2004

Indirect key derivation schemes for key management of access hierarchies

Cacic, Brian John

Indirect Key Derivation Schemes for Key Management of Access Hierarchies

Brian John Cacic

Department of Computer Science
Lakehead University
May 2004

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-96991-6
Our file Notre référence
ISBN: 0-612-96991-6

Preface

In this thesis, we study the problem of key management within an access hierarchy. Our contribution to the key management problem is an *indirect* key derivation approach we call the HMAC-method. It is called the HMAC-method, because it is based on hashed message authentication codes (HMACs) built from a fast, single, dedicated hash function (SHA-1). It is intended to provide an efficient indirect key management method for large access hierarchies resembling tree structures. We are able to achieve better tree traversals using a technique we created called *path addressing*. Our *path addressing* scheme allows us to efficiently calculate relationships between security classes, determine traversal paths, and improve the performance of indirect key derivation. We also present our *cached key* update scheme which is meant to improve the indirect key derivation schemes on tree hierarchies by delaying key updates when changes to the structure of the access hierarchy are necessary, but the re-calculation and re-assignment of keys would either be costly or inconvenient.

For access hierarchies represented as weakly/strongly connected directed acyclic graphs, we suggest modifications to our *path addressing* and key derivation scheme which could allow our HMAC-method to be applied to these types of hierarchies.

Along the way, we discuss various current key management methods and discuss certain pragmatic issues that can arise which affect the applicability and implementation of a key management method.

Acknowledgements

I would like to take this opportunity to thank those individuals who have helped me during my thesis work.

First, I would like to thank Dr. Wei for being my thesis advisor. His guidance during the research and process of this thesis has been invaluable, and I am happy to have had the opportunity to work with him and to learn from him.

Next, I would like to thank the faculty and staff of the Computer Science department at Lakehead University. The faculty has been wonderful to me and I have enjoyed the time I spent with them both as a student and graduate assistant.

Finally, I would like to thank those closest to me – my family. Through thick and thin my family has supported me with patience and understanding. I am truly blessed to have such wonderful support.

Thank you all very much.

Contents

Preface I

Acknowledgements II

List of Figures V

List of Tables VII

1 Introduction 1
1.1 Interest and Motivation 1
1.2 A Survey of Solutions to the Key Management Problem 3
1.3 Our Contributions to Key Management Methods 4
1.4 Thesis Outline 5

2 Relevant Topics in Cryptography 6
2.1 One-Way Functions 6
2.2 The RSA Cryptosystem 7
2.2.1 Security of the RSA Cryptosystem 8
2.3 Hash Functions 10
2.3.1 Classification of Hash Functions 11
2.3.2 Constructing Hash Functions 11
2.3.3 Security of Hash Functions 15
2.4 Message Authentication Codes 19
2.4.1 Block Cipher Based MAC 19
2.4.2 Dedicated Hash Functions 19
2.4.3 Security of Message Authentication Codes 20
2.5 Summary 20

3 Previous Research 22
3.1 Introduction 22
3.2 Direct versus Indirect Approaches 22
3.3 Direct Approaches to Key Management using RSA 23
3.3.1 The Akl-Taylor Method of Key Management 23
3.3.2 Other RSA Based Key Management Methods 26
3.4 Indirect Approaches to Key Management using One-Way Functions 33
3.4.1 Sandhu's Indirect Approach 34
3.4.2 Yang's Indirect Approach 35
3.5 Summary 37

4 Key Management Using HMACs 38
4.1 Introduction 38
4.2 The HMAC-method 39
4.2.1 Adding and Removing Security Classes from the Hierarchy 42
4.2.2 Security of the HMAC-method 53
4.3 Considerations and Limitations 54
4.4 Modified Path Addressing Scheme for DAG Hierarchies 57
4.5 Summary 60

5 Implementing Key Management Methods 62
5.1 Introduction 62
5.2 Direct Versus Indirect Key Management Methods 62
5.2.1 Structural Properties of the Hierarchy 63
5.2.2 Key Sizes and Key Updates 67
5.2.3 Role and Design of the Central Authority 69
5.3 Summary 70

6 Conclusions and Future Research 72

List of Figures

2.1 A general model of a hash function operating on a message M of size n. 12
2.2 The general construction of a hash function from a symmetric block cipher. 13

3.1 Public parameter assignment within the Akl-Taylor method. . 24
3.2 Graph illustrating Hwang-Yang leaf-group and leaf-classes. . . 31
3.3 A simple access hierarchy. 32
3.4 A simple hierarchy showing relative subordinate-principal positions. 37

4.1 A tree-structured access control hierarchy. 40
4.2 Adding a new security class to a leaf position in a simple tree hierarchy. 43
4.3 A simple tree hierarchy. 44
4.4 Adding a new security class between two classes with empty key caches. 44
4.5 Adding a new security class between two classes were the subordinate's key cache is empty. 45
4.6 Adding a new security class between two classes with full key caches. 46
4.7 Adding a new security class as the principal to a class that has a full key cache. 47
4.8 Security class (with leaf-classes) being removed. 50
4.9 Security class (with non-leaf subordinates) being removed. . . 51
4.10 Removing a subordinate from a principal with a full key cache. 52
4.11 Directed acyclic graph structured hierarchy. 58
4.12 A weakly connected DAG hierarchy. 60

5.1 A small and shallow tree shaped access hierarchy. 64
5.2 A broad and shallow tree shaped access hierarchy. 65
5.3 A hierarchy of a university student record database. 67
5.4 A hierarchy of a university student record database where an addition of a common security class allows us to apply *indirect* methods more efficiently. 67

List of Tables

1.1 A hierarchy of army officers and their cryptographic keychains. 2

2.1 Performance in Mbits/s of several hash functions and symmetric block cipher hash functions. 16

3.1 Prime number assignment and public parameter calculation for the Akl-Taylor hierarchy shown in (Figure 3.1). 25

3.2 Ray's key assignment scheme. 32

4.1 Summary of public parameters assigned to security classes belonging to (Figure 4.1). 41

4.2 Modified path addresses for security classes in a DAG access hierarchy. 59

5.1 Space complexity for direct key derivation schemes. n is the number of security classes in the hierarchy, y the number of primes the Hwang-Yang method requires (Sec. 3.3). 64

Chapter 1

Introduction

1.1 Interest and Motivation

Within multi-user computer systems, we would like to have a methodology that allows for the secure and trusted sharing of information amongst users of the system. In essence, we wish to control an individual's access to the information stored on the system. We require an access control mechanism.

Access control mechanisms have been a part of computer operating system development since the creation of MULTICS in 1965 [40]. The principle of *least privilege* is at the heart of all access control mechanisms. The principle stipulates that users of a computer system receive no more access than required to perform their responsibilities [16]. Implementation efforts have given rise to three prominent mechanisms: discretionary access control (DAC); mandatory access control (MAC); and role-based access control (RBAC) [16]. Shared amongst these mechanisms is the concept that users are divided and grouped into *classes*, which can be organized into a hierarchical structure according to the class's importance or level of trust. Thus, the priveleges a user may have on the computer system is determined according to the class he belongs to and the privileges associated with his class [16].

However, these access control mechanisms only attempt to limit the actions of users on stored information or computer resources. The access hierarchies by themselves do not provide ways in which the information may be kept in trust or secret [16]. We require a method which extends access control mechanisms to support the encryption, decryption, and keyed access of computer information and resources. But, this stipulation brings forth

Officer	Keychain
General	General, Major, Colonel, Captain, Lieutenant
Major	Major, Colonel, Captain, Lieutenant
Colonel	Colonel, Captain, Lieutenant
Captain	Captain, Lieutenant
Lieutenant	Lieutenant

Table 1.1: A hierarchy of army officers and their cryptographic keychains.

new challenges.

First, most mainstream cryptographic cipher systems are key based [37]. Whether they be symmetric or asymmetric, these modern cryptosystems hinge on the necessity of a shared key or accessible key for all interested participants. Within an access control mechanism, the problem lies not in the implementation of the cryptosystem or integration of the cryptosystem into the mechanism, but in the management of the keys amongst the participants within the classes of the access hierarchy [2], [3].

For example, let us examine a simple access hierarchy of army officers sharing a single computer workstation. In descending order of authority, we have the General, Colonel, Captain, and Lieutenant (Table 1.1).

Amongst this group of officers, all information stored on the workstation is kept secret through some cryptosystem. Each officer encrypts his information with his key. The hierarchy of officers and the actions they may perform on the workstation is controlled by an access control mechanism. Within this access hierarchy, a senior officer is given the authority to access information belonging to any subordinate officer. For example, the General may access everyone's information, but the Lieutenant may only access his own. As such, there will come a time when a senior officer must have access to a subordinate's information. Fortunately, the hierarchy defines such a relationship and the access control mechanism will allow it. But, information stored at each level in the hierarchy is encrypted with a different key. How can the senior officer access the information? As a simple solution, the army could order all officers to share their keys with each senior officer (Table 1.1). This simple method of key sharing is inefficient because the General is left to manage the keys of all his subordinates. If, for example, there are 10,000 men in the army (including the General), then the General must hold and manage 10,000 keys. Obviously, this simple sharing solution is too inefficient.

How can we effectively address this key management problem without having to resort to a shared common key amongst all security classes?

This question is known as the key management problem. We will search for methods that allow each class within a hierarchy its own security key, but will allow a class the ability to access or derive keys belonging a subordinate class over which they have the proper authority.

1.2 A Survey of Solutions to the Key Management Problem

Akl and Taylor proposed the first solution to the key management problem in an access control hierarchy (Sec. 3.3.1) [2]. The foundation of their solution was RSA's modular exponentiation arithmetic (Sec. 2.2). In their method, they would assign each security class a distinct prime number which in turn was used to calculate a public parameter. These values were used in the modular exponentiation equation to generate a key for each security class within the access hierarchy. The ability to access keys for other security classes and derive those keys was determined using the appropriate public parameters within the modular exponentiation equation. While many agreed that their method had merit, it was criticized for being inefficient [20], [34], [3], [26], [18].

The inefficiency arose from the computation and assignment of the public parameters. The initial algorithm generated numbers that were very large to store when the hierarchy itself contained many security classes [3]. Also, when a class was added or removed, all the parameters and keys had to be recomputed [18]. Later, Akl and MacKinnon proposed two new procedures which sought to find an efficient method of generating and assigning public parameters. The first method produced values that were smaller than the original algorithm, but proved to be susceptible to co-operative key recovery attacks [3]. The second method addressed the problem of security, but was not as efficient in generating smaller values as the first; it was slightly better than the original [3]. MacKinnon *et al.* found an optimal method for generating and assigning public parameters [26]. However, MacKinnon concluded the method was still inefficient for use within large access hierarchies [26]. Thus, the efficient calculation and assignment of public parameters was left unsolved [26]. Also, the inefficiencies of adding and removing security classes was never addressed; this too was left as an open problem [26] [18].

A second key management solution came when Sandhu proposed a method that used DES to generate and derive keys within the hierarchy [36]. The key assignment and derivation method was recursive; each subordinate received a key that was a digest of the direct principal's key and public parameter. Using his recursive approach, Sandhu avoided the costly storage inefficiencies of the Akl-Taylor method. However, Sandhu's proposed method of traversing the tree and generating keys with DES was criticized for being slow. Also, it was unable to deal with hierarchies represented as directed acyclic graphs. [18], [20].

The works of Akl-Taylor and Sandhu have since defined the two major approaches to the key management problem [34], [20]. The first approach is called the *direct* approach because access to other security classes and their keys can be determined using the appropriate set of public parameters [20]. Most approaches in this category build on the work of the Akl-Taylor method. These approaches use RSA's modular exponentiation arithmetic as their key generating and derivation function. Most proposals suggest new algorithms for assigning parameters to security classes, and using those parameters within the RSA modular exponentiation equation to generate and derive keys for other security classes. The second approach, commonly called the *indirect* approach, focuses on devising schemes in which hash functions and other one-way function constructions are used as the key derivation algorithm [20]. These approaches propose a recursive key derivation method that applies some public identifier and principal key to the inputs of one or several one-way functions.

1.3 Our Contributions to Key Management Methods

Our contribution to the key management problem is an *indirect* approach we call the HMAC-method. It is called the HMAC-method, because it is based on hashed message authentication codes (HMACs) built from a fast, single, dedicated hash function (SHA-1). It is intended to provide an efficient *indirect* key management method for large access hierarchies resembling tree structures. We can achieve better key traversals using a technique we created called *path addressing. Path addressing* allows us to determine security class relationships before generating keys and thus improve the responsiveness and efficiency of a single function *indirect* key management scheme.

We also present our *cached key* update scheme which is meant to provide flexibility to our *indirect* approach by delaying unnecessary key updates when changes to the structure of the access hierarchy are necessary, but the re-calculation and re-assignment of keys would either be costly or inconvenient. This can allow users within the security classes to continue working with their present keys, but delay a key update to a more convenient time.

For access hierarchies represented as weakly/strongly connected directed acyclic graphs, we suggest a modification to our *path addressing* and key derivation scheme which would allow our single function *indirect* approach to be applied to hierarchies resembling DAGs. The scheme we propose works, but it is not optimal for all DAG hierarchies.

Finally, we discuss some pragmatic issues surrounding the applicability and implementation of *direct* and *indirect* approaches.

1.4 Thesis Outline

In chapter two, we provide a review on some topics in cryptography relevant to understanding the design of key management methods. Chapter three will discuss previous research in the field of key management in an access hierarchy. We will show how key management solutions have evolved over time and discuss the two most prominent approaches for key assignment and derivation. In chapter four, we introduce our key management method called the HMAC-method. In chapter five, we discuss some pragmatic issues surrounding the applicability and implementation of *direct* and *indirect* approaches. Finally, we close with chapter six where we summarize our research and suggest a direction of future research.

Chapter 2

Relevant Topics in Cryptography

To facilitate our discussion of key management methods, we review some topics in cryptography that play an important role in the current research into key management and in the method presented in this thesis. We cannot replace a textbook or course on cryptography, so where a greater understanding of theory is necessary, we refer the reader to the cited literature.

We shall review the following topics: one-way functions, the RSA cryptosystem, hash functions, and message authentication codes.

2.1 One-Way Functions

One-way functions play a pivotal role in modern cryptography. They are important for creating public-key cryptosystems, message authentication codes, hash functions, and digital signature schemes [27].

One-way functions are functions which are easy to compute but difficult to inverse. That is, we look for problems that are computationally difficult to solve in a reasonable amount of time; for example, the problem of factoring a number N that is the composite of two large prime numbers [37]. As yet, there is no polynomial time algorithm that can perform the prime factorization of a number that is composed from sufficiently large prime factors [25]. To compose the number is easy, to factor it becomes difficult. Other computationally difficult problems include the discrete logarithm within a finite group, the graph colouring problem and the quadratic residue problem [25].

One-way functions are excellent candidates for constructing cryptographic schemes.

However, a question that still remains unanswered is whether one-way functions are mathematically correct. There has never been any mathematical proof that one-way functions truly exist or that they can be constructed [37]. To be clear, it is best to say that we *conjecture* the one-wayedness of a particular function or problem, knowing that in the future this may not be the case. For example, someone may find a prime factorization algorithm that runs in polynomial time; thus, the problem of factoring a number which is the composite of two primes would become easy to solve and would no longer be considered sufficiently one-way for use in a cryptosystem.

Let us examine how one-way functions can lead to higher constructions, such as the RSA cryptosystem and hash functions.

2.2 The RSA Cryptosystem

Since first implemented in the Akl-Taylor method of key management, RSA's modular arithmetic equation has appeared in many key management methods [2], [18], [10], [19], [20].

RSA is a public-key (asymmetric) cryptosystem. Unlike traditional symmetric cryptosystems that use one secret key, RSA uses two keys – one for encryption (a public key) and one for decryption (a secret key). It has been designed in such a way that it is considered computationally infeasible to deduce one key from the other. The algorithms that produce public-key cryptosystems are commonly referred to as *trapdoor* one-way functions. These functions are similar to the one-way functions discussed in the previous section; however, unlike true one-way functions, the inverse of the function is computationally feasible to deduce using a known *trapdoor*.

RSA is built on the problem of factorization (Sec. 2.1). The RSA encryption function, E, is the function that is easy to compute while the decryption function, D, is computationally infeasible to deduce unless a trapdoor is known. More formally, we may describe the RSA cryptosystem as follows [39]:

Let n be the product of two unique large primes p and q. We define the keyspace[1] $\mathcal{K}$ of the cryptosystem to be

[1] The set of all possible keys.

$$\mathcal{K} = (n,p,q,e,d) : n = pq, ed = 1 \mod (\phi(n)),$$

where $\phi(n) = (p-1)(q-1)$ is the Euler totient function of n[2]. For $\mathcal{K} = (n,p,q,e,d)$, we define the encryption function to be

$$E_{\mathcal{K}}(x) = x^e \mod n, \tag{2.1}$$

for some message x. The decryption function is

$$D_{\mathcal{K}}(y) = y^d \mod n, \tag{2.2}$$

for some ciphertext y. The values n and e are public, while the values p, q, and d are secret.

2.2.1 Security of the RSA Cryptosystem

As discussed, RSA is based on the conjecture that the factorization problem is one-way. Therefore, it should be computationally infeasible for anyone else but the proper recipient to decrypt the ciphertext. The trapdoor is knowing the factorization of $n = pq$, known only to the recipient. Knowledge of the trapdoor allows one to compute $\phi(n) = (p-1)(q-1)$ and use the Extended Euclidean Algorithm [24] to compute the decryption exponent d for use in (Eqn. 2.2).

Key management solutions that use the RSA modular exponentiation equation rely on the one-way trapdoor conjecture of factorization. However, there are known security attacks against RSA that one must be aware of when using it to devise key management schemes. Some attackers try to find ways of factoring n, while others will attempt to attack a flaw in an implementation of the cryptosystem, and others may attack the protocol in which RSA is being used [37], [25].

Because n and e are public, a rudimentary attack is to try to factor the value of n in order to recover the decryption key d [39]. Currently, the best known algorithm for factoring numbers is the Number Field Sieve (NFS) algorithm [25]. Recently, NFS was used in the RSA Factoring Challenge to factor an n value 576-bits (174 digits) long in seven months [25]. As a

[2] The totient function is defined as the number of positive integers which are relatively prime to n. Two numbers are relatively prime if their greatest common divisor is 1.

consequence, RSA Laboratories now recommends using an n value of 1024-bits or longer [25]. An attacker could also try to randomly guess values of d, but this brute force attack is far less efficient than using an algorithm like NFS [37].

There are three other attacks which target implementation flaws that can occur within RSA: low encryption exponent attack, low decryption exponent attack, and the common modulus attack.

Low encryption exponent attacks exploit a weakness in RSA that occurs when a low encryption exponent (e) value is used to encrypt $e(e+1)/2$ linearly independent messages, or e identical messages [37]. Similarly, if the decryption exponent, d, is up to one-quarter the size of n and $e < n$, then d could be recovered [37]. These attacks can be avoided by ensuring that all RSA parameters are properly selected [37].

The common modulus attack is critical to RSA-based key management solutions because many of them implement a common modulus as part of their key assignment and derivation schemes. Consequently, designers of RSA centric key management solutions are cautious of placing any critical values as the residue within the modular exponentiation equation. We can demonstrate the attack with a simple example.

Consider two people who share a common modulus n. The first person has the encryption and decryption exponent, e_1 and d_1 respectively. Similarly, the second person has his own encryption and decryption exponents, e_2 and d_2 respectively.

Let m be a plaintext message that both people will encrypt.

$$\begin{aligned} c_1 &= m^{e_1} \mod n \\ c_2 &= m^{e_2} \mod n \end{aligned} \tag{2.3}$$

Since an attacker knows the values of c_1, c_2, e_1, e_2 and n, he may also recover m.

Because e_1 and e_2 are relatively prime, you can use the extended Euclidean algorithm to find parameters r and s such that

$$re_1 + se_2 = 1$$

Using the extended Euclidean algorithm calculate ${c_1}^{-1}$. The message is recovered with

$$(c_1)^{-r} \times {c_2}^s = m \mod n.$$

Two other attacks also exploit a common modulus [37]. One uses probabilistic methods to factor n, while the other uses a deterministic algorithm to calculate a secret key without factoring n. We refer you to [37] for details.

2.3 Hash Functions

Hash functions are a special form of one-way functions. A hash function, H, takes a message[3], M, of arbitrary finite length and generates a unique fixed-length value, h, called a ***hash***[4]. That is,

$$h = H(M).$$

In addition to the definition above, hash functions possess an additional set of features [27]. They are as follows:

1. ***Compression:*** H maps an input M of arbitrary finite length to a an output h of fixed length.
2. ***Ease of computation:*** Given M, it is easy to compute h.
3. ***Preimage resistance:*** Given h, it is hard to compute M such that $H(M) = h$.
4. 2^{nd}***-preimage resistance:*** Given M, it is hard to find another message, M', such that $H(M) = H(M')$.
5. ***Collision resistance:*** It is hard to find two random messages, $M \neq M'$, such that $H(M) = H(M')$.

The last two constraints set hash functions apart from one another. These constraints define the *collision resistance* properties of hash functions. A collision occurs when a hash function produces the same message digest for two different messages. Given that a hash function takes an arbitrary finite length message of m-bits and produces a fixed n-bit message digest, where

[3] Also known as a *pre-image*

[4] Also known as a *message digest* or simply *digest*

$n < m$, the instance of collisions is unavoidable. Hash functions that meet only the fourth constraint are known as *weakly* collision resistant. This means that if we know M, it should be difficult for us to use that knowledge to find another message M' which produces a collision with $M - h(M) = h(M')$. Hash functions that adhere to the fifth constraint are known as *strongly* collision-resistant. That is, it is computationally difficult to find two random messages, not equal to one another, that will produce the same message digest. A hash function that is *strongly* collision-resistant implies it is also *weakly* collision-resistant, but the reverse is not true [28].

2.3.1 Classification of Hash Functions

Hash functions can be subdivided into two families:

- modification detection codes (MDCs), and
- message authentication codes (MACs).

MDCs are typically used to provide a representative image (digest) of some given input. MDCs are commonly used in digital signature schemes, i.e., a digest is created for a message and the digest is encrypted with the owner's secret-key [37]. MDCs may be further classified as *one-way* functions or *collision resistant* functions. The two categories differ depending on the features they implement. *One-way* functions only provide features (1-4)(Sec. 2.3), while *collision resistant* hash functions implement features (1-5) (Sec. 2.3). Most of the strong hash functions used today are collision resistant hash functions and are designed to implement all five features listed in (Sec. 2.3).

The second family of hash functions, MACs, are known as keyed hash functions. MACs can provide message authentication without reliance on a secondary cryptographic construction. When a message digest is created with a MAC, only parties holding the proper secret key may re-calculate and verify the digest [27]. We continue our discussion on MACs in (Sec. 2.4).

2.3.2 Constructing Hash Functions

It is not easy to design collision resistant hash functions [28], [12], [37], [27]. First, the function should work for inputs of arbitrary finite length and produce a fixed-length digest. Second, the function should be one-way. Finally, the function should implement the collision resistance properties. Not surprisingly, there are only a handful of collision-resistant hash functions that have managed to withstand scrutiny [37], [27].

In general, a hash function is composed of three stages: a pre-processing stage, an iterated processing stage, and a transformation stage (Figure 2.1).

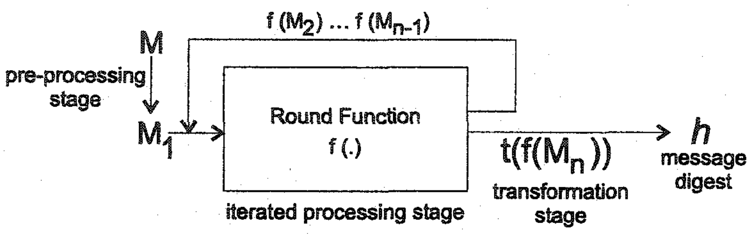

Figure 2.1: A general model of a hash function operating on a message M of size n.

A hash function works on message blocks of size m-bits. In the preprocessing stage, the message is padded to a multiple of m-bits (if necessary) and concatenated to a message block indicating the unpadded length of the message. This padding technique is known as MD strengthening [27]. The compression function is iterated on the formatted message, where the input to each round of the compression function is the intermediate message digest (H_{i-1}) and the next message block, M_i. The hash of the last message block (H_i), undergoes a final transformation $t(H_i)$ to become the message digest.

We may represent this process mathematically as

$$H_0 = IV; h_i = f(M_i, h_{i-1}); h(x) = t(x),$$

where IV represents some predefined initialization value used to start the hashing process.

The underlying compression function, f, is not limited to a particular mathematical construction. In fact, many different constructions have been proposed. Everything from cellular automata [12], [11], to algebraic matrices [17], to modifications of the Merkle-Hellman knapsack algorithm [11]. However, many of these hash functions were found to be insecure [11], [37], [9]. The three most popular constructions used to build hash functions today are: block ciphers, modular arithmetic, and dedicated hash functions.

Block Ciphers

Block ciphers, like DES, can be used to produce hash functions [27]. By using a strong block cipher the supposition is that the hash function should be as secure as the underlying block cipher, but that is not always the case [30], [8], [41], [33], [38].

The compression function is built from the underlying block cipher encryption function, E (Figure 2.2)). If given a message M of size n and a block cipher whose block size is m, we may produce a message digest as follows [27]:

- Preprocess M. Divide M into blocks the size of m.
- Start the first round of hashing with some random initial value, IV, and the first message block M_1.
- The remaining intermediate digests are generated by: $H_i = E_I(K) \oplus T$, where the values of I, K, and T can be: M_i, H_{i-1}, $M_i \oplus H_{i-1}$, or some constant C.

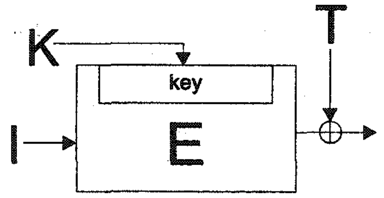

Figure 2.2: The general construction of a hash function from a symmetric block cipher.

Because the values of I, K, and T can be taken from the set $M_i, H_{i-1}, M_i \oplus H_{i-1}, C$, this provides the possibility of $4^3 = 64$ different constructions for $f()$ [37]. Bart Preneel studied them all and found that fifteen of them are trivially weak, thirty-seven are insecure, and the remaining twelve have varying levels of security [37]. Of the twelve, four are secure against all attacks and the remaining eight are secure against everything but a fixed-point attack (Sec. 2.3.3) [37].

Block cipher hash functions are generally categorized according to the size of the message digest they produce and the rate at which the hash is calculated. Hash functions which produce message digests equal to the block size, or twice the size of the underlying block cipher are called *single-length* and *double-length* block cipher hash functions, respectively [27].

The rate of a hash function (r) measures the number of block encryptions (s) required to process each successive n-bit message block. The rate is calculated as: $r = 1/s$ [27]. Thus, a hash function with a rate of 1/2 is twice as slow to produce a message digest as a hash function of rate 1.

Single-block or double-block hash functions with a rate of 1 are the most desirable because of the speed at which they can generate a message digest. In practice, it is very difficult to develop double-block hash functions with rate 1; most have a rate < 1 [27] [31]. Also, research has shown that most single-block hash functions with rate 1 created to date are not sufficiently strong enough to withstand a collision (birthday) attack (Sec. 2.3.3) [31]. Recent research by Knudsen, Lai, and Preneel also suggests that certain double-block hash functions built from 64-bit block ciphers are also not sufficiently strong enough to protect against a collision attack [23] [22].

In recent years, new block ciphers with block and key sizes greater than 128-bits have been introduced [38]. However, some research suggests that not all of these new block ciphers are secure enough to use as hash functions [38], [29]. Also, almost all have a hash rate < 1, making them less efficient than their dedicated hash function counterparts, and in some instances, an increase in key size can cause a decrease in performance [38].

Modular Arithmetic

The modular exponentiation arithmetic of the RSA cryptosystem can be used as a hash function [37]. If the message to be hashed, M, is used in the place of the residue; n is the modulo (the product of two primes p, q), and e is relatively prime to $(p-1)(q-1)$, then the hash function becomes[5]

$$H(M) = M^e \mod n.$$

Compromising the hash is as hard as factoring n. Compared to other symmetric block cipher constructions and dedicated hash functions, RSA modular arithmetic as a hash function is less efficient. Therefore, it is not recommended to use RSA or RSA-like constructions as hash functions [37], [25].

[5] $H(M)$ is the hash value.

Dedicated Hash Functions

Dedicated hash functions are specifically designed only to create message digests. These functions are designed to compress and permute the input message via a series of rounds. Rounds typically involve the bits of a message block being XOR'd with one or more of the following values: the values of a previous block, the values from an S-box or constant, or the values resulting from a previous round [27]. The construction of each dedicated hash function is unique; however, designing good dedicated hash functions is very hard [37].

The following is the number of reliable dedicated hash functions available today[5]:

- MD5 (128-bit digest);
- SHA-1, SHA-256, SHA-384, SHA-512 (Secure Hash Standard);
- RIPEMD-128, RIPEMD-160;
- Tiger (optimized for 64-bit processors);
- WHIRLPOOL; and
- Subhash.

Dedicated hash functions are designed for speed and efficiency, and their run-time performance in software is better than most block cipher and modular arithmetic hash functions [37], [25]. Table 2.1 from [32] highlights the performance of dedicated hash functions with respect to one another and with respect to symmetric block ciphers.

2.3.3 Security of Hash Functions

Attacks on hash functions fall into one of the following categories [4]:

- attacks independent of the algorithm, and
- attacks dependent on the algorithm.

We shall quickly review both categories.

Algorithm	Performance (Mbits/s)
Hash Functions	
MD5	136.2
RIPEMD-128	77.6
RIPEMD-160	45.3
SHA-1	54.9
TIGER	34.9
Symmetric Block Ciphers	
DES	16.9
IDEA	9.75
CAST	16.2
Blowfish	26.5

Table 2.1: Performance in Mbits/s of several hash functions and symmetric block cipher hash functions.

Attacks Independent of the Algorithm

These attacks depend on the length of the message digest (m) and/or, in the case of MACs (Sec. 2.4), the key length (l). Such attacks are the collision (birthday) attack, the exhaustive key search attack, the preimage (random) attack, and the pseudo attack [4].

The collision (birthday) attack examines the probability of producing two equivalent message digests from the same hash function [39]. For example, given a hash function, $h : X \rightarrow Y, n = |X| > m = |2Y|$ we can see that the probability of collision is at least $\frac{1}{2}$. To find a collision, we choose k random values of $x \epsilon X$, compute $h(x) = z, z \epsilon Z$, and see if a collision results. The lower bound on the probability of collision is dependent on k and n, but not m [39]. Stinson shows that if the estimate for this lower bound is taken to be 50% then $k \approx 1.17\sqrt{n}$. That is, if we hash at least $1.17\sqrt(n)$ random x's, the probability of a collision occurring will be $\approx$ 50%. Stinson provides a complete treatment of the attack, so we refer the reader to [39] for more detail.

In general, MDCs which produce larger message digests are less susceptible to the collision attack [27]. Much of the literature agrees that hash functions producing digests larger than 128-bits are computationally secure for the time being [27], [37]. The general principle is that given a hash function with a digest length of m-bits, the time complexity of a successful collision attack is given as $O(2^{m/2})$ [27].

The exhaustive key search attack applies only to MACs (Sec. 2.4). If an attacker can obtain a (*message,digest*) pair, then it is possible for him to perform an exhaustive key search to find a key that transforms the *message* into the corresponding *digest* [4]. Since collisions in hash functions are unavoidable, it is theoretically possible to find more than one valid key. Generally, if the key l-bits in size, then the probability of an attacker finding the correct key is 2^{-l}. However, in some instances if the attacker has access to a sufficiently large number of (*message,digest*) pairs created with the same key, then the search space for keys could be reduced [4].

In the preimage (random) attack, an attacker chooses a message at random and hopes that the digest he produces is equal to the authentic one [4]. The probability of success for this attack is 2^{-m}, where m is the length of the message digest. To thwart this attack, it is recommended to use hash functions that produce digests longer than 64-bits [39], [27], [37].

Finally, the pseudo attack tries to find a pseudo key that will produce the same MAC digest as an authentic (*key,message*) pair [4]. The goal for the attacker is to try to find a key that could possibly identify him as a legitimate holder of the authentic secret key. As noted in [4], this does not mean that the pseudo key will produce valid digests for other (*key,message*) pairs. Rather, the attacker is seeking evidence of authority by using a fraudulent key instead of the authentic one. The probability of success for this attack is similar to the exhaustive key search attack [4].

Attacks Dependent on the Algorithm

These attacks target specific weaknesses in specific hash function constructions. There are four categories of attack: meet-in-the-middle, correcting block, fixed-point, and differential cryptanalysis [4].

A meet-in-the-middle attack is a variation of the birthday attack [4]. It can be applied to hash functions whose compression functions are invertible on intermediate digests H_i or message block M_i. The goal is to produce a fraudulent message that will produce the original message digest [4]. To start, the attacker chooses some initial value, IV, and generates the first intermediate digest, $H_1 = f(IV, M_1)$. Working backward, the attacker inverts the compression function on the authentic digest to recover the last authentic message block M_n and the last intermediate digest H_{n-1}. The attack continues until it meets in the middle. Then, the chaining variables are compared. The probability that the chaining variables are the same is

$$P \approx 1 - e^{-x_1 x_2/2^m},$$

where m is the length of the message digest [4] and x_1, x_2 are the number of forward and backward samples the attacker undertakes. If chaining variables match, then the concatenation of the all the fraudulent and recovered message blocks form a collision on M. This attack can be thwarted by carefully choosing compression functions that are not easily inverted, or by using a transformation function that is difficult to invert [4].

The correcting block attack attempts to change one or more message blocks, without affecting the message digest [4]. Den Boer and Bosselaers successfully used this attack against one round of MD5 [13]. A method to help thwart this attack is to ensure that all bits of the message digest are dependent on all bits of the message blocks [13]. Dobbertin [14] demonstrated how changing the recommended initialization variable (IV) to the compression function of the dedicated hash function MD5 could result in unwanted collisions. Dobbertin's attack creates suspiciouns around MD5 [37] and since then, designers are cautious of modifying the parameters of dedicated hash functions [37].

In the fixed-point attack, an attacker looks for a particular chaining variable H_{i-1}, such that $f(H_{i-1}, M_i) = H_{i-1}$. That is, the message block and compression function has no effect on the chaining variable [4] — a fixed-point. A fixed-point allows an attacker to substitute a fraudulent message block at each fixed-point location. One can overcome this attack by adding redundancy to the hash function in a similar manner one would use to thwart the correcting block attack [4].

Finally, differential cryptanalysis was first proposed by Biham and Shamir [8], who demonstrated how the technique could be used to compromise a reduced version of DES, a full 16 round version of DES, and the hash functions SNEFRU and N-HASH. Using differential cryptanalysis, an attacker monitors the input and output of a hash function on a chosen set of (*message,digest*) pairs. By studying how a specific difference in the message affects the digest, the attacker can gain information about the underlying block cipher or compression function [8]. In hash functions, one can use differential cryptanalysis to deduce collisions. Berson showed how the technique could be applied to produce a collision on a single round of MD5 [7]. Unlike other attacks, one cannot easily thwart differential cryptanalysis. A hash function's ability to thwart such an attack is dependent on the underlying strength of the hashing process and compression function [4].

2.4 Message Authentication Codes

Message authentication codes (MACs) are keyed collision resistant hash functions. Their properties are similar to those of regular hash functions, but the message digest is also dependent on a secret key [6]. All parties wishing to verify the message digest must possess the secret key [27] [6].

As noted in [37], MACs are typically used to verify file transfers or to provide message authenticity without the need of an additional cryptosystem. For example, Alice and Bob share a secret key K. Alice calculates a MAC digest using K on some file. Alice transfers the digest and the file to Bob. Assuming that no one else has obtained K, only Bob may verify the digest. And, since he knows the key is only shared between him and Alice, Bob is assured that if the two digests match, then Alice is likely the source of the file. If the digests do not match, Bob will suspect tampering.

Because MAC codes are simply keyed hash functions, their constructions are rather trivial [27]. We simply introduce a key at some step in the process. There are two common types of message authentication codes: dedicated hash function based and block-cipher based [37].

2.4.1 Block Cipher Based MAC

We have discussed the properties of block ciphers as hash functions in (Sec. 2.3.2). To extend these hash function constructions to MACs, we use the block cipher in cipher block chaining mode (CBC) and use the key to encrypt each message block as it goes through a round. The last encrypted message block becomes the message digest [27], [39].

2.4.2 Dedicated Hash Functions

One may use dedicated hash functions, like MD5 and SHA-1, as a MAC. The resulting construction is often called a Hashed Message Authentication Code (HMAC) [6]. To use a dedicated hash function as a MAC, one must simply concatenate the key to the message in some fashion and calculate the message digest. For example,

$$H = (K, M).$$

Preneel notes that this basic construction may not be secure if an attacker can obtain enough digests calculated with the same key [37] [33]. One can employ certain techniques to strengthen HMACs, and we shall discuss those issues in the next section.

2.4.3 Security of Message Authentication Codes

The security of a message authentication code hinges upon the security of the underlying hash function and the security of the key.

As alluded to in the previous section, MACs built from dedicated hash functions depend on both the security of the underlying hash function and the concatenation of the key with the message.

Bart Preneel studied HMAC constructions and found that certain constructions such as $H(M,K), H(K,M,K)$, and $H(K_1, M, K_2)$ (where K_1 and K_2 are different) could be compromised if an attacker obtained many message digests calculated with the same key [33]. In response, he recommends the following constructions [37] [33]:

- $H(K_1, H(K_2, M))$
- $H(K, H(K, M))$
- $H(K, p, M, K)$, where p pads K to a full message block.

For dedicated hash functions, Preneel has created MDx-MAC [33], a strengthening technique that can harden dedicated hash functions against key recovery attacks with only a slight decrease in digest throughput [33].

2.5 Summary

The key management methods based on the *direct* approach of key assignment and derivation functions use the RSA modular exponentiation algorithm. As discussed, this algorithm represents a trap-door one-way function whose strength is dependent on the factorization of a well chosen modulus.

In lieu of recent factorization results, RSA recommends using moduli larger than 576-bits. When parameters for the function are chosen, the user should take care to ensure secret information is not compromised with a common modulus attack, low exponent attack, or low decryption exponent attack.

Key management methods based on the *indirect* approach of key assignment and derivation use other one-way functions, but hash functions are common. A hash function's security relies heavily on the underlying preprocessing stage and compression function. Consequently, developing hash functions is still a challenge. Some hash functions are more susceptible than others to certain attacks, so careful consideration must be taken when choosing an appropriate hash function. In general, hash functions with larger digests are more secure. Given the inefficiency of modular arithmetic, and the performance of block ciphers, dedicated hash functions are more favoured when a reliable, secure, and efficient hash function is required. However, the number of dedicated hash functions available for use today is limited.

The HMAC-method we propose in this thesis uses message authentication codes built from dedicated hash functions (HMACs). Message authentication codes concatenate a key along with the pre-image to produce a digest. Only users holding the appropriate key may compute and verify the digest. Certain attacks exist on message digests, but these occur under special circumstances. If such circumstances arise, certain constructions can be used to thwart the attack, or one may use Preneel's MDx-MAC method to strengthen the HMAC's security.

Now that we have reviewed the relevant topics in cryptography, we may examine previous research into the key management problem. In chapter three, we will see how RSA and hash functions have been applied to the key management problem.

Chapter 3

Previous Research

3.1 Introduction

This chapter presents the previous research in the field of key management for access hierarchies. We will cover both the *direct* and *indirect* approaches to key management that have been proposed and shown to be secure.

3.2 Direct versus Indirect Approaches

All *direct* approaches to key management proposed thus far build on the Akl-Taylor method [3]. That is, using RSA's modular exponentiation arithmetic and a set of public parameters, a user in a principal security class can directly calculate the key belonging to a subordinate security class, provided the principal has the authority to do so. With limitations in the Akl-Taylor method, new proposals suggest new algorithms for assigning parameters to security classes, and using those parameters within the modular exponentiation equation to generate and derive keys.

The *indirect* approaches to key management focus on devising schemes in which other one-way functions are used as the key derivation algorithm. Most methods propose a recursive traversal of the hierarchy and the key derivation method applies some public identifier and principal key to the inputs of one or several one-way functions.

3.3 Direct Approaches to Key Management using RSA

Akl and Taylor first proposed the problem of key management in an access hierarchy [2]. The goal then was to devise a method in which each security class within an access hierarchy maintains a minimum number of keys, but can derive keys of subordinate classes through a set of public parameters and known cryptographic algorithms. Their method was based on RSA's modular exponentiation encryption function (Sec. 2.2). We shall discuss how their model performed, its advantages and disadvantages and how others have modified their work to produce new RSA-based key management methods.

3.3.1 The Akl-Taylor Method of Key Management

The Akl-Taylor method of key management [3] starts with a few assumptions:

- The access hierarchy is controlled by some trusted central authority (CA) responsible for marshaling and monitoring all actions performed within the hierarchy and amongst security classes. It is assumed that the CA is a secure environment that provides no viable communication channel vulnerable to attack.

- All users within the access hierarchy are divided into security classes, $SC = SC_1, \ldots, SC_n$, which are partially ordered by the binary relation $\leq$. The resulting relations, $SC_j \leq SC_i$, means that users belonging to security class SC_i have access to information stored at the subordinate security class SC_j; however, the reverse relation does not hold. That is, subordinates are not allowed access to information stored at the principal (SC_i).

The key assignment and transformation scheme is devised as follows. A random secret key, K_0 is generated and assigned to the CA. The CA generates a value, N, where $N = pq$ and p, q are prime. This value will be used as a common modulus in the key transformation function. Next, each security class in the hierarchy is assigned a distinct prime number and unique public parameter (PB) so that $PB_i \mid PB_j$ if and only if $SC_j \leq SC_i$. The PBs are computed by the algorithm

$$PB_j = \prod_{SC_i \not\leq SC_j} e_i,$$

where e_i is the distinct prime number assigned to SC_i. An example of the prime number assignment and PB calculation is shown in (Figure 3.1) and listed in (Table 3.1).

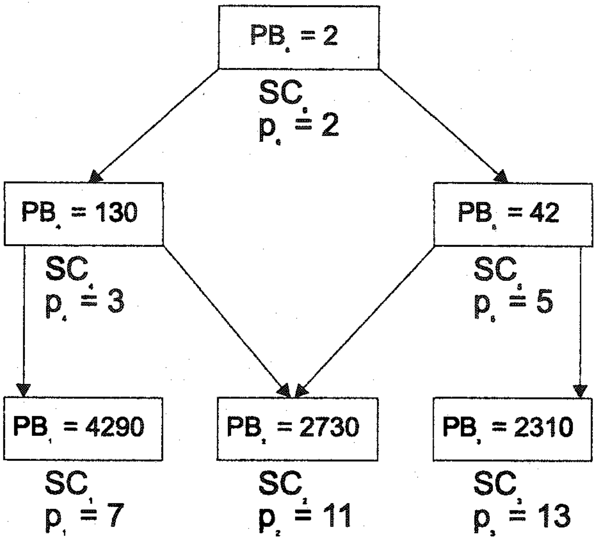

Figure 3.1: Public parameter assignment within the Akl-Taylor method.

Keys are then assigned to security classes using a key generation and transformation function based on RSA's modular exponentiation (Sec. 2.2):

$$K_j = K_0^{PB_j} = \left[K_0{}^{PB_j}\right]^{PB_j/PB_i} = K_i^{PB_i/PB_j} \mod N. \tag{3.1}$$

Thus, for a user in security class SC_i to derive the key of a subordinate class, SC_j, SC_i must use (Eqn. 3.1), his key (K_i), and obtain from the CA the appropriate public parameters (PB_i, PB_j, N). If $PB_i | PB_j$, then SC_i is principal to SC_j and (Eqn. 3.1) will successfully derive K_j.

The security of the Akl-Taylor method is sound [2], [3], [26], [18]. Since

$$\gcd(PB_i, PB_j) \neq 1,$$

Security Class SC_i	Prime e_i	$PublicPB_i$
SC_1	7	$4230 = e_2 \times e_3 \times e_4 \times e_5 \times e_6$
SC_2	11	$2730 = e_1 \times e_3 \times e_4 \times e_5 \times e_6$
SC_3	13	$2310 = e_1 \times e_2 \times e_4 \times e_5 \times e_6$
SC_4	3	$130 = e_3 \times e_5 \times e_6$
SC_5	5	$42 = e_3 \times e_4 \times e_6$
SC_6 (CA)	2	$2 = e_6$

Table 3.1: Prime number assignment and public parameter calculation for the Akl-Taylor hierarchy shown in (Figure 3.1).

where gcd represents the greatest common divisor, the common modulus attack (Sec. 2.2.1) cannot be used to recover the master key, K_0, from (Eqn. 3.1). Thus, the security of the system equal to that of RSA.

However, the Akl-Taylor method was not without its faults. First, the method does not allow for the efficient addition of security classes to the hierarchy [1], [3], [18], [20]. For each security class added into the hierarchy, one must undertake a whole new round of public parameter calculation and key assignment [1], [3], [18], [20] [34]. Also, the method used to assign and calculate PBs results in values that quickly grow as the number of security classes in the hierarchy increases [1]. MacKinnon and Akl show that for hierarchies that contain as few as twenty security classes, the largest public parameter generated and stored will be 278970415063349480483707695 [1]. Thus, they concluded their assignment of PBs to security classes would be inefficient for larger hierarchies [1], [20]. In the following years, MacKinnon and Akl devised new methods for the efficient generation and assignment of public parameters. In the following years, they discovered two new methods [1]. One method was susceptible to a collaborative attack, but the other showed promising results. However, this second algorithm was still not optimal [1]. Soon after, MacKinnon *et al.* derived, what they called, the canonical assignment method, which somewhat reduced the size public parameters, but not enough when the hierarchy contained many classes [1], [18]. In addtion, it was difficult to find an optimal canonical algorithm that could make an optimal assignment for all hierarchies [26], [20]. Thus, the problem of generating and assigning PBs was left open. So too was the problem of adding security classes in an efficient fashion.

In subsequent years, new *direct* key management methods built upon the work of the Akl-Taylor method. Most of these new methods propose new procedures of assigning and generating public parameters, which try to improve the efficiency of public parameter storage, and the addition and removal of security classes to the underlying hierarchy. For the purpose of brevity, we shall only highlight those changes to the Akl-Taylor model that result in a new and secure key management system and refer the reader to the literature for further details.

3.3.2 Other RSA Based Key Management Methods

Harn and Lin proposed the first modification to the Akl-Taylor method [18]. Their improvement came in the way the PBs were calculated and security keys were generated. As shown in (Figure 3.1), the Akl-Taylor method assigns prime numbers and calculates PBs in a top-down fashion. The Harn-Lin method proposed the opposite – they calculated the PBs and derived the keys from the bottom-up. This simple change to the system allowed for a method to add classes, and removed the necessity for a secret key controlled by the CA.

In the Harn-Lin method, security classes may be removed without any overhead. However, additions were still not efficiently accommodated [18]. If a security class was added to any position within the hierarchy, then all security classes principal to the added class would require new prime numbers, new public parameters, and new keys. Harn-Lin commented that their method was more storage efficient for public parameters than the Akl-Taylor method [18]. However, as commented by Hwang, the improvement was not significant [20]. As the number of security classes increases, more prime numbers are needed and stored and the public parameters continue to grow in size. Overall, the achievement of the Harn-Lin scheme is its ability to add security classes without affecting all portions of the hierarchy and the removal of a secret key K_0 held by the CA.

Within their method Harn and Lin made slight modifications to the key generation function. The CA still generates a common modulus N as before, but the CA no longer generates a random secret key K_0. Rather, the CA chooses K_0 as a value in the range $[2 \ldots N-1]$ and makes it public. Each security class SC_i receives a prime number e_i whose multiplicative inverse d_i is calculated by:

$$d \equiv e^{-1} \mod \phi(N),$$

where $\phi(N)$ represents the Euler totient of N (Sec. 2.2). To assign a key to SC_i required a public parameter PB_i such that:

$$PB_i = \prod_{SC_j \leq SC_i} e_j.$$

The key was then generated using

$$K_i = K_0^{\prod_{SC_j \leq SC_i} d_j \mod \phi(N)} \mod N.$$

For a principal security class SC_i to derive the key of its subordinate, SC_j, the key transformation function becomes

$$K_j = K_i^{PB_i/PB_j} \mod N.$$

With these modifications to the Akl-Taylor method, Harn and Lin still thwart the common modulus attack because the recoverable value K_0 is already public [18]. Thus, their system is as secure as RSA.

Following Harn-Lin, Chick and Tavares proposed their variation of the Akl-Taylor method [10]. Their system did not require security classes and did not assign a master key to the CA. They modified the access hierarchy to define the binary relation $\leq$ on the set of services, $S_1, \ldots, S_n$, provided by a system. The resulting relationship, $S_j \leq S_i$, means that if a user is granted privileges to the services of S_i, he is also conferred the services of S_j. Further, each service is assigned an access key, SK_i, also subject to the binary relation $\leq$. That is, $SK_j \leq SK_i$, indicates that service key SK_j can be derived from service key SK_i.

The *master key* (MK) is re-defined as a compact representation of a subset of services. For any master key, MK_i, the relation $SK_j \leq MK_i$ indicates that the master key can derive the service keys for the specified subset of services. The master keys are managed, but not assigned to the CA. To use the computer system, a user must be assigned an appropriate master key that he can use to derive service keys for his required services.

Assignment of prime numbers (e) and public parameters (PB) is similar to the Akl-Taylor model, and we refer the reader to [10] for specific details. The noticeable additions are the public parameter T defined as

$$T = \prod_{n=1}^{N} e_n,$$

where N is the number of primes generated by the CA, and the public parameter v_j, defined for each MK_j as

$$v_j = \prod_{SK_i \leq MK_j} e_j.$$

A master key is defined as

$$MK_j = K_0^{T/v_j}.$$

To calculate a service key, SK_i, from its master key, MK_j, the generation equation as follows:

$$\begin{aligned} SK_i &= (K_0)^{T/PB_i} \\ &= ((K_0)^{T/v_j})^{v_j/PB_i} \\ &= (MK_j)^{v_j/PB_i} \text{ iff } SK_i \leq MK_j \end{aligned}$$

Thus, any user who is assigned a master key MK_j can use this master key and the appropriate public parameters to derive the service key, SK_i, for all services $SK_i \leq MK_j$.

The Chick-Tavares key management method is unique amongst all RSA based methods, because it is the only one that does not use a hierarchy of security classes. Rather, the users' access to objects and information is limited to the services provided by their master key [10]. However, if a new service is added to the hierarchy, or someone required access to a new service, the master key belonging to the service subset would have to be recalculated. Also note that the size of the service keys can be rather large if a user is holding access to many services. This could be inefficient if one uses only some services within the subset very sparingly. The Chick-Tavares method is as secure as the Akl-Taylor method. Users cannot collaborate to recover keys – K_0 is public. Thus, the strength of the overall system is equal to that of RSA [10].

In more recent years, the research of Harn-Lin has inspired others to find different ways in which to assign public parameters to the hierarchy of users, and the ways in which to use the RSA modular exponentiation function for key generation and transformation. What follows is a discussion of this research, and the proposed methods.

Hwang and Yang [20] proposed a variation on the Akl-Taylor method that attempts to reduce the magnitude of the prime numbers and thus, the amount of storage required for public parameters. They did this by adopting Harn and Lin's method of bottom-up assignment [18] and by using *composed prime sets* to reduce the number of primes required and the magnitude of the public parameters [20].

Their approach views the access hierarchy as a tree where the most principal security class is the root and the most subordinate are the leaves. Key generation and assignment proceeds as follows [20]:

- The CA chooses two large prime numbers p and q and computes the public parameter $N = pq$, where p and q are secret and N is public.
- The CA chooses a public parameter g between $[2 \ldots N-1]$ such that g and N are relatively prime.
- The CA selects a set of primes e_i and calculates the multiplicative inverse d_i for each prime.
- Each leaf class is assigned a composed prime set, z_i, which is not a subset of z_l if $SC_i \not\leq SC_l$. SC_i and SC_l are leaf security classes. The CA stores the composed prime set for each leaf-class.
- The CA assigns to each non-leaf security class, SC_i, a distinct prime number e_i. SC_i is assigned a composed prime set x_i which is a union of its e_i and the composed prime set of all its direct subordinate classes.
- The CA calculates a public parameter PB_i and secret key K_i for each leaf security class SC_i:

$$PB_i = \prod e_l,$$

$$K_i = g^{f(SC_i)\prod d_l \mod \phi(N)} \mod N,$$

where $e_l \in x_i$ and $f(.)$ is a one-way function. To thwart the common modulus attack, Hwang and Yang discard PB_i and K_i if $PB_i K_i$ mod $\phi(N)$.

- The CA calculates a public parameter PB_j and secret keys K_j for all non-leaf security classes SC_j.

$$PB_j = \prod e_l$$

$$K_j = g^{\prod d_l \mod \phi(N)} \mod N,$$

where $e_l \in x_j$ and d_l is the multiplicative inverse of each e_l.

Finally, if a security class wishes to derive the secret key, K_i of a subordinate class, the formulas are:

$$K_i = \begin{cases} (K_j^{PB_j/PB_i})f(SC_i) \mod N, & \text{if } SC_i \text{ is a leaf class,} \\ K_j^{PB_j/PB_i} \mod N, & \text{if } SC_i \text{ is a non-leaf class.} \end{cases}$$

The size and storage requirements for prime numbers and public parameters is smaller than many of the RSA-based methods reviewed thus far. For example, an access hierarchy with one thousand security classes under the Akl-Taylor method would require one thousand distinct prime numbers and would result in very large public parameters [3], [20]. For certain structures of hierarchies, Hwang and Yang's scheme can greatly reduce the number of unique prime numbers; thus, the size of the public parameters can be kept much smaller. Consequently, less storage is needed for the public parameters, but there is the added requirement of having to store the composed prime set for each leaf-class. Hwang determines the size of a leaf's composed prime set, by using the equation

$$y = \left(\sum_{\text{for all } LG_i} g_i \right) + n_a + n_t,$$

where:

- LG_i is the i^{th} leaf-group. A leaf-group is any security class that is the direct ancestor of a leaf security class (Figure 3.2). The numbers of members belonging to LG is denoted as n_{lg}.

- g_i is a number such that $\binom{g_i}{k} \geq n_{lg}$, where k denotes the number of primes that distinguish a leaf security class from each leaf group.

- n_a denotes the number of leaf security classes which have two or more immediate ancestors, and n_t denotes the number of non-leaf security classes in the hierarchy.

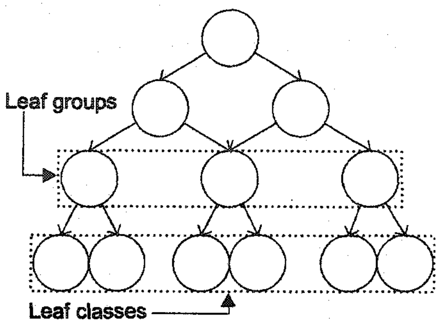

Figure 3.2: Graph illustrating Hwang-Yang leaf-group and leaf-classes.

Addition and removal of security classes is also possible, but certain additions can create the need for key regeneration on large sub-portions of the hierarchy [20]. When additions are made as leaf-classes the CA must ensure that the composed prime set assigned to the new class does not match any of the other composed prime sets. Also, all security classes principal and accessible to the added security class must have new public parameters calculated and keys assigned [20]. Further, in hierarchies where a large number of leaf-classes have more than one direct principal class, the Hwang-Yang method can be no more efficient than the Harn-Lin method, and any efficiencies that the composed prime sets brought is lost.

As noted earlier, the Hwang-Yang method of key management takes great caution to generate proper values so as to foil common modulus attacks. Hwang and Yang conjecture that the security of their method is as strong as the underlying RSA prime factorization problem.

Finally, we review a recent solution proposed by Ray *et al.* [34]. The solution proposed by Ray is best described with a diagram (Figure 3.3). In (Figure 3.3), we see an access hierarchy consisting of five security classes. Table 3.2 shows how the method proposed by Ray *et al.* assigns keys to the security classes in the hierarchy.

Each security class the hierarchy has a key which is composed from the moduli of its principals. For each $SC_j \leq SC_i$, the modulus is calculated using the procedure,

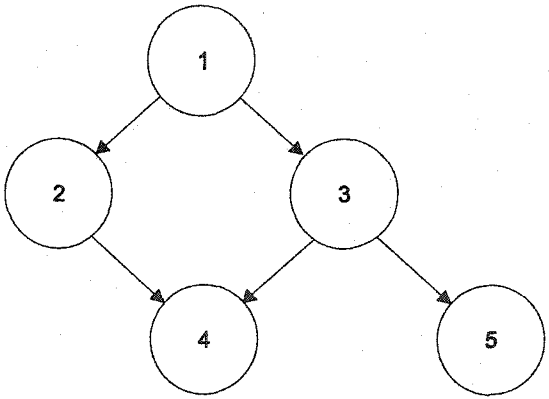

Figure 3.3: A simple access hierarchy.

Person	Assigned Security Key
1	$K_1 = (e, d_1, N_1)$
2	$K_2 = (e, d_2, N_1 \times N_2)$
3	$K_3 = (e, d_3, N_1 \times N_2)$
4	$K_4 = (e, d_4, N_1 \times N_2 \times N_4)$
5	$K_5 = (e, d_5, N_1 \times N_3 \times N_5)$

Table 3.2: Ray's key assignment scheme.

$$\begin{aligned} f &= r \times N_i \quad \text{for a random factor } r\text{, such that} \\ \gcd(f, N_r) &= 1 \quad \text{for some random } N_r\text{, such that} \\ N_j &= N_r \times f. \end{aligned}$$

Once the modulus has been determined for the security class, a decryption key is calculated using the public exponent e and the class modulus, such that

$$e_j d_j = 1 \mod \phi(N_j).$$

With each security class receiving a decryption key d, modulus N and public exponent e, the authors propose using this scheme with RSA (Sec. 2.2) to encrypt and decrypt information shared amongst members of the hierarchy.

Ray *et al.*'s method has the possibility of defining access requirements that do not match the access hierarchy's organization. For example, if 2 is not allowed access to 4's information, then 4's key should not be composed using 2's modulus (Figure 3.3). By generating keys from a subset of moduli, one can essentially compose multiple access restrictions to information and security classes [34].

Ray *et al.* state that the security, stability, and efficiency of the overall system has yet to be proven and tested [34]. In our examination of their method, we noted that most of the examples given in their research dealt with very small shallow hierarchies; thus, keys were composed from a few moduli. However, as the hierarchy becomes broader and deeper, the magnitude of the moduli increase as does the size of the decryption keys. If RSA is to be used as the underlying cryptosystem, larger keys will cause a decrease in performance of RSA [25], [34]. In response to this concern, the researchers suggest using methods, such as Fast Fourier Transforms, to speed encryption and key generation [34], although these solutions have yet to be tested. While this is a logical approach to the problem of large keys, we feel this approach has practical applications to only small shallow security hierarchies.

In terms of security, Ray *et al.*'s method is resistant to the the common modulus attack [34] and should be as secure as RSA. However, e must be sufficiently large or else a low-exponent attack (Sec. 2.2.1) is possible [34], [37].

3.4 Indirect Approaches to Key Management using One-Way Functions

The solutions we examine in this section differ from the RSA-based solutions in that the key derivation schemes all manifest an *indirect* key derivation behaviour. That is, for a principal security class to derive the key of a subordinate, some recursive procedure is executed until the desired key is derived. This is in contrast to the RSA-methods that allow security classes *direct* access to another security class's secret key via a set of public parameters into the RSA modular exponentiation function. Surprisingly, little research and experimentation has been done in the area of *indirect* key derivation constructions, and we believe there may be room for efficiency improvements. Thus, we will review the solutions proposed to date.

3.4.1 Sandhu's Indirect Approach

Sandhu proposed a method of key management on an access hierarchy represented as a simple tree [36]. Following the same assumptions as Akl-Taylor, Sandhu was the first to propose a recursive or *indirect* method of key derivation based on one-way function families [36].

His method used DES encryption as the key generation and derivation function. The key assignment and derivation scheme proceeds as follows:

- The security class hierarchy is represented as a simple tree. That is, there are no subordinate classes with more than one principal class.
- The CA is assigned to the highest security class in the hierarchy (root of the tree).
- The CA generates for itself a random key, K_0.
- If security class SC_i is an immediate subordinate of security class SC_j, then the key assigned to SC_i is,

$$K_i = E_{K_j}(name(SC_i)).$$

Here, E is the DES encryption function operating on the message block $name(SC_i)$ (the name of the security class) with a key of K_j (belonging to SC_j). The 64-bit ciphertext from DES becomes the key, K_i, assigned to security class SC_i.

At the time, Sandhu raised concerns over the efficiency, block size, key size, and ciphertext size of DES [36]. DES operates on a 56-bit key, but each round of DES produces an output that is 64-bits long [37]. This output becomes the key for the next subordinate. Thus, the output must be reduced to a 56-bit key. Sandhu noted that there existed a possibility that a degeneracy of keys from 64-bits asssigned to the security class to 56-bits used in DES may result in a collision of keys and a breach of security [36]. As for the names assigned to security classes, names larger than 64-bits was discouraged because that would require additional rounds of DES which would lead to a decrease in key generation and derivation performance [36]. However, research suggests that single round DES used in this manner is vulnerable to differential cryptanalysis [8]; thus, keys could be recovered.

We also note that Sandhu's method of traversal to generate keys does not verify access relationships before commencing the traversal [36]. In this sense, traversals to security classes where access relationships fail to exist consume time in generating no viable key. In design of our HMAC - method, we propose a solution that can verify these access relationships before we start generating the key.

3.4.2 Yang's Indirect Approach

Recently, Yang proposed a key management method for object-oriented role-based access control hierarchies [42].

To begin, Yang defines a set of one-way hash functions

$$\mathcal{H} = \{H_1, H_2, \ldots, H_n\},$$

where n is the maximum number of direct subordinate roles present in the hierarchy. Each hash function obeys the properties set forth in (Sec. 2.3). Key assignment proceeds as follows:

- For each role not part of the hierarchy, the CA generates a random key K and assigns it to the role.
- For each role that is part of the hierarchy, but does not have a principal role, the CA assigns a random key K_{role}.
- If a role R_j has only one direct principal role, R_i, and R_j is the i^{th} subordinate of R_i, then the key for R_j will be $H_i(K_{Ri})$.
- If a role R_j has more than one direct principal role, and R_j is the i^{th} direct subordinate of its left-most parent, j^{th} direct subordinate of R_m, and k^{th} direct subordinate of R_n, then the key for R_j becomes $H_i(H_i(K_{R_i}), \ldots, H_j(K_{R_m}), \ldots, H_k(K_{R_n}))$, $1 \leq i, j, k \leq n$.

Yang's method of key management is as secure as the underlying hash functions used in $\mathcal{H}$. It also improves upon Ravi Sandhu's method because hierarchies other than simple trees can be represented [42]. However, we believe there may be some implementation issues with the method that should be addressed.

For example, in implementing this method one would likely prefer to use the fastest hash functions available – dedicated hash functions. However, as noted in (Sec. 2.3.2) there are only a small number of dedicated hash functions available. If the maximum number of direct subordinate roles of a single role does not exceed the number of dedicated hash functions available, we may benefit from their use. Any more, and alternatives must be sought.

The first alternative would be to build more dedicated hash functions. As discussed in (Sec. 2.3), building secure dedicated hash functions is hard, and modifying existing ones could lead to an undermining of security (Sec. 2.3.3). Our next alternative would be to use block cipher hash functions (Sec. 2.3.2). The benefit with using the block cipher hash functions is that only one block cipher algorithm is required. Multiple hash functions can be created by simply changing the initialization value given to the block cipher (Sec. 2.3.2). This solution seems more favourable, but the penalty is that performance of block cipher hash functions is typically half that of dedicated hash functions [33] [35] [32] (Sec. 2.3.2), which will lead to slower key throughput if long tree traversals are required to generate keys. Also, not every block cipher can be used as a hash function, so particular attention must be paid to the inner workings of the cipher and its suitability as a hash function before being implemented [29] [38].

Also, in studying Yang's method, it appears that some time could be consumed trying to verify role relationships before generating a key. In Figure 3.4, we show a position role hierarchy where each position role is labeled with its position relative to its principal role. In Yang's method, positions from the left of a principal role determine which hash functions from $\mathcal{H}$ are required to generate the position role's key. From this hierarchy, it seems hard for B to efficiently verify his access to F unless he examines a diagram or representation of the hierarchy. If a public parameter for F could return its relative child position, 1, B would generate the key for D. If the public parameter returned F's relative position from the root (2,1), B would generate the key for H. Two keys which are both incorrect. An alternative strategy could be for B to search all his subordinates for F, finding nothing, or for the search to start from F and stop at A if B was not found. In either situation, generating the wrong keys and performing exhaustive searches could become time consuming if they are frequent, or if the hierarchy is broad and deep.

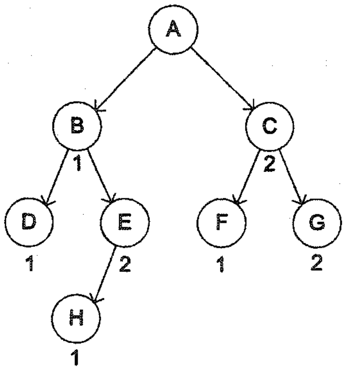

Figure 3.4: A simple hierarchy showing relative subordinate-principal positions.

3.5 Summary

We have discussed the two major approaches to the problem of key management in a hierarchy. Much of the research has been focused on modifying the Akl-Taylor model of using RSA modular exponentiation. Models following the Akl-Taylor method tackle problems with the assignment of prime numbers, storage of prime numbers and public parameters, and addition and removal of security classes from the hierarchy. The merit of these solutions is that one can easily verify principal-subordinate relationships in order to derive keys.

Indirect key derivation solutions rely on the structure of the hierarchy to provide key generation and derivation techniques. Sandhu's proposed method and approach is inefficient for large hierarchies, and the use of DES is questionable in both security and performance. Yang's method is unique, but its performance within hierarchies may degrade if long principal-subordinate paths must be searched to verify access and generate a key.

The method we present in the next chapter is not concerned with adding yet another RSA approach. Rather, we believe there may still be room to improve indirect methods of key derivation. In the chapter four, we introduce our HMAC-method and our enhancements that may help to improve the efficiency of *indirect* key derivation methods.

Chapter 4

Key Management Using HMACs

4.1 Introduction

Our contribution to the key management problem is a method we developed called the HMAC-method. The HMAC-method is an *indirect* key approach concerned with improving the efficiency of generating and deriving keys in a tree structured access hierarchy. Our method uses HMAC constructions (Sec. 2.4) built from a single and secure fast hash function, SHA-1.

In comparison to previous *indirect* approaches ((Sec. 3.4)), our method improves key generation through the use of accessibility queries using a technique we call *path addressing.* Also, by using SHA-1, we can keep key sizes small (160-bits) without sacrificing their security (Sec. 4.2.2). However, the method is also very flexible. We use only one hash function, so should concerns be raised over the use of SHA-1 or 160-bit keys, we can substitute SHA-1 with a better hash function without affecting our key generating and derivation procedures.

During the development of the HMAC-method, we faced issues that previous researchers faced: namely, the efficient addition and removal of security classes to and from the hierarchy, and accommodation of *indirect* approaches to security hierarchies structured as weakly/strongly directed acyclic graphs. With that, we present our scheme to dealing with the key update problem through a technique we devised called the *cached key* update strategy, and we present a modified *path addressing* scheme which attempts to address the problem of traversing hierarchies that are structured as weakly/strongly connected directed acyclic graphs.

4.2 The HMAC-method

We begin with a set of assumptions:

- The access hierarchy is defined and controlled by some trusted central authority (CA). It is assumed that the CA is in a secure environment that provides no viable communication channel vulnerable to attack.

- All users within the CA's environment are divided into security classes, $SC = SC_1, ..., SC_n$, which are partially ordered by the binary relation $\leq$. The resulting relations, $SC_j \leq SC_i$, means that users belonging to security class SC_j have access to information stored at the subordinate security class SC_j; however, the reverse relation does not hold. That is, subordinates are not allowed access to information stored at the principal (SC_i).

- Users belonging to a security class SC_i only know direct relationships. That is, members in SC_i know who their direct principal security class (SC_p) is, and who their direct subordinate classes (SC_s) are.

- For now, we assume the hierarchy is represented as a simple tree. That is, no security class has more than one direct principal security class, and the most principal security class is located at the root of the tree. Later, we modify the method to handle more general hierarchies represented as directed acyclic graphs.

- All keys within the hierarchy expire after time t_r. After which, new keys are generated and assigned to the security classes. Maintaining the common principle of good key management [37], we place this restriction to discourage exhaustive key search attacks and cryptanalysis of key-encrypted information.

We denote $H(K|M)$, to be the hashed message authentication code (HMAC) that uses the dedicated hash function SHA-1 (H), with a key (K), concatenated with a security class property (M).

Hierarchy preparation and key assignment proceeds as follows:

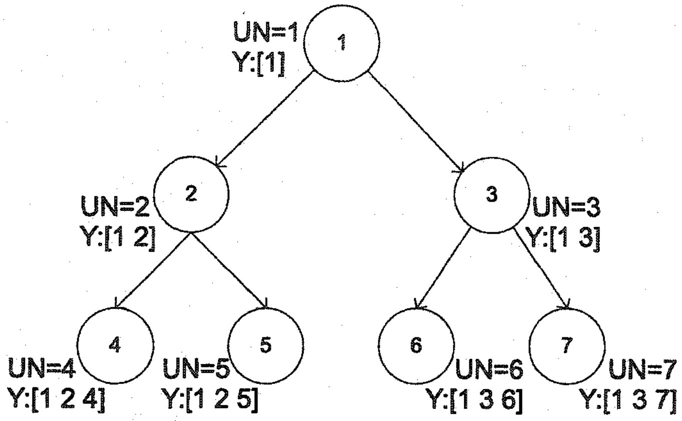

Figure 4.1: A tree-structured access control hierarchy.

1. Each security class (SC_i) in the hierarchy contains three properties: a human readable name (C_i), a unique number (UN_i), and a *path address* array (Y_i). These properties are the public parameters other security classes are allowed to view.

2. The human readable name, C, is the name of the security class that allows users to discriminate one security class from another. Common names can be any length.

3. A unique number (UN) is assigned sequentially, starting at 1 at the root and in a left-to right top-down manner, to each security class in the hierarchy (Figure 4.1). Later, as we add classes to the hierarchy, regardless of their position within the hierarchy, we assign them the next number in the sequence.

4. The path array (Y) acts as an address for a security class (Table 4.1). The address assigned to a security class records the UN traversal path starting from the root to the security class. The last entry in a security class's path array should correspond to its UN.

5. The CA assigns the root of the tree (the most principal security class) a randomly generated 1024-bit master key, K_1, which is kept secret (Sec. 4.2.2).

Unique Number (UN)	Path Array (Y)
1	[1]
2	[1 2]
3	[1 3]
4	[1 2 4]
5	[1 2 5]
6	[1 3 6]
7	[1 3 7]

Table 4.1: Summary of public parameters assigned to security classes belonging to (Figure 4.1).

6. A security class SC_k is assigned a key dependent on its direct principal security class SC_i as follows:

$$K_k = H(K_i|UN_k). \tag{4.1}$$

When a user belonging to a security class SC_i wishes to derive the key for security class SC_k, and SC_k is the direct subordinate of SC_i, then K_k is obtained using

$$K_k = H(K_i|UN_k).$$

Otherwise, if SC_k is not a direct subordinate of SC_i, SC_i proceeds as follows:

1. SC_i retrieves the path array for SC_k, Y_k.
2. Using a sequential search on the array, SC_i checks for his UN_i within Y_k.
3. If the search returns FALSE, then SC_i knows that it lacks the sufficient permission to access security class SC_k and does not proceed to generate the key. If the search returns TRUE, SC_i stops and records the index x at which its UN_i was located and proceeds to the next step.
4. Starting from $x + 1$ to the end of Y_k, SC_i generates the key using the SHA-1 HMAC. For example, if the portion of the array is $[UN_i, UN_j, UN_k]$ then the key derivation step is:

$$\begin{aligned} K_j &= H(K_i|UN_j) \\ K_k &= H(K_j|UN_k) \end{aligned} \tag{4.2}$$

4.2.1 Adding and Removing Security Classes from the Hierarchy

Ideally, it would simplify all key management solutions if the hierarchy remains static. Unfortunately, this is not always the case. As users come and go, or as an organization changes, the need to add and remove security classes from the hierarchy will arise. As such, our key management method should handle these changes.

We identified four cases for adding and removing security classes from the hierarchy. They are as follows:

1. Adding a security class to a leaf position,
2. Removing a security class from a leaf position,
3. Adding a security class to an interior position, and
4. Removing a security class from an interior position.

Adding and removing security classes to or from a leaf position is trivial. In Figure 4.2 8 is added to a leaf position, becoming the new subordinate to 2. It is assigned a common name C_8 and a unique number $UN_8 = 8$. The path array for 8 (Y_8) is created by inheriting the path array from 2 (Y_2 = [1,2]) and appending UN = 8 to the end of the path (Y_8 = [1,2,8]). Removing a security class from a leaf position in the hierarchy can be done without any affect to any principal classes.

Adding and removing security classes to and from interior (non-leaf) positions presents some challenges. In studying the key management problem, we saw that all previous *direct* and *indirect* methods dealt with the problem in a similar manner. Their designers chose to immediately re-calculate and update the keys for the affected classes [18], [34], [20], [36], [42]. This may or may not be advantageous in all situations. For example, the necessity to immediately add or remove a security class cannot be delayed or overlooked, but the disturbance caused to users within the affected security classes, or the time and resources required to update the keys would be costly or inconvenient. In these instances, it would be better to have a method that

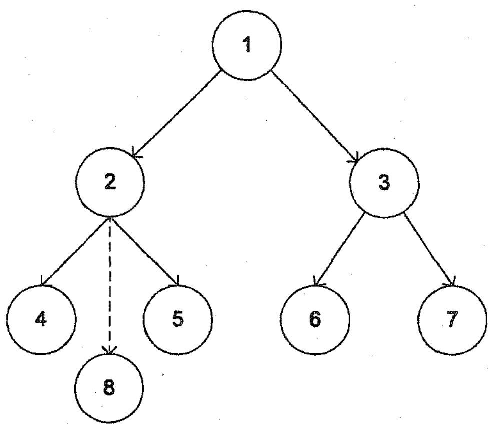

Figure 4.2: Adding a new security class to a leaf position in a simple tree hierarchy.

could delay a key update until a more convenient time arises, or as in the HMAC-method, a pre-specified key freshness time (t_r) expires.

To address the problem of key updates, we created an update strategy called the *cached key* update strategy. The cost associated with the method is that it requires a newly added security classes to have additional storage allocated for one extra key (a key cache), and a modification to the key derivation process.

Cached Key Update Strategy

The *cached key* update strategy is best understood with illustrations. In Figure 4.3, we show our simple tree access hierarchy. For brevity, we refer to security classes by their UNs (e.g. (8)).

For internal additions to the hierarchy the rule-set is as follows:

- (Figure 4.4) If a new security class (28) is added between two classes whose key caches are empty, (1,2), the CA assigns the new class (28) a path address from 1 (Y_{28} =[1,28]) and a key from its direct principal class ($K_{28} = H(K_1|28)$). The CA also provides the key for the direct subordinate (2) to the new class (28), which the new class (28) will store in its key cache. If additional classes (25) are added to the new class

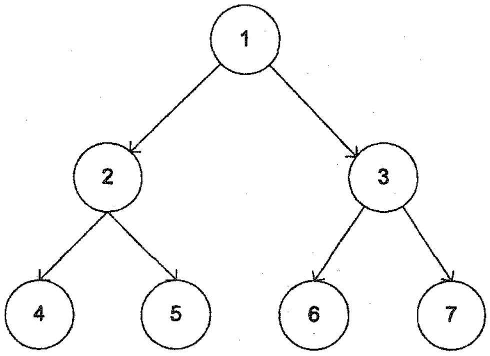

Figure 4.3: A simple tree hierarchy.

(28) as direct subordinates, they are assigned a path and key relative to the new class (28) – (25: K_{25}=H(K_{28}|25), (Y_{25} = [1,28,25])).

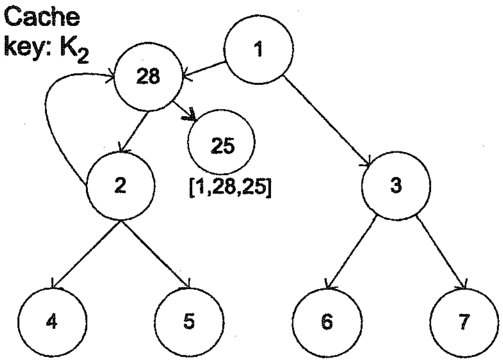

Figure 4.4: Adding a new security class between two classes with empty key caches.

- (Figure 4.5) If a new security class (38) is added between two classes where the subordinate key cache is empty (2) and the principal full (28), the new class (38) is given a path from 1 (Y_{38} =[1,28,38]), a key derived from its parent's key (K_{38}=H(K_{28}|38)), and a key from the direct subordinate (2) which the class (38) will store in its key cache. If additional classes (25) are added to the new class (38) they will receive a path containing the new class (Y_{25} =[1,28,38,25]) and a key generated from the new class (K_{25} =H(K_{38}|25)).

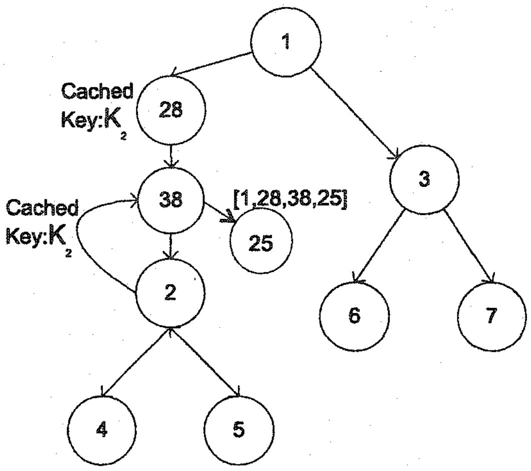

Figure 4.5: Adding a new security class between two classes were the subordinate's key cache is empty.

- (Figure 4.6) If a new security class (48) is added between two classes where each key cache is full (28,38), we initiate an update. Key caches are cleared (28,38). The new class (48) is given a path from 1 (Y_{48} =[1,28,48]) and a key derived from its principal's key (K_{48} =H($K_2$8|48)). The classes subordinate to the new class (38,2,4,5) have their keys regenerated and paths updated.

- (Figure 4.7) If a new security class (58) is added as a principal to a class that has a full key cache (28), we initiate an update. Key caches are cleared. The new class (58) is given a path from 1 [1, 58]. The classes subordinate to the new class (28,38,2,4,5,...) have their keys regenerated and paths updated.

The process to remove a key follows a similar method to addition. The ruleset is as follows:

- (Figure 4.8) If the security class being removed (8) has subordinates that are leaf-classes (9, 10), the leaf-classes (9, 10) are assigned to the

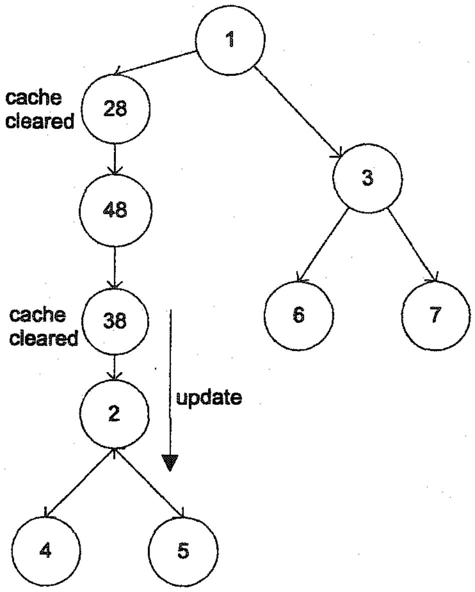

Figure 4.6: Adding a new security class between two classes with full key caches.

principal (2). Because the paths from the principal (2) to new subordinates (9,10) is short, there are two options. First, ff the update of keys and paths to the subordinates (9,10) would cause no inconvenience, then the keys and paths may be updated immediately. Otherwise, the principal (2) caches the key of the outgoing class (8).

- (Figure 4.9) If the security class being removed (5) has subordinates that are not leaf-classes (8,9,10), the principal (2) receives the outgoing class's (5) key to store in the key cache, and the subordinates belonging to the outgoing class (9,10) are added as subordinates to the principal (2).

- (Figure 4.10) If the security class being removed (8) is not a leaf-class and is subordinate to a principal whose key cache is full (2), the principal receives the subordinate classes (9,10) of the outgoing class (8), clears its (2) cache, and updates paths and keys to all its subordinate classes (9,10).

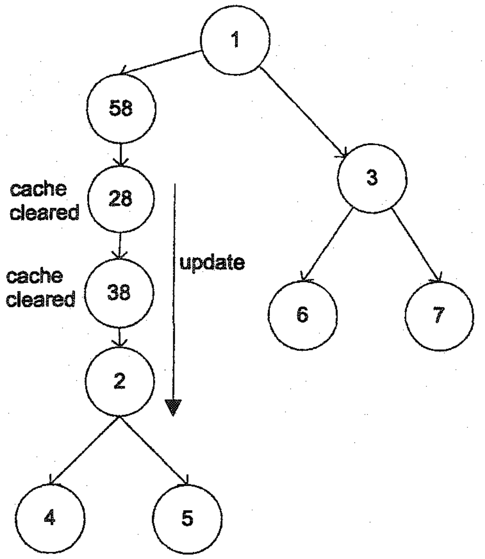

Figure 4.7: Adding a new security class as the principal to a class that has a full key cache.

Each security class holding a cached key must modify its search strategy when searching a path array. For example, in Figure 4.4 if 28 requests the path array for 4 (Y_4), then 28 searches Y_4 for its UN and the UN belonging to the cached key (K_2). If it locates the UN belonging to its cached key (K_2), then 28 uses the cached key (2) to derive the key for 4 by following the path and using the cached key. Otherwise if 28 finds its UN within Y_4, it uses its key to follow the path and recursively generate the key for 4 (Eqn. 4.2).

Another benefit of the cached key strategy is that we may be able to accommodate additions where previous deletions occurred. For example, in Figure 4.8 if a class was added into the position previously held by 8, we could modify our addition strategy to have the CA simply reassign the new class a $UN = 8$ and the key held in cache by 2 (K_8). This would suggest that rather than removing a class completely from the hierarchy, the better strategy might be to have the CA maintain a deletion list to keep track of classes that are removed. This way, if a new class is re-introduced to a deletion point, adding it can be accommodated much more easily than undergoing a completely new addition step.

To summarize, the *cached key* update strategy proposed here is to address concerns with the overhead and costs incurred if we update the hierarchy in response to every addition and deletion of a security class. As the name suggests, the *cached key* update strategy sacrifices a small amount of storage per affected security class. Updates to portions of the hierarchy are delayed if such updates would be costly or inconvenient. Fortunately for additions, the cached key is assigned to the newly added security class, thus it may be easier to assign a cached key to a new class than to update keys in the affected subordinates. For example, a new employee or level of management is brought into the organization and must begin work immediately, but it would be costly and inconvenient to re-calculate and re-issue keys during the work day. It would be easier to give the new level of management or worker an additional key for a brief period of time then to re-issue keys to all security classes.

A removed class provides its key to its principal so that the principal may access the inherited subordinates. Using a deletion list and cached keys, we may also be able to accommodate unique situations where classes are deleted, yet new classes are re-introduced into the same position sometime later. We should reiterate that the *cached key* update strategy may not be suitable for all situations. The nature of the keys and organization may warrant the simple *immediate* update strategy that previous *indirect* and *direct* methods took. For example, an employee leaves abruptly and management wishes to change all the keys immediately.

In our *cached key* update strategy, the size of the key cache determines the delay between key updates. Increasing the number of cached keys will increase the delay between key updates. However, increasing the size of the key cache will require some modifications to the rule-set of adding and removing classes. With one key cache, we are able to cache keys belonging to immediate subordinates. Consequently, in situations where more than one key from a subordinate would need to be cached, we currently initiate an update. With larger key caches the rule-sets will need to be modified to reflect the fact that keys from lower subordinates will need to be cached as well. The need for larger or smaller key caches will be dependent upon the nature of the organization the hierarchy represents and/or the key expiry and key update schedule within an organization.

We believe the *cached key* update strategy helps improve the flexibility of our HMAC-method with respect to previous *indirect* methods. With respect to *direct* methods of key management, we found no method proposed to date that can adequately accommodate a similar key update strategy for the addition of security classes. This limitation arises from the fact that in the access relationships between security classes within the hierarchy the public parameters must remain factors of one another. Otherwise, the test for divisibility $(PB_1|PB_2)$ (Sec. 2.2) fails and key derivation cannot take place.

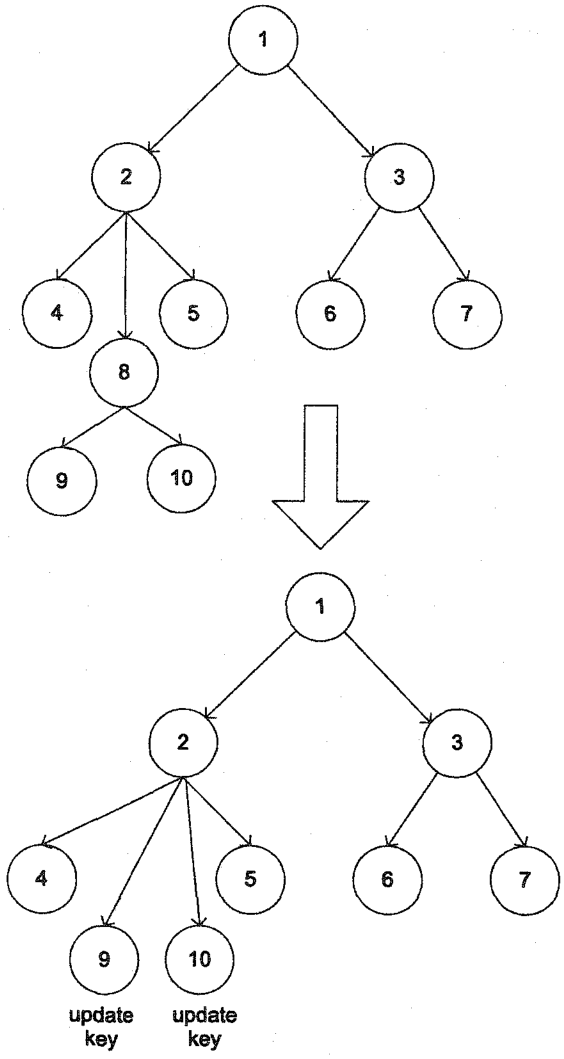

Figure 4.8: Security class (with leaf-classes) being removed.

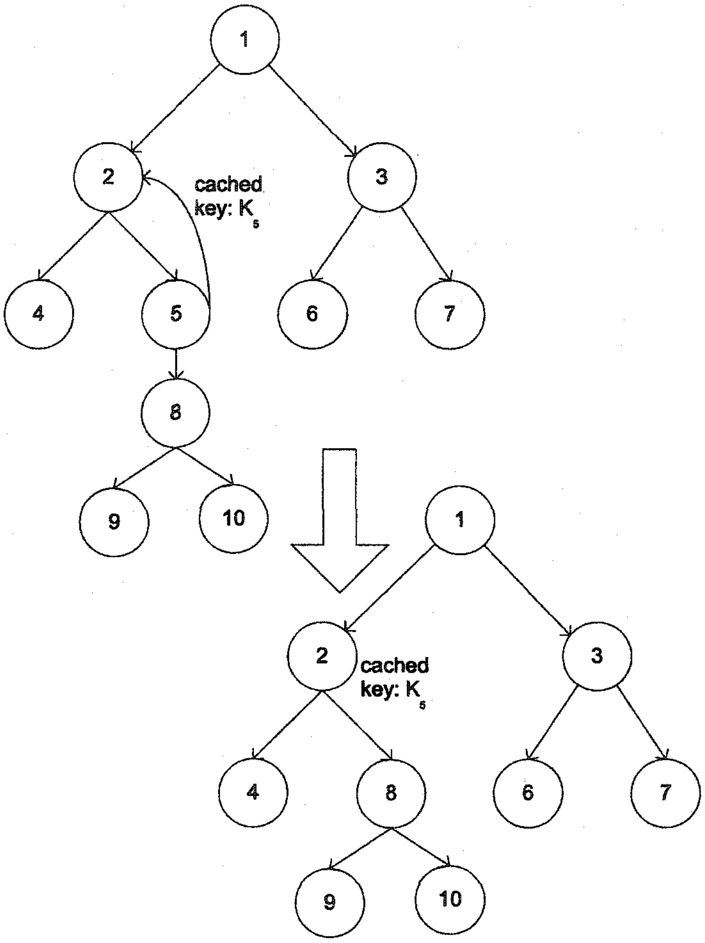

Figure 4.9: Security class (with non-leaf subordinates) being removed.

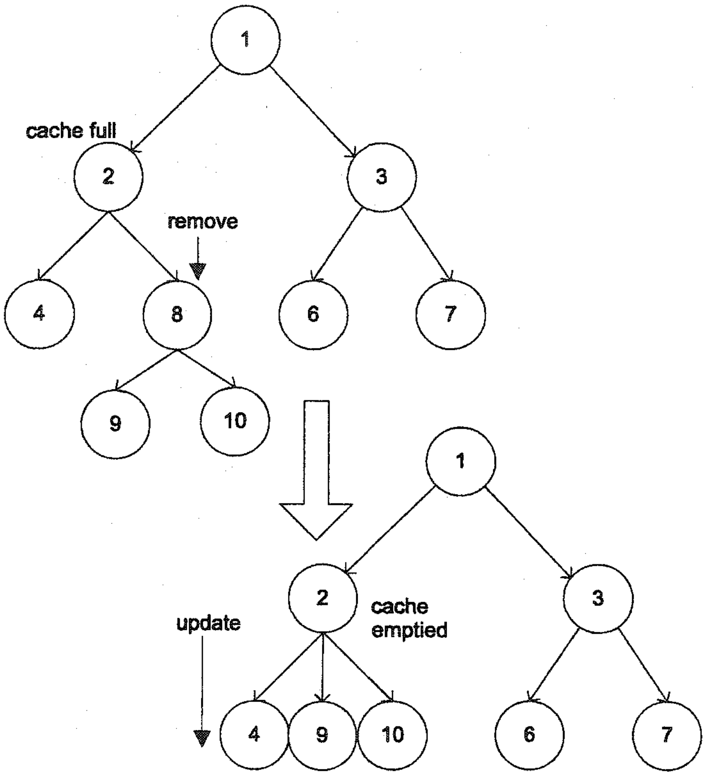

Figure 4.10: Removing a subordinate from a principal with a full key cache.

4.2.2 Security of the HMAC-method

The security of the HMAC-method lies in the underlying security of the master key, the SHA-1 dedicated hash function and in the HMAC construction.

We chose the master key be at least 1024-bits in size, because the master key can derive all keys within the hierarchy. Such a large key size helps thwart random guessing of the master key. The probability of an attacker guessing the key would be 2^{-1024}. An exhaustive key search of the key would require the attacker to generate and test all 2^{1024} possible key combinations. We believe this key size will provide adequate protection of the master key.

As discussed in Sec. 2.3, there is no mathematical means in which to show that a particular hash function is truly one-way. We cannot say that a fixed-output hash function is truly collision resistant (Sec. 2.3). However, as we discussed in Sec. 2.3.3, there are certain features which help hash functions resist attack.

We chose to specifically use SHA-1 in the HMAC-method, because it produces a digest length of 160-bits, resulting in 2^{160} possible message digests (Sec. 2.3.3). Under the birthday attack, to force a collision under SHA-1 would require an exhaustive search and comparison of at least 2^{80} message digests (Sec. 2.3.3). However, without having access to the message digests that are being used as keys for security classes, the attacker would have to generate all 2^{160} message digests and test each one he creates against each security class in the hierarchy. This increases the complexity of the attack. If the attacker tries to simply guess a key, his probability of success is 2^{-160} (Sec. 2.3.3).

Also, in choosing SHA-1 for our hash function, we note that as of the date this thesis was written, no one has successfully attacked SHA-1 [37]. While this does not prove SHA-1 is undeniably secure, it does speak to how well SHA-1 was designed and how strong the hash function has proved to be. If in the future SHA-1 was found to be insecure, the dedicated hash function used in the HMAC-method can be easily replaced with a better one. For example, nothing prevents the use of RIPEMD-160 [15] or an MDx-MAC enhanced [33] hash function (Sec. 2.4). In a worst case scenario, if we have no suitable dedicated hash functions available, then we may use a block cipher hash function at the expense of throughput.

As for our HMAC construction, we based our choice of construction in accordance with Preneel's recommendations for HMAC security (Sec. 2.4.3).

Bart Preneel studied HMAC constructions and found that certain constructions could be compromised if an attacker obtained many message digests calculated with the same key [33]. In response, he recommended some constructions to strengthen HMACs (Sec. 2.4.3) [37] and proposed a modification, MDx-MAC, that can strengthen dedicated hash functions used in HMAC constructions [33].

We are not at danger in using the construction $H(K|M)$ within the HMAC-method. As Preneel noted, the attacker attempts to use a birthday attack to create collisions on the digests [33]. For this attack to work, the attacker would require at least 2^{80} message digests (keys) from the hierarchy to compare against his results. Otherwise, an attacker is left no other option but to randomly guess a key or generate all $2^1 60$ possible keys and try them against all classes in the hierarchy.

We are also not at danger in using a single number as the security class property within the HMAC construction. The first keys generated by the HMAC-method are created with the 1024-bit random key assigned to the principal class. When keys are re-calculated, the 1024-bit random key is also re-assigned. So, from key assignment to key assignment the keys for security classes will be different. Also, SHA-1 is designed to use all bits of the pre-image within the rounds of the compression function [21]. Thus, if the derivation of two keys only differs by a single number (8 bits), under SHA-1 they should be different [21].

4.3 Considerations and Limitations

One drawback of any *indirect* key management method is the computational complexity of the key derivation process. Consequently, the best approach to improve efficiency is to try and optimize the derivation process.

In the HMAC-method, we looked to achieve better traversals then [36] by implementing the *path addressing* array. The array allows us to verify class relationships before undertaking a key derivation process. Thus, we search only the path that can lead us to the desired security class. We do not spend time searching all portions of the sub-tree and creating keys as we go. In the worst case, we search through an entire path array; however, we do avoid the cost of searching the entire hierarchy, and we do avoid the cost of having to create keys as we search a path. With path addressing, we only create keys once we verify the searcher's UN is in the path.

In choosing the format of the path array, we found that using just numbers is more efficient that using a string concatenation strategy. For example, we could have represented a path to a security class by concatenating the names of security classes along the path and using some delimiter to separate names. We found this approach too inefficient. First, named paths are much longer than numbered paths. For example, assume a security class is represented by a modest 8 character (80-bit) common name (C), and we have a leaf security class whose path is 50 traversals away from the root. Using a name concatenation approach, the leaf class would be assigned a name $80 \times 50 = 4000$ bits long. If we use 16-bits to represent the numbers in our *path addressing* array, the numeric path to the same leaf class would require $50 \times 16 = 800$ bits – 80% less storage than name concatenation. Also, having a numeric array allows us to search more efficiently and save time when trying to verify an access relationship. With a string concatenation approach, we must sequentially search through each character of the string looking for name delimiters, construct the names, perform a string comparison, then continue to move along the string. With numbers in an array, search and comparison can occur much faster.

In comparison to Yang's method [42], we believe our path addressing scheme is more efficient because Yang's principal-subordinate search strategy is local in scope. That is, for a principal to derive the key of a subordinate lower in the hierarchy, a path must be traversed through the hierarchy by following through the principal-subordinate relationships. Once the proper sequence of principal-subordinate relationships is discovered, then the corresponding sequence of hash functions is called to create a key. Otherwise, the search halts and returns no result, and we incur the penalty of using time to conduct a search without a successful key. In our approach, we simply scan though the path address array and avoid traversing other hierarchy relationships. As soon as a security class finds its UN within the *path addressing* array, it begins to assemble the key using the remaining numbers in the array.

In our *cached key* update strategy, the size of the key cache determines the delay between key updates. Increasing the number of cached keys will increase the delay between key updates. Using a single key cache requires that a class make room for an additional 160-bit key, an added comparison during a path search, and an additional time factor during key derivation as the cached key is retrieved and used. However, if performance is a concern or key updates are inconvenient and must be delayed to a more opportunistic time, we believe our *cached key* update strategy improves the applicability and efficiency of the *indirect* approach in these situations.

In addition to the *cached key* update strategy, the HMAC-method also benefits from the performance of using the SHA-1 hash function. Preenel's examination of hash functions showed that an optimized SHA-1 implementation had a digest throughput of approximately 54.9Mbits/sec [32]. Given that our keys are 160-bits[1] concatenated with the string representation of the UN, we can estimate the maximum key generating throughput of SHA-1 in the HMAC-method to be $\approx$ 300,000 keys / sec. If searching a path array is as efficient as key generation, this suggests that the HMAC-method could adequately deal with path searches and key generations on very large and deep hierarchies. Empirical evidence from experimentation would be required before a more definitive metric in throughput performance could be reached.

The consequence of using both the *path address* array and the *cached key* update strategy is that we must hold an additional set of public parameters and cached keys in order to implement these improvements. For a hierarchy containing n security classes, the total space required for the *path address* arrays would be $O(n^2)$. With cached keys, we require an additional 160-bits per key. Given the minimum 512-bit key lengths the RSA-based methods are required to produce, the additional 160-bits for a cached key seems small. With the benefits a *cached key* update strategy could have for an organization, we feel the additional 160-bits of space is acceptable. If the cached key update strategy is not required, there is the option to use the *immediate* key update strategy that the other methods use. The option to use either one simply adds flexibility to our approach.

Finally, in studying Yang's approach to *indirect* key management, he was able to accommodate hierarchies structured as directed acyclic graphs by using a family of hash functions [42]. In the HMAC-method, we have presented thus far, our method is only designed to efficiently handle hierarchies structured as simple trees. To address this shortcoming, we present a design that modifies the *path address* array so that the HMAC-method could be applied to hierarchies represented as directed acyclic graphs (DAGs).

[1] Except for the root of the tree – that is 1024-bits.

4.4 Modified Path Addressing Scheme for DAG Hierarchies

In access hierarchies represented as directed acyclic graphs, a subordinate security class can have more than one direct principal class (Figure 4.11). Consequently, the subordinate requires a key that can allow both direct principal classes access to the subordinate.

With *indirect* approaches, directed acyclic graphs present issues regarding traversals of paths. Referring to Figure 4.11, when either 2 or 3 wishes to access information stored at 5, they must have some knowledge about the composition of 5's key. Similar to Yang's approach, we chose to generate the subordinate's key by composing the direct principals' keys. For example, in Figure 4.11 the key assigned to 5 would be

$$K_5 = H(H(K_2|5)|H(K_3|5)).$$

Thus, in order for either 2 or 3 to derive the key for 5, either 2 would require the knowledge of the sub-key $H(K_3|5)$, or 3 would require the knowledge of the sub-key $H(K_2|5)$. We chose to have the CA cache the sub-keys. A consequence of this approach is that principal classes which share a direct subordinate will rely upon the CA to provide the cached key.

However, the more immediate problem was how to represent this security class as having a key composed from two or more direct subordinates. Yang's solution was to use multiple hash functions, but we preferred the flexibility of having a single fast dedicated hash function within the HMAC-method. The best solution we could devise modified the *path address* array (Y).

Using Figure 4.11 as an example, Table 4.2 shows how each security class's path address would appear under the modified addressing scheme. For brevity, we refer to security classes by their unique numbers (UN).

Our modification was to change the structure of the path address from being an array to being a list that could contain nested lists. A nested list within the list (e.g. ((2 3) , 5)), indicates that the security class holds a key which is composed from the sub-key belonging to the security classes in the nested list. For example, from Table 4.2 the address list for security class 5, ((2 3) , 5), indicates that 5 is composed from the sub-keys of security class 2 and security class 3. Security classes that are not composed of sub-keys are simply represented as a list of numbers (e.g. 4, (1 , 2 , 4)). Next, we modified the key derivation process to reflect the modified *path list* addressing scheme.

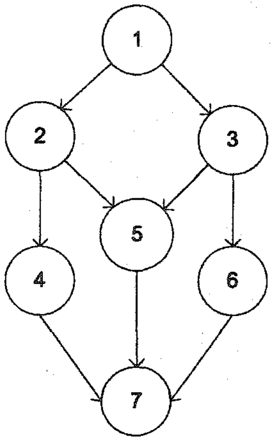

Figure 4.11: Directed acyclic graph structured hierarchy.

If the first element in a *path list* is not a nested list (e.g. (1 , 2 , 4)), then we know that a direct path exists to the desired class and we operate on the list as if we were using the *path address* array from a security class in a simple tree hierarchy.

If the first element in a *path list* is a nested list (e.g. ((2 3) , 5)), we implement an *expand-and-search* strategy. For example, in Figure 4.11 if security class 1 wanted to access security class 5, it requests the *path list* for security class 5, (((2 3) , 5)), notices the nested list as the first element, and proceeds to follow the expand-and-search strategy:

1. The security class searches the nested list looking for its UN. In our example, 1 searches the list ((2 3) , 5) and does not find itself in the nested list.

2. If the UN is not located within the nested list, the address for the first element in the nested list is expanded. In our example, the list ((2 3) 5) is expanded and becomes (((1 2) 3) , 5).

3. The principal security class repeats steps 1-2 until it finds its UN within an expanded list. In our example, 1 will find itself in the expansion of 2's address list: (((1 2) 3) , 5). If 1 did not find itself in the

Security Class	Path Address
1	(1)
2	(1, 2)
3	(1, 3)
4	(1, 2, 4)
5	((2 3), 5)
6	(1, 3, 6)
7	((4 5 6), 7)

Table 4.2: Modified path addresses for security classes in a DAG access hierarchy.

expanded list of 2, it would move onto the next element, 3, and perform the expand-and-search again.

4. Once the UN is found within a list, the key represented for that list is generated. In our example 1 will create the key for 2 by following the path address $((((1\ 2)\ 3)\ , 5) \rightarrow ((((K_2)\ 3)\ , 5)$.

5. At this point, search-and-expand stops and the security class will request the CA to produce the sub-keys for the other members of the sub-list. In our example, having generated the key for 2, 1 will stop expand-and-search and request the CA to produce 3's sub-key for 5: $((K_2, K_{3_5}), 5)$.

6. Having received the remaining sub-keys from the CA, the principal can combine them in order to produce the key for the desired subordinate. In our example, 1 will create 5's key using $H(H(K_2|5)|K_{3_5})$.

With this new derivation method, the best case scenario is that the first element in the nested list produces a valid key and the search-and-expand is aborted so that the remaining sub-keys can be requested. The worst case scenario is that all members of the nested list undergo search-and-expand and no keys are produced. This situation could occur frequently in weakly connected directed acyclic graph access hierarchies. For example, Figure 4.12 shows just such a hierarchy. If 1 were to request access to 4, it would spend time performing search-and-expand only to find that it lacked the proper access. Having spent time with the problem we leave it as open and state that although our newly devised *path list* addressing scheme could accommodate DAG hierarchies, it is not an optimal method for all DAG hierarchies.

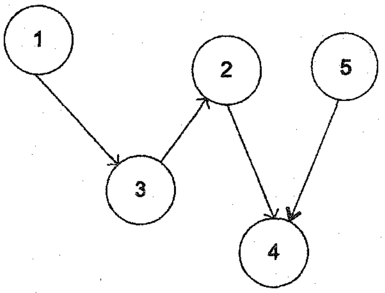

Figure 4.12: A weakly connected DAG hierarchy.

4.5 Summary

We have introduced the HMAC-method of key management for an access control hierarchy. We propose a solution that seeks to improve the efficiency and applicability of *indirect* approaches to simple tree hierarchies. The HMAC-method is as secure as the underlying SHA-1 hash function and chosen HMAC construction.

Our contributions to the *indirect* approach for key derivation was the creation of a *path array* and *cached key* update strategy. The *path array* allows us to assign numeric addresses to security classes in the hierarchy so that we may verify access relationships before we generate keys, improving performance by not producing unnecessary intermediate keys or searching the entire hierarchy. The *cached key* update strategy was in response to our observations that there may be a better strategy to updating the hierarchy under structural changes. The *cached key* update strategy allows for changes to the hierarchy, but key updates to any affected security classes can be reasonably delayed if immediately updating the keys to affected subordinates would be costly or inconvenient.

Finally, observing that Sandhu was unable to address DAG hierarchies, we set out to modify our *path addressing* scheme so that the single function HMAC-method could also apply to DAG hierarchies. Yang was able to deal with the problem by using multiple hash functions, but we wished to continue using one hash function. As such, we were able to make modifications to the *path addressing* scheme so that DAG hierarchies could be representend. Unfortunately, the search-and-expand method developed for use with the modified scheme is not optimal. Finding an optimal method is left as an open problem.

In the next chapter, we will address some observations we have on factors we believe can affect the suitability and implementation of key management methods.

Chapter 5

Implementing Key Management Methods: Analysis and Considerations

5.1 Introduction

In the previous chapter, we introduced the HMAC-method and our improvements to previous *indirect* approaches. We addressed issues of efficiency when determining role relationships, strategies to handle key updates, and weakly/strongly connected directed acyclic graphs.

In this chapter, we examine the pragmatic issues which can affect the implementation and applicability of *direct* and *indirect* methods.

5.2 Direct Versus Indirect Key Management Methods

The comparison of *direct* and *indirect* methods is difficult. This is in part to the nature of how each method attempts to solve the problem of key management. In the *direct* approach, the challenge is to discover an optimal solution of assigning primes to the members of a hierarchy. For small hierarchies, the methods can provide reasonably-sized public parameters, but for larger hierarchies, the public parameters can become quite large to store and manipulate. However, *direct* methods can address directed acyclic graph hierarchies more easily than the *indirect* approaches.

In the *indirect* approaches, storage requirements are not as large. The challenge in *indirect* approaches is to efficiently derive the keys recursively from the structure of the hierarchy. Directed acyclic graphs can prove difficult to address because, parties holding authority over a common subordinate require cached sub-keys. Having cached sub-keys may also alter the way in which users interact with the central authority. The benefit of the indirect methods is that procedures to assign keys to the hierarchy is easily done, and we may construct flexible key update strategies to address a dynamic hierarchy.

In studying the key management problem, we observed that certain factors pertaining to the hierarchies, and certain pragmatic factors with respect to an implementation can arise, and that in understanding these issues, we may be able to design more applicable key management schemes. That is, rather then attempting to fit one particular key management method to all hierarchies, a better approach would tailor the key management method to the structure of the hierarchy, the nature of the organization, and the application of the keys.

5.2.1 Structural Properties of the Hierarchy

The three structural properties affecting all hierarchies are breadth, depth, and connectedness. Each one will vary depending on the nature of the organization and the relationships between security classes within the hierarchy.

We summarize the space complexity of the Akl-Taylor, Harn-Lin, and Hwang-Yang methods in table Table 5.1 [20]. In the method of Ray *et al.* (Sec. 3.3) (not shown Table 5.1), a user's key is dependent on the size and number of moduli (m and d respectively) composed to produce his key. In the HMAC-method, our storage complexity is $O(n^2)$ and the largest public parameter would be the longest path address (l).

Breadth and Depth of a Hierarchy

For very small and shallow hierarchies, as shown in Figure 5.1, both *indirect* and *direct* schemes work well. The public parameters stored by the *direct* schemes are small and the paths traversed by the *indirect* approaches are short. Given the direct approaches small storage requirement and $O(1)$

Method	Number of Primes	Maximum Public Parameter	Storage Space
Akl-Taylor	n	$\prod^{n}_{i=1} e_i$	$O(n^3 \log n)$
Harn-Lin	n	$\prod^{n}_{i=1} e_i$	$O(n^3 \log n)$
Hwang-Yang	y	$\prod^{n}_{i=1} e_i$	$O(n^3 \log y)$

Table 5.1: Space complexity for direct key derivation schemes. n is the number of security classes in the hierarchy, y the number of primes the Hwang-Yang method requires (Sec. 3.3).

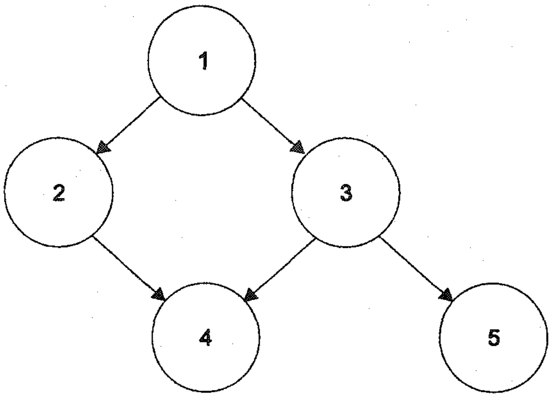

Figure 5.1: A small and shallow tree shaped access hierarchy.

computational complexity to derive keys, *direct* approaches may be more appropriate for small hierarchies than *indirect* approaches.

With broad and shallow hierarchies (Figure 5.2), we see a dramatic difference between the *direct* and *indirect* approaches. Here, the public parameters of *direct* approaches will increase rapidly as more security classes are added to the hierarchy.

Amongst the *direct* approaches, the Hwang-Yang method should be more efficient for storage than the Harn-Lin approach because of Hwang-Yang's use of *composed prime sets* (Sec. 3.3). However, while the Hwang-Yang method does use fewer primes, the public parameters may not necessarily be smaller. The hierarchy shown in Figure 5.2 was taken from [20] where we attempted to implement the Hwang-Yang approach. In following their recommendations

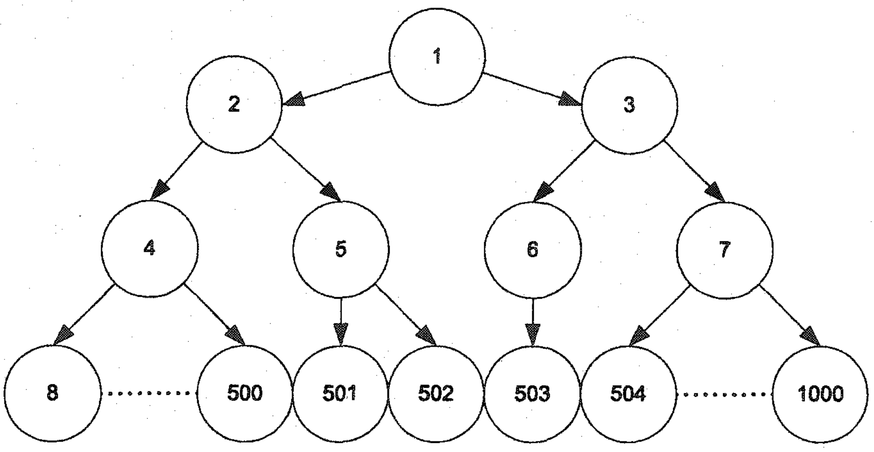

Figure 5.2: A broad and shallow tree shaped access hierarchy.

of prime number selection and composed prime set assignment, our use of the first 42 primes did not show an improvement over the 1000 primes used by the Harn-Lin method. The public parameter stored for the most principal class under Harn-Lin was 3393 digits long and in the Hwang-Yang approach it was 5204 digits long. We attribute this discrepancy to the selection and assignment of the composed prime sets. The Hwang-Yang approach should produce smaller numbers, but only if the optimal selection and assignment of composed prime sets is found. This suggests that although the Harn-Lin approach uses more primes, it can be more efficient and more applicable than the Hwang-Yang approach. For the Hwang-Yang approach to produce smaller numbers when implemented will require solving the larger problem of optimally assigning composed prime sets [20]. From our results, we found $O(n^3 \log n) \leq O(n^3 \log y) \leq O(n^3 \log n)$, for Harn-Lin, Hwang-Yang, and Akl-Taylor respectively.

For the hierarchy in Figure 5.2, we expect Ray *et al*'s method and our HMAC-method to perform well. Under the HMAC-method, the small path addresses can quickly be searched. However, compared to Ray *et al's* method where key sizes will increase as one moves deeper into the hierarchy, our HMAC-method will assign the same sized key to all classes.

As the hierarchies become deeper and broader, the prime numbers within the *direct* methods will continue to increase and public parameters will continue to grow in magnitude. Within Ray *et al.*'s method, key sizes for subordinates will continue to increase, and in the HMAC-method the number and length of path addresses will become longer. With very large hierarchies, it becomes difficult to predict which method will perform the best and at which point these systems begin to fail or become cumbersome.

Weakly/Strongly Connected Hierarchies

Connectedness of a hierarchy describes the degree of relationships between the security classes. In linear or tree hierarchies, these relationships are well defined with classes having only one direct principal and no shared subordinate classes.

Shared classes increase the connectedness of a hierarchy. Within *direct* approaches shared classes will result in larger public parameters for the principals of the common subordinate, while in and Yang's method and ours, shared classes result in sub-keys having to be cached for each principal of the common subordinate. In the HMAC-method, we incur the overhead of the additional search-and-expand method we use to verify access relationships before we create keys.

Unfortunately, the degree of connectedness between classes will vary depending on application and organization and in some instances one method may be more applicable than another. For example, consider Figure 5.3 which illustrates a hierarchy where a few administrative users share access to many different subordinates. This hierarchy could represent a student records database in a university, where each individual student record is protected by the student's key. If there are 7000 students attending the university, then under the *direct* approaches, each administrative class is assigned a public parameter composed of 7001 prime numbers. Under our HMAC-method and Yang's method there would be 49000 shared sub-keys. Finally, under Ray *et al.*'s approach the student would hold a key 8 times larger than the administrative staff.

At first glance, it would appear that Ray *et al.*'s approach seems the most efficient of the three. However, in this situation we can improve the efficiency of both *indirect* approaches by changing the connectedness of the hierarchy and implementing a common security class shared amongst the administrators for accessing the database (Figure 5.4). Now, the number of sub-keys drops from 49000 to 7. Unfortunately, in this instance, our changes will not improve the magnitude of the public parameters within the *direct* methods.

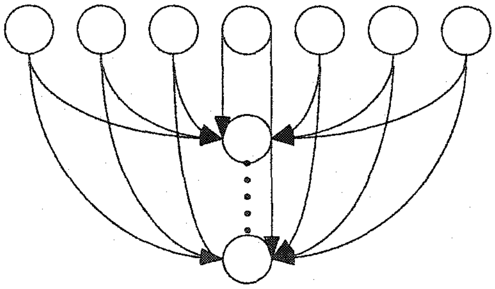

Figure 5.3: A hierarchy of a university student record database.

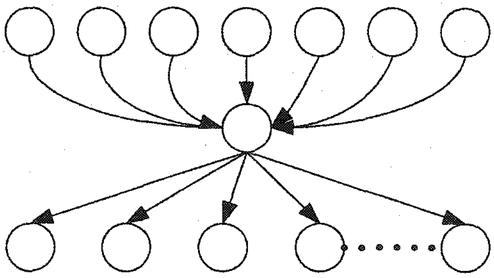

Figure 5.4: A hierarchy of a university student record database where an addition of a common security class allows us to apply *indirect* methods more efficiently.

Thus, the role connectivity plays within a hierarchy is also important. There may be instances, such as the student database, where understanding the nature and intention of the relationships may allow us to manage connectivity better and improve the applicability of key management methods we have available.

5.2.2 Key Sizes and Key Updates

No encryption key should be used for an indefinite period of time [37]. The longer a key is used, the greater the chances of the key being lost, compromised, or stolen, and the greater the chances the data protected by the key becomes cryptanalyzed [37]. Thus, key updates should occur within any key management method. However, the size and type of key, and how and when we update keys can bring forth interesting questions regarding the applicability of one method over another.

The size and type of key is different between *indirect* and *direct* approaches. Within Yang's approach and ours, the keys used are generated from hash functions with digest spaces of 2^m, where m is the digest size. Thus, for an attacker to recover a key, he must exhaustively search all 2^m possible combinations of keys. For digest sizes of 160-bits or more, this provides a sufficiently strong key for most encryption/decryption of data with a symmetric cryptosystem [37]. With *direct* approaches using full RSA implementations (Hwang-Yang, Harn-Lin, Ray *et al.*), this is not always the case. The weakness in RSA keys is their weakness to factorization [25] (Sec. 2.2). The consequence is that in order for RSA keys to provide a key strength similar to a hash function's symmetric key, RSA keys must be longer – typically 512-bits or more [25].

These differences in key length may not be important for some application of keys. For example, if keys are used only to authenticate users to services or to authenticate their identity, the length and type of key is unimportant, but if the intention is to use the keys within symmetric cryptosystems, then key type and key size becomes a consideration.

Because most symmetric cryptosystems have key sizes fixed between 64 and 256 bits [37][38], the much larger 512-bit RSA key will need to be truncated. How truncation will affect the key's security is difficult to determine [36]. The solution may be to use smaller RSA keys and simply have more key updates, but if information is stolen between key updates and the modulus within the smaller RSA key is known, then factorization of the key and recovery of the data is possible. In this respect, the more appropriate symmetric keys that Yang's method and our HMAC-method produce would be more suitable for use with symmetric cryptosystems.

As for key updates, the social organization of the hierarchy can have an effect on the suitability of a key management method. For example, in some organizations members near the bottom of the hierarchy are the most transient. In corporations, these may represent the workers or contract employees.

There may be situations where, for cost or convenience, an organization decides to have monthly key updates, but workers are added and removed to the hierarchy throughout the month. With *direct* approaches, the addition of a new user to the hierarchy requires that all classes principal to the subordinate must have their public parameters updated and keys re-calculated.

With our HMAC-method, Yang's method, and Ray *et al.*'s method this does not occur because key updates occur from principal to subordinate. Thus, in the situation we propose with the contract employees and the corporation, we may add new security classes to the bottom of the hierarchy without initiating key updates to principal classes before they should occur.

An additional benefit that our HMAC-method brings to key updates is the flexibility for a principal to initiate a key update on just its portion of the hierarchy. Because we calculate keys for subordinates through a combination of the principal's key and the subordinate's unique number, if we need to refresh the keys for a particular part of the hierarchy, we may assign new unique numbers to the direct subordinates, update their paths, and calculate new keys and paths for the remaining subordinates. In situations where we would like to update keys to a specific area of the hierarchy, we are free to do this before a scheduled key update without affecting classes principal to us. Another method that can do this is Ray *et al.*'s where a subordinate's key can be re-calculated with a new modulus and composing it with the moduli from its principals (Sec. 2.2). Other methods cannot do this. Keys must either be re-calculated for the entire hierarchy (Akl-Taylor, Yang) or for affected sub-portions of the hierarchy (Harn-Lin, Hwang-Yang).

5.2.3 Role and Design of the Central Authority

The central authority (CA) has appeared throughout most of the literature [3], [10], [18], [20], [36], [42], yet very little discussion goes into the design and implementation of it.

For example, in our HMAC-method and Yang's method we call upon the CA to cache the sub-keys produced for shared subordinates. One concern with this approach is that the sub-keys, if combined together, can produce a valid key for a shared security class. In an implementation, the pragmatic issue would be how to securely transfer the sub-keys to just the intended recipients.

One solution would secure communication channel between the requester and the CA with encrypted connections (SSL), then use a challenge/response system between the requester and the CA. Here, the requester generates his sub-key for a shared subordinate and sends it to the CA. The CA then verifies the received sub-key against the set it stores for the intended subordinate. If the requester's sub-key matches one of the cached keys stored for the subordinate, the CA replies to the requester and transfers the remaining sub-keys for the subordinate back to the requester. Once the sub-keys are assembled, the requester can produce the key for the shared subordinate. The issue that remains is in securing the CA from attacks such that sub-keys are not stolen or replaced with frauds.

Other pragmatic issues surrounding the central authority center around its role with public parameters used in both the *direct* approaches and our HMAC-method.

For example, in the HMAC-method we store path addresses as arrays of integers. Thus, a 16-bit integer array could allow us to address up to $2^1$6 security classes. How we store and distribute this array is a pragmatic issue. For example, we may store the address arrays using a database, such as Oracle or PostgreSQL, that supports the variable array type, or we may store the addresses as a simple Java dictionary where the key is the security class name and the value is an address object or vector array. Similarly, as public parameters within the *direct* approaches grow beyond the $2^3$2 and $2^6$4 bit integer capacity of many computers, special large integer libraries are required to store and manipulate these numbers. Which libraries are used and how they implement large integer support can affect the performance of *direct* approaches.

Common to both approaches is the issue of the central authority's availability. If we consolidate the hierarchy, public parameters, and requests to one central authority, we must ensure that the CA is reliable and secure. If the central authority is prone to failure or susceptible to attack, we may want to replicate the CA across a network. If so, we must ensure consistency such that changes to the hierarchy are replicated in a timely manner.

5.3 Summary

In this chapter we raised some pragmatic issues surrounding the implementation and applicability of key management methods.

Implementing the Hwang-Yang approach from chapter three, we found that while their approach uses fewer prime numbers, the success of reducing the public parameters is dependent on solving the problem of selecting and assigning the appropriate composed prime sets to the proper leaf security classes. For larger and deeper hierarchies, judging the performance of a key management method becomes difficult without some form of metric or results collected from experimentation.

Further, we discussed and illustrated the importance of connectedness within the hierarchy. Our HMAC-method and Yang's method must cache sub-keys in order to address security classes with two or more direct principal classes. In some respects, this may be a limitation of the *indirect* approach. However with the example of the university student database, we demonstrated that by changing the structure of the hierarchy without affecting the nature of the relationships between the security classes, we produce a hierarchy that still maintains its purpose and intent, but can allow more than one key management method to apply.

In the remaining sections, we raised some pragmatic issues surrounding the keys, key updates, and central authority. If the intent of the access hierarchy is to provide support for encryption and decryption of information there are some considerations that must be taken into account when selecting the appropriate keys and the corresponding key management method. In updating the hierarchy, the nature of the organization being represented may have certain business operations that favour the use of one key management method and its key update flexibility over another. Finally, the central authority which is present in many methods is addressed. We discuss some factors affecting the design of the CA with respect to the cached sub-keys of the *indirect* approaches, reliability, security, and replication.

Chapter 6

Conclusions and Future Research

In this thesis, we studied the problem of key management within an access hierarchy. Our contribution to the key management problem is an *indirect* key derivation approach called the HMAC-method. It is called the HMAC-method, because it is based on hashed message authentication codes (HMACs) built from a single, fast, dedicated hash function (SHA-1). It is intended to provide a more efficient indirect key management method for large access hierarchies resembling tree structures. We are able to achieve better tree traversals using a technique we created called *path addressing*. Our *path addressing* scheme allows us to more efficiently calculate relationships between security classes, determine traversal paths, and improve the performance of the indirect key derivation method. We also presented our *cached key* update scheme which is meant to improve the indirect key derivation schemes by delaying key updates when changes to the structure of the access hierarchy are necessary, but the re-calculation and re-assignment of keys would either be costly or inconvenient.

For access hierarchies represented as a weakly/strongly connected directed acyclic graph (DAG), we suggested modifications to our *path addressing* and key derivation scheme which could allow our HMAC-method to be applied to these types of hierarchies; however our solution was not optimal.

Finally, we raised some pragmatic issues surrounding the implementation and applicability of key management methods. By implementing the Harn-Lin and Hwang-Yang approach, we found that the Hwang-Yang approach does produce fewer primes, but requires the solution to an optimization problem in order for small public parameters to result. We also demonstrated that in certain applications of key management, there may be changes to the hierarchy we can perform that allows us to apply more than one key management method without affecting the nature of the relationships. Other issues we raised discussed the type, size, and application of keys, key updates and the role and implementation of the central authority.

Future research into key management should include the design and implementation of a framework that allows the testing of key management methods. Much of the work in the research field thus far has been theoretical and results from design and implementation could be of benefit to understanding the performance and applicability of key management methods. Problems left open from this thesis are the optimal addressing of shared security classes within the HMAC-method and the optimal assignment of composed prime sets to leaf classes in the Hwang-Yang method.

Further, an interesting research direction into key management may be in combining *direct* and *indirect* approaches. An idea to explore would be dividing the hierarchy into sub-hierarchies each placed under a different CA. We could use the Harn-Lin *direct* approach to generate and assign each CA a master key, which we then use with *indirect* approaches to generate keys for classes within the sub-hierarchy. Thus, we move around large portions of the hierarchy using *direct* methods, but traverse the smaller pathed sub-hierarchies with *indirect* methods. This may yield a method of average performance, yet a system that may be applicable to distributed key management methods.

Bibliography

[1] Selim G. Akl and Stephen MacKinnon. New key generation algorithms for multilevel security. In *IEEE Symposium on Security and Privacy*, pages 72–79, Oakland, CA, April 1983. IEEE, IEEE.

[2] Selim G. Akl and Peter D. Taylor. Cryptographic solution to a multilevel security problem. In *Advances in Cryptology - Proceedings of Crypto '82*, pages 237–250, Santa Barbara, August 1982. Springer-Verlag.

[3] Selim G. Akl and Peter D. Taylor. Cryptographic solution to a problem of access control in a hierarchy. *ACM Trans. Comput. Syst.*, 1(3):239–248, 1983.

[4] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk. Cryptographic hash functions: A survey, 1995.

[5] Paulo S. L. M. Barreto. The hashing function lounge. Internet, December 2003. http://planeta.terra.com.br/ informatica/ paulobarreto/ hflounge.html.

[6] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. *Lecture Notes in Computer Science*, 1109:1–15, 1996.

[7] T. A. Berson. Differential cryptanalysis mod 2^{32} with applications to MD5. In R. A. Rueppel, editor, *Advances in Cryptology — Eurocrypt '92*, Berlin, 1992. Springer-Verlag.

[8] Eli Biham and Adi Shamir. Differential cryptanalysis of Snefru, Khafre, REDOC-II, LOKI and Lucifer (extended abstract). *Lecture Notes in Computer Science*, 576:156–171, 1991.

[9] Peter Camion and Jaques Patarin. The knapsack hash function proposed at crypto'89 can be broken. In D.W. Davies, editor, *Advances in Cryptology - EUROCRYPT '91: Workshop on the Theory and Application of Cryptographic Techniques*, volume 547 of *Lecture Notes in Computer Science*, pages 39–53, Brighton, UK, Jauary 1991. Springer-Verlag.

[10] Gerald C. Chick and Stafford E. Tavares. Flexible access control with master keys. In *Proceedings on Advances in cryptology*, pages 316–322. Springer-Verlag New York, Inc., 1989.

[11] Daemen, Govaerts, and Vandewalle. A framework for the design of one-way hash functions including cryptanalysis of damgard's one-way function based on a cellular automaton. In *ASIACRYPT: Advances in Cryptology – ASIACRYPT: International Conference on the Theory and Application of Cryptology.* LNCS, Springer-Verlag, 1991.

[12] Ivan Bjerre Damgård. A design principle for hash functions. In *Proceedings on Advances in cryptology*, pages 416–427. Springer-Verlag New York, Inc., 1989.

[13] Bert den Boer and Antoon Bosselaers. Collisions for the compression function of MD-5. *Lecture Notes in Computer Science*, 765:293–304, 1994.

[14] H. Dobbertin. Cryptanalysis of md5 compress, 1996.

[15] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. Ripemd-160: A strengthened version of ripemd. In *Proceedings of the Third International Workshop on Fast Software Encryption*, pages 71–82. Springer-Verlag, 1996.

[16] D. Ferraiolo and R. Kuhn. Role-based access controls. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[17] S Harari. Non-linear, non-commutative functions for data integrity. In *Proc. of the EUROCRYPT 84 workshop on Advances in cryptology: theory and application of cryptographic techniques*, pages 25–32. Springer-Verlag New York, Inc., 1985.

[18] L. Harn and H.-Y. Lin. A cryptographic key generation scheme for multilevel data security. *Computer Security*, 9(6):539–546, 1990.

[19] Min-Shiang Hwang. A new dynamic key generation scheme for access control in a hierarchy. *Nordic J. of Computing*, 6(4):363–371, 1999.

[20] Min-Shiang Hwang and Wei-Pang Yang. Controlling access in large partially ordered hierarchies using cryptographic keys. *J. Syst. Softw.*, 67(2):99–107, 2003.

[21] National Ins. Secure hash standard. Technical Report FIPS PUB 180, National Institute of Standards and Technology, April 1995.

[22] Lars R. Knudsen, Xuejia Lai, and Bart Preneel. Attacks on fast double block length hash functions. *Journal of Cryptology*, 11(1):59–72, January 1998.

[23] Lars Ramkilde Knudsen and Xuejia Lai. New attacks on all double block length hash functions of hash rate 1, including the parallel-DM. *Lecture Notes in Computer Science*, 950:410–418, 1995.

[24] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, third edition, March 2000.

[25] RSA Laboratories. Frequently asked questions about today's cryptography. http://www.rsasecurity.com/rsalabs/faq/, May 2000.

[26] Stephen J. MacKinnon, Peter D. Taylor, Henk Meijer, and Selim G. Akl. An optimal algorithm for assigning cryptographic keys to control access in a hierarchy. *IEEE Trans. Comput.*, 34(9):797–802, 1985.

[27] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., 1996.

[28] Ralph C. Merkle. One way hash functions and des. In *Proceedings on Advances in cryptology*, pages 428–446. Springer-Verlag New York, Inc., 1989.

[29] F. Mirza and S. Murphy. An observation on the key schedule of twofish, 1999.

[30] Bart Preneel, Rene Govaerts, and Joos Vandewalle. Differential cryptanalysis of hash functions based on block ciphers. In *Proceedings of the*

1st ACM conference on Computer and communications security, pages 183–188. ACM Press, 1993.

[31] Bart Preneel, Rene Govaerts, and Joos Vandewalle. Hash functions based on block ciphers: a synthetic approach. In *Advances in Cryptology - Proceedings of Crypto '93*, pages 368–378. Springer-Verlag Heidelberg, August 1993.

[32] Bart Preneel, Vincent Rijmen, and Antoon Bosselaers. Recent developments in the design of conventional cryptographic algorithms. *Lecture Notes in Computer Science*, 1528:105–130, 1998.

[33] Bart Prenel and Paul C. van Oorschot. Mdx-mac and building fast macs from hash functions. In *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*, pages 1–14. Springer-Verlag, 1995.

[34] Indrakshi Ray, Indrajit Ray, and Natu Narasimhamurthi. A cryptographic solution to implement access control in a hierarchy and more. In *Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 65–73. ACM Press, 2002.

[35] Michael Roe. Performance of block ciphers and hash functions - one year later. In Bart Preneel, editor, *Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994, Proceedings*, volume 1008 of *Lecture Notes in Computer Science*, pages 359–362. Springer, 1995.

[36] Ravinderpal S. Sandhu. Cryptographic implementation of a tree hierarchy for access control. *Inf. Process. Lett.*, 27(2):95–98, 1988.

[37] Bruce Schneier. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C.* John Wiley & Sons, Inc., 1995.

[38] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Performance comparison of the aes submissions. Technical report, CounterPane Inc., 1999.

[39] Douglas Stinson. *Cryptography: Theory and Practice,First Edition.* CRC/C&H, 1996.

[40] Tom Van Vleck. Multics software features. Webpage, December 2003.

[41] Hongjun Wu, Feng Bao, and Robert H. Deng. Cryptanalysis of some hash functions based on block ciphers and codes. *Informatica*, 26(3), 2002.

[42] Cungang Yang. *A Secure Object-Oriented Role-Based Access Control Model for Distributed Systems*. PhD thesis, University of Regina, Regina, Saskatchewan, August 2003.