The application of hash chains and hash structures to cryptography

Thomas Page

Technical Report
RHUL–MA–2009–18
4 August 2009

Department of Mathematics
Royal Holloway, University of London
Egham, Surrey TW20 0EX, England

http://www.rhul.ac.uk/mathematics/techreports

The application of hash chains and hash structures to cryptography

Thomas Page

A thesis submitted for the degree of Doctor of Philosophy.

Royal Holloway, University of London

July 27, 2009

Declaration

These doctoral studies were conducted under the supervision of Prof. Keith Martin and Dr. Siaw-Lynn Ng.

The work presented in this thesis is the result of original research carried out by myself, whilst enrolled in the Department of Mathematics as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Thomas Page
July 27, 2009

Abstract

In this thesis we study how hash chains and other hash structures can be used in various cryptographic applications. In particular we focus on the applications of entity authentication, signatures and key establishment.

We study recursive application of hash functions to create hash chains, hash trees and other hash structures. We collate all these to form a catalogue of structures that we apply to various cryptographic applications.

We study existing work on authentication and create many entity authentication schemes based on structures from our catalogue.

We present a novel algorithm to find efficient signature schemes from any given hash structure. We study some suggestions for suitable hash structures and define a particular scalable hash structure complete with a simple message to signature map that is the most efficient such scheme of which we know.

We explore k-time signature schemes and identify two new properties, which we call perforated and porous.

We look at the application of hash structures to key establishment schemes. We compare the existing schemes and make improvements on many. We present a new key establishment scheme, and show a link between certain k-time signatures and certain key establishment schemes.

We look at the other applications of hash structures, and suggest areas in which our catalogue could be used for further development.

Contents

Acknowledgements

There have been many people who have helped and inspired me to complete this thesis, and I am very grateful for all their support. There are a few who I would like to thank personally, without each of whom I would never have finished.

I would like to thank my supervisors Professor Keith Martin and Doctor Siaw-Lynn Ng for their supervision. I am in awe of the rigour, persistence and friendliness they both showed, even after our meeting had continued for over double the allotted time. In particular I would like to thank Keith for the placement of approximately half of the commas in the following chapters, and Siaw-Lynn for not making corrections to the other half. I have lost count of the number of drafts of this thesis that existed, but Keith and Siaw-Lynn have managed to read and make constructive criticism on all of them.

I would next like to thank my parents. Their advice throughout my time as a postgraduate has been extremely helpful. Particular thanks to my Dad for proofreading the whole thesis and to my Mum for her practical suggestions about the postgraduate process.

Finally I would like to thank my wife Naomi, whose constant encouragement has spurred me on while 'writing up'. Despite her ambivalence towards maths, Naomi has been a keen student while I attempted to explain the work contained in the following pages.

Notation

Symbol	Meaning
$\mathbb{Z}$	The set of integers.
$\{0,1\}^n$	The set of n-bit strings.
$\{0,1\}^*$	The set of finite bit strings.
$\{0,1\}^\infty$	The set of infinite bit strings.
$\|A\|$	The cardinality of set A.
2^A	The power set of A.
x_*	The set of all items of the form x_i.
$x >> y$	The value x is much greater than y.
$f(x) := x$	The function $f(x)$ is defined to be x.
$x\|\|y$	The value x concatenated with the value y.
$x \oplus y$	The bit-wise exclusive-or of x and y.
$f^n(x)$	Recursively defined as the function f applied to $f^{n-1}(x)$.
$g \circ f(x)$	The composition $g(f(x))$.
$\begin{pmatrix} n \\ r \end{pmatrix}$	The number of r-element subsets of an n-element set.
$\mathbb{P}(X)$	The probability of event X.
$A \rightarrow B : m$	A sends the value m to B via an unprotected channel.
$A \xrightarrow{A} B : m$	A sends the value m to B via an authenticated channel.
$A \xrightarrow{S} B : m$	A sends the value m to B via a secured channel.
$A \rightarrow P : m$	A publishes the value m with no authentication.
$A \xrightarrow{A} P : m$	A publishes the value m in an authenticated way.

Chapter 1

Introduction

This thesis considers and evaluates the ways in which hash chains and other hash structures may be applied in many different areas of cryptography.

We begin in Chapter 2 by studying various existing definitions relating to our topic and subsequently develop three working definitions of hash functions. We go on to explore the relationship between hash functions and pseudo-random functions, present ideas from the NIST SHA-3 project, and parameterise hash functions by security and output size, for reference later in the thesis.

Chapter 3 builds on Chapter 2's foundations by consolidating and categorising a large number of structures formed from hash functions. We create the concept of an almost perfect a-ary tree. We choose definitions of hash trees and Merkle trees which will be useful in later chapters. After briefly studying the relationship between the properties of a hash function and the hash chain it forms, we explore the notion of a generalised hash directed acyclic graph (mentioned in other literature), and expand it into a formal definition.

Chapter 4 is concerned with how hash structures from Chapter 3 can be used in entity authentication. We explore this by comparing various existing one-time, unlimited-time and n-time entity authentication schemes. We identify two adversaries that we use in evaluating the security of these schemes. In the course of this investigation we also identify a new property of some entity

authentication schemes, which we refer to as *public verifiability*. We develop several existing entity authentication schemes to provide public verifiability. We present an unlimited-time scheme in which the authenticating and verifying parties have the theoretical minimum storage requirements. Next we go on to create two entity authentication schemes suitable for use with many verifying parties, based on hash chains and Merkle trees respectively. In further consideration of Merkle trees we show a relationship between their use in an entity authentication scheme and a one-time signature scheme. We exhibit an entity authentication scheme based on the hierarchical chain construction, and describe its potential use in sensor networks. Furthermore we suggest similar adaptations of three other hash structures.

In Chapter 5 we look at the topic of signatures, beginning by comparing one-way trapdoor based signature schemes with hash-based one-time signature schemes, and evaluating various existing one-time signature schemes. We prove that any set of Vaudenay's rake signature patterns with constant average chain position is a signature scheme and exhibit some properties of these signature schemes. We also show that the optimal choice of parameters for a Vaudenay's rake signature scheme is dependent on more than simply the size of the signature space required.

We prove that for any signature scheme based on a hash tree there is a scheme based on a Merkle sub-tree with the same number of signature patterns. We present and evaluate two algorithms — the first of which finds a large signature scheme based on any given hash structure, and the second of which finds the largest such signature scheme. We then use this second scheme to find graphs admitting particularly efficient one-time signature schemes. We generalise a signature scheme due to Bleichenbacher and Maurer and demonstrate that (in the limit) the original scheme is the most efficient. We then proceed to create two one-time signature schemes, which are the most efficient for signing blocks of six bits and eight bits respectively. We identify and define two new properties of k-time signature schemes which we call *porous* and *perforated*, and subsequently suggest a new method for the construction of porous k-time signature schemes. Finally we explore the concept of efficiency for k-time signature schemes, and present two distinct

but useful definitions.

Chapter 6, on key establishment schemes, begins with a discussion comparing many key predistribution schemes (KPSs) and key distribution schemes (KDSs). From this we generalise some existing KPSs to form a new scheme which we call the inverted hash tree KPS. We then highlight a similarity in the way that hash chains are used by Leighton and Micali to make a KPS, and by Reyzin and Reyzin to make a k-time signature scheme. We present our own KPS for lattice-shaped hierarchies, and then generalise it to many more hierarchies. In looking at a key escrow scheme by Joye and Yen we observe that it can be applied to any set of group keys formed by a KPS. We develop a KDS due to Chang et al. by facilitating the addition of new users and reducing the TA's storage requirements. We exhibit a flaw in a KDS by Kurnio et al. We note the inconsistencies in the literature over the definitions of forward and backward secrecy, and propose definitions for the terms *strong forward* and *strong backward* secrecy. We compare several hash-based key refreshment schemes and then go on to propose a key refreshment scheme based on the chained pseudo-random number generator, and a simple extension of it.

In Chapter 7 we make some brief comparative observations about various hash-based schemes in several other areas of cryptography, including micropayments, auctions and pseudo-random number generators. We also mention hash structure based schemes for interval release cryptography and preimage computation. We generalise an interval release scheme due to Joye and Yen to many dimensions.

Chapter 2

Hash functions

2.1 Introduction

In this chapter we will review definitions of hash functions with an ultimate goal of fixing three definitions of hash functions that we will use in the rest of this thesis. We discuss the main properties — preimage, second preimage and collision resistance, as well as the relationship between hash functions and pseudo-random functions. Finally we comment on the ongoing NIST project to standardise hash functions for future use.

2.1.1 Differences in definitions

Many definitions of hash functions exist. One aim in this chapter is to explore current definitions and the reasoning behind them.

Existing definitions differ in three main areas. Firstly definitions differ over the domain and range of a hash function. Some prefer to err on the side of theoretical generality by allowing the domain to be any set, and the range to be any finite set, while others opt for a more practical approach by fixing the domain and range to be sets of binary strings restricted by their length.

The second variation between definitions is the (often subtle) difference in concepts of "hard" and "easy". Original papers on the topic of hash functions were understandably simplistic in how they defined computational challenges. As cryptographers have developed notions such as provable security, they

have also improved and expanded precisely what these properties mean. Even now that rigorous definitions have been published, many informal definitions are still used because they are easier to understand and express the essence of what is required.

It is clear that if a problem is "computationally infeasible" then we should not be able to use a computer to find a solution; however it is difficult to come up with an absolute number of operations that the computer should have to do to find a solution. Unless specified otherwise we will use the following definitions for computationally infeasible and computationally easy from [88].

Definition 2.1. A problem is **computationally infeasible** if the best known algorithm to solve it requires an unreasonable amount of computational time.

Definition 2.2. A problem is **computationally easy** if the best known algorithm solves it in (expected) polynomial time, and only requires available resources.

Thirdly, definitions differ over the precise properties that a hash function needs. Some authors have a specific application in mind and so define a hash function to have application-specific properties, while other authors propose a more general notion of a hash function and define variants of it with extra properties. Exploring the differences in terms of properties will be the primary focus of this chapter.

2.1.2 Initial informal definitions and motivation

The term "hash function" is due to computer science, and refers to a function which compresses an arbitrary length input bit string to an output bit string of fixed finite length [111, 141]. These functions are primarily used to speed up the process of finding stored data [151]. However the term "hash function" has since been adopted by cryptographers, and desirable properties of cryptographic hash functions, such as "one-wayness" and "collision resistance", have been identified.

When discussing the various definitions used for hash functions in cryptography it will be useful to have fixed informal definitions of several properties that a *general function* might have. We emphasise that these properties could apply to a general function because we have not yet defined what we mean by a "hash function".

Definition 2.3. A function $f : D \to R$ is **preimage resistant** if for a given $y \in R$ it is computationally infeasible to find an $x \in D$ such that $f(x) = y$.

Definition 2.4. A function $f : D \to R$ is **second preimage resistant** if for a given $x \in D$ it is computationally infeasible to find $x' \in D$ with $x' \neq x$ such that $f(x') = f(x)$.

Definition 2.5. A function $f : D \to R$ is **collision resistant** if it is computationally infeasible to find $x, x' \in D$ with $x' \neq x$ such that $f(x) = f(x')$.

There are some well known upper bounds on the computation required to exhibit a preimage, second preimage or collision [83]. For a function with an n-bit range a preimage can be found for a given value by simply guessing at input values. On average a preimage will be found after trying no more than half of the input values, which equates to 2^{n-1} function evaluations.

Finding a second preimage requires approximately the same computation as finding a preimage, however a finding a collision is typically much easier. Using the birthday paradox we can see that for a function with an n-bit range a search of input values will provide a collision in an average of $2^{n/2}$ function evaluations.

The reason for choosing to define these three properties can be justified by three potential applications of such functions.

- **Preimage resistance**

  If a computer system stores a user's password, then there is a chance that a hacker will be able to recover that password. However if the computer applies a preimage-resistant function f to the password p before storing it, then a hacker can only recover $f(p)$. The preimage resistance of f means that from this value the hacker cannot deduce a valid password.

- **Second preimage resistance**

  To justify second preimage resistance, consider a respected software writer who wants clients to be confident that they have received the correct program before they run it. The software creator could apply a second preimage resistant function f to their program file p and publish the result on their official website. A user can apply f to the program file p' that they have obtained, and check that $f(p') = f(p)$. If the two values match, they can be confident that they have the original program because it is computationally infeasible for anyone to make another program file with the same output under f.

- **Collision resistance**

  Collision resistance is essential for a function f to be used as the compression function in the process of making a digital signature for a message m. The process has two steps; compressing the message to $f(m)$, and then processing $f(m)$ with a signature key to get the signature s. Typically another person can verify the signature by processing s using the corresponding verification key, and checking that the result is $f(m)$ (which they can compute from m). If the hash function f is not collision resistant, then it may be possible to find two messages which have the same signature regardless of the private signature key used.

2.1.3 Confusion in the literature

One of the first authors to define a hash function was Merkle [84], who defined a hash function f to be any function with the following properties:

1. The function f can be applied to an input of any size.
2. The output of f is a bit string of fixed length.
3. The output $f(x)$ is computationally easy to calculate for any x.
4. The function f is preimage resistant (see Definition 2.3).
5. The function f is second preimage resistant (see Definition 2.4).

Rabin used the same definition in [114]. This definition is also sometimes known as a *weak hash function* [83]. Other authors define hash functions without preimage resistance or second preimage resistance [127, 131], or with additional properties such as collision resistance [78], or such that the input should include a key [134]. Many more authors use the term hash function without specifying the precise properties they require.

Preneel highlights in his thesis [111] the different terms used for output of a hash function.

> "... the result of a hash function has been given a wide variety of names in the cryptographic literature: hashcode, hash total, hash result, imprint, (cryptographic) checksum, compression, compressed encoding, seal, authenticate, authenticator, authentication tag, fingerprint, test key, condensation, Message Integrity Code (MIC), message digest, etc."

We will refer to the output of a hash function as the "output", the "hash value", or simply the "hash".

2.1.4 Baseline definition of a hash function

We now propose a baseline definition for a hash function. This definition is too weak for most cryptographic uses, but is inclusive of any other definition of a hash function. It is also consistent with many other sources [27, 83, 88, 111].

Definition 2.6. A **hash function** is a function $f : D \to R$ such that:

1. The domain D is a set that may be infinite or finite.
2. The range R is a finite set.
3. The function is computationally easy to evaluate for any given input $x \in D$.

We leave open the possibility that D is finite, since for many practical situations it will be possible to put a limit on the size of the input. Note

that we do not require that a hash function compresses (therefore we allow $R = D$) and therefore our definition includes one-way permutations, which are particularly useful for hash chaining (see Chapter 3).

Even amongst the sources we have cited there is discrepancy over whether D and R *must* conform to stricter constraints such as being sets of binary strings, or sets of fixed length binary strings. In [64], condition 1 is that D is "the set of all binary sequences of some specified minimum length or greater". Due to the nature of information, we are tempted to write that D and R must be finite length bit strings. In fact for most software-based applications, efficient hash functions should operate on 64-bit blocks (assuming the processor has a 64-bit architecture), and so D and R should both be of the form $\{0, 1, 2, \ldots, 2^{64} - 1\}^*$.

A common practice when using hash functions to validate files is to use a binary file (i.e. a member of $\{0, 1\}^*$ or $\{0, 1, 2, \ldots, 2^{64} - 1\}^*$ (see above)) as the input, and to output text representing a hexadecimal string of the form $\{0, 1, \ldots, 9, a, \ldots, f\}^n$. Although it is clear that the hexadecimal string could equally be represented as a member of $\{0, 1\}^{4n}$, and indeed the hash function was probably designed with that in mind, it is not strictly true that $\{0, 1, \ldots, 9, a, \ldots, f\}^n \subset \{0, 1\}^*$, and so we leave $R \subseteq D$ as a suggestion rather than a strict rule.

2.2 Preimage-resistant and second preimage resistant functions

2.2.1 Motivation and Definitions

The idea of "one-way functions" originates from Wilkes [153] (page 91), where it was used in connection with login procedures. The definition of a one-way function seems to be agreed by the majority of authors (for example [83, 134]) to be the same as a preimage-resistant function (see Definition 2.3).

However it is also widely agreed that a "one-way *hash* function" is a hash function (as in Definition 2.6) which is preimage resistant *and second*

preimage resistant [111, 83, 141].

Menezes et al. in [83] (Remark 9.19) acknowledge that this discrepancy between the definition of one-way in the context of hash functions and one-way in the context of general functions causes ambiguity.

It should be noted that in [111], Preneel designs his definition of a "one-way function" to match up with the standard definition of a "one-way hash function" (i.e. both of them are preimage resistant *and* second preimage resistant).

Occasionally a less restrictive and more logical definition of "one-way hash function" appears, omitting second preimage resistance [88], but this non-standard use just adds further ambiguity to the term "one-way", and so we will try to avoid using the term "one-way" whenever possible. It is quite conceivable to think of a situation when we require preimage resistance but not second preimage resistance (for an example see Section 2.1.2), so we propose to use the term "preimage-resistant hash function" for this.

Definition 2.7. A **preimage-resistant hash function** $f : D \to R$ is a function with the following properties:

1. The domain D is a set that may be infinite or finite.

2. The range R is a finite set.

3. The function is computationally easy to evaluate for any given input $x \in D$.

4. The function f is preimage resistant (see Definition 2.3).

As mentioned in Section 2.1.3, a hash function which is both preimage resistant and second preimage resistant is also known as a "weak hash function" [83, 127].

Definition 2.8. A **weak hash function** $f : D \to R$ is a function with the following properties:

1. The domain D is a set that may be infinite or finite.
2. The range R is a finite set.
3. The function is computationally easy to evaluate for any given input $x \in D$.
4. The function f is preimage resistant (see Definition 2.3).
5. f is second preimage resistant (see Definition 2.4).

In other words a weak hash function is a preimage-resistant hash function that is also second preimage resistant.

2.2.2 Adversarial models

If we want to know whether a cryptographic scheme is secure, we are really asking whether someone might be able to break it somehow. A traditional approach to this is to model the scheme as a game which is set for some adversary. To win the game the adversary must break the scheme with some non-negligible probability, and we hope that there is no such way to do so.

In some cryptographic schemes there are parameters which the adversary may or may not know (for example the plaintext in an encryption scheme). In some security models an adversary may have access to previous data (such as plaintext/ciphertext pairs), or may be able to create data of their own before the game begins (for example by persuading the challenger to encrypt some messages for him). It is important to define exactly what an adversary must do in order to win the game/break the scheme.

It is sometimes possible to prove the equivalence of the security of a cryptographic scheme and the difficulty of a well known problem in mathematics (for example the Rabin public-key encryption scheme reduces to the difficulty of factoring products of two large primes). In this way we can prove that either our scheme is secure, or the well known problem can be easily solved (and we hope the former is true).

2.2.3 Formalisation

As we just saw, it is important to be precise when stating security properties. Historically the informal definitions of hash functions have presented problems for cryptographers trying to prove the security properties of algorithms that use hash functions.

A property of both preimage-resistant hash functions and weak hash functions is preimage resistance (for a given $y \in R$ it is computationally infeasible to find an $x \in D$ such that $f(x) = y$).

The way that y is chosen from R is very important since, if defined wrongly, preimage-resistant hash functions would not even exist. As an example, if the adversary is allowed to choose their own y then they could first pick $x \in D$ and then chose $y = f(x)$, allowing trivial inversion. In [122] the authors examine as an example the following definition of preimage resistant quoted from [83]:

> *preimage resistance — for essentially all pre-specified outputs*, it is computationally infeasible to find any input which hashes to that output, i.e., to find any preimage x' such that $h(x') = y$ when given any y for which a corresponding input is not known.

Although one immediately understands the sense of what is meant by this definition of preimage resistance, it is not clear, even in theory, how to check if a function exhibits this property. How would one *pre-specify* the outputs? Should they be chosen uniformly at random from R, or uniformly at random from D and then hashed, or is the challenger allowed to pick a particularly difficult set of outputs?

For the purposes of this thesis we will consider challenge values y to be chosen uniformly at random in R, but there are other reasonable ways of chosing y.

Rogaway and Shrimpton [122] considers three challenges which could all be considered tests of preimage resistance. These challenges are expressed with keyed hash functions in mind, but for an unkeyed hash function a key could be introduced as a value appended to the front of the message before

hashing. The three challenge settings are with a random key and a random y, a fixed key and a random y, and with a random key and a fixed y. It is possible to consider a choice of key as the same as a choice of hash function from a family of hash functions. Rogaway and Shrimpton applied each of these settings to the problem of finding a preimage for a hash function (i.e. preimage resistance as in Definition 2.3) and studied the relationships between them.

We are interested in the case of a fixed key and random y, since this best corresponds to most uses of hash functions in practice.

As well as preimage resistance, weak hash functions also raise the problem of formally defining second preimage resistance. One well-known attempt to formally define second preimage resistant functions is the universal one-way hash function (UOWHF) [91].

Essentially, a UOWHF family is such that no adversary is "good" at finding a second preimage for some challenge value x (which it can choose), when the hash function is selected uniformly at random from the family. For the adversary to be "good", they have to do better on average than an exhaustive search would manage.

Much more recently Rogaway and Shrimpton gave a rigorous study of preimage-resistant and second preimage resistant functions (as well as collision-resistant functions, which we will look at in the next section) [122]. As previously mentioned, the basic considerations are whether the challenge value is randomised, and whether the choice of hash function is randomised. For preimage resistance the challenge is $f(x)$, whereas for second preimage resistance the challenge is x. If both the challenge and the key are fixed then there exists a trivial algorithm which breaks either property (assuming a solution exists) — the algorithm that tries the correct value first. This amounts to six properties for hash functions, which are the first six properties given in Table 2.1.

The relationship between these properties is discussed in Section 2.5.

Name	Property	Where found
Pre	Preimage resistant with random challenge and random key	
ePre	Preimage resistant with fixed challenge and random key	
aPre	Preimage resistant with random challenge and fixed key	Property 4 in Definition 2.7
Sec	Second preimage resistant with random challenge and random key	
eSec	Second preimage resistant with fixed challenge and random key	UOWHF [91]
aSec	Second preimage resistant with random challenge and fixed key	Property 5 in Definition 2.8
Coll	Collision resistant with random key (no challenge)	Definition 2.9 in Section 2.3.1

Table 2.1: The seven properties discussed in [122]

2.3 Collision-resistant hash functions

The idea of a "collision-resistant hash function" (CRHF), or "strong hash function", has existed for a long time, although precise definitions differ. Other commonly used names for this type of function include "cryptographic hash function" [138, 148], and "secure hash function" [134].

2.3.1 A traditional formal definition

One of the definitions was due to Merkle [86], who defined a "strong hash function" to be any function f with the following properties:

1. The function f can be applied to an argument of any size.
2. The output of f is a bit string of fixed length.
3. The output $f(x)$ is relatively easy to compute for any x.
4. The function f is collision resistant (see Definition 2.5).

This definition is also used by Menezes et al. in [83].

However, there are many slightly different definitions. In [26], [127] and [141], a "collision-resistant" or "strong" hash function f is any function with the following properties:

1. The function f can be applied to an argument of any size.
2. The output of f is a bit string of fixed length.
3. The output $f(x)$ is relatively easy to compute for any x.
4. The function f is preimage resistant (see Definition 2.3).
5. The function f is collision resistant (see Definition 2.5).

One potential definition of collision resistance is that a function is collision resistant if there does not exist an adversary which can exhibit a collision in (on average) less than the time it would take to do an exhaustive search for one. However if collisions do exist (and they usually will, since typically $|D| > |R|$), then under this model no collision-resistant functions exist. This is because there exist adversary algorithms which try any pair of values as their first guess, so if the distinct values x and x' happen to collide then there exists an adversary algorithm which tries those two values first. This adversary will succeed in a very short time every time it is run, and so the function is not collision resistant in the traditional sense.

However, the traditional interpretation of preimage resistance and second preimage resistance are both fine. In both of these the adversary is supplied with a randomly chosen challenge value, meaning that any adversary which simply guesses at values of D in a set order will (on average) do no better than an exhaustive search for a solution.

This leads us to the idea that we would like to randomise the challenge for collision resistance in the same way that it is randomised for preimage and second preimage resistance. Since we have no challenge value (the adversary can unrestrictedly pick values as potential collisions) the only thing left to randomise is the choice of hash function.

Consequently there seems to be a need to define collision resistance in terms of collision-resistant hash function *families* and then, when needed, a hash function can be chosen from the family. Damgård [25] defined a "collision-free hash function family", which is a bit misleading as it does actually contain collisions; they are just hard to find. We will refer to the family as a fixed-size collision-resistant hash function family and define it as follows:

Definition 2.9. A **fixed-size collision-resistant hash function family** is a set of hash functions with the following properties:

- There is a probabilistic polynomial-time algorithm which, given a security parameter, selects a member of the family uniformly at random.[1]
- All functions in the family are computationally easy to evaluate.
- The problem of finding $x \neq x'$ such that $f(x) = f(x')$ for a given f in the family is computationally infeasible to solve.

It is important to note that we have a *family* of hash functions, which is such that no adversary algorithm can easily find a collision for a hash function chosen at random from the family.

In Table 2.1 this type of collision resistance is our final property, and the relationships between fixed-size collision-resistant hash functions and the other types of hash functions in the table are explored in Section 2.5.

2.3.2 A working definition

In practice a particular hash function (for example SHA-1) is chosen for a particular application. This is obviously different from the formal definition above, as there is no family from which SHA-1 is chosen. This discrepancy between (historical) theory and practice is picked up by Rogaway, and we will now discuss his simple solution that appears in [121].

[1] The security parameter is used in [25] to help define what they mean by "computationally infeasible".

In most traditional security proofs there is a protocol and a cryptographic primitive which we make some assumption about (for example a hash function which we assume is collision resistant). The proof typically shows that if there is an algorithm which breaks the protocol then there also exists an algorithm that violates the assumption (this would show either the protocol is secure, or that collision-resistant hash functions do not exist).

As we have already discussed above (in Section 2.3.1) for any given hash function there does exist an algorithm that finds a collision for it, so it is not sensible to use this as the assumption. The idea which Rogaway [121] uses is to restructure the proof of security, and thereby base the security on a different assumption. Rogaway defines three forms of proof as follows:

1. existential form (C0): If there is an effective algorithm A for attacking protocol Π then there is an effective algorithm C for finding collisions in H.

2. code-constructive form (C1): If you know an effective algorithm A for attacking protocol Π then you know an effective algorithm C for finding collisions in H.

3. blackbox-constructive form (C2): If you possess effective means A for attacking protocol Π then you possess effective means C for finding collisions in H.

These second two forms are much better for collision resistance because it is generally true (for "good" hash functions) that no one can find an effective algorithm for producing collisions, and also that no one has the computational power to find collisions.

We will use Rogaway's "code-constructive form" for our definition of a strong hash function.

Definition 2.10. A **strong hash function** $f : D \to R$ is a function with the following properties:

1. The domain D is a set that may be infinite or finite.
2. The range R is a finite set.
3. The function is computationally easy to evaluate for any given input $x \in D$.
4. The function f is preimage resistant (see Definition 2.3).
5. The function f is second preimage resistant (see Definition 2.4).
6. No one can find an efficient algorithm to find a collision for f.

In other words a strong hash function is a weak hash function with the addition of Property 6.

Clearly we cannot hope to *prove* that a function is collision resistant in this sense without proving that no efficient algorithm exists to find collisions, but in practice it is a realistic definition.

2.4 Pseudo-randomness

Although many authors do not mention pseudo-randomness in their discussion of hash functions, for some applications it is a required property. Real life applications of hash functions include pseudo-random number generators [46], stream cipher keystream generators [8] and encryption key generators (see Section 6.5). Proofs of security often formally model hash functions as "random oracles", which are effectively functions producing random output [5, 147]. A few authors require pseudo-randomness of output as a property in their definition of a hash function; for example [78] gives the following properties that their definition of a hash function f must have:

1. The function f can be applied to an argument of any size.
2. The output of f is a bit string of fixed length.

3. The output $f(x)$ is relatively easy to compute for any x.

4. The function f is preimage resistant (see Definition 2.3).

5. The function f is collision resistant (see Definition 2.5).

6. *On any input x, the output hashed value $f(x)$ should be computationally indistinguishable from a uniformly random binary string of the same length.*

It is not clear how the x in Property 6 should be chosen so as to compare the output to a random string. It is also not clear how to define *computationally indistinguishable* for a fixed (unkeyed) hash function. This definition is informal and aims to aid understanding of hash functions, not to be rigorous.

Putting aside rigour for the moment, it seems intuitive that if the output is random then no amount of information about the output gives any information about the input, and so it would immediately satisfy preimage resistance. However since we want the hashing process to be deterministic ($f(x)$ is a fixed value) the idea of pseudo-randomness is appealing.

It is not easy to define what is meant by the output of a hash function being pseudo-random. With applications such as pseudo-random number generators it is clear that the aim should be that output values should bear no correlation to previous output values, but with hash functions there is no obvious ordering of the output words.

In [98] Okamoto uses the term "correlation-free one-way hash functions". In [2] Anderson gives a simplified definition, defining a hash function f as "correlation free" if the best way to find two distinct values x and x' such that $f(x)$ has most of its bits in common with $f(x')$ is just to guess. He goes on to give a proof that it is a strictly stronger notion than collision resistance.

The proof has two parts. Clearly if you can easily find collisions then you can also easily find output values with many bits in common. Secondly, if you have a hash function f for which it is hard to find collisions, then you can define a new hash function which is not "correlation free", but which is still collision resistant, namely $g(x) := x_1||f(x_2)$, where x_1 is the first k bits of x and x_2 is the rest.

Anderson continues in [2] by giving several other potentially desirable properties which are not implied by this definition of "correlation free". These include "complementation freedom" (it is infeasible to find x and x' such that $f(x)$ is the complement of $f(x')$), "addition freedom" (it is infeasible to find x, x' and x'' such that $f(x) = f(x') + f(x'')$) and "multiplication freedom" (it is infeasible to find x, x' and x'' for some set of N such that $f(x) = f(x')f(x'') \mod N$).

However, intuitively pseudo-randomness should include all of these definitions since if the output of a function is pseudo-random then it should not be feasible to distinguish the output from a uniform random variable (assuming the input is not repeated). This much stronger notion of pseudo-randomness is the definition of a "random oracle".

In [5] a *random oracle* R is defined as a map from $\{0,1\}^*$ to $\{0,1\}^\infty$ chosen by selecting each bit of $R(x)$ uniformly and independently, for every x. Although no protocol actually uses an infinitely long output, this removes the need to specify how long "sufficiently long" is.

Clearly a fixed hash function can never satisfy this definition, but many cryptographers model hash functions as random oracles in order to aid proofs of security [5, 72, 56].

As we have seen, there exist some attempts to define the properties required for a "pseudo-random hash function", some practical, and some impractical, but we will not try to resolve this complex issue here.

2.5 Relationships between the definitions

2.5.1 Introduction

We have now defined several types of hash functions including preimage-resistant hash functions (Definition 2.7), weak hash functions (Definition 2.8) and strong hash functions (Definition 2.10), which are all hash functions with different additional properties. We would like to know which of these properties are the strongest, and which properties are independent of one another. We will also discuss the work in [122], which explores all the relationships between the types of hash function given in Table 2.1.

2.5.2 Second preimage resistance and collision resistance

One common statement in introductory texts on hash functions is that collision resistance implies second preimage resistance [83, 111]. To understand this we need to go back to the informal definition of collision resistance (Definition 2.5): f is such that it is computationally hard to find $x \neq x'$ with $f(x) = f(x')$.

If a function is not second preimage resistant (aSec in Table 2.1) then for any given x it is feasible to find $x' \neq x$ with $f(x) = f(x')$. Consequently it is also easy to find a collision (by picking our own value for x), and so this definition of collision resistance implies second preimage resistance.

Due to technicalities, neither of the formalisations of collision resistance that we have looked at preserve this implication.

The family of collision-resistant hash functions in Definition 2.9 are collision resistant as a group, whereas for second preimage resistance we want a specific function to be second preimage resistant. This type of collision resistance is referred to as Coll in Table 2.1 and [122].

The second formalisation of collision resistance is the one in our definition of a strong hash function (Definition 2.10), which technically does not imply second preimage resistance either. Instead it is implied that no one is able to find a second preimage, but this does not ensure that there does not exist a program to find one (which is the standard assumption for second preimage resistance).

However it seems reasonable to assume that in practice any hash function thought to be collision resistant will also be second preimage resistant.

2.5.3 Preimage resistance and collision resistance

Another well-studied relation is the link between collision resistance and preimage resistance. It is agreed that preimage resistance (aPre in Table 2.1) is not a stronger condition than collision resistance.

In [128] it is shown that preimage resistance does not necessarily follow from collision resistance. Whereas in [134], they use the Pre definition of

preimage resistance from Table 2.1, and show that Coll implies Pre.

Our statement of collision resistance in the definition of a strong hash function (Definition 2.10) does not imply preimage resistance for the same reason that it does not imply second preimage resistance: it is a statement about the capability of humans and preimage resistance is a statement about hypothetical algorithms.

In [83] (Note 9.20) it is explained that although there exist pathological examples of collision-resistant hash functions which are not preimage resistant, for collision-resistant hash functions used in practice it seems *reasonable to assume* that they are also preimage resistant.

2.5.4 Preimage resistance and second preimage resistance

The final relationship that we will consider is the one between preimage resistance and second preimage resistance. It is generally agreed that preimage resistance does not imply second preimage resistance, but we would like to know if the converse is true.

From [122] we find that under certain conditions second preimage resistance implies preimage resistance. The implication is that if the best adversary (for running time t) has advantage δ when guessing a second preimage for a randomly chosen challenge x, then the best preimage guessing adversary (for running time t minus the time to hash c messages) has advantage of at most $c\delta + 2^{m-n}$, where n is the hash length and m is the maximum input text length.

For practical hash functions with unlimited input size m this does not really say anything. However we will often be dealing with applications where the domain is the same as the range of the hash function, and so this is a result worth noting.

The relationships between the other properties in Table 2.1 are also explored in [122], and the full set of cases is given in Table 2.2.

Property	⇒ Pre	⇒ ePre	⇒ aPre	⇒ Sec	⇒ eSec	⇒ aSec	⇒ Coll
Pre	✓	✗	✗	✗	✗	✗	✗
ePre	✓	✓	✗	✗	✗	✗	✗
aPre	✓	✗	✓	✗	✗	✗	✗
Sec	*	✗	✗	✓	✗	✗	✗
eSec	*	✗	✗	✓	✓	✗	✗
aSec	*	✗	*	✓	✗	✓	✗
Coll	*	✗	✗	✓	✓	✗	✓

Table 2.2: The relationships between the seven properties discussed in [122]. The * symbol indicates that there is an implication under certain conditions.

2.5.5 The relationships between our hash functions

We have defined three types of hash function which we will focus on: preimage-resistant hash functions (Definition 2.7), weak hash functions (Definition 2.8) and strong hash functions (Definition 2.10).

Although in the above we have seen that our definitions of preimage resistance, second preimage resistance and collision resistance do not technically imply each other, it seems that in practice (excluding pathological examples) collision-resistant functions are also second preimage resistant, and second preimage resistant functions are also preimage resistant. However even without this unsubstantiated claim, our types of hash function do have an ordering. Strong hash functions are a specific type of weak hash function, and weak hash functions are a specific type of preimage-resistant hash function. This can be seen by comparing properties of the definitions.

2.6 Parameterising hash functions

Traditionally the security of a hash function is given by the number of bits that the output has. However for some applications we would like to use a fast hash function which need not be as secure as the output size suggests.

We therefore suggest two parameters for any hash function: the output size and the security parameter. The output size is simply the number of bits that the output of the hash function has.

The security parameter $s \in [0, 1]$ relates the output size to the actual

security. The simplest way to explain this is with an example. If we have a 0.5-secure 256-bit preimage-resistant hash function then the amount of effort to find a preimage is approximately $2^{(256\times 0.5)-1} = 2^{128-1}$ (therefore the same amount of effort that would be required to find a preimage for a 1-secure 128-bit preimage-resistant hash function). We would typically hope that the function is faster to evaluate than a 1-secure 256-bit preimage-resistant hash function.

Definition 2.11. An ***s*-secure *l*-bit strong hash function** ($s \in [0,1]$, $l \in \mathbb{Z}$) is a hash function with an l-bit output such that the amount of effort to find a preimage or second preimage is $2^{l\cdot s-1}$ and the amount of effort to find a collision is $2^{(l\cdot s/2)}$.

Whenever we do not specify s for a hash function we assume that it is 1, and whenever we do not specify l we assume it to be any suitably large value (such as 128 or 256).

2.7 The NIST project to create SHA-3

We have seen that there is a reasonable amount of confusion in the literature about exactly what properties a hash function should have. On the whole we have seen that as definitions get more rigorous, the gap between theory and practice grows.

In recent years, a number of attacks on hash functions have been devised. In particular there have been attacks on MD5 [146] and SHA-1 [145], hash functions which have both been in widespread use. Since NIST (the National Institute of Standards and Technology) was responsible for the standardisation of SHA-1, it took on responsibility for the creation of a replacement hash function standard.

SHA-2 [2] already existed at the time, and consequently NIST suggested that for certain uses (digital signatures, digital time stamping and other applications that require collision resistance) SHA-2 should be used as a

[2]The *SHA-2* family of hash functions consists of SHA-224, SHA-256, SHA-384 and SHA-512, which were standardised as alternatives to SHA-1 [94].

replacement for SHA-1 [92]. Many cryptographers have speculated that similarities in structure between SHA-1 and SHA-2 may indicate that attacks on SHA-2 will be found in the next few years [22].

NIST organised two workshops [22, 96] and then used the feedback from these workshops to create a specification for a new hash function [95]. The selected hash function family would become known as SHA-3, and is intended to replace instances of SHA-1 and SHA-2.

2.7.1 The properties required of SHA-3

The NIST call [95] listed the following properties required of any submitted hash function:

- The hash function should be publicly disclosed and available worldwide without royalties or intellectual property restrictions.
- The hash function should be suitable for implementation on a wide range of platforms.
- The hash function must be capable of producing output of length 224, 256, 384 and 512 bits.

NIST also specified that any submitted hash function would be ranked by the following criteria (listed in order of importance):

1. Security — Any submission must be secure for use in many applications, including digital signatures, key derivation and pseudo-random number generation. Additionally the submissions are expected to be preimage resistant, second preimage resistant and collision resistant.
2. Cost — Submissions should ideally be computationally efficient on a wide range of platforms. They will also be rated on the amount of data storage required, and the number of gates required for hardware implementations.
3. Algorithm and implementation characteristics — Any flexibility allowed by the hash function will be viewed in a positive light. For

example it would be good to provide an intuitive way to increase security at the cost of greater computational complexity (such as allowing the number of rounds to be increased). Assuming many algorithms are presented which are similar in terms of all the previously listed criteria, then they will be ranked by the relative design simplicity.

2.7.2 Relevance to this thesis

Any hash function satisfying the properties in Section 2.7.1 will be suitable for use as a preimage-resistant, weak or strong hash function by our definitions (assuming the definitions of preimage resistant, second preimage resistant and collision resistant match).

Since this thesis is founded on the assumption of the existence of 'cryptographically secure' hash functions, the SHA-3 project is of interest to us. However the remainder of the work presented here is only dependent on the properties that a hash function has, not on the specific hash function. Consequently we will look no further at the development of the SHA-3 project in this thesis.

2.8 Conclusion

For the purposes of this thesis we have chosen to define three types of hash functions: preimage-resistant hash functions, weak hash functions and strong hash functions. The formal defining of types of hash functions is a very complex area of ongoing research, however these three definitions will suffice for our subsequent discussions.

Chapter 3

Hash structures

In this chapter we will discuss many different methods of combining hash functions, which will be used in later chapters. The major contribution of this chapter is to consolidate and categorise all structures formed from hash functions, and this will serve as an essential reference for later chapters. We will look at *hash chains*, *hash trees* and *hash directed acyclic graphs* (or DAGs), before moving on to *generalised hash DAGs*. We will then identify a number of specific instances of generalised hash DAGs, including *inverted hash trees*.

3.1 Hash chains

We begin this section with an example, motivating the further study of hash chains. We will then look at some basic properties of hash chains and outline how these properties make hash chains useful for certain applications. We end the section by mentioning some further properties of hash chains.

3.1.1 Motivation and definition

As we saw in Chapter 2, a hash function's range is usually a subset of its domain. This naturally leads to the idea of applying a hash function to its own output.

One of the earliest examples of this being used is Lamport's password authentication scheme [69], which we discuss in more detail in Chapter 4, but give a summary now.

We consider the scenario of a user who wants to log on to a computer remotely, and who has to send the password across an insecure channel. To prevent an eavesdropper intercepting the password and then using it at a later time, the user could have a set of passwords $\{x_0, x_1, \ldots, x_{1000}\}$ stored on the computer, and once a password is used it becomes invalid.

Alternatively, to minimise the storage requirements for the computer and the user, the value x_0 could be chosen by the user and the other values are defined as $x_i = f(x_{i-1})$ for some preimage-resistant hash function f. To initiate the system it is only necessary to store x_{1000} on the protected computer. The computer also stores the index of the password that should be used next (initially 1).

When the user wishes to remotely access the computer for the ith time they send their username to the computer, which replies with a request for the ith password. The user enters x_0 into the terminal which computes the value x_{1000-i} by applying the hash function i times to x_0. The value x_{1000-i} is then sent to the computer. The computer checks that applying the function f to the value received gives the next chained value $x_{1000-i+1}$ (which the computer will have stored from the last session). The computer now stores x_{1000-i} (and $i+1$) and no longer needs to remember $x_{1000-i+1}$ (or i).

We will refer to this construction of linked hash functions as a *hash chain*.

Definition 3.1. A **hash chain** $\mathcal{C}$ is a set of values $\{x_0, \ldots, x_n\}$ for $n \in \mathbb{Z}$ such that $x_i = f(x_{i-1})$ for some hash function f, where $i \in [1, n]$ and x_0 is a valid input for f.

The *length* of a hash chain is the number of hash function evaluations required to create the hash chain. A hash chain with values $\{x_0, \ldots, x_n\}$ has length n. Note that this is very similar to the definition of the length of a path in graph theory.

3.1.2 Basic properties of a hash chain

It is useful to think of a hash function as a directed link from an input to an output. A hash chain will look like a line of directed links:

$$x_0 \longrightarrow x_1 \longrightarrow x_2 \longrightarrow \cdots \longrightarrow x_n$$

1. In the definition of a hash function (Definition 2.6) we demand that a hash function is computationally easy to evaluate for any given input $x \in D$. This in turn implies that values in a hash chain will be easy to evaluate for any given input $x_0 \in D$.

2. As we noted in the example of Lamport's password authentication scheme, a hash chain is very efficient to store; we only require the input value x_0 in order to calculate any other value in the chain.

3. If our hash function's output is in some sense pseudo-random (see Section 2.4) then we can represent a large set of pseudo-random values by just one value (x_0). It should be noted that "pseudo-random" values generated in this way are not suitable for all situations, as we will see later in Section 3.4.4.

4. A final trivial property of a hash chain is that any value x_i can be calculated from any value x_j if $i \geq j$.

3.1.3 Basic applications

The properties from Section 3.1.2 each have an impact on the applications that may be able to employ hash chains.

The fact that the hash chain is directed makes it suitable for situations where repeated authentication is required, for example Lamport's password authentication scheme [69], which we mentioned in Section 3.1.1. This requires us to use a preimage-resistant hash function, otherwise it becomes possible to work out future authentication values from previous ones (that is it becomes possible to work out x_{i-1} from x_i).

The ease of creation and storage of a hash chain, make it a particularly suitable tool for platforms with limited memory and processing power.

If the hash function used in our hash chain allows us to generate a large amount of pseudo-random data then we can conceivably use it for some cryptographic applications where we might use a shared one-time pad (see for example [125]). Later, in Section 3.4.4, we will discuss the issues surrounding this, and a more suitable way of combining hash functions to generate random numbers.

The property that any x_i can be calculated from any x_j if $i \geq j$ makes some applications more robust. For example, if we have a noisy channel and are trying to authenticate with Lamport's password authentication scheme then the receiver might receive a corrupted password (which it would rightly disregard). However this value may still have been intercepted (before corruption) by an eavesdropper, and so the next value from the hash chain should be sent. The receiver can still authenticate this value by applying the hash function twice.

3.1.4 Further properties of hash chains

We will now look at preimage resistance, second preimage resistance, collision resistance and pseudo-randomness in hash chains.

3.1.4.1 Preimage resistance

If there exists an algorithm A which takes as input $f(x)$ (for arbitrary x) and outputs a preimage x' in time t then there exists a trivial extension to this algorithm which finds a preimage for f^n in time $t \cdot n$.

Algorithm 1:
Description: An algorithm which takes as input $f^n(x)$, and uses a preimage finding algorithm A to find an nth preimage in time $t \cdot n$.

FIND_NTH_PREIMAGE(U)
(1) $x_n = f^n(x)$
(2) **for** $i = n$ **to** 1 **step** -1
(3) $x_{i-1} = A(x_i)$
(4) **return** x_0

This provides an upper bound, but it may be possible to invert f^n more efficiently than this.

As an extreme example, consider a one-way permutation f. For any permutation $f : S \to S$ there exists a period p such that for all $i \geq 0$ and all $x \in S$, $f^{i+p}(x) = f^i(x)$. Suppose the period of our function is n. Now consider the (very long) hash chain $\{x, \ldots, f^{n-1}(x)\}$. We can find x from $f^{n-1}(x)$ by simply applying f to it ($f^n(x) = x$), which is much easier than inverting $n - 1$ times.

3.1.4.2 Second preimage resistance

If there exists an algorithm B taking as input x, and producing a second preimage $x' \neq x$ for f in time t, then the same algorithm will also find a second preimage for f^n in approximately time t (under the assumption that evaluation of f^n can be done as efficiently as evaluation of f). We now explain this claim.

To find a second preimage of x under f^n we use B to find $x' = B(x)$, a second preimage for f. We know that $x' \neq x$, but also we have that

$$f^n(x) = f^{n-1}(f(x)) = f^{n-1}(f(x')) = f^n(x'),$$

and so x' is a second preimage for x under f^n.

The most efficient algorithm for finding a second preimage under f may

not be the most efficient algorithm for finding a second preimage under f^n, but it provides us with an upper bound on the 'second preimage resistance' of f^n.

In fact it may be easier to find a second preimage for the hash chain as we can pick any start value not on the chain and hash it repetitively, hoping to get any of the values on the chain. If we collide with some point on the chain then we will stay on the chain, and so one of the values we have already considered will be a second preimage under f^n (under the assumption that we do not coincide with the original hash chain at a point $f^i(x)$ until at least i hash applications).

3.1.4.3 Collision resistance

Finding a collision under chained hash functions is similar to finding a second preimage. If we know a collision for f then we also know a collision for f^n, but collisions for f^n are not necessarily collisions for f, so intuitively finding a collision for a hash chain should be easier.

3.1.4.4 Pseudo-randomness

As was discussed in Section 2.4, we have not given a precise definition of what it means for a hash function to be pseudo-random. It seems intuitive that if we were not 'expecting' $f(x)$ then we also will not be 'expecting' $f^2(x)$ or $f^n(x)$ but, by considering a one-way permutation again, we can see that if we have a very long chain with n equal to the period of our hash function then we may not be able to guess anything about $f(x)$ from x, but we will know the value of $f^n(x)$ for any value of x.

3.1.4.5 Random mapping properties

In the above discussion we looked at various properties which we may want f^n to have, and whether properties of f provide these properties. One property that it is possible to overlook is that hash functions are often designed to map arbitrary length strings to finite strings, and so fail to be bijective by design. Consequently it is quite possible that the range of f^2 is smaller than

the range of f. In fact it is possible (if very unlikely) that for some i the range of f^i is only one value. This would obviously make any hash chains formed from f useless if they needed to be longer than i values long.

It is also certain that cycles (possibly only cycles of length 1) will exist in the digraph formed by the application of f to the range of f (every vertex x has a directed edge leading to vertex $f(x)$). We do not need to worry if these cycles are very long compared to the length of the hash chain, but if the cycle is shorter than the length of the chain then we may get repetition of values in our hash chain, which would be a problem for many applications (for example Lamport's password scheme).

There has been much research looking into properties of chained functions [40, 109, 112]. In [40] Flajolet and Odlyzko prove that for a random mapping f with range 2^l, the size of the image after k applications of f is approximately $(1 - \tau_k) \times 2^l$, where $\tau_0 = 0$ and $\tau_{i+1} = e^{\tau_i - 1}$. They also prove that the average cycle length will be approximately $\sqrt{\pi 2^l/8}$, with the expected number of cycles of length r equal to $1/r$.

These results show that when working with a hash chain formed from an l-bit hash function, the resulting security is likely to be less than l bits. However if 2^l is large compared to n then the security will not be drastically affected. Thus by assuming $2^l >> n$ for the rest of this thesis we can assume that a hash chain formed with a preimage-resistant hash function f (respectively second preimage resistant, collision-resistant) will be approximately preimage resistant (respectively second preimage resistant, collision resistant) at each step, as well as f^n being approximately preimage resistant (respectively second preimage resistant, collision resistant).

Further study of the properties of random mappings falls outside the scope of this thesis.

3.1.5 Infinite-length hash chains

While hash chains require very little storage, a disadvantage for many situations, such as Lamport's password authentication scheme (see Section 3.1.1),

is that a hash chain is finite[1] and thus any application eventually requires the hash chain to be replaced by a new chain.

'Infinite-length hash chains' [9] (or 'Chameleon chains' [32, 33]) use a trapdoor one-way function, such as exponentiation modulo a composite number, instead of the hash function. This allows traversal in one direction along the chain by anyone who knows the composite (public key), but only people who know the factorisation (private key) can easily compute values in the other direction.

Although this approach may be useful in some applications, we will not consider it further, as trapdoor one-way functions are significantly slower than hash functions, and speed is one of the key reasons for using hash chains.

3.2 Hash trees

We can use the compression property of a hash function to combine two (or more) input values to produce a single output value. This observation invites the idea of having values at the nodes of a tree, and hashing along the edges towards the root. We explain why this might be useful in Section 3.2.4.

3.2.1 Basic graph theory definitions

In this section we define some basic terminology from graph theory.

A *graph* $\mathcal{G}$ is made up of a set of *vertices* $\mathcal{V}$ and *edges* connecting pairs of vertices together. The set of edges $\mathcal{E}$ is contained in the set of unordered vertex pairs and an individual edge between vertices v and w is denoted vw (or equivalently wv).

In our definition we do not allow edges for which both ends are the same (*loops*), or two edges which share the same pair of end points. If there is an edge between two vertices we say the vertices are *neighbours*. The *degree* of a vertex is the number of neighbours it has.

[1] Hash chains have finite length as the end value must be computed before the rest of the values can be used.

A *path* is a list of vertices $\mathcal{P} = v_0, \ldots, v_n$ such that no vertex appears more than once on the list, and such that any two consecutive vertices on the list are neighbours. The *length* of a path containing $n+1$ vertices is n (the number of edges between consecutive vertices on the path). If there is a path with start vertex v and end vertex w, then we say that v and w are *connected*. If there is no path starting at v and ending at w then we say they are *disconnected*. A graph is *connected* if all vertices are pairwise connected.

A *cycle* is a path (with 3 or more vertices) whose last vertex is a neighbour of the first. The length of a cycle containing n vertices is n (the number of edges traversed when following the cycle).

A *tree* is a connected graph with no cycles. A *leaf* of the tree is any vertex with degree one. A *rooted tree* is a tree with one special vertex labelled as the *root* vertex. A *leaf* of a rooted tree is any vertex with degree one, with the possible exception of the root, which is never a leaf.

The *parent* of a vertex v in a rooted tree is the neighbouring vertex on the unique path from v to the root. The root has no parent. Vertex v is the *child* of another vertex w if w is the parent of v. The leaves of a rooted tree have no children. A vertex in a rooted tree may have several children, but will have exactly one parent (with the exception of the root).

The *siblings* of a vertex v in a rooted tree are the other children of the parent of v. Obviously the root has no siblings. Note that it is not necessary that the siblings of a leaf are also leaves. We define the *sibling set* of a path $\mathcal{P}$ to be the set of all vertices that have a sibling in $\mathcal{P}$.

If a path exists from the root to a vertex w through a vertex v then we say that w is a *descendant* of v, and that v is an *ancestor* of w.

In a rooted tree, the *height* $h(v)$ of a vertex $v \in \mathcal{V}$ is the length of the (unique) path connecting it to the root. The height of a rooted tree is the maximum height of a vertex in it.

A *binary tree* is a rooted tree such that any vertex has at most two children. An *a-ary tree* is a rooted tree such that any vertex has at most a children.

Note that a 1-ary tree is a path, and a 2-ary tree is a binary tree.

A *proper* a-ary tree is a tree where all vertices have either 0 or a children.

A *perfect* a-ary tree is a proper a-ary tree such that the height of all leaves is the same.

We will introduce a new definition of our own that will be useful later.

Definition 3.2. An **almost perfect** a-ary tree is an a-ary tree with the following two properties.

- Any vertex that is not the root or a leaf has exactly $a - 1$ siblings.
- The difference in height between any two leaves is at most 1.

Note that any perfect tree is also almost perfect.

3.2.2 Definition of a hash tree

Definition 3.3. A **hash tree** is a rooted tree on an ordered vertex set $\mathcal{V}$, with a value x_v associated with each vertex $v \in \mathcal{V}$, computed using a hash function f. The value associated with each vertex depends on its position in the tree, and is given by:

$$x_v = \begin{cases} f(x_{c_0}||\ldots||x_{c_{r-1}}) & \text{if } v \text{ has } r > 0 \text{ children } (c_0, \ldots, c_{r-1}) \\ \text{Any valid input} & \text{if } v \text{ is a leaf.} \end{cases}$$

In other words, to make a hash tree from a rooted tree we choose values for each leaf, and for every vertex v with children $w_0, \ldots, w_{d-1}$ we set $x_v = f(x_{w_0}||\ldots||x_{w_{d-1}})$. It is important that the vertices have some fixed ordering, otherwise changing the order of the children will alter the value x_v. An example of a hash tree is given in Figure 3.1 to help illustrate some of the terminology.

We name hash trees by the underlying tree type; for example, a *binary hash tree* is a hash tree on a binary tree.

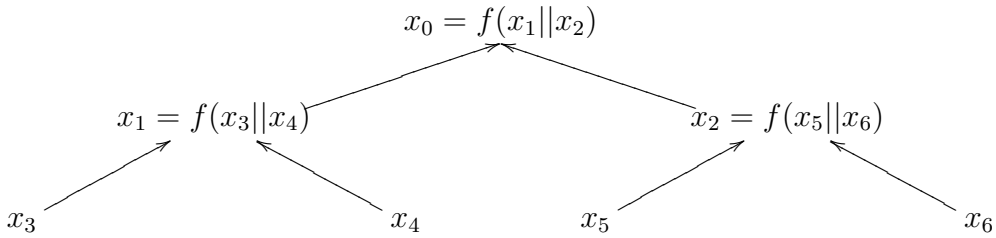


Figure 3.1: A perfect binary hash tree with height 2.

3.2.3 Merkle trees

In this section we look at a particular type of hash tree which we will refer to as a *Merkle tree*.

Definition 3.4. A **Merkle tree** is a hash tree where every leaf has no sibling.

Many authors use the term Merkle tree to refer to any hash tree, but in this thesis it refers to the class of hash trees in Definition 3.4. We note that a Merkle tree as we have defined it here corresponds more closely to the construction given by Merkle in [85], than our definition of a hash tree.

Although Merkle trees are clearly a type of hash tree, it can be useful to think of a Merkle tree as being formed by taking any hash tree, and hashing the leaf values before using them in the hash tree.

The reason for the preliminary hashing in a Merkle Tree is usually to ensure that the data going into the tree is of a sensible size. If each of the leaves represents a whole file of data, then it makes sense to hash the file before trying to do anything more complicated with it. As we will see later (Section 5.2.3.2), it is sometimes also useful to have the leaves of the hash tree 'protected' by an extra hash.

We will name types of Merkle Trees by the underlying hash tree. For example the Merkle tree in Figure 3.2 is a perfect binary Merkle tree, because it is the extension of a perfect binary hash tree (see Figure 3.1) to a Merkle tree. We will refer to the *height* of a Merkle tree as the height of the underlying hash tree. Consequently the Merkle tree in Figure 3.2 has height 2, even though it can also be seen as a hash tree with height 3.

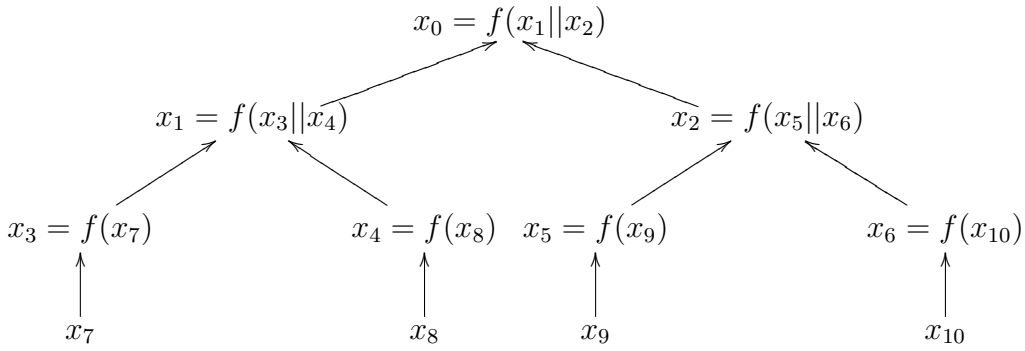


Figure 3.2: A perfect binary Merkle tree with height 2.

3.2.4 Motivation

To understand our motivation for the use of hash trees, consider how they can be used to help with version management of computer files. This scheme is based on Merkle's original use of a hash tree to verify the public keys of instances of a one-time signature scheme [85].

A set of files are grouped into folders, and the folders may also be grouped into other folders, in such a way that everything is ultimately contained in a top level folder. The folders form a rooted tree structure with the root vertex represented by the top level folder, and the leaves represented by the files.

A hash tree is then formed from the rooted tree, using the hash of each file as the value at each leaf, and calculating the values at the other vertices from the leaves.

The root value is published in some respected location, and then all users can check that they have the same set of files in folders as they are intended to have (for example for a program to execute correctly). If a user has a corrupted file then the value they obtain at the root of their hash tree will be different from the published value, and they will know they need to replace at least one file. If some of the other values are published then a user can narrow down further which file is corrupted, and so will have to obtain replacements for fewer files.

It is of course possible to publish the hash of each file, but this may not be appropriate if we want to minimise the amount of published data, or if

obtaining published data is costly for the user.

We will explore further this scheme in Section 5.1.1, including how this scheme is particularly useful when we consider version management.

3.3 Generalising hash function structures

3.3.1 Directed acyclic graphs and hash DAGs

Hash chains and hash trees can both be generalised by a structure we will call a "hash DAG", based on a directed acyclic graph.

Definition 3.5. A **directed acyclic graph (DAG)** is a graph in which every edge has a direction associated with it, and which contains no cycle conforming to the directions of the edges.

A **source** of a DAG is a vertex which no edge leads to.

A **sink** of a DAG is a vertex from which no edge leaves.

It can easily be proved that every (finite) DAG contains at least one source and one sink.

To maintain consistency with hash trees, we define a vertex v to be a *child* of a vertex w if there is a directed edge from v to w, and w to be a *parent* of v if v is a child of w.

Definition 3.6. A **hash DAG** is a DAG on an ordered vertex set $\mathcal{V}$, with a value x_v associated with each vertex $v \in \mathcal{V}$, computed using a hash function f. The value associated with each vertex depends on its position in the tree and is given by:

$$x_v = \begin{cases} f(x_{c_0}||\ldots||x_{c_{r-1}}) & \text{if } v \text{ has } r > 0 \text{ children } \{c_0, \ldots, c_{r-1}\} \\ \text{Any valid input} & \text{if } v \text{ has no children.} \end{cases}$$

3.3.2 Vertices with the same set of children

If two vertices in a hash DAG have the same set of children then they will also have the same value. In many applications we will wish to ensure that the

values at all vertices are different (or at least that they are not predictably the same value).

There are a number of ways to approach this issue, the simplest of which is to treat all vertices which share the same set of children in an identical way. If we wish vertices to have different values then we should simply design the directed acyclic graph so that they do not share all the same children.

An alternative method is to use an *input variable* y_v for each vertex v (except the leaves). By this we mean hashing the concatenation of the values at the child vertices with a (parent) vertex dependent input value.

There are different ways to define the input variables, such as generating them at random.

In the next section we will look at *generalised hash DAGs*, which is inclusive of input variable schemes. The remainder of the hash structures in this chapter are types of *generalised hash DAG*.

3.3.3 A generalised hash DAG

The next structure generalises a hash DAG by allowing a choice of hash function for each parent vertex. This allows vertices that share the same set of children to have values that cannot easily be derived from each other. Although this structure has been used in the literature (for example [10]), we believe this to be the first formal definition.

Definition 3.7. A **generalised hash DAG** is a DAG on an ordered vertex set $\mathcal{V}$, with a value x_v associated with each vertex $v \in \mathcal{V}$, computed using a family of hash functions f_*. The value associated with each vertex depends on its position in the tree and is given by:

$$x_v = \begin{cases} f_v(x_{c_0}||\ldots||x_{c_{r-1}}) & \text{if } v \text{ has } r > 0 \text{ children } \{c_0, \ldots, c_{r-1}\} \\ \text{Any valid input} & \text{if } v \text{ has no children.} \end{cases}$$

An example of a generalised hash DAG is given in Figure 3.3.

It is important also to note that the generalised hash DAG is a very general and unwieldy structure. We will prefer to use more restrictive structures whenever possible.

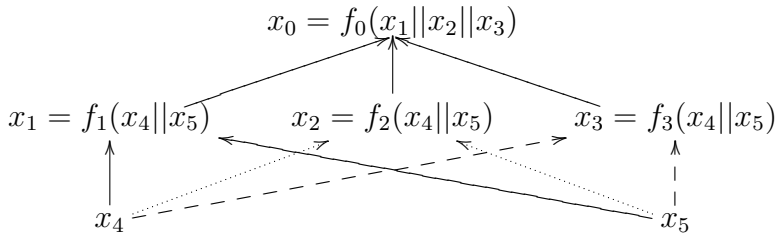


Figure 3.3: An example of a generalised hash DAG. Note that by suitably defining the functions f_1, f_2 and f_3, we can ensure x_1, x_2 and x_3 are independent random variables.

3.4 Useful examples of generalised hash DAGs

We now examine some useful examples of generalised hash DAGs. In particular we look at rainbow chains, hierarchical chain constructions, chained pseudo-random number generators, inverted hash trees, sandwich chains, general hash chains and comb skipchain constructions.

3.4.1 Rainbow chains

Using a different hash function for each link of a hash chain generates a *rainbow chain.*

Definition 3.8. A **rainbow chain** $\mathcal{C}$ is a hash chain which uses a (potentially) different hash function for each link. That is $x_i = f_{i-1}(x_{i-1})$ for $i \in [1, n]$, and x_0 is any valid input.

Figure 3.4 provides an example of a rainbow chain. The term 'rainbow chain' is due to Oechslin [97], and comes from the idea of associating each of the functions with a colour.

$$x_0 \xrightarrow{f_0} x_1 \overset{f_1}{\cdots\!\succ} x_2 - \overset{f_2}{-} \succ \cdots \overset{f_{n-1}}{\rightsquigarrow} x_n$$

Figure 3.4: A rainbow chain of length n.

Rainbow chains have the obvious drawback compared to hash chains that the construction requires knowledge of several hash functions (or a family of

hash functions). However the major advantage when using rainbow chains is the prevention of small cycles.

3.4.2 Joining hash chains

Another useful construction is the idea of using the values of one hash chain to seed other hash chains.

We first consider a simple case with just one new hash chain seeded from value x_i of the original chain (see Figure 3.5). It is clear that the new chain cannot use the same hash function as the original hash chain, otherwise we will end up with the second value of the new chain being x_{i+1}, the third value being x_{i+2}, and so on. Consequently we use different hash functions for the additional chains.

$$\begin{tikzcd}
x_0 \arrow[r, "f"] & \cdots \arrow[r, "f"] & x_i \arrow[r, "f"] \arrow[dr, "f'"] & x_{i+1} \arrow[r, "f"] & \cdots \arrow[r, "f"] & x_n \\
 & & & y_{i+1} \arrow[r, "f'"] & \cdots \arrow[r, "f'"] & y_n
\end{tikzcd}$$

Figure 3.5: A hash chain seeded from another hash chain.

3.4.3 Hierarchical chain construction

Another construction which uses hash chains to seed new hash chains is the *hierarchical chain construction* from [76].

Definition 3.9. A **hierarchical chain construction** with dimension d is a set of hash chains $\mathcal{C}_{0,\emptyset}, \mathcal{C}_{1,\{a_0\}}, \ldots, \mathcal{C}_{d,\{a_0,\ldots,a_{d-1}\}}$, with $a_i \in \{0, \ldots, n_i\}$ where n_i is the length of all chains in dimension i. The chain $\mathcal{C}_{0,\emptyset}$ is seeded from some chosen value, and all the other chains are defined inductively from it. The chain $\mathcal{C}_{l,\{a_0,\ldots,a_{l-1}\}}$ is seeded with the $(a_{l-1})^{th}$ value of the chain $\mathcal{C}_{a_{l-1},\{a_0,\ldots,a_{l-2}\}}$.

We give an example of a hierarchical chain construction in Figure 3.6. In this example the initial seed value is $x_{0,0,0}$, and $x_{i,j,k}$ is defined by

$$x_{i,j,k} = f_2^k(f_1^j(f_0^i(x_{0,0,0}))).$$

To help reinforce our naming scheme, we give some examples of named hash chains below.

Hash chain name	Set of vertices contained in the hash chain
$\mathcal{C}_{0,\emptyset}$	$\{x_{0,0,0}, x_{1,0,0}, x_{2,0,0}, x_{3,0,0}\}$
$\mathcal{C}_{1,\{3\}}$	$\{x_{3,0,0}, x_{3,1,0}, x_{3,2,0}\}$
$\mathcal{C}_{2,\{3,1\}}$	$\{x_{3,1,0}, x_{3,1,1}, x_{3,1,2}, x_{3,1,3}, x_{3,1,4}\}$

We give an application of a hierarchical chain construction in Section 4.4.3.

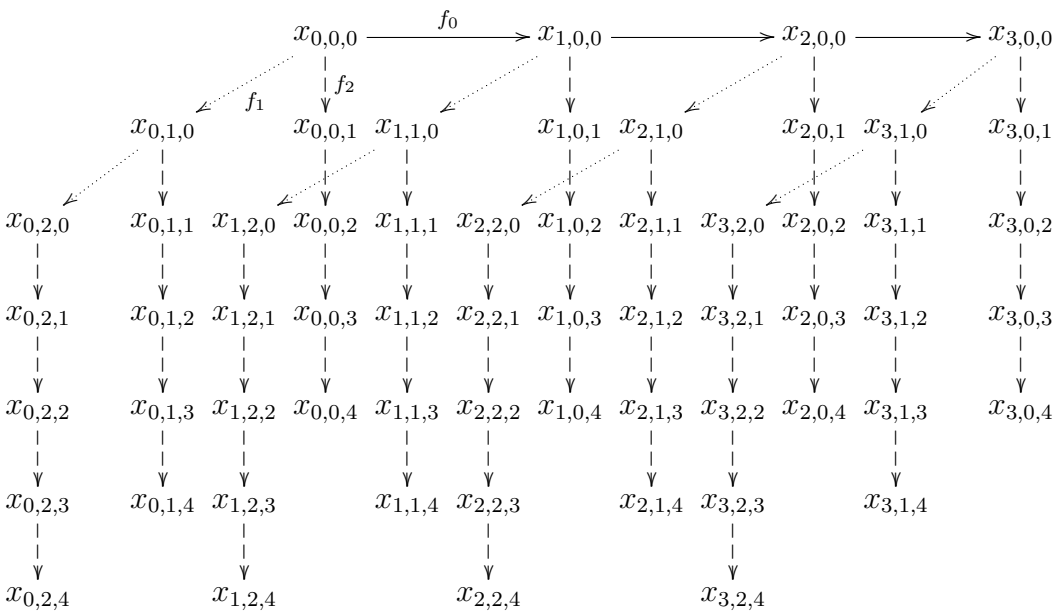


Figure 3.6: A hierarchical chain construction in 3 dimensions and lengths $\{n_0, n_1, n_2\} = \{3, 2, 4\}$.

3.4.4 A chained pseudo-random number generator

A particularly useful example of the hierarchical chain construction is the following construction, which is similar to the construction of the key stream generators in many stream ciphers. The generator is seeded with an initial state, and then the key stream is produced by applying two functions, an update function and output function. We refer to this construction as a *chained pseudo-random number generator* (Figure 3.7).

Definition 3.10. A **chained pseudo-random number generator** is a hierarchical chain construction in 2 dimensions with lengths $\{n, 1\}$.

The values at the ends of the secondary chains can be used as pseudo-random numbers, and are all generated from a single seed.

$$\begin{tikzcd}
x_0 \arrow[r, "f"] \arrow[d, "g"] & x_1 \arrow[r, "f"] \arrow[d, "g"] & x_2 \arrow[r, "f"] \arrow[d, "g"] & \cdots \\
y_0 & y_1 & y_2 & \cdots
\end{tikzcd}$$

Figure 3.7: A chained pseudo-random number generator seeded by x_0, using hash functions f and g.

3.4.5 Inverted hash trees

Definition 3.11. An **inverted hash tree** is a rooted tree on an ordered vertex set $\mathcal{V}$, with a value x_v associated with each vertex $v \in \mathcal{V}$, computed using a family of hash functions f_*. The value associated with each vertex depends on its position in the tree, and is given by:

$$x_v = \begin{cases} f_v(x_p) & \text{if } v \text{ has a parent } p; \\ \text{Any valid input} & \text{if } v \text{ is the root.} \end{cases}$$

Inverted hash trees exist for all types of trees, and so it may be useful to consider the same trees we looked at in Section 3.2.1. For example, Figure 3.8 shows a perfect binary inverted hash tree.

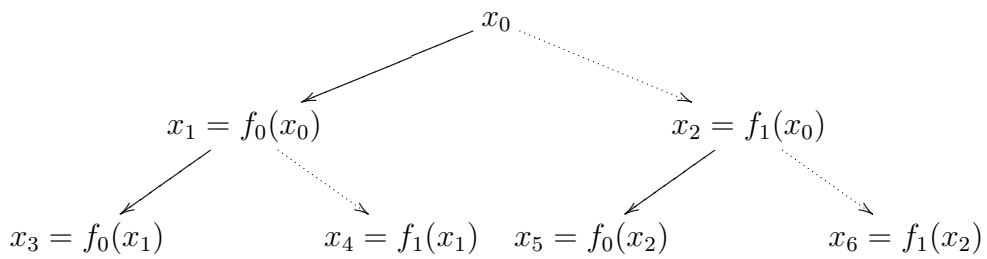


Figure 3.8: A perfect binary inverted hash tree with height 2.

One type of tree which will be of particular interest when considering inverted hash trees is the tree of depth one. This is used in [70] as part of a key establishment scheme. The authors refer to the tree as a *star-like tree*. An example is shown in Figure 3.9 to illustrate this name.

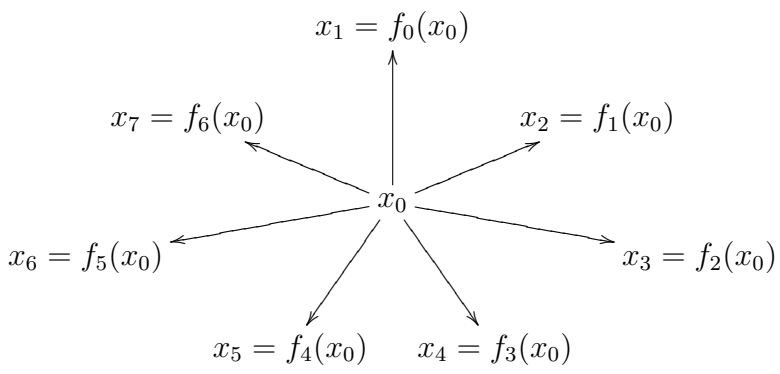


Figure 3.9: A *star-like* inverted hash tree with seven leaves.

It can also be seen that a hierarchical chain construction is in fact an inverted hash tree with one main branch, sub-branches from that, and so on.

3.4.6 Sandwich chain construction

Another idea to connect hash trees together is the sandwich chain construction from [55]. Here we can imagine a main hash chain $\mathcal{C} = \{x_{0,0}, \ldots, x_{n+1,0}\}$ with secondary hash chains $\mathcal{C}_{x_i}$ seeded from each value $x_{i,0}$ of the main chain. The secondary chains use a different hash function from the main chain's hash function, instead being selected from a family of hash functions g_*. The secondary chains are tied together at the end by a connected set of values $\{x_{-1,r}, x_{0,r}, \ldots, x_{n,r}\}$. Figure 3.10 may help when trying to envisage this, and a more rigorous definition follows.

Definition 3.12. A **sandwich chain** of length n and depth r, is a set of values $\{x_{i,j}\}$ defined as follows.

$$x_{i,j} = \begin{cases} \text{Any valid input} & \text{if } i = 0, j = 0 \text{ or } i = -1, j = r \\ f(x_{i-1,j}) & \text{if } j = 0, i \in [1, n+1] \\ g_{x_{i+1,0}}(x_{i,j-1}) & \text{if } j \in [1, r-1], i \in [0, n] \\ f(x_{i-1,j}||x_{i,j-1}||x_{i+1,0}) & \text{if } j = r, i \in [0, n] \end{cases}$$

We will see an example of an application of a sandwich chain in Section 4.5.1.

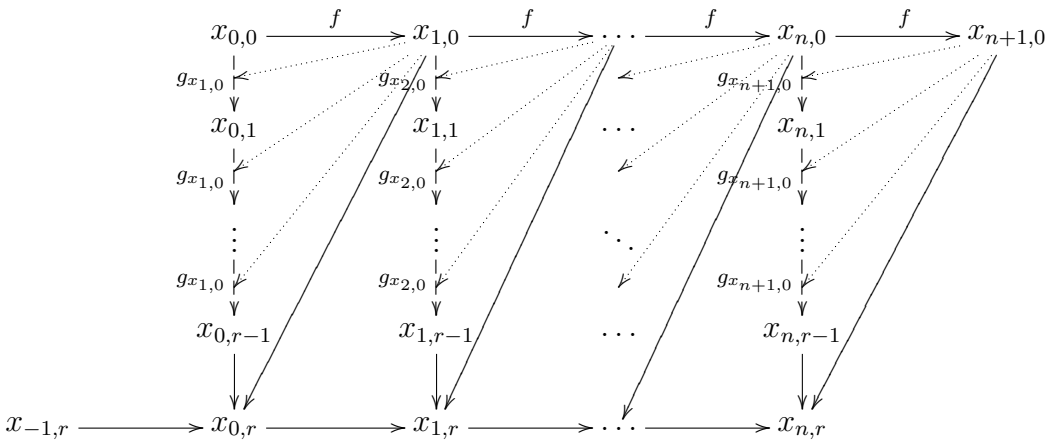


Figure 3.10: A sandwich chain.

3.4.7 General hash chain

In [16], Bradford and Gavrylyako create a general hash chain. They observe that each value in the chain could be dependent on any number of its predecessors, and a chain would still be created.

Definition 3.13. A **general hash chain** of length n and depth d is a set of values $\{x_0, \ldots, x_n\}$ such that $x_i = f(x_{i-d}||x_{i-d+1}||\ldots||x_{i-1})$ for $i \in [d, n]$ and $x_0, \ldots, x_{d-1}$ are valid inputs for f, for some hash function f.

We will see an application of the general hash chain in Section 4.5.3.

3.4.8 Hash chain with breakpoints

In [42], Goyal creates a structure based on a hash chain. They define every d^{th} chain value to be dependent on an additional value, as well as the previous chain value.

Definition 3.14. A **hash chain with breakpoints** of length rd with breakpoints distance d apart, is a set of values $s \cup \{x_0, \ldots, x_{rd}\} \cup \{b_0, \ldots, b_r\}$ computed using a family of hash functions f_*, such that:

$$\begin{aligned} s &= \text{Any valid input;} \\ b_i &= f_i(s); \\ x_i &= \begin{cases} f_0(s||b_0) & \text{if } i = 0; \\ f_0(x_{i-1}||b_{i/d}) & \text{if } i \text{ is divisible by } d; \\ f_0(x_{i-1}) & \text{otherwise.} \end{cases} \end{aligned}$$

We will see an application of this construction in Section 4.5.4.

3.4.9 Comb skipchain construction

Our final hash structure in this chapter is the *comb skipchain construction* from [55]. The construction is a combination of a hash chain and a *one-time signature scheme.*

Definition 3.15. A **digital signature** is a data string which associates a message (in digital form) with some originating entity.

A **digital signature generation algorithm** (*signature generation algorithm*, or *signing algorithm*) is a method for producing a digital signature.

A **digital signature verification algorithm** (or *verification algorithm*) is a method for verifying that a digital signature is authentic (i.e. was indeed created by the specified entity). A **digital signature scheme** consists of a signature generation algorithm and an associated verification algorithm.

A **one-time signature scheme** is a signature scheme that can sign at most one message without risk of forgery.

We will discuss one-time signature schemes more in Section 5.2.

Definition 3.16. A **comb skipchain construction** is a one-time signature scheme with a single public key p_0, together with a hash chain seeded from p_0.

A comb skipchain construction is shown in Figure 3.11. We will see a use of the comb skipchain in Section 4.5.2.

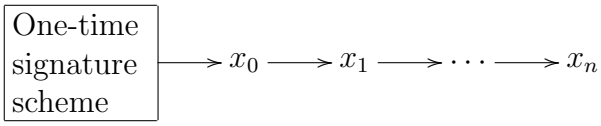


Figure 3.11: A comb skipchain construction.

3.5 Conclusion

In this chapter we defined a number of structures which will be used in later chapters. These included hash chains, hash trees, hash DAGs, generalised hash DAGs, hierarchical chain constructions, inverted hash trees and sandwich chains. We believe this chapter to be the first such catalogue in the literature, and it will prove very useful in later chapters.

Chapter 4

Entity authentication

In this chapter and the next we investigate how the hash structures from Chapter 3 can be used to study issues of authentication, and compare these to solutions based on other cryptographic primitives.

In Section 4.1 we begin by giving a brief overview of authentication, and then focus on entity authentication. In Section 4.2 we look at hash based one-time entity authentication schemes. We study unlimited-time entity authentication schemes in Section 4.3, which are initialised over an expensive secure channel. In Section 4.4 we then study a large number of n-time entity authentication schemes, based on hash structures, and which do not require a secure channel. In Section 4.5 we present an assortment of ideas that improve on the schemes in the earlier sections. Finally in Section 4.6 we compare the schemes from this chapter.

4.1 Introduction to authentication

In this section we briefly distinguish between message and entity authentication, before introducing the notions of channels and adversaries. We also define a property of an entity authentication scheme which we have not seen defined elsewhere. We end the section by identifying the costs that will be used to assess the schemes in the rest of the chapter.

We note that there exists a large amount of research into the field of authentication [15, 28, 31, 83].

4.1.1 Types of authentication

We now summarise the types of authentication as described by Menezes et al. in [83].

Authentication can be divided into two types, entity authentication and message authentication. The aim of *message authentication* is to provide assurance that a message has not been tampered with (either accidentally or maliciously) and to provide evidence of the origin of that message. A recipient has assurance of the *data origin* of a message if they are sure of the source it. Message authentication includes a data integrity check, as authenticating the origin of a message without being sure of its contents is nonsensical.

Entity authentication provides evidence of the identity of a person (or another entity such as a computer or computer program). This proof of identity normally requires a freshness mechanism, such as a time stamp, as entity authentication is used to demonstrate that an entity is present at a particular point in time [65, 66].

The aim of this chapter is to consider the application of the hash structures from Chapter 3 to entity authentication. In Chapter 5 we will consider the application of the same set of hash structures to message authentication (and in particular to signatures).

4.1.2 Public verifiability

The following property of an entity authentication scheme does not appear to have been defined elsewhere. We will later exhibit some schemes that have this property.

Definition 4.1. An entity authentication scheme has **public verifiability** if the verifier can later prove to any third party that the authentication was valid.

Public verifiability of an entity authentication scheme is similar to non-repudiation of a message authentication scheme. If a message authentication scheme has non-repudiation, then it is possible for anyone to identify the

author. Therefore it is possible to prove to any third party that the message authentication is valid.

A real-world example that requires public verifiability is a long-distance race, where each participant is required to pass through a series of checkpoints. If the participants authenticate to the checkpoint officials using a scheme with public verifiability then, once the race is over, the officials can prove to the race judge (or anyone else) the identities of the participants that they met. If the officials also authenticate to the participants using the same scheme then the participants can also prove that they went through each checkpoint.

4.1.3 Channels

During an entity authentication attempt, an *authenticator* A aims to convince a *verifier* B of their identity. For some systems we may wish to have a party T who is trusted by all the other parties.

They will be communicating with each other through *channels* with different properties, where by a channel we mean any method of communicating between two entities, such as a telephone wire, an optic cable or electromagnetic waves. When information is made public we denote it as being sent to some public location P.

In the schemes in this chapter we will make use of the following types of channel (summarised in Table 4.1).

Description	Notation used
A uses an unprotected channel	$A \rightarrow B : m$
A uses an authenticated channel	$A \xrightarrow{A} B : m$
A uses a secured channel	$A \xrightarrow{S} B : m$
A publishes m with no authentication	$A \rightarrow P : m$
A publishes m in an authenticated way	$A \xrightarrow{A} P : m$

Table 4.1: Notation for different methods of passing information.

4.1.3.1 Unprotected channel

The simplest way in which A can send a message m to B is via an unprotected channel. We represent this communication as $A \rightarrow B : m$.

4.1.3.2 Authenticated channel

In order to provide authentication most protocols require an initial authentication process upon which later authentications can be based. Typically this initial process is costly, slow or inconvenient, and so using it each time authentication is required is not feasible. We write $A \xrightarrow{A} B : m$ to mean that A sent m to B and B has assurance of the data origin[1] and freshness[2] on this channel. We refer to this channel as an *authenticated channel*.

4.1.3.3 Secure channel

We will describe a channel as a *secure channel* if it is an authenticated channel, with the additional property that messages sent through it maintain confidentiality.

We remark that a channel that is protected against eavesdroppers, but which is not also an authenticated channel, is open to certain attacks. As an example consider a bank which encrypts payment instructions in the form $E_k(a), E_k(b), E_k(c)$ where a and b are the account numbers which are to be debited and credited, and c is a number representing the amount of money to be transferred. The bank is using a confidential channel because any eavesdropper who does not know k cannot work out a, b or c. However if an adversary deposits a cheque for a small amount addressed to them, and then changes the bank's message from $E_k(a), E_k(b), E_k(c)$ to $E_k(a), E_k(b), r$, with r chosen at random, then they have a good chance of increasing the amount their account receives (as it is likely that $E_k^{-1}(r)$ is quite large).

[1] As mentioned in 4.1.1 we need assurance of data integrity to have assurance of data origin. Here we use data integrity to include reordering and deletion of parts of the message.

[2] We need freshness here as we will be using this channel for initialisation. We will always wish to avoid an attack in which an adversary can reuse an old initialisation message, along with the recorded authentication values. See [66] for a more detailed discussion.

If the channel was a true secure channel, the bank would notice that either the data origin or the message integrity have been violated, and so would not process the message in the usual way.

We assume that use of a secure channel is relatively expensive and so will usually only be used for initial key agreements. To show that a channel is secure we write $A \xrightarrow{S} B : m$.

4.1.3.4 Unprotected publishing

If an entity A wishes to send a value to a large number of users, as in the case of broadcast, then they can *publish* it, which we denote by $A \rightarrow P : m$. We will assume that there is an agreed means of retrieval of a message sent in this way. We assume that the 'cost' of publishing is about the same as sending the message over an unprotected channel.

Although we have not differentiated here between publishing a message and broadcasting it, the two have slightly different properties. If a message is broadcast then typically only those users 'listening' at the time will receive it. If a message is published then no user receives it immediately; instead they must retrieve the message from a published location.

4.1.3.5 Authenticated publishing

In most cases the sender A of a published message m will want to allow recipients to be able to verify the origin of m. We will assume that they can guarantee that the data will be quickly available to anyone that wants it.[3] This type of authenticated publishing will be denoted as $A \xrightarrow{A} P : m$. We will make the reasonable assumption that authenticated publishing is more 'costly' than publishing.

[3]We note that for some applications this could be very costly.

4.1.4 Adversarial models

With respect to entity authentication, the main aim of an adversary E is to persuade the verifier B that they (E) are an authorised entity (for example A).

An adversary may be satisfied if they can simply prevent two entities A and B from carrying out entity authentication successfully, and depending on the adversary's capabilities this may be very easy. One attack which may be possible, depending on B's capabilities, is a denial of service (DoS) attack, whereby the adversary sends so many false messages to B that either A's messages do not get received or B is so busy processing spoof messages that B takes a prohibitively long time to process A's request. Denial of service attacks fall outside the scope of this research.

We will look at two types of adversary:

Definition 4.2. An **active adversary** E_A can eavesdrop on unprotected and authenticated channels, and can send messages on unprotected channels.

A special type of active adversary is the *record and replay adversary* that records a message sent along an unprotected channel and replays it at a later time.

Definition 4.3. A **man-in-the-middle adversary** E_M intercepts all messages along all channels. It can edit, block or create new messages along unprotected channels. It can block messages along authenticated and secure channels. It cannot read messages that are sent along secure channels, but it can read all other messages.

A man-in-the-middle adversary is always able to attack a system by not passing any communications between the authenticating and verifying parties. We assume that the man-in-the-middle adversary has a secondary objective to avoid being discovered. If the adversary fails to pass any information along the channels to which they have gained access then the authenticating parties may resort to setting up new channels to use, which might not benefit the adversary.

We do not consider attacks in which a man-in-the-middle adversary edits messages on authenticated channels (or secure channels), as these changes would be detected by the receiver, and the spoof messages would be rejected.

Two other common types of attack are as follows (the adversaries capable of performing each attack are given in brackets).

- **Record and replay** (E_A and E_M): The adversary records an authentication message from one session, and replays it to the verifier during a later session.

- **Intercept and replay** (E_M): The adversary intercepts an authentication message from one session, preventing the verifier receiving it. They then use the message to falsely authenticate to the verifier during a later session.

We note that for both attacks there is no dependence on the presence of the authenticator.

4.1.5 Entity authentication phases

The use of hash functions and one-way functions to build password based entity authentication schemes was first proposed in the 1970s [36, 69, 153]. In the next section we will examine how hash functions, and in particular hash structures, can be applied to entity authentication.

All our schemes can be viewed as a combination of at most three phases. Firstly, the *initialisation phase* allows entities to create and share secret information and commitments. During the *entity authentication phase* the verifier validates the identity of the authenticator. Finally, some schemes have an additional phase in which the verifier proves to a third party that they have had contact from the authenticator.

4.1.6 Associated costs

Each of the schemes that we look at in this chapter has associated costs. We will now briefly discuss some of these.

1. Complexity - Possibly the most obvious cost associated with any protocol is the amount of computation needed. In many authentication protocols this is best approximated by the amount of computation required to perform modular exponentiation. Since we are interested in light-weight hash-based schemes where we hash short 'messages', a more appropriate measure of computational complexity is the number of hash function evaluations required per phase of the entity authentication protocol.

2. Communication cost - Another cost is the number of values that need to be sent during each phase. We will mostly be interested in the number of values sent during the entity authentication phase.

3. Channel type - As discussed in Section 4.1.3, different types of communication channel have different associated costs.

4. Security - Although security is often thought of as a property of the scheme, it can also be seen as cost. Some schemes may be very efficient in terms of complexity and communication costs, but these may come at the expense of weaker security.

4.2 One-time entity authentication schemes

In this section we present several *one-time entity authentication schemes*. They can be used only once before they become vulnerable to record and replay adversaries. If we are not worried about record and replay adversaries then they can be used many times.

Definition 4.4. A **one-time** entity authentication scheme is an entity authentication scheme which can only be securely used once before another initialisation phase must be executed.

All of the schemes in this section can be applied to the scenario of an entity A using a password to log onto a computer network. One potential problem with password dependent schemes is that, if A is a human, then A will tend to choose passwords fairly predictably. For the purposes of this chapter we will assume passwords and other secrets to be chosen in a secure manner, for example by a pseudo-random number generator.

In the rest of this section we present four schemes which are all 'one-time'. We end the section by comparing the schemes and exploring the strengths and weaknesses of each.

4.2.1 Trivial one-time password entity authentication

Our first example is a very simple authentication scheme based on the idea of a password. In the *trivial one-time password entity authentication scheme*, the authenticator creates a shared secret value with the verifier. As proof of identity, the authenticator reveals the secret value. Authentication based on passwords that are only used once is a well known principle in cryptography [28, 83]. We include this scheme as it forms a benchmark against which other entity authentication schemes can be compared. The trivial one-time password entity authentication scheme also emphasises how powerful a secure channel can be.

Entity Authentication Scheme 4.1: The trivial one-time password entity authentication scheme.

Initialisation

- A chooses a value x at random.[4]
- $A \xrightarrow{S} B : x$

Entity authentication

- $A \rightarrow B : x$
- B verifies that A has sent the correct value.

[4]We note that when choosing secret values randomly they should be chosen uniformly from a suitably large set - for example the set of bit strings of length 256.

Security

- If the authentication phase is allowed to be used more than once without first rerunning the initialisation phase then an active adversary can perform a record and replay attack.
- A man-in-the-middle adversary can perform an intercept and replay attack to masquerade as A.

Since only A and B know the password x, B can be sure that A is the sender.

4.2.2 Basic one-time hash based entity authentication

We now look at how we can use hash functions to improve on Entity Authentication Scheme 4.1.

The *basic one-time hash based entity authentication scheme* has an advantage over Entity Authentication Scheme 4.1. The initialisation phase does not require setting up a shared secret, and therefore only needs an authenticated channel, not a secure channel.

Another advantage of this scheme over Scheme 4.1 is that even if an adversary obtains access to the hard drive of B's computer (and learns $f(x)$) then they still cannot falsely authenticate A to B. The storage of the hash of passwords for this reason is a well established practice [83, 89].

Entity Authentication Scheme 4.2: The basic one-time hash based entity authentication scheme.

Initialisation

- A picks a value x at random, and computes the hash of it using a preimage-resistant hash function f (which is specified as a system parameter).
- $A \xrightarrow{A} B : f(x)$

Entity authentication

- $A \rightarrow B : x$
- B hashes x and checks it is the same as the value A originally sent.

Security

- If the authentication phase is allowed to be used more than once without first rerunning the initialisation phase then an active adversary can perform a record and replay attack.
- A man-in-the-middle adversary can perform an intercept and replay attack to pose as A.

This scheme avoids using a secure channel because it does not matter who knows $f(x)$. It is only important that A is the only entity that knows x.

4.2.3 Traditional password entity authentication

The *traditional password entity authentication scheme* (Scheme 4.3) is used by many applications in computer networks. In the initialisation phase A passes $f(x)$ to B using a secure channel (for example typing it into B's computer). This makes no obvious gain in security over Scheme 4.2, as one of our assumptions is that it is difficult to find x from $f(x)$.

Entity Authentication Scheme 4.3: The traditional password entity authentication scheme.

Initialisation

- A picks a value x at random.
- $A \xrightarrow{S} B : f(x)$

Entity authentication

- $A \rightarrow B : x$
- B hashes x and verifies it is the same as the value A sent in the initialisation phase.

Security

- If the authentication phase is allowed to be used more than once without first rerunning the initialisation phase then an active adversary can perform a record and replay attack.
- A man-in-the-middle adversary can perform an intercept and replay attack to pose as A.

Just as for the previous schemes, a record and replay adversary can record x when it is sent over the unprotected channel and replay it to B in an attempt to impersonate A. This attack will fail as long as each value x is only used once. However since Scheme 4.3 is often used for many sessions without reinitialisation; this constitutes a break of the scheme under the assumption of the existence of an active adversary.

Despite this weakness, this scheme is widely adopted. One possible explanation that this scheme is used in practice more often than Scheme 4.2 is that the use of a secure channel for initialisation gives a false sense of protection.

4.2.4 Basic one-time hash based entity authentication with public verifiability

We now present an extension of Scheme 4.2, which adds public verifiability (see Section 4.1.2) at the expense of authentically publishing the initialisation value. This idea forms the basis of Entity Authentication Scheme 4.10 which we will look at later.

Entity Authentication Scheme 4.4: The basic one-time entity authentication scheme with public verifiability.

Initialisation

- A creates a random key x and computes the hash $f(x)$.
- $A \xrightarrow{A} P : (f(x); A, B)$

Entity authentication

- $A \rightarrow B : x$
- B hashes x and checks it against the value received during the initialisation phase.

Proof of authentication

- B proves to some third entity C that A was authenticated by: $B \rightarrow C : x$.
- C can check against the authenticated public value $f(x)$, which was set up in the initialisation stage.

B has now proved communication with A to C. There are two potential attacks which are relevant to the proof of authentication stage.

1. B may wish to fool C into thinking they know x before A has revealed it. This is clearly flawed, as B would have to compute a preimage of $f(x)$.
2. An adversary may wish to prevent B from communicating x to C. A man-in-the-middle adversary is able to do this but, as discussed in Section 4.1.4, this may result in the adversary being detected.

The remaining security analysis is identical to Entity Authentication Scheme 4.2.

This scheme can only be used once before another initialisation phase is required. If it is used multiple times then, like all the previous schemes, this scheme could be broken by a record and replay attack.

This scheme can be used for any application where A wishes to pass authority to B at a later date. Also A does not have to trust C with x. A hypothetical situation is if A is a well respected entity, paying for a service that B is providing, and C is A's bank. A publishes $(f(x); A, B)$ as a commitment to pay B on completion of a service. If necessary B can check with C

that A has the funds to pay for the service. Once B has provided the service, A will send them x, which B can present to C to receive payment.

We note that in this example there is no guarantee that A will provide B with x at the appropriate time, or even that A knows a preimage of $f(x)$. Instead this scheme relies on A being well respected, and we assume that if A did not behave fairly then with time they would lose the respect attributed to them. Schemes which do not depend on this kind of assumption are outside the scope of this research, but are studied in more detail in [37, 101, 137].

4.2.5 Summary of one-time entity authentication schemes

The elementary one-time entity authentication schemes that we have discussed in this section are summarised in Table 4.2 in terms of:

- Complexity — The number of hash function evaluations required by A in the initialisation phase (Init), and by A and B respectively in the entity authentication phase (Auth).
- Comm. Cost — The communication cost in number of cryptographic values sent by A during the initialisation phase (Init), and the entity authentication phase (Auth). We assume all cryptographic values to have the same number of bits as the output of the hash function used in the scheme.
- Init. Channel — The (most expensive) type of channel used during the initialisation phase: secure channel (S), authenticated channel (A) or authenticated publishing (AP).
- Security E_A — The number of times the scheme can be used securely in the presence of an active adversary.
- Security E_M — The capability of a man-in-the-middle adversary to attack the scheme: E_M can successfully pose as A for the duration of the scheme by intercepting one message (✓); E_M can successfully pose as A n times by intercepting one message (n✓); E_M cannot pose as A (✗).

- Public Verif. — Whether the scheme has public verifiability or not.

Scheme	Complexity		Comm. Cost		Init.	Security		Public
	Init	Auth	Init	Auth	Channel	E_A	E_M	Verif.
4.1	0	$0, 0$	1	1	S	1	✓	
4.2	1	$0, 1$	1	1	A	1	✓	
4.3	1	$0, 1$	1	1	S	1	✓	
4.4	1	$0, 1$	3	1	AP	1	✓	✓

Table 4.2: Summary of the one-time entity authentication schemes.

From Table 4.2 we can see that the scheme with the least computational complexity is Scheme 4.1. The other schemes all require the same effort.

Scheme 4.4 is very similar to Scheme 4.2, the main performance differences being that Scheme 4.4 provides public verifiability but requires three times as much communication during the initialisation phase, as well as access to public storage space.

Schemes 4.2 and 4.4 do not require a secure channel, and so are cheaper to set up than the other schemes.

All the schemes in this section are limited by the fact that they can only be used once before a reinitialisation must take place.

4.3 Simple 'unlimited-time' entity authentication schemes

We begin this section by giving a definition of 'unlimited-time' entity authentication schemes. In Section 4.3.2.1 we look at unlimited-time challenge-response entity authentication schemes. In Section 4.3.2.2 we look at how unlimited-time schemes can be made using clocks or counters. We then present a novel scheme with minimal storage requirements for both parties. Finally we compare the schemes in this section.

4.3.1 Unlimited-time entity authentication

In the remainder of this chapter we look at schemes that can be used multiple times for each initialisation. We distinguish between two types of such scheme: *unlimited-time* entity authentication schemes and *n-time* entity authentication schemes.

Definition 4.5. An **unlimited-time** entity authentication scheme is an entity authentication scheme that can be used an unlimited number of times, until the secret values expire due to vulnerability to brute force attack.

In Section 4.4 we will look at 'n-time' entity authentication schemes, which can be used several times, but which are likely to need re-initialising within the lifetime of the scheme.

Definition 4.6. An ***n*-time** entity authentication scheme is an entity authentication scheme that can be used a finite number of times before it must be reinitialised.

By 'unlimited-time' entity authentication schemes we refer to entity authentication schemes which, for all practical purposes, can be used as many times as is required without repeating the initialisation phase. In theory they can be broken by brute force if left long enough without re-initialising, as an adversary could guess the values transferred in the initialisation phase.

An n-time entity authentication scheme may be secure for something in the region of 2^{10} authentications, depending on the parameters used. This compares to an unlimited-time entity authentication scheme, which may be secure for something more like 2^{40} authentications, again depending on how the scheme is set up and used.

We can convert our trivial one-time password entity authentication scheme (Scheme 4.1) into an unlimited-time scheme by including a proof of freshness. We now give two simple ways in which this could be done.

4.3.2 Traditional approaches

In this section we look at two traditional approaches to unlimited-time entity authentication which are both in widespread use. Challenge-response oper-

ations and timestamps both allow repeated authentication and have been examined in the literature [65, 66].

Throughout this section we apply a strong hash function f to the concatenation of a key k with a message m. There exist attacks on this construction if the underlying hash function has particular structural properties (for example if the Merkle-Damgård construction is used [84]). These attacks led to the creation of other constructions such as HMAC [4] and these should be considered for any practical implementations. However it is beyond the scope of this thesis to consider the structural design of hash functions, so we simply assume that our hash function is not vulnerable to these attacks.

4.3.2.1 Basic challenge-response scheme

Challenge-response forms the basis of many authentication protocols [129, 160]. In one example of this type of scheme, A and B initially establish a shared key x. Later B sends a random challenge text to A, who hashes the challenge text with the key and sends it back to B. The challenge text is a proof of freshness.

Entity Authentication Scheme 4.5: The basic challenge-response scheme.

Initialisation

- The entity A creates a random key x.
- $A \xrightarrow{S} B : x$

Entity authentication

- B creates a random message m. This message should be chosen from a suitable large set in a way that ensures messages are not reused. The potential shortcomings of choosing messages that are too predictable or that could be reused is discussed in detail in [41].
- $B \rightarrow A : m$
- A hashes m together with the key x and sends it back: $A \rightarrow B : f(x||m)$.

The authentication can be repeated an unlimited number of times, which is a substantial improvement on all of the schemes in Section 4.2. It is not vulnerable to record and replay attacks, since B will choose a different m each time, and we assume that finding $f(x||m)$ for a new message m is a hard problem without knowledge of the key x, even after observing several authentication phases.

Even a man-in-the-middle adversary E_M cannot find the key x, so the only way the adversary can create $f(x||m)$ is by passing the challenge message m to A and pretending to be B. However, if A wishes to authenticate and sends $f(x||m)$ to E_M, then the man-in-the-middle adversary has done nothing more than passing the messages on. Since we gave the adversary power to intercept and edit unprotected messages as they wish, this does not show a weakness in the scheme. Rather, it illustrates that the existence of a man-in-the-middle adversary is a very strong assumption.

A disadvantage of this scheme is that the entity authentication stage requires two messages to be sent (all previous schemes only require one). However this scheme easily makes up for this drawback with the increase in security compared with previous schemes. The amount of computation required per authentication is approximately the same as for Scheme 4.2.

A bigger disadvantage of this scheme compared to Schemes 4.1, 4.2 and 4.4 is the reliance on a secure channel for the initialisation phase. For many applications we would prefer to initialise with an authenticated channel as it is far less expensive to set up.

4.3.2.2 Authentication with a counter or a time-dependent variable

Another well studied proof of freshness is to use a timestamp. We look at two very similar schemes which use a counter and a time-dependent variable respectively. If A and B have a synchronised counter then this can be considered as a logical clock [67]. Both entities set their counter value t to zero during the initialisation phase.

Entity Authentication Scheme 4.6: The basic counter based entity authentication scheme.

Initialisation

- The entity A creates a random key x.
- $A \xrightarrow{S} B : x$.
- A and B both create a counter t, which they initialise to 0.

Entity authentication

- A hashes the key with the current counter value t.
- $A \rightarrow B : (f(x||t), t)$.
- If the received value t is less than B's counter then B rejects the authentication attempt. If t is at least as large as B's counter then B continues to the next step.
- B hashes t with the shared secret key x. B accepts A's proof of identity if the computed value matches the value received from A.
- A sets their counter to be $t + 1$.
- If the authentication was successful then B sets their counter to be $t+1$.

The analysis of Scheme 4.6 is very similar to that of Scheme 4.7, which is based on a time-dependent variable.

Entity Authentication Scheme 4.7: The basic time-dependent variable entity authentication scheme.

Initialisation

- The entity A creates a random key x.
- $A \xrightarrow{S} B : x$.

Entity authentication

- A hashes the key with a time-dependent variable t.
- $A \rightarrow B : f(x||t)$.[5]
- B finds the time-dependent variable for when A sent their authentication message, and hashes it with the shared secret key x. B accepts A's proof of identity if the computed value matches the value received from A.

In Schemes 4.6 and 4.7, if the adversary can influence the counters or clocks then the adversary might hope that A would no longer be able to authenticate to B. Neither scheme has specific protection against this type of attack.

In Scheme 4.6, if A's counter gets ahead of B's counter then the scheme still functions well, so a successful *desynchronising attack* by an adversary must get B's counter ahead of A's. One way to do this is for the adversary to somehow authenticate to B, without A knowing.

Any implementation of Scheme 4.7 must address the following three issues, which do not affect Scheme 4.6.

- A and B must have synchronised clocks.
- The clocks may run at different speeds.
- There may be delays between the messages being sent and received.

Despite these problems, both schemes are suitable for many applications and feature in relevant ISO standards [57].

[5]This step could be replaced with $A \rightarrow B : (f(x||t); t)$ as long as B is satisfied with the time-dependent variable being one of a set of values [30, 62]. This relaxes the necessity for accurately synchronised clocks. For example, if A uses the current time to the nearest thousandth of a second then B will probably be satisfied with a variety of different values of t, but will not want to have to try them all to see which one A used.

4.3.3 Storing one less value

Scheme 4.7 is very simple and it is quick to compute each authentication value (only one hash evaluation). It requires storage of at most three values for each entity: x, t and the value $f(x||t)$ at the time it is used.

In this section we present an unlimited-time entity authentication scheme based on the chained pseudo-random number generator (see Figure 4.1), which only requires the storage of at most two values. These two values are x_t and y_t (as it is used). Note that t does not need to be stored.

Entity Authentication Scheme 4.8: The minimum storage unlimited-time entity authentication scheme.

Initialisation

- The entity A creates a random key x_0.
- $A \xrightarrow{S} B : x_0$.

Entity authentication number t

- A and B both share the secret value x_t from the last session (or x_0 if they have only performed the initialisation phase).
- $A \rightarrow B : g(x_t)$.
- B checks the received value against x_t.
- A and B replace their secret stored value x_t with $x_{t+1} = f(x_t)$.

This scheme requires the knowledge of two hash functions, which we denote f and g. They could be based on the same hash function h, for example if we define f and g as follows:

$$f(*) = h(0||*);$$

$$g(*) = h(1||*).$$

This scheme is identical in analysis to Scheme 4.7, except that it only requires the storage of at most two values. In particular it can be seen that

$$\begin{tikzcd} x_0 \arrow[r, "f"] \arrow[d, "g"] & x_1 \arrow[r, "f"] \arrow[d, "g"] & x_2 \arrow[r, "f"] \arrow[d, "g"] & \cdots \\ y_0 & y_1 & y_2 & \cdots \end{tikzcd}$$

Figure 4.1: A chained pseudo-random number generator using hash functions f and g. For the tth authentication A and B can compute the authentication value from the stored value x_{t-1}.

the scheme may suffer if it is likely an authentication attempt may go missing. The scheme can be extended to provide protection against this by B checking ahead upon failed authentications to see if they have been sent a value for a future session. Each additional authentication value they check against will provide more resistance against lost messages, but also reduces the security of the scheme as it is easier to guess one of the valid authentication values.

It can be seen that two values is the minimum storage possible in a environment where we are concerned by the possibility of an active adversary E_A. If A and B only ever have one stored value, then it must be the value that is sent by A to B during the authentication stage. E_A can obtain this value, and then knows as least as much information as A. Consequently E_A will be able to compute subsequent authentication values and masquerade as A to B.

4.3.4 Summary of unlimited-time schemes

In this section we have seen some schemes which can be used to authenticate an unlimited number of times once the initialisation phase has been performed. This is a significant improvement over the schemes we looked at in Section 4.2, which required reinitialisation each time an entity authentication was required.

All the unlimited-time schemes in this section suffer from the drawback that they require the use of a secure channel during the initialisation phase.

In the next section we will focus on schemes that can provide several authentications per initialisation, and which only require an authenticated channel to perform the initialisation.

Scheme	Complexity		Comm. Cost		Init.	Security		Public	Notes
	Init	Auth	Init	Auth	Channel	E_A	E_M	Verif.	
4.5	0	$1,1$	1	$1,1$	S	∞	✗		
4.6	0	$1,1$	1	$2,0$	S	∞	1✓		
4.7	0	$1,1$	1	$1,0$	S	∞	✗		A
4.8	0	$2,2$	1	$1,0$	S	∞	1✓		B

Table 4.3: Summary of the one-time entity authentication schemes. For an explanation of the values and column headings, see Section 4.2.5.
Notes
A: There is the possibility that a man-in-the-middle adversary could perform an intercept and replay attack before the time-dependent value expires. This attack would be of the 1✓ form. This scheme also requires the existence of synchronised clocks.
B: Whereas the other schemes in this section require the storage of at least 3 values, Scheme 4.8 only requires the storage of 2 values.

Table 4.3 gives a summary of the unlimited-time entity authentication schemes presented in this section.

4.4 n-time entity authentication schemes

In this section we look at n-time entity authentication schemes based on structures given in Chapter 3. We have already defined n-time entity authentication schemes in Definition 4.6. We begin by looking at hash chain based entity authentication schemes, then move on to hash tree based schemes, and finally schemes based on the hierarchical chain construction.

4.4.1 n-time hash chain based schemes

All the previously discussed schemes are either limited to one use, or they rely on a secure channel in the initialisation phase. In this section we show how hash chains can be used to create n-time authentication schemes which do not rely on secure channels.

4.4.1.1 Hash chain entity authentication

The first scheme we look at is the hash chain entity authentication scheme due to Lamport [69]. Many variations of this are used in the literature, but they all use essentially the same idea as Scheme 4.9 [7, 44, 47, 48, 49, 52, 54, 71, 105].

Entity Authentication Scheme 4.9: The hash chain entity authentication scheme due to Lamport [69].

Initialisation

- A creates a key x, and uses it to seed a hash chain of length n.
- $A \xrightarrow{A} B : f^n(x)$.
- B stores the next session number i (initially $i = 1$).

Entity authentication number i

- A announces that they intend to authenticate to B.
- B sends A the session number i.
- $A \rightarrow B : f^{n-i}(x)$.
- B checks that the hash of $f^{n-i}(x)$ is equal to the previously received authentication value $f^{n-i+1}(x)$.
- B stores $f^{n-i}(x)$ for use in the next authentication.

Entity A can authenticate themselves to B a total of n times before a new hash chain must be initialised.

In Scheme 4.2 a man-in-the-middle adversary could completely break the scheme by stealing A's authentication value. The same can be done here, but the adversary will only break the scheme for that authentication attempt. For subsequent authentications the adversary must repeat the process.

4.4.1.2 Hash chain entity authentication with public verifiability

In the same way that we added public verifiability to Scheme 4.2, we can add it to the hash chain entity authentication scheme (Scheme 4.9).

Entity Authentication Scheme 4.10: The hash chain entity authentication with public verifiability.

Initialisation

- A creates a key x, and uses it to seed a hash chain of length n.
- $A \xrightarrow{A} P : (f^n(x); A, B)$.

Entity authentication number i

- $A \rightarrow B : f^{n-i}(x)$.
- B checks that the hash of $f^{n-i}(x)$ is equal to the previously received authentication value $f^{n-i+1}(x)$.
- B stores $f^{n-i+1}(x)$ for use during the next authentication.

Proof of authentication number i

- $B \rightarrow C : f^{n-i}(x), i$.
- C checks that the ith hash of $f^{n-i}(x)$ is equal to the published value $f^n(x)$.

Note that C could store values that B sends to them, and then hash just the number of times needed to check against the previous value they received. It is not necessary for B to send C every authentication value. If B needs to give a proof of authentication to a different entity C' then they can easily repeat the proof of authentication phase with C'.

This scheme is vulnerable to loss of synchronisation in a very similar way to Scheme 4.8 and a potential solution is discussed in Section 4.3.3.

4.4.1.3 Entity authentication with many verifying parties

All the schemes we have looked at so far are limited in that they can only cater for one verifying party. The most obvious way to adapt the earlier schemes to many verifying parties is to initialise a separate instance of the scheme with each verifier. We would like to reduce the number of times the initialisation phase has to be run because it uses an expensive authenticated (or secure) channel.

We observe that Scheme 4.9 can be adapted to facilitate many verifiers. We change the initialisation phase so that the hash chain's end value is published over an authenticated channel.

Entity Authentication Scheme 4.11: Hash chain entity authentication scheme for many verifiers.

Initialisation

- A creates a key x, and uses it to seed a hash chain of length n.
- $A \xrightarrow{A} P : f^n(x)$.

Entity authentication number i

We will refer to the verifying entity for this session as B_i.

- $A \rightarrow B_i : f^{n-i}(x)$.
- A publishes the hash chain value and the verifier's identity: $A \rightarrow P : (f^{n-i}(x), B_i)$.
- B_i checks that the ith hash of $f^{n-i}(x)$ is equal to the public value $f^n(x)$.
- B_i checks that the published value is the first time $f^{n-i}(x)$ has been published, and that it is published along with their identity.

Scheme 4.11 requires one access to an authenticated channel for the initialisation, and can provide many entity authentications to different verifying entities. This is clearly an improvement on running a separate initialisation phase for each verifier.

The scheme is secure against an active adversary since there is no way for the adversary to use information obtained from previous session to impersonate A.

The scheme is not secure against a man-in-the-middle adversary. If the adversary blocks the publication of $(f^{n-i}(x), B_i)$ then they can authenticate to a party of their choice B^* by using $f^{n-i}(x)$ and publishing an edited version of A's message: $(f^{n-i}(x), B^*)$.

A disadvantage of the above scheme is that it is session-based. That is, A must wait until B_i is satisfied with the ith authentication before they can begin authenticating to B_{i+1}. If session $(i+1)$ starts before session i finishes then it is possible for the authentication value from session $i+1$ to be used to compute the authentication value for session i.

4.4.1.4 Summary of hash chain based schemes

In this section we saw how hash chains could be used as an efficient improvement on both the one-time schemes of Section 4.2 that require re-initialisation before every session, and the unlimited-time schemes of Section 4.3 that require a secure channel to set up.

We presented Entity Authentication Scheme 4.11, which is an extension of Scheme 4.9 to facilitate many verifiers. Although Scheme 4.11 is suitable to authenticate to many verifiers, it is limited by being session-based.

4.4.2 Hash tree and Merkle tree entity authentication schemes

We saw in the last section that we could use hash chains to facilitate many authentications for each initialisation phase. We can create a scheme with similar properties by using a Merkle tree (see Section 3.2.3).

Merkle trees also facilitate new schemes that were not possible with hash chains. In this section we will first look at the Merkle tree based entity authentication scheme, which in itself provides no advantages over Scheme 4.9, but which can be adapted to form two more attractive schemes.

The first is a new scheme designed for many verifying parties. This scheme improves on Scheme 4.11 as it is not session-based. The second is a challenge-response scheme which does not use a secure channel, and which provides more protection against a man-in-the-middle adversary than any previous such scheme.

4.4.2.1 Merkle tree based entity authentication

We can use a Merkle tree to form a simple entity authentication scheme which performs similarly to the hash chain based Scheme 4.9.

This scheme is in fact worse than the hash chain entity authentication scheme in terms of computational complexity and communication costs. The scheme makes an appearance here because it can easily be adapted to make a scheme suitable for authenticating to many people, and also a challenge-response scheme. Both of these schemes have properties that are arguably more elegantly provided by the Merkle tree than any other type of hash structure.

Entity Authentication Scheme 4.12: The n-time Merkle tree based entity authentication scheme. For a small example of this scheme see Figure 4.2.

Initialisation

- A generates a Merkle tree of height h from a large set of randomly chosen values (one for each leaf).[6] The values at each vertex are calculated as the hash of their children (see Section 3.2 for more details).
- A sends the root value x_r to B: $A \xrightarrow{A} B : x_r$.

Entity authentication

- A picks a leaf l whose value x_l has not been used before.
- A creates the sibling set X_l (see Section 3.2.1) of the path from l to the root.

[6]Note that this large set of values can be generated from a single master key MK. We create the ith random value as $f(MK||i)$. In this way A's storage requirement can be greatly reduced.

- $A \rightarrow B : (x_l, X_l; l)$.
- B computes, in turn, each value along the path from the leaf to the root.
- B checks that the received values are valid by comparing the computed root value with A's authenticated root value x_r received during the initialisation phase.
- B also stores the leaf l in a list of used leaves and checks the list to ensure this leaf has not been used before. Storing the whole list can be avoided if there is a prearranged order in which the leaves will be used.

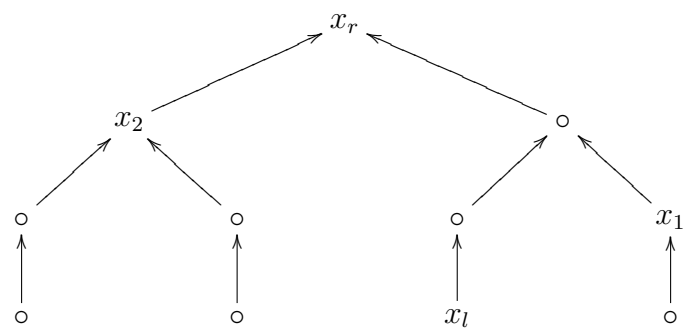


Figure 4.2: The Merkle tree based entity authentication scheme. The value at the chosen leaf is marked x_l and the vertices whose values are in X_l are marked x_1 and x_2. When B receives x_l and X_l, they can check the values are authentic by hashing them together and comparing with the root value x_r. For the example above they compute $f(x_2||f(f(x_l)||x_1))$, which should be equal to x_r.

A sends out the value at leaf l, and all the values of the siblings of vertices on the path from l to the root r. From this, B can check that everything hashes correctly to give the root (public key) value. A must also send out some information as to which leaf has been provided. The ordering of the values in X_l can be prearranged.

A Merkle tree is preferable to any other type of hash tree in this application as it avoids sending information about other leaves in X_l. If we do

not use a Merkle tree then there exists a leaf l with a sibling s. The authentication values for leaf l will be completely revealed by the authentication values for any leaf that is a descendant of s. Consequently, if A used a leaf descended from s before l then they would not later be able to use leaf l securely. This situation is completely avoided if we restrict the scheme to Merkle trees.

An entity using Scheme 4.12 can authenticate to a verifying party once for each leaf in their Merkle tree. By varying the in-degree of the vertices in the tree it is possible to trade-off computational complexity with the size of the authentication message. Using a binary Merkle tree will minimise the amount of communication, whereas using a Merkle tree of height 1 will minimise the computation during both the initialisation and authentication phases.

This scheme has the same security against active and man-in-the-middle adversaries as the hash chain entity authentication scheme (Entity Authentication Scheme 4.9). However this scheme requires many values for each authentication (whereas the hash chain entity authentication scheme only requires one). This scheme also requires more hash evaluations to verify each authentication value than the hash chain entity authentication scheme.

4.4.2.2 Merkle tree based entity authentication for many verifying parties

As mentioned earlier, Entity Authentication Scheme 4.12 is not quite as useful as Scheme 4.9. However with some minor changes it can be adapted to provide entity authentication to a large number of verifiers.

Scheme 4.11 is based on hash chains and was designed for many verifiers. However, due to the nature of hash chains, it was not possible to authenticate to different entities at the same time using this scheme. In this section we present a scheme using Merkle trees to overcome this problem.

Entity Authentication Scheme 4.13: The Merkle tree based entity authentication for many verifying parties.

Initialisation

- A generates a Merkle tree of height h as in Scheme 4.12.
- A publishes the root value x_r: $A \xrightarrow{A} P : x_r$.

Entity authentication number i

We will refer to the verifying entity for this authentication as B_i.

- A performs the Entity Authentication phase of Scheme 4.12 with B_i.
- A publishes the leaf used, the leaf value x_l and the verifier's identity: $A \xrightarrow{A} P : (l, x_l, B_i)$. We assume that the server on which the data is published stores entries with the approximate time and date that they were published.
- B_i can check the received values as in Scheme 4.12.
- B_i should check that the leaf has not already been used by searching the published list. The verifier can also check the time that the value was published to ensure that it has not been used to authenticate to them before.[7]

A must use a weak hash function f (Definition 2.8). This will avoid a verifier being able to attack the scheme by finding a second preimage of a received authentication set.

A must publish each leaf value when they use it, along with the identity of the person it was used to authenticate to. This prevents an active adversary replaying an authentication set that has already been used, or passing off A's authentication as their own.

[7] If this was not the case it could lead to an attack in which an active adversary authenticates to B_i immediately after A has authenticated to them, using exactly the same values that A used, except that the adversary does not publish anything. Assuming B_i does not keep records of the values used to authenticate to them, then they will accept it as if it were a completely new authentication.

Anyone authenticating A must check the given leaf value against the published list of expired leaf values. It is not necessary for A to publish the values X_l during the entity authentication phase, as the second preimage resistance of f prevents anyone from finding an alternative authentication set with a different leaf value.

A man-in-the-middle adversary can attack this scheme by stealing the authentication values for a particular leaf and blocking the publishing step. They can then provide the values they received when they wish to pose as A, and publish the appropriate values.

Scheme 4.13 is more suitable than Scheme 4.11 for use with multiple users, as the authentication values can be revealed in any order without affecting the security of each other. Scheme 4.11 is based on hash chains and requires each entity authentication to be completed before the next can be started.

We note that if the number of leaves is significantly smaller than the range of f, then the leaf values can be chosen at random without replacement. This allows the publishing step in the entity authentication phase to be replaced by $A \xrightarrow{A} P : (x_l, B_i)$, as the leaf value can serve as an index.

4.4.2.3 Merkle tree based challenge-response entity authentication

In Section 4.3.2.1 we saw how challenge-response schemes can be used to strengthen authentication against some man-in-the-middle adversary attacks. Our first challenge-response scheme (Scheme 4.5) was dependent on access to a secure channel for the initialisation phase.

We can use the Merkle tree to make a challenge-response authentication scheme. This scheme is not prone to record and replay attacks, as the verifier picks the leaf value they want A to provide at random from the unused leaves (in the same way as the verifier picks a random challenge message m in Scheme 4.5). Our tree must have a large number of leaves to keep the set of unused leaves big enough that the challenge is a meaningful obstacle to a man-in-the-middle adversary.

A needs to send much more data than for either of the basic challenge-

response scheme (Scheme 4.5) or the time-dependent variable scheme (Scheme 4.7), but this scheme has the huge advantage that A and B do not need to use a secure channel.

Entity Authentication Scheme 4.14: The Merkle tree based challenge-response entity authentication scheme.

Initialisation

- A follows the initialisation phase of Entity Authentication Scheme 4.12.

Entity authentication

- B chooses a leaf l at random from the set of unused leaves and sends it to A: $B \rightarrow A : l$.
- A follows the entity authentication phase of Entity Authentication Scheme 4.12 with leaf l.

This allows B to choose which leaf should be provided, so it is B's responsibility to pick a leaf that has not been asked for before (for example by storing the list of leaves they have already asked for).

If an adversary can pose as B to A then they can obtain many authentication values. If they subsequently pose as A to B then there is a chance that B will choose a leaf that they know the correct authentication value for. This chance can be reduced by using a large number of leaves.

4.4.3 Hierarchical chain construction scheme

An issue with the hash chain and Merkle tree based schemes described above is the amount of computation needed for the initialisation phase. All of these schemes require A to compute the root value before any entity authentication can take place.

We consider the application of a sensor network (for example see [76, 107, 159]), for which the initialisation phase for each sensor is performed by a central computer before the sensors are distributed. We assume that the sensors are required to authenticate to an entity B using a light-weight

hash based entity authentication scheme such as one of the schemes in this chapter. In this application the central computer may need to repeat the initialisation phase a large number of times, so it will be advantageous to use an entity authentication scheme with a quick initialisation phase.

It may suffice in this case to deploy a scheme that does not check for a man-in-the-middle adversary during every session.

By using a hierarchical chain construction, we can maintain n-time entity authentication while substantially reducing the number of hashes in the initialisation phase.

In Scheme 4.15 we describe the two dimensional hierarchical chain construction based entity authentication scheme was suggested in [76]. It allows a much smaller initialisation time than the n-time schemes we have seen so far. To provide for n authentication instances, only approximately $\sqrt{n}$ hash applications need to be done to compute the initialisation value from the chain's seed. This compares favourably with Scheme 4.9, which is based on a hash chain, for which a hash application needs to be done for each authentication instance required. It also compares favourably with Schemes 4.12, 4.13 and 4.14 based on a Merkle tree, for which approximately two hash applications need to be done for each authentication instance required.

The decrease in initialisation time comes at the cost of a compromise in security. However, for some applications such as sensor networks, this may be an acceptable compromise.

Entity Authentication Scheme 4.15: The two dimensional hierarchical chain construction based entity authentication scheme from [76]. An example is depicted in Figure 4.3.

Initialisation

- A creates a two dimensional hierarchical chain construction (see Section 3.4.3, Definition 3.9) with chain lengths $\{n, m\}$ using a randomly chosen seed $x_{0,0}$. A need not compute all the values in the construction now, only those necessary to find $x_{n,m}$.
- $A \xrightarrow{A} B : x_{n,m}$.

Entity authentication number i

- For the ith instance of entity authentication, A finds the unique values α and β satisfying $i = \alpha m + \beta$ and $\beta \in [0, m-1]$.
- If $\beta \in [0, m-2]$ then A sends the next value in the current sub-chain: $A \rightarrow B : x_{(n-\alpha),(m-\beta-1)}$.
- B can verify the value by hashing and checking against the previously received value. B stores this value for the next authentication phase.
- If $\beta = m-1$ then A sends the final value in the current sub-chain, and begins the next sub-chain: $A \rightarrow B : \left(x_{(n-\alpha),0}, x_{(n-\alpha-1),m}\right)$. Note that A must have computed the values in the new sub-chain by this point.
- B can verify the first value $x_{(n-\alpha),0}$ by hashing and comparing with the previously received value $x_{(n-\alpha),1}$. B can also check this value by hashing and checking against the value $x_{(n-\alpha+1),0}$. By doing this B can convince themselves that this whole sub-chain was authentic.
- Note that B cannot check $x_{(n-\alpha-1),m}$ yet, but must wait until $x_{(n-\alpha-1),0}$ is released, and then check the whole sub-chain.

$$
\begin{tikzcd}
x_{0,0} \arrow[r] \arrow[d, dotted] & x_{1,0} \arrow[r] \arrow[d, dotted] & x_{2,0} \arrow[r] \arrow[d, dotted] & x_{3,0} \arrow[d, dotted] \\
x_{0,1} \arrow[d, dotted] & x_{1,1} \arrow[d, dotted] & x_{2,1} \arrow[d, dotted] & x_{3,1} \arrow[d, dotted] \\
x_{0,2} \arrow[d, dotted] & x_{1,2} \arrow[d, dotted] & x_{2,2} \arrow[d, dotted] & x_{3,2} \arrow[d, dotted] \\
x_{0,3} \arrow[d, dotted] & x_{1,3} \arrow[d, dotted] & x_{2,3} \arrow[d, dotted] & x_{3,3} \arrow[d, dotted] \\
x_{0,4} & x_{1,4} & x_{2,4} & x_{3,4}
\end{tikzcd}
$$

Figure 4.3: A hierarchical chain construction in 2 dimensions with chain lengths $\{n, m\} = \{3, 4\}$.

This scheme works in a similar way to using many short hash chains. There is opportunity for an extra check that the seeds of the chains $(x_{i,0})$

also form a hash chain, but the only way to know that a given leaf value $x_{i,m}$ of a sub-chain is part of the hierarchical chain construction is to wait until the value $x_{i,0}$ is revealed.

This scheme is protected against attack by an active adversary as B can validate each authentication value with the previous one, and the authentication value changes each time.

A man-in-the-middle adversary E_M can only attack one step at a time with an intercept and replay attack. However if E_M attacks the scheme when $\beta = 0$ then they can substitute $x_{(n-\alpha),m}$ for the end value of a sub-chain of their own. This attack would be imperceptible for the whole length of that sub-chain. Only when B asks for the base value of the spoof sub-chain, and finds that it does not hash to give $x_{(n-\alpha+1),0}$, will it become apparent that the adversary is not A. This attack also prevents A from using their own sub-chain to prove their identity, so A must sacrifice it and move onto the next sub-chain.

The scheme as it stands is vulnerable to denial of service attacks and message loss. Extensions to this scheme that address these issues are presented in the original paper.

The d-dimensional hierarchical chain construction can also be extended to an entity authentication scheme [76]. The analysis of this scheme is very similar to the analysis of Entity Authentication Scheme 4.15.

4.4.4 Comparison of schemes in this section

A summary of the n-time entity authentication schemes from this section is given in Table 4.4. Here we use the value 1^* to denote a value that is 1 during the majority of sessions, but which occasionally has a different value.

Scheme	Complexity		Comm. Cost		Init.	Security		Notes
	Init	Auth	Init	Auth	Ch.	E_A	E_M	
4.9	n	$0, 1$	1	1, 0	A	n	1✓	
4.10	n	$0, 1$	3	1, 0	AP	n	1✓	A
4.11	n	$0, i$	1	$3, 0$	AP	n	1✓	B
4.12	$O(dn)$	$0, O(\log_d n)$	1	$O(\log_d n), 0$	A	n	1✓	
4.13	$O(dn)$	$0, O(\log_d n)$	1	$O(\log_d n), 0$	AP	n	1✓	B
4.14	$O(dn)$	$0, O(\log_d n)$	1	$O(\log_d n), 1$	A	n	✗	
4.15	$2\sqrt{n}$	$0, 1^*$	1	$1^*, 0$	A	n	$\sqrt{n}$✓	C

Table 4.4: Summary of the n-time entity authentication schemes from this section. For an explanation of the values and column headings, see Section 4.2.5.
Notes
A: This scheme has public verifiability.
B: These schemes can be used to authenticate to many verifiers.
C: Occasionally A sends two values during the authentication phase. When this happens, B must perform two hash evaluations. During the authentication phase A must perform approximately $(n - \sqrt{n})$ hash evaluations to find the values that were not calculated in the initialisation phase. Each value must be found before the session in which it is needed.

4.5 Other (mainly n-time) hash-based entity authentication schemes

In this section we look at some other schemes from the literature. We begin by looking at weakened hash functions and suggest an extension of the sandwich chain to an entity authentication scheme. We then explore the comb skipchain construction, which offers unlimited-time entity authentication without requiring access to a secure channel. We finish by looking at entity authentication schemes based on the general hash chain and on a hash chain with breakpoints.

4.5.1 Using weakened hash functions to improve efficiency

In this section we will use two types of hash functions. The first is a 1-secure l-bit strong hash function for l suitably large (for example 128 or 256). This is the type of hash function that we have dealt with for much of this thesis. The second function is an $(1-\epsilon)$-secure l'-bit hash function, for $\epsilon \approx 0$ and $l' < l$ (for example, l' could be 32 or 64). We will refer to the latter as a *weakened hash function*.

For the purposes of this section we assume that there may exist a weakened hash function that is faster than a standard hash function. This assumption is intuitive, but we do not have a concrete example of a weakened hash function that proves the point. There has been little research published on the subject of weakened hash functions and, as we saw in Section 2.7, there has been a certain amount of trouble creating 'full size' hash functions with predictable security.

Scheme 4.16 is based on a set of weakened hash chains that are tied together at both ends by stronger commitments to form a sandwich chain [55]. Each weakened hash chain can be used for entity authentication in the same way that the hash chain is used in Scheme 4.9. However each weakened hash chain is only secure for a time t, after which we assume an adversary can find arbitrary preimages. When a weakened hash chain expires, or when it is exhausted, the strong commitments are used to allow the authenticator to start using the next weakened hash chain.

An example sandwich chain is shown in Figure 4.4 and the full definition is given in Section 3.4.6.

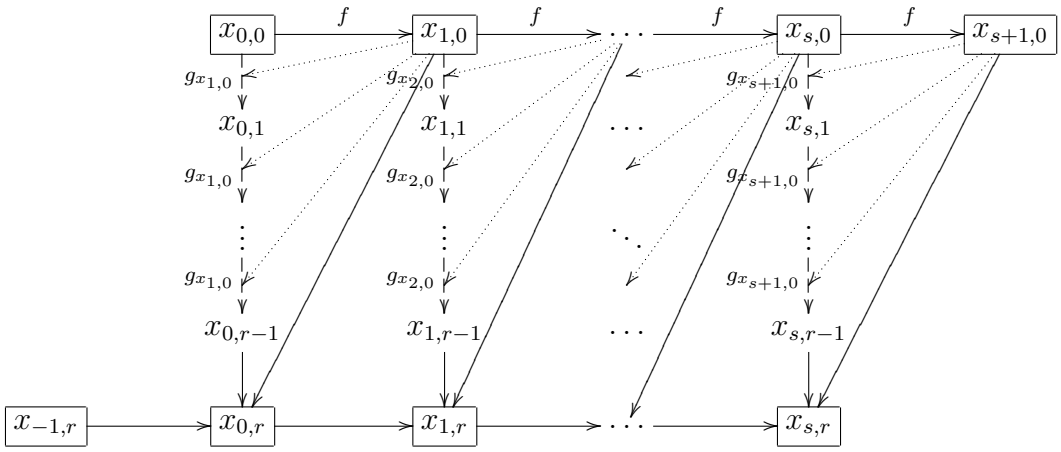


Figure 4.4: A sandwich chain of length s and depth r. Hash function f is a strong hash function (Definition 2.10), whereas $g_{i,j}$ is a weakened hash function. The values in boxes are full sized cryptographic values (e.g. 256 bits) and the other values are the output of a weakened hash function, which may be shorter (e.g. 32 bits). The weakened hash function for each vertical chain is chosen by the value at the top of the next chain.

Entity Authentication Scheme 4.16: A sandwich chain based entity authentication scheme. We will use the notation from the sandwich chain in Figure 4.4 for this description.

Initialisation

- A creates a sandwich chain of length s and depth r, seeded from two randomly chosen values $x_{0,0}$ and $x_{-1,r}$.
- $A \xrightarrow{A} B : (x_{s+1,0}, x_{s,r})$.

Entity authentication

- Suppose that the most recent sandwich chain value that B has received is $x_{i,j}$, where $j > 0$. Hence for this purpose the value of interest that B received during the initialisation phase is $x_{s,r}$ and not $x_{s+1,0}$.
- If time t has not passed since A sent $x_{i,r}$, and $j \geq 2$, then A sends the next value in the current weakened hash chain: $A \rightarrow B : x_{i,j-1}$.

- In this case B checks the weakened hash $g_{x_{i+1,0}}(x_{i,j-1})$ against $x_{i,j}$ to validate A's claim.

- If time t has passed since A sent $x_{i,r}$, or $j = 1$, then A sends the last value of the current weakened hash chain $x_{i,0}$, and the first value of the next $x_{i-1,r}$: $A \rightarrow B : (x_{i,0}, x_{i-1,r})$.

- In this case B can validate A's claim by checking both values are as they should be. The first value $x_{i,0}$ can be hashed and B should find that $f(x_{i,0}) = x_{i+1,0}$.[8] The second value $x_{i-1,r}$ can be checked using the equality $x_{i,r} = f(x_{i-1,r}||x_{i,r-1}||x_{i+1,0})$.

Given that a fast weakened hash function exists, Scheme 4.16 potentially requires less computation during both the initialisation phase and the entity authentication phase than any other n-time entity authentication scheme in this chapter.

Due to the nature of the values in the sandwich chain, an active adversary E_A cannot imitate A without computing the preimage of a hash digest. This value may be the output of a weakened hash function, and so E_A's task is vastly easier than for a preimage-resistant hash function. However, we have assumed the fastest that E_A can find such a preimage is in time t, by which time A and B have moved on to using a new weakened hash function.

Since the weakened hash function for sub-chain i is selected by the value $x_{i+1,0}$, it is impossible for E_A to begin attacking the chain before $x_{i+1,0}$ is released. If the weakened hash function was chosen in a predictable manner (for example by using the sub-chain index i) then E_A could begin attacking the sub-chain before it came into use (for example by compiling rainbow tables [97] for it).

A man-in-the-middle adversary cannot change any of the values that A sends, as they are all linked to values which B has already received. They can however intercept an authentication message that A sends and use it instead.

[8] Note that B could also check $x_{i,0}$ against the rest of the weakened hash chain. However A might have sent $x_{i,0}$ precisely because the weakened hash chain has become compromised.

4.5.2 Comb skipchain construction

The comb skipchain construction (Definition 3.16) is equivalent to seeding a hash chain with the public key of a *one-time signature scheme* (see Definition 3.15). The signature from the one-time signature scheme is used (when appropriate) to authenticate another comb skipchain end value (see Figure 4.6). This idea was presented by Hu et al. in [55], which extends work from [53]. In [43] a similar method of joining one-time signature schemes to hash chains is suggested, but it is less suited to entity authentication.

The comb skipchain can be used to authenticate an unlimited number of times from one initialisation. In addition it does not require a secure channel. It is the first scheme that we have discussed with both of these properties.

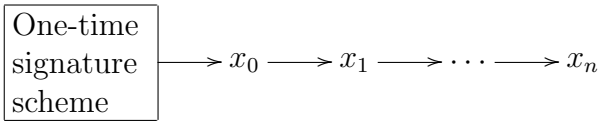


Figure 4.5: A comb skipchain construction.

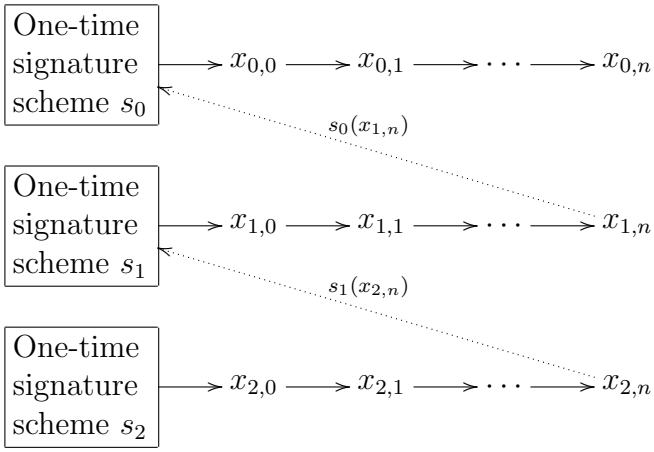


Figure 4.6: A figure showing how three comb skipchain constructions are tied together to make one long chain. The advantage over using hash chains is that the one-time signature can be used to verify the last value in the next chain, no matter what that value is. Thus more comb skipchains can be generated as and when they are needed, without the need for another initialisation phase.

Entity Authentication Scheme 4.17: The comb skipchain entity authentication scheme. This description is based on the comb skipchain constructions in Figure 4.6.

Initialisation

- A forms a comb skipchain seeded from a one-time signature scheme s_0 and a hash chain $\{p_0, x_{0,0}, \ldots, x_{0,n}\}$ of length $n+1$ seeded from its public key p_0.
- A sends the end value $x_{0,n}$ to B using an authenticated channel $A \xrightarrow{A} B : x_{0,n}$.

Entity authentication

- If the last value that A sent to B was $x_{i,j}$, with $j \in [1, n]$, then A sends $x_{i,j-1}$ to B: $A \rightarrow B : x_{i,j-1}$.
- If the last value that A sent to B was $x_{i,0}$ then A sends the value p_i to B: $A \rightarrow B : p_i$.
- In either of these cases B can verify A's identity by hashing the value just received to check that the result is the same as the previous value received.
- If the last value that A sent to B was p_i then A creates a new one-time signature scheme s_{i+1} and uses it to seed a new comb skipchain.
- A creates a signature on the end value from the new chain using the one-time signature scheme that seeded the previous chain. A sends this value and its signature to B: $A \rightarrow B : (x_{i+1,n}, s_i(x_{i+1,n}))$.
- B can verify the signature against the public key value p_i, and accepts A and the value $x_{i+1,n}$ as authentic if successful.

This scheme is secure against active adversaries and may only be attacked one value at a time by man-in-the-middle adversaries using an intercept and replay attack.

A typical one-time signature scheme requires more computation than a hash chain, so this approach will not be suited to the most light-weight of applications. However, without access to a secure channel and with only one use of an authenticated channel, it can provide unlimited entity authentications.

4.5.3 General hash chain

One as yet unconsidered problem with the hash chain based entity authentication scheme (Scheme 4.9) arises if an adversary somehow manages to get hold of the ith authentication value x_i while the authenticating parties are only up to the jth authentication value x_j ($j < i$).

The adversary would be able to generate all authentication values between x_i and x_j from x_i, and therefore could impersonate A several times. This is addressed by Bradford and Gavrylyako using general hash chains (see Section 3.4.7) [16].

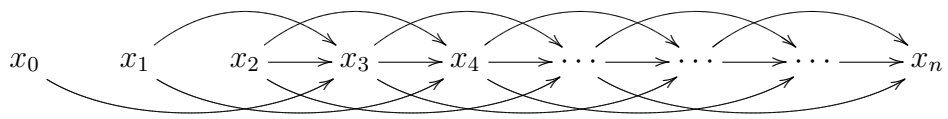


Figure 4.7: A general hash chain of length n and depth 3. The first three values $\{x_0, x_1, x_2\}$ are the seeds. All the remaining values are generated from these three values using the equation $x_i = f(x_{i-3}||x_{i-2}||x_{i-1})$.

Entity Authentication Scheme 4.18: The general hash chain entity authentication scheme from [16] with a general hash chain of length n and depth d. See Figure 4.7 for an example of a general hash chain.

Initialisation

- A picks d random values $\{x_0, \ldots, x_{d-1}\}$, and uses them to seed a general hash chain of length n and depth d.
- A sends a list of values to B: $A \xrightarrow{A} B : \{x_n, x_{n-1}, \ldots, x_{n-d+1}\}$.
- B stores the next session number i (initially $i = 1$).

Entity authentication number i

- A announces they intend to authenticate to B.
- B sends A the session number i.
- $A \rightarrow B : x_{n-i}$.
- B checks that $x_{n-i} = f(x_{n-i-d}||x_{n-i-d+1}||\ldots||x_{n-i-1})$ using the previously received authentication values.

A general hash chain of length n and depth d only allows $n - d + 1$ authentications, because d of the values are used during the initialisation phase. Note that if $d = 1$, we have exactly the same scheme as Entity Authentication Scheme 4.9.

The only difference in the security analysis between this scheme and Scheme 4.9 is that if an adversary gets hold of an authentication value x_i before it has been used, that adversary cannot generate any other authentication values from the stolen value. In contrast, if an adversary obtains an authentication value x_i in Scheme 4.9 then they can generate all authentication values x_j with $j > i$. With Scheme 4.18, an adversary has to steal d consecutive values before they can begin generating other authentication values.

4.5.4 Hash chain with breakpoints

Another construction which deals with the issue of an adversary who has discovered a hash chain value ahead of time is a scheme due to Goyal [42]. The idea is to have the hash chain generated from two sources, so that if an adversary gets hold of an authentication value before it is used then it only reveals a few other values. The hash chain with breakpoints is defined in Section 3.4.8 and an example is shown in Figure 4.8. We give an entity authentication scheme based on this in Scheme 4.19.

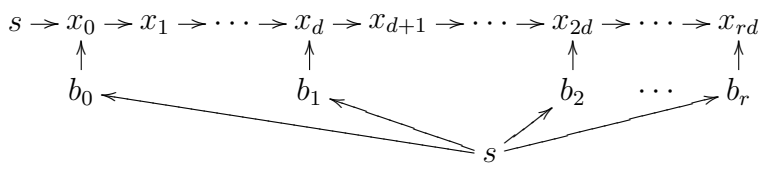


Figure 4.8: Hash chain with break points distance d apart seed s (from [42]).

Entity Authentication Scheme 4.19: Entity authentication based on a hash chain with breakpoints distance d apart.

Initialisation

- A chooses a seed s at random and generates a hash chain of length rd with breakpoints distance d apart.
- A sends the end value to B: $A \xrightarrow{A} B : x_{rd}$.

Entity authentication number i

- If i is not divisible by d then A sends the next value of the chain $A \to B : x_{rd-i}$.
- If i is divisible by d then A sends the next authentication value and the corresponding breakpoint value: $A \to B : (x_{rd-i}, b_{r-i/d})$.
- B can verify whatever they are sent by hashing to compare with the previous authentication value.

If a value is stolen by an adversary before it has been used then it only reveals a few values after it, the breakpoints prevent all the other values being derived from it. If, instead, an adversary somehow steals a breakpoint value then they obtain no authentication values.

4.5.5 Summary of schemes in this section

In Table 4.5 we provide a comparison of the schemes we have looked at in this section. Here we use the value 1^* to denote a value that is 1 during the majority of sessions, but which occasionally has a different value.

Scheme	Complexity		Comm. Cost		Init.	Security		Notes
	Init	Auth	Init	Auth	Channel	E_A	E_M	
4.16	$O(n)$	$0, 1^*$	2	$1^*, 0$	A	n	1✓	A
4.17	$s + O(n)$	$0, 1^*$	1	$1^*, 0$	A	∞	1✓	B
4.18	n	$0, 1$	d	$1, 0$	A	n	1✓	C
4.19	$O(n)$	$0, 1$	1	$1^*, 0$	A	n	1✓	D

Table 4.5: Summary of the entity authentication schemes in Section 4.5. For an explanation of the values and column headings see Section 4.2.5.
Notes
A: It is possible that short digest hash functions may require less computation than the hash functions used in other schemes. Occasionally A will send two values to B, and when this happens B may have to perform $O(\sqrt{n})$ hash evaluations.
B: We assume we are using the one-time signature scheme from Section 5.4.3, and that our cryptographic values have b bits. The value s can be computed as $(\frac{b}{0.32} - 1)$. Every n sessions A sends about $\frac{6b}{8}$ values, and B must perform approximately $\frac{s}{2}$ hash evaluations to verify.
C: If somehow an adversary gets hold of an authentication value before it has been used, they cannot use it to generate other values in the chain (as they would be able to in Scheme 4.9). An adversary must steal d consecutive values before this attack can be used.
D: If somehow an adversary gets hold of an authentication value early on in the chain, they can only discover at most d authentication values. If the same thing happened in Scheme 4.9 then potentially all the authentication values would be compromised. Every d sessions A must send two values to B.

4.6 Conclusions

In this chapter we investigated the application of hash structures from Chapter 3 to entity authentication. We presented the notion of public verifiability, and showed that some existing schemes are publicly verifiable. We presented an entity authentication scheme based on the chained pseudo-random number generator which has the minimum storage requirements possible of an entity authentication scheme secure against eavesdroppers. We presented a Merkle tree based entity authentication scheme suitable for use with many verifiers. We compared these with other existing hash-based schemes from the literature.

Chapter 5

Signatures

In the previous chapter we studied the applications of hash functions and hash structures to entity authentication. In this chapter we continue our investigation by looking at the applications of hash structures to message authentication, and in particular, signatures.

We begin the chapter with a brief introduction to message authentication. We then continue by studying some existing one-time signature schemes, and look at how one-time signature schemes can be formed on generalised hash DAGs. We present an algorithm which finds signature schemes for any given generalised hash DAG. We then present some concrete examples of one-time signature schemes based on generalised hash DAGs, including two new examples which are more efficient than any other usable schemes that we know of. Finally, we look at k-time signature schemes and address the issue of creating a suitable efficiency measure. We present the notions of 'perforated' and 'porous' k-time signature schemes, and look at some schemes that illustrate these definitions.

5.1 Introduction to message authentication

We start this section by looking at message integrity and message authentication codes. We look at how hash structures are used in data integrity schemes. We give some important definitions relating to signatures and introduce the notion of a one-time signature.

5.1.1 Checksums and data integrity schemes

A very simple form of message authentication is authentication with a checksum. A *checksum* is a hash function (Definition 2.6) used to detect accidental corruption of data. The exact choice of hash function will be application dependent (for example see [149] for an analysis of the choice of hash function used to compute ISBN-10 checksums).

Although checksums can be used to detect accidental changes to a message, they do not give much protection against active adversaries. There is no requirement for a checksum to be second preimage resistant, and so it may be easy for the adversary to find another message with the same checksum.

The following scheme is very suitable for large file systems, or for version management of large projects. It is most well suited for environments where only a few files (or only a few parts of files) change at once. It is used in many peer to peer file sharing applications [3, 73, 140, 152].

Data Integrity Scheme 5.1: A data integrity scheme for a file system containing files $\{m_0, \ldots, m_{n-1}\}$, using a weak hash function f (Definition 2.8). This scheme can also be used to provide data integrity for a large file split into n sections $\{m_0, \ldots, m_{n-1}\}$.

Initialisation

- The file manager A computes and stores the hash of each file $\{f(m_0), \ldots, f(m_{n-1})\}$.
- A creates a hash tree seeded by the hashes of each file and stores all the intermediate values. Note that if we consider the hash tree together with the original files then we have a Merkle tree (Definition 3.4).
- $A \xrightarrow{A} P : x_r$, where x_r is the root value of the hash tree.

Update

- When a file m_i is changed, A computes the hash of the new file.
- A only needs to perform a hash for each vertex on the path from the affected leaf to the root, as the rest of the hash tree values have not changed.

- A can perform multiple updates at once by recomputing all affected hash tree values.

Choosing the shape of the Merkle tree is an application dependent problem. If some files are very related, and are likely to be changed at the same time, then it would be sensible to place them on leaves that are siblings of each other.

5.1.2 Message authentication codes

The simplest form of message authentication secure against an active adversary is a 'message authentication code' (or 'MAC'). Communicating parties share a secret key x and append to a message m the message authentication code $MAC(x, m)$.

It is beyond the scope of this document to discuss MACs in detail, but there are many relationships between MACs and hash functions. For a more detailed discussion of MACs, see [83] Section 9.5.

Message Authentication Scheme 5.2: Message authentication using a message authentication code.

Initialisation

- A chooses a secret key x and shares it with the authorised recipient B: $A \xrightarrow{S} B : x$.

Message authentication

- If A wishes to send an authenticated message m over an unprotected channel then A generates a MAC for the message $MAC(x, m)$.
- A sends the message and the MAC to B: $A \rightarrow B : (m,\ MAC(x, m))$.
- B uses the message m and the secret key x to generate the MAC. Assuming that it is the same value they received from A, they accept the message as authentic.

An active adversary cannot change the message m without also changing the MAC; otherwise the recipient B will discover that the message has been altered. The adversary cannot generate a MAC for any new message without knowledge of the key x.

Although message authentication codes are robust, efficient and fairly straightforward, relying on a MAC for message authentication has three major limitations:

1. The message can only be authenticated by a party who knows the secret key.

2. Any party who knows the secret key can forge a MAC. This is not an issue for applications where the recipients have no reason to impersonate the sender. However in some applications it may be necessary to guard against this, or provide the capacity to change which users have access to the secret key.

3. Scheme 5.2 requires a secure channel for initialisation. In the next section we will look at signatures, which in general do not have this constraint.

5.1.3 Hash functions and conventional digital signatures

The aim of a digital signature scheme is to facilitate the binding of a message with the originating entity. A digital signature indicates that the signer 'agrees' with the contents of the message.

Menezes et al. give the following definitions [83]:

Definition 5.1. A **digital signature** is a data string which associates a message (in digital form) with some originating entity.

A **digital signature generation algorithm** (*signature generation algorithm*, or *signing algorithm*) is a method for producing a digital signature.

A **digital signature verification algorithm** (or *verification algorithm*) is a method for verifying that a digital signature is authentic (i.e. was indeed created by the specified entity). A **digital signature scheme** consists of a signature generation algorithm and an associated verification algorithm.

Although we have used a definition of a digital signature scheme which contains two algorithms, there is a third which is also sometimes included. Before any messages can be signed, a 'key generation' algorithm must be run by each entity to generate a private and public key pair (k_s, k_p). The public key k_p is sent or made available to anyone who may be required to validate a signed message.

The *size* of a signature scheme is the order of the image of the signature generation algorithm.

We note that some signature schemes allow message recovery from the signature itself, whereas others form a separate signature which should be sent along with the original message. Some of our schemes can be used with message recovery, but we will only be considering the appended signature model, as this is by far the most common in practice.

The signing algorithm maps messages from a (potentially unbounded) set $\mathcal{M}$ to a finite set $\mathcal{S}$. Most schemes use a signing algorithm consisting of a strong hash function f (see Definition 2.10) to map $\mathcal{M}$ onto a finite set R, followed by a mathematical algorithm $s : R \times \mathcal{K} \to \mathcal{S}$ taking the output of f and the private key k_s as inputs.

The hash function f must also be second preimage resistant otherwise the scheme will be susceptible to an obvious forgery. An adversary can take a valid message signature pair and create a new one using the same signature by appending it to a second preimage of the message.

The hash function must be collision resistant if it is to resist an attack

in which the adversary can get the signer to sign a message. In this attack, the adversary finds a collision m, m' under the hash function, asks for the signature for m, and then appends it to m'.

An advantage of using a hash function as the first part of the signing algorithm is that it allows us to extend a signature scheme that can sign messages from a finite set to a signature scheme that can cope with any message. Another related reason is that for very long messages, reducing the message size using a hash function is much more efficient than applying a (slower) mathematical algorithm to the whole message. The mathematical algorithm should also be resistant to a 'lucky adversary' who guesses the output correctly, by making sure that the output space is reasonably large (128- or 256-bit strings are commonly used).

The verification algorithm usually works by inverting the mathematical algorithm used in the signing phase using the public key and comparing the result with the hash of the message (computed from the received message). If the two are the same then the signature is accepted as valid.

For the rest of this chapter we will mainly focus on the mathematical function s, rather than the strong hash function f. Consequently we will treat the message m and the hash $f(m)$ as 'equivalent', and may use the term 'message' when we actually mean $f(m)$.

5.1.4 Digital signatures based on hash functions

Many digital signature schemes are based on one-way trapdoor functions, which are functions that are always easy to compute in one direction, but only easy to compute in the opposite direction with some additional information (in this case the private key). In practice, for most trapdoor functions there is no proof of the absolute hardness of computation in the reverse direction. Instead the reverse computation is shown to be as difficult as some underlying mathematical problem. Table 5.1 summarises some existing trapdoor related signature schemes.

However, signature schemes can be based on hash functions instead of trapdoor functions. Many such signature schemes are one-time signature

Signature scheme	Underlying mathematical problem
Full domain hash [6]	The RSA problem
DSA [93]	Discrete logarithm problem
ECDSA [155]	Elliptic curve discrete logarithm problem
ElGamal signature scheme	Discrete logarithm problem
Rabin signature scheme	Large integer factorisation problem

Table 5.1: Some existing trapdoor related signature schemes and their underlying mathematical problems.

schemes (see Definition 3.15), that is signature schemes which can sign at most one message without risk of forgery. In compensation for this obvious drawback, it is usual for one-time signature schemes to be more computationally efficient than 'many-time' trapdoor-based signature schemes (although this is not necessarily the case). Some one-time signature schemes are not primarily based on hash functions, but as they are also generally less efficient we will not consider them further.

Another benefit of only using hash functions is less dependence on specific mathematical problems. For example, if an efficient solution to the discrete logarithm problem is found then DSA signatures will become insecure, whereas if a signature scheme solely based on SHA-256 is broken then it may be possible to continue using the scheme by deploying another hash function instead. If quantum computing becomes more practical then all of the mathematical problems in Table 5.1 may become feasible to solve [113, 126].

A third advantage of one-time signatures over *deterministic* one-way trapdoor based signatures is that the latter are subject to attacks based on repeated signature use. One-time signatures are not vulnerable to such attacks because the signature for a particular message changes each time the scheme is reinitialised, which is once per message.

Some signature schemes can be used at most k times before a forgery is possible, and we will refer to these as *k-time signature schemes*. There are also schemes for combining several one-time signature schemes under one public key to make k-time signature schemes, or schemes which can be used indefinitely. We will look at these in Section 5.5.

5.2 One-time signatures

We begin this section by looking at some one-time signature schemes based on hash functions and hash chains. Next we consider the Vaudenay's rake one-time signature scheme, which generalises all the previous schemes. After this we consider the improvements that can be made to these schemes by using hash trees and Merkle trees. Finally we generalise from hash trees to generalised hash DAGs, and look at some important results on the efficiency of one-time signature schemes due to Bleichenbacher and Maurer [11].

5.2.1 Simple chain-based schemes

In this section we will look at three simple hash-based and hash chain based one-time signature schemes. We start by looking at arguably the most simple example of a one-time signature scheme, the Diffie-Lamport one-time signature scheme. Next we study the Winternitz one-time signature scheme, which generalises the Diffie-Lamport scheme by using hash chains instead of hash functions. Finally we examine the Diffie-Lamport-Merkle one-time signature scheme, which uses an improvement suggested by Merkle to improve the efficiency of the original Diffie-Lamport scheme.

5.2.1.1 The Diffie-Lamport one-time signature scheme

The first published one-time signature scheme based on hash functions was due to Rabin in 1978 [114]. It is rather inefficient and so we will instead look at Lamport's similar scheme from 1979 [68], which is slightly more efficient and much more straightforward.

The scheme uses a preimage-resistant hash function f. It is important that an adversary cannot find a preimage, as this would allow them to forge a signature on an unsigned message.

Signature Scheme 5.3: The Diffie-Lamport scheme for signing messages with n-bit hash outputs.

Key Generation Algorithm

- Signer A picks $\{x_0, \ldots, x_{2n-1}\}$ uniformly at random from $\{0,1\}^n$.
- Signer A computes $\{y_0, \ldots, y_{2n-1}\}$ by $y_i = f(x_i)$.
- The set $\{y_0, \ldots, y_{2n-1}\}$ is published as A's public key.

Signature Generation Algorithm

- To sign a hash value m, A releases as the signature the following set of private key values:

$$\{s_0, \ldots, s_{n-1}\} = \{x_{2i+m_i} : i \in [0, n-1], m_i \text{ is the } i^{th} \text{ bit of } m\}.$$

Verification Algorithm

- The verifier can check the signature $\{s_0, \ldots, s_{n-1}\}$ by checking that for each bit of the message m_i the following equality holds:

$$f(s_i) = y_{2i+m_i}.$$

If an adversary is to change any bit of the message and preserve the signature then they must find the private key value for the corresponding bit. This is equivalent to finding a preimage of the hash function for that public key value and so the scheme is secure.

However, if A signs two messages using the same private key then there is a good chance that the adversary will be able to use the two signatures to forge a third signature. We give an example of this in Figure 5.1.

5.2.1.2 The Winternitz one-time signature scheme

Shortly after Lamport published the Diffie-Lamport scheme, Winternitz suggested a generalisation of it. His idea appears in Merkle's paper [85].

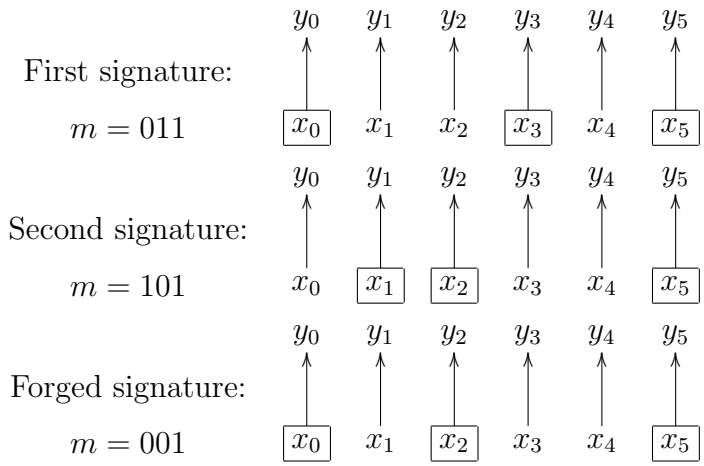


Figure 5.1: TOP and MIDDLE: A Diffie-Lamport one-time signature scheme is used to reveal two signatures (the signatures for the messages 011 and 101). BOTTOM: An adversary could use these two signatures to forge others (for example the signature for the message 001).

Instead of representing the message in binary, Winternitz suggested representing it in another base (say base r). We now consider a message m from a message space of n digit numbers in base r. We still use public and private keys consisting of $2n$ values, but now we define the public key values as the end values of hash chains seeded with x_i: $y_i = f^{r-1}(x_i)$. The scheme is based on the use of a preimage-resistant hash function f (see Definition 2.7).

Signature Scheme 5.4: The Winternitz scheme for signing messages with n digit hash outputs in base r.

Key Generation Algorithm

- Signer A picks $\{x_0, \ldots, x_{2n-1}\}$ uniformly at random from $\{0,1\}^n$.
- Signer A computes $\{y_0, \ldots, y_{2n-1}\}$ as $y_i = f^{r-1}(x_i)$.
- The set $\{y_0, \ldots, y_{2n-1}\}$ is published as A's public key.

Signature Generation Algorithm

- Signer A computes $s_{2i} = f^{r-m_i-1}(x_{2i})$ and $s_{2i+1} = f^{m_i}(x_{2i+1})$ for each digit m_i of the message.
- The signature is the set of values $\{s_0, \ldots, s_{2n-1}\}$.

Verification Algorithm

- The verifier can check $y_{2i} = f^{m_i}(s_{2i})$ and $y_{2i+1} = f^{r-m_i-1}(s_{2i+1})$ for each digit m_i of the message.

If we set r to 2 then the Winternitz scheme becomes the Diffie-Lamport scheme (Scheme 5.3), except that one of the two signature values for each digit of m is equivalent to a value in the public key for that digit and so does not need releasing.

If an adversary wishes to create a forgery on a message that was not signed then they must change at least one digit of the message and find a valid signature for that digit. Without loss of generality, we consider a forgery for the ith digit m_i. If the adversary increases the digit then they will have to find a preimage for $s_{2i} = f^{r-m_i-1}(x_{2i})$. If they decrease the digit then they will have to find a preimage for $s_{2i+1} = f^{m_i}(x_{2i+1})$. We assume that this is hard for a preimage-resistant hash function, so the Winternitz scheme is secure.

5.2.1.3 The Diffie-Lamport-Merkle one-time signature scheme

In [84] Merkle suggested an improvement to Scheme 5.3 which was based around only signing the 'one' bits of the message. This means that only one secret key value is needed for each bit of the message. However, in order to avoid forgeries which simply replace 1's in the message with 0's, it is necessary to append to the message a count of the zeros. This means that the private and public keys need to be $n + \lceil \log_2(n+1) \rceil$ values long (compared to $2n$ values in the original Diffie-Lamport scheme).

Signature Scheme 5.5: The Diffie-Lamport-Merkle scheme for n-bit messages. An example of the values used in this scheme when $n = 7$ is given in Figure 5.2.

Key Generation Algorithm

- Signer A picks $\{x_0, \ldots, x_{n+\lceil \log_2(n+1) \rceil - 1}\}$ uniformly at random from $\{0,1\}^n$.
- Signer A computes $\{y_0, \ldots, y_{n+\lceil \log_2(n+1) \rceil - 1}\}$ by $y_i = f(x_i)$.
- The set $\{y_0, \ldots, y_{n+\lceil \log_2(n+1) \rceil - 1}\}$ is published as A's public key.

Signature Generation Algorithm

- Signer A defines m' as the concatenation of m, the message to be signed, with a $\lceil \log_2(n+1) \rceil$-bit count of the number of 'zero' bits in m.
- A releases as the signature the following set of private key values:

$$\{x_i : m'_i = 1, \text{ where } m'_i \text{ is the } i^{th} \text{ bit of } m'\}.$$

Verification Algorithm

- The verifier computes m' from m, as described in the signature generation algorithm.
- The verifier then checks that for all 'one' bits of m' they have been provided with a valid preimage for the corresponding public key value.

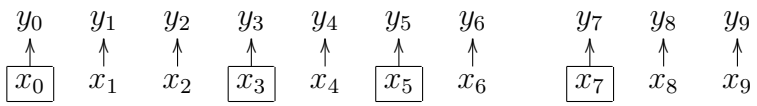


Figure 5.2: The Diffie-Lamport-Merkle one-time signature scheme for 7-bit messages. The values corresponding to the signature for the message $m = 1001010$ are highlighted with boxes. Note that 1001010 has four zeros, and the binary representation of four is 100, so $m' = 1001010100$, as defined in Signature Scheme 5.5.

If an adversary wants to make a forgery from an existing signature then they must (at least) either change a 0 to a 1 or (if no 0's have been changed to 1's) a 1 to a 0. If they try to change a 0 to a 1 then they must find a preimage of the hash function at the public key value corresponding to the changed bit. If they only change 1's to 0's then the adversary increases the number of 0's in the message, and so at least one 0 will change to a 1 in the count of the number of zeros field. This will require finding a preimage of the hash function for the corresponding public key value. Since it is assumed to be infeasible to find a preimage of the hash function, the scheme is considered secure.

There is also a fairly straightforward extension of the Winternitz scheme (Scheme 5.4) to a Winternitz-Merkle scheme. When signing messages with n digits in base r, the scheme requires $n + \lceil \log_r(n(r-1)+1) \rceil$ hash chains of length $r-1$. The first n are used as before to ensure that the signed message digits cannot be decreased, and the remaining digits are used to sign the difference between $n(r-1)$ and the sum of the message digits. The latter signing ensures that message digits cannot be increased without computing a preimage of a hash function.

5.2.2 Vaudenay's rake

In this section we will look at a natural generalisation of the schemes from the last section, Vaudenay's rake. We will define some terms that will come in useful later in the chapter and consider how to find the most efficient one-time signature scheme on Vaudenay's rake (which we will call 'Vaudenay's optimal rake one-time signature scheme').

5.2.2.1 Vaudenay's rake one-time signature scheme

The Winternitz-Merkle one-time signature scheme is not the most efficient use of hash chains for signing messages. In [142] Vaudenay proposed a generalisation of the Winternitz scheme and the Winternitz-Merkle scheme, which we will refer to as 'Vaudenay's rake' and which is in Signature Scheme 5.6.[1]

[1]The description of this scheme as a *rake* is due to Bleichenbacher and Maurer [10].

Vaudenay also proposed a specific instance of the scheme, which Bleichenbacher and Maurer [10] proved was an optimal use of hash chains to produce signatures, in the sense that for the same size of message space the 'optimal rake' scheme typically requires fewer hash chains, and so has a smaller private and public key. The 'optimal rake' scheme is described in detail in Signature Scheme 5.7.

Definition 5.2. A **signature pattern** for a one-time signature scheme based on a set of r hash chains $\{\{x_{0,0}, \ldots, x_{0,l}\}, \ldots, \{x_{r-1,0}, \ldots, x_{r-1,l}\}\}$ is a set U_i of vertices, such that U_i contains exactly one vertex from each hash chain.[2]

We denote the set of signature patterns by $\mathcal{U}$.

We let $U_i = \{u_{i,0}, \ldots, u_{i,r-1}\}$, such that $u_{i,k}$ is the vertex in signature pattern U_i that is also in the hash chain $\{x_{k,0}, \ldots, x_{k,l}\}$.

Therefore, for each U_i there is a set of integers $\{s_0, \ldots, s_{r-1}\}$ such that $u_{i,k} = x_{k,s_k}$ for all k. For convenience we may also refer to $\{s_0, \ldots, s_{r-1}\}$ as the signature pattern.

An example of a signature pattern is given in Figure 5.3.

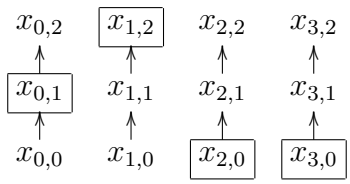


Figure 5.3: Vaudenay's rake with 4 chains of length 2. A signature pattern $U_i = \{u_{i,0}, u_{i,1}, u_{i,2}, u_{i,3}\} = \{x_{0,1}, x_{1,2}, x_{2,0}, x_{3,0}\}$ is marked with the boxes.

The signing algorithm for Vaudenay's rake one-time signature scheme is made up of three functions. The first, as discussed in Section 5.1.3, is a hash function $f : \mathcal{M} \to R$. The other two are the *signature pattern function* and the *evaluation function*.

[2]We will usually wish to consider families of signature patterns, and so use U_i to represent a signature pattern, as opposed to U.

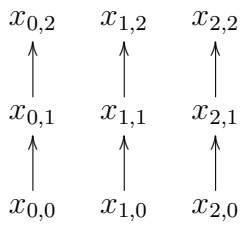
Definition 5.3. The Vaudenay's rake **signature pattern function** $\theta : f(\mathcal{M}) \to \mathcal{U}$ is a function which takes as input a message $m \in \mathcal{M}$, and outputs a signature pattern $U_i \in \mathcal{U}$ from the scheme. The function should be easy to compute and easy for the signer to publish. The function should also be such that for any two signature patterns U_i and U_j there exist hash chains $\{x_{k,0}, \ldots, x_{k,l}\}$ and $\{x_{k',0}, \ldots, x_{k',l}\}$ such that $u_{i,k}$ is not on the path from $x_{k,0}$ to $u_{j,k}$ and $u_{j,k'}$ is not on the path from $x_{k',0}$ to $u_{i,k'}$.

The Vaudenay's rake **evaluation function** $\psi : \mathcal{U} \times \mathcal{K} \to \mathcal{S}$ is a function which takes a signature pattern and a private key and outputs a signature. The signature consists of the value at each vertex in the signature pattern.

The Vaudenay's rake **verification function** $\upsilon : \mathcal{M} \times \mathcal{S} \times \mathcal{K} \to \{0, 1\}$ takes a message, signature and public key and outputs a value indicating whether the signature is valid for the message and public key. This can be computed by finding the signature pattern from the message and then hashing from the values given in the signature to find the end value of each chain, and then comparing with the public key.

In Figure 5.4 we give an example of Vaudenay's rake one-time signature scheme and the corresponding signature pattern function. We note that for small examples like this it is acceptable to use a look-up table as the signature pattern function, but for larger message spaces (for example the set of 128-bit hash values) a look-up table could become unmanageable.

We note that the Winternitz and Winternitz-Merkle schemes are both based on a set of hash chains where each signature consists of a set of values, one from each chain. To prevent a forgery, the set of values for any signature must not be obtainable from the set of values for any other signature. Vaudenay's generalisation (Scheme 5.6) also uses a set of hash chains and each signature consists of a value from each chain with the same property required to prevent forgeries.



Hash value	Values revealed as signature
000	$x_{0,0}$, $x_{1,1}$, $x_{2,2}$
001	$x_{0,0}$, $x_{1,2}$, $x_{2,1}$
010	$x_{0,1}$, $x_{1,0}$, $x_{2,2}$
011	$x_{0,1}$, $x_{1,1}$, $x_{2,1}$
100	$x_{0,1}$, $x_{1,2}$, $x_{2,0}$
101	$x_{0,2}$, $x_{1,0}$, $x_{2,1}$
110	$x_{0,2}$, $x_{1,1}$, $x_{2,0}$

Figure 5.4: An example of Vaudenay's rake one-time signature scheme. On the left is a set of three hash chains on which the signature scheme is based. On the right is a look-up table for the signature pattern function of our signature scheme.

Signature Scheme 5.6: Vaudenay's rake one-time signature scheme from [142].

Key Generation Algorithm

- Signer A picks r values $\{x_{0,0}, \ldots, x_{r-1,0}\}$ at random. This will be the private key k_s for the scheme.
- A creates r hash chains, each of length l, seeded by $\{x_{0,0}, \ldots, x_{r-1,0}\}$.
- A publishes the end value of each chain $\{x_{0,l}, \ldots, x_{r-1,l}\}$ as the public key k_p for the scheme.
- It is essential that the signature pattern function θ is known, or can easily be found, by any potential verifier.

Signature Generation Algorithm

- The signer A applies the signature pattern function θ to the $f(m)$ to find which signature pattern should be used.
- A computes the value at each vertex indicated by the signature pattern from the private key values $\{x_{0,0}, \ldots, x_{r-1,0}\}$.
- A releases the set of all these values $\{s_0, \ldots, s_{r-1}\}$ as the signature.

Verification Algorithm

- The verifier B applies the signature pattern function to the message m to find the signature pattern for the message.
- For each value s_i in the signature, B hashes along the chain to find the chain's end value $x_{i,l}$. B then checks that this is the same as the corresponding value in the public key.
- If all the signature values validate against the public key then B accepts the signature as authentic.

It is important that A only uses the signature scheme once, as if they reveal signatures on more than one message it is likely that an adversary could forge signatures on other messages. We consider the example of Signature Scheme 5.6 with three chains of length two, and the signature pattern function given in Figure 5.4. If A signs two messages with hash values 011 and 101, then an eavesdropper can obtain the values $x_{0,1}, x_{1,1}, x_{2,1}, x_{0,2}$ and $x_{1,0}$. Using $x_{2,1}$ the adversary can compute $x_{2,2}$ and then will possess all the values they need to create a forgery on a message with hash value 010 ($x_{0,1}, x_{1,0}$ and $x_{2,2}$).

Before we look at the largest Vaudenay's rake scheme for a given set of hash chains (which we will call *Vaudenay's optimal rake scheme*), we will first explore a particular family of Vaudenay's rake one-time signature schemes.

5.2.2.2 Constant sum Vaudenay's rake one-time signature schemes

In this section we examine a family of Vaudenay's rake one-time signature schemes. We look at some research by Bleichenbacher and Maurer [10] who suggested a method to compute the size of these signature schemes, but did not give a proof. We present a new method by which the size of the signature schemes can be computed, and provide a proof of correctness. Finally we give a result from [10] which shows that this family contains the largest signature scheme for any given number and length of hash chains.

We start by presenting the family of signature schemes and, since we could not find it in the literature, we also provide a proof that these are indeed signature schemes.

Theorem 5.4. *For a set of r hash chains seeded by $\{x_{0,0}, \ldots, x_{r-1,0}\}$ and each of length l, we can form a signature scheme by including all signature patterns $\{x_{0,s_0}, \ldots, x_{r-1,s_{r-1}}\}$ satisfying:*

$$\sum_{i=0}^{r-1} s_i = k, \tag{5.1}$$

where k is a constant.

Proof Firstly we note that any set of vertices $\{x_{0,s_0}, \ldots, x_{r-1,s_{r-1}}\}$ is a signature pattern, because there is exactly one vertex in it from each hash chain.

For the set of signature patterns to be a signature scheme they must also satisfy the property that, for any two signature patterns $\{x_{0,s_0}, \ldots, x_{r-1,s_{r-1}}\}$ and $\{x_{0,s'_0}, \ldots, x_{r-1,s'_{r-1}}\}$, there exist hash chains $x_{c,*}$ and $x_{c',*}$ such that x_{c,s_c} is not on the path from $x_{c,0}$ to x_{c,s'_c} and $x_{c',s'_{c'}}$ is not on the path from $x_{c',0}$ to $x_{c',s_{c'}}$. This is equivalent to saying that for any two signature patterns there exist c and c' such that $s_c > s'_c$ and $s'_{c'} > s_{c'}$. Since the two ordered sets of values s_* and s'_* are not identical, there must exist c such that, without loss of generality, $s_c > s'_c$. Given this, and that $\sum_{i=0}^{r-1} s_i = k = \sum_{i=0}^{r-1} s'_i$, there must also exist c' such that $s'_{c'} > s_{c'}$. ∎

We will refer to the signature scheme on r hash chains of length l formed by all signature patterns $\{x_{0,s_0}, \ldots, x_{r-1,s_{r-1}}\}$ satisfying (5.1) as $S(r, l, k)$. We are mainly interested in the size of $S(r, l, k)$.

In [10] Bleichenbacher and Maurer gave, without proof, an equation to compute $|S(r, l, k)|$:

$$|S(r, l, k)| = \sum_{j=0}^{\lfloor k/(l+1) \rfloor} (-1)^j \binom{r}{j} \binom{r + k - j(l+1) - 1}{r - 1}. \tag{5.2}$$

Since no proof of (5.2) was provided (and since we have not been able to create a proof ourselves) we present another method to compute $|S(r,l,k)|$, for which there is a fairly straightforward proof of correctness.

Theorem 5.5. *For a signature scheme $S(r,l,k)$ as described above, the following relations hold:*

$$|S(1,l,k)| = \begin{cases} 1 \text{ if } k \in [0,l] \\ 0 \text{ otherwise.} \end{cases} \tag{5.3}$$

$$|S(r,l,k)| = \sum_{i=0}^{\min(l,k)} |S(r-1,l,k-i)| \text{ if } r \geq 2 \tag{5.4}$$

Proof In (5.3) we are considering signature schemes with only one hash chain. We observe that all signature patterns contain only one value $\{x_{0,s_0}\}$, and that s_0 is restricted to values in $[0,l]$. Consequently if $k \in [0,l]$ then $|S(1,l,k)| = 1$, otherwise $|S(1,l,k)| = 0$.

To prove (5.4), we rewrite (5.1) to get:

$$k = s_{r-1} + \sum_{i=0}^{r-2} s_i,$$

and note that we are interested in the number of combinations of values of s_* which add to make k. The number of ways to make $k - s_{r-1}$ with $r-1$ chains has already been computed (by induction this is true, as (5.3) implies it is true for $r = 2$), and so we can sum the combinations over the different values of s_{r-1}. Hence,

$$|S(r,l,k)| = \sum_{s_{r-1}=0}^{l} |S(r-1,l,k-s_{r-1})|.$$

However, when $s_{r-1} > k$, there are no ways to add non-negative values and get a negative value. Consequently, we get (5.4):

$$|S(r,l,k)| = \sum_{i=0}^{\min(l,k)} |S(r-1,l,k-i)| \text{ if } r \geq 2.$$

■

From (5.3) and (5.4) we can compute $|S(r, l, k)|$ for all $r \geq 1$, $l \geq 0$ and $k \geq 0$.

In [10] Bleichenbacher and Maurer used the Sperner property from [132] to show that the signature scheme $S(r, l, \lfloor \frac{rl}{2} \rfloor)$ is the largest signature scheme that can be formed on r chains of length l. We will refer to the family $S(r, l, \lfloor \frac{rl}{2} \rfloor)$ as *Vaudenay's optimal rake one-time signature scheme.*

5.2.2.3 Vaudenay's optimal rake one-time signature scheme

Signature Scheme 5.7: Given a set of r hash chains of length l, $\{x_{0,0}, \ldots, x_{0,l}\}$, $\ldots, \{x_{r-1,0}, \ldots, x_{r-1,l}\}$, Vaudenay's optimal rake one-time signature scheme is the specific instance of Signature Scheme 5.6 where the set of signature patterns is defined to be every signature pattern of the form $\{x_{0,s_0}, \ldots, x_{r-1,s_{r-1}}\}$ satisfying:

$$\sum_{i=0}^{r-1} s_i = \left\lfloor \frac{rl}{2} \right\rfloor .$$

Although Signature Scheme 5.7 is the largest signature scheme on a given set of hash chains, it is not necessarily the best scheme to use in practice. To use Vaudenay's rake one-time signature scheme it is also necessary to have an efficient signature pattern function. There is currently no known 'efficient' mapping for Vaudenay's scheme.[3]

5.2.2.4 Further optimisation

Using Signature Scheme 5.7 we can, for given r and l, find the largest signature scheme possible. However, in practice we are more likely to know the size of our message space and want to know how large r and l must be. One way to handle this is by trial and improvement; try a pair of values r and l, find the largest signature scheme, adjust r and l, then repeat.

Even if we find a small pair of values r and l which provide us with a large enough signature scheme, it is likely that we will get more than one pair. For example, if we wanted to be able to provide a signature on a message space

[3] We have seen no signature pattern functions in the literature. The best such function that we know of is a look-up table.

of size 50 then we could choose 8 chains of length 1, 5 chains of length 2, 4 chains of length 4, 3 chains of length 8 or 2 chains of length 49. We give the maximum size of signature scheme for each of these pairs of values in Table 5.2.

r	l	$\lfloor \frac{rl}{2} \rfloor$	$\lvert S(r, l, \lfloor \frac{rl}{2} \rfloor) \rvert$
8	1	4	70
5	2	5	51
4	4	8	85
3	8	12	61
2	49	49	50

Table 5.2: Some Vaudenay's optimal rake signature schemes that cater for a message space of size at least 50.

Choosing the 'best' signature scheme to use from this list is an application dependent problem. For example:

- if we wish to minimise the size of the private and public keys then we must minimise r, and so $S(2, 49, 49)$ is the best choice;
- if we wish to minimise the storage for the signing party then we should minimise the number of vertices in the chains, and so $S(5, 2, 5)$ is the best choice;
- if we wish to minimise the number of hash function calculations that are needed to compute the values for a signature (or equivalently the number of hash applications to verify a signature) then we should minimise $\lfloor \frac{rl}{2} \rfloor$, and so $S(8, 1, 4)$ is the optimal scheme.

Although it is clear that there cannot be a universal measure of efficiency suitable for all applications, work has been done to identify reasonable measures. In Section 5.2.4.1 we will discuss in more detail a measure of efficiency due to Bleichenbacher and Maurer [11].

5.2.3 Using hash trees and Merkle trees for one-time signature schemes

In this section we look at two applications of Merkle trees to one-time signature schemes. The first is in the reduction of the public key size, at the expense of more computation during the initialisation and verification phases. The second application is to use the Merkle tree as the basis of the one-time signature scheme itself, which can reduce the size of the signature from the schemes in earlier sections.

5.2.3.1 Reducing public and private key sizes

All of the one-time signature schemes that we have looked at so far require the signer to store a collection of private keys. It is possible to generate all the keys from a single master key, which greatly reduces the amount of storage required for the signer. We can use a pseudo-random number generator seeded with the master key to generate a set of independent private keys.[4]

As we discussed in Section 3.3.2, there are many ways of generating independent values from a master value using a hash function. If we use one of these with one of the hash-based signature schemes described earlier then the overall structure formed is an inverted hash tree (see Section 3.4.5).

Similarly, most one-time signature schemes require each user to publish a collection of public keys. The public key values can be reduced to one master public value by hashing all the public key values together. If we use a strong hash function (Definition 2.10) then the verifier can have the same confidence in the signature by checking the collection of public key values is correct, as by checking the master public value is correct.

If we use one of the one-time signature schemes described earlier in this chapter and then hash the public key values to form a master public key, then the overall structure used is a Merkle tree (see Definition 3.4). In the next section we will look at other Merkle tree based one-time signature schemes.

[4]Independent in the sense that no information can be derived about one of the private keys from any subset of the others. Trivially, the private keys are not independent of the master key.

These reductions add extra computation to the signature generation and verification algorithms, but greatly reduce both the storage requirement for the private key and the message costs of distributing the public key.

5.2.3.2 Merkle tree based one-time signature schemes

Merkle trees can also be used as the basis of a one-time signature scheme, in a similar way to a set of hash chains. In Section 5.2.2.1 we gave some definitions which helped to discuss signature schemes based on hash chains. We now give some similar definitions for hash trees.

Definition 5.6. A **signature pattern** for a one-time signature scheme based on a hash tree is a set of values U_i from the hash tree such that each path from a leaf to the root contains exactly one value from U_i.

Definition 5.7. A **one-time signature scheme** based on a hash tree is a set of signature patterns such that for any two signature patterns U_i and U_j there exist two paths P_k and $P_{k'}$ satisfying the following two conditions:

1. The vertex $U_i \cap P_k$ is not on the path from $U_j \cap P_k$ to the root;
2. The vertex $U_j \cap P_{k'}$ is not on the path from $U_i \cap P_{k'}$ to the root.

We note that Definition 5.3 can be applied to hash chains or to hash trees.

As an example, the Merkle tree in Figure 5.5 has a signature pattern marked on it, and there are a total of 4 such signature patterns which together form a signature scheme.

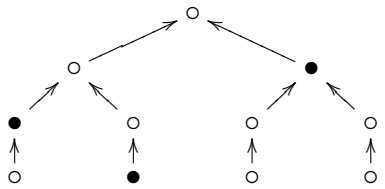


Figure 5.5: An example of a signature pattern on a Merkle tree is marked by the solid circles.

In fact one-time signature schemes can be created on any type of hash tree, not just Merkle trees. However, if the hash tree is not a Merkle tree

then some of the leaves can be deleted to form a Merkle tree with one-time signature schemes of the same size. We now present a construction by which this can be achieved (Theorem 5.8) and then we give a proof that the Merkle tree admits a signature scheme of the same size (Theorem 5.9).

Theorem 5.8. *For any hash tree on a tree $\mathcal{T}$ with leaves $\mathcal{L}$ and non-leaves $\mathcal{N}$, we can define a subtree $\mathcal{M}$ by:*

$$\mathcal{M} = \mathcal{N} \cup \{v \in \mathcal{L} : v \textit{ has no siblings on its left, and no siblings in } \mathcal{N}\}.$$

The hash tree on $\mathcal{M}$ is a Merkle tree.

Proof In Definition 3.4 we define a Merkle tree as a hash tree where every leaf has no sibling.

It can be seen that any leaf in $\mathcal{L}$ that is also in $\mathcal{M}$ has no siblings.

There are no leaves in $\mathcal{M} \cap \mathcal{N}$, as every vertex in $\mathcal{N}$ either has children in $\mathcal{N} \cap \mathcal{M}$ or a child in $\mathcal{L} \cap \mathcal{M}$. ■

Figure 5.6 gives a hash tree and the Merkle sub-tree formed by the method in Theorem 5.8. Also shown is a signature pattern on the hash tree, and the equivalent signature pattern on the Merkle tree formed from the hash tree. We now give a formal analysis of the equivalence of signature patterns on hash trees and signature patterns on Merkle sub-trees. This result is implied for *binary* hash trees during the study of tree-based digital signatures by Bleichenbacher and Maurer [12], but we present it here as they did not explicitly provide it, nor did they imply the result on general hash trees.

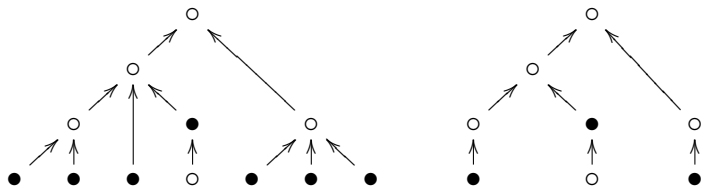


Figure 5.6: An example showing a signature pattern on a hash tree and the equivalent signature pattern on the corresponding Merkle tree. The signature patterns are marked by the solid circles.

Theorem 5.9. *For any signature scheme on a hash tree on a tree $\mathcal{T}$ there is an equivalent signature scheme on the Merkle tree on $\mathcal{M}$ (as defined in Theorem 5.8) with the same number of signature patterns.*

Proof From each signature pattern U_i in the original signature scheme we create a signature pattern U_i' on the Merkle tree by:

$$U_i' = U_i \cap \mathcal{M}.$$

To prove the theorem, we first show that U_i' is a signature pattern and then that the set of signature patterns form a one-time signature scheme.

We show that U_i' is a signature pattern. Every leaf in $\mathcal{M}$ was also a leaf in $\mathcal{T}$, as was shown in the proof of Theorem 5.8. Consequently, every path from a leaf in $\mathcal{M}$ to the root of $\mathcal{M}$ is also a path from a leaf in $\mathcal{T}$ to the root of $\mathcal{T}$. Every path from a leaf to the root in $\mathcal{M}$ must intersect U_i' at exactly the same vertex that the path intersected U_i, and no others. Hence U_i' is a signature pattern.

We now show that the set of signature patterns U_*' form a one-time signature scheme. Assume U_*' is not a signature scheme. Hence we can find two signature patterns U_i' and U_j' such that there is no pair of paths satisfying the three conditions given in Definition 5.7. Without loss of generality we assume that for every path P_k on $\mathcal{M}$ the vertex $U_i' \cap P_k$ is on the path from $U_j' \cap P_k$ to the root.

U_i and U_j are signature patterns on $\mathcal{T}$. We note that $U_i' \subseteq U_i$, $U_j' \subseteq U_j$ and all the paths in $\mathcal{M}$ are also in $\mathcal{T}$. Consequently (without loss of generality) we must have a pair of paths P_k and $P_{k'}$ satisfying the three conditions, with $U_i \cap P_{k'} \notin \mathcal{M}$.

The only vertices in $\mathcal{T} \backslash \mathcal{M}$ are leaves which have siblings, at least one of which, v, is in $\mathcal{M}$. Any signature pattern containing $U_i \cap P_{k'}$ must also contain a vertex on each path from v to the leaves descended from v. One of these vertices is also in $\mathcal{M}$, and so is also in U_i'. Hence any path P_k through v in $\mathcal{M}$ does not have $U_i' \cap P_k$ on the path from $U_j' \cap P_k$ to the root, and we have a contradiction. ∎

This theorem allows us to restrict further analysis of hash tree based signature schemes to those based on Merkle trees.

5.2.4 Generalised hash DAG one-time signature schemes

In [10], Bleichenbacher and Maurer generalise all the previous schemes by proposing a one-time signature scheme based on a generalised hash DAG. They also gave the following useful definitions for analysing generalised hash DAG one-time signature schemes.

Given a DAG $\mathcal{G} = (V, E)$, we can define the *secret key pattern* $S(\mathcal{G}) \subset V$ as the set of vertices with in-degree 0, and the *public key pattern* $P(\mathcal{G}) \subset V$ as the set of vertices with out-degree 0. The secret key values are the values associated (by the generalised hash DAG) with the secret key pattern vertices. Similarly, the public key values are those values at the vertices of the public key pattern.

Given a set of vertices $X \subset V$, we say that a vertex $v \in V$ is *computable* from X if either $v \in X$ or v has at least one parent, and all its parents are computable from X. Given two sets $X, Y \subset V$ we say Y is computable from X if all its members are computable from X. We say that a set X is *verifiable* (with respect to the public key) if $P(\mathcal{G})$ is computable from X.

Definition 5.10. A **minimal verifiable set** (or MVS) is a verifiable set X such that no proper subset of X is verifiable.

A **signature pattern** for a one-time signature scheme based on a generalised hash DAG is a minimal verifiable set.

We only wish to consider minimal verifiable sets since smaller signature patterns will equate to lower communication costs in the protocol. The set of minimal verifiable sets of a graph $\mathcal{G}$ is denoted $\mathcal{W}(\mathcal{G})$.

Two minimal verifiable sets $X, Y \in \mathcal{W}(\mathcal{G})$ are *compatible* if neither is computable from the other. A set of MVSs are compatible if they are compatible pairwise.

Remark 5.11. *A minimal verifiable set intersects every path from $S(\mathcal{G})$ to $P(\mathcal{G})$.*

Definition 5.12. A **one-time signature scheme** based on a generalised hash DAG is a compatible set of MVSs.

We briefly give a few definitions of well studied combinatorial objects that will be of use to us during the rest of this section.

Definition 5.13. A **poset** (or **partially ordered set**) is a set S, together with a binary comparison operator $\leq$. For every element $a \in S$ we have $a \leq a$. For every pair of distinct elements $a, b \in S$ we have exactly one of $a \leq b$, $b \leq a$, or a and b are incomparable. Finally if $a \leq b$ and $b \leq c$ we have $a \leq c$.

Definition 5.14. In a poset $(S, \leq)$ a **chain** is a subset of S such that for every pair of elements $a, b \in S$ we have either $a \leq b$ or $b \leq a$.

In a poset $(S, \leq)$ an **antichain** is a subset of S such that every pair of elements $a, b \in S$ are incomparable.

Definition 5.15. The **associated poset** of $\mathcal{W}(\mathcal{G})$ is the poset formed by the set $\mathcal{W}(\mathcal{G})$ and the binary comparison operator of computability. That is $a \leq b$ if a is computable from b.

The largest signature scheme that can be formed from a particular generalised hash DAG is the largest compatible set of minimal verifiable sets. This is equivalent to the largest antichain in the associated poset of $\mathcal{W}(\mathcal{G})$, and the size of this antichain is the *width* of $\mathcal{W}(\mathcal{G})$, which we will denote by $w(\mathcal{W}(\mathcal{G}))$.

The authors of [10] define two functions $\nu(n)$ and $\mu(n)$, which are the maximal number of MVSs for a DAG on n vertices, and the maximum size of signature scheme that can be formed on a graph with n vertices. The functions are defined as:

$$\nu(n) = \max\{|\mathcal{W}(\mathcal{G})| : \mathcal{G} = (V, E) \text{ with } |V| = n\}; \tag{5.5}$$

$$\mu(n) = \max\{w(\mathcal{W}(\mathcal{G})) : \mathcal{G} = (V, E) \text{ with } |V| = n\}. \tag{5.6}$$

We define a DAG $\mathcal{G}$ of order n to be an **optimal DAG on n vertices** if $w(\mathcal{W}(\mathcal{G})) = \mu(n)$.

In [11], the same authors prove an interesting relation between $\mu(n)$ and $\nu(n)$ which holds for all $n \geq 1$.

Result 5.16. *Equation 1 of Bleichenbacher and Maurer [11]:*

$$\frac{\nu(n)}{n} \leq \mu(n) \leq \nu(n).$$

The lower and upper bounds in Result 5.16 are attained respectively by a hash chain on n vertices, and a graph of n unconnected vertices.

It is clear that by generalising hash chains and hash trees to generalised hash DAGs we may find schemes with the same number of vertices but more signatures. An example is given in Figure 5.7, which shows, for nine vertices, the tree and DAG which admit the largest signature scheme (as found by exhaustive search).

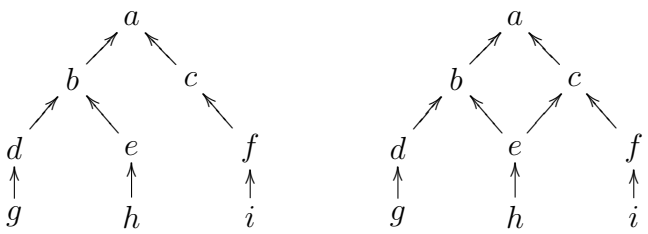


Figure 5.7: An optimal tree and an optimal DAG on 9 vertices. The optimal signature scheme for the tree contains 4 signature patterns $(\{g, h, c\}, \{d, h, f\}, \{g, e, f\}, \{d, e, i\})$, whereas the optimal signature scheme for the DAG contains 5 signature patterns $(\{g, h, c\}, \{d, h, f\}, \{g, e, f\}, \{d, e, i\}, \{b, h, i\})$.

5.2.4.1 Efficient schemes

The authors of [11] also focus on finding efficient schemes. To compare the efficiency of schemes they focus on the underlying DAG. To simplify things further they insist that every scheme should have only one public key (as discussed in Section 5.2.3.1).

They also insist that every vertex in the DAG has in-degree at most two. This can be achieved by replacing vertices with in-degree greater than two with a binary hash tree, and replacing any edges that left the old vertex with edges that leave the root of this new tree. We give an example in Figure 5.8.

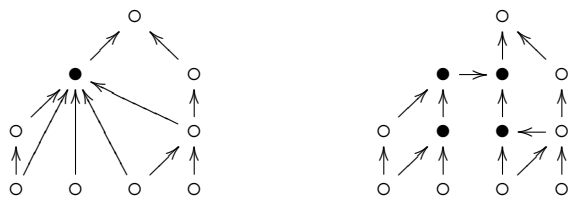


Figure 5.8: On the left is a DAG where the filled in vertex has in-degree greater than two. It can be replaced by a binary hash tree, as shown in the diagram on the right, to ensure that all vertices now have in-degree of at most two.

We assume that all the seed vertices are formed from a single private key, as described in Section 5.2.3.1; however this vertex is not included as part of the DAG for the purposes of counting the vertices.

With this model, all vertices in a generalised hash DAG except the single seed vertex require a hash function application to compute. Consequently, Bleichenbacher and Maurer [11] approximate the amount of 'work' needed to use a particular one-time signature scheme by the number of vertices it contains. They approximate the payoff of the signature scheme's DAG as the size of the largest one-time signature scheme based on it (that is the size of the largest compatible set of minimal verifiable sets).

For a generalised hash DAG of order n, they define the efficiency $\eta(\Gamma)$ of a one-time signature scheme based on a set Γ of minimal verifiable sets to be

$$\eta(\Gamma) = \frac{\log_2 |\Gamma|}{n+1}. \tag{5.7}$$

As an example, consider the Diffie-Lamport scheme for k-bit messages with a binary hash tree to make a single public key value (see Figure 5.9).

The scheme is designed to sign k-bit messages and so has minimal verifiable sets for each of the 2^k messages, that is $|\Gamma| = 2^k$. To calculate the efficiency we also need to find n, the number of vertices, of which there are $6k - 1$. The efficiency $\eta(\Gamma)$ can be computed as follows:

$$\eta(\Gamma) = \frac{\log_2 |\Gamma|}{n+1} = \frac{\log_2 2^k}{6k} = \frac{1}{6}. \tag{5.8}$$

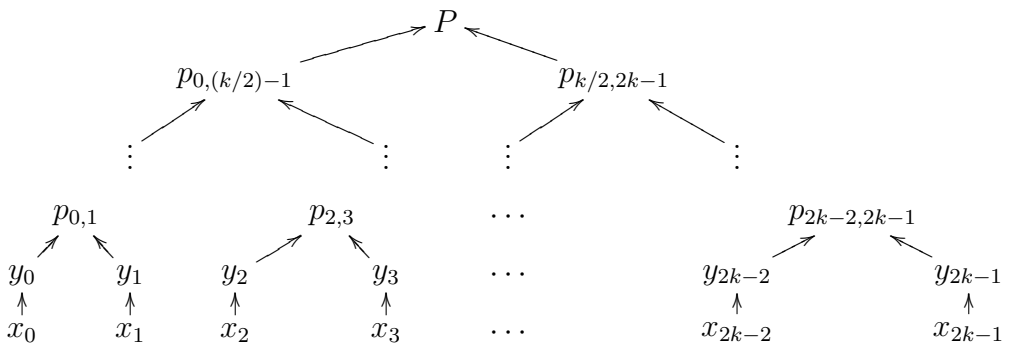


Figure 5.9: The Diffie-Lamport one-time signature scheme for k-bit messages with a binary hash tree to combine all the public key values to a single value.

The authors also prove that the maximum efficiency of a one-time signature scheme based on a hash tree is less than 0.42 and give a construction of a sequence of signature schemes based on a sequence of generalised hash DAGs with

$$\lim_{n\to\infty} \eta(\Gamma_n) > 0.47. \tag{5.9}$$

In Section 5.4.1 we will look in more detail at this construction and in Sections 5.4.2 and 5.4.3 we will give some constructions of our own.

First we present a method by which $\nu(n)$ and $\mu(n)$ can be investigated. In Section 5.3.2 we exhibit a collection of algorithms that finds the associated poset of a DAG $\mathcal{G}$, and hence gives us a lower bound on $\nu(n)$ for small n. In Section 5.3.3 we present several algorithms that find the largest signature scheme on $\mathcal{G}$, and consequently give us a lower bound on $\mu(n)$.

5.3 Finding the largest compatible set of minimal verifiable sets for a given graph

In this section we will present several algorithms that build towards an algorithm that efficiently finds and outputs a large one-time signature scheme on a given DAG $\mathcal{G}$. In Section 5.2.4 we saw that this was the same as finding a large compatible set of minimal verifiable sets for $\mathcal{G}$.

We start this section by explaining how we split the task of finding a large set of compatible minimal verifiable sets from a DAG $\mathcal{G}$ into two smaller problems. We then present algorithms that build towards several solutions to the first problem and we compare the efficiency of these algorithms. We then look at algorithms to solve the second problem, and again study their performance. Finally we present the results obtained by implementing our algorithms.

5.3.1 Introduction

We are interested in $\mu(n)$, the maximum size of a one-time signature scheme on a DAG of order n. By trying many graphs, our program finds a lower bound on this, which we will call $m(n)$. For each signature scheme, our program can optionally also output the DAG $\mathcal{G}$, the set of MVSs $\mathcal{W}(\mathcal{G})$ and the signature patterns $\Gamma(\mathcal{G})$ that make up the signature scheme found.

The method by which we select graphs is itself of interest. Originally we tested randomly chosen graphs, edges added randomly with the restriction that no vertex had an in-degree greater than k. We were not satisfied by the rate at which $m(n)$ increased, and so we redesigned the graph selection algorithm to keep track of the best graph found so far, and to make small changes to that graph. This significantly improved the rate at which $m(n)$ increased.

A list of all the algorithms we present in this section is given in Table 5.3.

The problem of finding a large set of compatible minimal verifiable sets from a DAG $\mathcal{G}$ seems to split logically into two problems:

1. Finding the set of minimal verifiable sets $\mathcal{W}(\mathcal{G})$ from the DAG;

2. Finding a large set of compatible minimal verifiable sets $M(\mathcal{G})$ from $\mathcal{W}(\mathcal{G})$ (and $\mathcal{G}$).

Algorithm	Description
Algorithm 2	An algorithm to check whether a given subset U of $\mathcal{G}$ is a minimal verifiable set or not.
Algorithm 3	An algorithm to check whether a given subset U of $\mathcal{G}$ is a verifiable set.
Algorithm 4	An algorithm to check whether a given verifiable subset U of $\mathcal{G}$ is minimally verifiable or not.
Algorithm 5	An algorithm to output all the paths in a DAG $\mathcal{G}$ with source set $S(\mathcal{G})$ and sink set $P(\mathcal{G})$.
Algorithm 6	A trivial but inefficient exhaustive search algorithm to find all the minimal verifiable sets in a DAG $\mathcal{G}$.
Algorithm 7	A second trivial but inefficient exhaustive search algorithm to find all the minimal verifiable sets in a DAG $\mathcal{G}$ with paths Paths($\mathcal{G}$).
Algorithm 8	An algorithm to output all minimal verifiable sets that is very often more efficient than Algorithms 6 and 7.
Algorithm 9	An algorithm to construct the associated poset on the set of minimal verifiable sets MVS($\mathcal{G}$) ordered by computability.
Algorithm 10	The Bron-Kerbosch algorithm - a reasonably efficient recursive algorithm to find the largest compatible set of MVSs for a DAG $\mathcal{G}$.
Algorithm 11	A very efficient algorithm to find a large compatible set of MVSs for a DAG $\mathcal{G}$.

Table 5.3: A list of the algorithms presented in this section. Those in the top half are found in Section 5.3.2 and those in the bottom half are found in Section 5.3.3.

5.3.2 Problem 1 — Finding the set of minimal verifiable sets from the DAG

We can check if a subset U of $\mathcal{G}$ is a minimal verifiable set by checking that it contains at least one vertex from each path from a source of $\mathcal{G}$ to a sink (i.e. it is verifiable), and by checking that each vertex in the subset is the only vertex of the subset present in at least one path (i.e. the subset is minimal). This straightforward scheme is given in Algorithm 2, which makes calls to Algorithms 3 and 4.

Algorithm 2:
Description: An algorithm to check whether a given subset U of $\mathcal{G}$ is a minimal verifiable set or not.

```
IS_SET_MINIMAL_AND_VERIFIABLE(U)
(1)  if IS_SET_VERIFIABLE(U) and IS_SET_MINIMAL(U)
(2)      return true
(3)  else
(4)      return false
```

Running time: $O(|U| \cdot |Paths(\mathcal{G})|)$.

Algorithm 3:
Description: An algorithm to check whether a given subset U of $\mathcal{G}$ is a verifiable set.

```
IS_SET_VERIFIABLE(U)
(1)  foreach path P in G
(2)      if P and U have no vertices in common
(3)          return false
(4)  return true
```

Running time: $O(|Paths(\mathcal{G})|)$.

It is clear that Algorithms 3 and 4 require us to have a list of all the paths from sources to sinks in $\mathcal{G}$. Algorithm 5 is a classic depth first search algorithm [150] for finding paths in a graph $\mathcal{G}$ with source set $S(\mathcal{G})$ and sink set $P(\mathcal{G})$.

Algorithm 4:
Description: An algorithm to check whether a given verifiable subset U of $\mathcal{G}$ is minimally verifiable or not.

```
IS_SET_MINIMAL(U)
(1)  foreach vertex u in U
(2)     Initialise flag VertexIsCritical to false
(3)     foreach path P in Paths(G)
(4)        if P ∩ U = {u}
(5)           Set VertexIsCritical to true
(6)     if not VertexIsCritical
(7)        return false
(8)  return true
```

Running time: $O(|U| \cdot |Paths(\mathcal{G})|)$.

Algorithm 5:
Description: An algorithm to output all the paths in a DAG $\mathcal{G}$ with source set $S(\mathcal{G})$ and sink set $P(\mathcal{G})$.

FIND_PATHS($\mathcal{G}$, $S(\mathcal{G})$, $P(\mathcal{G})$)

See Appendix A for the full algorithm.

To find every minimal verifiable set in $\mathcal{G}$ we could simply try all subsets of $\mathcal{G}$ and check each with Algorithm 2. This naive approach is given in Algorithm 6. However the number of sets to be checked for each DAG is 2^n, which proved prohibitively large by about $n = 10$.

Algorithm 6:
Description: A trivial but inefficient exhaustive search algorithm to find all the minimal verifiable sets in a DAG $\mathcal{G}$.

```
FIND_MINIMAL_VERIFIABLE_SETS_1(G)
(1)  foreach subset U of G
(2)     if IS_SET_MINIMAL_AND_VERIFIABLE(U)
(3)        print U
```

Running time: $O\left(2^{|\mathcal{G}|} \cdot |\mathcal{G}| \cdot |Paths(\mathcal{G})|\right)$.

Another way to find all minimal verifiable sets is to form subsets of $\mathcal{G}$ by taking one vertex from each path and checking with Algorithm 4 to see if the set is minimal (it will trivially be verifiable). This method is given in

Algorithm 7. Unfortunately this too becomes a prohibitive method when we are dealing with DAGs that have five or six paths of length about three or four vertices.

Algorithm 7:
Description: A second trivial but inefficient exhaustive search algorithm to find all the minimal verifiable sets in a DAG $\mathcal{G}$ with paths Paths($\mathcal{G}$).

```
FIND_MINIMAL_VERIFIABLE_SETS_2(Paths(G))
(1)  Create a set Pointers with a pointer for each path.
(2)  foreach combination of Pointers with the ith pointer
     pointing to a vertex from the ith path
(3)      Create a set U containing the union of all vertices
         pointed to by Pointers.
(4)      if IS_SET_MINIMAL(U)
(5)          print U
```

Running time: $O\left(PathLen(\mathcal{G})^{|Paths(\mathcal{G})|} \cdot |\mathcal{G}| \cdot |Paths(\mathcal{G})|\right)$, where $PathLen(\mathcal{G})$ is the average path length in $\mathcal{G}$.

We devised a method of combining elements of these two methods of searching to create a much more efficient exhaustive search. There are two important observations.

1. If a DAG has a vertex v that is in almost all of the paths then we can find all the minimal verifiable sets containing v very quickly, and then we can ignore v for the rest of the search.

2. If a DAG has a very short path P then since all minimal verifiable sets intersect P, we can consider every subset U of the vertices in P, and then only consider paths that do not intersect U.

The actual algorithm we have used is recursive in order to optimise the benefit of the two above observations. It is given in Algorithm 8.

The best way to illustrate these two observations is with an example, so we will consider the DAG in Figure 5.10. We need to have an ordered list of the paths in our DAG, so these are given in Table 5.4.

Algorithm 8:
Description: An algorithm to output all minimal verifiable sets that is very often more efficient than Algorithms 6 and 7.
Input: A list of all the paths in $\mathcal{G}$, an empty set (PartialMVS).
Output: A list of all the minimal verifiable sets in $\mathcal{G}$.
FIND_MINIMAL_VERIFIABLE_SETS(Paths, PartialMVS)
See Appendix A for the full algorithm.

Algorithm 6 would require us to test all subsets of $\mathcal{G}$ and check each with Algorithm 2. Since there are seven vertices, this would take approximately $2^7 = 128$ calls to Algorithm 2.

Algorithm 7 would require us to check all sets formed by taking one vertex from each path with Algorithm 4. Although this algorithm is marginally quicker than Algorithm 2, we need to check $2 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \cdot 4 = 8192$ subsets, and so this method would almost certainly be slower than Algorithm 6.

However, if instead we use Algorithm 8 then we only end up checking 12 sets with Algorithm 2.[5] This is much faster than either of the aforementioned methods.

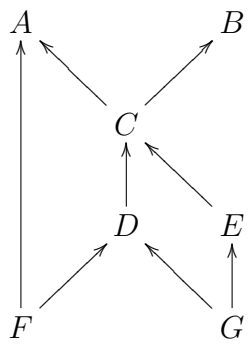


Figure 5.10: An example DAG.

[5]These sets in order are CF, CA, C, AB, ADE, ADG, AD, AFE, AFG, AF, FG, FDE. This assumes that, if the algorithm allows a choice, the paths are selected lowest index first and vertices are chosen lexicographically.

Path index	Vertices in path
0	FA
1	FDCA
2	FDCB
3	GDCA
4	GDCB
5	GECA
6	GECB

Table 5.4: The paths in the example DAG in Figure 5.10.

5.3.3 Problem 2 — Finding a large compatible set of minimal verifiable sets from the set of all minimal verifiable sets

Once we have found the set of all MVSs $\mathcal{W}(\mathcal{G})$ for a given DAG $\mathcal{G}$, the next task is to find a large compatible subset of $\mathcal{W}(\mathcal{G})$. Ideally we would like to find the largest compatible subset, but we also wish our program to finish in a sensible length of time.

For the purposes of this section it will be good to have a graph by which we can compare performances of different algorithms. We have chosen to use the graph given in Figure 5.11, since it is not too hard to analyse. For a 13 vertex graph it does not have an extreme number of minimal verifiable sets (54), nor does it permit an overly large one-time signature scheme (size 12).

5.3.3.1 Discussion of Algorithm 9

The first step in finding a large set of compatible sets is to construct the associated poset of the DAG. The associated poset of $\mathcal{G}$ consists of the set of minimal verifiable sets and the relation of whether one MVS is computable from another (see Section 5.2.4 for definitions).

We note that two members $i, j \in \text{MVS}(\mathcal{G})$ must satisfy exactly one of the following four conditions: $i = j$, $i < j$, $i > j$ or i and j are incomparable. If $i = j$ then i and j must be same element. If they are distinct and there is no path P for which $i \cap P$ is nearer the source than $j \cap P$ then $i < j$. If $j < i$

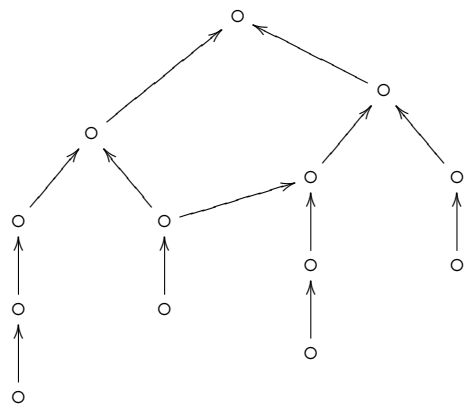


Figure 5.11: An example graph with 13 vertices. This graph contains 54 minimal verifiable sets, of which at most 12 can combined to form a compatible set.

then we have $i < j$ and finally if none of the other conditions are satisfied then i and j are incomparable.

Algorithm 9 uses this observation to determine the relationship between each pair of elements in MVS($\mathcal{G}$).

Algorithm 9:
Description: An algorithm to construct the associated poset on the set of minimal verifiable sets MVS($\mathcal{G}$) ordered by computability.
Input: A list of the minimal verifiable sets of $\mathcal{G}$, a list of the paths in $\mathcal{G}$.
Output: An array containing the poset relations between the members of MVS($\mathcal{G}$).
FIND_ASSOCIATED_POSET(MVS($\mathcal{G}$), Paths($\mathcal{G}$))
See Appendix A for the full algorithm.

Once we have identified which minimal verifiable sets are computable from each other, and which are compatible with each other, we are ready to attempt to construct a large compatible set of minimal verifiable sets.

5.3.3.2 Discussion of Algorithm 10

Finding the largest compatible set of MVSs on a DAG is the same as the maximum clique problem [59] on the graph with MVSs as vertices and edges between pairs of MVSs that are compatible.

In [18] Bron and Kerbosch present an algorithm for finding the largest clique in a graph. This method is given in Algorithm 10.

Algorithm 10:
Description: The Bron-Kerbosch algorithm to find the largest clique $\mathcal{G}$ [18]. Initially the algorithm is called as BK($\emptyset, V(\mathcal{G}, \emptyset)$. The function $\Gamma(v)$ returns the neighbours of the vertex v.
Input: Three sets of vertices R, P and X
Output: A list C containing the vertices in a maximal clique.
BK(MVS($\mathcal{G}$), Poset[MVS($\mathcal{G}$),MVS($\mathcal{G}$)])

```
(1)   Set C = ∅.
(2)   if P = ∅ and X = ∅
(3)
(4)      if |R| > |C|
(5)         Set C = R.
(6)   else
(7)      foreach v ∈ P
(8)         P = P − {v}
(9)         R_new = R ∪ {v}
(10)        P_new = P ∩ Γ(v)
(11)        X_new = X ∩ Γ(v)
(12)        BK(R_new, P_new, X_new)
(13)        X = X ∪ v
(14)  print C
```

Our experiments show Algorithm 10 to be a useful algorithm on graphs with up to 14 vertices, and all but the best graphs on 15 and 16 vertices.[6] For some 15 vertex DAGs with around about 100 MVSs, the algorithm was still running after an hour.

[6]Here we use 'best graphs' to mean those graphs that admit the largest signature schemes.

5.3.3.3 Discussion of Algorithm 11

If a large compatible set A of minimal verifiable sets is compatible with another minimal verifiable set b then we can create a larger compatible set $A \cup b$. Consequently we can form an estimate of how good a minimal verifiable set is by measuring the number of minimal verifiable sets that it is compatible with.

We can create a subset iteratively by including, at each iteration, the MVS which is compatible with the largest number of MVSs, until there are no more MVSs to include in the subset. This algorithm is presented as Algorithm 11. It is a greedy algorithm, because we only look one step ahead at each step.

Algorithm 11 is more efficient than Algorithm 10. We can approximate the number of computations needed to find a large compatible set in this way by looking at the line which is repeated the most times. Line 15 of Algorithm 11 is the line inside more loops than any other (one 'while' loop and three 'foreach' loops). For a DAG with 54 minimal verifiable sets and maximal compatible verifiable set of order 12, this line is inside loops of size at most $12, 54, 54$ and 12 (Lines 3, 5, 12 and 14 respectively). Multiplying these values together, we find that Line 15 is visited no more than 4.2×10^5 times,[7] which would take about 0.4 seconds at a speed of a million computations a second.

5.3.3.4 Comparison of algorithms for finding large compatible sets

We have presented two algorithms for finding large compatible sets of MVSs for a DAG $\mathcal{G}$. Algorithms 10 and 11 are both of use to us, since they provide a trade-off between size of output set and running time.

Algorithm 10 takes an excessive time for some graphs of order 15 and increasingly so for many graphs on more vertices. Although we have found some DAGs for which Algorithm 11 does not find the maximal compatible set, they have all been too large to evaluate with Algorithm 10.

[7] We measured this with our implementation of Algorithm 11 for the DAG given in Figure 5.11 and found that the program visited Line 15 about 5300 times. It is clear that our estimation of algorithm complexity is generous towards rival algorithms.

Algorithm 11:
Description: A greedy algorithm to find a large compatible set of minimal verifiable sets on a DAG $\mathcal{G}$. The constant INCOMPARABLE is defined in Appendix A.

```
FIND_A_LARGE_COMPATIBLE_SET(MVS(G),
Poset[MVS(G),MVS(G)])
(1)   Set MVSsLeft = |MVS(G)|.
(2)   Create an empty set CompSet.
(3)    while MVSsLeft > 0
(4)      BestScore = -1
(5)       foreach MVS i
(6)         Set flag Incomparable to true
(7)         foreach MVS k from the set CompSet
(8)           if Poset[i, k] differs from INCOMPARABLE
(9)             Set flag Incomparable to false
(10)        if Incomparable = true
(11)          Set Score to 0
(12)           foreach MVS j
(13)             Set flag Comparable to true
(14)              foreach MVS k from the set CompSet
(15)                if Poset[j, k] differs from INCOMPARA-
                    BLE
(16)                  Set flag Incomparable to false
(17)             if Incomparable = true and Poset[i, j] =
                 INCOMPARABLE
(18)               Increment Score by 1
(19)          if Score > BestScore
(20)             BestScore = Score
(21)             BestMVS = i
(22)     Add BestMVS to the set CompSet
(23)     Set MVSsLeft to BestScore
```

We also note that in Section 5.4.3, where we exhibit a one-time signature scheme based on a hash DAG, we do not use all of the MVSs found in order to simplify the signature pattern function.

5.3.4 Results

The results in Tables 5.5, 5.6 and 5.7 have been produced by a program using Algorithms 5, 8, 9 and 11. The program tested a succession of graphs and output information each time it improved the value of $m(n)$ (defined in Section 5.3.1). The first graph was either selected randomly by the program, or entered by hand. We used a hill-climbing method of making small changes to the graph while maintaining the maximum in-degree, and if a graph improved on the value of $m(n)$ then it replaced the original graph, and future graphs would be derived from it.

Due to the random nature of the DAGs tested by our program, there is always a chance that by testing one more graph we will find an improvement on the previously found signature scheme. Consequently we made a decision to run the program for ten minutes for each value of n and output all the DAGs found.

For reasons discussed in Section 5.2.4.1, Bleichenbacher and Maurer only consider DAGs with in-degree at most two. In [10] they presented a list of the size of the largest one-time signature schemes they had found for DAGs with a fixed order n.

Using our program we have been able to improve on a number of their results. In fact, for all values of $n \geq 13$ we have improved on the size of the best signature scheme known. These results are presented in Table 5.5.

This table also shows a number of other trends. By plotting the results with a spreadsheet application and asking for a line of best fit, we find the approximate results: $|\mathcal{W}(\mathcal{G})| \approx 0.75e^{0.34n}$ and $m(n) \approx 0.37e^{0.29n}$. From Result 5.16, we can see that if $\nu(n)$ and $\mu(n)$ are functions of the form ae^{bn} then the constant b must be the same for both $\nu(n)$ and $\mu(n)$.

We then investigated the value of $m(n)$ for graphs with maximum in-degree three. In Table 5.6 we present our results, along with the values

of $m(n)$ from Table 5.5 for maximum in-degree two. Unsurprisingly, we can find larger signature schemes on the same number of vertices if we allow the in-degree to be increased.

From our results we get the approximate relation $m(n) \approx 0.40e^{0.30n}$ for DAGs with in-degree three. We note that this is similar to our approximation for in-degree two DAGs, the main difference being the linear factor has increased.

In Table 5.6, and also in Table 5.7, there is a bracketed row for which our program could not find the best signature scheme from $\mathcal{W}(\mathcal{G})$. Instead we discovered this DAG by hand. We note that if we had been using Algorithm 10 instead of Algorithm 11 then the program would not have overlooked these results, but it may not have found them in the allotted time.

We went on to investigate the value of $m(n)$ of DAGs with the maximum in-degree increased to four, the results of which are in Table 5.7. Since this includes all the DAGs previously tested, it might be expected that the value of $m(n)$ found would be larger than in the previous tables. When there are 11 or fewer vertices, the larger in-degree did not make an impact on $m(n)$. At 12 vertices our program found a DAG that admitted a signature scheme larger than any previously tested 12 vertex DAG.

However for DAGs with 13 vertices or more, our program did not manage to find improvements on any of the results in Table 5.6, and by 20 vertices it did not even find the DAG that admitted a signature scheme with 141 patterns in the ten minutes that we ran the program for. We believe that this is because the increase in maximum in-degree also increased the number of DAGs that could be checked. By the time n reached 20, the size of the search space became the overwhelming factor.

Our program helped us to identify DAGs that facilitated large signature schemes, and in the next section we will see two signature schemes that we have designed with the help of our program. Both of the schemes have maximum in-degree two, so that they are comparable to the work by Bleichenbacher and Maurer [10, 11, 12].

Vertices n	Edges	Paths	$\lvert\mathcal{W}(\mathcal{G})\rvert$	$m(n)$	Old value of $m(n)$ from [10]
8	8	4	12	4	4
8	9	5	12	4	4
8	10	5	11	4	4
9	9	4	17	5	5
9	9	4	18	5	5
9	10	4	12	5	5
9	11	5	12	5	5
9	11	5	16	5	5
9	13	13	17	5	5
10	12	6	24	7	7
10	12	6	24	7	7
10	13	6	24	7	7
11	13	8	31	9	9
12	14	8	45	12	12
12	15	8	41	12	12
13	15	8	65	16	15
13	16	10	65	16	15
13	17	10	59	16	15
14	16	8	95	22	20
14	17	9	95	22	20
15	19	13	108	28	25
16	18	11	159	36	33
17	21	14	251	55	45
18	22	15	351	69	57
19	24	16	492	90	79
20	26	28	639	131	101
21	27	28	935	173	139

Table 5.5: Best lower bounds found by our program for $\mu(n)$ and previous best lower bounds from [10] for DAGs with in-degree at most 2.

Vertices n	Edges	Paths	$\lvert\mathcal{W}(\mathcal{G})\rvert$	$m(n)$	$m(n)$ when in-degree is 2
8	8	4	12	4	4
9	11	8	16	6	5
9	12	12	16	6	5
(10	9	3	28	8)	7
10	12	11	21	7	7
11	14	7	34	11	9
11	15	8	34	11	9
12	13	6	51	13	12
12	14	7	51	13	12
12	15	8	50	13	12
12	15	9	48	13	12
12	15	9	51	13	12
12	15	10	47	13	12
13	16	9	78	19	16
13	16	10	78	19	16
13	16	12	80	19	16
13	17	11	78	19	16
13	17	13	78	19	16
13	17	13	80	19	16
14	17	11	104	26	22
14	18	12	102	26	22
14	18	12	105	26	22
14	18	13	104	26	22
14	18	15	102	26	22
15	18	11	149	35	28
15	18	11	152	35	28
15	19	13	156	35	28
15	19	15	152	35	28
16	19	9	230	49	36
16	20	10	230	49	36
20	23	12	763	141	131

Table 5.6: Best lower bounds found by our program for $\mu(n)$ when the in-degree is at most 3. Note that the program did not find the bracketed row (this was found by hand).

Vertices n	Edges	Paths	$\lvert\mathcal{W}(\mathcal{G})\rvert$	$m(n)$	$m(n)$ when in-degree is 3
8	8	4	13	4	4
9	9	5	17	6	6
9	11	8	17	6	6
(10	9	3	28	8)	(8)7
10	10	5	25	7	(8)7
10	10	5	23	7	(8)7
10	11	6	25	7	(8)7
11	14	7	34	11	11
12	15	8	44	14	13
13	12	4	82	19	19
13	15	6	80	19	19
13	15	6	78	19	19
14	17	10	105	26	26
14	17	9	107	26	26
15	19	15	133	35	35
15	19	18	133	35	35
15	18	13	155	35	35
20	24	12	687	132	141

Table 5.7: Best lower bounds found by our program for $\mu(n)$ when the in-degree is at most 4. Note that the program did not find the bracketed row (this was found by hand).

5.4 Concrete examples of DAGs which facilitate efficient one-time signature schemes

In this section we present some examples of one-time signature schemes based on a generalised hash DAG for particular families of DAGs. First we present a family of DAGs due to Bleichenbacher and Maurer [11] that performs very well in the limit, and as the number of vertices tends to infinity it can be shown to contain a signature scheme of very high efficiency 0.47. We prove that this scheme is the optimal choice in a family of schemes.

Unfortunately, for the generalised hash DAG from [11], there is no easy way to find a construction of a good signature scheme, and consequently no signature pattern function is known. Without further work this scheme is primarily of theoretical interest.

We then present our scheme, capable of signing blocks of six bits with an efficiency of 0.3, and we exhibit a signature pattern function for it. Finally we present another more complex scheme that can be used to sign eight-bit blocks with an efficiency of 0.32.

5.4.1 Bleichenbacher and Maurer's DAG

In [11] Bleichenbacher and Maurer give a sequence of DAGs $\mathcal{H}_n$ and prove that there exists a sequence of signature schemes on them with efficiencies that tend to over 0.47. Figure 5.12 gives the DAG $\mathcal{H}_2$ based on two blocks of 12 vertices. In general the DAG is formed by stacking n 12-vertex blocks, and connecting the top three vertices of each block to the block above, as shown for the bottom block.

In [11] it is shown that there are $4^3 = 64$ signature patterns for each block. Of these, there are 13 patterns which result in vertices in lower blocks being unnecessary for a minimal verifiable set, and so there are $64 - 13 = 51$ combinations which can be used. They then consider the DAG $\mathcal{H}_n$ for which there are n blocks and so there are 51^n signature patterns, that is $|\mathcal{W}(\mathcal{H}_n)| \geq 51^n$. They then show, using Result 5.16, that there exists a signature scheme

on $\mathcal{H}_n$ with at least $\frac{51^n}{12n+5}$ signatures. This scheme has efficiency at least

$$\eta(\mathcal{H}_n) \geq \frac{\log_2(51^n/(12n+5))}{12n+6},$$

which converges to about 0.47 as n grows:

$$\lim_{n\to\infty} \eta(\mathcal{H}_n) \geq \frac{\log_2(51)}{12} \approx 0.47.$$

The efficiency of the graph can be improved, since the bottom three vertices can be merged into one vertex, but this makes no difference to the limit and would make the diagram less clear.

We now generalise the DAG $\mathcal{H}_n$ in Figure 5.12 to a DAG $\mathcal{H}_{n,r}$. It consists of n blocks, each containing r chains of length r. The top of the r chains in the top block are compressed to a single vertex by a binary hash tree (of order $r-1$). The top vertex of every other chain is connected to the block above by r edges. If this vertex tops the ith chain of a block then the r edges lead to the r lowest vertices in the block above satisfying:

$$(\text{chain number} + \text{height in chain}) \bmod r = i.$$

There are r additional source vertices I below the bottom block that are connected to the bottom block in the same manner.

A lower bound ξ_r on the efficiency $\lim_{n\to\infty} \eta(\mathcal{H}_{n,r})$ for these DAGs can be found in a similar manner to the original computation for $\mathcal{H}_n$ from [11].

For a single block in $\mathcal{H}_{n,r}$ (r chains of length r) we can form $(r+1)^r$ MVSs by taking one vertex from each chain. The construction in [11] selects a subset of these as *suitable block MVSs* $\{\varsigma_*\}$. We define a function $q(\varsigma_i, j)$ taking a suitable block MVS ς_i and a block index j which outputs the set of vertices corresponding to the MVS on that block.

To find the most efficient scheme we choose $\{\varsigma_*\}$ to be as large as possible so that all sets of vertices of the form:

$$I \cup q(\varsigma_{i_0}, 0) \cup q(\varsigma_{i_1}, 1) \cup \ldots \cup q(\varsigma_{i_{n-1}}, n-1)$$

are signature patterns. The lower bound ξ_r is found from a sequence of

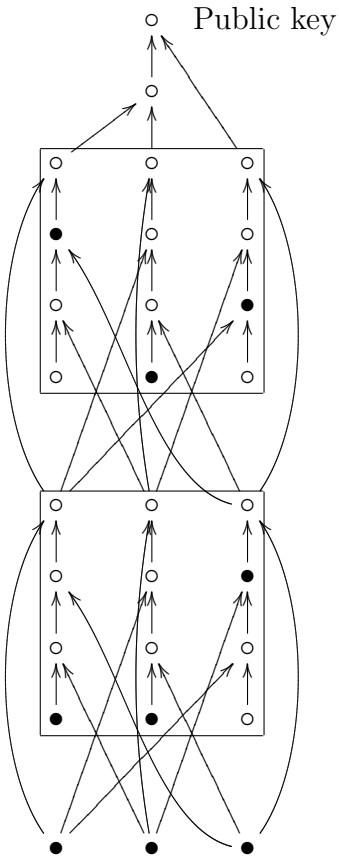


Figure 5.12: The DAG $\mathcal{H}_n$ due to Bleichenbacher and Maurer on 2 blocks (i.e. $\mathcal{H}_2$). A signature pattern is shown for the graph by the solid vertices. It can be shown that there are 51 signature patterns in each block that can be combined to make a minimal verifiable set on the whole DAG.

signature schemes formed from these signature patterns.

Theorem 5.17. *The lower bound ξ_r is maximised when $r = 3$.*

Proof When $r = 2$ we can see by inspection that $|\{\varsigma_*\}|$ is at most 6.

When $r = 3$ we know from [11] that there are 51 suitable block MVSs.

For $r \geq 4$ it will suffice to know that there are no more than $(r + 1)^r$ suitable block MVSs.

For each block in $\mathcal{H}_{n,r}$ we can pick any suitable block MVS and take the union of these sets of vertices and I to form a signature pattern. There are $|\{\varsigma_*\}|^n$ ways of doing this. This gives us a lower bound on the maximum

number of MVSs a DAG on $|\mathcal{H}_{n,r}|$ vertices can contain, and by Equation 5.16 a lower bound on the maximum efficiency of a one-time signature scheme on $\mathcal{H}_{n,r}$:

$$\frac{\log_2\left(|\{\varsigma_*\}|^n/|\mathcal{H}_{n,r}|\right)}{|\mathcal{H}_{n,r}|+1} = \frac{\log_2\left(|\{\varsigma_*\}|^n\right) - \log_2\left(nr(r+1)+5\right)}{nr(r+1)+6}.$$

We set ξ_r equal to the limit of this as n tends to infinity:

$$\xi_r = \frac{\log_2|\{\varsigma_*\}|}{r(r+1)}.$$

Evaluating this for small values of r gives the following:

r	$\lvert\{\varsigma_*\}\rvert$	ξ_r
2	6	0.43
3	51	0.47
4	$\leq (4+1)^4$	$\leq$0.46

We observe that for $r \geq 4$ we have

$$\xi_r \leq \frac{\log_2|\{(r+1)^r\}|}{r(r+1)} = \frac{\log_2(r+1)}{r+1},$$

which is strictly decreasing, and so ξ_r is maximised when $r = 3$. ■

5.4.2 A one-time signature scheme for multiples of six bits

As was mentioned in Section 5.2.4.1, the Lamport-Diffie scheme can be designed to sign messages of arbitrary length and has efficiency $\frac{1}{6}$. In Section 5.4.1 we saw that for arbitrarily sized messages we can construct a generalised hash DAG on which there provably exists a signature scheme with efficiency 0.47. There is currently no simple way to find this scheme, nor is there a simple map from the message space to the set of signature patterns.

We present a scheme with efficiency 0.3, which can be used to sign arbitrarily large messages, and also has a fairly straightforward map from the message space to the set of signature patterns.

The generalised hash DAG scheme shown on the right in Figure 5.7, and reproduced in Figure 5.13, is both simple and fairly efficient (on its own it has an efficiency of about 0.26). We will refer to this DAG as $\mathcal{G}$ in this section. We could increase the message space and maintain the efficiency by connecting several copies of $\mathcal{G}$ by joining their sinks with a hash tree.

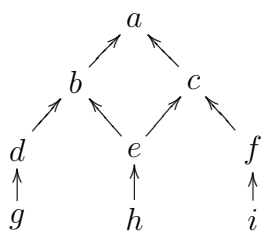


Figure 5.13: A generalised hash DAG $\mathcal{G}$ for which $w(\mathcal{W}(\mathcal{G}))$ is maximised compared to other DAGs of order 9.

Instead, we will investigate the structure of the signature patterns a little deeper, in order to improve the efficiency while maintaining a useable signature pattern function.

The set of all signature patterns of $\mathcal{G}$ is shown in Figure 5.14, with paths leading between all signature patterns which are incompatible.

All DAGs have an associated poset, and given any poset we can group the elements into layers such that all the elements in a layer are incomparable (and in this case compatible). With the poset in Figure 5.14 we get seven layers of sizes $1, 1, 2, 5, 5, 3$ and 1.

Theorem 5.18. *The DAG $\mathcal{H}$ (Figure 5.15) formed by joining the roots of two copies of $\mathcal{G}$ to a new vertex admits a one-time signature scheme of size 64.*

Proof If we combine two copies of $\mathcal{G}$ by connecting their sinks to a new vertex, we form a DAG which we will call $\mathcal{H}$. Since both halves of $\mathcal{H}$ have the minimal verifiable sets shown in Figure 5.14, we can form a minimal verifiable set for $\mathcal{H}$ by choosing an MVS for the left copy of $\mathcal{G}$ and an MVS for the right copy of $\mathcal{G}$.

To construct a signature scheme for $\mathcal{H}$ we need to pick signature patterns which are compatible, which can be done easily with the aid of the layers of Figure 5.14. We pick a pattern from the ith layer of the left copy of $\mathcal{G}$ and a pattern from the jth layer of the right copy of $\mathcal{G}$ such that $i + j = 7$ (where ghi is in the first layer, ghf is in the second layer, etc.). This provides a total of 64 signatures, which are shown in Table 5.8. ■

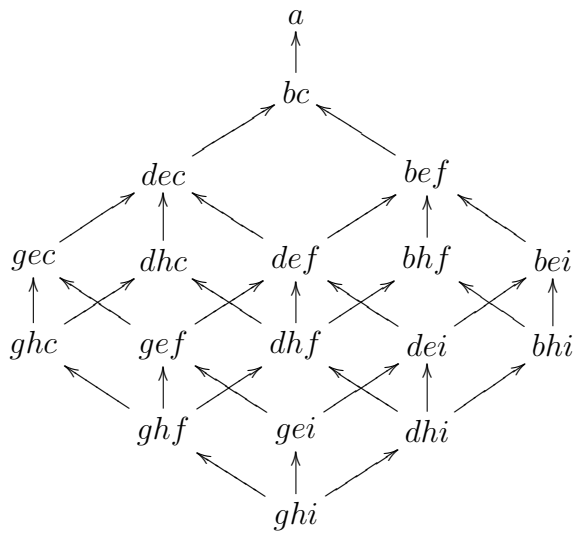


Figure 5.14: The Hasse diagram showing the relation between the minimal verifiable sets of the DAG in Figure 5.7. The third row from the bottom is the efficient signature scheme suggested.

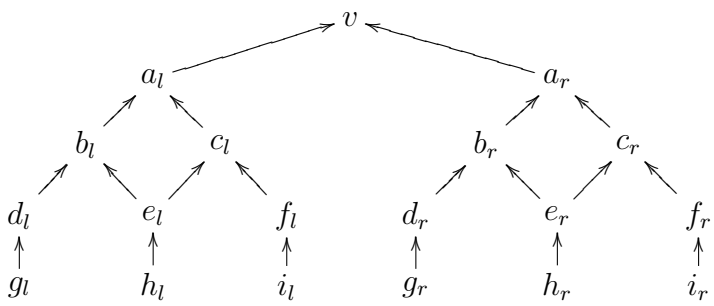


Figure 5.15: The graph $\mathcal{H}$ formed by joining two copies of $\mathcal{G}$ to new vertex v.

We can combine r copies of $\mathcal{H}$ with a hash tree and use Table 5.8 to provide 64 signature options for each copy. This gives us a total of $64^r = 2^{6r}$ signatures, and we can sign a $6r$-bit message by signing each block of 6 bits with a copy of $\mathcal{H}$.

This scheme has $(19 \times r) + (r - 1) = 20r - 1$ vertices and provides for a message space of size 2^{6r}, giving an efficiency of $\eta = \frac{\log_2(2^{6r})}{(20r-1)+1} = \frac{6}{20} = 0.3$.

5.4.3 A one-time signature scheme for multiples of eight bits

In Section 5.4.2 we presented a complete one-time signature scheme suitable for signing blocks of six bits. It was noted that by combining several copies of the generalised hash DAG it was possible to sign any multiple of six bits without affecting the efficiency of 0.3.

In this section we present another one-time signature scheme, this time capable of signing eight-bit blocks with an efficiency of 0.32. Again we can combine many copies of the underlying generalised hash DAG without affecting the efficiency.

Sig. Pattern Index	Pattern for left $\mathcal{G}$	Pattern for right $\mathcal{G}$
0	bc	ghi
1	dec	ghf
2	dec	gei
3	dec	dhi
4-6	bef	ghf,gei,dhi
7-11	gec	ghc,gef,dhf,dei,bhi
12-16	dhc	ghc,gef,dhf,dei,bhi
17-21	def	ghc,gef,dhf,dei,bhi
22-26	bhf	ghc,gef,dhf,dei,bhi
27-31	bei	ghc,gef,dhf,dei,bhi
32-63	Same as 0-31, with left and right patterns swapped	

Table 5.8: The map showing how the message space is mapped to the 64 signature patterns of the optimal signature scheme on $\mathcal{H}$.

The signature scheme will be based on the generalised hash DAG $\mathcal{G}_{24}$ shown in Figure 5.16. It has 24 vertices and, assuming it is seeded from a master private key (as described in Section 5.2.3.1), it will require 24 hash function calls to compute the root value from the master private key (one hash to compute the value at each vertex).

The mapping for each eight-bit value from 0 to 255 to a compatible signature pattern on $\mathcal{G}_{24}$ is shown in Table 5.9. It is clear from the size of Table 5.9 compared to the size of Table 5.8, that this scheme would be more complicated to implement than the scheme in Section 5.4.2. However this scheme is more efficient and arguably more suitable for systems based on bytes, or multiples of bytes, of information (which many modern systems are).

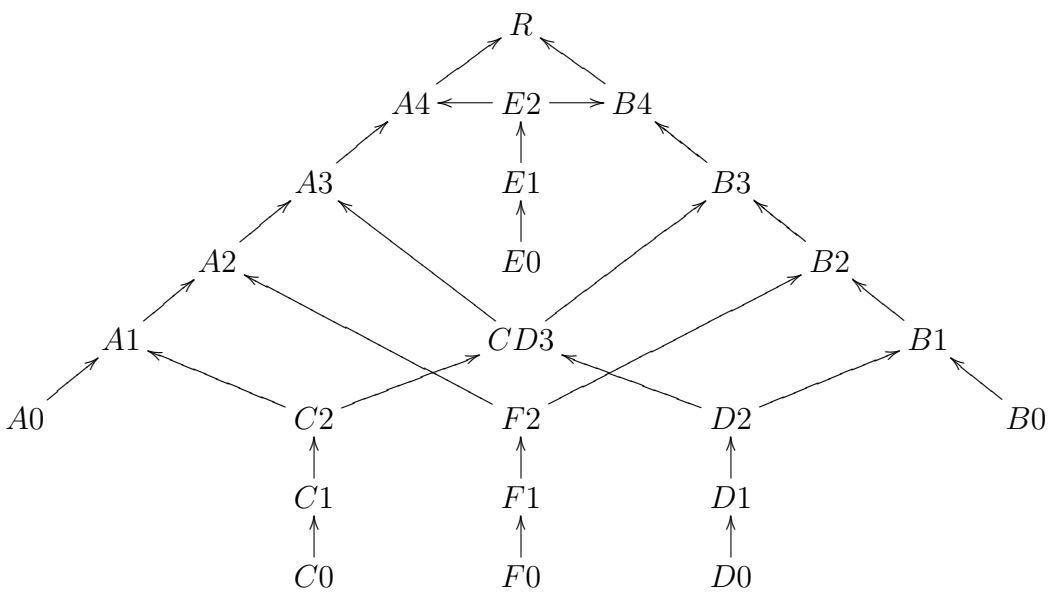


Figure 5.16: The generalised hash DAG $\mathcal{G}_{24}$. There are many compatible minimal verifiable subsets of this DAG, 256 of which are used by our one-time signature scheme. The vertices have been labelled to aid the description of this scheme.

Theorem 5.19. *The signature patterns in Table 5.9 are pairwise compatible, and so form a signature scheme.*

Sketch proof We divide the signature patterns into three types:

1. Signature patterns 0 to 210;
2. Signature patterns 211 to 254;
3. Signature pattern 255.

Any two Type 1 patterns can be checked to be compatible by viewing $\mathcal{G}_{24}$ as six chains labelled A to F and observing that all of the signature patterns have the same total distance from the end of the six chains.

A Type 1 and a Type 2 pattern, or any two Type 2 patterns, can be checked to be compatible in a similar way by treating vertex $CD3$ as an extra vertex in chain C.[8]

There is only one Type 3 signature pattern, so we do not need to consider whether two Type 3 patterns are compatible.

The Type 3 pattern does not reveal any information about $D0, D1$ or $D2$ and, since all of the other patterns contain one of these values, the signature pattern 255 does not reveal any other Type 1 or Type 2 pattern.

The Type 3 pattern contains both $E0$ and $F0$. The only other signature patterns that reveal both $E0$ and $F0$ contain either $A3, A4, B3$ or $B4$. Consequently these signature patterns do not reveal $A1$ and $B1$, and so do not reveal the Type 3 signature pattern.

Since the Type 3 pattern does not reveal any other pattern, and is not revealed by any other pattern, then it is compatible with all other signature patterns. Since all signature patterns are pairwise compatible, they form a signature scheme. ∎

This is not the largest signature scheme that can be formed on $\mathcal{G}_{24}$, but it is a practical scheme for signing multiples of 8 bits. We were unable to

[8]We note that it is possible to include another 44 compatible signature patterns by treating $CD3$ as part of chain D in the same way. However, since our aim was to create a scheme capable of signing eight-bit messages, we did not include these in our mapping.

Sig. Pattern Index	Signature Pattern					
0	A0	B0	C2	D2	E2	F2
1		B1	C1	D2	E2	F2
2			C2	D1	E2	F2
3				D2	E1	F2
4				D2	E2	F1
5		B2	C0	D2	E2	F2
6-8			C1	As 2-4		
9			C2	D0	E2	F2
10-11				D1	As 3-4	
12				D2	E0	F2
13					E1	F1
14					E2	F0
15-17		B3	C0	As 2-4		
18-23			C1	As 9-14		
24-25			C2	D0	As 3-4	
26-28				D1	As 12-14	
29				D2	E0	F1
30					E1	F0
31-36		B4	C0	As 9-14		
37-43			C1	As 24-30		
44-46			C2	D0	As 12-14	
47-48				D1	As 29-30	
49				D2	E0	F0
50-53	A1	B0	As 1-4			
54-63		B1	As 5-14			
64-79		B2	As 15-30			
80-98		B3	As 31-49			
99-105		B4	C0	As 24-30		
106-111			C1	As 44-49		
112-113			C2	D0	As 29-30	
114				D1	E0	F0
115-159	As 5-49, A&B values swapped					
160-210	As 64-114, A&B values swapped					
211-216	A0	B1	CD3	As 9-14		
217-223		B2	CD3	As 24-30		
224-229		B3	CD3	As 44-49		
230-232		B4	CD3	As 112-114		
233-254	As 211-232 with A&B values swapped					
255	A1	B1	CD3	E0	F0	

Table 5.9: The map showing how the message space is mapped to 256 signature patterns on $\mathcal{G}_{24}$.

find a smaller generalised hash DAG that contained 256 compatible signature patterns.

Theorem 5.20. *By replacing the n leaves of a binary tree of order $2n-1$ with n copies of $\mathcal{G}_{24}$ we obtain a graph $\mathcal{G}_{24,n}$ that admits a one-time signature scheme capable of signing n-byte messages with an efficiency of* 0.32.

Proof We can define a signature pattern function for $\mathcal{G}_{24,n}$ that maps the ith byte of the message to a signature pattern for the ith copy of $\mathcal{G}_{24}$ using the map in Table 5.9. From Theorem 5.19 we can see by inspection that these signature patterns for $\mathcal{G}_{24,n}$ will be compatible and so form a signature scheme.

We have $(2^8)^n$ signature patterns, one for every n byte message. The graph $\mathcal{G}_{24,n}$ has 24 vertices in each of the n copies of $\mathcal{G}_{24}$ and $n-1$ vertices in the tree. The efficiency of this scheme is therefore:

$$\eta = \frac{\log_2(2^{8n})}{(25n-1)+1} = \frac{8}{25} = 0.32.$$

■

This scheme could be used to sign a 128-bit hash value with a total of $25 \times \frac{128}{8} - 1 = 399$ calls to a hash function.

5.5 *k*-time signatures

In this section we extend the idea of one-time signatures to k-time signatures. We first present some new results about the efficiency of k-time signature schemes. Then we define two new concepts: *perforated* and *porous* k-time signature schemes. We note that there already exist k-time signature schemes, which are both perforated and porous, and present a conceptual design which is porous, but not perforated.

5.5.1 Efficiency of *k*-time signatures

Although the idea of a k-time signature scheme based on a generalised hash DAG has been around for a while (for example [12]) there has not been much work into developing the theory.

Definition 5.21. A ***k*-time signature scheme** based on a generalised hash DAG is a signature scheme that contains at least k signature patterns, and knowledge of the values of up to k signature patterns does not allow a forgery of any other signature pattern.

As an example, consider the DAG in Figure 5.17 on which the following 2-time signature scheme is based: ide, cje, cdk, fgh.

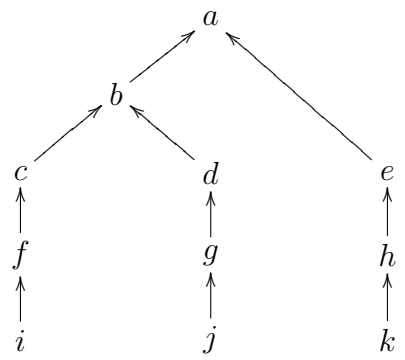


Figure 5.17: A tree which admits a 2-time signature scheme.

We will represent the set of signature patterns for a k-time signature scheme as Γ_k in order to highlight the difference in definition between a k-time scheme and a one-time scheme.

We look at two different ways in which a k-time signature scheme can be used:

1. The signatures may be used simultaneously or verified in a different order to that in which they were sent.

2. The signatures for each message may be created and verified before the next signature is released. We also assume that in this scenario, the k messages are all different.

These two different scenarios result in different amounts of information being signed. This can result in a k-time signature scheme having two different efficiencies depending on the way in which it is used. We present two generalisations of the definition of efficiency for a one-time signature scheme.

We give the efficiency η_1 for Scenario 1 in (5.10). Our signature scheme allows us to sign any k of the $|\Gamma_k|$ possible messages. Consequently there are a total of $\begin{pmatrix} |\Gamma_k| \\ k \end{pmatrix}$ ways to exhaust the signature scheme.

We give the efficiency η_2 for Scenario 2 in (5.11). The amount of data signed by the scheme increases from $\log_2 \begin{pmatrix} |\Gamma_k| \\ k \end{pmatrix}$ bits to $\log_2 \left(k! \begin{pmatrix} |\Gamma_k| \\ k \end{pmatrix} \right)$ bits, as the ordering of the signatures is significant in Scenario 2.

In the following equations n is the number of vertices in the DAG on which the k-time signature scheme is based:

$$\eta_1(\Gamma_k) = \frac{\log_2 \begin{pmatrix} |\Gamma_k| \\ k \end{pmatrix}}{n+1}; \tag{5.10}$$

$$\eta_2(\Gamma_k) = \frac{\log_2 \left(k! \begin{pmatrix} |\Gamma_k| \\ k \end{pmatrix} \right)}{n+1}. \tag{5.11}$$

In the following two theorems we show that by using the first measure of efficiency, no k-time signature scheme is more efficient than a one-time signature scheme, but by using the second measure, some k-time schemes are more efficient than any one-time scheme on the same DAG.

Theorem 5.22. *If we use η_1 as our measure of efficiency for a k-time signature scheme then, for any k-time signature scheme on any DAG $\mathcal{G}$, there exists a one-time signature scheme on $\mathcal{G}$ with at least the same efficiency.*

Proof We consider the set of all sets of k compatible signature patterns from Γ_k:

$$\Theta = \{\theta : \theta = \{U_1, \ldots U_k\} \text{ with } U_i \in W(\mathcal{G})\}.$$

There exists a supremum for any subset of $W(\mathcal{G})$ as it is a lattice. We claim that the set of signature patterns $\Upsilon = \{\sup(\theta) : \theta \in \Theta\}$ is a one-time signature scheme with the same efficiency as the k-time signature scheme.

We can see that Υ has at most $|\Theta| = \begin{pmatrix} |\Gamma_k| \\ k \end{pmatrix}$ elements by considering its design.

We can also show that Υ has at least $|\Theta|$ elements. If we assume the opposite then there must exist distinct θ_0 and θ_1 such that $\sup(\theta_0) = \sup(\theta_1)$. This implies that from the signature patterns in θ_0 we can compute the supremum of the signature patterns in θ_1. From the supremum of θ_1 we can compute all of the signature patterns in θ_1, and so we can also compute them from θ_0. Since θ_1 contains at least one signature pattern that is not in θ_0 this contradicts the definition of a k-time signature scheme (Defintion 5.21).

So $|\Upsilon| = \begin{pmatrix} |\Gamma_k| \\ k \end{pmatrix}$ and the efficiency of the one-time signature scheme Υ is the same as the k-time signature scheme Γ_k:

$$\eta(\Upsilon) = \frac{\log_2 \begin{pmatrix} |\Gamma_k| \\ k \end{pmatrix}}{|\mathcal{G}| + 1} = \eta_1(\Gamma_k).$$

■

In Table 5.10 we show the set of supremums for the 2-time signature scheme we described above, based on the hash tree in Figure 5.17.

We have proved that for any k-time signature scheme, there is a one-time scheme on the same DAG whose efficiency (η_1) is at least as good. Normally, however, there will be a strictly more efficient one-time scheme than k-time scheme on the graph. For example, for the DAG in Figure 5.17, the most efficient one-time signature scheme contains seven signature patterns (cgk, cjh, fdk, fje, idh, ige, fgh) but there is no k-time signature scheme with as good efficiency ($k \geq 2$).

Theorem 5.23. *There exists a k-time signature scheme with unbounded efficiency (η_2) as k tends to infinity.*

Proof Consider a binary Merkle tree with $(3k-1)$ vertices and k leaves. We can form a k time signature scheme on this tree by creating k signature patterns, each containing exactly one of the leaves. The efficiency (η_2) of

this scheme is $\frac{\log_2(k!)}{3k}$. Since $\log_2(k!) \approx k\log_2(k) - k$ (by Sterling's approximation [38]) we can approximate the efficiency by $\frac{1}{3}(log_2(k) - 1)$, and so it is unbounded as k tends to infinity. ■

We note that in practice this unbounded efficiency is not very relevant, as the application would place an upper bound on the parameter k. However it is interesting, even for small values of k.

In [10], Bleichenbacher and Maurer give a proof that no one-time signature scheme based on a tree has an efficiency greater than 0.42. Furthermore, as we mentioned in Section 5.4.1, the most efficient one-time signature scheme known has efficiency tending to about 0.47 as the size of the underlying DAG tends to infinity. We now present a simple 5-time signature scheme whose efficiency (η_2) is greater than that under the assumptions made.

On a 17-vertex Merkle tree with six leaves we can form a 5-time signature scheme using the method in Theorem 5.23. The efficiency of this signature scheme is $\eta = \frac{\log_2(720)}{17+1} \approx 0.53$. Since this signature scheme is based on a tree, we know that any one-time signature scheme based on the same DAG has a maximum efficiency of at most 0.42.

Sig. pattern from k-time scheme		Supremum on the
First signature	Second signature	associated poset
ide	cje	ije
ide	cdk	idk
ide	fgh	igh
cje	cdk	cjk
cje	fgh	fjh
cdk	fgh	fgk

Table 5.10: For this example we use a 2-time signature scheme based on the graph in Figure 5.17. If the signatures are used simultaneously (as in efficiency measure η_1) then a k-time signature scheme can be equated to a one-time signature scheme on the same DAG. This table shows a 2-time signature scheme and the one-time signature scheme formed by the supremums of its signature patterns.

5.5.2 Perforated and porous k-time signature schemes

When looking at the security of any scheme, we make an assumption that the adversary has only a finite time to perform an attack, and that any 'attacks' that take more than a certain amount of computation are not a concern.

All the schemes in this section take this assumption further. They all start with a scheme that would be insecure in practice for small parameters but, as the parameters increase, attacks on the scheme become as inefficient as brute force attacks on the underlying hash functions.

We define two types of k-time signature scheme, perforated and porous k-time signature schemes.

Definition 5.24. A **perforated** signature scheme is a signature scheme for which there is a small chance that it is not possible to sign a particular message.

Definition 5.25. A **porous** signature scheme is a signature scheme for which some subsets of signatures will allow forgeries on another signature.

If a signature scheme is porous then we hope that any forgeries are only existential forgeries, or that the adversary cannot find the corresponding messages to the revealed signatures.

All the k-time hash based signature schemes from the literature that we have seen are both perforated and porous [104, 108, 118]. In the next section we provide a k-time signature scheme that is porous but not perforated, but for which we have not been able to find a signature pattern function.

5.5.3 Towards a porous k-time signature scheme

In this section we will look at a conceptual design of a porous signature scheme. This scheme provides a signature for every message, thus it is not also perforated.

As we discussed in Section 5.5.1, it is possible to construct a k-time signature scheme using a generalised hash DAG. We also discussed that the measure of efficiency for such a scheme was dependent on the application.

It is hard to design efficient schemes where groups of k signatures do not reveal other valid signatures. However, (in a suitable security model) it is enough that the adversary cannot forge a signature from k random signatures for a message that they can find. In other words, we were concerned during our earlier discussion of generalised hash DAG based signature schemes that no new signature is revealed by the given signatures. In some situations, it may be acceptable for such a signature to exist, provided that the adversary cannot find a message that corresponds to it.

This would allow us to use a set of signature patterns which are not all pairwise compatible, as long as it is hard, given a random signature pattern, to find a different message whose signature pattern is computable from it.

This is potentially a useful observation, since it is often easy to enumerate the set of all signature patterns, whereas it is often hard to enumerate a maximal set of compatible signature patterns. A porous signature scheme may thus be constructed, provided that we can find a signature pattern function for a large set of *mainly compatible* signature patterns.

5.6 Conclusions

In this chapter we looked at the applications of hash structures to message authentication. We then studied one-time and k-time signature schemes based on hash structures. We presented some results related to Vaudenay's rake one-time signature schemes. We proved that for any one-time signature scheme on a hash tree, there exists an equivalent one-time signature scheme on a sub-tree, where the sub-tree is the largest Merkle tree contained in the original tree.

We presented a set of algorithms to efficiently find large signature schemes on given generalised hash DAGs. We used these algorithms to find two new one-time signature schemes, both of which are more efficient than any previously proposed scheme. It is an open problem to find schemes that are more efficient than ours, or which have similar efficiency but simpler signature pattern functions. It is also an open problem to find the maximum efficiency that a one-time signature scheme can attain.

We investigated the definition of efficiency for k-time signature schemes, and presented a two-part solution. We proposed two new definitions for k-time signature schemes, perforated and porous. It is an open problem to find a signature pattern function for our scheme of Section 5.5.3 in order to construct a porous k-time signature scheme.

Chapter 6

Key establishment schemes

We begin this chapter with an introduction to key establishment schemes and related definitions. We then study group key predistribution schemes, group key distribution schemes and (briefly) group key agreement schemes. We consider the use of hash structures in these applications and present some new schemes. We end the chapter by looking at some methods which allow the lifetime of a key to be extended using hash structures.

6.1 Introduction

In many applications we wish to provide a group of entities with a reliable method by which they can establish a shared key. Key establishment is studied by Martin in [79], and we adhere to similar notation and definitions.

The term *key establishment* refers to processes that ensure the correct keys are present in the correct places within a network. Often this takes place during the initialisation phase of a scheme, but it can also occur when fresh keys need to be distributed. Most schemes that we consider will have a *trusted authority* (or *TA*) denoted $\mathcal{T}$, which is an entity that is trusted by all members of the scheme.

Consider a set of users $\mathcal{U} = \{U_1, \ldots, U_n\}$ who wish to obtain shared keys with each other. We define a set $\mathcal{C}$ of subsets of $\mathcal{U}$, called the *communication structure*, such that for any $A \in \mathcal{C}$ we wish to establish a key k_A that is

shared by each user in A. The key k_A is called the *group key* of A.

For any key establishment scheme we normally wish to prevent certain sets of users colluding to compute group keys that they are not meant to know. The set $\mathcal{X}$ of such sets is called the *exclusion structure* of the scheme and is more precisely defined as:

$$\mathcal{X} = \{B \in \mathcal{U} : B \text{ cannot compute } k_A \text{ if } A \cap B = \emptyset \text{ for all } A \in \mathcal{C}\}.$$

A (t, w)-*threshold key establishment scheme* is a key establishment scheme where $\mathcal{C}$ is the set of all sets of at most t users, and $\mathcal{X}$ is the set of all sets of at most w users.

We define two specific types of collusion security: *w-security*, where $\mathcal{X} = \{\text{All sets of at most } w \text{ users}\}$, and *full collusion security* where $\mathcal{X} = \{\text{All sets of users}\}$.

Any key establishment scheme can be broken down into three stages: initialisation, key establishment and update.

1. *Initialisation* typically involves a TA creating the necessary values and sending secret data to each user. A TA may also publish some values which are available to everyone, which we denote by Pub.

2. The *key establishment* stage is where a group of users $A \in \mathcal{C}$ establish their common key k_A. This is the crux of the scheme, and different types of schemes will involve different amounts of work and communication during this stage.

3. Finally, in some schemes there is an additional *update* stage where some of the group keys are changed. This may involve the TA changing the public data Pub, or the users computing new keys from information they already have. Some users may have been revoked from the scheme, or others may have joined, in which case the sets of users in $\mathcal{C}$ may have changed. Alternatively in the case of *key refreshment*, the group keys are simply updated with new values.

We distinguish between different types of key establishment schemes.

- *Group key predistribution schemes* (or KPSs) are key establishment schemes where the TA is only involved during the initialisation phase.
- *Group key distribution schemes* (or KDSs) are key establishment schemes where the TA is involved during the key establishment phase.
- *Group key agreement schemes* (or KASs) are key establishment schemes where group keys are primarily derived by the users.

Typically, KPSs will have no update phase, as most schemes designed with update phases require the TA to be active during them. Often KASs to have no initialisation phase, since they are commonly designed so that the first communication between users is also the first time a key is needed, and thus constitutes the key establishment phase.

Most of the schemes that we will encounter will guarantee a key for each group in $\mathcal{C}$. We call these schemes *deterministic*. In some schemes for a group of users in $\mathcal{C}$ there is a small chance that they will not be able to generate a group key. These schemes are called *probabilistic* (for example HARPS [115]).

Schemes will be rated by several measures of efficiency:

- *Secret storage* — the amount of data that each user must keep secret. We assume that all the secret data is stored in uniform size *cryptographic values* and so we count the number of values instead of the number of bits;
- *Public communication* — the number of cryptographic values that must be made available to all participating parties during the scheme;
- *TA* (or *user*) *computation cost* — the amount of computation needed to be done by the TA (or user).

For the computation costs, we assume that 'basic' operations (such as exclusive-or and concatenation) take one unit of time, and we will often treat them as negligible. We assume that a hash function requires t_f units

of time, and an encryption or decryption function requires t_E units of time (which will typically be larger than t_f).

We will give a joint analysis of the initialisation and key establishment phases, but will address any update phases separately. This is because the initialisation and key establishment phases will be performed once to set up a key, whereas the update phases are likely to be performed repetitively, and affect all users.

6.2 Group key predistribution schemes

In this section we look at key predistribution schemes, starting with some simple benchmark schemes. We will then study some schemes based on hash structures from Chapter 3, and see the potential advantages offered. Finally we look briefly at the idea of key escrow, and we show how hash chains can be used to provide it for most predistribution schemes.

6.2.1 Introduction

We begin with a formal definition of a predistribution scheme.

Definition 6.1. A **$(\mathcal{C}, \mathcal{X})$-key predistribution scheme** (or KPS) is a key establishment scheme with communication structure $\mathcal{C}$ and exclusion structure $\mathcal{X}$ such that any user in $A \in \mathcal{C}$ can compute k_A from the information provided in the initialisation phase, and any set of users $A' \in \mathcal{X}$ cannot compute k_A (assuming $A' \cap A = \emptyset$).

We now describe two trivial key predistribution schemes.

Key Establishment Scheme 6.1: The most trivial KPS where each user gets one key for each group they are in.

Initialisation

- $\mathcal{T}$ generates a key k_A for each group $A \in \mathcal{C}$.
- $\mathcal{T}$ securely sends k_A to each user in A.

Analysis

Secret storage	Number of groups containing user
Public communication	0
TA computation cost	0
User computation cost	0

This scheme has full collusion security. There is no need for a key establishment phase; however the scheme suffers from being very expensive in terms of user secret storage.

We now look at another trivial scheme that requires greater public storage but less communication over a secure channel, and less secret storage for each user.

Key Establishment Scheme 6.2: A KPS with minimal secret storage for each user, but large public storage.

Initialisation

- Each user U_i is securely supplied with a master key K_i by $\mathcal{T}$.
- $\mathcal{T}$ encrypts each group key with the master key for each user in it, and publishes the result $E_{K_i}(k_A)$.

Key establishment

- If user $U_i \in A$ ($A \in \mathcal{C}$) wishes to obtain k_A, they simply decrypt $E_{K_i}(k_A)$.

Analysis

Secret storage	1
Public communication	$\sum_{A \in \mathcal{C}} \lvert A \rvert$
TA computation cost	$\sum_{A \in \mathcal{C}} \lvert A \rvert t_E$
User computation cost	t_E

This scheme can be improved if $\mathcal{C}$ has a convenient structure by using some session keys to encrypt session keys for larger groups. For example, if group A is completely contained in group B then the members of group A

would all be able to decrypt $E_{k_A}(k_B)$ and so $\mathcal{T}$ can publish this value instead of the set of values $\{E_{K_i}(k_B) : i \in A\}$.

A final fundamental scheme we will look at is Scheme 6.3, but first we must define *key distribution patterns*.

Definition 6.2. A **$(\mathcal{C}, \mathcal{X})$-key distribution pattern** (or **KDP**) is a set of indices $\mathcal{I} = [0, v-1]$ and a set of subsets of these indices $\mathcal{B}$. Each user U_i is assigned a subset $B_i \in \mathcal{B}$ such that for any $A \in \mathcal{C}$ and $A' \in \mathcal{X}$ $(A \cap A' = \emptyset)$ we have

$$\bigcap_{U_i \in A} B_i \nsubseteq \bigcap_{U_j \in A'} B_j.$$

Key distribution patterns were first studied by Mitchell and Piper in [87].

Key Establishment Scheme 6.3: A basic KDP-based KPS.

Initialisation

- $\mathcal{T}$ generates a sub-key x_i for each index $i \in \mathcal{I}$, and securely sends the sub-key to user U_j if $i \in B_j$.
- $\mathcal{T}$ publishes the KDP, so each user can find out which sub-keys each other user has been given.

Key establishment

- The group key k_A for a group $A \in \mathcal{C}$ is computed from the group sub-key index set $B_A = \bigcap_{U_i \in A} B_i$ as the following bit-wise exclusive-or:

$$k_A = \bigoplus_{j \in B_A} x_j.$$

Analysis

Secret storage	$\lvert B_j \rvert$
Public communication	$\sum_j \lvert B_j \rvert$
TA computation cost	0
User computation cost	$\lvert B_j \rvert$

It could be suggested that using exclusive or (XOR) to combine the subkeys x_j leaves the scheme vulnerable, as a compromised key could reveal the subkeys. This can be overcome by replacing XOR with a preimage resistant hash function, but at the expense of more computation.

Two common types of key distribution patterns are the *trivial inclusion KDP* and the *trivial exclusion KDP*, based on the communication structure and the exclusion structure respectively.

To create a *trivial inclusion KDP* we have $v = |\mathcal{I}| = |\mathcal{C}|$. We assume the communication set $\mathcal{C} = \{A_1, \ldots, A_v\}$. The index set is defined as $B_i = \{j : U_i \in A_j\}$. When Key Establishment Scheme 6.3 is based on this KDP, the only users with sub-key x_j are precisely those in group A_j.

To create a *trivial exclusion KDP* we have $v = |\mathcal{I}| = |\mathcal{X}|$. We assume the exclusion set $\mathcal{X} = \{A'_1, \ldots, A'_v\}$. The index set is defined as $B_i = \{j : U_i \notin A'_j\}$. When Key Establishment Scheme 6.3 is based on this KDP there is a sub-key owned by all the users not in $A'_j \in \mathcal{X}$, and so every excluded set of users cannot form any group keys they are not meant to. The trivial exclusion KDP was first given in [39].

6.2.2 Existing hash-based key predistribution schemes

In this section we explore key predistribution schemes based on hash functions. Using some of the hash structures described in Chapter 3 we can form key predistribution schemes that have better properties than the fundamental schemes in Section 6.2.1. We also note that there exist many key predistribution schemes that are based on one-way trapdoor functions, such as [13], [14] and [100]. These schemes are also better, in some respects, than the schemes from Section 6.2.1. However we do not consider them further, as schemes based on hash functions are typically more efficient and better suited to light-weight devices.

6.2.2.1 Inverted hash tree key predistribution schemes (IHT KPS)

We present a new type of hash-based KPS, which we call *inverted hash tree key predistribution schemes*. This scheme is a generalisation of a KPS scheme

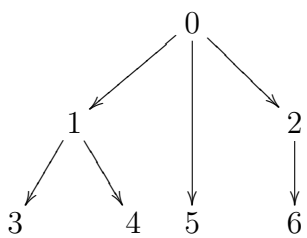
due to Lee and Stinson from [70], and many of their ideas also apply to our new scheme.

Both our scheme and the original assign values to each user from a set of inverted hash trees. In the original scheme, each user receives a value from every tree. Our generalisation allows the TA to choose a subset of trees for each user. This increases the range of KPSs that can be produced from a given set of inverted hash trees.

Definition 6.3. For given $\mathcal{C}$ and $\mathcal{X}$, and for a rooted tree $\mathcal{G}$, we can form a matrix $M = (\alpha_{i,j})$ with entries from $(V(\mathcal{G}) \cup \infty)$. Vertex ∞ is considered a descendant of all vertices in $\mathcal{G}$. We form M with the jth column corresponding to the user U_j and a row for each of b copies of the tree (for some value of b). M is a **$(\mathcal{C}, \mathcal{X}, \mathcal{G})$-inverse hash tree key distribution pattern** (IHT KDP) if, for any disjoint sets $A \in \mathcal{C}, B \in \mathcal{X}$, there exist values $\kappa(A, B) \in [0, b-1]$ and $\lambda(A, B) \in A$ such that the following two conditions hold:

1. $\alpha_{\kappa(A,B),\lambda(A,B)}$ is a descendant of $\alpha_{\kappa(A,B),j}$ for all $U_j \in A$;
2. $\alpha_{\kappa(A,B),\lambda(A,B)}$ is not a descendant of $\alpha_{\kappa(A,B),j}$ for all $U_j \in B$.

Figure 6.1 has an example of a $(\mathcal{C}, \mathcal{X}, \mathcal{G})$-inverse hash tree key distribution pattern for six users, where $\mathcal{C}$ and $\mathcal{X}$ are both the set of all subsets of at most two users.



$$\begin{pmatrix} 0 & 3 & 4 & 5 & 2 & 6 \\ \infty & 0 & 3 & 4 & 2 & 6 \\ 4 & 5 & 0 & 3 & 2 & 6 \\ 5 & 2 & 2 & 1 & \infty & 0 \\ \infty & 4 & 3 & 5 & 5 & 1 \end{pmatrix}$$

Figure 6.1: An $(\mathcal{C}, \mathcal{X}, \mathcal{G})$-IHT KDP for six users with $b = 5$. $\mathcal{G}$ is shown on the left, and $\mathcal{C}$ and $\mathcal{X}$ are both the set of all subsets of at most two users. The matrix $M = (\alpha_{i,j})$ is shown on the right.

Key Establishment Scheme 6.4 constructs a KPS from a IHT KDP.

Key Establishment Scheme 6.4: An inverted hash tree key predistribution scheme (IHT KPS) based on a tree $\mathcal{G}$ and a IHT KDP M.

Initialisation

- $\mathcal{T}$ publishes M, $\mathcal{G}$ and the description of the hash function f.
- $\mathcal{T}$ generates b inverted hash trees (see Section 3.4.5), each based on the tree $\mathcal{G}$, and seeded from the root values $\{x_{0,0}, \ldots, x_{b-1,0}\}$. The value at vertex v of hash tree i is called $x_{i,v}$.
- The value $x_{i,\infty}$ is always defined to be 0 and consequently does not need to be sent to, or stored by, any users.
- $\mathcal{T}$ securely sends user U_j the values $\{x_{i,\alpha_{i,j}} : i \in [0, b-1], \alpha_{i,j} \neq \infty\}$.

Key establishment

- To compute the group key for a group $A \in \mathcal{C}$, a set of inverted hash tree values are selected:

$$X_A = \bigcup_{B \in \mathcal{X}} x_{\kappa(A,B),\alpha_{\kappa(A,B),\lambda(A,B)}}.$$

  The set X_A is the 'most secret' set of values that all the members of A can compute.

- The group key for $A \in \mathcal{C}$ is defined as

$$k_A = \bigoplus_{x \in X_A} x.$$

Analysis

Secret storage	$\approx b$
Public communication	$\approx b\|\mathcal{U}\|$
TA computation cost	0
User computation cost	$\|X_A\|$

We see from the properties in Definition 6.3 that all members of A can compute the group key, and that for any excluded group B there is a part of the group key they will not be able to compute, namely $x_{\kappa(A,B),\alpha_{\kappa(A,B),\lambda(A,B)}}$. Consequently the members of B will gain no information about the group key.

Since the TA has to create an inverse hash tree for each row of the matrix M, we prefer instances of M with fewer rows.

6.2.3 Hierarchies and key establishment schemes

It is often the case that a communication structure can be represented by a poset, such as the one shown in Figure 6.2.

Definition 6.4. A *security hierarchy* is a poset of *security levels* $\mathcal{S}$ with the relation of security. All users are associated with a set of security levels such that, if a user is associated with a security level $s \in \mathcal{S}$ then they are also associated with all lower security levels than s.

If we are creating a key establishment scheme for a hierarchy based on a poset then we may be able to take advantage of the structure by making the group keys relate to each other. In the given example, we could let the sales director's key be obtainable from the managing director's key, reducing the amount of information that the managing director needs to store.

A *totally ordered hierarchy* is such that all levels are comparable, that is any pair of security levels $s, s' \in \mathcal{S}$ are such that either $s \leq s'$ or $s' \leq s$. A *tree-shaped hierarchy* is such that for any two upper bounds s, s' of a security level, either $s \leq s'$ or $s' \leq s$.

6.2.3.1 Tree-shaped hierarchy based KPS schemes

The earliest published KPS for a hierarchy is the scheme in [1] by Akl and Taylor, which is designed for a totally ordered hierarchy. The members of the smallest group are given a key x_0, which is used to seed a hash chain, and the ith value along the hash chain is used as the group key for the ith smallest

group. Each user only needs to store one group key, from which all the other keys they are entitled to can be computed.

Key Establishment Scheme 6.5: The KPS scheme for a totally ordered hierarchy given by Akl and Taylor in [1]. We label the groups in $\mathcal{C}$ as $\{A_0, \ldots, A_l\}$ such that $A_i \subseteq A_{i+1}$ for all $i \in [0, l-1]$.

Initialisation

- $\mathcal{T}$ picks a value x_0 at random and uses it to seed a hash chain of length l.
- $\mathcal{T}$ securely sends x_0 to all members of A_0 and x_i to all members of A_i/A_{i-1} (for $1 \leq i \leq l$).

Key establishment

- A user who has been given x_i can compute the group key for A_j as long as $j \geq i$. They compute the group key as $f^j(x_0) = f^{(j-i)}(x_i)$.

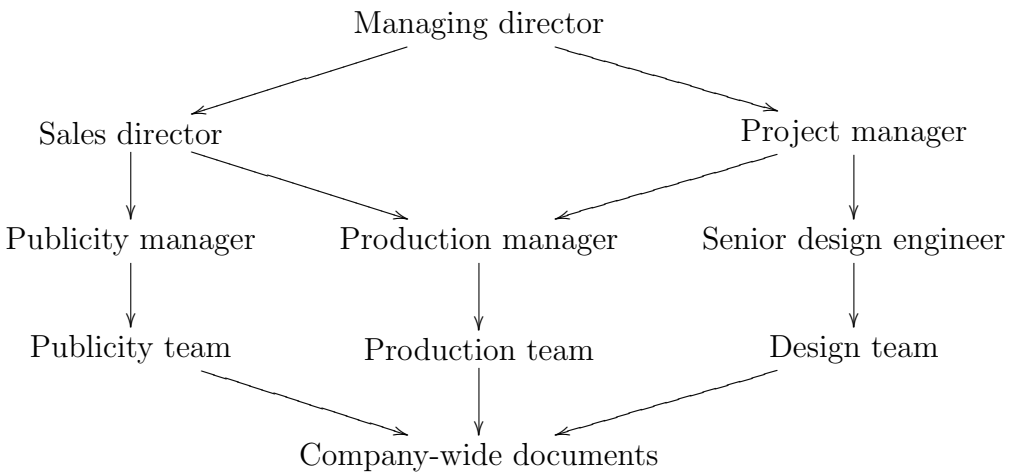


Figure 6.2: A Hasse diagram showing the hierarchy of a business. If there is a path from security level A to security level B then any user with access to security level A is allowed to read all documents produced at security level B.

Analysis

Secret storage	1
Public communication	0
TA computation cost	lt_f
User computation cost	$\leq lt_f$

Many authors have extended the above scheme to allow for tree-shaped hierarchies [24, 124, 156, 157, 158], by which we mean a hierarchy such that for any two sets $A_i, A_j \in \mathcal{C}$ which both contain the user U_k, we have either $A_i \subseteq A_j$ or $A_i \subseteq A_j$. Many of the above schemes define the hash function for each edge in the inverted hash tree differently, but apart from that they are all essentially Key Establishment Scheme 6.6.

Key Establishment Scheme 6.6: A key predistribution scheme based on an inverted hash tree for tree-shaped hierarchies. We label the groups in $\mathcal{C}$ as $\{A_0, \ldots, A_{v-1}\}$ such that if $A_i \subseteq A_j$ then $i \leq j$.

Initialisation

- $\mathcal{T}$ picks a value x_0 at random and uses it to seed an inverted hash tree with the same structure as $\mathcal{C}$.
- $\mathcal{T}$ securely sends x_i to all members of $A_i \backslash (\bigcup_{j<i} A_j)$ (note that each user only receives one key).

Key establishment

- The group key for a group A_j is the inverted hash tree value at the vertex corresponding to A_j.
- A user who has been given x_i can compute the group key for any group A_j that they are a member of. They compute the group key by applying the appropriate sequence of hash functions.

Analysis

Secret storage	1
Public communication	0
TA computation cost	$\lvert\mathcal{C}\rvert t_f$
User computation cost	$\approx \log \lvert\mathcal{C}\rvert \times t_f$

This is a deterministic key predistribution scheme with minimal use of a secure channel.

Another KPS for tree-shaped hierarchies is suggested by Ramkumar and Memon in [116]. The scheme is an extension of HARPS, formed by restricting the values a group can receive to be values from a subset of the chains that its parent's values are from. This restriction allows the parent group to derive the child group's key from its own secret information. This scheme differs from Scheme 6.6 because it has a small chance of failing to provide a group with a group key, as was also the case with HARPS [115].

6.2.3.2 General hierarchy-based KPS

The main problem with using hash functions for key predistribution schemes comes if there is more than one covering element for a particular security level (for example in Figure 6.2 'Production manager' has two covering elements, 'Sales director' and 'Project manager'). In this case it is infeasible for users from all these security levels to be able to compute the same group key from the different information that they have been given.

Using the 'Production manager' example from Figure 6.2, we need to give the 'Sales director' and 'Project manager' groups distinct information which can be used to find the 'Production manager' group key. One way to do this is to use Key Establishment Scheme 6.6, ignoring the requirement for the 'Project manager' group to be able to compute the 'Production manager' group's key, and then supply the 'Project manager' group with the required key. This is a very practical scheme and for this example would be very efficient; however it has the drawback that some users have to store more than one key.

Another obvious, but inefficient scheme is to create two copies of all documents created by the 'Production manager' and 'Production team' groups. These would be encrypted with different keys derived from the 'Sales director' group key and the 'Project manager' group key. In our example this would double the encryption time. It may also cause other problems with version management for the organisation.

The first attempt to solve this problem, while keeping user storage to a minimum, was by Gudes in [45]. Gudes hoped to find collisions in the hash functions at appropriate points, although did not explain how one should go about finding the collisions. We note that if collisions are too easy to find on a mapping where the codomain and domain are equal (here we have the application of a hash function to its own range), then the potential key space will be vulnerable to an exhaustive search attack.

A similar scheme is described in [157] by Zheng et al. The scheme requires the existence of a family of functions U such that for r given inputs and one given output it is easy to find a function in U that maps all the inputs to the given output. This 'collision' property allows for all the parents to find the child's key, without compromising their own security. They also require that, given this function, it is computationally hard to find the output without knowledge of at least one of the inputs. This ensures that the child's key cannot be found trivially from the function, since it would be pointless to use the function if this were the case.

Even if such functions are found then they are unlikely to be as efficient as the scheme in the next section.

Other methods of extending Key Establishment Scheme 6.6 to general hierarchies have been created by Zhong [158], Yang and Li [156] and Crampton et al. [23].

6.2.3.3 Key predistribution for lattice-shaped hierarchies

In this section we present a scheme suitable for 'lattice-shaped hierarchies'. A *d-dimensional lattice-shaped hierarchy* is a hierarchy where every security level $A \in \mathcal{C}$ has an associated set of indices $(a_{A,0}, \ldots, a_{A,d-1})$, and A is strictly more secure than B if $a_{A,i} \leq a_{B,i}$ for all i.

If we have a d-dimensional lattice-shaped hierarchy then we can use the following new key predistribution scheme.

Key Establishment Scheme 6.7: A key predistribution scheme suitable for d-dimensional lattice-shaped hierarchies.

Initialisation

- $\mathcal{T}$ creates d hash chains $(x_{0,0}, \ldots, x_{0,l_0}), \ldots, (x_{d-1,0}, \ldots, x_{d-1,l_{d-1}})$ of lengths $l_0, \ldots, l_{d-1}$, where:
$$l_i = \max_{A \in \mathcal{C}} (a_{A,i}) .$$
- $\mathcal{T}$ securely supplies each member of $A \in \mathcal{C}$ who is not also part of a more secure group with the set $\{x_{0,a_{A,0}}, \ldots, x_{d-1,a_{A,d-1}}\}$.

Key establishment

- Any member who is in A and also in B, where A is more secure than B, can compute the associated index set of B by hashing, because $a_{A,i} \leq a_{B,i}$ for all i.
- The group key of A is
$$\bigoplus_{0 \leq i \leq d-1} x_{i,a_{A,i}}.$$

Analysis

Secret storage	d
Public communication	0
TA computation cost	$\sum_i l_i t_f$
User computation cost	$\leq \sum_i l_i t_f$

We note that this KPS is also suitable for hierarchies that are contained within a lattice-shaped hierarchy. We give an example of this in Figure 6.3.

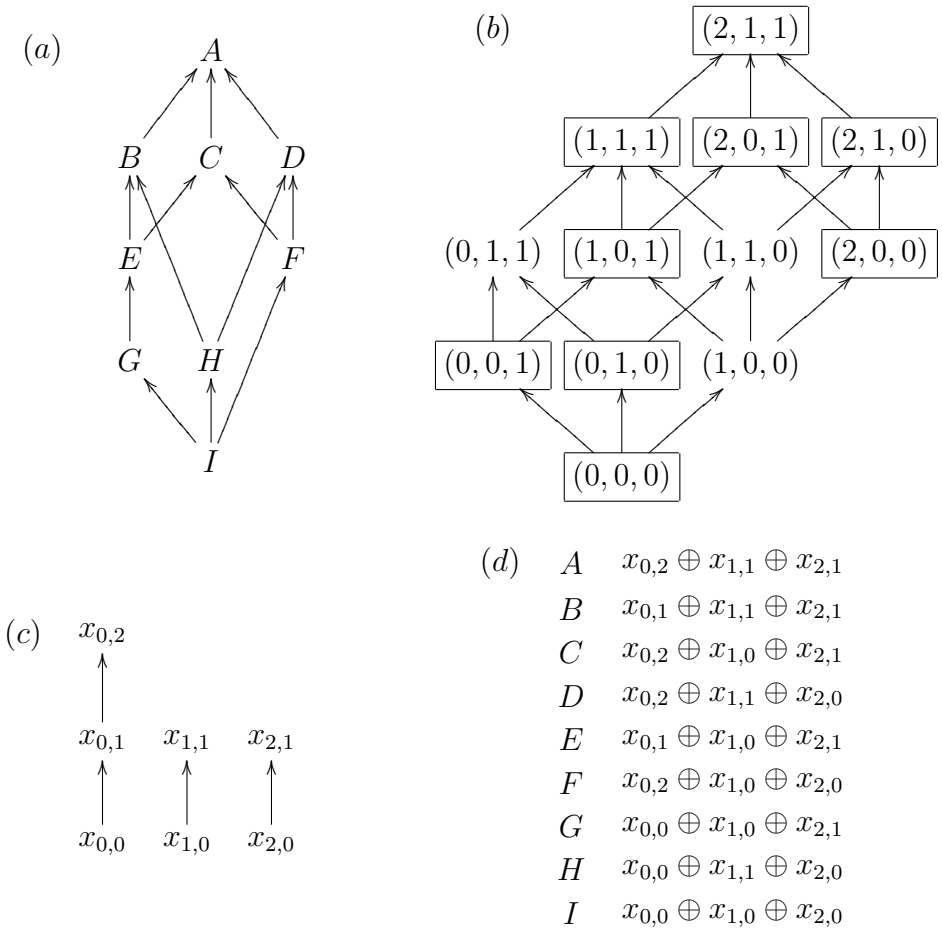


Figure 6.3: An example of applying Key Establishment Scheme 6.7 to a hierarchy contained in a lattice-shaped hierarchy.
(a) The hierarchy which requires keys. I is the most secure group and A is the least secure group.
(b) The lattice in which the hierarchy is contained. The boxed nodes correspond to the groups in the hierarchy.
(c) The hash chains generated for Key Establishment Scheme 6.7 when used with the lattice in (b).
(d) A table showing the group key for each group in the hierarchy.

6.2.3.4 A generalisation to many more hierarchies

Using generalised hash DAGs (Definition 3.7) we can extend Key Establishment Scheme 6.7 to many more types of hierarchy.

Key Establishment Scheme 6.8: Our key predistribution scheme based on a generalised hash DAG.

Initialisation

- $\mathcal{T}$ creates a generalised hash DAG $\mathcal{G}$.
- $\mathcal{T}$ securely supplies each member of $A \in \mathcal{C}$ who is not also part of a more secure group with the corresponding MVS (Definition 5.10).

Key establishment

- Any member who is in A and also in B, where A is more secure than B, can compute the MVS of B by hashing.
- The group key of A is the exclusive-or of all the values in its MVS.

Analysis

Secret storage	$\lvert\text{MVS}\rvert$
Public communication	0
TA computation cost	(Number of vertices − Number of sources)t_f
User computation cost	$\leq$ (Number of vertices − Number of sources)t_f

We would like to use a small DAG whose associated poset $\mathcal{W}(\mathcal{G})$ contains our hierarchy. We note that, for any given hierarchy, the algorithms in Section 5.3 could be adapted to find a small DAG that is suitable.

An example of a hierarchy and a suitable DAG for that hierarchy is given in Figure 6.4.

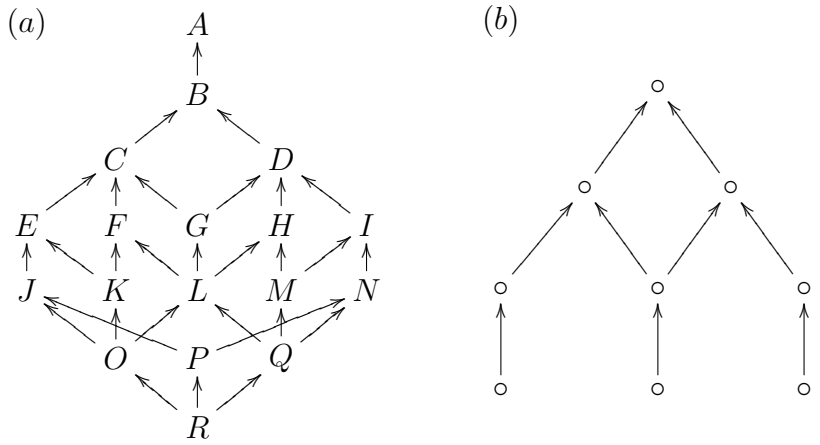


Figure 6.4: On the left is a hierarchy, and on the right is a generalised hash DAG which can provide keys with the relationship needed for the hierarchy.

6.2.3.5 Providing key escrow for key predistribution schemes

A consideration for key establishment schemes is that some applications require keys to be held in escrow by a third party. The third party will typically only use the key if the user associated with that key does something wrong, or is suspected of illicit behaviour. For example, a user's key might be kept secret by the third party until the law authorities obtain a court order to use it.

In [58] Joye and Yen describe a scheme which allows the key escrow agent to efficiently release a user's secret keys for a particular time interval. The TA gives each user the seeds to two hash chains x_0 and y_0, both of length n, and defines the session key for time period t_i as $(x_i \oplus y_{n-i})$. The authors refer to this construction as a 'one-way cross-tree'.

If the authorities request access to key values between t_i and t_j $(i < j)$ then the TA can give them x_i and y_j, and then the authority can only compute the requested group keys.

This scheme has the drawback that the TA cannot supply key values for two separate intervals without also revealing all the session keys in between.

We observe that this scheme can be applied to any set of group keys formed by a standard KPS. If the original KPS provides a group key k_A then

the key escrow version of the KPS uses two hash chains seeded by $f(k_A||0)$ and $f(k_A||1)$ to compute session keys for the same group.

6.3 Group key distribution schemes

This section is mainly included for completeness; some of the most well known uses of hash structures are key distribution schemes. Our contributions in this section are to present a summary of some of the better KDSs, to propose a 'user add' phase for a KDS from [21] (Scheme 6.9), and to present a flaw in the update phase of a KDS from [63] (Section 6.3.3.3).

6.3.1 Introduction

Group key distribution schemes allow for some communication between the TA and the users during the key establishment phase.

Definition 6.5. A **$(\mathcal{C}, \mathcal{X})$-key distribution scheme** (or KDS) is a key establishment scheme with communication structure $\mathcal{C}$ and exclusion structure $\mathcal{X}$ such that:

1. Any user in $A \in \mathcal{C}$ can compute k_A from the information provided in the initialisation phase u_i and the information provided by $\mathcal{T}$ during the key establishment phase $v_{i,A}$;

2. Any group of users $B \in \mathcal{X}$ disjoint from A cannot compute k_A from the information provided to them during the initialisation and key establishment phases.

6.3.2 Different schemes for different applications

In this section we look at two very common scenarios which key distribution schemes are applied to.

In the first scenario there is only one stream of data, and most users in the scheme are assumed to want to receive it. Users may be added or revoked from the stream. An example of this scenario is an encrypted digital

TV channel (or package of channels). All users are assumed to want access to the programmes, but some may forget to pay their subscription fee and so may have their access temporarily revoked (or permanently revoked if they opt out altogether). We will refer to schemes that fit this scenario as *revocation schemes*, as the group key is known by all but a small number of revoked users.

The second scenario may have several streams of encrypted data, each with only a few users able to decrypt them. Each stream, each group membership and each group key, is not assumed to be related to other streams, group memberships and group keys. Also, each stream has a short lifetime compared to data streams in revocation schemes.

An example of this scenario is pay-per-view sporting events. Any users who want to watch a particular game must pay to obtain a key for just that event. Alternatively, the users who want to watch an event pay and then the broadcast centre encrypts the game in such a way that only the subscribed members can view it. The set of viewers for one event are not assumed to have any links to the set of viewers for another.

We will refer to a scheme providing many groups with keys as a *broadcast encryption scheme*.

Ideally the user set $\mathcal{U}$ for revocation and broadcast encryption schemes should be adaptable during an update phase.

6.3.3 Logical key hierarchy

If the TA predistributes enough group keys, so that each user has at least one key, then the TA can use a suitable subset of them to encrypt a universal group key. This subset of keys is often referred to as a set of *key encryption keys* (or *KEKs*).

One idea for a predistibuted set of group keys is to arrange the users as the leaves of a tree (often a binary tree), and create a key encryption key at each node in the tree. Each user receives all the key encryption keys along the path from their leaf to the root.

A type of key distribution scheme based on a tree is the *logical key hierar-*

chy (or LKH) and was independently proposed in [144] and [154]. It was first suggested as being based on an almost perfect tree, but easily generalises to any tree. Users are assigned a leaf of the tree and are privy to all the values along the path from their leaf to the root.

By decreasing the height of the tree and increasing the average degree we can reduce the storage requirements for each user, but increase the average number of KEKs that must be used to send a message to an arbitrary set of users.

6.3.3.1 Using hash structures for a logical key hierarchy

In [82], McGrew and Sherman proposed an improved version of the logical hierarchy scheme based on [144] and [154]. In it they replace the independent keys at each node with keys formed by a hash tree.

The hash function used by McGrew at al. to combine child key values c_1 and c_2 is $M(f(c_1), f(c_2))$, where f is a preimage-resistant hash function and M is a mixing function. The authors suggest that exclusive-or is a suitable choice for the function M, and for simplicity we assume this to be the case.

Another adaptation of the LKH is proposed in [19] by Canetti et al. The scheme reduces the overhead required for user add and user revoke by defining all new keys to be related to each other as values in a hash chain. In this way, each user only needs to be told the most secure new value that they are privy to, and from that they can compute all the others.

Although these schemes are designed for binary trees, they can be generalised to other trees (for example see [63]). Canetti et al. [20] also generalise the binary tree logical key hierarchy scheme to an a-ary tree scheme where each leaf is associated with a small group of users.

In [117] Reddy and Nalla present a key agreement scheme similar to the logical key hierarchy scheme, which uses the Diffie Hellman key exchange method [34]. Their work is an extension of the two party ID-based authenticated key agreement protocol in [130] to a many party scheme based on a logical key hierarchy. The scheme has advantages over schemes based on symmetric cryptography, but it is much slower and less suitable for use on light-weight devices.

6.3.3.2 The revocation scheme due to Chang et al.

A revocation scheme based on the LKH can be generated from the revocation scheme described by Chang et al. in [21].

A maximum number of users 2^N is specified. This is the number of key sets in the system, and no user may be allowed to know more than one key set. There are $2N$ keys $\{k_{0,0}, k_{1,0}, k_{0,1}, k_{1,1}, \ldots, k_{0,N-1}, k_{1,N-1}\}$, and every key set contains either $k_{0,i}$ or $k_{1,i}$ for all values of i.

Key Establishment Scheme 6.9: The revocation scheme described by Chang et al. in [21].

Initialisation

- $\mathcal{T}$ generates $2N$ independent keys $\{k_{0,0}, k_{1,0}, k_{0,1}, k_{1,1}, \ldots, k_{0,N-1}, k_{1,N-1}\}$ uniformly at random .
- For each user that registers to the group, $\mathcal{T}$ assigns them a unique N-bit identifier $X_{N-1}X_{N-2}\cdots X_0$.
- The TA securely sends user $X_{N-1}X_{N-2}\cdots X_0$ all keys of the form $k_{X_i,i}$.

Key establishment

- The TA generates a session key SK and broadcasts it encrypted with $k_{0,0}$ and again encrypted with $k_{1,0}$, so that all users can obtain it.

Update — User revoke

- The TA considers the list of users to be revoked and finds a minimal set of derived keys $\{\kappa_0, \ldots, \kappa_{r-1}\}$, such that all non-revoked users can find at least one, while ensuring that all freshly revoked users cannot. Each key κ_i is an exclusive-or of a subset of $\{k_{0,0}, k_{1,0}, \ldots, k_{0,N-1}, k_{1,N-1}\}$. This is not a simple problem, and the best method to solve it is outside the scope of this document.[1]

[1] According to [21], the TA should use the Quine-McCluskey algorithm of [81].

- The TA broadcasts the encryption

$$E_{SK}(E_{\kappa_0}(SK^*), \ldots, E_{\kappa_{r-1}}(SK^*)).$$

In [21] the authors do not provide a method for user add, but we suggest below an extension of the scheme to allow new users to join it.

Update — User add

- If there are already 2^N users in the scheme then the new user is rejected until one of the existing users leaves the scheme.
- The new user is assigned an unassigned N-bit identifier. If the user has previously been part of the system then they must be allocated the same N-bit identifier as they had previously. If this identifier is currently allocated to another user then they may not presently rejoin the scheme (otherwise they would know the keys from both key sets, which would total more than N).
- The TA securely sends the new user the keys associated with their identifier and the current session key.

Analysis

	Base	Add	Revoke
Secret storage	N	No extra	No extra
Public communication	0	0	$\approx r$
TA computation cost	$2t_E$	0	$(r+1)t_E$
User computation cost	t_E	0	$2t_E$

We can use a result from Section 5.2.3.1 to improve the scheme further for the TA. The TA's storage can be vastly reduced by generating the $2n$ keys from a master key $k_{\mathcal{T}}$ known only to $\mathcal{T}$. For example, we can set $k_{0,i} = f(0||i||k_{\mathcal{T}})$ and $k_{1,i} = f(1||i||k_{\mathcal{T}})$. In this way the TA can easily compute any of the keys from the master key, and knowledge of some of the user keys does not compromise any others.

Scheme 6.9 does not protect against collusion. If a freshly revoked user colludes with any other revoked user then they will usually be able to compute the new session key (there are some exceptions).

6.3.3.3 Key recovery for logical key hierarchies

Some key distribution schemes address the problem that occurs if a user misses a key update message relevant to them (for example [77, 106]).

The simplest solution is for the user to request the current key encryption keys from $\mathcal{T}$. However each time $\mathcal{T}$ responds to such a request there is a large communication cost incurred.

In [63] Kurnio et al. describe a variation of the LKH which solves the problem more efficiently. We first give the underlying scheme, which is an extension of the scheme in [20] due to Canetti et al. to allow for simultaneous addition and revocation of users.

Key Establishment Scheme 6.10: The first key establishment scheme described by Kurnio et al. in [63] for an a-ary tree.

Initialisation

- $\mathcal{T}$ creates an a-ary tree $\mathcal{G}$ with n leaves and assigns a random key x_v to each vertex v. Each user is assigned a unique leaf, and each value x_v is securely sent to all users whose leaf is a descendant of v. If a user's leaf is at depth d then they will receive d values.

- There is one KPS communication group per vertex:

$$\mathcal{C}' = \{B_v = \{U_i : U_i\text{'s leaf is a descendant of } v\}\}.$$

Key establishment

- If $\mathcal{T}$ wants to communicate with a group A then he decomposes A into a minimal set $\{B_{v_0}, \ldots, B_{v_r}\}$ of disjoint subgroups in $\mathcal{C}'$.

- $\mathcal{T}$ picks a random session key k_A.

- For each subgroup B_{v_j}, $\mathcal{T}$ encrypts k_A with the group key x_{v_j} and broadcasts $E_{x_{v_j}}(k_A)$.
- Each user in A can decrypt precisely one of the encryptions of k_A.

Update — Multiple user add and revoke

- $\mathcal{T}$ deletes the leaves of any revoked users and creates leaves for any new users (this may require the creation of other vertices, depending on the restrictions imposed on the shape of the tree).
- $\mathcal{T}$ deletes any vertices with only one child, and promotes the child vertex to take their place.
- $\mathcal{T}$ creates a subtree of the vertices which the revoked users know the value of, or which the new users need to know the value of. They then split the subtree into a set of disjoint chains C.
- $\mathcal{T}$ creates a hash chain for each chain in C, seeded with a random value, and assigns the hash chain values to the vertices of the chain.
- $\mathcal{T}$ sends any new user their leaf value.
- For every vertex in the subtree, $\mathcal{T}$ broadcasts an encryption of its value with any child from main tree, except for parent-child pairs which are part of the same hash chain.

We give an example of the multiple user add and revoke phase in Figure 6.5.

The full scheme from [63] also includes a key recovery system to counter the problem of users missing the update messages. The system is designed for the recovery of a single key, and so the full scheme is best considered as a revocation scheme with the root key being the session key.

For each session key K_i, $\mathcal{T}$ creates two key recovery values, P_i chosen at random, and $S_i = K_i \oplus P_i$. With the ith session update, the TA broadcasts $2t$ encrypted key recovery values, $E_{K_i}(P_{i-t}, \ldots, P_{i-1}, S_{i+1}, \ldots, S_{i+t})$. Since P_* and S_* are chosen with a random element, no information about session keys

can be recovered from a single key recovery message. However a user can recover session key K_i if they can find P_i and S_i, by decrypting one of the key recovery messages from sessions $(i-t)$ to $(i-1)$, and one of the key recovery messages from sessions $(i+1)$ to $(i+t)$.

We present a flaw in the key recovery from the full scheme in [63]. We will refer to the diagrams in Figure 6.5. We start with the left diagram and assume that user U_5 initially knows k_{10}, k_2 and k_0, where $k_0 = K_1$ the session key from the first session. Then the scheme is updated to give the right diagram in Figure 6.5, but U_5 does not receive the update message, and consequently cannot decrypt anything during the second session, including the re-keying information. Assume that for the next t sessions the changes do not affect the key $f(k'_{13})$ at the vertex U_8 and U_5 have in common, and so U_5 cannot recover any more information, including the t key recovery messages. Consequently U_5 has permanently missed the session information from the second session. U_5 will continue to miss another session for each

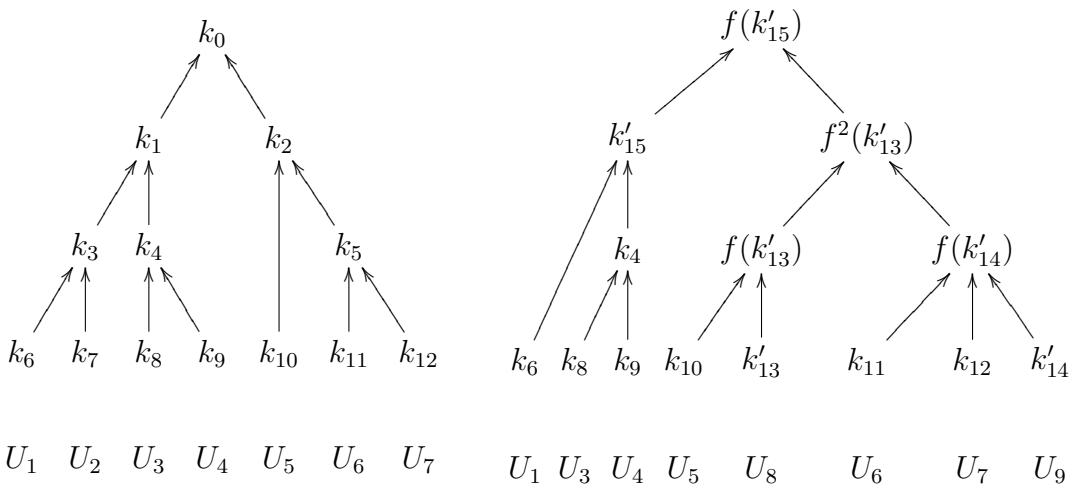


Figure 6.5: The left diagram shows the keys in the initial setup, and the right diagram shows the updated diagram. User U_2 is removed, and users U_8 and U_9 are added. New keys are marked with a dash for clarity. U_8 and U_9 are securely sent their leaf keys, and then $\mathcal{T}$ broadcasts the following encryptions $E_{k_6}(k'_{15})$, $E_{k_4}(k'_{15})$, $E_{k_{10}}(f(k'_{13}))$, $E_{k_{11}}(f(k'_{14}))$, $E_{k_{12}}(f(k'_{14}))$, $E_{f(k'_{14})}(f^2(k'_{13}))$, $E_{f^2(k'_{13})}(f(k'_{15}))$, from which all users can update their keys.

session that $f(k'_{13})$ remains unchanged.

This problem can be overcome, but the only methods we have found greatly increase the communication cost.

6.4 Group key agreement schemes

A third type of key establishment scheme that we will not consider in any detail are group key agreement schemes. In these schemes users communicate amongst themselves during the key establishment phase to derive group keys. A typical scenario which is well suited for key agreement schemes, is secure conferencing. Here users, who already 'know' or 'trust' each other, may wish to communicate as a group in a secure environment.

A large proportion of these schemes are based on the Diffie-Hellman key exchange protocol [34].

If the group have access to a secure channel then one user can pick a random group key and securely send it to the others. If the users do not have access to a secure channel then the best known schemes that are secure against eavesdroppers all use public key cryptography.

As hash structures do not appear to be of great use in key agreement, we will not consider these further.

6.5 Extending the lifetime of a key

In this section we compare several hash-based key refreshment schemes. We propose a scheme based on the pseudo-random number generator, and a simple extension with improved efficiency.

6.5.1 Introduction

In many applications keys are refreshed periodically. This may be required when new members join or members leave a group.

Also, assuming there is a brute force adversary who systematically guesses the key, periodic key refreshment will increase the expected time before the

key is guessed, as well as ensuring that the adversary can only use the compromised key for a limited time.

A *key refreshment scheme* is a scheme which facilitates the update of group keys. Usually the update will be instigated by a trusted authority or will be carried out periodically.

Although the literature is not consistent, we choose to define 'backward secrecy' and 'forward secrecy' in what we believe to be the most standard way (for example [35, 61]).[2]

Definition 6.6. A key refreshment scheme has **backward secrecy** if all subsets of future session keys reveal no information about past session keys.

A key refreshment scheme has **forward secrecy** if all subsets of past session keys reveal no information about future session keys.

Some key refreshment schemes derive the session key from a long term secret, while others derive it from a secret value that is often updated. In both cases each user stores more information than just the session key, so there will be a difference between an adversary who discovers a session key and an adversary who gains access to all the values held by a user. We propose the following two definitions, which relate to the latter type of adversary.

Definition 6.7. A key refreshment scheme has **strong backward secrecy** if it has backward secrecy, and an adversary who has access to all values held by all current users cannot compute past session keys.

A key refreshment scheme has **strong forward secrecy** if it has forward secrecy, and it is possible to revoke users from the scheme, such that a collusion of all revoked users cannot compute future session keys.

We summarise each of the key refreshment schemes in this section with a table. We denote the secrecy provided by the letters B, F, SB and SF for backward, forward, strong backward and strong forward secrecy respectively. For each scheme we also give the storage each user requires (as for the 'communication cost' in Section 4.2.5). We also compare the computation needed

[2]We note that some authors swap the definitions of backward and forward secrecy (for example [29]).

to compute the next key from the existing one for each scheme (using the notation from Section 6.1).

6.5.2 Key refreshment using the session number

For any key establishment scheme it is possible to refresh the group keys by initialising the scheme again, and for some it is possible just to run the key establishment phase again. However, if instead we use the group key as a generator (or 'master key') for other keys, we can reduce the amount of group communication required.

One of the simplest ways to do this securely is by using a preimage-resistant hash function f to hash the session number with the master key to form the session key.

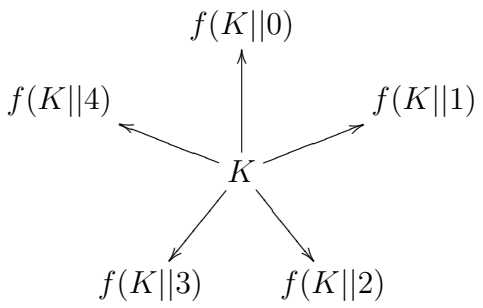


Figure 6.6: The session keys for the first five sessions as generated by Key Refreshment Scheme 6.11.

Key Refreshment Scheme 6.11: Key refreshment using a session number. We assume that each group A has a master key K_A set up using a key establishment scheme.

Key establishment

- In session s the session key for group A is $k_{A,s} = f(K_A||s)$.

Analysis

User storage	3
Secrecy	F, B
User computation cost	t_f

A disadvantage of this scheme is that each user must store the master key K_A and also the session number s. Assuming each user is in a large number of groups, and the session number is the same for all groups, then storing one additional value is not much of a sacrifice.

Another shortcoming of the above scheme is that, although it offers backward and forward secrecy, it does not offer strong backward or strong forward secrecy.

6.5.3 A hash chain based key refreshment scheme

We can use a hash chain to achieve strong backward secrecy, removing the need for each user to keep track of the session number.

Key Refreshment Scheme 6.12: A key refreshment scheme using a hash chain based on a preimage-resistant hash function f in order to avoid each user storing a session number. We assume that each group A has a master key K_A set up using a key establishment scheme.

Key establishment

- In session s the session key for group A is $k_{A,s} = f^s(K_A)$. However, since the user already knows $k_{A,s-1}$, they can compute the next session key as $k_{A,s} = f(k_{A,s-1})$.

Analysis

User storage	1
Secrecy	B, SB
User computation cost	t_f

Although this scheme is very suitable if strong backward secrecy is required, it is not suitable if it is possible that a key might be guessed or leaked. If one session key is compromised then all future session keys are also compromised.

6.5.4 A chained pseudo-random number generator based key refreshment scheme

A simple scheme which avoids the problem of a compromised session key leaking the value of future session keys is to use the chained pseudo-random number generator (Section 3.4.4).

Key Refreshment Scheme 6.13: A key refreshment scheme based on the chained pseudo-random number generator. We assume that each group A has a master key K_A set up using a key establishment scheme.

Key establishment

- In session s the session key for group A is $k_{A,s} = f_2(f_1^s(K_A))$.
- Each user does not need to store the master key K_A, but instead should store the value $x_{A,s} = f_1^s(K_A)$ from the underlying hash chain (see Figure 6.7). With this, the user can compute the next session key as $k_{A,s+1} = f_2(f_1(x_{A,s}))$.

Analysis

User storage	2
Secrecy	F, B, SB
User computation cost	$2t_f$

This scheme is much more resistant to compromised session keys than Key Refreshment Scheme 6.12. If an adversary manages to obtain a session key then it gives them no information about previous or future session keys. It is slightly slower than the hash chain based scheme, requiring two hashes per session, and it requires the user to store two values (the session key and the hash chain value), but this should be efficient enough for all but the most light-weight of applications.

Even though each session key does not reveal future session keys, the chained pseudo-random number generator scheme does not provide strong forward secrecy, as any user can compute all session keys ahead of the point they join the scheme.

Any user joining the scheme will not be able to compute previous keys.

$$\begin{tikzcd}
K_A = x_{A,0} \arrow[r, "f_1"] \arrow[d, dashed, "f_2"] & x_{A,1} \arrow[r, "f_1"] \arrow[d, dashed, "f_2"] & x_{A,2} \arrow[r, "f_1"] \arrow[d, dashed, "f_2"] & \cdots \\
k_{A,0} & k_{A,1} & k_{A,2} & \cdots
\end{tikzcd}$$

Figure 6.7: The key refreshment values for group A using Key Refreshment Scheme 6.13. The top row of values forms a hash chain. One value is stored per session in order to derive future session keys.

We observe that Scheme 6.13 can be improved if all users are aware when a user is added to the scheme. Users simply compute their keys as in Scheme 6.11 until a user is added, at which point they hash the master key, as in Scheme 6.13. An example of this scheme is given in Figure 6.8.

$$\begin{tikzcd}
k_{A,0} & k_{A,2} & k_{A,3} & k_{A,6} & k_{A,7} \\
K_A = x_{A,0} \arrow[u] \arrow[d] \arrow[r] & x_{A,1} \arrow[ul] \arrow[ur] \arrow[dl] \arrow[dr] \arrow[rr] & & x_{A,2} \arrow[ul] \arrow[ur] \arrow[d] \arrow[r] & \cdots \\
k_{A,1} & k_{A,4} & k_{A,5} & k_{A,8} &
\end{tikzcd}$$

Figure 6.8: An extension of Scheme 6.13 so that users only need to compute two hashes when a user is added.

6.5.5 Key refreshment with strong forward secrecy

If we require that a key refreshment scheme has strong forward secrecy then we need to be able to revoke users. Consequently, it is a good idea to use a revocation scheme to generate the group key. We can use any of the schemes from Section 6.3.

Many revocation schemes will provide strong forward and strong backward secrecy. They typically require much greater user storage than the other key refreshment schemes that we have seen in this section.

6.6 Conclusions

In this chapter we compared several key establishment schemes. We presented a generalisation of an inverted hash tree key predistribution scheme (KPS). We suggest that it would be of interest to find the optimal tree shape for this scheme.

We presented a KPS for lattice-shaped hierarchy, and generalised it to many more hierarchies. It is an open problem to find, given a hierarchy, the most efficient DAG to form a KPS using Key Establishment Scheme 6.8.

We presented a flaw in the key recovery system of the key distribution scheme by Kurnio et al. We also exhibited some key refreshment schemes based on hash structures including the chained pseudo-random number generator, which we believe to be very well suited to light-weight applications.

Chapter 7

Other applications and future work

In this chapter we provide a brief overview of several other cryptographic applications of hash structures. In each case we feel that it may be possible to make improvements on the techniques described in the current literature, but have not had time to explore these in detail in this thesis.

7.1 Micropayment schemes

One of the best known uses of hash chains is in micropayment schemes. A company that frequently deals with small amounts of money, such as a mobile phone operator, could potentially suffer large administrative costs if it processed these transactions every time. Micropayment schemes are specifically designed to efficiently process large volumes of small payments.

One of the most well known micropayment schemes is Payword, which was simultaneously developed by a number of different authors [50, 103, 119]. Note the similarity between this scheme and Entity Authentication Scheme 4.9.

Micropayment Scheme 7.1: A simplified version of Payword, with customer A, service provider B and trusted authority $\mathcal{T}$.

Initialisation

- A pays $\mathcal{T}$ the sum of n pence, which will be spent paying B in penny increments.
- $\mathcal{T} \xrightarrow{S} A : x$ for some random value x.
- $\mathcal{T} \xrightarrow{A} B : f^n(x)$.

Payment number i

- When A wants to pay B the ith penny, A sends B the hash chain value $f^{n-i}(x)$.
- B checks that the hash of this value matches the value received in payment $i-1$, and stores $f^{n-i}(x)$ in place of $f^{n-(i-1)}(x)$.

Collection of payments

- Whenever B wishes to collect the payments to date, they send $\mathcal{T}$ the most recent value that they have received.
- $\mathcal{T}$ hashes this value repetitively to compare with $f^n(x)$, and thereby verify that it is part of the hash chain.
- $\mathcal{T}$ then pays B appropriately from A's funds.

The basic idea is adapted or extended by other papers, for example [102] and [60]. A scheme by Lin et al. [74] is based on Payword, but instead uses values from a hierarchical chain construction as opposed to a hash chain. We observe that many other n-time entity authentication schemes from Chapter 4 could be extended into micropayment schemes (for example Schemes 4.16, 4.18 and 4.19), however we have not had an opportunity to analyse these fully.

7.2 Auctions

A similar application to micropayments is auctions. Here the bidders provide the auctioneer with commitments to pay, as opposed to actual payments. An example of an auction scheme that makes use of a hash chain for these commitments is given in [135]. This scheme is for an *English auction*. However other types of auction exist, including the *first-price sealed-bid auction*.

Definition 7.1. An **English auction** is an auction in which the auctioneer accepts increasingly higher bids, until no competing bidder improves on the current bid (within some reasonable time). The item is sold to the highest bidder at a price equal to their bid.

Definition 7.2. A **first-price sealed-bid auction** is a two-phase auction in which all bidders privately submit their bids during the *bidding phase*, after which the auctioneer views the bids and determines the highest bidder during the *opening phase*. The item is sold to the highest bidder at a price equal to their bid.

Suzuki et al. use hash chains to facilitate an online first-price sealed-bid auction in [136].[1] Their hash-based scheme has a major advantage over any traditional physical implementation. In a traditional physical scheme the second phase must either be made public, in which case rival parties will find out important information about one another's valuations, or kept secret, in which case the bidders are forced to trust that the auctioneer has not colluded with any of the bidders. In the hash-based scheme all users can be assured of the auction's correctness, without losing bids being revealed. We now give a simplified version of their scheme.

[1] In fact their scheme is better described as a *Dutch auction*, which in theory should have the same outcome as a sealed-bid first-price auction [80]. A *Dutch Auction* is an auction in which the auctioneer starts with a high asking price and gradually lowers it until a bidder is willing to accept the auctioneer's price.

Auction Scheme 7.2: A simplified version of the first-price sealed-bid auction from [136]. The auctioneer advertises the lot being auctioned, together with some system parameters: a maximum price of m pence, which should be chosen higher than anyone will bid, and two distinct bit strings z_{yes} and z_{no}.

Bidding phase

- Each bidder makes their valuation of the lot: p pence ($p < m$). Each bidder chooses a seed at random x_{p-1}. They create a chain with values:

$$x_p = f(z_{\text{yes}}||x_{p-1});$$
$$x_i = f(z_{\text{no}}||x_{i-1}), \text{ for } p+1 \leq i \leq m.$$

- Each bidder securely sends x_m to the auctioneer.

Opening phase

- The auctioneer announces each price decreasing from $(m-1)$ pence.
- At i pence, each bidder sends the value x_i to the auctioneer.
- The auctioneer checks if $x_{i+1} = f(z_{\text{no}}||x_i)$ for each user. If this holds for all users then the auctioneer announces the next price ($i-1$ pence).
- If the values do not match then the auctioneer checks if $x_{i+1} = f(z_{\text{yes}}||x_i)$. If this is the case, they announce the end of the auction and the winning bid. Otherwise the bidder has not submitted a valid chain and the auctioneer should take appropriate action.

In [110], Prakobpol and Permpoontanalarp present a variation of the previous scheme using a hierarchical chain construction. In [99], Omote and Miyaji present an auction scheme using hash chains differently; each chain value is used to mask a digit of a user's bid.

In 1961, Vickrey proposed an alternative to the first-price sealed-bid auction, which has become known as the *Vickrey auction* [143].

Definition 7.3. A **Vickrey auction** (or **second-price sealed-bid auction**) is a two-phase auction in which all bidders privately submit their bids during the bidding phase, and then the auctioneer views the bids and determines the highest bidder during the opening phase. The item is sold to the highest bidder at a price equal to the second highest bid.

Any computational implementation of a Vickrey auction must address issues that do not affect first-price sealed-bid auctions [123]. One such example is that a corrupt auctioneer must not be able to insert additional bids after the opening phase has begun, otherwise they could insert a bid between the highest and second highest bids in order to increase their revenue.

Many Vickrey auction schemes exist, including several that do not depend on public key cryptography [17, 75, 90]. However all of these solutions require multiple independent auctioning authorities, and some reveal private bid values to some of these authorities (ideally in a Vickrey auction only the second highest bid should be obtainable).

We created a draft Vickrey auction scheme using hash chains, based on Auction Scheme 7.2, but it is prohibitively costly in terms of computation. It remains an open problem to design a hash-based Vickrey auction with a single auctioneer, such that all bidders can be satisfied that the auction has been carried out correctly [139].

7.3 Pseudo-random number generation

As mentioned in Section 2.4, randomness is intrinsically linked with hash functions, and many applications of hash functions require their outputs to be pseudo-random. There are several well-known ways of generating pseudo-random numbers using a hash function.

Perhaps the most common technique for generating a pseudo-random sequence $\{x_0, x_1, \ldots\}$ is using a star-shaped inverted hash tree defined as follows:

$$x_i = f(s||i), \text{ where } s \text{ is the seed.}$$

In Section 6.5.2 we saw how this method of generating pseudo-random numbers could be used for key refreshment.

In Section 4.3.3 we saw that in some situations it may be appropriate to use the chained pseudo-random number generator (Definition 3.10). However this scheme will suffer if the underlying hash chain is comparable in length to the size of the hash's range (see Section 3.1.4.5 for more details).

Other methods of combining hash functions to create pseudo-random numbers exist, including [46] and [133].

If a hash function can be used to generate pseudo-random numbers from a seed then it can also be used to encrypt a message. Many stream ciphers use the key as a seed to generate pseudo-random numbers, and then exclusive-or it with the message. One recent example of a stream cipher that is based on a hash function in this way is Salsa20 [8].

7.4 Information sealing

In this section we look at two methods of sealing information. The first allows information to be kept secret for a certain amount of time, and the second facilitates the efficient release of consecutive records from an ordered set of data.

7.4.1 Time-release cryptography

In certain applications it may be useful to seal a piece of information such that it can only be read after a certain amount of time has passed. For example, in a sealed-bid auction, a bidder may want to keep their bid secret until the opening phase.

In [120] Rivest et al. suggest that a trusted agent $\mathcal{T}$ using a hash chain could provide a service where they respond to two types of request. If a user asks for the current hash chain value, $\mathcal{T}$ will provide it. From this value the user can compute all previous hash chain values. If the user asks $\mathcal{T}$ to encrypt a message m so that it can be decrypted at time t, then the agent will respond with the encryption $E_{k_t}(m)$, where k_t is the hash chain value that will be released at time t.

7.4.2 Interval release cryptography

In Section 6.2.3.5 we saw a scheme from [58] which facilitated the possibility of interval release cryptography. A server creates two hash chains x_* and y_*, each of length n. From these, the server defines an ordered list of keys of the form $(x_i \oplus y_{n-i})$. The server can then provide a user with two values x_i and y_j, and using these values the user can find keys for the interval $[i, j]$.

We note that this can easily be extended to more dimensions by using two more hash chains for each additional dimension.

For example, pixel data in a video may have the three dimensions column, row and time. The video's owner can create six hash chains $x_{\text{column},*}$, $x_{\text{row},*}$, $x_{\text{time},*}$, $y_{\text{column},*}$, $y_{\text{row},*}$, $y_{\text{time},*}$ and use these to create a key for each pixel of the form: $x_{\text{column},c} \oplus x_{\text{row},r} \oplus x_{\text{time},t} \oplus y_{\text{column},C-c} \oplus y_{\text{row},R-r} \oplus y_{\text{time},T-t}$.

The video's owner can then provide the same copy of the video to many users, but provide them with a different viewing configuration. One user may pay for widescreen access, while another may pay to see extended footage. The owner only needs to send six hash chain values for any configuration that a user pays for.

7.5 Generating rainbow tables

Lastly we consider a very different application of hash structures. Many password based applications store the hash of the password (computed using a preimage-resistant hash function f), as in Entity Authentication Scheme 4.3. Assuming the user is allowed to choose their own password, there is a reasonable chance that it will be found in a dictionary, or list of common passwords. Typically, a search of these lists will be computationally easy, especially compared to the computationally infeasible task of a search of all potential passwords.

In some scenarios the adversary may be able to obtain the hash of the password, and may wish to avoid submitting large numbers of incorrect passwords. An efficient way to find the password is using *rainbow tables* [97].

To form a rainbow table, we use the hash function f and a family of

reduction functions g_* with domain equal to the range of f, and range equal to the list of common passwords L. The rainbow table $x_{*,*}$ has r rows and $(l+1)$ columns, and has entries from L defined by:

$$x_{i,j} = \begin{cases} \text{a random member of } L & \text{if } i = 0 \\ g_i(f(x_{i-1,j})) & \text{otherwise.} \end{cases}$$

We note that by considering the combination of the hash function and the reduction function as a new hash function, each row of the rainbow table $x_{*,j}$ is a rainbow chain of length l (Definition 3.8).

To reduce the memory required to store the rainbow table, only the first and last columns are stored (that is $\{x_{0,*}\}$ and $\{x_{l,*}\}$).

If a password is hashed somewhere in our rainbow table then we can find the password p from the hash output $f(p)$. We search each column of the table in turn by applying the appropriate sequence of reductions and hashes to $f(p)$, and check to see if the result is a member of $\{x_{l,*}\}$. If it is, then we find the first value of that row, and hash and reduce until we obtain the preimage p.

The precursor to rainbow tables was based on hash chains [51]; however rainbow chains are better suited to this application.

7.6 Conclusions

In this chapter we gave an overview of the application of hash structures to micropayments, auctions, pseudo-random number generation, interval release cryptography and hash preimage computation, and indicated several open problems.

Chapter 8

Conclusions

In this thesis we have considered many applications of hash functions and hash structures to cryptography.

We began in Chapter 2 by examining the definitions of types of hash function, and found that there exists a certain amount of conflict in the literature. We presented a set of three definitions suitable for this thesis. We explored the relationship between hash functions and pseudo-random functions. We suggested a parameterisation of hash functions by security and output size. We summarised the progress made by NIST in defining properties required from a new hash function SHA-3.

In Chapter 3 we presented a toolkit of hash structures, which we believe to be the first comprehensive overview of its kind. We briefly studied the relationship between the properties of a hash function and the hash chain formed from it. We compared infinite-length hash chains with hash chains and observed that infinite-length hash chains are less suitable for use in restricted environments. We made a distinction between hash trees and Merkle trees, and defined a useful type of tree, the almost perfect a-ary tree. We gave a formal definition of a generalised hash directed acyclic graph (DAG).

In Chapter 4 we studied entity authentication, and the schemes that can be formed using the hash structures from Chapter 3. We also presented a new notion of public verifiability, and several hash-based schemes which possessed

it. We identified two adversaries that were useful in comparing existing and new entity authentication schemes. We presented a novel hash-based scheme with minimal storage requirements for authenticating and verifying parties. We created two new entity authentication schemes which provide entity authentication to many verifying parties. We showed a relationship between the use of Merkle trees in a entity authentication scheme and a one-time signature scheme. We suggested an extension of the hierarchical chain construction to an entity authentication scheme suitable for sensor networks. We also suggested entity authentication schemes based on sandwich chains, comb skipchains and hash chains with breakpoints.

In Chapter 5 we studied the application of hash structures to signature schemes. We compared one-way trapdoor based signature schemes with hash-based one-time signature schemes. We proved that any set of Vaudenay's rake signature patterns with constant average chain position is a signature scheme. We then proved some results about the size of these signature schemes. We showed that for a given signature space size there is not a unique best choice for the optimal Vaudenay's rake signature scheme used. We proved that for any signature scheme based on a hash tree there is another signature scheme based on a Merkle sub-tree that has the same number of signature patterns.

We then presented a novel algorithm to find a large one-time signature scheme based on a given hash structure. We also presented a scheme to find the largest signature scheme, although this proved prohibitively slow for large structures. We used our algorithms to explore the relationship between the number of vertices in a graph, and the efficiency of one-time signature scheme that it admitted. We generalised a scheme by Bleichenbacher and Maurer and demonstrated that the original scheme is the most efficient in the limit. We created two efficient one-time signature schemes, which also have simple message to signature space maps. These signature schemes are the most efficient such schemes that we know of. We also explored k-time signature schemes, and presented the concepts of perforated and porous k-time signature schemes. We studied the definition of efficiency for k-time signature schemes, and presented two useful definitions. We suggested a new method for the construction of a porous k-time signature scheme.

In Chapter 6 we studied the application of hash structures to key establishment schemes and compared many key predistribution and key distribution schemes (KPSs and KDSs) from the literature. We combined ideas from two key predistribution schemes to create the inverted hash tree KPS. We presented a hash chain based KPS for lattice-shaped hierarchies. We generalised this scheme, replacing hash chains with generalised hash DAGs, making the scheme suitable for use with many other hierarchies. We observed that a key escrow scheme by Joye and Yen can be applied to any set of group keys formed by a KPS. We improved a scheme by Chang et al. to facilitate the inclusion of new users and to reduce the storage space required by trusted authority. We presented a flaw in a KDS due to Kurnio et al. We noted the inconsistency in the literature over the definitions of forward and backward secrecy. We proposed definitions of strong forward and backward secrecy, and used these definitions to compare several hash-based key refreshment schemes. We proposed a key refreshment scheme based on the chained pseudo-random number generator, and a simple extension to improve its efficiency.

In Chapter 7 we briefly studied the application of hash structures to several other areas of cryptography, including micropayments, auctions and pseudo-random number generators. We also mention hash structure based schemes for interval release cryptography and preimage computation. We generalise an interval release scheme due to Joye and Yen to many dimensions.

Bibliography

[1] S. G. Akl and P. D. Taylor. Cryptographic solution to a problem of access control in a hierarchy. *ACM TOCS*, 1(3):239–248, 1983.

[2] R. Anderson. The classification of hash functions. In *IMA Conference in Cryptography and Coding*, pages 83–94, 1993.

[3] A. Babenhauserheide. Phex 3.0.0 released. Website from 7th June 2009. `http://www.phex.org/mambo/content/view/80/58/`.

[4] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology, CRYPTO '96, LNCS*, volume 1109, pages 1–15, 1996.

[5] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *CCS 93, ACM*, pages 62–73, 1993.

[6] M. Bellare and P. Rogaway. The exact security of digital signatures – how to sign with RSA and Rabin. In *Advances in Cryptology, EUROCRYPT '96, LNCS*, volume 1070, pages 399–416, 1996.

[7] F. Bergadano, D. Cavagnino, and B. Crispo. Chained stream authentication. In *SAC '00, LNCS*, volume 2012, pages 144–157, 2001.

[8] D. J. Bernstein. The Salsa20 stream cipher. In *SKEW '05, ECRYPT*, 2005. `http://www.ecrypt.eu.org/stream/salsa20p2.html`.

[9] K. Bicakci and N. Baykal. Infinite length hash chains and their applications. In *WETICE '02, IEEE*, pages 57–61, 2002.

[10] D. Bleichenbacher and U. M. Maurer. Directed acyclic graphs, one-way functions and digital signatures. In *Advances in Cryptology, CRYPTO '94, LNCS*, volume 839, pages 75–82, 1994.

[11] D. Bleichenbacher and U. M. Maurer. On the efficiency of one-time digital signatures. In *Advances in Cryptology, ASIACRYPT '96, LNCS*, volume 1163, pages 145–158, 1996.

[12] D. Bleichenbacher and U. M. Maurer. Optimal tree-based one-time digital signature schemes. In *STACS '96, LNCS*, volume 1046, pages 363–374, 1996.

[13] R. Blom. An optimal class of symmetric key generation systems. In *Advances in Cryptology, EUROCRYPT '84, LNCS*, volume 209, pages 335–338, 1985.

[14] C. Blundo, A. De Santis, A. Herzberg, S. Kutten, U. Vaccaro, and M. Yung. Perfectly-secure key distribution for dynamic conferences. In *Advances in Cryptology, CRYPTO '92, LNCS*, volume 740, pages 471–486, 1993.

[15] C. Boyd and A. Mathuria. *Protocols for Authentication and Key Establishment*. Springer, 2003.

[16] P. G. Bradford and O. V. Gavrylyako. Hash chains with diminishing ranges for sensors. In *ICPPW '04, IEEE*, pages 77–83, 2004.

[17] F. Brandt. Cryptographic protocols for secure second-price auctions. In *CIA '01, LNAI*, volume 2182, pages 154–165, 2001.

[18] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[19] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast security: A taxonomy and some efficient constructions. In *INFOCOM '99, IEEE*, volume 2, pages 708–716, 1999.

[20] R. Canetti, T. Malkin, and K. Nissim. Efficient communication-storage tradeoffs for multicast encryption. In *Advances in Cryptology, EUROCRYPT '99, LNCS*, volume 1592, pages 459–474, 1999.

[21] I. Chang, R. Engel, D. Kandlur, D. Pendarakis, and D. Saha. Key management for secure internet multicast using boolean function minimization techniques. In *INFOCOM '99, IEEE*, volume 2, pages 689–698, 1999.

[22] S. Chang and M. Dworkin. Workshop report: The first cryptographic hash workshop. Technical report, National Institute of Standards and Technology, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899, 2005. `http://www.csrc.nist.gov/pki/HashWorkshop/2005/HashWshop_2005_Report.pdf`.

[23] J. Crampton, K. Martin, and P. Wild. An exposition of key assignment schemes. Unpublished manuscript, 2005.

[24] J. Crampton, K. Martin, and P. Wild. On key assignment for hierarchical access control. In *CSFW '06, IEEE*, pages 98–111, 2006.

[25] I. B. Damgård. Collision free hash functions and public key signature schemes. In *Advances in Cryptology, EUROCRYPT '87, LNCS*, volume 304, 1987.

[26] I. B. Damgård. *The application of claw free functions in cryptography.* PhD thesis, Aarhus University, Mathematical Institute, 1988.

[27] D. W. Davies and W. L. Price. *The Application of Digital Signatures Based on Public Key Cryptosystems.* National Physical Laboratory, 1980.

[28] D. W. Davies and W. L. Price. *Security for computer networks: An introduction to data security in teleprocessing and electronic funds transfer.* John Wiley & Sons, Ltd., 1984.

[29] H. Delfs and H. Knebl. *Introduction to Cryptography: principles and applications*. Springer, 2002.

[30] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981.

[31] A. Dent and Mitchell C. *User's Guide To Cryptography And Standards*. Artech House Publishers, 2004.

[32] R. Di Pietro, A. Durante, L. V. Mancini, and V. Patil. Short paper: Practically unbounded one-way chains for authentication with backward secrecy. In *SECURECOMM '05, IEEE*, pages 400–402, 2005.

[33] R. Di Pietro, A. Durante, L. V. Mancini, and V. Patil. Addressing the shortcomings of one-way chains. In *ASIACCS '06, ACM*, pages 289–296, 2006.

[34] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[35] W. Diffie, P. C. van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, 1992.

[36] A. Evans Jr., W. Kantrowitz, and E. Weiss. A user authentication scheme not requiring secrecy in the computer. *Communications of the ACM*, 17(8):437–442, 1974.

[37] S. Even. A protocol for signing contracts. *SIGACT News, ACM*, 15(1):34–39, 1983.

[38] W. Feller. *An Introduction to Probability Theory and Its Applications*. Wiley, third edition, 1968.

[39] A. Fiat and M. Naor. Broadcast encryption. In *Advances in Cryptology, CRYPTO '93, LNCS*, volume 773, pages 480–491, 1994.

[40] P. Flajolet and A. M. Odlyzko. Random mapping statistics. In *Advances in Cryptology, EUROCRYPT '89, LNCS*, volume 434, pages 329–354, 1990.

[41] L. Gong. Variations on the themes of message freshness and replay or the difficulty in devising formal methods to analyze cryptographic protocols. In *CSFW '93, IEEE*, pages 131–136, 1993.

[42] V. Goyal. Construction and traversal of hash chain with public links. Cryptology ePrint Archive, Report 2004/371, 2004. `http://eprint.iacr.org/2004/371`.

[43] V. Goyal. How to re-initialize a hash chain. Cryptology ePrint Archive, Report 2004/097, 2004. `http://eprint.iacr.org/2004/097`.

[44] B. Groza and T. Dragomir. On the use of one-way chain based authentication protocols in secure control systems. In *ARES '07, IEEE*, pages 1214–1221, 2007.

[45] E. Gudes. The design of a cryptography based secure file system. *IEEE Transactions on Software Engineering*, 6(5):411–420, 1980.

[46] Z. Gutterman, B. Pinkas, and T. Reinman. Analysis of the linux random number generator. In *SP '06, IEEE*, pages 371–385, 2006.

[47] N. Haller. The S/KEY one-time password system. In *Proceedings of the Symposium on Network and Distributed System Security, ISOC*, pages 151–157, 1994.

[48] N. Haller, C. Metz, P. Nesser, and M. Straw. Requests for comments: 2289, a one-time password system. The Internet Engineering Task Force, February 1998. `ftp://ftp.rfc-editor.org/in-notes/rfc2289.txt`.

[49] L. Harn and W. Hsin. On the security of wireless network access with enhancements. In *WiSe '03, ACM*, pages 88–95, 2003.

[50] R. Hauser, M. Steiner, and M. Waidner. Micro-payments based on iKP. In *SECURICOM '96*, pages 67–82, 1996.

[51] M. E. Hellman. A cryptanalytic time-memory trade off. *IEEE Transactions on Information Theory*, 26:401–406, 1980.

[52] K. Hong, S. Jung, and F. S. Wu. A hash-chain based authentication scheme for fast handover in wireless network. In *WISA '05, LNCS*, volume 3786, pages 96–107, 2006.

[53] Y. Hu, D. Johnson, and A. Perrig. Sead: Secure efficient distance vector routing for mobile wireless ad hoc networks. In *WMCSA '02, IEEE*, pages 3–13, 2002.

[54] Y. Hu, A. Perrig, and D. Johnson. Packet leashes: A defense against wormhole attacks in wireless ad hoc networks. In *INFOCOM '03, IEEE*, volume 3, pages 1976–1986, 2003.

[55] Y. C. Hu, M. Jakobsson, and A. Perrig. Efficient constructions for one-way hash chains. In *ACNS '05, LNCS*, volume 3531, pages 423–441, 2005.

[56] R. Impagliazzo and S. Rudich. Limits on the provable consequences of one-way permutations. In *STOC '89, ACM*, pages 44–61, 1989.

[57] ISO/IEC. *ISO/IEC 9798-4, Information Technology – Security Techniques – Entity Authentication – Part 4: Mechanisms Using a Cryptographic Check Function*. 1999.

[58] M. Joye and S. Yen. One-way cross-trees and their applications. In *PKC '02, LNCS*, volume 2274, pages 346–356, 2002.

[59] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[60] E. Kim, H. Kim, and K. Park. Provisioning protected resource sharing in multi-hop wireless networks. Cryptology ePrint Archive, Report 2006/382, 2006. `http://eprint.iacr.org/2006/382`.

[61] Y. Kim, A. Perrig, and G. Tsudik. Simple and fault-tolerant key agreement for dynamic collaborative groups. In *CCS '00, ACM*, pages 235–244, 2000.

[62] J. Kohl and C. Neuman. Requests for comments: 1510, the Kerberos network authentication service (V5). The Internet Engineering Task Force, September 1993. `ftp://ftp.rfc-editor.org/in-notes/rfc1510.txt`.

[63] H. Kurnio, R. Safavi-Naini, and H. Wang. A secure re-keying scheme with key recovery property. In *ACISP '02, LNCS*, volume 2384, pages 40–55, 2002.

[64] X. Lai and J. L. Massey. Hash functions based on block ciphers. In *Advances in Cryptology, EUROCRYPT '92, LNCS*, volume 658, pages 55–70, 1993.

[65] K. Lam and T. Beth. Timely authentication in distributed systems. In *ESORICS 92, LNCS*, volume 648, pages 293–303, 1992.

[66] K. Lam and D. Gollmann. Freshness assurance of authentication protocols. In *ESORICS 92, LNCS*, volume 648, pages 261–272, 1992.

[67] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[68] L. Lamport. Constructing digital signatures from a one-way function. Technical report, CSL-98, SRI International, October 1979.

[69] L. Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–772, 1981.

[70] J. Lee and D. Stinson. Tree based key distribution patterns. *SAC '05, LNCS*, 3897:189–204, 2006.

[71] S. Lee, H. Kim, and K. Chung. Hash based secure sensor network programming method without public key cryptography. In *WSW '06, ACM*, 2006.

[72] T. Leighton and S. Micali. Large provably fast and secure digital signature schemes based on secure hash functions, 1993. U.S. Patent No. 5,432,852.

[73] Limegroup. Hash tree - limewire consolidated api. Website from 7th June 2009. `http://www.limewire.org/nightly/javadocs/com/limegroup/gnutella/tigertree/HashTree.html`.

[74] I. Lin, M. Hwang, and C. Chang. The general pay-word: A micropayment scheme based on n-dimension one-way hash chain. *Designs, Codes and Cryptography*, 36(1):53–67, 2005.

[75] H. Lipmaa, N. Asokan, and V. Niemi. Secure Vickrey auctions without threshold trust. In *FC '02, LNCS*, volume 2357, pages 87–101, 2003.

[76] D. Liu and P. Ning. Efficient distribution of key chain commitments for broadcast authentication in distributed sensor networks. In *NDSS '03, ISOC*, 2003.

[77] D. Liu, P. Ning, and K. Sun. Efficient self-healing group key distribution with revocation capability. In *CCS '03, ACM*, pages 231–240, 2003.

[78] W. Mao. *Modern Cryptography: Theory and Practice*. Prentice Hall PTR, first edition, 2003.

[79] K. M. Martin. The combinatorics of cryptographic key establishment. *Surveys in Combinatorics, CUP*, pages 223–273, 2007.

[80] P. Mcafee and J. Mcmillan. Auctions and bidding. *Journal of Economic Literature*, 25(2):699–738, 1987.

[81] E. J. McCluskey. Minimization of boolean functions. *The Bell System Technical Journal*, 35(5):1417–1444, November 1956.

[82] D. A. McGrew and A. T. Sherman. Key establishment in large dynamic groups using one-way function trees. *IEEE Transactions on Software Engineering*, 29(5):444–458, 2003.

[83] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC, October 1996. `http://www.cacr.math.uwaterloo.ca/hac/`.

[84] R. C. Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Stanford University, 1979.

[85] R. C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology, CRYPTO '87, LNCS*, volume 293, pages 369–378, 1988.

[86] R. C. Merkle. One way hash functions and DES. In *Advances in Cryptology, CRYPTO '89, LNCS*, volume 435, pages 428–446, 1989.

[87] C. Mitchell and F. Piper. Key storage in secure networks. *Discrete Applied Mathematics*, 21(3):215–228, 1988.

[88] R. A. Mollin. *An Introduction to Cryptography*. CRC Press, Inc., 2000.

[89] R. Morris and K. Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, 1979.

[90] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *EC '99, ACM*, pages 129–139, 1999.

[91] M. Naor and M. Yung. Universal one-way hash functions and their cryptographic applications. In *STOC '89, ACM*, pages 33–43, 1989.

[92] National Institute of Standards and Technology. NIST's policy on hash functions. Website from 31st July 2008. `http://csrc.nist.gov/groups/ST/hash/policy.html`.

[93] National Institute of Standards and Technology. Digital signature standard. Technical Report 186, Federal Information Processing Standards Publications, May 1994.

[94] National institute of standards and technology. FIPS 180-2, secure hash standard, Federal Information Processing Standard (FIPS), publication 180-2. Technical report, Department of Commerce, August 2002.

[95] National Institute of Standards and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. Technical report, Department of Commerce, November 2007.

[96] J. Nechvatal and S. Chang. Workshop report: The second cryptographic hash workshop. Technical report, National Institute of Standards and Technology, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899, 2006. `http://www.csrc.nist.gov/pki/HashWorkshop/2006/SecondHashWshop~2006~Report.pdf`.

[97] P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Advances in Cryptology, CRYPTO '03, LNCS*, volume 2729, pages 617–630, 2003.

[98] T. Okamoto. Provably secure and practical identification schemes and corresponding signature schemes. In *Advances in Cryptology, CRYPTO '92, LNCS*, volume 740, pages 31–53, 1993.

[99] K. Omote and A. Miyaji. An anonymous aution protocol with a single non-trusted center using binary trees. In *ISW '00, LNCS*, volume 1975, pages 108–120, 2000.

[100] C. Padró, I. Gracia, S. M. Molleví, and P. Morillo. Linear key predistribution schemes. *Designs, Codes and Cryptography*, 25(3):281–298, 2002.

[101] H. Pagnia, H. Vogt, and F. Gartner. Fair exchange. *The Computer Journal, OUP*, 46(1):55–75, 2003.

[102] V. Patil and R. Shyamasundar. e-coupons: An efficient, secure and delegable micro-payment system. *Information Systems Frontiers*, 7(4–5):371–389, 2005.

[103] T. P. Pedersen. Electronic payments of small amounts. In *Security Protocols '96, LNCS*, volume 1189, pages 59–68, 1996.

[104] A. Perrig. The BiBa one-time signature and broadcast authentication protocol. In *CCS '01, ACM*, pages 28–37, 2001.

[105] A. Perrig, R. Canetti, J. D. Tygar, and D. Song. Efficient authentication and signing of multicast streams over lossy channels. In *SP '00, IEEE*, pages 56–73, 2000.

[106] A. Perrig, D. Song, and J. D. Tygar. ELK, a new protocol for efficient large-group key distribution. In *SP '01, IEEE*. IEEE Computer Society, 2001.

[107] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler. Spins: security protocols for sensor networks. *Wireless Networks*, 8(5):521–534, September 2002.

[108] J. Pieprzyk, H. Wang, and C. Xing. Multiple-time signature schemes against adaptive chosen message attacks. In *SAC '03, LNCS*, volume 3006, pages 88–100, 2004.

[109] J. M. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[110] N. Prakobpol and Y. Permpoontanalarp. Multi-dimensional hash chain for sealed-bid auction. In *WISA '03, LNCS*, volume 2908, pages 257–271, 2004.

[111] B. Preneel. *Analysis and design of cryptographic hash functions*. PhD thesis, Katholieke Universiteit Leuven, Belgium, 1993.

[112] B. Preneel and P. C. van Oorschot. On the security of iterated message authentication codes. *IEEE Transactions on Information Theory*, 45(1):188–199, 1999.

[113] J. Proos and C. Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves. *Quantum Information and Computation*, 3:317–344, 2003.

[114] M. O. Rabin. Digitalized signatures. *Foundations of Secure Computation, Academic Press*, pages 155–166, 1978.

[115] M. Ramkumar and N. Memon. An efficient key predistribution scheme for ad hoc network security. *IEEE Journal on Selected Areas in Communications*, 23(3):611–621, 2005.

[116] M. Ramkumar and N. Memon. A hierarchical key predistribution scheme. In *EIT '05, IEEE*, 2005.

[117] K. C. Reddy and Divya Nalla. Identity based authenticated group key agreement protocol. In *Advances in Cryptology, INDOCRYPT '02, LNCS*, volume 2551, pages 215–233, 2002.

[118] L. Reyzin and N. Reyzin. Better than BiBa: Short one-time signatures with fast signing and verifying. In *ACISP '02, LNCS*, volume 2384, pages 144–153, 2002.

[119] R. L. Rivest and A. Shamir. PayWord and MicroMint: Two simple micropayment schemes. In *1996 International Workshop on Security Protocols, LNCS*, volume 1189, pages 69–87, 1997.

[120] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical Report MIT/LCS/TR-684, Massachusetts Institute of Technology, 1996.

[121] P. Rogaway. Formalizing human ignorance collision-resistant hashing without the keys. In *Progress in Cryptology, VIETCRYPT '06, LNCS*, volume 4341, pages 211–228, 2006.

[122] P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *FSE '04, LNCS*, volume 3017, pages 371–388, 2004.

[123] T. Sandholm. Issues in computational Vickrey auctions. *IJEC*, 4(3):107–129, 2000.

[124] R. S. Sandhu. Cryptographic implementation of a tree hierarchy for access control. *Information Processing Letters*, 27(2):95–98, 1988.

[125] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, 1948.

[126] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[127] G. J. Simmons. *Contemporary Cryptology: The Science of Information Integrity*. IEEE, 1994.

[128] D. R. Simon. Finding collisions on a one-way street: Can secure hash functions be based on general assumptions? In *Advances in Cryptology, EUROCRYPT '98, LNCS*, volume 1403, pages 334–345, 1998.

[129] W. Simpson. Requests for Comments: 1994, PPP Challenge Handshake Authentication Protocol (CHAP). The Internet Engineering Task Force, August 1996. `ftp://ftp.rfc-editor.org/in-notes/rfc1994.txt`.

[130] N. P. Smart. An identity based authenticated key agreement protocol based on the Weil pairing. *Cryptology ePrint Archive*, 2001. `http://eprint.iacr.org/2001/111`.

[131] W. R. Speirs II and S. S. Wagstaff Jr. Dynamic cryptographic hash functions. Cryptology ePrint Archive, Report 2006/477, 2006. `http://eprint.iacr.org/2006/477`.

[132] E. Sperner. Ein Satz über Untermengen einer endlichen Menge. *Mathematische Zeitschrift*, 27(1):544–548, 1928.

[133] J. Staddon, S. Miner, M. Franklin, D. Balfanz, M. Malkin, and D. Dean. Self-healing key distribution with revocation. In *SP '02, IEEE*. IEEE Computer Society, 2002.

[134] D. Stinson. *Cryptography: Theory and Practice.* Chapman & Hall/CRC, second edition, 2002.

[135] S. G. Stubblebine and P. F. Syverson. Fair on-line auctions without special trusted parties. In *FC '99, LNCS*, volume 1648, pages 230–240, 1999.

[136] K. Suzuki, K. Kobayashi, and H. Morita. Efficient sealed-bid auction using hash chain. In *ICISC '00, LNCS*, volume 2015, pages 183–191, 2001.

[137] T. Tedrick. Fair exchange of secrets. In *Advances in Cryptology, CRYPTO '84, LNCS*, volume 196, pages 434–438, 1985.

[138] W. Trappe and L. C. Washington. *Introduction to Cryptography with Coding Theory.* Prentice Hall, second edition, 2005.

[139] J. Trevathan. Electronic auctions literature review. Unpublished manuscript, 2005. `http://www.cs.jcu.edu.au/~jarrod/lit.ps`.

[140] "USR56K". DC++ FAQ / Direct Connect FAQ - What is TTH (Tiger Tree Hashing) (#9677). Website from 1st August 2008. `http://www.dslreports.com/faq/9677`.

[141] H. C. A. van Tilborg. *Encyclopedia of Cryptography and Security.* Springer-Verlag New York, Inc., 2005.

[142] S. Vaudenay. One-time identification with low memory. In *EUROCODE '92, CISM*, pages 217–228, 1992.

[143] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *Journal of Finance*, 16(1):8–37, 1961.

[144] D. Wallner, E. Harder, and R. Agee. Requests for comments: 2627, key management for multicast: Issues and architectures. The Internet Engineering Task Force, June 1999. `ftp://ftp.rfc-editor.org/in-notes/rfc2627.txt`.

[145] X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In *Advances in Cryptology, CRYPTO '05, LNCS*, volume 3621, pages 17–36, 2005.

[146] X. Wang and H. Yu. How to break MD5 and other hash functions. In *Advances in Cryptology, EUROCRYPT '05, LNCS*, volume 3494, pages 19–35, 2005.

[147] Wikipedia. Random oracle — Wikipedia, the free encyclopedia. Website, 2006. `http://en.wikipedia.org/w/index.php?title=Random_oracle&oldid=78563553`.

[148] Wikipedia. Cryptographic hash function — Wikipedia, the free encyclopedia. Website, 2007. `http://en.wikipedia.org/w/index.php?title=Cryptographic_hash_function&oldid=100480738`.

[149] Wikipedia. International standard book number — Wikipedia, the free encyclopedia. Website, 2008. `http://en.wikipedia.org/w/index.php?title=International_Standard_Book_Number&oldid=229141685`.

[150] Wikipedia. Depth-first search — Wikipedia, the free encyclopedia. Website, 2009. `http://en.wikipedia.org/w/index.php?title=Depth-first_search&oldid=293976442`.

[151] Wikipedia. Hash function — Wikipedia, the free encyclopedia. Website, 2009. `http://en.wikipedia.org/w/index.php?title=Hash_function&oldid=291989208`.

[152] Wikipedia. Hash tree — Wikipedia, the free encyclopedia. Website, 2009. `http://en.wikipedia.org/w/index.php?title=Hash_tree&oldid=288106653`.

[153] M. V. Wilkes. *Time-Sharing Computer Systems*. Elsevier Science Ltd, 1968.

[154] C. K. Wong, M. Gouda, and S. S. Lam. Secure group communications using key graphs. *IEEE/ACM Transactions on Networking*, 8(1):16–30, 2000.

[155] X9F – Data & Information Security. Public Key Cryptography: The Elliptical Curve Digital Signature Algorithm (ECDSA). Technical report, Accredited Standards Committee, 2005.

[156] C. Yang and C. Li. Access control in a hierarchy using one-way hash functions. *Computers & Security*, 23(8):659–664, December 2004.

[157] Y. Zheng, T. Hardjono, and J. Seberry. New solutions to the problem of access control in a hierarchy. Technical report, Department of Computer Science, University of Wollongong, 1993. `http://www.sis.uncc.edu/~yzheng/publications/files/uow-cs-report-93-02.pdf`.

[158] S. Zhong. A practical key management scheme for access control in a user hierarchy. *Computers and Security*, 21(8):750–759, November 2002. `http://www.cse.buffalo.edu/~szhong/papers/hier.pdf`.

[159] L. Zhou and C. V. Ravishankar. A fault localized scheme for false report filtering in sensor networks. In *ICPS '05, IEEE*, pages 59–68, 2005.

[160] G. Zorn. Requests for comments: 2759, Microsoft PPP CHAP Extensions, Version 2. The Internet Engineering Task Force, January 2000. `ftp://ftp.rfc-editor.org/in-notes/rfc2759.txt`.

Appendix A

Algorithms

Algorithm 5:
Description: An algorithm to output all the paths in a DAG $\mathcal{G}$ with source set $S(\mathcal{G})$ and sink set $P(\mathcal{G})$.

```
FIND_PATHS(G, S(G), P(G))
(1)  foreach vertex in S(G)
(2)      Create a new list starting with this vertex.
(3)      while the previously considered vertex is not in P(G)
(4)          Append the first child of that vertex to the list.
(5)      print the list as a path of G
(6)      Set Pointer to point to the last but one vertex of the
         path.
(7)      while Pointer points to a vertex
(8)          if the vertex pointed to by Pointer has an older
             child than the child used in the previous list
(9)              Create a new list by copying all the vertices in
                 the previous list up to and including the one
                 pointed to by Pointer
(10)             while the previously considered vertex is not in
                 P(G)
(11)                 Append the first child of that vertex to the
                     list
(12)             print the list as a path of G
(13)             Set Pointer to point to the last but one vertex
                 of the path
(14)         else
(15)             Set Pointer to point to the previous vertex in
                 the previous path.
```

Algorithm 8:
Description: An algorithm to output all minimal verifiable sets which is very often more efficient than Algorithms 6 and 7
Input: A list of all the paths in $\mathcal{G}$, An empty set (PartialMVS)
Output: A list of all the minimal verifiable sets in $\mathcal{G}$

```
FIND_MINIMAL_VERIFIABLE_SETS(Paths, PartialMVS)
(1)  while we still have at least one path
      and all paths still contain vertices
(2)     if no vertex is in all paths and the shortest path has
        length at most one
(3)        foreach non-empty subset U of the shortest
           path P̂
(4)           Append the vertices in U to PartialMVS.
(5)           Make a temporary set of paths TempPaths to
              pass to the next recursion.
(6)           Copy all paths from Paths to TempPaths which
              do not contain vertices in U.
(7)           foreach vertex v in P̂ and not in U
(8)              Remove all references to v in TempPaths.
(9)           FIND_MINIMAL_VERIFIABLE_SETS(TempPaths,
              PartialMVS)
(10)          Remove the vertices in U from PartialMVS.
(11)       return
(12)    else if it is currently better to consider the vertex v*
        which appears in the most paths
(13)       Append vertex v* to PartialMVS.
(14)       Make a temporary set of paths TempPaths to pass
           to the next recursion.
(15)       foreach path P in Paths
(16)          if P does not contain v*
(17)             Copy P to TempPaths.
(18)       FIND_MINIMAL_VERIFIABLE_SETS(TempPaths,
           PartialMVS)
(19)       Remove vertex v* from PartialMVS.
(20)       Remove all references to vertex v* from all the lists
           in Paths.
(21)  if IS_SET_MINIMAL_AND_VERIFIABLE(PartialMVS)
(22)     print PartialMVS
```

Algorithm 9:
Description: An algorithm to construct the associated poset on the set of minimal verifiable sets MVS($\mathcal{G}$) ordered by computability.
Input: A list of the minimal verifiable sets of $\mathcal{G}$, A list of the paths in $\mathcal{G}$.
Output: An array containing the poset relations between the members of MVS($\mathcal{G}$).

```
FIND_ASSOCIATED_POSET(MVS(G), Paths(G))
(1)  Create an array to contain the poset relations Poset[MVS(G),MVS(G)].
(2)  Initialise all entries of the form Poset[i,i] to EQUAL, and all others to UNKNOWN.
(3)  foreach distinct pair of minimal verifiable sets i and j
(4)     foreach path P in Paths(G)
(5)        Set p_i to the vertex in P and i nearest the source of P.
(6)        Set p_j to the vertex in P and j nearest the source of P.
(7)        if p_i is nearer the source of P than p_j
(8)           if Poset[i,j] = UNKNOWN
(9)              Poset[i,j] = LOWER
(10)             Poset[j,i] = HIGHER
(11)          else if Poset[i,j] = HIGHER
(12)             Poset[i,j] = INCOMPARABLE
(13)             Poset[j,i] = INCOMPARABLE
(14)       else if p_j is nearer the source of P than p_i
(15)          if Poset[j,i] = UNKNOWN
(16)             Poset[j,i] = LOWER
(17)             Poset[i,j] = HIGHER
(18)          else if Poset[i,j] = HIGHER
(19)             Poset[j,i] = INCOMPARABLE
(20)             Poset[i,j] = INCOMPARABLE
(21) print Poset
```

In Algorithm 9 we use the following constants UNKNOWN, EQUAL, HIGHER, LOWER and INCOMPARABLE. If Poset$[i,j]$ is set to HIGHER then i is computable from j. If Poset$[i,j]$ is set to LOWER then j is computable from i. If Poset$[i,j]$ is set to INCOMPARABLE then i and j are not computable from each other.