.3.1
For two bit strings x and y, let $x||y$ denote the concatenation of x and y.

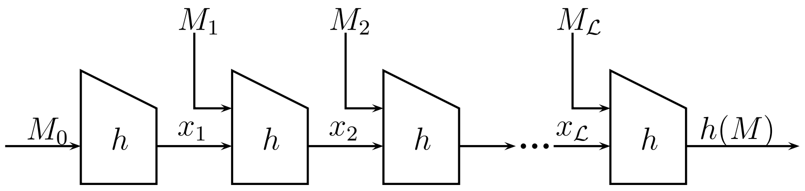
Definition 3.3.2 (Merkle-Damgård extender)
Let $h : \{0,1\}^n \to \{0,1\}^m$ be a compression function and let $l = n + \mathcal{L} \cdot (n-m)$ with $\mathcal{L} \geq 0$. The **Merkle-Damgård extender** *defines the hash function*

$$h' : \{0,1\}^l \to \{0,1\}^m$$

which computes $h'(M)$ as follows:

1. *Let the message be $M = M_0||M_1||M_2||\ldots||M_{\mathcal{L}}$ where $M_0 \in \{0,1\}^n$ and $M_i \in \{0,1\}^{n-m}$ for $i = 1, \ldots, \mathcal{L}$.*
2. *Let $x_1 = h(M_0)$ and iterate $x_{i+1} = h(x_i||M_i)$ for $i = 1, \ldots, \mathcal{L}$.*
3. *Then $h(M) = x_{\mathcal{L}+1}$.*

The process can be demonstrated by the following diagram:



Often the first message block is also initialized to $x_0 = 0^n$ (a string of zeros of length n) and the message is split up into blocks $x_1, \ldots, x_\mathcal{L}$ of equal length. If the hash function is "good", this does not make a difference, as 0^n is mapped to a random-looking string.

The message may not always have the correct length of $n+\mathcal{L}\cdot(n-m)$ for some $\mathcal{L}$ (or $\mathcal{L}\cdot(n-m)$ in the case just mentioned) to be able to be split up into blocks of the given length. In this case, *padding* is performed. Extra bits are appended to the end of the message in a specified way to make the final block the right length.

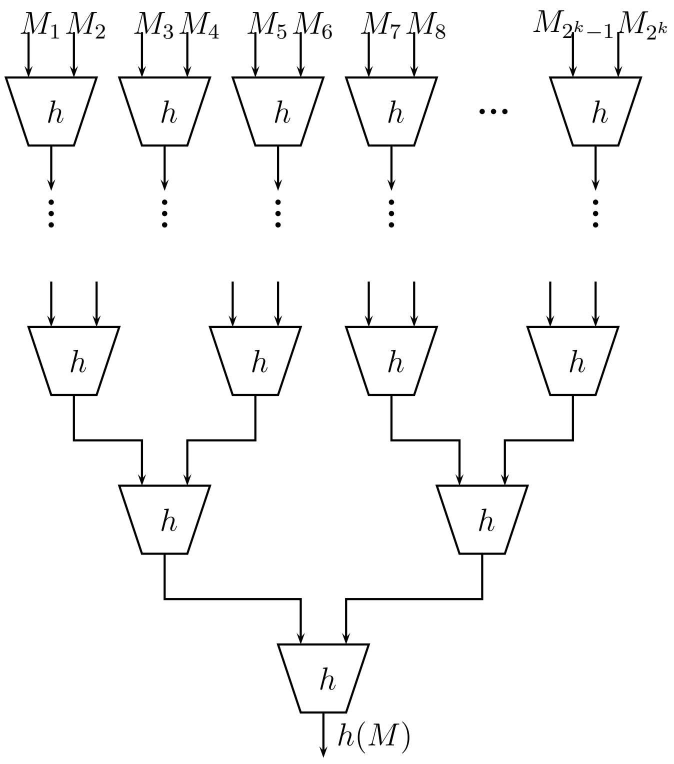
Padding is not a trivial process; there are many ways of doing it insecurely that open up opportunities for different attacks. Merkle and Damgård were aware of this, and they both suggested padding the message with zeros and then appending a binary representation of the length of the original message at the end. An extra block may have to be added if the message length more than fills up (does not fit) the last block. Although padding with zeros hides how many zeros were originally present at the end of the message (i.e. before adding the padding zeros), the length at the end allows the recovery of the original message. To make this process even more unambiguous, the length of the binary representation is often fixed, for example to 64 bits. It must be noted, however, that this puts at least a theoretical limit of 2^{64} bits on the total message length. This method of padding also prevents *length extension attacks* on hash functions used as MACs.

Another method of padding is adding a single bit 1 and then zeros and then the original message length, as it is done for MD5 (Section 2.1) and SHA-1 (Section 2.2).

These are considered the most secure ways of padding and are therefore the most widely used, but there are other methods. For a discussion see Preneel's PhD thesis [47].

The nice thing about the Merkle-Damgård extender is that it preserves collision resistance. If a collision resistant compression function h is used, then the resulting hash function h' is also collision resistant. This property and the efficiency of the extender have made it extremely popular and all hash functions used in practice today are based on it, even the ones for which collision resistance cannot be proven, such as MD5 and SHA-1.

It should also be mentioned that there is an even more efficient domain extender, also independently proposed by Merkle as well as Damgård in 1989, called *tree hashing* or *parallel hashing*. It can be used if several parallel processors are available and requires a compression function $h : \{0,1\}^{2m} \to \{0,1\}^m$. The message M is split into 2^k blocks $M_1, \ldots, M_{2^k}$ for some $k \in \mathbb{Z}^+$ (after appropriate padding), and each processor processes two blocks at a time. The procedure becomes clear from the diagram:



Tree hashing has also been proven to preserve collision resistance [41, 14].

3.4 Summary

Once different security properties have been proven for a hash function, it can be trusted to a much greater degree than any custom-designed hash function. Different types of provably secure hash functions may be used for different purposes. This chapter has laid the foundations for the following discussion of two of these types by giving a short overview of the theory of NP-completeness. It has also explained the Merkle-Damgård extender, a concept used in the design of many (custom-designed and provably secure) hash functions.

Chapter 4
Universal One-Way Hash Functions

After it became clear that collision resistant hash functions are very hard to design, Naor and Yung proposed a new cryptographic primitive they called *universal one-way hash functions* [43]. Although they achieve weaker security than CRHFs, they still suffice for many important cryptographic applications. Naor and Yung first used them to construct a signature scheme they called *one-way based secure digital signature scheme*, which they proved secure against the most general attack on digital signatures (i.e. existentially unforgeable under an adaptive chosen plaintext attack) assuming only the existence of a one-way function. Previously, all provably secure signature schemes had been based on the stronger mathematical assumption that trapdoor one-way functions exist. More specifically, they constructed a UOWHF from a one-way function and showed that it suffices for hashing messages prior to signing them with a digital signature scheme. It should be noted, however, that the signer can cheat by choosing the key before the target message, making this scheme vulnerable to attacks by the signers.

When they were first introduced in 1989, UOWHFs were more of theoretical interest and far away from any practical use. But since then, many UOWHF designs have been proposed and made more practical, and they are now considered an attractive alternative to CRHFs by many.

This chapter gives a more formal definition and explains as an example one of the most natural and promising designs of UOWHFs, the *subset sum hash function.*

4.1 Universal One-Way Hash Functions

Simply stated, a UOWHF is a family of hash functions, indexed by a key, that are one-way and second preimage resistant. More formally, security is defined in terms of an adversary algorithm which tries to find second preimages. The algorithm first has to commit to a challenge input and is then given a key which selects the hash function from the family. Then it attempts to produce a second preimage for that particular hash function.

Definition 4.1.1 (UOWHF)
Let m and n be positive integers with $m \leq n$. A (t, ε) **universal one-way hash**

function family *is a finite collection $\mathcal{H}$ of functions*

$$h_K : \{0,1\}^n \rightarrow \{0,1\}^m$$

indexed by a key $K \in \mathcal{K}$ (where $\mathcal{K}$ is the key space), and such that any attack algorithm $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ running in time t has success probability at most ε in the following game:

- **$\mathcal{A}$ Commits:** *$\mathcal{A}_1$ runs (with no input) and outputs a hash function input $s_1 \in \{0,1\}^n$ and a state.*
- **Key Sampling:** *A key $K \in \mathcal{K}$ is chosen uniformly at random and revealed to $\mathcal{A}$.*
- **$\mathcal{A}$ Collides:** *$\mathcal{A}_2$ runs with input K, s_1 and state and outputs a second hash function input $s_2 \in \{0,1\}^n$.*

$\mathcal{A}$ succeeds in the game if it finds a valid collision for h_K, i.e. if $s_1 \neq s_2$ but $h_K(s_1) = h_K(s_2)$.

Although this really defines a UOWHF *family*, it is often just referred to as a UOWHF.

The *state* in this definition simply means that $\mathcal{A}$ may keep a state throughout the entire game. $\mathcal{A}$ consist of two parts $\mathcal{A}_1$ and $\mathcal{A}_2$ that are executed one after the other but essentially, $\mathcal{A}$ keeps running and may thus remember some extra information. Formally, this *state* information is described as an output of $\mathcal{A}_1$ and an input to $\mathcal{A}_2$.

From this definition it becomes clear why the keys are needed. When using the concept of an attacker algorithm, the security wanted cannot be defined for a single hash function. The essence of the property of second preimage resistance is that the first input is *fixed* and an adversary has to find a collision for this given input. Since the adversary algorithm is allowed to *choose* the first input in this model, it cannot know the hash function at the time it commits (otherwise it could use information about the hash function to choose the input that would make it easier to find a second preimage). Only after the algorithm has committed to the first input can it know which hash function it will be using. That is why formally the property of second preimage resistance can only be defined for a family of hash functions. It is important that each function of the family is selected with equal probability, that is, uniformly at random from the family.

So essentially, this is simply another way of making sure that the attacker cannot choose the first input depending on the hash function. This is slightly different from the concept used for one-way hash functions, where the hash function is fixed and the first input is given to the algorithm, which then attempts to find a collision for that input.

Notice that the so-called keys are not secret keys. They do not refer to any secret knowledge, which is the common use of the term “key” in cryptography. The keys are rather the parameters that index the members of the family and are publicly known.

Also note that the members of this family are not proper hash functions in the sense that they take an input of fixed length n rather than of arbitrary length. This is where the Merkle-Damgård extender defined in Section 3.3 comes into play. It is in fact a very common methodology to first construct a so-called *compression function* which hashes a (relatively short) n-bit string to a shorter m-bit string, hence compressing the input by $n - m$ bits, and then using a domain extender (such as Merkle-Damgård) to build a hash function which compresses an input of arbitrary length to an m-bit output. The point of this system is that security can be proven in two steps: If it can be proven that a domain extender preserves certain properties of a compression function, then it remains only to show that the compression function has the desired properties.

The simplest and most natural such domain extender is the well-known and efficient Merkle-Damgård extender. Merkle and Damgård both proved independently [41, 14] that their extender preserves collision resistance. In other words, if a compression function is collision resistant, then the hash function obtained by applying the Merkle-Damgård extender to it is also collision resistant. Unfortunately it was shown by Bellare and Rogaway [4] that the Merkle-Damgård extender is not guaranteed to preserve second preimage resistance. Hence the construction cannot be used for extending a UOWHF in general.

Hong, Preneel and Lee [26] showed that there is a way of using the Merkle-Damgård extender to build hash functions with UOWHF security. They defined a stronger security for compression functions called *higher order UOWHF security* and showed that these higher order UOWHFs can be used to build hash functions with UOWHF security from the Merkle-Damgård extender.

Before giving the definition of higher order UOWHFs, we need to step back and look at ordinary UOWHFs again. Instead of selecting the key K after the algorithm $\mathcal{A}$ has committed to the first input, it is also possible to first select K without revealing it to $\mathcal{A}$, then letting $\mathcal{A}$ commit, and then giving $\mathcal{A}$ the selected key. Since the random selection of K is independent of $\mathcal{A}_1$'s behaviour, the order of these actions makes no difference to the success probability of $\mathcal{A}$, as long as the key is not revealed to the adversary until after it has committed to an input. Hence the game of Definition 4.11 is essentially equivalent to the following game:

- **Key Sampling:** A key $K \in \mathcal{K}$ is chosen uniformly at random (but not yet revealed to $\mathcal{A}$).
- **$\mathcal{A}$ Commits:** $\mathcal{A}_1$ runs (with no input) and outputs a hash function input $s_1 \in \{0,1\}^n$ and a *state*.

- **Key Revealed:** The key K is given to $\mathcal{A}$.
- **$\mathcal{A}$ Collides:** $\mathcal{A}_2$ runs with input K, s_1 and *state* and outputs a second hash function input $s_2 \in \{0,1\}^n$.

When considering UOWHF security in terms of this game, a simple extension can be made: The adversary $\mathcal{A}$ is allowed to make r adaptive queries to an oracle for the selected hash function h_K before committing. This means that without knowing which function h_K is being used, the adversary can give r inputs q_i to the oracle and will receive the values $h_K(q_i)$ for $i = 1, \ldots, r$ in return. Each query may depend on the results of all previous queries (this is what is meant by "adaptive"). $\mathcal{A}$ may then use this extra information obtained from the queries to choose the first input. A function that is secure even under this stronger attack is called an r^{th} *order UOWHF* and also denoted by UOWHF(r).

Definition 4.1.2 (r^{th} order UOWHF (UOWHF(r)))
Let m and n be positive integers with $m \le n$. A (t, ε) r^{th} **order universal one-way hash function family** *is a finite collection $\mathcal{H}$ of functions*

$$h_K : \{0,1\}^n \to \{0,1\}^m$$

indexed by a key $K \in \mathcal{K}$ (where $\mathcal{K}$ is the key space), and such that any attack algorithm $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ running in time t has success probability at most ε in the following game:

- **Key Sampling:** *A uniformly random key $K \in \mathcal{K}$ is chosen (but not yet revealed to $\mathcal{A}$).*
- **Oracle Queries:** *$\mathcal{A}_1$ runs (with no input) and makes r adaptive queries $q_1, \ldots, q_r \in \{0,1\}^n$ to an oracle for h_K, receiving answers $y_1, \ldots, y_r \in \{0,1\}^m$ where $y_i = h_K(q_i)$ for $i = 1, \ldots, r$.*
- **$\mathcal{A}$ Commits:** *$\mathcal{A}_1$ outputs a hash function input $s_1 \in \{0,1\}^n$ and a state.*
- **Key Revealed:** *The key K is given to $\mathcal{A}$.*
- **$\mathcal{A}$ Collides:** *$\mathcal{A}_2$ runs with input K, s_1 and state and outputs a second hash function input $s_2 \in \{0,1\}^n$.*

$\mathcal{A}$ succeeds in the game if it finds a valid collision for h_K, i.e. if $s_1 \neq s_2$ but $h_K(s_1) = h_K(s_2)$.

Clearly a 0^{th} order UOWHF is just a normal UOWHF. Also, an r^{th} order UOWHF is automatically a t^{th} order UOWHF for any $t \le r$. The converse, however, is not true. A UOWHF is not necessarily a higher order UOWHF; in fact there exist UOWHFs which are not even first order UOWHFs [4]. It is also interesting to note that Hong, Preneel and Lee [26] show that the classes of UOWHFs of same order form a chain between CRHF and UOWHF classes.

Using this concept of r^{th} order UOWHFs, Hong, Preneel and Lee proved that if the order of the underlying UOWHF is r, then the $(r+1)$-round Merkle-Damgård extension is also a UOWHF. Here $(r+1)$ is often called the *extension factor*.

Note that there are other domain extender algorithms which preserve UOWHF security without the higher order property. Bellare and Rogaway [4] proposed two types of constructions, one with a linear structure which was later improved by Shoup [62] and is usually called the *Shoup XOR-mask domain extender*, and the other with a tree structure which extended the work of Naor and Yung [43] and was further improved by Sarkar [54, 55] and Lee, Chang, Lee, Sung and Nandi [35]. However, these extenders all have the undesirable property that the length of the key increases with the length of the message. Hence it is still preferable to use Merkle-Damgård where possible, since it keeps the key size constant.

Finally, after having taken in the above concepts, a more precise definition of *security* can be stated. Given an adversary algorithm $\mathcal{A}$ which succeeds in a specified game with probability ε in running time t, security is high if $\mathcal{A}$ succeeds with low probability even if it is given a lot of time, hence if ε is small and t is large, or $\frac{t}{\varepsilon}$ is big. Similarly, security is low if $\mathcal{A}$ succeeds with high probability in little computation time, hence if ε is large and t is small, or $\frac{t}{\varepsilon}$ is small. Most authors scale this ratio by taking the (base 2) logarithm of the run-time/success probability ratio.

Definition 4.1.3 (security)
The **security of a UOWHF** *is defined as* $\log_2(\frac{t}{\varepsilon})$, *where* t *is the running time and* ε *is the success probability of an attacker algorithm in the game from Definition 4.1.1.*
The **security of a UOWHF(r)** *is defined as* $\log_2(\frac{t}{\varepsilon})$, *where* t *is the running time and* ε *is the success probability of an attacker algorithm in the game from Definition 4.1.2.*

4.2 The Subset Sum Hash Function

Do universal one-way hash functions really exist? Yes, many constructions have been proven to be UOWHFs or even higher order UOWHFs. One natural construction that has recently been proven to achieve higher order UOWHF security is the *subset sum hash function*. It is based on the well-known NP-complete subset sum problem.

The subset sum problem is: Given a set a of n numbers, each m bits long, and a target sum T, find a subset $s \subseteq a$ whose sum is T. Addition is performed modulo an m-bit integer p.

Definition 4.2.1 (Subset Sum Problem ($\mathsf{SubSum}(n, m, p)$))
Let $m, n, p \in \mathbb{Z}^+$ *with* $m < n$ *and* $2^{m-1} < p \leq 2^m$. *Given a uniformly random*

vector of integers $a = (a_1, \dots, a_n) \in \mathbb{Z}_p^n$ and a target integer T chosen uniformly at random from $\mathbb{Z}_p$, find a subset $s = (s_1, \dots, s_n) \in \{0,1\}^n$ such that

$$\sum_{i=1}^{n} s_i a_i \equiv T \pmod{p}.$$

The condition $2^{m-1} < p \leq 2^m$ ensures that $p-1$ is exactly m bits long and therefore addition modulo p always results in an integer of m bits.

The subset sum problem is a special case of the *knapsack problem* and was first shown to be NP-complete by Richard Karp in 1972 [30] by transformation from the *partition problem.* Karp proved the NP-completeness of 21 well-known combinatorial and graph theoretical problems, each infamous for their intractability, just one year after the first NP-complete problem (the *boolean satisfiability problem*) had been demonstrated by Cook [10].

The first to suggest using subset sum in cryptography were Merkle and Hellman in 1978 [42], and many schemes in public key cryptography have since been based on the subset sum problem. Since computing subset sums only involves addition, these schemes are much more efficient than schemes based on the intractability of number theoretic problems such as factoring and discrete logarithms, which involve multiplication or even exponentiation. However, none of these schemes have been proven to be as secure as subset sum (meaning it could not be proved that breaking them is as hard as solving the subset sum problem), and, in fact, most of them have been broken.

Impagliazzo and Naor [28] were the first to use the subset sum problem to construct a hash function (and a pseudo-random generator). They describe their approach of "only" constructing a hash function rather than an entire public key cryptosystem "less ambitious", since "many important tasks in cryptography [such as digital signatures and pseudo-random generation] do not require the full power of public key cryptography".

The subset sum hash function is defined as follows:

Definition 4.2.2 (subset sum hash function $\mathcal{H}_{SS}(n, m, p)$)
Let $m, n, p \in \mathbb{Z}^+$ with $m < n$ and $2^{m-1} < p \leq 2^m$. The **subset sum hash function family** *$\mathcal{H}_{SS}(n, m, p)$ is the collection of functions*

$$h_a : \begin{cases} \{0,1\}^n \to \{0,1\}^m \\ s \mapsto h_a(s) := \sum_{i=1}^{n} s_i a_i \pmod{p} \end{cases}$$

where

- *$a = (a_1, \dots, a_n) \in \mathbb{Z}_p^n$ is the key (hence $\mathcal{K} = \mathbb{Z}_p^n$ is the keyspace) and*
- *$s = (s_1, \dots, s_n) \in \{0,1\}^n$ is the message or hash function input.*

It is clear immediately that inverting any hash function $h_a \in \mathcal{H}_{SS}$ is as hard as solving the subset sum problem. The key corresponds to the set of numbers, the message corresponds to the subset, and the hash value corresponds to the target integer. Given the key and a hash value, it is hard to find a message that hashes to the given value. That makes the subset sum hash function one-way in a natural way.

More than that, assuming the hardness of the subset sum problem, the subset sum hash function $\mathcal{H}_{SS}$ was also proven to be a UOWHF by Impagliazzo and Naor [28]. Their exact result can be stated as follows.

Theorem 4.2.3 (Impagliazzo-Naor)
If the subset sum problem $\mathsf{SubSum}(n, m, p)$ *is* (t, ε)*-hard, then the subset sum hash function family* $\mathcal{H}_{SS}(n, m, p)$ *is a* (t', ε') *universal one-way hash function family, where* $t' = t - O(mn)$ *and* $\varepsilon' = 2n\varepsilon$.

Steinfeld, Pieprzyk and Wang [64] further strengthened this result by showing that the subset sum hash function family $\mathcal{H}_{SS}$ is actually an r^{th} order UOWHF for small $r = O(\log m)$, still assuming only the hardness of the subset sum problem $\mathsf{SubSum}(n, m, p)$. More concretely, they bounded the way the security of $\mathcal{H}_{SS}$ as an r^{th} order UOWHF deteriorates with increasing r. Their exact result is

Theorem 4.2.4 (Steinfeld-Pieprzyk-Wang)
Let $m, n, p \in \mathbb{Z}^+$ *with* $m < n$, p *a prime satisfying* $2^{m-1} < p \leq 2^m$, *and* $r < \log_3(p) - 1$. *If the subset sum problem* $\mathsf{SubSum}(n, m, p)$ *is* (t, ε)*-hard, then the subset sum hash function family* $\mathcal{H}_{SS}(n, m, p)$ *is a* (t', ε') r^{th} *order universal one-way hash function family, where*

$$t' = t - O(r^2 n \cdot T_M(p)) \qquad \textit{and} \qquad \varepsilon' = 2^{r+1}(n - r) \cdot \varepsilon + \tfrac{3^{r+1}}{2^m}$$

and $T_M(p)$ *denotes the time to perform a multiplication modulo* p.

In other words, the function's security as an r^{th} order UOWHF deteriorates by (at most) about r bits relative to the UOWHF case $r = 0$. This can now be combined with the result of Hong, Preneel and Lee [26] to conclude that the Merkle-Damgård extender can be applied to the subset sum compression function with extension factor $r + 1$ while losing (at most) about r bits of security.

The functions in the subset sum hash function family $\mathcal{H}_{SS}(n, m, p)$ hash an n-bit input to an m-bit output. Hence $\mathcal{H}_{SS}(n, m, p)$ can be used as a compression function family to construct a hash function family $\mathcal{H}'_{SS}(l, m)$ which hashes an l-bit input to an m-bit output, where l could be much larger than n.

The Merkle-Damgård extender can be applied as follows. First the message needs to be padded to a length of $l = n + \mathcal{L} \cdot (n - m)$ bits, where $\mathcal{L}$ is a positive integer. This means that the message can be divided into a total of $\mathcal{L} + 1$ blocks, where the first block has n bits and all other blocks have $n - m$ bits.

So assume that $l = n + \mathcal{L} \cdot (n - m)$ and define the Merkle-Damgård family $\mathcal{H}'_{SS}(l, m)$ as follows:

Definition 4.2.5 ($\mathcal{H}'_{SS}(l, m)$)
Let $l = n + \mathcal{L} \cdot (n - m)$ where $\mathcal{L} \in \mathbb{Z}_+$ and n, m, p as before. A key $a \in \mathbb{Z}_p^n$ of $\mathcal{H}'_{SS}(l, m)$ is just a uniformly random key of $\mathcal{H}_{SS}(n, m, p)$. The Merkle-Damgård hash function family $\mathcal{H}'_{SS}(l, m)$ is the family of functions h'_a. An input message $M \in \{0, 1\}^l$ is hashed using the function h'_a as follows:

1. *Split M into one n-bit block $x_0 \in \{0, 1\}^n$ and $\mathcal{L} = \frac{l-n}{n-m}$ $(n - m)$-bit blocks $M_1, \ldots, M_{\mathcal{L}}$.*
2. *For $i = 1, \ldots, \mathcal{L}$, compute $x_{i+1} = h_a(x_i || M_i)$.*
3. *Return $x_{\mathcal{L}+1} \in \{0, 1\}^m$.*

Since Steinfeld, Pieprzyk and Wang [64] have shown that $\mathcal{H}_{SS}(n, m, p)$ is an r^{th} order UOWHF for some $r > 0$, $r = O(\log m)$, the result [26] by Hong, Preneel and Lee leads to the conclusion that the family $\mathcal{H}'_{SS}(l, m)$ is a UOWHF for $l \leq (r + 1)(n-m)$, assuming only the hardness of the subset sum problem $\mathsf{SubSum}(n, m, p)$.

As promising as these results sound, there are several problems that do not quite allow the subset sum hash function to be used yet. One of them has already been mentioned and concerns the length of the message. Hash functions are required to hash messages of arbitrary length, but $\mathcal{H}'_{SS}(l, m)$ restricts the input length l to at most $r + 1$ blocks of length $n - m$. Since the security of an r^{th} order UOWHF drops by approximately r bits (compared to the corresponding 0^{th} order UOWHF), the security of $\mathcal{H}'_{SS}(l, m)$ drops by one bit for every round in the Merkle-Damgård extension and therefore for every block of the message. That leads to a trade-off between message length and security and may make the message length quite small if one desires to keep reasonable security. Also, since $r = O(\log m)$, longer messages will require a longer hash value.

Another problem is the key size. In the compression function $\mathcal{H}_{SS}(n, m, p)$, each of the n message bits requires one m-bit number in the key. Hence the key $a = (a_1, \ldots, a_n)$ consists of n m-bit numbers and is therefore $n \cdot m$ bits long. Furthermore, Steinfeld, Pieprzyk and Wang [64] show that $m \geq 1352$ is needed to achieve reasonable security (more concretely, to make the success probability of a dangerous attack (a *lattice attack*) less than 2^{-80}). Since $m < n$, that results in a key size of more than $1352^2 = 1827904$ bits. This is better than the key size that results from using one of the other domain extenders, as Steinfeld, Pieprzyk and Wang [64] have shown, but it is still very long.

Finally, there remains the general problem with relating the security of a hash function to an NP-complete problem. An NP-complete problem is hard in the *worst case*, but not necessarily hard in *every case*. Some or even many instances of an NP-complete problem may be easy to solve; NP-completeness simply says that there is no efficient algorithm which solves all cases. The subset sum problem can for example be solved easily if the target number just happens to be zero or one of the numbers in a, to give a very obvious example. This is not directly a problem, but before using the subset sum hash function, one should at least be sure that the *average case* is hard (and not just the worst case). It makes sense to think that most cases will be hard, but we could not find any work on the proportion of "easy" instances of the subset sum problem. This might be worth further investigation.

There exist very good approximation algorithms for the subset sum problem. However, an approximation is not good enough. To find a valid preimage of a given hash, one needs to find a subset that sums up *exactly* to that value, and not one that sums up to a value close to the target. But does a subset that approximates an integer look very similar to a subset that actually gives the integer? Also, if any sum of numbers can be specified with at most m bits, then the polynomial time approximation algorithm for subset sum can be turned into an exact algorithm (i.e. one that solves subset sum) with running time polynomial in n (the number of numbers in the set) and 2^m [65]. This is an exponential-time algorithm and may not be efficient enough to be of any use, depending on the size of m. Still, these problems should be further investigated.

Although more work has to be done to make UOWHFs and in particular the subset sum hash function usable in practice, the approach seems promising. Subset sum is rather efficient and at least there is a concrete and proven idea of how secure it is, even if there are some issues left to research.

Chapter 5

A New Preimage Resistant Hash Function

After having thoroughly investigated the techniques used to design hash functions and prove different security properties, we now propose a new hash function. There are several reasons motivating the design.

Firstly, creating something new is much more interesting than just investigating what other people have done. After all, good new hash functions are desperately needed, and finding one could potentially be a significant contribution to the research on cryptographic hash functions.

Secondly, seeing the natural way in which the subset sum problem could be turned into a hash function led to the question whether there are any other NP-complete problems that could serve the same purpose. In the very elaborate list of several hundred NP-complete problems in [20], I found the graph theory problems particularly inspiring and thought that the Hamiltonian cycle problem might be a suitable choice. Out of that came the design of a new hash function, which is based on this problem. It is called HamHash for obvious reasons.

Upon investigating the nature of HamHash and continually modifying the construction to achieve certain properties, it turned out to be a very unconventional design. It is provably a PRHF (recall from Chapter 3 that a PRHF is *preimage resistant*). HamHash is also non-deterministic: Every time the hash is applied to a message, it may produce a different result. This is definitely an unconventional way of hashing. In fact we are not aware of any previous work on hash functions which are only preimage resistant, or on non-deterministic hashing (besides the Digital Signature Algorithm (DSA), which is related in the sense that it is non-deterministic, but it is not directly a hash function). Still, these properties are sufficient for many authentication purposes such as password storage. The weak security assumption makes them much easier to design than CRHFs, OWHFs or even UOWHFs, and they become an attractive alternative.

This chapter explains the construction of HamHash and examines its properties. It also presents some possible applications and a sample implementation.

5.1 Graph Theory

A quick summary of the graph theory background needed for the construction of HamHash seems appropriate at this point.

Definition 5.1.1 (graph)
A **graph** G *is an ordered pair* $G = (V, E)$ *where*

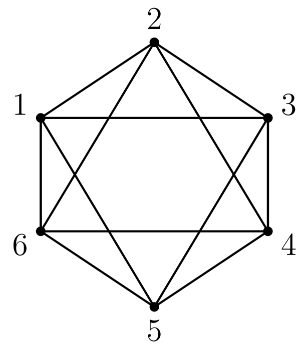
- $V = \{1, \dots, v\}$ *is a finite set of* v **vertices** *and*
- E *is a finite set of unordered pairs* $\{u, w\}$*, called* **edges***, of distinct vertices* $u, w \in V$.

Note that in the more general literature on graph theory this would be referred to as a

- finite (V and E are *finite*),
- undirected (edges are *unordered* pairs of vertices),
- simple (no loops, i.e. edges are *distinct* pairs of vertices, and no repeated edges, i.e. E is not a multiset), and
- labelled (the vertices are *labelled* $1, \dots, v$)

graph. For the purposes of this thesis however, all graphs will be of this kind and therefore only be called *graphs*. The number of vertices is always denoted by v.

Graphs are often represented as drawings, where vertices are black dots and edges are lines joining two vertices.



A graph can also be represented as an **adjacency matrix**. For a graph $G = (V, E)$ on v vertices this is simply a matrix $A = (a_{ij}) \in \{0, 1\}^{v \times v}$ of size $v \times v$, where an entry $a_{ij} = 0$ if the edge $\{i, j\}$ is not present and $a_{ij} = 1$ if $\{i, j\} \in E$. For example, the above graph can be represented by the adjacency matrix:

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

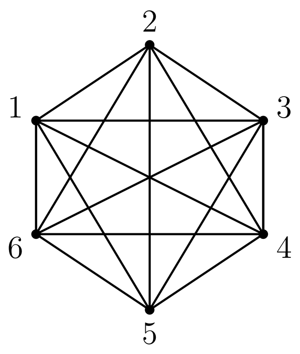
Note that for an undirected graph, the adjacency matrix is always symmetric, as $\{i, j\} \in E$ if and only if $\{j, i\} \in E$. The entries a_{ii} on the diagonal are always zero, since loops are not allowed.

Definition 5.1.2 (degree)
Let $G = (V, E)$ be a graph. A vertex $u \in V$ has **degree** *d if it is incident with d edges, that is, if there are d edges $\{u, w\}$ in E.*

For example, every vertex in the above graph has degree 4. Therefore, the graph would be referred to as **4-regular**.

Definition 5.1.3 (complete graph)
A **complete graph** *is one in which all possible edges are present.*

The complete graph on six vertices is:



Definition 5.1.4 (path, cycle)
A **path** *of length l in the graph $G = (V, E)$ is a sequence $u_0u_1 \dots u_l$ of distinct vertices $u_0, \dots, u_l \in V$ (i.e. no vertex is visited more than once) where $\{u_i, u_{i+1}\} \in E$ for $i = 0, \dots, l-1$. If $l \geq 2$, $u_0 = u_l$, and all other vertices are distinct, then $u_0u_1 \dots u_l$ is called a* **cycle**.

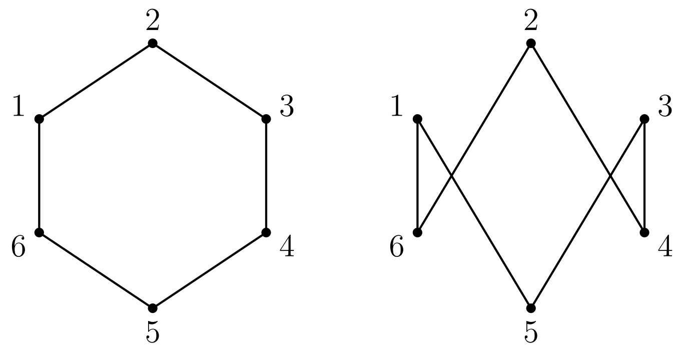
A path of length 4 in the star graph would be 12346. An example of a cycle is 2462.

5.1.1 The Hamiltonian Cycle Problem

Definition 5.1.5 (Hamiltonian path)
A path that visits every vertex in a graph exactly once is a **Hamiltonian path**.

Definition 5.1.6 (Hamiltonian cycle)
A cycle that visits every vertex in a graph exactly once is a **Hamiltonian cycle**.

Two obvious Hamiltonian cycles in the star graph are:



Hamiltonian cycles are named after the Irish mathematician, physicist and astronomer Sir William Rowan Hamilton who invented the *icosian game* (also called *Hamiltonian game* or *Hamilton's puzzle*), which involves finding a Hamiltonian cycle along the edges of a dodecahedron, in 1857 [27].

Hamilton already knew that finding a Hamiltonian cycle in a given graph is a non-trivial task. In fact, this problem is NP-complete.

Definition 5.1.7 (Hamiltonian cycle problem $\mathsf{HamCycle}(v)$)
Given a graph G on v vertices, find a Hamiltonian cycle in this graph.

This is among the first 21 problems proven to be NP-complete by Richard Karp in 1972 [30], by transformation from the *vertex cover problem.* It has also been proven that the Hamiltonian cycle problem remains NP-complete

- if G is planar, cubic, 3-connected, and has no face with fewer than 5 edges [19],
- if G is bipartite [34],
- if G is the square of a graph [7], and
- if a Hamiltonian path for G is given as part of the instance [45].

The problem is solvable in polynomial time, however, if G has no vertex with degree exceeding 2 or if G is an line graph [37].

5.1.2 Random Graphs

In applications it is often interesting to know how an algorithm (or hypothesis) works on a "typical" instance: a random graph. A random graph is produced by starting out with a set $V = \{1, \ldots, v\}$ of vertices and then adding edges between these at random. Different random graph models produce different probability distributions on graphs. The most natural, most common and most useful model for the purposes of this thesis is the **binomial model** introduced by Erdös [18]. It is called $G(v, p)$ and includes each possible edge independently with probability p. This means that the probability of a given graph G is

$$\text{Probability}(G) = p^{|E(G)|} \cdot (1-p)^{\binom{v}{2} - |E(G)|}$$

where $|E(G)|$ denotes the number of edges in G. Notice that for $p = \frac{1}{2}$ every graph $G \in G(v,p) = G(v, \frac{1}{2})$ has the same probability:

$$\text{Probability}(G) = \left(\frac{1}{2}\right)^{|E(G)|}$$

So $G(v, \frac{1}{2})$ represents the uniform probability space on the set of graphs with v vertices.

5.1.3 Graph Theory and Hashing

Little work has been done on graph theory related hash functions. Some constructions based on special types of graphs such as expander graphs, Cayley graphs, graphs with large girths and Ramanujan graphs have been proposed [49, 9, 68], but none of them are used in practice. Their security relies on special properties of these graphs rather than NP-complete problems.

Charles, Goren and Lauter [9] have an interesting design, where each vertex in a k-regular sparse graph has a label, and the message is used to determine a walk in the graph. The output is the label of the final vertex in the walk, and collision resistance is based on the fact that finding distinct paths between vertices is hard in sparse graphs, where the edge-vertex ratio is small.

HamHash bases its security on the properties of Hamiltonian graphs (specifically the hardness of finding a Hamiltonian cycle in a Hamiltonian graph) in a much more natural way.

5.2 The Algorithm HamHash

Now it is time to present the technical details of HamHash. As mentioned before, it is a new design, invented as a part of the work on this thesis, and based on the Hamiltonian cycle problem.

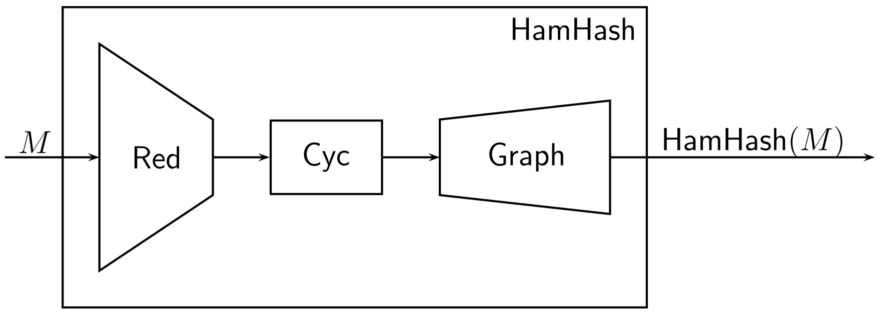
5.2.1 The Main Idea

As a proper hash function, HamHash takes an input message (bit string) of arbitrary length and outputs a hash value of fixed length. The output is a graph in the form of an adjacency matrix, which can be rearranged to form a bit string if desired. The size of the output depends on the size of the graph, more specifically on the number of vertices v in the graph, which must be fixed in advance.

The basic idea is as follows. The input message is first mapped to a Hamiltonian cycle on v vertices in a unique way. The core part of the algorithm, and the part on which its security is based, then constructs a graph that contains this given Hamiltonian cycle.

Later on, it can be verified that a hash belongs to a given message by recomputing the corresponding Hamiltonian cycle from the message and checking that it is contained in the graph given by the hash value. Conversely, finding a preimage given only a hash value requires finding the Hamiltonian cycle in the graph, and is therefore very hard.

The functioning of the algorithm can best be described in the following diagram:



Since a Hamiltonian cycle on v vertices can be represented by fewer bits than an entire graph on v vertices, the construction of interpreting a message as a cycle and then turning it into a graph would produce an output of bigger size than the input and therefore not be a proper hash function. Hence we first apply a reduction function Red, which reduces the arbitrary size message M to an output that has an fixed size appropriate to represent a Hamiltonian cycle in the graph. From this output, the Hamiltonian cycle is computed by the function Cyc. Finally, Graph adds the edges in this cycle and more random edges to an (initially empty) graph H, the adjacency matrix of which becomes the output of the entire hash function $\mathsf{HamHash}$.

5.2.2 Notation

Before having a closer look at the three functions Red, Cyc and Graph, we need to define some sizes and assign some names. These are valid for the rest of this chapter.

- $M \in \{0,1\}^*$ is the input message to the hash function $\mathsf{HamHash}$ and therefore also the input to the reduction function Red.
- $H = (V, E)$ is the graph that $\mathsf{HamHash}$ works with. H does not show up in the actual algorithm (only the adjacency matrix that represents it). It is only the idea behind the algorithm, but it is often helpful to be able to refer to the graph directly.
- v is the number of vertices in H. The vertices are labelled $1, \ldots, v$, i.e. $V = \{1, \ldots, v\}$.
- $m = \binom{v}{2} = \frac{1}{2}v(v-1)$ is the number of possible edges in H.
- Let c denote the number of possible Hamiltonian cycles in a complete graph $C = (V, E)$ on v vertices. Since $V = \{1, \ldots, v\}$, each Hamiltonian cycle can be represented as a permutation of the numbers $1, \ldots, v$. There are $v!$ such permutations. However, $2v$ such permutations represent the same

cycle (since it does not matter which of the v vertices is picked as the start vertex, and if we walk around the cycle forwards or backwards). Hence $c = \frac{v!}{2v} = \frac{1}{2}(v-1)!$.

- $\mathsf{HamCycles}(v)$ denotes the set of all Hamiltonian cycles in the complete graph C and has c elements.
- $n = \lfloor \log_2(c) \rfloor$ is the size of the output of Red and the input to Cyc.
- $A \in \{0,1\}^{v \times v}$ is the adjacency matrix that represents H. Since H is an undirected graph (i.e. A is symmetric) and has no loops (i.e. the diagonal of A contains all zeros), A contains some redundant information that should be left out, since the output of HamHash should be as small as possible. More specifically, H can be uniquely represented by the bottom left triangle (without the diagonal) of A, so let $A_\triangle = (a_{ij})_{i>j}$ be the lower triangular matrix of A (without the diagonal). Technically, $A_\triangle$ is the output of HamHash and also the output of Graph. When it does not matter or when it is clear what is meant, however, H, A and $A_\triangle$ may be used interchangeably to describe the hash value.
  $A_\triangle$ contains exactly one entry for each possible edge in H. Therefore $A_\triangle$ can be rearranged to form a string of length m in a bijective way (e.g. by concatenating the rows of the matrix). Hence we may write $A_\triangle \in \{0,1\}^m$ and use it to refer to both the lower triangular matrix and the string, leaving this trivial task to be performed where necessary. For the purposes of this thesis it is usually easier to look at the matrix rather than the string.

5.2.3 The Function Red

The reduction function

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

makes up the first part of the hash algorithm HamHash and takes the arbitrary length message M to a bit string $\mathsf{Red}(M)$ of length n. The purpose of this function is simply to reduce the size of the message, and any function that achieves this may be chosen by the implementor of the algorithm. This section only makes some suggestions.

It will later be shown (see Section 5.3.2) that the preimage resistance of HamHash does not depend on the properties of Red. Still it turns out that there are better and worse choices. Although the properties of Red are not used in any security proof, it is still a good idea to eliminate any weaknesses here if possible, and to achieve extra properties (e.g. avalanche effect) that are desired but need not be proven.

For example, simple truncation of M to length n (i.e. taking the first n bits of M) would do the job. However this would mean that everything after the n^{th} bit has no influence on the hash value, thus not achieving a satisfactory avalanche effect and also making it trivial to produce collisions (by simply altering any part of the message after the n^{th} bit).

A much better choice is any non-cryptographic hash function with digest size $\geq n$ (simply truncate the digest if it is greater than n bits). This already jumbles up the message (and the more a hash function jumbles up a message, the better) and takes every part of it into account.

The perhaps best and recommended choice for Red is one of the custom-designed cryptographic hash functions, such as MD5, SHA-1, SHA-256 or even SHA-512, depending on the necessary output size. These have been shown to achieve a very good avalanche effect and although they have been broken for collisions and may soon be broken for preimages and second preimages (really, who knows), finding these still requires a fairly high computational effort, even though it is not impossible.

For the observant reader who may be wondering what Cyc and Graph are for if Red is already a cryptographic hash, let us stress again that the aim here is to produce a hash function with *provable* preimage resistance. This cannot be done for any custom-designed hash function, but it is nevertheless true for HamHash.

Lastly, the rather obvious fact that Red must be efficiently computable should be noted. Hash functions are generally desired to be efficient, and the efficiency of HamHash also depends on that of Red. This is another reason why custom-designed hash functions are a good choice: They can be evaluated very efficiently on 32-bit machines.

5.2.4 The Function Cyc

Although

$$\mathsf{Cyc} : \{0,1\}^n \to \mathsf{HamCycles}(v)$$

is the most trivial part of the algorithm, it is the most complicated to explain, since great care must be taken when converting n-bit strings to Hamiltonian cycles on v vertices. The reason for this is again that any possible weaknesses have to be eliminated at any stage in the design. In this case we try to eliminate trivial collisions by mapping the set of n-bit strings injectively to the set of Hamiltonian cycles. In other words, every input is mapped to a *different* cycle, making collisions impossible. To see how this can be achieved, a unique notation for each such cycle must first be defined.

If the tuple $(p_0, \ldots, p_{v-1})$ denotes a permutation of the numbers $\{1, \ldots, v\}$, then this also describes a Hamiltonian cycle $p_0 \ldots p_{v-1} p_0$ in the graph on the vertices $\{1, \ldots, v\}$.

However, this description is not unique. The tuples $(p_0, p_1, \ldots, p_{v-1})$ and $(p_1, p_2, \ldots, p_{v-1}, p_0)$ describe the same cycle, for example. Such cases can be ruled out by fixing the first vertex, that is, by representing a cycle as a tuple $(1, p_1, \ldots, p_{v-1})$ where $\{p_1, \ldots, p_{v-1}\} = \{2, \ldots, v\}$.

Still, even with the first entry fixed to be 1, there are still two descriptions for each cycle. For example, $(1, p_1, \ldots, p_{v-1})$ describes the same cycle as $(1, p_{v-1}, p_{v-2}, \ldots, p_1)$, only "backwards". This can be ruled out by requiring $p_1 > p_{v-1}$. This fixes the "direction" in which we "walk" around the cycle by requiring that from vertex 1, we always go in the direction of the adjacent vertex with the bigger number.

Hence each Hamiltonian cycle in a graph on v vertices can be described in a unique way by a tuple $(p_0, p_1, \ldots, p_{v-1})$ where

- $\{p_0, \ldots, p_{v-1}\} = \{1, \ldots, v\}$,
- $p_0 = 1$, and
- $p_1 > p_{v-1}$,

which represents the cycle $p_0 p_1 \ldots p_{v-1} p_0$. While for $p_1, \ldots, p_{v-2}$ all values $2, \ldots, v$ are allowed, p_{v-1} may only take values which are less than p_1. This description will be used from now on.

The set of all Hamiltonian cycles in a complete graph on v vertices can now be described as

$$\begin{aligned}\mathsf{HamCycles}(v) = \{(p_0, \ldots, p_{v-1}) \mid \ & \{p_0, \ldots, p_{v-1}\} = \{1, \ldots, v\}, \\ & p_0 = 1, p_1 > p_{v-1}\}.\end{aligned}$$

Next we can define an ordering on the set $\mathsf{HamCycles}(v)$. More specifically, the elements are put in lexicographic order. This means they are first sorted by the element p_1 (since p_0 is fixed). All the tuples with the same p_1 are then sorted by p_2, etc. All sorting is done in ascending order.

Now we can finally return to the problem we are really trying to solve: describing an injective function

$$\mathsf{Cyc} : \{0,1\}^n \to \mathsf{HamCycles}(v)$$

which maps bit strings of length n to the set $\mathsf{HamCycles}(v)$. Every such bit string can be interpreted as the binary representation of a number between 0 and $2^n - 1$. Let d be the decimal number represented by $\mathsf{Red}(M)$, which is the input to the function Cyc. All that is left to do then is to map d to the d^{th} element in the set $\mathsf{HamCycles}(v)$ (where we start counting elements in $\mathsf{HamCycles}$(v) from zero), and we are finished (with Cyc).

Recall that c was defined to be the size of $\mathsf{HamCycles}(c)$, and $n = \lfloor \log_2(c) \rfloor$. Hence $n \leq \log_2(c)$ and therefore $2^n \leq c$, so $\{0,1\}^n$ will be mapped to a subset of $\mathsf{HamCycles}(v)$, more specifically to the first 2^n elements of $\mathsf{HamCycles}(v)$.

We now go on to describe the algorithm that computes this mapping. It uses

- a lookup table for the pairs of elements p_1, p_{v-1}. This is much easier than giving an algorithm that calculates the mapping, and the table only has $\frac{1}{2}(v-1)(v-2)$ elements, which is a size that can be handled by any computer program. The elements t_{ij} of the table can be computed as follows.

```
        $k = 0$;
        for($i$ =3; $i \leq v$; $i$++){
            for($j = 2$; $j < i$; $j$++){
                $t_{k1} = i$;
                $t_{k2} = j$;
                $k$++;
            }
        }
```

  The elements t_{k1} are used to determine the entry p_1, and the t_{k2} determine p_{v-1}.
- The algorithm also uses an indexed list a of unused elements. It is initialised as $a = (2, 3, \ldots, v) = (a_0, \ldots, a_{v-4})$ with the elements p_1 and p_{v-1} missing, since they have already been used. Once another element is used, it is removed from the list and all other elements "move up", that is, if a_i is removed, then a_{i+1} becomes the new a_i, etc.

The algorithm now works as follows.

```
        $p_0 = 1$;
        $p_1 = t_{\lfloor \frac{d}{(v-3)!} \rfloor, 1}$;
        $p_{v-1} = t_{\lfloor \frac{d}{(v-3)!} \rfloor, 2}$;
        $d = d \bmod (v-3)!$;
        $a = (2, 3, \ldots, v)$ without $p_1$ and $p_{v-1}$;
        for($i = v - 4$; $i \geq 0$; $i$--){
            $p_{v-i-2} = a_{\lfloor \frac{d}{i!} \rfloor}$;
            remove $p_{v-i-2}$ from $a$;
            $d = d \bmod i!$;
        }
```

5.2.5 The Function Graph

The function

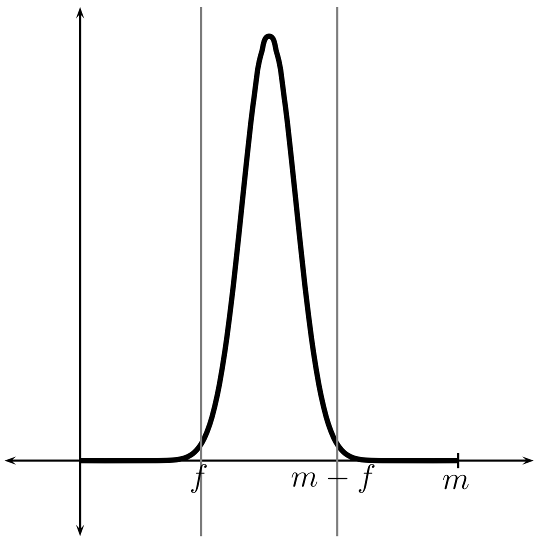
$$\mathsf{Graph} : \mathsf{HamCycles}(v) \to \{0,1\}^m$$

finally takes the Hamiltonian cycle $p = (p_0, \ldots, p_{v-1})$ computed by Cyc and constructs the graph H from it.

The algorithm starts by initialising the adjacency matrix A. Then the edges in the Hamiltonian cycle p must of course be added to A. Next more random edges are added. The subtlety about this part is the decision as to how many edges to add. Since each output graph should be equally likely, the number e of total edges is picked randomly according to the binomial distribution, that is, such that

$$\text{Probability}(e) = \frac{\binom{e}{m}}{2^m}.$$

Here it is also important to rule out trivial cases. Recall that the security of the hash will rely on the fact that it is hard to find a Hamiltonian cycle in a graph. Now if the graph has too few or too many edges, it becomes rather easy to find a Hamiltonian cycle. For example, if e was picked smaller than v, no additional edges would be added and the Hamiltonian cycle would be evident immediately, making the task of finding a preimage trivial (for more explanation on preimages, see Section 5.3.2). Hence e is picked from the set $\{f, \dots, m-f\}$ where $0 \le f \le \frac{m}{2}$ can be picked according to how many cases one wants to eliminate. This "cuts off" the ends of the binomial distribution, ruling out the trivial cases.



Having said all of that, Graph can now be described as follows:

1. **Initialise $A = (a_{ij})_{i,j=1,\dots,v}$.**

$$a_{ij} = \begin{cases} 0 & \text{for } i \neq j \\ 1 & \text{for } i = j \end{cases}$$

2. **Add the edges of $p = (p_0, \dots, p_{v-1})$ to A.**

$$\begin{cases} a_{p_i p_{i+1}} = a_{p_{i+1} p_i} = 1 \text{ for } i = 0, \dots, v-2 \\ a_{p_{v-1} p_0} = a_{p_0 p_{v-1}} = 1 \end{cases}$$

3. **Pick the number of edges.** Pick $e \in \{f, \ldots, m-f\}$ randomly according to the binomial distribution.
4. **Generate uniformly random edges and add them to A until H has e edges.**

```
count = $v$;
while(count < $e$){
    generate two random numbers $r_1, r_2 \in \{1, \ldots, v\}$;
    if($a_{r_1r_2} == 0$){
        add edge $r_1r_2 : a_{r_1r_2} = a_{r_2r_1} = 1$;
        count++;
    }
}
```

5. **Output.** $\mathsf{HamHash}(M) = A_\triangle$

Note that here A is initialised with ones on the diagonal. The reason for this is that step 4 checks whether it is legal to add an edge by looking at the corresponding entry in A. Since an edge should not be added or counted when $r_1 = r_2$, the diagonal entries cannot be 0. This does not matter later on because the diagonal is not part of the output anyway.

Note also the reason for working with the entire adjacency matrix when only the lower triangle is the output. This is also done to simplify checking if an edge is already present. It is easier to check the entry $a_{r_1r_2}$ than to first work out which of r_1 and r_2 is smaller or whether they are equal to make sure we stay in the bottom triangle of A.

5.2.6 The Algorithm HamHash

Putting Red, Cyc and Graph together to form the algorithm HamHash now remains only a formality. To see how the components fit together, examine this summary:

$$\mathsf{Red} : \begin{cases} \{0,1\}^* & \rightarrow & \{0,1\}^n \\ M & \mapsto & \mathsf{Red}(M) \end{cases}$$

$$\mathsf{Cyc} : \begin{cases} \{0,1\}^n & \rightarrow & \mathsf{HamCycles}(v) \\ \mathsf{Red}(M) & \mapsto & (p_0, \ldots, p_{v-1}) \end{cases}$$

$$\mathsf{Graph} : \begin{cases} \mathsf{HamCycles}(v) & \rightarrow & \{0,1\}^m \\ (p_0, \ldots, p_{v-1}) & \mapsto & A_\triangle \end{cases}$$

Then HamHash is defined as:

$$\mathsf{HamHash} : \begin{cases} \{0,1\}^* & \rightarrow & \{0,1\}^m \\ M & \mapsto & \mathsf{Graph}(\mathsf{Cyc}(\mathsf{Red}(M))) \end{cases}$$

5.2.7 An Example

To see HamHash in action, we now consider a toy example for

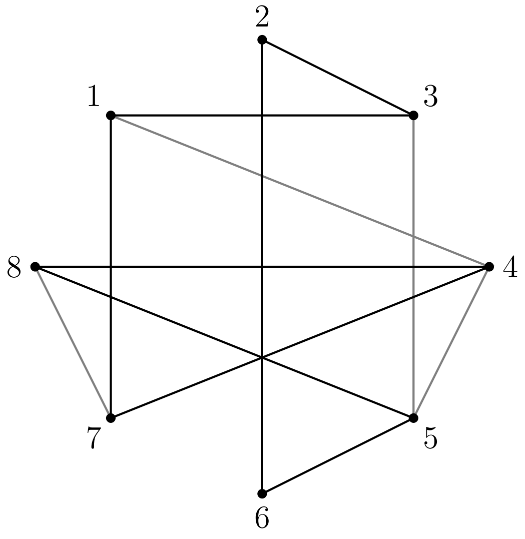
- $v = 8$,
- $m = \binom{v}{2} = \binom{8}{2} = 28$,
- $c = \frac{1}{2}(v-1)! = \frac{1}{2} \cdot 7! = 2520$, and
- $n = \lfloor \log_2(c) \rfloor = \lfloor \log_2(2520) \rfloor = 11$.

Now suppose the message is $M = 101010101010$ and for simplicity, Red is truncation to $n = 11$ bits. Then $\mathsf{Red}(M) = 10101010102$, which represents the decimal number $d = 2^{10} + 2^8 + 2^6 + 2^4 + 2^2 + 2^0 = 1365$.

Cyc then computes the 1365th cycle $\mathsf{Cyc}(10101010101) = (1, 7, 4, 8, 5, 6, 2, 3)$, which represents the cycle 174856231.

Suppose $e = 12$ is picked randomly (a reasonable choice according to the binomial distribution), and the four random edges to be added are picked (3,5), (4,5), (1,4) and (7,8).

This gives the graph



which is represented by the matrix

$$A = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{pmatrix}.$$

So

$$\mathsf{HamHash}(10101010101) = A_{\triangle} = \begin{pmatrix} 0 & & & & & & \\ 1 & 1 & & & & & \\ 1 & 0 & 0 & & & & \\ 0 & 0 & 1 & 1 & & & \\ 0 & 1 & 0 & 0 & 1 & & \\ 1 & 0 & 0 & 1 & 0 & 0 & \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

or $\mathsf{HamHash}(10101010101) = 0111000011010011001000001101$.

5.3 Properties of HamHash

More important than the functionality of HamHash are its properties. This section examines a range of different attributes of the new hash and, most importantly, presents the main theorem on its preimage resistance.

5.3.1 Non-Determinism and Verification

The possibly most exotic property of HamHash as a hash function is its *non-determinism.* Each time the hash is applied to the same message, its outcome may be different; in fact, it is very likely to be different. This means that a hash cannot be verified (as belonging to a given message) by simply recalculating it, as is usually done. To see how a non-deterministic hash can still be useful, some new definitions are required. New definitions for (second) preimage resistance and collision resistance are also needed.

Notation 5.3.1
Let HamHash*(M) denote the output of applying the algorithm* HamHash *to the message M once. There are many possible values of* HamHash*(M).*
Let HamHashes*(M) denote the set of all possible outputs of applying the algorithm* HamHash *to the message M.*

Note that $\mathsf{HamHash}(M) \in \mathsf{HamHashes}(M)$ for all $M \in \{0,1\}^*$, and also that $|\mathsf{HamHashes}(M)| \approx 2^{m-v}$ for large v. This will be justified more rigorously in Section 5.3.6.

$\mathsf{HamHashes}(M)$ and $\mathsf{HamHashes}(N)$ are not disjoint for any two messages M, N. For example, the complete graph is a valid hash of all messages (provided f is set to zero).

With the help of this notation we can now explain how to *verify* that a given hash H belongs to a certain message M. Recall that in conventional hashing, one would simply have to check that $\mathsf{HamHash}(M) = H$. This cannot be done here, since it is highly unlikely that they will be the same, even if H is a valid hash of

M. Instead, it must be checked that the Hamiltonian cycle which "belongs to" M is in the graph H. The function HamCheck achieves this.

$$\mathsf{HamCheck}(M, H) = \begin{cases} 1 & \text{(yes)} \quad \text{if } H \in \mathsf{HamHashes}(M) \\ 0 & \text{(no)} \quad \text{otherwise} \end{cases}$$

So $\mathsf{HamCheck}(M, H) = 1$ if and only if the cycle $\mathsf{Cyc}(\mathsf{Red}(M))$ is in the graph H. This can be verified quickly provided Red and Cyc are efficiently computable, since one then only has to check for ones in v entries of the adjacency matrix.

$\mathsf{HamHash}(M, H) = 1$ is equivalent to $h(M) = H$ for a conventional hash h, which now allows defining (second) preimage resistance and collision resistance in a way that looks almost identical to the conventional definitions from Section 1.2.

Definition 5.3.2 (preimage resistance)
A non-deterministic hash function HamHash *is called* **preimage resistant** *(or* **one-way***) if given a hash value H it is hard to find a message M such that*

$$\mathsf{HamCheck}(M, H) = 1.$$

Definition 5.3.3 (second preimage resistance)
A non-deterministic hash function HamHash *is called* **second preimage resistant** *if given a message M_1 it is hard to find another message M_2 such that*

$$\mathsf{HamCheck}(M_2, H) = 1 \quad \forall\, H \in \mathsf{HamHashes}(M_1).$$

Definition 5.3.4 (collision resistance)
A non-deterministic hash function HamHash *is called* **collision resistant** *if it is hard to find two messages M_1 and M_2 such that*

$$\begin{aligned} & \mathsf{HamCheck}(M_1, H_2) = 1 \quad \forall\, H_2 \in \mathsf{HamHashes}(M_2) \\ \textit{and} \quad & \mathsf{HamCheck}(M_2, H_1) = 1 \quad \forall\, H_1 \in \mathsf{HamHashes}(M_1). \end{aligned}$$

5.3.2 Preimage Resistance of HamHash

The most interesting result is that HamHash is preimage resistant, regardless of the properties of Red.

Theorem 5.3.5 (preimage resistance of HamHash)
Finding a preimage of HamHash *is at least as hard as solving* HamCycle*(v) and therefore NP-complete.*

The proof of this is a simple reduction proof. If we want to show that finding a preimage is at least as hard as solving $\mathsf{HamCycle}(v)$, that is, solving $\mathsf{HamCycle}(v)$ is at most as hard as finding a preimage, then we must show that if a preimage can be found, $\mathsf{HamCycle}(v)$ can also be solved. It must also be shown that $\mathsf{HamCycle}(v)$ can be solved in polynomial time if a preimage can be found in polynomial time.

Proof. Suppose there exists a polynomial time algorithm A which on input H produces a preimage, that is, a message M such that $\mathsf{HamCheck}(M, H) = 1$. By definition of $\mathsf{HamCheck}$ that means that $H \in \mathsf{HamHashes}(M)$ and therefore that H has the Hamiltonian cycle that corresponds to M, which is $\mathsf{Cyc}(\mathsf{Red}(M))$.

So we can define an algorithm A', which on input H outputs $\mathsf{Cyc}(\mathsf{Red}(A(H)))$ $= \mathsf{Cyc}(\mathsf{Red}(M))$. A' finds a Hamiltonian cycle in a given graph H and thus solves $\mathsf{HamCycle}(v)$. Since A is a polynomial time algorithm and Red and Cyc are polynomial time computable (otherwise $\mathsf{HamHash}$ would not be), A' is also a polynomial time algorithm.

It is clear that finding a preimage of $\mathsf{HamHash}$ is in NP since the correctness of the solution can easily be checked by computing $\mathsf{HamCheck}$(M,H). Therefore finding a preimage of $\mathsf{HamHash}$ is an NP-complete problem. □

This means that $\mathsf{HamHash}$ is a PRHF.

5.3.3 Second Preimage Resistance of $\mathsf{HamHash}$

Another important result, even though less exciting, is the fact that second preimage resistance of $\mathsf{HamHash}$ and second preimage resistance of Red are equivalent.

Theorem 5.3.6 (second preimage resistance of $\mathsf{HamHash}$)
$\mathsf{HamHash}$ *is second preimage resistant if and only if* Red *is second preimage resistant.*

Proof. Suppose $\mathsf{HamHash}$ is second preimage resistant. Then by definition given a message M_1 it is hard to find a message M_2 such that $\mathsf{HamCheck}(M_2, H) = 1$ for all $H \in \mathsf{HamHashes}(M_1)$. Now

$$\begin{aligned}
& \mathsf{HamCheck}(M_2, H) = 1 \quad \forall\, H \in \mathsf{HamHashes}(M_1) \\
\Leftrightarrow\ & H \in \mathsf{HamHashes}(M_2) \quad \forall\, H \in \mathsf{HamHashes}(M_1) \\
\Leftrightarrow\ & \mathsf{HamHashes}(M_1) = \mathsf{HamHashes}(M_2) \\
\Leftrightarrow\ & \mathsf{Cyc}(\mathsf{Red}(M_1)) = \mathsf{Cyc}(\mathsf{Red}(M_2)) \\
\Leftrightarrow\ & \mathsf{Red}(M_1) = \mathsf{Red}(M_2).
\end{aligned}$$

Hence given a message M_1 it is hard to find a message M_2 such that $\mathsf{Red}(M_1) = \mathsf{Red}(M_2)$ and Red is second preimage resistant (as a conventional deterministic hash function).

Conversely, suppose Red is second preimage resistant. Then given M_1 it is hard to find an M_2 such that $\mathsf{Red}(M_1) = \mathsf{Red}(M_2)$. As above, this is equivalent to: Given M_1 it is hard to find an M_2 such that $\mathsf{HamCheck}(M_2, H) = 1$ for all $H \in \mathsf{HamHashes}(M_1)$. So HamHash is second preimage resistant. $\square$

5.3.4 Collision Resistance of HamHash

Collision resistance behaves the same as second preimage resistance. It is equivalent in HamHash and Red.

Theorem 5.3.7 (collision resistance of HamHash)
HamHash *is collision resistant if and only if* Red *is collision resistant.*

Proof. Suppose HamHash is collision resistant. Then by definition it is hard to find two messages M_1, M_2 such that

$$\begin{array}{ll} & \mathsf{HamCheck}(M_1, H_2) = 1 \quad \forall\ H_2 \in \mathsf{HamHashes}(M_2) \\ \text{and} & \mathsf{HamCheck}(M_2, H_1) = 1 \quad \forall\ H_1 \in \mathsf{HamHashes}(M_1) \\ \Leftrightarrow & \mathsf{HamHashes}(M_1) = \mathsf{HamHashes}(M_2) \\ \Leftrightarrow & \mathsf{Cyc}(\mathsf{Red}(M_1)) = \mathsf{Cyc}(\mathsf{Red}(M_2)) \\ \Leftrightarrow & \mathsf{Red}(M_1) = \mathsf{Red}(M_2). \end{array}$$

Hence it is hard to find M_1, M_2 such that $\mathsf{Red}(M_1) = \mathsf{Red}(M_2)$ and Red is collision resistant (as a deterministic hash).

Conversely, suppose Red is collision resistant. Then it is hard to find M_1, M_2 such that $\mathsf{Red}(M_1) = \mathsf{Red}(M_2)$. As above, it is then hard to find M_1, M_2 such that $\mathsf{HamCheck}(M_1, H_2) = 1\ \forall\ H_2 \in \mathsf{HamHashes}(M_2)$ and $\mathsf{HamCheck}(M_2, H_1) = 1\ \forall\ H_1 \in \mathsf{HamHashes}(M_1)$, and so HamHash is collision resistant. $\square$

More significant than the resistance to collisions produced deliberately (by an attacker, say) might in this case be the question of collisions occurring “accidentally”. Even non-cryptographic hash functions are no good if two different inputs often hash to the same value, and the same is true for cryptographic hashes. For HamHash, it should be very unlikely that two unequal messages are verified by the same hash value. This, of course, also depends on the quality of the function chosen for Red, for any collision in the reduction function produces a collision in HamHash, as seen above. Still, the quality of Red shall not concern us here, since it depends on the choice of the function and good choices are available. Cyc can never produce collisions, since it is bijective. Hence the interesting question to ask is: Given two distinct Hamiltonian cycles p and q, how likely is it that a valid “hash” of p (meaning $\mathsf{Graph}(p)$) also verifies q? In other words, how big is the probability that $\mathsf{Graph}(p)$ also contains the cycle q?

To answer this question, some simplifying assumptions must be made (as in many of the following sections, where it will be explained why these are reasonable). Assume that an output H of Graph has approximately half edges and half non-edges (see Section 5.3.7), and that each edge is present with probability $\frac{1}{2}$ (see Section 5.3.6). Then the probability that a given cycle q is present in a graph $H = \mathsf{Graph}(p)$ is $\left(\frac{1}{2}\right)^v$, since q fixes v edges that must be present. If, for example, $v = 128$ (again a reasonable choice, see Section 5.3.9), then $\left(\frac{1}{2}\right)^v = 2^{-128} \approx 2.9 \cdot 10^{-39}$, which is reasonably small. Hence the unlikeliness of (accidental) hash collisions can be considered satisfactory.

5.3.5 Avalanche Effect in HamHash

There are more properties of hash functions that are desired but need not be proven. A significant one is the *avalanche effect*, which is the property that adjacent bit strings have completely different hashes. Recall from Section 1.2 that this is more precisely defined as: When an input to a hash function is changed slightly (e.g. one bit is flipped), the output changes significantly (i.e. approximately half the output bits flip). If a hash function does not exhibit the avalanche effect to a significant degree, a cryptanalyst may be able to make predictions about the input, given only the output, which may be sufficient to partially or completely break the algorithm. Thus, "constructing a cipher or hash to exhibit a substantial avalanche effect is one of the primary design objectives" [2].

Experiments conducted with the help of the implementation (see Section 5.5) suggest that HamHash achieves a good avalanche effect. Even when only one letter of the input changes, hash values look completely different. For good readability, we have chosen $v = 8$.

```
HamHash("abcdefg") = 1110111010100001011011011111
HamHash("abcdefh") = 0100001011110010111100111110
HamHash("abcdefi") = 1100110001111010100000100101
```

However, this requires some further theoretical investigation. The avalanche effect is trivially present for HamHash if Red achieves a good avalanche effect, but let us examine what happens without this assumption, that is, how does the output $H = \mathsf{HamHash}(M)$ change if $\mathsf{Red}(M)$ only changes slightly? The random edges produced in Graph (and those are all edges in H except for the ones resulting from the cycle) change completely every time the hash is applied, and therefore also if the hash is applied to a slightly changed $\mathsf{Red}(M)$. In fact, the presence of each edge (and therefore the value of each bit in H) changes with probability $\frac{1}{2}$, since each edge occurs with equal probability and approximately half of the edges are present. That only leaves the edges in the cycle to worry about.

How does $\mathsf{Cyc}(\mathsf{Red}(M))$ change if $\mathsf{Red}(M)$ changes by one bit? This may not always change approximately half of the edges in $\mathsf{Cyc}(\mathsf{Red}(M))$. Especially if one of the least significant bits in $\mathsf{Red}(M)$ is changed, that will only change the number $d = \mathsf{Red}(M)$ by a small power of 2, resulting in a tuple that has many of the same entries. So this may not produce a satisfactory avalanche effect for the function Cyc. Note that this only concerns some cases, though, and only the v edges resulting from the cycle, which is a very small percentage of the total number of edges $e \approx \frac{m}{2}$ for large v (e.g. $\frac{v}{e} \approx 3.15\%$ for $v = 128$). Although this should not concern us too much, it might be a good argument for choosing a reduction function which already achieves a good avalanche effect.

5.3.6 Surjectivity of HamHash

Another interesting property to examine is the *surjectivity* of HamHash. This concerns questions like "Is every graph a possible output of HamHash?" and "Are all outputs equally likely?"

The immediate response to the first question is "no". HamHash is not surjective. It is obvious that any output graph must contain a Hamiltonian cycle. Therefore all graphs which do not contain a Hamiltonian cycle, and in particular those with less than v edges, are not possible outputs of HamHash. However, it is worth examining how many such graphs exist.

It is hard to find a formula for the number of graphs on v vertices without a Hamiltonian cycle. But the use of Erdös' random graph model provides a helpful result. Bollobás [6] proves that an element of $G(v, \frac{1}{2})$ has a Hamiltonian cycle with probability tending to one as $v \to \infty$. Hence for large v, most graphs have Hamiltonian cycles, leaving very few graphs to be impossible outputs of HamHash.

Further, if the parameter f in Graph is set to zero, any graph that contains at least one Hamiltonian cycle is a possible output of HamHash, yielding not perfect but quite good surjectivity. If $f \gg 0$ is chosen, this will restrict the surjectivity by making certain outputs impossible.

It is also an important objective in hash function design that all outputs are equally likely. Otherwise an attacker could again make predictions about the output which may sometimes help in breaking the hash. Consider only the possible outputs of HamHash and examine if they all occur with equal probability. It is easy to see that this is at least roughly the case, if it is assumed that Red is a good hash function, that is, that all n-bit strings are possible and equally likely outputs of Red. When converting $\mathsf{Red}(M)$ to a cycle, not all cycles are possible, since n is chosen $n = \lfloor \log_2(c) \rfloor$. In all cases that leads to $2^n < c$, which means that the last $c - 2^n$ cycles cannot occur. Hence some of the edges resulting from the cycle might occur with greater probability than others. But again, as mentioned before, this is only a small percentage of the total edges in the graph. The random edges added in Cyc naturally all occur with equal probability, as they are chosen uniformly at random. Hence every possible output graph of HamHash has at least roughly equal probability.

5.3.7 Randomness of HamHash

It is also important that a hash function behaves like a *random* function in that it should be impossible to predict any output bits given a particular input without applying the function.

Without actually applying any statistical tests to the output of HamHash, it is probably valid to say that the output is "random-looking". The number of edges is on average about $\frac{1}{2}$, leading to approximately half zeros and half ones in the output. Each edge is equally likely, and therefore each bit is one with probability $\frac{1}{2}$ and zero with probability $\frac{1}{2}$.

Test results obtained with the help of the implementation of HamHash (see Section 5.5, $v = 8$) illustrate this property nicely.

```
HamHash("") = 1011110001110001001000011100 Ham-
Hash("0") = 0111001010101111100000110101 Ham-
Hash("maike") = 0100110100100010000110100100
HamHash("Maike") = 0010011110101011010101011010
```

It is also fairly clear that given an input it is impossible to predict any output bits without actually applying at least parts of the HamHash algorithm, **if** it is impossible to predict the random numbers. This phenomenon touches on a big problem in cryptography - how to generate *cryptographically secure* random numbers, that is, in an unpredictable way. This will be discussed in more detail in Section 5.6.3.

Note also that the application of Red and Cyc alone allow the prediction of exactly v output bits, namely the ones in the cycle corresponding to the message.

5.3.8 Efficiency of HamHash

Hash functions are usually building blocks of complex cryptographic protocols, and such protocols may require many evaluations of a hash function. They are therefore required to be efficiently computable. This is often a disadvantage of provably secure hash functions as compared to custom-designed ones. Provably secure hash functions often involve operations that are very "expensive". HamHash cannot be as efficient as the custom-designed algorithms, which are designed to be fast on 32-bit machines and only involve logical bit operations, but it is rather efficient for a provably secure hash function. Consider the three different parts that make up the algorithm.

The efficiency of Red of course depends on the choice of the function. Since this is allowed to be a custom-designed hash function, very efficient choices are available.

Cyc is in $O(v)$, so it has linear complexity, which is very efficient.

The core part is the efficiency of Graph. This depends solely on how fast random number generation is. On average, about $2(\frac{e}{2} - v) \approx e$ random numbers between 1 and v must be generated. For $v = 128$, we need about 4000 random numbers or 28000 random bits.

There are very efficient random number generators, but not all are suitable for uses in cryptography (for a discussion of this see Section 5.6.3). HamHash requires a cryptographically secure random number generator. The best such generators are hardware random number generators, which obtain entropy from physical processes. The fastest currently available physical random number generators are able to produce about 32 random Mbits per second [25]. HamHash needs less than 30 kbits, which can be produced in less than $\frac{1}{1000}$ of a second. Hence Graph is very efficient, and so all of HamHash is efficient.

5.3.9 Bits of Security of HamHash

One of the most important questions left to answer is how big v must be chosen to achieve reasonable security. To answer it, we must examine the number of Hamiltonian cycles present in an average output of HamHash.

As explained before, all graphs are approximately equally likely outputs of HamHash and therefore $G(v, \frac{1}{2})$ is a good model to use. Greenhill [22] proves that the expected number of Hamiltonian cycles in a random graph $H \in G(v, \frac{1}{2})$ is $\frac{v!}{2v}\left(\frac{1}{2}\right)^v$. Now suppose an attacker were to launch a brute force attack to find a preimage to a given hash H. Since there are 2^n possible cycles that a message M can be mapped to, and $\frac{v!}{2v}\left(\frac{1}{2}\right)^v$ of these are present in HamHash(M), the attacker would have to try $\frac{2^n}{\frac{v!}{2v}\left(\frac{1}{2}\right)^v}$ messages on average in order to find a valid preimage. Now

$$\begin{aligned}
\frac{2^n}{\frac{v!}{2v}\left(\frac{1}{2}\right)^v} &= \frac{2^n}{(v-1)!2^{-(v+1)}} = \frac{2^{n+v+1}}{(v-1)!} = \frac{2^{\lfloor \log_2(c) \rfloor} . 2^{v+1}}{(v-1)!} \\
&\leq \frac{2^{\log_2(c)} . 2^{v+1}}{(v-1)!} = \frac{c \cdot 2^{v+1}}{(v-1)!} = \frac{\frac{1}{2}(v-1)! \cdot 2^{v+1}}{(v-1)!} = 2^v
\end{aligned}$$

so v vertices give at least v bits of security, a rather nice result.

Since HamHash is not collision resistant and therefore not open to birthday attacks, at least 128 bits of security are desired (see Section 1.3.1). This gives the following values for a secure implementation of HamHash.

$$\begin{aligned}
v &= 128 \\
m &= 8128 \\
c &\approx 1.5 \cdot 10^{213} \\
n &= 708
\end{aligned}$$

This gives a hash value of approximately 1 kilobyte, which is larger than we are used to, but should not be a problem with the memory capacities available these days.

5.4 Applications of HamHash

A new type of hash function is not any good if it cannot be used for anything. Generally, preimage resistance suffices for authentication purposes. Although more research into the applications of PRHFs is necessary, this section gives at least two possible applications for which HamHash could be used.

5.4.1 Password Storage

Password storage as explained in Section 1.4.2 is a great way to use HamHash. Recall that the hash values of the passwords are stored in a password file, rather than the passwords themselves. If the file is compromised by an attacker, he/she still does not have access to the passwords since HamHash is preimage resistant. In this scenario an attacker would be interested in gaining access to an account, therefore he/she would not be interested in finding second preimages. One preimage would be enough. He/she would also not be interested in finding collisions, because he/she could not do anything with them.

Also, it does not matter that the hash is non-deterministic. It can still be verified that the stored hash belongs to the entered password, which is enough to authenticate a user.

5.4.2 Game Solution

Suppose I know the solution to some game. My friend plays the same game and she wants to check whether her solution is correct, but I do not want to give her my solution. So I give her the hash of my solution. That way, she can check her own solution, but she cannot obtain my solution from it because she cannot calculate a preimage. Again, it does not matter that the hash is non-deterministic, since the hash still verifies a correct solution. Second preimage resistance or collision resistance are not required.

5.5 Implementation of HamHash

This section presents a "proof of concept" implementation of the HamHash algorithm. This is not one that should be used in practice, it is simply to show how the algorithm works on a computer. The Java code can be found in the appendix and on the CD-ROM provided with this thesis.

The algorithm uses a truncation of SHA-1 as the reduction function, since SHA-1 is already implemented in Java. Since SHA-1 has a 160-bit output, at most 41 bits of security can be achieved. The parameters v and f may be chosen by the user.

The program `HamHash` takes two arguments, which allow the user to specify v and f (in this order). Any further arguments are ignored. If only one argument is given, f is set to zero. If no arguments are given, the defaults $v = 41$ and $f = 0$ are used.

The program provides an interactive menu and allows a user to compute and verify hashes. Hash values are shown as a lower triangular matrix and also concatenated as a string. The menu is self explanatory and the use of the program should be fully understood by looking at the following sample execution.

```
>HamHash 20
Welcome!

You have chosen 20 bits of security.
v = 20
m = 190
n = 55
f = 0

Which task would you like to perform?  Type
0 to exit
1 to compute a hash
2 to verify a hash
1
Enter a message:
One doesn't discover new lands without consenting to lose sight
of the shore for a very long time.  André Gide

The hash is:
1
00
111
1001
01111
001011
1101110
01001111
110011010
0001001010
11001110011
101000000000
1001000100001
01100110010111
010101100010100
1100000010010001
00010001000001010
```

```
011010110111100011
1011101001101101000

The hash string is:
10011110010111100101111011100100111111001101000010010101100111001
1101000000000100100010000101100110010111010101100010100110000001 0
0100010001000100000101001101011011110001110111010011 01101000

Which task would you like to perform?  Type
0 to exit
1 to compute a hash
2 to verify a hash
2
Enter a message:
One doesn’t discover new lands without consenting to lose sight
of the shore for a very long time.  André Gide
Enter a hash string:
10011110010111100101111011100100111111001101000010010101100111001
11010000000001001000100001011001100101110101011000101001100000010
010001000100010000010100110101101111000111011101001101101000
VALID HASH

Which task would you like to perform?  Type
0 to exit
1 to compute a hash
2 to verify a hash
2
Enter a message:
One doesn’t discover new lands without consenting to lose sight
of the shore for a very long time.  André Gide
Enter a hash string:
00011110010111100101111011100100111111001101000010010101100111001
11010000000001001000100001011001100101110101011000101001100000010
010001000100010000010100110101101111000111011101001101101000
VALID HASH

Which task would you like to perform?  Type
0 to exit
1 to compute a hash
2 to verify a hash
```

```
2
Enter a message:
One doesn't discover new lands without consenting to lose sight
of the shore for a very long time.  André Gide
Enter a hash string:
00000000010111100101111011100100111111001101000010010101100111001
11010000000001001000100001011001100101110101011000101001100000010
0100010001000100000101001101011011110001110111010011011010000
INVALID HASH

Which task would you like to perform?  Type
0 to exit
1 to compute a hash
2 to verify a hash
0

Good bye!
```

This implementation has not been optimised to achieve speed. Rather, it has been made to have a very similar structure to the description of the algorithm in this chapter.

Also note that the implementation uses the standard method of random number generation implemented in Java. These random numbers are not really suitable for cryptographic purposes.

At this point, we should also comment on the choice of f. Although the implementation allows a user to set f, we have found that it is not really necessary. The binomial distribution has such a small probability on both ends that very small or very large values for e do not seem to occur in practice. It really depends on the level of concern of the person running the program.

5.6 Problems and Further Research

As with any new design, there are still unsolved problems in HamHash that require further research. This thesis also gives the direction for several new areas of research which are not directly related to current problems in HamHash but rather extensions of this work. Many of them I would be very interested to explore myself.

5.6.1 Digest Size

The large output size is what many might criticise first about HamHash. We have shown that $v = 128$, which will achieve the security necessary today, produces a hash value of about 1 kilobyte, which is much larger than that of the hash functions we are using (e.g. MD5: 128 bits, SHA-1: 160 bits). However, many provably secure hash functions have rather large hash values (recall that the hash of SubSum is at least 1352 bits) and this is becoming less of a problem as memory becomes cheaper and more available.

Often large hash values are simply truncated. Note that HamHash does not allow this, as it would make it impossible to verify a hash. The entire Hamiltonian cycle has to be present in the output graph.

5.6.2 Reduction Function

We suggested the use of a custom-designed hash function as the reduction function Red. However it was shown that the output of this function must have at least $n = 708$ bits for $v = 128$. As of today, no custom-designed hash function that produces such a big output exists. To our knowledge, the largest are SHA-512 and WHIRLPOOL with a digest size of 512 bits. There are rumours of SHA-1024 but it does not seem to actually exist. Still, it should be possible to further extend the SHA-family to produce an output larger than 708 bits (probably 1024 bits), as it has been extended before from 160 to 256 to 512 bits. This would definitely be worth some further investigation.

5.6.3 Cryptographically Secure Random Number Generation

Graph requires the generation of random numbers, and this is anything but a trivial process. While a proper discussion of random number generation would fill books and clearly be beyond the scope of this document, we just briefly outline the problem and make a few comments.

Random numbers can be generated in two different ways: either by computational methods (*pseudo-random number generators*) or from physical processes (*hardware random number generators*). Sometimes these methods are combined to form hybrid models. Sequences of random numbers must always pass certain statistical tests which indicate that they are indistinguishable from 'true' random numbers.

However, good statistical properties are not enough for the purposes of HamHash. The random numbers must not only be "random-looking", they must also withstand serious attack. For example, if an attacker could predict which random edges were added, they could easily subtract them from an output graph to obtain the Hamiltonian cycle. Pseudo-random generators that fulfill such properties are called *cryptographically secure pseudo-random number generators* and must more specifically satisfy the following two requirements [12].

- Firstly, they must satisfy the "next bit test": Given the first k bits of a sequence, there is no polynomial-time algorithm that can predict the $(k+1)^{\text{th}}$ bit with probability bigger than $\frac{1}{2}$.
- Secondly, they must withstand "state compromise extensions": Even if part or all of its state has been compromised, it must be impossible to reconstruct the stream of random numbers prior to the revelation.

Such a cryptographically secure pseudo-random number generator would suffice for use in HamHash, but finding one is a very difficult problem. Similar to the situation in hash functions design, constructions exist which are assumed to have these properties, but it is hard to prove them.

The preferred method would be a physical random number generator. “There is general agreement that, if there are such things as ‘true’ random numbers, they are most likely to be found by looking at physical processes which are, as far as we know, unpredictable” [51]. Many cryptographically secure pseudo-random number generators use entropy obtained from a physical source (e.g. mouse movement or keyboard input). However, this method also has its problems. It is usually slower than purely computational pseudo-random number generators, and unexpected correlations have been found in several supposedly independent physical processes.

Several pseudo-random number generators have been standardised for cryptographic purposes [50]. The Blum Blum Shub Algorithm [29] is also a very promising design. It has an unusually strong security proof, which relates the quality of the generator to to computational difficulty of integer factorisation. Hardware random number generators that are efficient and claim to be cryptographically secure are available [25]. How good they really are remains an open question. It only remains to stress that cryptographically secure random number generation is a huge problem and that the method used in HamHash must be selected carefully.

5.6.4 Hardness of HamCycle(v)

The security of HamHash relies fully on the hardness of the Hamiltonian cycle problem. However, as with any NP-complete problem, HamCycle(v) may not be hard in all cases, since NP-completeness only proves the hardness of the general case, and not of every instance of a problem. In fact, it can easily be seen that finding a Hamiltonian cycle is easy if there are very few or very many edges in a graph. Manber [38] gives an efficient algorithm for finding a Hamiltonian cycle in a graph where all vertices have degree $\geq \frac{v}{2}$. Liu [37] proves that the problem is solvable in polynomial time for instances where the graph has no vertex exceeding degree 2 or is a line graph. Remember that these first two easy cases (i.e. too few and too many vertices) may be ruled out by setting the parameter f in Graph appropriately (by the pigeonhole-principle, there must be at least one vertex with degree greater than 2 if $f > v$, and there must be at least one vertex with degree less than $\frac{v}{2}$ if $m - f < \frac{v^2}{4}$), and these results should be considered when selecting f. In addition, the results stated in Section 5.1.1 also show that HamCycle(v) remains NP-complete in many special cases. Thus one can be very confident that the problem is hard in most cases, which may be considered satisfactory for many cryptographic applications. However, no exact results on the proportion of polynomial-time solvable cases seem to be available at this time.

5.6.5 Approximation Algorithms

Ways to deal with NP-complete problems, and especially approximation algorithms, are a large area of research in the theory of NP-completeness. For the use of NP-complete problems in cryptography, inapproximability results are even more interesting. This opens up a new area of research (see [1], for example), in which much work is yet to be done.

An approximation algorithm for the Hamiltonian cycle problem would attempt to approximate a Hamiltonian cycle by finding a large cycle in a graph. However, no approximation algorithm seems to exist for the Hamiltonian cycle problem, and even if it did, it would not defeat the security of HamHash. A large cycle is not sufficient to produce a preimage. A full Hamiltonian cycle has to be found to defeat preimage resistance. Zuckerman's proof [69] that the Hamiltonian cycle problem is "absurdly hard to approximate" (more concretely, that there is no polynomial-time algorithm that can achieve a usable approximation of HamCycle) gives even more certainty that approximation algorithms cannot defeat the security of HamHash.

There are good approximation algorithms for the Hamiltonian path problem, but again they do not help in breaking HamHash, because a true Hamiltonian cycle is needed.

5.6.6 Hash Functions Based on NP-Complete Graph Problems

An interesting area for further research might be the design of more hash functions based on NP-complete graph problems such as *vertex cover, dominating set, independent set, monochromatic triangle, clique*, or any of the many *partition, colorability* or *subgraph problems*. Some ideas from this work could prove useful in that.

5.6.7 Applications of Non-Deterministic Hashes and PRHF

Another area where much work is left to be done is determining in which applications the new designs could be used. Although it has been explained that non-deterministic PRHFs are sufficient for authentication purposes, it is worth investigating where else they might be used. This can be done separately for non-deterministic hash functions and PRHFs.

5.6.8 Stronger Non-Deterministic Hash Functions

If it turns out that non-deterministic hash functions have applications, or that they even have advantages over deterministic hash functions, research into non-deterministic hashes with stronger security properties might be worthwhile. For example, are there non-deterministic hash functions which are second preimage resistant, or maybe even collision resistant?

5.6.9 Another Definition of "Hard"

There is another fairly common definition of hardness, which is often referred to as *computational infeasibility.*

Definition 5.6.1 (computationally infeasible)
A problem is considered **computationally infeasible** *if there exists no polynomial time algorithm A that can solve the problem with non-negligible probability.*

Here a probability p is considered non-negligible if $p > \frac{1}{q(n)}$ where q is a polynomial and n is the size of the input to the algorithm A.

This definition of hardness is even better than NP-completeness because it bounds the proportion of polynomial-time solvable problems (for a fixed input size). Unfortunately, it is also much harder to handle than the well studied class of NP-complete problems. A very interesting question is whether HamHash or other hash functions are still secure under the more strict requirement of computational infeasibility for hard operations. Studying the general connections between computationally infeasible problems and NP-complete problems might also be worthwhile.

5.7 Conclusion

This chapter has introduced a new type of hash function and a concrete design of its kind. It has shown that HamHash as a non-deterministic hash function is provably preimage resistant and therefore a PRHF. However, it is not second preimage resistant or collision resistant unless we have a reduction function with these properties, which would make HamHash pointless. Still, its security is sufficient for some applications. It is an interesting design and if more research is done to overcome its problems, it may very well be of practical importance, forming the first design of its kind.

Conclusion

Cryptographic hash functions are an interesting and challenging area of research. Much progress has been made, and much work is yet to be done. Provably secure hash functions are highly desirable, but their design is very difficult.

The most practical and most widely used method in current research is relating security properties to NP-complete problems. This is an established method of quantifying “hardness”, and the task of proving security properties boils down to writing reduction proofs, which is often achievable. The large number of known NP-complete problems gives a great variety to choose from and leads to many different possibilities of designing provably secure hash functions. Many can be turned into hash functions in very natural ways.

Still, there are many problems with this approach, and for every new design, these have to be investigated and solved individually. Provably secure hash functions are usually much slower than custom-designed ones, since NP-complete problems often involve operations that are very slow on a computer, such as exponentiation or multiplication. Although NP-completeness gives an indication of how hard it is to solve a problem, there is always the possibility of approximation. For some problems, polynomial-time approximation algorithms may exist that approximate the problem well enough so that they can be used for successful attacks. It would be desirable to prove that such algorithms do not exist (this is the aim of inapproximability theory), but that is again a very difficult task. And even if no approximation algorithm exists, NP-completeness gives no indication of how many trivial (or efficiently solvable) instances a problem has. Other definitions of “hard” that take this problem into account (i.e. that limit the proportion of easy problem instances) exist but are much harder to work with.

Despite some still existing difficulties, this seems to be the way to go. It is time to stop relying on hash functions with questionable security for applications as important as digital signatures and message authentication codes. The last two years of cryptanalysis have clearly shown that current hash algorithms do not provide the security we desire. I think anyone who fully understands the implications of these developments should be willing to accept small disadvantages like longer running time to gain provable security, and I hope that provably secure hash functions will soon be ready to replace custom-designed ones.

Appendix

```
/////////////////////////////////////////////////////////////////////////////////
// Maike Massierer 12/12/2006                                                  //
// class HamHash - implementation of hashing algorithm HamHash                 //
/////////////////////////////////////////////////////////////////////////////////

import java.io.*;
import java.security.*;
import java.math.BigInteger;
import java.lang.Integer;
import java.util.ArrayList;
import java.util.Random;
import cern.jet.random.Binomial;
import cern.jet.random.engine.MersenneTwister;

public class HamHash{

    /////////////////////
    // class variables //
    /////////////////////

    private static int v;           //number of vertices
    private static int m;           //number of possible edges
    private static int n;           //length of Red output
    private static int f;           //security parameter
    private static int[][] t;       //lookup table

    //lookup table for n in {0,...,41}
    private static int[] tn = {0, 0, 0, 0, 1, 3, 5, 8, 11, 14, 17, 20, 24, 27,
                               31, 35, 39, 43, 47, 51, 55, 60, 64, 68, 73, 78,
                               82, 87, 92, 96, 101, 106, 111, 116, 121, 126,
                               131, 137, 142, 147, 152, 158};

    //lookup table to convert a nibble to a hex char
    static char[] hexChar = {'0', '1', '2', '3',
                             '4', '5', '6', '7',
                             '8', '9', 'a', 'b',
                             'c', 'd', 'e', 'f'};

    ////////////////////
    // public methods //
    ////////////////////

    //computeHamHash - computes hash value of msg
    public static int[][] computeHamHash(String msg){
        return Graph(Cyc(Red(msg)));
    }

    //hamCheck - checks if hash is a valid hash value of msg
    //0 = invalid, 1 = valid
    private static int hamCheck(String msg, String hash){
        //invalid if string has wrong size
        if(hash.length() != m){
            return 0;
        }
        //invalid if not all characters are 0 or 1
        for(int i = 0; i < hash.length(); i++){
            if(hash.charAt(i) != '0' && hash.charAt(i) != '1'){
                return 0;
            }
        }

        //turn hash into lower triangular matrix A
        int[][] A = new int[v][v];
        int k = 0;
        for(int i = 0; i < v; i++){
            for(int j = 0; j < i; j++){
                A[i][j] = Integer.parseInt(hash.substring(k, k+1));
                k++;
            }
        }

        //compute cycle corresponding to mgs
        int[] p = Cyc(Red(msg));

        //check if all edges belonging to the cycle are present in A
        int result = 1;
        for(int i = 0; i < v-1; i++){
            if(p[i] > p[i+1]){
                if(A[p[i]-1][p[i+1]-1] != 1){
                    result = 0;
                }
            } else{
                if(A[p[i+1]-1][p[i]-1] != 1){
                    result = 0;
                }
            }
        }

        if(p[v-1] > p[0]){
            if(A[p[v-1]-1][p[0]-1] != 1) result = 0;
        } else{
            if(A[p[0]-1][p[v-1]-1] != 1) result = 0;
        }

        return result;
    }

    /////////////////////
    // private methods //
    /////////////////////

    //Red - computes SHA-1 digest of text, truncated to length n
    private static String Red(String text){
        byte[] msgDigest = new byte[1];

        try{
            MessageDigest md = MessageDigest.getInstance("SHA");
            msgDigest = md.digest(text.getBytes());
        } catch (NoSuchAlgorithmException nsae) {
            System.out.println("SHA-1 not available. Good bye!");
            System.exit(1);
        }

        //return bit string of length n
        return toBinaryString(msgDigest).substring(0, n);
```

```
}

//Cyc - computes cycle corresponding to binary string bin
private static int[] Cyc(String bin){
    int[] p = new int[v];        //cycle
    p[0] = 1;

    BigInteger d = new BigInteger(bin, 2);

    p[1] = t[(d.divide(factorial(v-3))).intValue()][0];
    p[v-1] = t[(d.divide(factorial(v-3))).intValue()][1];

    d = d.mod(factorial(v-3));

    //list a = (2,...,v) without p_1 and p_{v-1}
    ArrayList a = new ArrayList();
    for(int i = 2; i <= v; i++){
        if(i != p[1] && i != p[v-1]){
            a.add(new Integer(i));
        }
    }

    for(int i = v-4; i >= 0; i--){
        p[v-i-2] = Integer.parseInt((a.remove
                ((d.divide(factorial(i))).intValue())).toString());
        d = d.mod(factorial(i));
    }

    return p;
}

//Graph - computes graph with cycle p and more random edges
private static int[][] Graph(int[] p){
    //initialise A with 1s on diagonal, 0s everywhere else
    int[][] A = new int[v][v];
    for(int i = 0; i < v; i++){
        A[i][i] = 1;
    }

    //add edges of cycle to A
    for(int i=0; i < v-1; i++){
        A[p[i]-1][p[i+1]-1] = 1;
        A[p[i+1]-1][p[i]-1] = 1;
    }
    A[p[v-1]-1][p[0]-1] = 1;
    A[p[0]-1][p[v-1]-1] = 1;

    //pick random number e in {f,...,m-f} of edges
    //according to binomial distribution
    int e = -1;
    MersenneTwister twister = new MersenneTwister(new java.util.Date());
    Binomial binGenerator = new Binomial(m, 0.5, twister);
    while(e < f || e > m-f){
        e = binGenerator.nextInt();
    }

    //generate random edges and add them to A
    int r1 = 0;
    int r2 = 0;
    int count = v;
    Random generator = new Random();
    while(count < e){
            //generate two random numbers between 0 and v-1
            r1 = generator.nextInt(v);
            r2 = generator.nextInt(v);

            //if the edge r1r2 is not there yet, add it
            if(A[r1][r2] == 0){
                A[r1][r2] = 1;
                A[r2][r1] = 1;
                count++;
            }
        }

        return A;
    }

    //setConstants - sets v, m, n, f, t
    //with vv in {6,...,41} number of vertices (= bits of security)
    private static void setConstants(int vv, int ff){
        v = vv;                         //number of vertices
        m = v*(v-1)/2;                  //number of possible edges
        n = tn[v];                      //length of Red output
        f = ff;                         //security parameter

        t = new int[(v-1)*(v-2)/2][2];  //lookup table
        int k = 0;
        for(int i = 3; i < v; i++){
            for(int j = 2; j < i; j++){
                t[k][0] = i;
                t[k][1] = j;
                k++;
            }
        }

        //output
        System.out.println("\nYou have chosen " + v + " bits of security.");
        System.out.println("v = " + v);
        System.out.println("m = " + m);
        System.out.println("n = " + n);
        System.out.println("f = " + f);
    }

    //menu
    private static int menu(){
        int choice = 0;

        System.out.println("\nWhich task would you like to perform? Type");
        System.out.println("0 to exit");
        System.out.println("1 to compute a hash");
        System.out.println("2 to verify a hash");

        //open up standard input
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        //read user input
        try {
            choice = Integer.parseInt(br.readLine());
        } catch (IOException ioe) {
                System.out.println("IO error trying to read. Good bye!");
                System.exit(1);
        } catch (NumberFormatException nfe) {
                System.out.println("Must enter a number. Try again!");
                return 1;
        }

        switch(choice){
        case 1:
            computeHash();
            break;
        case 2:
            verifyHash();
            break;
        }

        return choice;
    }

    //computeHash - user interface for computing a hash
    private static void computeHash(){
        System.out.println("Enter a message:");
        String msg = "";

        //open up standard input
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        //read user input
        try {
            msg = br.readLine();
        } catch (IOException ioe) {
                System.out.println("IO error trying to read. Good bye!");
                System.exit(1);
        }

        //compute hash
        int[][] A = computeHamHash(msg);

        //output
        printHash(A);
        printHashString(A);
    }

    //verify Hash - user interface for verifying a hash
    private static void verifyHash(){
        System.out.println("Enter a message:");
        String msg = "";

        //open up standard input
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        //read user input
        try {
            msg = br.readLine();
        } catch (IOException ioe) {
                System.out.println("IO error trying to read. Good bye!");
                System.exit(1);
        }

        System.out.println("Enter a hash string:");
        String hash = "";

        //read user input
        try {
            hash = br.readLine();
        } catch (IOException ioe) {
                System.out.println("IO error trying to read. Good bye!");
                System.exit(1);
        }

        //hamCheck
        int result = hamCheck(msg, hash);

        //output
        if(result == 1){
            System.out.println("VALID HASH");
        } else{
            System.out.println("INVALID HASH");
        }
    }

    //printHash - prints lower triangle of matrix A
    private static void printHash(int[][] A){
        System.out.print("\nThe hash is:");
        for(int i = 0; i < v; i++){
            for(int j = 0; j < i; j++){
                System.out.print(A[i][j]);
            }
            System.out.println();
        }
    }

    //printHashString - prints lower triangle of matrix A as string
    private static void printHashString(int[][] A){
        System.out.println("\nThe hash string is:");
        for(int i = 0; i < v; i++){
            for(int j = 0; j < i; j++){
                System.out.print(A[i][j]);
            }
        }
        System.out.print("\n");
    }

    //factorial - calculates BigInteger factorial
    private static BigInteger factorial(int n){
        if(n == 0){
            return new BigInteger("1");
        }

        BigInteger bn = new BigInteger((new Integer(n)).toString());

        for(int i = 1; i < n; i++){
            bn = bn.multiply(new BigInteger((new Integer(n-i)).toString()));
        }

        return bn;
    }

    //toHexString - converts a byte array to a hex string
    //with possible leading zero
    private static String toHexString(byte[] b){
        StringBuffer sb = new StringBuffer(b.length * 2);
        for (int i = 0; i < b.length; i++){
            //look up high nibble char
            sb.append(hexChar[(b[i] & 0xf0) >>> 4]);

            //look up low nibble char
            sb.append(hexChar[b[i] & 0x0f]);
            }
        return sb.toString();
    }

    //toBinaryString - converts a byte array to a binary string
    private static String toBinaryString(byte[] b){
        String hex = toHexString(b);
        String bin = "";
        for(int i = 0; i < hex.length(); i++){
            bin += hexDigitToBinaryString(hex.charAt(i));
        }

        return bin;
    }

    //hexDigitToBinaryString - converts a hex digit to a binary string
    private static String hexDigitToBinaryString(char d){
        switch(d){
        case '0': return "0000";
        case '1': return "0001";
        case '2': return "0010";
        case '3': return "0011";
        case '4': return "0100";
        case '5': return "0101";
        case '6': return "0110";
        case '7': return "0111";
        case '8': return "1000";
        case '9': return "1001";
        case 'a': return "1010";
        case 'b': return "1011";
        case 'c': return "1100";
        case 'd': return "1101";
        case 'e': return "1110";
        case 'f': return "1111";
        default: return "error";
        }
    }

    //////////
    // main //
    //////////
    public static void main(String[] args){
        System.out.println("Welcome!");

        //setup
        if(args.length == 0){                //no args: v = 41, f = 0
            setConstants(41, 0);
        } else if (args.length == 1){        //1 arg sets v, f = 0
            int vv = Integer.parseInt(args[0]);

            if(vv < 6 || vv > 41){           //invalid choice for v
                System.out.println("v must be between 6 and 41. Good bye!");
                System.exit(1);
            }

            setConstants(vv, 0);
        } else{                              //2 args set v and f
            int vv = Integer.parseInt(args[0]);
            int ff = Integer.parseInt(args[1]);

            if(vv < 6 || vv > 41){           //invalid choice for v
                System.out.println("v must be between 6 and 41. Good bye!");
                System.exit(1);
            }
            if(ff < 0 || ff > vv*(vv-1)/4){ //invalid choice for f
                System.out.println("f must be less than or
                                    equal to v(v-1)/4. Good bye!");
                System.exit(1);
            }

            setConstants(vv, ff);
        }

        //menu
        int done = 1;
        while(done != 0){
            done = menu();
        }

        System.out.println("\nGood bye!");
    }
}
```

References

[1] S Arora. The Approximability of NP-hard Problems. *Proceedings of the thirtieth annual ACM symposium on Theory of computing,* pp 337-348. ACM Press, New York, 1998.

[2] Avalanche Effect. *Wikipedia,* 2006. Viewed 05/12/2006, `http://en.wikipedia.org/wiki/Avalanche_effect.`

[3] P Barreto. The Hashing Function Lounge, 2006. Viewed 25/09/2006, `http://paginas.terra.com.br/informatica/paulobarreto/hflounge.html.`

[4] M Bellare and P Rogaway. Collision-Resistant Hashing: Towards Making UOWHFs Practical. Advances in Cryptology - CRYPTO '97. *Lecture Notes in Computer Science* 1294, pp 470-484. Springer-Verlag, Berlin, 1997.

[5] B den Boer and A Bosselaers. Collisions for the Compression Function of MD5. Advances in Cryptology - EUROCRYPT '93. *Lecture Notes in Computer Science* 765, pp 293-304. Springer-Verlag, Berlin, 1994.

[6] B Bollobas. *Random Graphs*, 2nd edn. Cambridge University Press, Cambridge, 2001.

[7] H J Broersma, L Xiong, and K Yoshimoto. Toughness and Hamiltonicity in k-trees. Memorandum No. 1576, Faculty of Mathematical Sciences, University of Twente, 2001.

[8] F Chabaud and A Joux. Differential Collisions in SHA-0. Advances in Cryptology - CRYPTO '98. *Lecture Notes in Computer Science* 1462, pp 56. Springer-Verlag, Berlin, 1998.

[9] D X Charles, E Z Goren, and K E Lauter. Cryptographic Hash Functions from Expander Graphs. *Cryptology ePrint Archive,* Report 2006/021, 2006. Viewed 13/12/2006, `http://eprint.iacr.org/2006/021.pdf.`

[10] S A Cook. The Complexity of Theorem Proving Procedures. *Third Annual ACM Symposium on Theory of Computing,* pp 151-158. ACM, 1971.

[11] T H Cormen, C E Leiserson, and R L Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, 1996.

[12] Cryptographically Secure Pseudorandom Number Generator. *Wikipedia,* 2006. Viewed 06/12/2006, `http://en.wikipedia.org/wiki/CSPRNG.`

[13] I B Damgård. Collision Free Hash Functions and Public Key Signature Schemes. Advances in Cryptology - EUROCRYPT '87. *Lecture Notes in Computer Science* 304, pp 203-216. Springer-Verlag, Berlin, 1988.

[14] I B Damgård. A Design Principle for Hash Functions. Advances in Cryptology - CRYPTO '89. *Lecture Notes in Computer Science* 435, pp 428-446. Springer-Verlag, Berlin, 1989.

[15] S Diffie and M E Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory* 22(6), pp 644-654. 1976.

[16] H Dobbertin. Cryptanalysis of MD5 Compress. Announcement on the Internet, 1996.

[17] D Eastlake and P Jones. US Secure Hash Algorithm 1 (SHA1), Request for Comments (RFC) 3174, 2001. Viewed 11/12/2006,
`http://www.ietf.org/rfc/rfc3174.txt.`

[18] P Erdös and A Rényi. On Random Graphs. *Publicationes Mathematicae Debrecen* 6, pp 290-297. Institute of Mathematics, University of Debrecen, Hungary, 1959.

[19] M R Garey and D S Johnson. The Complexity of Near-Optimal Graph Coloring. *Journal of the ACM* 23(1), pp 43-49. ACM Press, New York, 1976.

[20] M R Garey and D S Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 2000.

[21] P Gauravaram. Cryptographic Hash Functions. Talk at ICE-EM workshop on stream ciphers and hash functions, 2006.

[22] C Greenhill. Graph Theory Lecture Notes. Lecture at UNSW, Session 1, 2006.

[23] Z G Gutin and P Moscato. The Hamiltonian Page, 2000. Viewed 09/12/2006,
`http://www.densis.fee.unicamp.br/~moscato/Hamilton.html.`

[24] Hash Function. *Wikipedia,* 2006. Viewed 08/12/2006,
`http://en.wikipedia.org/wiki/Hash_function.`

[25] HG400 Random Number Generators. random.com.hr, 2005. Viewed 09/12/2006,
`http://random.com.hr/products/hg400/index.html.`

[26] D Hong, B Preneel, and S Lee. Higher Order Universal One-Way Hash Functions. Advances in Cryptology - ASIACRYPT '04. *Lecture Notes in Computer Science* 3329, pp 201-213. Springer-Verlag, Berlin, 2004.

[27] Icosian Game. Wolfram Math World, 2003. Viewed 30/11/2006,
`http://mathworld.wolfram.com/IcosianGame.html.`

[28] R Impagliazzo and M Naor. Efficient Cryptographic Schemes Provably as Secure as Subset Sum. *Journal of Cryptology* 9, pp 199-216. Springer-Verlag, Berlin, 1996.

[29] P Junod. Cryptographic Secure Pseudo-Random Bits Generation: The Blum-Blum-Shub Generator. 1999. Viewed 11/12/2006,
`http://crypto.junod.info/bbs.pdf.`

[30] R M Karp. Reducibility Among Combinatorial Problems. *Complexity of Computer Computations,* pp 85-103. Plenum Press, New York, 1972.

[31] V Klima. Finding MD5 Collisions - a Toy for a Notebook. *Cryptology ePrint Archive,* Report 2005/075, 2005. Viewed 06/12/2006, `http://eprint.iacr.org/2005/075.`

[32] V Klima. Tunnels in Hash Functions: MD5 Collisions Within a Minute. *Cryptology ePrint Archive,* Report 2006/105, 2006. Viewed 06/12/2006, `http://eprint.iacr.org.`

[33] D E Knuth. *The Art of Computer Programming, Vol. 1 Fundamental Algorithms,* 2nd edn. Addison-Wesley Publishing Company, Reading, 1973.

[34] M S Krishnamoorthy. An NP-hard Problem in Bipartite Graphs. *ACM SIGACT News* 7(1), pp 26. ACM Press, New York, 1975.

[35] W Lee, D Chang, S Lee, S Sung, and M Nandi. New Parallel Domain Extenders for UOWHF. Advances in Cryptology - ASIACRYPT '03. *Lecture Notes in Computer Science* 2894, pp 208-227. Springer-Verlag, Berlin, 2003.

[36] A Lenstra, X Wang, and B de Weger. Colliding X.509 Certificates. *Cryptology ePrint Archive,* Report 2005/067, 2005. Viewed 06/12/2006, `http://eprint.iacr.org/2005/067.`

[37] C L Liu. *Introduction to Combinatorial Mathematics.* McGraw-Hill, New York, 1968.

[38] U Manber. *Introduction to Algorithms.* Addison-Wesley, Reading, 1989.

[39] MD5. *Wikipedia,* 2006. Viewed 15/12/2006, `http://en.wikipedia.org/wiki/Md5.`

[40] MD5CRK. *Wikipedia,* 2006. Viewed 15/12/2006, `http://en.wikipedia.org/wiki/MD5CRK.`

[41] R Merkle. One Way Hash Functions and DES. Advances in Cryptology - CRYPTO '89. *Lecture Notes in Computer Science* 435, pp 428-446. Springer-Verlag, Berlin, 1990.

[42] R Merkle and M Hellman. Hiding Information and Signature in Trapdoor Knapsack. *IEEE Transaction on Information Theory* 24(5), pp 525-530. 1978.

[43] M Naor and M Yung. Universal One-Way Hash Functions and their Cryptographic Applications. *Proceedings of the Twenty-first ACM Symposium on Theory of Computing,* pp 33-43. ACM Press, New York, 1989.

[44] NIST Brief Comments on Recent Cryptanalytic Attacks on Secure Hashing Functions and the Continued Security Provided by SHA-1. NIST, 2004. Viewed 06/12/2006, `http://csrc.nist.gov/news-highlights/NIST-Brief-Comments-on-SHA1-attack.pdf.`

[45] C H Papadimitriou and K Steiglitz. Some Complexity Results for the Traveling Salesman Problem. *Proceedings of the 8th Annual ACM Symposium on Theory of Computing,* pp 1-9. Association for Computing Machinery, New York, 1976.

[46] J Pieprzyk. Hash Functions: Provable Security versus Custom Design. Talk at ICE-EM workshop on stream ciphers and hash functions, 2006.

[47] B Preneel. Analysis and Design of Cryptographic Hash Functions. PhD Thesis, Katholieke Universiteit Leuven, 1993.

[48] B Preneel. The State of Cryptographic Hash Functions. Lectures on Data Security. *Lecture Notes in Computer Science* 1561, pp 158-182. Springer-Verlag, Berlin, 1999.

[49] J J Quisquater. Cryptology and Graph Theory. Presentation, UCL Crypto group, a member of EIDMA, 2005. Viewed 13/12/2006, `http://www.win.tue.nl/diamant/symposium05/abstracts/quisquater.pdf.`

[50] Random Number Generation. Computer Security Resource Center, Cryptographic Toolkit, NIST, 2006. Viewed 05/12/2006, `http://csrc.nist.gov/CryptoToolkit/tkrng.html.`

[51] Random Number Generator. *Wikipedia,* 2006. Viewed 09/12/2006, `http://en.wikipedia.org/wiki/Random_number_generator.`

[52] B Rijmen and E Oswald. Update on SHA-1. *Cryptology ePrint Archive,* Report 2005/010, 2005. Viewed 06/12/2006, `http://eprint.iacr.org/2005/010.`

[53] R L Rivest. The MD5 Message-Digest Algorithm, Request for Comments (RFC) 1321, 1992. Viewed 11/12/2006, `http://www.ietf.org/rfc/rfc1321.txt.`

[54] P Sarkar. Construction of UOWHF: Tree Hashing Revisited. *Cryptology ePrint Archive,* Report 2002/058, 2002. Viewed 11/12/2006, `http://eprint.iacr.org/2002/058.`

[55] P Sarkar. Domain Extenders for UOWHF: A Generic Lower Bound on Key Expansion and a Finite Binary Tree Algorithm. *Cryptology ePrint Archive,* Report 2003/009, 2004. Viewed 11/12/2006, `http://eprint.iacr.org/2003/009.`

[56] B Schneier. *Applied Cryptography; Protocols, Algorithms, and Source Code in C,* 2nd edn. John Wiley & Sons, Inc., New York, 1996.

[57] B Schneier. New Cryptanalytic Results Against SHA-1. Schneier on Security, 2005. Viewed 07/12/2006, `http://www.schneier.com/blog/archives/2005/08/new_cryptanalyt.html.`

[58] B Schneier. SHA-1 Broken. Schneier on Security, 2005. Viewed 07/12/2006,
`http://www.schneier.com/blog/archives/2005/02/sha1_broken.html.`

[59] Secure Hashing. Computer Security Resource Center, Cryptographic Toolkit, NIST, 2006. Viewed 07/12/2006,
`http://csrc.nist.gov/CryptoToolkit/tkhash.html.`

[60] SHA Hash Functions. *Wikipedia,* 2006. Viewed 09/12/2006,
`http://en.wikipedia.org/wiki/Sha1.`

[61] SHA-1 Hash Function under Pressure. Heise Security, 2006. Viewed 07/12/2006,
`http://www.heise-security.co.uk/news/77244.`

[62] V Shoup. A Composition Theorem for Universal One-Way Hash Functions. Advances in Cryptology - EUROCRYPT '00. *Lecture Notes in Computer Science* 1807, pp 445-452. Springer-Verlag, Berlin, 2000.

[63] S S Skiena. *The Algorithm Design Manual.* Springer-Verlag, New York, 1997. Viewed 11/12/2006,
`http://ranau.cs.ui.ac.id/book/AlgDesignManual/BOOK/BOOK/BOOK.HTM.`

[64] R Steinfeld, J Pieprzyk, and H Wang. Higher Order Universal One-Way Hash Functions from the Subset Sum Assumption. Public Key Cryptography - PCK 2006. *Lecture Notes in Computer Science* 3958, pp 157-173. Springer-Verlag, Berlin, 2006.

[65] Subset Sum Problem. *Wikipedia,* 2006. Viewed 9/12/2006,
`http://en.wikipedia.org/wiki/Subset_sum_problem.`

[66] X Wang, D Feng, X Lai, and H Yu. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. *Cryptology ePrint Archive,* Report 2004/199, 2004. Viewed 07/12/2006,
`http://eprint.iacr.org/2004/199.`

[67] What is MD5? Tech FAQ. Viewed 09/12/2006,
`http://www.tech-faq.com/md5.shtml.`

[68] G Zemor. Hash Functions and Graphs with Large Girths. Advances in Cryptology - EUROCRYPT '91. *Lecture Notes in Computer Scienece* 0547, pp 508-511. Springer-Verlag, Berlin, 1991.

[69] D Zuckerman. On Unapproximable Versions of NP-Complete problems. *SIAM Journal on Computing* 25(6), pp 1293-1304. 1996.