Mahalingam Ramkumar

Symmetric Cryptographic Protocols

Springer

Symmetric Cryptographic Protocols

Mahalingam Ramkumar

Symmetric Cryptographic Protocols

Springer

Mahalingam Ramkumar
Mississippi State University
Mississippi State
Mississippi
USA

ISBN 978-3-319-07583-9 ISBN 978-3-319-07584-6 (ebook)
DOI 10.1007/978-3-319-07584-6
Springer Cham Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014941873

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To
my late mom,
Rajalakshmi.

Preface

Symmetric cryptography deals with:

1. the construction of efficient pseudo random functions (PRF), which are the building blocks of symmetric cryptography, and
2. symmetric cryptographic protocols, which are strategies to utilize the building blocks to solve some of our our day-to-day problems.

This book does not concern itself with the building blocks themselves; several well studied PRFs in the form of block ciphers and hash functions already exist. The focus of this book is instead on the (often under appreciated) range and utility of protocols and constructions that utilize symmetric PRFs.

Lack of widespread appreciation of the scope of symmetric cryptography has led to the unwarranted use of more expensive asymmetric cryptography in situations where symmetric cryptography is adequate. Perhaps, it is the sheer elegance of asymmetric primitives that instills in us the desire to honor them—by utilizing them even in situations where symmetric cryptography is adequate. This is one situation that this book aims to rectify.

The specific topics addressed in this book include:

1. various key distribution strategies for unicast, broadcast, and multicast security associations, and
2. strategies for constructing compact and efficient digests of dynamic databases.

A unified treatment of seemingly unrelated protocols is made possible by the fact that only three basic strategies, viz., hash chains, hash trees, and the surprising uniqueness of random subsets, are reused in a variety of different ways in different protocols.

Ultimately, the utility a cryptographic algorithm stems from the ability to leverage well-deserved assumptions regarding the properties of such algorithms; that we can virtually guarantee the existence of specific relationships between various inputs and outputs of the algorithm; for example, that the preimage of a cryptographic hash was chosen *before* the image was computed, and not vice-versa. Cryptographic algorithms are building blocks for the construction of application-specific cryptographic protocols, to enable enforcement of application-specific requirements, between various (application-specific) inputs and outputs.

By themselves, cryptographic protocols (unfortunately) do not provide the necessary (application-specific) *context* to the inputs and outputs. It is up to security protocols that utilize cryptographic protocols to do so. Consequently, practical security protocols will always need to make some *additional* noncryptographic assumptions regarding the environment in which the cryptographic protocol is executed, and the privacy of keys employed by the algorithms.

Almost every security issue we face in our day-to-day lives stems from the simple fact that many such noncryptographic assumptions turn out to be unjustified. For example, while the secure socket layer (SSL) is perfectly safe as a cryptographic protocol, when used as a security protocol, many vulnerabilities can crop up—like the recent Heart-bleed vulnerability, or the fact that SSL as a security protocol relies on the integrity of the public key infrastructure (PKI), which in turn relies on unverifiable assumptions regarding the integrity of PKI certificate authorities.

Perhaps the only practical recourse is to invest in an infrastructure to realize sufficiently trustworthy hardware modules. Such modules should be capable of guaranteeing a safe environment in which a wide variety of cryptographic protocols—necessary for a wide range of applications—can run unmolested. Only the well-deserved trust in the assumed properties of cryptographic algorithms, and the integrity of such hardware modules, can then be bootstrapped to realize security protocols—without the need to make unjustifiable assumptions like the integrity of software and hardware components in general purpose computers or the integrity of personnel/organizations with access to sensitive data processed in the computers. The versatility and low resource requirement for protocols based on symmetric PRFs make them very well suited for such an approach. Simple fixed functionality involving only PRF and logical operations, executed within the confines of deliberately resource limited modules, can be more readily verified to be free of malicious functionality. Almost every security protocol outlined in this book pays extra attention to additional constraints that may be imposed due to the fact that the security protocols will need to be executed inside a trustworthy (and severely resource limited) boundary.

Chapter 1 is a brief review of well-known properties of symmetric PRFs like hash functions and block ciphers. Chapter 2 outlines some useful constructions using PRFs that are reused throughout this book.

Chapter 3 presents key predistribution schemes for pairwise authentication—strategies that are traditionally considered as *nonscalable*. Two such schemes, the modified Leighton–Micali scheme (MLS) and the identity tickets (IT) schemes are, however, shown to be "scalable enough" for most practical applications. Chapter 4 outlines a strategy for employing such schemes in conjunction with trustworthy hardware modules with trivial functionality to secure the domain name system (DNS). This approach is compared with the current security protocol, DNSSEC, for securing DNS.

Chapters 5 and 6 present various scalable key predistribution schemes. Chapter 5 outlines many of the advantages of probabilistic schemes over deterministic schemes. Chapter 6 outlines three scalable schemes realized as extensions of nonscalable schemes discussed in Chap. 3. Chapter 7 highlights special considerations for protecting the integrity of secrets inside resource limited tamper-responsive boundaries. Such considerations are taken into account to reevaluate the strengths of various key distribution schemes, and the overhead associated with each approach.

Chapter 8 reviews strategies for multicast security associations like one-to-many associations (or broadcast security) and group security associations facilitated using broadcast encryption. While most broadcast encryption schemes employ a tree-like structure, "flat" schemes based on probabilistic key distribution have some compelling advantages. The utility of such schemes for practical deployments of publish–subscribe systems is also discussed in this chapter.

Chapter 9 presents a useful authenticated data structure, the ordered Merkle tree (OMT), and it's utility in assuring the integrity of a wide variety of dynamic databases maintained by untrusted entities. Two variations of the OMT, viz., the index ordered Merkle tree (IOMT), and the domain ordered Merkle tree (DOMT), are discussed. Simple algorithms intended to be executed by a trusted resource limited "verifier," to assure the integrity of a database maintained by an untrusted "prover," are presented.

Chapter 10 discusses a new *credential transaction model* as a specification of application-specific security protocols. For any system with a desired set of assurances, the strategy is to identity different roles for participants in the system, and a set of "permitted credential transactions" for each role. The permitted credential transactions are chosen to guarantee that no desired assurance is violated. Thus, as long as we can assure the integrity of credential transactions, we can assure the integrity of the entire system (that all desired assurances are met).

The credential transaction model permits the design of a universal trusted base—as a hypothetical specification for trusted *credential management modules* (CMM). Irrespective of the specific nature of the system, CMMs are entrusted with the task of assuring the integrity of credential transactions. Only assumptions regarding the integrity of PRFs, and the integrity of simple algorithms executed inside CMMs to verify the integrity of credential transactions, are bootstrapped by the security protocol (the transaction model) to realize all desired assurances. Such an approach eliminates the need for unjustifiable trust in complex hardware/software components, and personnel with the ability to influence the operation of such computers. The core functional components of CMMs include functionality described in Chap. 7 for unicast security associations, and functionality for maintaining OMTs, described in Chap. 9.

Starkville, MS
April 2014

Mahalingam Ramkumar

Contents

1 Introduction . . . 1

1.1 Cryptographic Algorithms . . . 1

1.1.1 Symmetric Cryptographic Algorithms . . . 2

1.1.2 Asymmetric Algorithms . . . 3

1.2 Using Cryptographic Algorithms . . . 4

1.2.1 Block Cipher Modes . . . 4

1.2.2 Hash Function . . . 5

1.2.3 Hashed Message Authentication Code . . . 6

1.2.4 Asymmetric Encryption and Signatures . . . 6

1.3 Cryptographic Protocols and Security Protocols . . . 7

1.3.1 Security Protocols . . . 7

1.3.2 Symmetric Protocols . . . 8

1.3.3 Symmetric Security Protocols . . . 9

2 Some Useful Constructions . . . 11

2.1 Hash Chains . . . 11

2.1.1 Hash Accumulator . . . 12

2.1.2 Hash Tree . . . 12

2.2 Random Subsets . . . 15

2.2.1 $\mathcal{S}_i \subset \mathcal{S}^n$ . . . 15

2.2.2 $(\mathcal{S}_i \cap \mathcal{S}_j) \subset \mathcal{S}^n$ . . . 16

3 Nonscalable Key Distribution Schemes . . . 19

3.1 Online KDC . . . 20

3.1.1 NS Protocol . . . 20

3.1.2 Leighton–Micali Protocol . . . 20

3.2 Offline KDC . . . 22

3.2.1 Basic KDS for Static Small-Scale Networks . . . 22

3.2.2 Key Distribution for Dynamic Networks . . . 23

3.3 MLS Key Distribution . . . 23

3.3.1 Identity Ticket (IT) Scheme . . . 25

3.4 Comparison . . . 26

3.4.1 MLS with Multiple KDCs ... 27
3.4.2 MLS Applications ... 28

4 MLS for Internet Security Protocols ... 31
4.1 Domain Name System ... 31
4.1.1 DNS Records ... 32
4.2 Securing DNS ... 34
4.2.1 Link-Security Approaches ... 35
4.3 DNSSEC ... 36
4.3.1 Authenticated Denial ... 36
4.3.2 DNS-Walk ... 37
4.4 MLS Based Alternative to DNSSEC ... 38
4.4.1 Extending Link-Security Approaches ... 38
4.4.2 Principle of TCB-DNS ... 39
4.4.3 Computing Link Secrets ... 42
4.5 The TCB-DNS Protocol ... 44
4.5.1 The Atomic Relay Algorithm ... 44
4.5.2 Preparation of TCB-DNS Master File ... 46
4.5.3 Verification of RRSets ... 47
4.5.4 Proof of Correctness ... 50
4.6 Practical Considerations ... 51
4.6.1 TCB-DNS vs. DNSSEC ... 52
4.6.2 Authenticated Denial ... 53
4.6.3 Overhead ... 55
4.6.4 Replay Attacks ... 56
4.6.5 DNSSEC with TSIG ... 56
4.6.6 NSEC3 Opt-Out ... 57
4.7 Alternative to IPSec ... 58
4.7.1 IPSec Operation ... 59
4.7.2 IPSec Issues ... 60
4.7.3 IPSec Alternative Leveraging TCB-DNS ... 60

5 Scalable Key Distribution Schemes ... 63
5.1 Certificates Based Schemes ... 63
5.2 Identity Based Schemes ... 64
5.2.1 Identity-Based Key Predistribution Schemes ... 64
5.2.2 Blom’s Schemes ... 65
5.3 Probabilistic KPSs (PKPS) ... 67
5.3.1 Allocation of Subsets ... 67
5.3.2 Random Preloaded Subsets ... 68
5.3.3 Hash-Chain KPS ... 68
5.3.4 Hashed Random Preloaded Subsets (HARPS) ... 70
5.4 (n, p)-Security of HARPS ... 71
5.4.1 Probability of Winning a Round ... 72
5.4.2 Optimization of Parameters ... 73

5.5 Deterministic Versus Probabilistic KPSs 75
5.5.1 KPS Complexity 77
5.5.2 Complexity Versus Desired Collusion Resistance n 78
5.5.3 Using External Resources 78
5.5.4 Low Complexity Hardware 79
5.5.5 Multiple KDCs and Renewal 79
5.5.6 Exploiting Multi-path Diversity 80
5.5.7 Conclusions 80

6 Scalable Extensions of Nonscalable Schemes 81
6.1 Parallel Basic KPS 81
6.2 Parallel Leighton–Micali Scheme (PLM) 82
6.3 (n, p)-Security of PBK and PLM 84
6.3.1 Optimal Choice of Parameters m and M 84
6.4 Subset Keys and Identity Tickets (SKIT) 85
6.4.1 (n, p)-Security of SKIT 86
6.4.2 Optimal Choice of Parameters 87
6.5 Comparison of KPSs 87
6.6 Beyond (n, p)-Security 90
6.6.1 (n, ϕ, p_a)-Security of RPS 91
6.6.2 (n, ϕ, p_a)-Security of PBK/PLM 93
6.6.3 (n, ϕ, p_a)-Security of SKIT 95
6.6.4 Addressing Message Injection Attacks 95
6.7 PLM for Sensor Networks 96
6.7.1 Classical Sensor Network Model 97
6.7.2 Assumptions 97
6.7.3 Key Distribution for Sensor Networks 98
6.7.4 Key Establishment 99
6.7.5 Performance and Overhead 100
6.8 Conclusions 101

7 Using PKPSs with Tamper-Responsive Modules 103
7.1 Core Principles 103
7.1.1 Active and Passive Shields 104
7.1.2 State Transitions 105
7.1.3 Single-Step Countermeasures 107
7.2 The DOWN Policy 107
7.2.1 DOWN-Enabled Modules 108
7.2.2 DOWN with Other Asymmetric Schemes 109
7.2.3 DOWN With ID-Based Schemes 111
7.2.4 DOWN Assurance and Complexity 113
7.2.5 DOWN with PKPSs 114
7.3 A Second Look at Key Predistribution Scheme (KPS) Complexity 114
7.3.1 Generic Device Model 115

7.4 Comparison of KPSs ... 117
7.4.1 Deployment Complexity ... 117
7.4.2 Complexity During Regular Operation ... 120
7.4.3 PLM ... 122
7.4.4 PBK ... 122
7.4.5 RPS and HARPS ... 123
7.5 KPS Algorithms ... 124
7.5.1 MLS ... 126
7.5.2 Scalable KPSs ... 126
7.6 Security Protocols Utilizing $f_{pw}()$ ... 127
7.6.1 Atomic Relay Protocols ... 128
7.6.2 Atomic Authentication Relay Algorithm ... 128
7.6.3 Atomic Path Secret Relay Algorithm ... 130
7.6.4 Accepting Relays ... 131
7.7 Conclusions ... 133

8 Broadcast Authentication and Broadcast Encryption ... 135
8.1 Certificates-Based Broadcast Authentication (BA) ... 135
8.1.1 One-Time Signatures (OTS) ... 135
8.1.2 Timed Efficient Stream Loss Tolerant Authentication (TESLA) ... 137
8.2 Identity-Based Broadcast Authentication (BA) Using Key Predistribution ... 138
8.2.1 Reducing Signature Size ... 139
8.2.2 Effect of Decrypt Only When Necessary (DOWN) Assurance ... 141
8.3 Broadcast Encryption ... 142
8.3.1 Tree-Based Broadcast Encryption (BE) Schemes ... 142
8.3.2 Broadcast Encryption (BE) Using Probabilistic Key Distribution ... 144
8.3.3 Broadcast Encryption (BE) by Sources Other Than Key Distribution Center (KDC) ... 145
8.4 Performance of Probabilistic Key Predistribution Scheme Broadcast Encryption (PKPS BE) ... 145
8.4.1 Performance Bounds ... 147
8.4.2 Over-Provisioning Keys ... 148
8.4.3 Hashed Random Preloaded Subsets (HARPS) vs. Random Preloaded Subsets (RPS) ... 149
8.5 Models for Broadcast Encryption (BE) ... 152
8.5.1 $G = N$ Models ... 152
8.5.2 $N >> G$ Models ... 153
8.5.3 Batch Sizes for External Sources ... 155
8.6 Application of Probabilistic Key Predistribution Scheme Broadcast Encryption (PKPS BE) in Publish–Subscribe Systems . 157
8.6.1 Desirable Features ... 157

8.6.2 PKPS-BE vs. T-BE for Pub–Sub Systems 158
8.6.3 Pub–Sub Operation 160

9 Authenticated Data Structures 163
9.1 Merkle Tree as an ADS 164
9.1.1 Merkle Tree Protocols 165
9.2 Ordered Merkle Tree 167
9.2.1 OMT Leaves 167
9.2.2 OMT Nodes 169
9.2.3 Verification and Update Protocols 170
9.2.4 Insertion of OMT Leaves 171
9.2.5 Reordering OMT Leaves 173
9.2.6 Index Ordered Merkle Tree 174
9.2.7 Domain Ordered Merkle Tree 175
9.2.8 Summary of OMT Properties 176
9.3 OMT Algorithms in Trusted Resource Limited Boundaries 177
9.3.1 Self-Certificates 178
9.3.2 Core OMT Functions 179
9.3.3 OMT Functions Exposed by **T** 180
9.3.4 Root Equivalence Certificates 183
9.3.5 Module **T** State 186
9.3.6 Using Module Functions 188
9.3.7 Context/Application Dependent Functions 189
9.4 Infrastructural Requirements 191

10 Universal Trusted Computing Bases 195
10.1 Practical Systems 195
10.1.1 Complexity and Ignorance 195
10.1.2 System Security Model 197
10.2 Trusted Platform Modules 198
10.2.1 Realizing a TCG Trusted Platform 198
10.2.2 Pitfalls of the TCG Approach 199
10.3 Trinc 200
10.3.1 Virtual Counters 202
10.4 Credential Management Modules 203
10.4.1 Credential Transaction Model 204
10.4.2 Consequential Transactions 207
10.4.3 Virtual Networks 207
10.4.4 VN State Changes 208
10.4.5 CMM State and VN State 209
10.4.6 Changing VN State 210
10.4.7 CMMs as ADS Constructors and Verifiers 211
10.5 CMM System Architecture 212
10.5.1 CMM Universe 213
10.5.2 Creation of Virtual Networks 213

10.5.3 Intra-VN Key Distribution 215
10.5.4 VN Links 215
10.6 Credential Transaction Model of Representative Systems 216
10.6.1 Credential Transaction Model for DNS 217
10.6.2 DNS Transactions 219
10.6.3 Transaction Models for Other Systems 221

11 Conclusions 223

References 227

Index 233

Acronyms

HMAC	Hashed Message Authentication Code
MAC	Message Authentication Code
KDC	Key Distribution Center
PKI	Public Key Infrastructure
OTS	One Time Signature
KDS	Key Distribution Scheme
KPS	Key Predistribution Scheme
PKPS	Probabilistic Key Predistribution Scheme
RPS	Random Preloaded Subsets
HARPS	Hashed Random Preloaded Subsets
BKP	Basic Key Predistribution
MLS	Modified Leighton–Micali Scheme
IT	Identity Tickets
PBK	Parallel Basic Key Predistribution Scheme
PLM	Parallel Leighton–Micali Scheme
SKIT	Subset Key and Identity Tickets
ADS	Authenticated Data Structures
OMT	Ordered Merkle Tree
IOMT	Index Ordered Merkle Tree
DOMT	Domain Ordered Merkle Tree
VN	Virtual Network
DNS	Domain Name System
BGP	Border Gateway Protocol
AS	Autonomous System

Chapter 1
Introduction

Cryptography (literally, *secret writing*) has come a long way from it's original scope—viz., *encryption* of messages. Modern cryptography offers several sophisticated tools and strategies to use the tools to achieve well defined security goals. Specifically, modern cryptography deals with:

1. Construction of cryptographic tools
2. Analysis of the strengths of cryptographic tools
3. Protocols to utilize the tools

The tools in the cryptographic tool-box take the form of deterministic *cryptographic algorithms*. Well known examples of cryptographic algorithms [1] include encryption/decryption algorithms like DES, AES, RSA, hashing algorithms like MD5SUM, SHA-1, SHA-2, digital signature algorithms like DSA, RSA, etc. Strategies to leverage the algorithms to achieve specific security goals are *cryptographic protocols*, or more generally, *security protocols*.

1.1 Cryptographic Algorithms

Cryptographic algorithms can be classified into two broad categories:

1. *Symmetric* cryptographic algorithms are composed of repetitive simple operations on small bit-strings—operations like bit-wise logical operations, addition, and rotation/permutation of bits. The term "symmetric" is due to the fact that traditionally, most[1] well known algorithms in this category were used for encryption and decryption, where both the encryption and decryption operation utilized the *same* key.
2. *Asymmetric* cryptographic algorithms are mainly composed of modular multiplication operations involving large numbers. The term "asymmetric" is due to the

[1] Hashing algorithms, which also fall under this category, do not require keys.

M. Ramkumar, *Symmetric Cryptographic Protocols,* DOI 10.1007/978-3-319-07584-6_1,

fact that such algorithms use two different keys—a private key which is intended to be a secret, and a public key which is intended to be made public.

1.1.1 *Symmetric Cryptographic Algorithms*

Commonly used symmetric algorithms include block-ciphers and hash functions.

1.1.1.1 Block Ciphers

Block cipher algorithms (Chap. 7 in [2]) specify an encryption algorithm $E()$ and a decryption algorithm $D()$,

$$C = E(P, K) \text{ and } P = D(C, K), \tag{1.1}$$

where P is the "plain-text" message encrypted by algorithm $E()$ using a key K to yield the "cipher-text" C. Using the same key K, algorithm $D()$ decrypts the cipher text C to obtain the plain text P. In Eq (1.1)

1. P and C are bit-strings of length b, where b is the block size.
2. K is a bit-string of length k, where k is the key size.

For example, in the erstwhile data encryption standard (DES) algorithm, P and C are 64-bit strings ($b = 64$), and K is a 56-bit string ($k = 56$). The more recent Advanced Encryption Standard (AES) prescribes a block cipher with three key lengths—128, 192, and 256 bits. In AES-128, P, C and K are all 128 bits long ($b = k = 128$); in AES-192, $b = 128$ and $k = 192$; in AES-256, $b = 128$ and $k = 256$.

For a (b, k) block cipher each of the 2^k possible keys define a table with 2^b rows and two columns. The first column can be seen as consisting of all possible 2^b plain-text blocks $\mathbf{0}^b \cdots \mathbf{1}^b$ arranged in an order; the second column is a fixed permutation of the first column. A block-cipher with key length k thus defines 2^k such fixed permutations—one corresponding to each possible key.

Some of the important properties of modern block ciphers are as follows:

1. Given $C = E(P, K)$ it is impractical to determine P without the knowledge of K.
2. Given any number (say, n) of plain-text-cipher-text pairs (P_i, C_i) where $C_i = E(P_i, K)$ for $i = 1 \cdots n$, the easiest way to find the key K is still through brute-force search of all 2^k possible keys.
3. A cipher is regarded as a strong cipher if the 2^k brute-force search complexity is deemed impractical.

1.1.1.2 Cryptographic Hashing

In a cryptographic hashing algorithm $H()$,

$$O = H(S, B) \tag{1.2}$$

1. O and S are bit-strings of length u (the digest size).
2. B is a bit-string of size l (the block-size).

Specifically, B is a block of bits which is combined with the previous state S to yield the digest (or next state) O. In MD5SUM [3] O and S are 128-bit strings ($u = 128$), and B is a 512-bit string ($l = 512$). The secure hash algorithm (SHA) standard [4] prescribes SHA-1 with $u = 160$ and $l = 512$, and SHA-2 with $u = 256$ and $l = 512$. The recently standardized SHA-3 algorithm supports multiple digest sizes ($u = 224/256/384/512$) and multiple block sizes ($l = 1152/1088/832/576$).

It is convenient to represent the two inputs to the hash algorithm $H()$ as the preimage $X = S \parallel B$. Some of the important properties of a cryptographic hash algorithm are as follows:

1. Given $O = H(X)$ it is impractical to determine a second preimage $X' \neq X$ satisfying $O = H(X')$. For a hash function with u-bit digest the easiest way to determine a second preimage is by brute-forcing—choosing random preimages and then verifying if the corresponding digest is the desired value O. One can expect a brute-force attack to require on the order of 2^u attempts.
2. It is impractical to determine two different preimages X and $X' \neq X$ such that $H(X) = H(X')$. For a hash function with u-bit digest the easiest way to determine two colliding preimages is by brute-forcing (choosing two random preimages and then verifying if they yield the same digest). One can expect a brute-force attack to require on the order of $2^{u/2}$ attempts.

1.1.2 Asymmetric Algorithms

An asymmetric cryptographic algorithm [1] can be seen as consisting of several subalgorithms

1. $K_u = F_{gen}(K_r)$: an algorithm to compute the public-key K_u corresponding to the private key K_r. Typically, the private key K_r is chosen randomly. Some algorithms however may have some restrictions on this choice (for example, only prime numbers, only points on a elliptic-curve over which the algorithm operates, etc.).
2. Algorithms $C = F_{enc}(P, K_u)$ and $P = F_{dec}(C, K_r)$ for encryption of a plain text C using the public key K_u to derive the cipher-text C, and for decryption of the cipher text C using the corresponding private key K_r;
3. Algorithms $S = F_{sign}(M, K_r)$ and $\{0, 1\}^1 = F_{ver}(M, S, K_u)$ to derive a value S (a digital signature) that is a function of a value M and the private key K_r,

and to verify that the value M and the signature S are indeed consistent with the corresponding public key K_u.

Some of the important properties of a good asymmetric algorithm are as follows:

1. Functions $F_{gen}()$, $F_{enc}()$, $F_{dec}()$, F_{sign}, and $F_{ver}()$ are *not* prohibitively expensive.
2. It is computationally infeasible to determine the private key K_r given the corresponding public key K_u;
3. Given $C = F_{enc}(P, K_u)$, it is computationally infeasible to determine P except with the knowledge of the private key K_r.
4. Given $S = F_{sign}(M, K_r)$ it is computationally infeasible to determine $M' \neq M$ satisfying $F_{ver}(M', S, K_u) = 1$; similarly, given a random M, without the knowledge of K_r, it is infeasible to determine S satisfying $F_{ver}(M, S, K_u) = 1$.

1.2 Using Cryptographic Algorithms

Currently, block-ciphers are often used for encryption of bulk-data using one of several *modes of operation*[2]. The encryption key is typically conveyed to the intended receiver using an asymmetric encryption technique.

Hash algorithms like MD5SUM, SHA etc., are commonly used as building blocks for hash functions. More specifically, the Merkle–Damgard construction [6] is used to realize a hash function by repeated application of a hashing algorithm. Hash functions, in turn, are commonly used for two purposes: i) computing a digest for bulk-data; and ii) jointly computing a digest—a hashed message authentication code (HMAC) [7]—of the bulk-data and a secret key. In the former scenario, the digest may signed using an asymmetric signature scheme for authentication of bulk-data.

1.2.1 Block Cipher Modes

For encryption of bulk-data using a block cipher [5], the data is broken into n chunks of plain-text blocks $P_1 \cdots P_n$—each block with b bits (if a b-bit block-cipher $E_K()/D_K()$ used). The output, after encryption of bulk-data, takes the form of an initial value (IV) chosen by the sender, and n cipher text blocks $C_1 \cdots C_n$—one corresponding to each plain text block.

If cipher block chaining (CBC) mode is used

$$C_j = E_K(C_{j-1} \oplus P_j), j = 1 \cdots n, \text{ and } C_0 = \text{IV} \tag{1.3}$$

[2] Popular modes include CBC (cipher block chaining), CFB (cipher feed back), OFB (output feed back), and CTR (counter).

where K is the key shared between the sender and the receiver. On receipt of $C_0 = \mathrm{IV}$ and cipher text blocks $C_1 \cdots C_n$ the receiver computes

$$P_j = D_K(C_j) \oplus C_{j-1}, j = 1 \cdots n. \tag{1.4}$$

If cipher feed back (CFB) mode is used

$$C_j = E_K(C_{j-1}) \oplus P_j, j = 1 \cdots n, \text{ and } C_0 = \mathrm{IV} \tag{1.5}$$

and the receiver computes

$$P_j = E_K(C_{j-1}) \oplus C_j, j = 1 \cdots n. \tag{1.6}$$

If output feed back (OFB) mode is used

$$C_j = O_j \oplus P_j, j = 1 \cdots n, \text{ where } O_0 = \mathrm{IV} \text{ and } \mathrm{O_j = E_K(O_{j-1})}, \tag{1.7}$$

and

$$P_j = O_j \oplus C_j. \tag{1.8}$$

Finally, in the CTR (counter) mode,

$$C_j = P_j \oplus E_K(\mathrm{IV + j}), \mathrm{j = 1 \cdots n}, \tag{1.9}$$

and

$$P_j = C_j \oplus E_K(\mathrm{IV + j}) \tag{1.10}$$

1.2.2 Hash Function

A hash function $d = \mathcal{H}(M)$ takes bulk-data M as input and outputs a u-bit digest d. The Merkle–Damgard [6] construction for realizing the hash function[3] using a hashing algorithm $O = H(S, B)$ (with block-size l and digest size u), is as follows.

The L bit bulk-data M (for example, a file) is padded to length $L + l_p$, where $1 \leq l_p \leq l$, to ensure that $L + l_p = ln - 64$. The length L is represented as a 64-bit value and appended at the end to result in ln bits. The ln bit data is segmented into n l-bit chunks $B_1 \cdots B_n$.

A standard hash algorithm is associated with a fixed initial value d_0 (u bit value, same as the digest size). Let

$$d_i = H(d_{i-1}, B_i), i = 1 \cdots n. \tag{1.11}$$

[3] When the MD5SUM hash algorithm is used, the resulting hash function is the "MD5SUM hash function."

Now, the value $d = d_n$ is regarded as the digest for L-bit data M.

The digest is also referred to as the *commitment* for the bulk-data (say, file) M. For example, the commitment d for a file may be stored in a trusted location while the file M itself is stored in an untrusted location. If a file M' fetched from the untrusted location satisfies $d = \mathcal{H}(M')$ the verifier is convinced that $M' = M$ (or, the file was not modified).

1.2.3 Hashed Message Authentication Code

According to the hashed message authentication code (HMAC) standard [8], a hash function $\mathcal{H}()$ can be used to compute the HMAC μ for bulk-data M, and a secret key K, as

$$\mu = \mathrm{HMAC(M, K)} = \mathcal{H}((\mathrm{K} \oplus \mathrm{p_o}) \parallel \mathcal{H}((\mathrm{K} \oplus \mathrm{p_i}) \parallel \mathrm{M})), \tag{1.12}$$

where $p_o = 0x5c5c\ldots5c5c$ is the standard "outer pad" and $p_i = 0x3636\ldots3636$ is the standard "inner pad."

When bulk-data M is exchanged over an open insecure channel (where it may be modified by attackers), the key K shared by the sender and receiver is used to compute the HMAC which is sent along with M. The receiver verifies the integrity of the HMAC, and on successful verification, is convinced that:

1. The HMAC was computed by a process privy to K.
2. M was not modified in transit.

1.2.4 Asymmetric Encryption and Signatures

In practice, a prerequisite for utilizing an asymmetric key pair—say key pair (K_r^A, K_u^A) generated by an entity A (using algorithm $F_{gen}()$)—is the existence of a trusted certificate authority that certifies a binding between the owner A and the public key K_u^A.

As asymmetric algorithms are 2 to 3 orders of magnitude more expensive that symmetric algorithms, they are used sparingly. They are used for two main purposes:

1. For conveying a symmetric key (which is used for bulk-encryption using a symmetric block-cipher)
2. For signing the digest of bulk-data M (where the digest is computed using a standard hashing algorithm)

Any entity can send encrypted data $C_0 \cdots C_n$ to the intended receiver A, along with the encryption key K. Specifically, the encryption key K is sent as

$$K' = F_{enc}(K, K_u^A), \tag{1.13}$$

which can be decrypted only by the intended receiver A as

$$K = F_{dec}(K', K_r^A). \tag{1.14}$$

To sign document (or bulk-data) M, an entity A i) computes the digest $d =$ hash(M), and ii) computes the signature as

$$S = F_{sign}(d, K_r^A) \tag{1.15}$$

Any entity receiving M and the signature S can compute $d =$ hash(M) and verify that

$$F_{ver}(d, S, K_u^A) = 1. \tag{1.16}$$

1.3 Cryptographic Protocols and Security Protocols

Broadly, a cryptographic protocol is a construction that utilizes one or more cryptographic algorithms. The purpose of such a protocol is the ability to make specific assertions regarding specific inputs and outputs to the protocol, based on the assumptions regarding the properties of algorithms.

As an example, consider the following two-step protocol:

1. Accept inputs M, K and m
2. Verify $m = H(M \parallel K)$

On successful verification, it is reasonable for the verifier to assert that "the input m was created exactly in the same manner—by computing $m = H(M \parallel K)$—by the process that provided the inputs M, K, and m."

Note that even while there may exist numerous other values $M' \neq M$ and $K' \neq K$ satisfying $m = H(M' \parallel K')$, it is safe to assert that such values were *not* used to generate m—as such a possibility would amount to the discovery of a second preimage $M' \parallel K'$ for the digest m—which is assumed to be impractical.

Thus, based on a specific assumption regarding the properties of the algorithm $H()$—that of second preimage resistance—one can assert the existence of a specific relationship between the values m, K, and M.

1.3.1 Security Protocols

By itself, a cryptographic protocol is not very useful, as it does not provide meaningful contexts to the inputs and outputs. This is where a security protocol comes in. A security protocol provides a context to a cryptographic protocol through *additional, noncryptographic* assumptions. For example, a security protocol that utilizes the two-step cryptographic protocol above may make two assertions, viz.,

1. The input m was computed as $m = h(M \parallel K)$.
2. m was computed by an entity (say) “Bob.”

The second assertion may rest on an additional noncryptographic assumption that (apart from the verifying process) only “entity Bob” has access to the key K. As the value K was should have been used to compute $m = h(M \parallel K)$, only Bob could have computed m.

Obviously, the strength of the assertions made by a security protocol rest on the correctness of *both* cryptographic and noncryptographic assumptions. It should come as no surprise that almost *every practical security issue we face in our day-to-day lives stems from the failure of noncryptographic assumptions*. In other words, security breaches rarely (if ever) result from an attacker breaking a cipher, or determining a collision in a hash function. Instead, they result from:

1. Improper storage of keys
2. Choice of weak keys
3. Bugs in software implementations of cryptographic protocols
4. Bugs in the environment in which the cryptographic protocols are executed—which may provide unknown avenues for attackers to gain improper access, modify inputs and outputs, or even modify the algorithm.

For the example considered above, for the verifier to assert that it was indeed Bob that originated the message m, some specific noncryptographic assumptions are as follows:

1. The process adopted by the verifier and Bob to establish a common secret K did not leak any information regarding the key to unauthorized parties.
2. The computing platform employed by Bob is completely under the control of Bob, and does not leak information regarding the key to unauthorized parties. Some of the not-so-obvious implications of this assumption are that every piece of hardware in the platform, and every bit of software executed by the platform is bug-free—as an accidental or malicious bug in hardware or software could possibly be exploited by an attacker to take control of Bob’s platform.
3. The computing platform employed by the verifier is bug-free. Note that a bug in the verification process in the verifier’s platform can result in a wrongful claim that the verification was successful (while it was not).

1.3.2 Symmetric Protocols

In the rest of this book we restrict ourselves to symmetric protocols—both cryptographic protocols and more general security protocols.

Central to symmetric cryptographic protocols are well-founded assumptions regarding specific properties of symmetric cryptographic algorithms. For purposes of this book, it is not necessary to differentiate between block-ciphers and hash functions, as both can be generalized as a pseudo-random function (PRF) of the form

$$Y = h(X) \tag{1.17}$$

where X is the concatenation of two inputs—possibly of different sizes u and l (for hashing algorithms), or b and k (for block-ciphers), and Y is the output, with the same size as one of the two inputs.

Without any loss of generality, in the rest of this book we shall assume that the input to $h()$ is a $(u + l)$-bit preimage, and the output is a u-bit digest. In other words,

$$h : \{0, 1\}^u \times \{0, 1\}^l \rightarrow \{0, 1\}^u, \tag{1.18}$$

is a deterministic mapping from a $(u + l)$-length bit-string to a length-u bit-string.

Some of the assumed properties of the PRF $Y = h(X)$ are as follows:

1. Given $Y = h(X)$, changing even a bit of the input will produce an output Y' that is no way related to Y.
2. Given a preimage X there is no way to make any reliable prediction regarding *any* bit of Y (without actually using $h()$). In other words, before using $h()$, every bit of the digest Y is equally likely to be 0 or 1.
3. Given a digest Y, the easiest way to determine a preimage X' that satisfies $Y = h(X')$ should be by brute-force search for a suitable candidate for X'. As any of the 2^u digests are equally likely, one can expect to search for (on an average) 2^u preimages before *accidentally* discovering an X' that yields the desired $Y = h(X')$.
4. The fastest way to determine any two preimages X and X' satisfying $h(X) = h(X')$ (the digests collide) should have brute-force search complexity of $2^{u/2}$.

In this book, in many of the symmetric cryptographic protocols constructed using $h()$, the size of the input to $h()$ may be indicated as an r-bit value, where $r \leq u + l$-bits. If $r < u + l$, it is to be assumed that some standard padding mechanism is used to pad the input with additional $(u + l - r)$ bits. Likewise, in some instances the input to $h()$ may be larger than $u + l$ bits. In such scenarios it is to be assumed that multiple applications of $h()$ will be used. To reduce the complexity of notations the pad and multiple applications of $h()$ will not be explicitly shown.

1.3.3 Symmetric Security Protocols

In the context of security protocols, it is obviously essential to minimize the scope of intangible (noncryptographic) assumptions. Furthermore, it is also necessary to provide an unambiguous description of such assumptions. Whenever such assumptions are necessary in this book, they simply take the form of “a trusted boundary within which cryptographic algorithms are executed.”

The main advantages of symmetric cryptographic primitives (compared to asymmetric primitives) stem from their simplicity and versatility. A symmetric PRF $h()$

typically demands substantially lower (by 2 to 3 orders of magnitude) computation and memory requirements compared a typical asymmetric algorithm. Not withstanding it's simplicity, a PRF $h()$ can be used in a wide variety of ways to realize an extensive range of useful cryptographic protocols.

The simplicity of symmetric cryptographic protocols—composed of simple logical operations and PRF operations—has the indirect effect of *strengthening* the non-cryptographic assumptions required for practical security protocols. Specifically, the assumption of "a trusted boundary within which cryptographic protocols/algorithms are executed" is indeed more justifiable if we deliberately constrain the complexity of operations performed inside the trusted boundary. Low complexity inside a trusted boundary implies lower scope to hide malicious or unintended functionality.

Some of the protocols described in this book aim to be less expensive alternatives to asymmetric protocols. Currently, asymmetric protocols are used mainly for:

1. Establishment of symmetric session secrets (for one-to-one authentication and/or privacy)
2. For signing message digests (for one-to-many, or broadcast authentication)

Consequently, the alternative protocols outlined in this book take the form of *key distribution schemes* for pairwise and broadcast authentication.

Other protocols described in this book include:

1. *Broadcast encryption* strategies for communicating a secret to all but a select few (explicitly excluded) receivers
2. A broad class of useful security protocols based on binary hash trees.

Chapter 2
Some Useful Constructions

Almost every protocol described in this book takes advantage of some, or all of the following three basic strategies of utilizing a PRF $h()$: (a) hash chains; (b) hash trees, which are also referred to as binary hash chains; and (c) the uniqueness of random subsets of large sets generated using PRF $h()$.

2.1 Hash Chains

A hash chain is [9] constructed through successive applications of the PRF $h()$ on a bit-string X_0.

For example, let

$$X_1 = h(X_0), X_2 = h(X_1), X_3 = h(X_2) \dots , X_n = h(X_{n-1}). \tag{2.1}$$

Such n successive applications that result in the value X_n can be conveniently represented by the notation

$$X_n = h^n(X_0). \tag{2.2}$$

From the properties of a PRF $h()$ it follows that given a value X_i from a hash chain, it is

1. Easy to compute X_j, if $j \geq i$, through $j - i$ applications of $h()$
2. Infeasible to compute X_j, if $j < i$

Furthermore, given two values U and V satisfying $h^x(U) = V$, even while there exists numerous values U' satisfying $h^x(U') = V$, it is safe to conclude that V was indeed generated by repeatedly hashing U.

Even while the input and output to $h()$ appear to have the same size (u-bits), it should be assumed that the input is padded with l fixed pad-bits to size $l + u$.

M. Ramkumar, *Symmetric Cryptographic Protocols,* DOI 10.1007/978-3-319-07584-6_2,

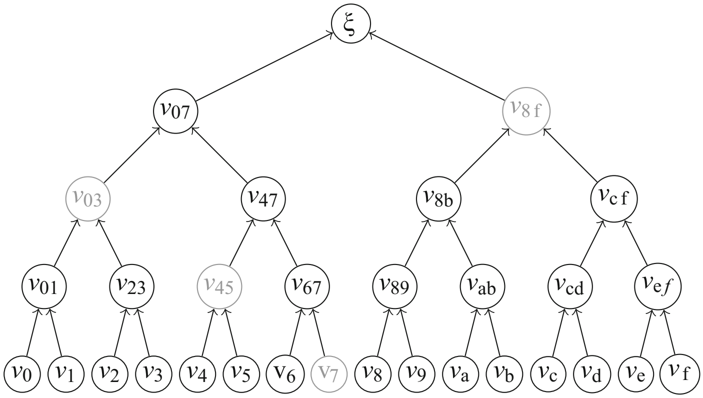


Fig. 2.1 A binary hash tree

2.1.1 Hash Accumulator

Given a list of values $v_1 \cdots v_n$, a hash accumulator computes an accumulated hash α as follows.

$$\begin{aligned}
\alpha_2 &= h(v_1 \parallel v_2)\\
\alpha_3 &= h(\alpha_2 \parallel v_3)\\
\alpha_4 &= h(\alpha_3 \parallel v_4)\\
&\vdots\\
\alpha &= h(\alpha_{n-1} \parallel v_n)
\end{aligned} \tag{2.3}$$

Each step in the accumulation of the hash is also referred to as hash-extension. For example, in the operation $h(\alpha_2 \parallel v_3)$, "$\alpha_2$ is hash-extended with v_3."

The accumulated hash can be seen as a commitment to all values $v_1 \cdots v_n$. Specifically, even while there are numerous possible sets of values which yield the same accumulated value α, given α and the values $v_1 \cdots v_n$, one can conclude that α was indeed computed by accumulating values $v_1 \cdots v_n$.

2.1.2 Hash Tree

A more common strategy for accumulating a set of values $v_1 \cdots v_n$ into a single commitment α is by arranging values $v_1 \cdots v_n$ as leaves of a binary hash tree. The binary hash tree is more commonly referred to as a Merkle tree [10]. For simplicity, we shall assume that n is a power of 2.

Figure 2.1 depicts a Merkle tree with $N = 16$ leaves $v_0 \cdots v_f$. A binary tree with N leaf-nodes has $2N - 1$ nodes spread over $\log_2 N + 1$ levels—levels $0 \cdots L = \log_2 N$.

At level 0 are the N leaf-nodes $v_0 \cdots v_f$. At level 1 are $N/2$ nodes, each obtained by hashing together two adjacent nodes in level 0. In the figure, the eight nodes $v_{01}, v_{23}, \ldots v_{ef}$ in level 1 are obtained as

$$\begin{aligned} v_{01} &= h(v_0 \parallel v_1) \\ v_{23} &= h(v_2 \parallel v_3) \\ &= \vdots \\ v_{ef} &= h(v_e \parallel v_f) \end{aligned} \tag{2.4}$$

Similarly, the four nodes at level 2 are each obtained by hashing together two adjacent nodes in level 1. For example,

$$v_{03} = h(v_{01} \parallel v_{23}). \tag{2.5}$$

Note that a tree with $N = 2^L$ leaves at level 0 has 2^{L-i} nodes in level i, where $i = 0 \cdots L$. The total number of nodes in the tree is thus

$$\sum_{i=0}^{L} 2^{L-i} = 2^{L+1} - 1 = 2N - 1 \tag{2.6}$$

The lone node at the top of the (inverted) tree is the *root* of the tree. The root is a compact commitment to all nodes.

Every node has a sibling. v_6 and v_7 are siblings (with a common parent v_{67}); likewise, v_{8b} and v_{cf} are siblings (with a common parent v_{8f}). Corresponding to any node at level 0 are $L - 1$ direct ancestors. For example, the ancestors of node v_6 are $v_{67}, v_{47}, v_{07,}$ and α — one in each level $1 \cdots L$. The root α is a common ancestor for all nodes.

Corresponding to every node in level 0 are L *complementary* nodes—one in each level $0 \cdots L - 1$. The $L = 4$ complementary nodes of v_6 are $v_7, v_{45}, v_{03,}$ and v_{8f}. Note that the complementary nodes of any node includes

1. The sibling of the node
2. The siblings of all ancestors

Together, the nodes complementary to v_6 can be interpreted as a commitment to all nodes except v_6. v_{8f} is a commitment to eight nodes $v_8 \cdots v_f$; v_{03} is a commitment to four nodes $v_0 \cdots v_3$; v_{45} is a commitment to v_4 and v_5; and v_7 is a commitment to itself.

Any node in the tree (except the root) is either a *right* child or a *left* child of its parent. For example v_7 is a right child of its parent v_{67}; v_{45} is a left child of its parent v_{47}. Thus, every node can be associated with an additional bit—say 0 if it a right-child and 1 if it is a left-child left.

The L complementary nodes of v_6 along with their orientations, viz.,

$$\{(v_7, 0), (v_{45}, 1), (v_{03}, 1), (v_{8f}, 0)\},$$

readily provide step by step *instructions* for mapping leaf v_6 to the root, through a sequence of L PRF operations. For example, following the instructions, we can compute the root α starting from v_6 as

$$\begin{array}{ll} v_{67} = h(v_6 \parallel v_7) & v_{47} = h(v_{45} \parallel v_{67}) \\ v_{07} = h(v_{03} \parallel v_{47}) & \alpha = h(v_{07} \parallel v_{8f}) \end{array}$$

Note that the orientation bit specifies the ordering of two nodes before hashing them together to compute the parent node. As v_7 is a right-child (orientation 0) it has to be placed to the right of v_6 before hashing. Similarly, as v_{45} is a left-child, it has to be placed to the left before hashing.

Also note that the four orientation bits $0, 1, 1, 0$ of the complementary nodes of v_6 (v_7, v_{45}, v_{03} and $v_{8\ f}$ respectively) can be readily obtained from the bits used to represent the index of v_6 in binary format (index $6 = 0110_b$). As a second example, the complementary nodes of v_8 are

1. Sibling v_9 which is a right-child (orientation 0)
2. Sibling v_{89} of ancestor v_{ab} (orientation 0)
3. Sibling v_{cf} of ancestor v_{8b} (orientation 0)
4. Sibling v_{07} of ancestor v_{8f} (orientation 1)

Once again note that the binary representation of the index $8 = 1000_b$ provides the necessary orientation bits (read from LSB to MSB).

Thus, given any leaf-node v at level 0, it's index i (where $0 \leq i \leq N-1$), and the set of its L complementary nodes $\mathbf{c} = \{c_0 \cdots c_{L-1}\}$, we can define a simple function

$$\alpha = f_{bt}(v, i, \mathbf{c}) \tag{2.7}$$

that maps v to the root α. The function $f_{bt}()$ can be algorithmically represented as follows:

```
$\alpha = f_{bt}(v, \mathrm{i}, \{c_0, c_1, \ldots c_{L-1}\})\{$
  FOR ($j = 0 \cdots L - 1$)
    IF (i IS EVEN) $v \leftarrow h(v \parallel c_j)$;
    ELSE $v \leftarrow h(c_j \parallel v)$;
    $i \leftarrow i >> 1$; //right shift by one bit
  RETURN $v$;
}
```

As the PRF $h()$ is preimage resistant, it is infeasible to determine alternate values $\tilde{v} \neq v$, and $\tilde{\mathbf{c}} \neq \mathbf{c}$ that will satisfy $f_{bt}(v, \tilde{\mathbf{c}}) = \alpha$.

In applications that employ Merkle trees the root α of the tree is stored in a trusted location. The other $N-2$ values can be stored in an untrusted location. If values v, $\mathbf{c}$ received from an untrusted source satisfy $f_{bt}(v, \tilde{\mathbf{c}}) = \alpha$, the verifier is convinced of the integrity of such values. More specifically, the verifier is convinced that values v and $\mathbf{c}$ were indeed used in the construction of the tree with root α.

2.2 Random Subsets

Several symmetric cryptographic protocols of interest to us in this book are based on the idea of allocation of random subsets of keys [11] from the pool of keys.

Consider a key-pool with P keys $K_1 \cdots K_P$. Let $\mathcal{S}_1 \cdots \mathcal{S}_N$ represent subsets of $k < P$ keys chosen randomly from the key pool.

Let $k/P = a < 1$. One strategy to choose subset of k keys on an average from a pool of P keys is by picking each key from the pool with probability $a = k/P$. Alternately, if it is desired that each subset should have exactly k keys, the pool of P keys may be divided into k sub-pools, each with P/k keys; from each of the P/k pools one key is picked randomly.

When the key pool and subsets are generated using a PRF $h()$ the generator could start with a single master key μ to generate the pool keys as

$$K_i = h(\mu \parallel i), q \le i \le P. \tag{2.8}$$

Any subset may be associated with a seed which determines the indexes of the keys chosen to be a part of the subset. For example, for a subset associated with a seed X, a random stream of bits generated from repeated application of $h()$ on X, for example, $X_1, X_2, \ldots$ generated as

$$X_1 = h(X), X_2 = h(X_2) \cdots \tag{2.9}$$

can be used to identify the indexes to be assigned to the subset.

Assume that n subsets are picked randomly. Let us represent by $\mathcal{S}^n$ the super set of n such subsets. In addition, we randomly choose two other subsets $\mathcal{S}_i$ and $\mathcal{S}_j$. Now, two specific questions of interest to us are

1. What is the probability p that *all* keys contained in a subset $\mathcal{S}_i$ is contained in $\mathcal{S}^n$?
   a) For a given n, p, what is the minimum value of the pool size P?
2. What is the probability that *all keys in the intersection* of $\mathcal{S}_i$ and $\mathcal{S}_j$ is contained in $\mathcal{S}^n$?
   a) For a given n, p, what is the minimum value of the pool size P?
   b) For a given n, p, what is the minimum value of the subset size k?

2.2.1 $\mathcal{S}_i \subset \mathcal{S}^n$

Consider a specific key in the subset $\mathcal{S}_i$. The probability that the same key is found in specific subset that was chosen to create $\mathcal{S}^n$ is a. The probability that the specific key is *not* found in any of the subsets in $\mathcal{S}^n$ is

$$\epsilon = (1 - a)^n \tag{2.10}$$

Thus, the probability that a specific key in the subset $\mathcal{S}_i$ is included in the union of n subsets $(1-\epsilon)$. Consequently, the probability that all k keys in $\mathcal{S}_i$ are included in the union of n subsets is

$$p(n) = (1-\epsilon)^k = (1-(1-a)^n)^k \approx (1-e^{-an})^{Pa} \tag{2.11}$$

Obviously, p increases with n. It is often of interest to us to achieve a target $p(n)$ using the least amount of keys. To derive an expression for P, Eq. (2.11) can be rewritten as

$$P = \frac{n \log p}{an \log(1-e^{-an})} = \frac{n \log(1/p)}{-an \log(1-e^{-an})} \tag{2.12}$$

For a desired $p(n)$ (i.e., if we fix p and n), the pool size P is minimized when the denominator $(-an \log(1-e^{-an}))$ is maximized, which occurs when $an = \log 2$. Corresponding to the choice of $a = \frac{\log 2}{n}$ the maximum value of the denominator is $(\log(1/2))^2 = (\log 2)^2$, and consequently the optimal values of P and k are

$$\begin{aligned} P &= \frac{n \log(1/p)}{(\log 2)^2} \\ k &= \frac{\log(1/p)}{(\log 2)^2} \end{aligned} \tag{2.13}$$

As a numerical example, if we desire $p(n = 1000) = e^{-23} \approx 1 \times 10^{-10}$ (probability of 1 in 10 billion), we choose $a = \frac{log2}{1000}$, and

$$\begin{aligned} P &= \frac{1000 \times 23}{\log(2)^2} \approx 47870 \\ k = Pa &= \frac{23}{(\log 2)} \approx 33. \end{aligned} \tag{2.14}$$

In other words, if random subsets each with 33 keys are randomly chosen from a pool of 47870 keys, the probability that the union of 1000 randomly chosen subsets will contain all keys in yet another randomly chosen subset, is about 1 in 10 billion.

2.2.2 $(\mathcal{S}_i \cap \mathcal{S}_j) \subset \mathcal{S}^n$

Consider a specific key in the pool of P keys. The probability that the key is present in both subsets $\mathcal{S}_i$ and $\mathcal{S}_j$, and therefore, in $\mathcal{S}_i \cap \mathcal{S}_j$, is a^2. The probability that the key *is present* in the intersection, but *not present* in the union $\mathcal{S}^n$ is

$$\epsilon = a^2(1-a)^n. \tag{2.15}$$

Thus, for any of the P keys, the probability that a key is present in the intersection of two sets, and in the union of n sets is $1-\epsilon$. The probability that all keys present in the intersection are present in the $\mathcal{S}^n$ is therefore

$$p = (1-\epsilon)^P = (1-a^2(1-a)^n)^P \approx (1-a(1-a)^n)^k. \tag{2.16}$$

In other words

$$P = \frac{\log p}{\log(1-a^2(1-a)^n)} \text{ and} \tag{2.17}$$

$$k = \frac{\log p}{\log(1-a(1-a)^n)} \tag{2.18}$$

From Eq. (2.18), it can be easily seen that for a given n, p, the number of keys in each subset, k, is minimized when $a(1-a)^n$ is maximized, which occurs when $a = 1/(n+1)$. The maximum value of $a(1-a)^n$ is then

$$\frac{1}{n+1}\left(1-\frac{1}{n+1}\right)^n = \frac{1/(n+1)}{1-1/(n+1)}\left(1-\frac{1}{n+1}\right)^{n+1} \approx \frac{1}{en} \tag{2.19}$$

Thus, for the optimal choice of $a = 1/(n+1)$,

$$p(n) = (1-\frac{1}{en})^k \approx e^{-k/en} \tag{2.20}$$

The minimal value k and the corresponding pool size $P = k/a$ are then

$$\begin{aligned} k &= en\log(1/p) \\ P &= en(n+1)\log(1/p) \end{aligned} \tag{2.21}$$

As a numerical example, if we desire $p(n = 1000) \approx e^{-23}$, we can choose $k = e \times 1000 \times 23 = 62520$ and $P = k/a = k(n+1) = 62582520$.

On the other hand, if we desire to minimize the key pool size P, from Eq. (2.18) we can see that it is required to maximize $a^2(1-a)^n$. This occurs for the choice of $a = 2/n$, corresponding to which the maximum value of $a^2(1-a)^n$ is

$$\frac{4}{n^2}(1-\frac{2}{n})^n \approx \frac{4}{n^2}\frac{1}{e^2} = \frac{4}{n^2e^2}. \tag{2.22}$$

As

$$p(n) = (1-a^2(1-a)^n)^P = (1-\frac{4}{n^2e^2})^P = e^{\frac{4P}{n^2e^2}}, \tag{2.23}$$

we have

$$\begin{aligned} P &= \frac{n^2e^2}{4}\log(1/p) \\ k = Pa &= \frac{ne^2}{2}\log(1/p) \end{aligned} \tag{2.24}$$

As a numerical example, if we desire $p(n = 1000) \approx e^{-23}$, we can choose $P = 42487073$ and $k = 84974$.

Chapter 3
Nonscalable Key Distribution Schemes

A key distribution scheme is a mechanism for distributing secrets and possibly some public (nonsecret) values to a group of participants. We shall simply refer to a group of participants as a *network*. The values distributed to *members* of the network enable them to engage in private and/or authenticated exchanges.

The main actors in a key distribution scheme include:

1. One or more key distribution centers (KDC) who *generate* secrets and public values
2. Members of the network who *utilize* such values for securing interactions between members

Within the network, every member has a unique identity. In a network with N members, let $M_1 \cdots M_N$ be the unique member identities. Depending on the nature of the network, members' identities may take different forms. For example, in a network where members exchange link-layer packets, the identities may be 6-byte medium access control (MAC) addresses. In a network where members exchange Internet protocol (IP) packets, the member identities may be 4-byte IPv4 or 16-byte IPv6 addresses. For interactions in the application layer, the member identities could take numerous application dependent forms forms like email addresses, login name, domain name, etc.

In this chapter, our focus is on key distribution schemes that facilitate establishment of pairwise secrets between members of a network. A secret K_{ij} shared by members M_i and M_j can be used by M_i and M_j to authenticate messages exchanged between them by computing message authentication codes using the shared secret, or for encrypting messages using a suitable block-cipher.

Key distribution schemes for facilitating pairwise secrets can be classified into two broad categories. In the first category are schemes which require a trusted KDC to be available online. Such schemes are discussed first in Sect. 3.1.

In the second category are schemes where KDCs are not available online. In such scenarios, KDCs interact with each member only once—for initial issue of keys and public values. Schemes with offline KDCs involve offline predistribution of keys by the KDC.

M. Ramkumar, *Symmetric Cryptographic Protocols,* DOI 10.1007/978-3-319-07584-6_3,

Key predistribution schemes can be broadly classified into nonscalable schemes, and scalable schemes. In Sects. 3.2 and 3.3 we restrict ourselves to nonscalable schemes.

3.1 Online KDC

If the trusted KDC is always available online, the symmetric Needham–Schroeder (NS) protocol [14] can be used for establishing authenticated pairwise secrets between members.

3.1.1 NS Protocol

The prerequisite for the symmetric NS protocol is that every member shares a secret with a trusted server T which acts as the KDC for the network. Let K_i be the secret shared between member M_i and the server T. Likewise, let K_j be the secret shared between T and a member M_j.

When M_i desires to communicate with M_j, M_i initiates a protocol that facilitates establishment of a shared secret K_{ij} between M_i and M_j. The simplified NS protocol can be represented as follows:

$$\begin{aligned} M_i \rightarrow T &: \{M_i, M_j\} \\ T \rightarrow M_i &: \{E_{K_i}(K_{ij}), E_{K_j}(M_i, K_{ij})\} \\ M_i \rightarrow M_j &: E_{K_j}(M_i, K_{ij}) \end{aligned}$$

In the protocol above, M_i requests the server T to issue a session secret for communicating with M_j. The server chooses a random session secret K_{ij} and sends two copies back to M_i — $E_{K_i}(K_{ij})$ encrypted using the secret K_i shared between the server and M_i, and the other—a *ticket* $E_{K_j}(M_i, K_{ij})$—encrypted using the secret K_j shared between the server and M_j. M_i sends the ticket to M_j, which can be decrypted by M_j to obtain the session secret K_{ij} and the identity M_i of the initiator of the protocol. At the end of the protocol, both M_i and M_j share a common secret K_{ij} issued by the trusted server.

The NS protocol is the basis for the Kerberos [15] protocol for establishing secure channels between end-user clients and various services offered to clients in a Kerberos realm.

3.1.2 Leighton–Micali Protocol

Leighton and Micali [16] proposed an alternative approach for establishing shared secrets using a trusted online server. Similar to the NS protocol, Leighton–Micali

(LM) scheme also assumes the existence of a shared secret between the server and every member. Once again, let K_i represent the secret shared between M_i and the server T. The simplified form of the LM protocol for establishing a shared secret between M_i and M_j is as follows:

$$\begin{aligned} M_i \rightarrow T &: \{M_i, M_j\} \\ T \rightarrow M_i &: P_{ij} = h(K_i \parallel M_j) \oplus h(K_j \parallel M_i). \end{aligned}$$

At the end of the protocol both M_i and $M_j B$ can compute a common secret

$$K_{ij} = h(K_j \parallel M_i). \tag{3.1}$$

The responder M_j, who has access to K_j, can directly compute $K_{ij} = h(K_j \parallel M_i)$. On the other hand, the initiator M_i—who does *not* have access to K_j, but has access to K_i—uses the value P_{ij} provided by T, to compute the secret K_{ij} as

$$\begin{aligned} K_{ij} &= h(K_i \parallel M_j) \oplus P_{ij} \\ &= h(K_i \parallel M_j) \oplus h(K_i \parallel M_j) \oplus h(K_j \parallel M_i) \\ &= h(K_j \parallel M_i). \end{aligned} \tag{3.2}$$

As P_{ij} does not reveal any information K_i or K_j (and even K_{ij}—except to entities who already possess K_i or K_j), the value P_{ij} need not be kept a secret. Values like P_{ij} are *pair-wise public values*. Note that $P_{ij} = P_{ji}$. For a network of size N there are $\binom{N}{2}$ such pair-wise public values.

The main advantage of the LM scheme compared to the NS approach stems from the fact that no authenticated exchange is required between the KDC and the initiator to receive the public value P_{ij}. It is therefore conceivable that values like P_{ij} are even be made available in a public repository, to obviate the need for the trusted KDC to be always online.

If the repository is untrusted, some additional strategies may be required for authenticating the integrity of the public values hosted by the untrusted repository. For this purpose, Leighton and Micali proposed a parallel scheme, where each node receives another independent secret. Corresponding to every public value like P_{ij} is an authenticator Q_{ij} for the public value, the integrity of which can be verified using the second secret issued to M_i or M_j.

As yet another possibility, the $\binom{N}{2}$ public values could be used to construct a Merkle hash tree (discussed earlier in Sect. 3) the root of which is known to every member. The $n = \binom{N}{2}$ values are interpreted as leaf-nodes of the tree. The untrusted repository will now be required to store an additional $n - 1$ internal nodes of the tree. Any public value provided by the untrusted repository should be accompanied by $L = \log_2\left(\binom{N}{2}\right)$ hashes which can be used to verify the integrity of the public value against the root.

3.2 Offline KDC

If the KDC is *not* available online, the chief considerations that will influence the choice of a suitable key distribution strategy for a specific application include the (a) network scale and dynamics, and (b) expected communication patterns between members of the network.

The network scale is a measure of the total number of members. In a network with dynamic scale, new participants may be added (or even removed) from the network. In some networks it may be necessary for every member to be able to interact with every other member. In such scenarios, every possible pair of members should be able to arrive at a pairwise secret. In some networks, every member may be required to establish a secret only with small number of other members. For example, in most client–server networks, a large number of clients may be required to communicate only with a small number of servers. Clients are not required to communicate directly with each other.

3.2.1 Basic KDS for Static Small-Scale Networks

In the well-known basic key distribution scheme, for a network with fixed number (say) N participants, the KDC is required to generate $\binom{N}{2}$ secrets $K_{ij}, 1 \leq i, j \leq N, i \neq j$—one corresponding each pair of members—and issue $N-1$ secrets to each member. Let K_{ij} be the pairwise secret shared between two members M_i and M_j. The $N-1$ secrets issued to a member M_i are $\{K_{ij}\}, 1 \leq j \leq i-1, i+1 \leq j \leq N$.

In practice, the KDC may choose a single master secret μ which can be used to generate any of the $\binom{N}{2}$ secrets. Some of the possible ways in which a pairwise secret K_{ij} can be computed by the KDC are

$$K_{ij} = h(\mu \parallel M_i \parallel M_j) \oplus h(\mu \parallel M_j \parallel M_i) \, \text{or}$$

$$K_{ij} = \begin{cases} h(\mu \parallel M_i \parallel M_j), & \text{if } M_i > M_j \\ h(\mu \parallel M_j \parallel M_i), & \text{if } M_j > M_i \end{cases}$$

Once the KDC has conveyed $N-1$ secrets to each member, the KDC has no further role to play in the day-to-day functioning of the network.

While the KDC could randomly choose $\binom{N}{2}$ secrets, generating them from a single secret μ is advantageous, as the KDC needs to maintain secure storage only for a single secret μ. To convey $N-1$ secrets to every member, the KDC may first convey a single secret to each member, and then convey the $N-1$ secrets by encrypting them using the secret previously conveyed. In other words, as long as the KDC is able to securely convey a single secret to each member, conveying the additional $N-1$ secrets does not pose any significant challenge.

As a practical example, the single secret conveyed to member M_i may simply be

$$K_i = h(\mu \parallel M_i). \tag{3.3}$$

Other secrets like K_{ij} could then be conveyed as

$$K'_{ij} = h(K_i \parallel M_j) \oplus K_{ij}. \tag{3.4}$$

To compute the key shared with M_j, member M_i can readily decrypt K'_{ij} as

$$K_{ij} = h(K_i \parallel M_j) \oplus K'_{ij}. \tag{3.5}$$

It is also advantageous for members themselves to store their $N - 1$ secrets in an encrypted manner. Each member needs to allocate secure storage only for a single secret—the $N - 1$ encrypted secrets can then be stored in an easily accessible and possibly unprotected location.

Corresponding to each of the $N - 1$ keys, member M_i will need to store two values—the member identity (like M_j), and the corresponding encrypted secret K'_{ij}. If identities and secrets are 128-bit (or 16 byte) values, for a network size of 1 million each member will need 32 MB of (unprotected) storage (in addition to protected storage for a single secret).

3.2.2 Key Distribution for Dynamic Networks

The basic key distribution scheme is obviously not well suited for large network scales and for networks with dynamic scales. Even while the rapidly decreasing cost of storage indicates that even the basic scheme can easily support static networks with tens of millions of members, such an approach is unsuitable for networks where new members may join at any time.

Consider a network which currently has N members. When a new (($N + 1$)th) member joins the network the KDC should issue:

1. N secrets to the new member
2. 1 secret to each of the N existing members

It is the second requirement which is especially impractical, as it would imply that every member will have to communicate with the KDC every time a new member joins the network.

Ideally, the interactions between the KDC and the members should be restricted to be a one-time affair—only when a member joins a network. This requirement can be met by using the modified Leighton–Micali scheme (MLS) described in the next section.

3.3 MLS Key Distribution

As the name implies, MLS [17] is derived from the LM scheme discussed in Sect. 3.1.2. In the LM scheme for N members $M_1 \cdots M_N$, recall that each member is issued one secret—let K_i be the secret issued to member M_i. In addition, there are $\binom{N}{2}$ public values of the form

$$P_{ij} = h(K_i \parallel M_j) \oplus h(K_j \parallel M_i). \tag{3.6}$$

For small N, the members themselves may each store $N - 1$ public values, for example, M_i stores all $N - 1$ P_{ij} for which $i \neq j, 1 \leq i, j, N$. With this approach, the need for accessing the public repository is eliminated.

The central idea behind the modified scheme stems from the realization that only one of the two members—M_i or M_j—requires access to the public value P_{ij}. In the LM scheme the initiator requires access to the value P_{ij}. Unfortunately, as M_i or M_j could be the initiator, both require access to P_{ij}. However, if we can specify a clear rule that unambiguously indicates which of the two members (M_i or M_j) will *always* need to use the public value—irrespective of where the member is an initiator or responder—then only that member will need access to the pairwise public value.

In MLS the rule that dictates which of the two members is required to use the public value is based on *when* the members joined the network. The member that joined the network later (the newer member) should use the public value. Older members do *not* need access to public values corresponding to newer members.

The advantage of imposing such a rule is that it solves the issue of dynamic network scale. Whenever a new member joins the network, only the new member is to be provided public values corresponding to every existing member. Unlike the basic KDS, the KDC does not need to convey an additional value to each of the existing members.

The MLS scheme is well suited for small to medium scale *dynamic* networks. Let us assume that the network is intended to support up to 2^n participants (for example, $n = 32$, or maximum network scale of 4 billion). For such a network, the KDC chooses a master secret μ. Each member is associated with a sequence number and an identity. Let the identity assigned to the first member inducted into the network be M_1. The member M_1 is assigned a sequence number 1. The secret issued to the member M_1 is computed by the KDC as

$$K_1 = h(\mu \parallel (M_1 \parallel 1)). \tag{3.7}$$

The second member with identity M_2 is issued a secret $K_2 = h(\mu \parallel (M_2 \parallel 2))$. In addition, member M_2 is issued one public value (corresponding to the single older member)

$$P_{21} = h(K_1 \parallel (M_2 \parallel 2)) \oplus h(K_2 \parallel (M_1 \parallel 1)). \tag{3.8}$$

The common secret between M_1 and M_2 is

$$K_{21} = h(K_1 \parallel (M_2 \parallel 2)). \tag{3.9}$$

Member M_1 can directly compute the secret using its secret K_1. M_2 computes the same secret as

$$\begin{aligned} K_{21} &= h(K_2 \parallel (M_1 \parallel 1)) \oplus P_{21} \\ &= h(K_1 \parallel (M_2 \parallel 2)). \end{aligned} \tag{3.10}$$

Similarly, the third member to be inducted into the network, receives one secret $K_3 = h(\mu \parallel (M_3 \parallel 3))$ and two public values P_{31} and P_{32}—one public value corresponding to each of the two current members. Thus, the millionth member to be inducted receives one secret and $999,999$ public values—one public value corresponding to each member inducted earlier.

In general, the common secret between two nodes M_x and M_y, where $x > y$ is computed as

$$K_{xy} = \begin{cases} h(K_y \parallel (M_x \parallel x)) & \text{by } M_y \\ h(K_x \parallel (M_y \parallel y)) \oplus P_{xy} & \text{by } M_x. \end{cases} \tag{3.11}$$

For simplicity of notation, we shall henceforth assume that n MSBs of member identities like M_i indicate the sequence number i. With this simplified representation, the secret K_i assigned to M_i is

$$K_i = h(\mu \parallel M_i). \tag{3.12}$$

The $i - 1$ public values assigned to U_i are

$$\begin{aligned} P_{i,1} &= h(K_1 \parallel M_i) \oplus h(K_i \parallel M_1) \\ P_{i,2} &= h(K_2 \parallel M_i) \oplus h(K_i \parallel M_2) \\ &\vdots \\ P_{i,i-1} &= h(K_{i-1} \parallel M_i) \oplus h(K_i \parallel M_{i-1}). \end{aligned} \tag{3.13}$$

3.3.1 Identity Ticket (IT) Scheme

An "identity ticket" is derived as a one way function of a label and a secret. For example, $h(K \parallel A)$ is an identity ticket corresponding to a key K and a label A. While identity tickets are conceptually similar to key based message authentication codes, *identity tickets are to be treated as secrets* (unlike MACs). The identity ticket $K_A = h(K \parallel A)$ is privy only to the entity A and entities who have access to the secret K (and can therefore readily *compute* the ticket).

In the identity ticket (IT) scheme the KDC chooses a master secret μ. Similar to MLS, some LSBs of the identity assigned to every member indicate a sequence number. Let the sequence number of member with identity M_i be i.

Each member is assigned

1. One secret
2. One identity ticket corresponding to every member inducted *before* M_i

The secret K_i assigned to M_i is

$$K_i = h(\mu \parallel M_i). \tag{3.14}$$

The $i - 1$ identity tickets issued to M_i are

$$\begin{aligned} T_{i,1} &= h(K_1 \parallel M_i) \\ T_{i,2} &= h(K_2 \parallel M_i) \\ &\vdots \\ T_{i,i-1} &= h(K_{i-1} \parallel M_i). \end{aligned} \tag{3.15}$$

The shared secret K_{ij} between M_i and M_j (where $i < j$) is $T_{j,i}$. The member M_i computes the ticket as $h(K_i, M_j)$ using it's secret K_i. The member M_j already possesses the ticket $T_{j,i} = h(K_i, M_j)$.

3.4 Comparison

In all three schemes every member needs to store only a single secret, and store $\mathbf{O}(N)$ values in unprotected bulk storage. Specifically, N is the current network size in MLS and IT schemes, and the static network size in the basic KDS.

Apart from the obvious advantage of the ability to support dynamic networks scales, MLS and IT offer several other advantages compared to the basic scheme. Specifically, MLS and IT.

1. Reduce the bulk-storage complexity for each member by a factor of four (on an average)
2. Halve the need for access to bulk storage
3. Eliminate the need for searching thorough stored values in bulk-storage

In the basic scheme for a network with N members, a member M_i had to store two values corresponding to each of the $(N-1)$ members—the identity and the encrypted secret. In the basic scheme, When M_i receives a message from M_j (authenticated using the shared secret K_{ij}), the following occurs:

1. M_i searches the bulk storage for a key with index M_j.
2. M_i fetches the encrypted key K'_{ij}.
3. M_i decrypts K'_{ij} to compute K_{ij}.

The sender M_j would have also obtained K_{ij} in the same manner, viz., searching its bulk-storage for a key with index M_i and decrypting the stored key.

In the MLS scheme, M_i will store $i-1$ public values—one corresponding to each member inducted before M_i. It is not necessary to explicitly store the indexes as the key corresponding to M_j $j < i$ can be stored in the j^{th} location. On an average, every member has to store only $N/2$ public values. Similarly, in the IT scheme M_i will store $i - 1$ tickets, one corresponding to each member inducted before M_i.

Thus, once M_i receives a message from M_j (assume $i > j$) authenticated using secret K_{ij}, M_i (to compute the secret K_{ij}) proceeds with the following:

1. It fetches the j^{th} public value P_{ij} (or the j^{th} ticket $T_{i,j}$) from a file in bulk storage with $i-1$ public values/tickets (there is no need to search).
2. It computes $K_{ij} = h(K_i \parallel M_j) \oplus P_{ij}$ (or the stored ticket is decrypted to yield $K_{ij} = T_{ij}$ in IT).

The sender M_j in this case, recognizing that $i < j$, would have computed K_{ij} as $K_{ij} = h(K_j \parallel M_i)$ (or $K_{ij} = h(K_j \parallel M_i)$ in IT scheme) without even requiring an access to its bulk storage.

To summarize, if both identities and public values are of the same size (say, 16 byte or 128-bit values), for the same network size MLS/IT demands only one-fourth the storage demanded by basic KDS. For any interaction between two members, only one of the two require access to bulk storage. The basic KDS requires both members to search for a specific 128-bit identity (a small subset of 2^{128} possible identities are stored in a list of size $N-1$). In MLS only one of the two members need to access storage, and even that member does not need to search the storage. As identities need not be stored, MLS/IT can also use efficiently use large bit-strings as identities if desired.

One compelling advantage of MLS over IT stems from the fact that distribution of nonsecret values to members can be more efficient than distribution encrypted tickets. For example, all public values could be stored in a repository and accessed by members as and when required. Henceforth, we shall restrict ourselves to MLS, even while all remaining discussions regarding MLS also applies to IT.

3.4.1 MLS with Multiple KDCs

MLS can be easily extended to support multiple KDCs. When m independent KDCs are used, each member will need to store m secrets in protected memory. However, the storage required for public values remains the same as the single KDC scenario.

Consider the scenario with $m = 2$ KDCs. Both KDCs choose an independent master secret, say μ^1 and μ^2. Every member receives one secret from each KDC. Let the secrets received by member U_i from the two KDCs be

$$\begin{aligned} K_i^1 &= h(\mu^1 \parallel M_i) \\ K_i^2 &= h(\mu^2 \parallel M_i). \end{aligned} \tag{3.16}$$

Both KDCs issue $i-1$ different public values to M_i. Specifically the values issued by the two KDCs are respectively,

$$\begin{aligned} P_{ij}^1 &= h(K_i^1 \parallel M_j) \oplus h(K_j^1 \parallel M_i), 1 \leq j \leq i-1 \\ P_{ij}^2 &= h(K_i^2 \parallel M_j) \oplus h(K_j^2 \parallel M_i), 1 \leq j \leq i-1. \end{aligned} \tag{3.17}$$

The member M_i combines each of the public values from both KDCs to compute

$$P_{ij} = P_{ij}^1 \oplus P_{ij}^2, 1 \leq j \leq i-1 \tag{3.18}$$

and stores the $i-1$ public values in bulk storage.

The shared secret between M_i and M_j (assume $i > j$) is

$$K_{ij} = h(K_j^1 \parallel M_i) \oplus h(K_j^2 \parallel M_i) \tag{3.19}$$

Specifically, M_j computes the shared secret K_{ij} directly using its secrets K_j^1 and K_j^2, without the need to access bulk storage. Member M_i computes the same secret as

$$\begin{aligned} K_{ij} &= h(K_i^1 \parallel M_j) \oplus h(K_i^2 \parallel M_j) \oplus P_{ij} \\ &= (h(K_i^1 \parallel M_j) \oplus P_{ij}^1) \oplus (h(K_i^2 \parallel M_j) \oplus P_{ij}^2) \\ &= h(K_j^1 \parallel M_i) \oplus h(K_j^2 \parallel M_i). \end{aligned} \tag{3.20}$$

3.4.2 MLS Applications

For most practical networks not all members are required to communicate with all other members. The necessary communication patterns within a network $\mathcal{N}$ can be modelled by dividing the network into two *possibly overlapping* sets $\mathcal{N}_1$ and $\mathcal{N}_2$ where every member in set $\mathcal{N}_1$ is required to establish a shared secret with every member in set $\mathcal{N}_2$.

The case where $\mathcal{N}_1 = \mathcal{N}_2 = \mathcal{N}$ (the two sets completely overlap) corresponds to the scenario where every member is required to share a secret with every other member.

In a scenario where $|\mathcal{N}_1| > |\mathcal{N}_2|$, the members $\mathcal{N}_1$ may each be required to store $|\mathcal{N}_2|$ public values. Thus, the storage requirements are limited by the size of the smaller set, $n_s = \min(|\mathcal{N}_1|, |\mathcal{N}_2|)$.

The value n_s is only limited by the unprotected (but readily accessible) storage available to every member. Even if every member is a mobile device, storage is not a practical concern. For example, if $n_s = 20 \times 10^6$, on an average, every member will need storage for 10 million public values—say 160 MB. The worst case for the storage requirement of 320 MB. The rapidly reducing costs of storage, and the fact that only one-time bandwidth overhead is required for disseminating public values, suggests that MLS (or IT) are indeed practical for most networks. It is also intuitively satisfying that members who join the network at a later time will require more storage. In most practical scenarios newer devices will possess better capabilities.

As an example, consider a network $\mathcal{N}$ consisting of an unlimited number of clients and a limited number of servers. Also, assume that it is necessary for the following:

1. Every client to be able to share a secret with every server
2. Every server to also be able to establish a secret with all clients and servers

In this case, the two overlapping sets are $\mathcal{N}_1 = \mathcal{N}$ (which includes both clients and servers) and $\mathcal{N}_2$ (which includes only servers). As long as the number of servers is not prohibitively high (for example, not more that several tens of millions), MLS/IT are indeed practical options.

As some practical examples,

1. The network $\mathcal{N}$ may be a global e-mail network, with an unlimited number of clients, and a substantially smaller number (for example, a few million) e-mail servers. In such an application, every client may need to interact only with a small number of servers. However a server may require to interact with any server, depending on the path to be used for relaying the e-mail.
2. The network may be the domain name system (DNS). The (practically unlimited) owners of domain names (or zones), and the even larger hosts that query the domain name system are the clients. The servers are DNS servers that store and relay domain name records created by zone owners to end users.

Currently, commonly used security protocols like secure socket layer (SSL) and TLS (transport layer security) rely on expensive asymmetric encryption and signature algorithms, in conjunction with a public key infrastructure (PKI), to facilitate secure interactions between a different types of servers and clients. The security protocol used DNSSEC (for securing DNS) uses asymmetric signature algorithms to authenticate DNS records created by owners of zones. Signature verification algorithms are used by clients who query DNS records. In the rest of this chapter we outline low complexity alternatives to DNSSEC and TLS/SSL using MLS key distribution.

Chapter 4
MLS for Internet Security Protocols

In this chapter we begin with an overview of the domain name system (DNS) (in Sect. 4.1), issues in securing DNS (Sect. 4.2), and the current security protocol, DNSSEC, for DNS (Sect. 4.3). Sections 4.4–4.6 outline the alternative to DNSSEC, and provide an in-depth comparison of the current and proposed protocols. Section 4.7 outlines a practical alternative to [35].

4.1 Domain Name System

The domain name system [18, 19] is a tree-hierarchical naming system for services that can be accessed over the Internet. At the top of the inverted DNS tree (see Fig. 4.1) is the *root*. Below the root are generic top level domains (gTLD) like `com`, `org`, `net`, `edu`, etc., and country-code top level domains (ccTLD) like `br`, `ca`, etc. A leaf named `b.cs.univ.edu` in the DNS tree is a server-host in a branch `cs.univ.edu`, which stems from a thicker branch `univ.edu`, which stems from an even thicker branch `.edu`, stemming from the root of the DNS tree.

A branch of the tree (including its sub-branches and leaves) under the administrative control of an owner, is a DNS zone. The owner for a zone is responsible for:

1. Assigning names for branches and leaves under the zone
2. Creating DNS resource records corresponding to such names
3. (Optionally) delegating an entity as the owner for a branch within the zone

The owner of the root zone has delegated a gTLD zone like `.edu` to a `.edu`-gTLD owner, who has in turn delegated the zone `univ.edu` to another entity, who may have delegated a zone `cs.univ.edu` to yet another authority. All DNS records for the zone `cs.univ.edu` have names that end with `cs.univ.edu`. The zone records are created by the owner of the zone. The zone owner also specifies the names of *authoritative* name servers (ANS) for the zone. A "zone master file" which includes the set of all DNS RRs for the zone is provided every ANS of the zone. ANSs for the root zone are the *root name-servers*. ANSs for gTLD zones are gTLD servers; ANSs for other zones are simply referred to as zone servers.

M. Ramkumar, *Symmetric Cryptographic Protocols,* DOI 10.1007/978-3-319-07584-6_4,

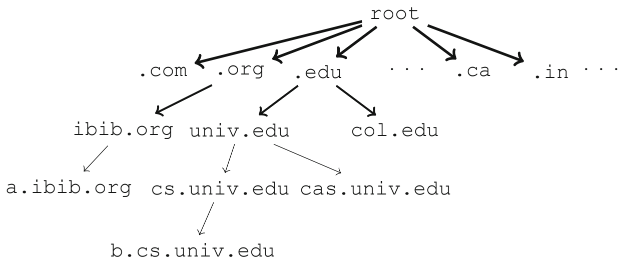


Fig. 4.1 The domain name system (DNS) tree. A leaf of the tree is typically a DNS name of a server (for example, web-server, mail-server, DNS servers etc.). Some internal nodes correspond to starting points of DNS zones. Associated with each zone is an authority (or owner), and authoritative name servers (ANS) for the zone

A client desiring to access a HTTP service `www.cs.univ.edu` requires the IP address of the web server with a domain name `www.cs.univ.edu`. This information is available in an A-type record in the master file for a zone under which the name `www.cs.univ.edu` falls, and can be obtained by querying any name server for the zone. To obtain this information, the querier only needs to know the IP address of one root name server. While the root name server cannot directly provide the answer to the query "`www.cs.univ.edu, A`," it can provide the names and IP address of the name servers for gTLD and ccTLD zones. In this case, the root server will respond with the names and IP addresses of all `.edu` servers.

The querier can now send the same query to any of the `.edu` servers, which will respond with the name and the IP address of the name servers for the zone `univ.edu`. When the same query "`www.cs.univ.edu, A`" is directed to a name server for the zone `univ.edu`, the response includes the names and IP addresses of name servers for the child zone `cs.univ.edu`.

Finally, any of the name servers for the zone `cs.univ.edu` can be queried to obtain the desired the A-type record. If the zone `cs.univ.edu` had not been subdelegated, then the name server for the zone `univ.edu` would have directly provided the response. Thus, knowing only the IP address of one root name server, any one can obtain any DNS record by specifying the name and type, and performing a series of queries.

4.1.1 DNS Records

Every DNS resource record (RR) [19] is a five-tuple consisting of (i) name, (ii) class, (iii) time-to-live (TTL), (iv) type, and (v) value; for example,

```
  name=www.cs.univ.edu, IN, TTL=2345, type=A,
value=159.43.7.82.
```

The class is always IN (for Internet RRs); the field TTL is specified in seconds, and indicates how long a RR can be cached. In the remaining chapter, to keep notations simple, we shall ignore the fields "class" and "TTL."

A-type RRs indicate an IP address in the value field. An NS-type record `name=cs.univ.edu, type=NS, value=ns1.dserv.net` indicates that `ns1.dserv.net` is a name server for the zone `cs.univ.edu`.

A set of records with the same name and type, but with different value fields, is collectively referred to as an RRSet. For example, the NS-type RRSet for the name `cs.univ.edu` may include two NS records—one indicating a name server `ns1.dserv.net`, and the other indicating another name server `ns1.cs.univ.edu`.

The NS type records are used for delegation. An RRSet of NS records for a delegated zone (say) `cs.univ.edu` can be found in the master file of the parent zone `univ.edu`. Similarly, the NS RRSet for `univ.edu` can be found in the master file of the zone `.edu`, and so on. Along with NS records which specify ANSs, the A-type records for the ANSs are also included in the master file as *glue* records.[1]

The zone owner is (obviously) always offline. Once the master file for zone has been provided to all zone servers, and the names of zone servers conveyed to the authority of the parent zone (and included as NS records in the master file of the parent zone), the zone owner simply expects the name servers to faithfully accept and respond to DNS queries regarding the zone.

4.1.1.1 Query-Response Process

DNS queries and responses [19] are typically payloads of UDP packets and have the same packet format. They include a header, and four sections: `QUESTION, ANSWER, AUTHORITY,` and `ADDITIONAL`. In a query packet `QUESTION` section indicates the queried name and type (all other sections are empty). The response has an identical `QUESTION` section. The `ANSWER` section contains the desired RRSet. The `AUTHORITY` section includes NS records for the zone. The `ADDITIONAL` section contains A-type glues for the NS records.

In practice, clients initiate queries using *stub-resolvers* running on their own host machine. Stub-resolvers do not directly query the zone name servers. Instead, they use local domain name servers (LDNS) as intermediaries. LDNSs are also referred to as preferred name servers, or local recursive resolvers, or caching-only name servers, and are typically operated by Internet service providers (ISP).

An application requiring the IP address of (say) `www.cs.univ.edu` queries a stub-resolver running on the same host. The stub-resolver redirects the query

[1] Note that zone `univ.edu` (or even `edu`) cannot be authoritative for the zone `dserv.net`. Thus, while `univ.edu` can provide an authoritative response regarding the *name* of the ANS for the child zone `cs.univ.edu`, it cannot provide an authoritative A-type record for the server `ns1.dserv.net`. To avoid possible circular dependency problems, the necessary *nonauthoritative* A-type records are included as glue records.

((`www.cs.univ.edu,A`)) to a LDNS. To do so, the host (or the stub-resolver) should know the IP address[2] of at least one LDNS.

All LDNSs are preconfigured with the IP addresses of root servers. The LDNS queries a root server for (`www.cs.univ.edu,A`), and receives NS records (with glued A-type records) for name servers of `.edu`. The LDNS then queries a `.edu` name server to receive NS records of `univ.edu`, and so on. Finally, a server for the zone `cs.univ.edu` responds to the query with the desired A-type RRSet, which is relayed back to the stub-resolver.

LDNSs may cache RRs for a duration specified by the TTL field in the RR, and may respond to queries from stub-resolvers using cached RRs. Similarly stub-resolvers may also cache RRs and respond to queries from applications running on the same host using the cached RRs.

4.2 Securing DNS

The main goal of attacks on DNS is to simply divert traffic away from genuine services, or more often, to divert such traffic to impersonators phishing for personal information from unsuspecting clients. A common strategy for attackers is to impersonate zone servers to provide fake DNS responses to LDNSs, thereby "poisoning the cache" of the LDNS, and consequently, the caches of many stub-resolvers which employ the poisoned LDNS.

The header of every DNS query includes a 16-bit transaction ID t_{id}; the UDP packet carrying the query indicates a 16-bit source port p chosen[3] by the querier. A DNS response for query will be accepted only if it is addressed to port p, and if it indicates an expected transaction ID t_{id}.

To create a fake response that will be accepted by an LDNS, an *out-of-path* attacker, who does not have plain-sight view of the query packet, will need to guess the values t_{id} and p. A typical strategy for an out-of-path attacker is to register a domain, run his own server for the domain, and query the targeted LDNS for a name under his domain. When the query from the LDNS is ultimately directed to the attacker's server, the attacker learns enough information to narrow down the two values t_{id} and p within small range.

Recently, Kaminsky [20] pointed out that DNS cache poisoning attacks can have even more severe consequences. Instead of attempting to poison RRs corresponding to a specific zone, the attacker can impersonate a root server and send fake glue records for "IP addresses of gTLD name servers." Thus, queries to every `.com` zone, for example, will then be directed to a computer under the control of the attacker, which could redirect such queries to other "name servers" under the attacker's control.

[2] Typically, IP addresses of LDNSs are provided to a host by a DHCP server. In UNIX-like machines the IP addresses of LDNS are stored in a file/etc/resolv.conf.

[3] Typically chosen by the operating system.

4.2.1 Link-Security Approaches

While properly randomizing the two 16-bit values (t_{id} and p) is a good first step, they offer no defense against *in-path* attackers. In-path attackers who may be in the same LAN as the server or the resolver,[4] or are conveniently situated in-between the resolver and the server, have plain-sight access to the values t_{id} and p in the UDP DNS packets, and can thus readily create fake responses. Securing links between DNS servers (for example, by using a secret shared between a resolver and the server queried by the resolver) can prevent such attacks.

Two entities A and B who share a secret K_{AB} can prevent even in-path attackers from impersonating them by (i) encrypting the message sent over the link using the shared secret K_{AB}, or (ii) appending a message authentication code (MAC) $h(V \parallel K_{AB})$ where V may be the hash of the contents of the DNS response.

Approaches like SK-DNSSEC [21] and DNSCurve [22] adopt such a strategy. In symmetric key DNSSEC [21] all LDNSs have the ability to establish secure channel with the root servers. ANSs higher in the hierarchy act as trusted servers and facilitate establishment of secrets with ANSs lower in the hierarchy, using the Needham–Schroeder protocol [14]. When a LDNS queries the root server for "`cs.coll.edu`, A", the root server's response includes a ticket which permits the LDNS to establish a secure channel with a `.edu` DNS server. The `.edu` DNS server then issues a ticket for securely communicating with an ANS for the zone `coll.edu`.

DNSCurve [22] employs a Diffie–Hellman scheme over a special elliptic curve $\mathcal{C}$ for setting up a private channel between DNSCurve enabled DNS servers. A DNSCurve enabled server A chooses a secret a. The secret between two DNSCurve enabled servers/resolvers A and B (where B's secret is b) is

$$K_{AB} = \mathcal{C}(b, \alpha) = \mathcal{C}(a, \beta), \tag{4.1}$$

where $\alpha = \mathcal{C}(a, S)$, $\beta = \mathcal{C}(b, S)$, and S is a public parameter.

Link-security approaches assume that the DNS servers themselves are trustworthy. Note that while link-security approaches protect DNS RRs from out-of-path attackers (who do not have access to values t_{id} and p) *and* in-path attackers (those with access to t_{id} and p), there is nothing that prevents an entity controlling the DNS server itself from modifying an RR. In practice, such an attacker can be the operator of a DNS server, or some other entity who has otherwise gained control of the DNS server. Such an attacker can receive RRs over protected links, illegally modify RRs, and relay fake RRs over "protected" links.

[4] Or, more generally, can exert some form of control over a computer in the same LAN as the server or the resolver.

4.3 DNSSEC

Ideally, the “middle-men” should not be trusted: only the zone owner should be trusted for providing information regarding the zone. This is the approach taken by DNSSEC [23], where every RRSet in the zone master file is individually signed by the zone owner.

A DNSSEC zone owner has an asymmetric key pair. The public portion of the key pair is certified by the owner of the parent zone. For example, the public key of the zone `cs.univ.edu` is signed by the owner of zone `univ.edu`. The public key of the zone `cs.univ.edu` can be obtained by querying for a DNSKEY-type RR for the name `cs.univ.edu`. To authenticate the public key in the DNSKEY RR, the parent zone `univ.edu` introduces two RRs in its zone file: a delegation signed (DS) RR which indicates a key-tag (a hash) for the public key of its child, and an RRSIG(DS) record which is the signature for the DS record.

For verifying the RRSIG(DS) record the public key of the parent zone `univ.edu` is required—which is the DNSKEY RR for the name `univ.edu`. To authenticate the public key of `univ.edu`, it is necessary to obtain the DS and RRSIG(DS) record from *its* parent zone—`.edu`, along with the DNSKEY RR for `.edu`. Finally, the public key of `.edu` can be verified by obtaining DS and RRSIG(DS) records from the root zone (by querying any root server). The public key of the root zone is assumed to be well publicized.

To summarize, corresponding to every RRSet for the zone `cs.univ.edu` is a RRSIG(RRSet) record which contains the digital signature for the RRSet. In response to a query for an RRSet, the corresponding RRSIG record is also included in the response. To verify the RRSIG, the required DNS RRs are:

1. DNSKEY RR of `cs.univ.edu`
2. DS, RRSIG(DS) corresponding to DNSKEY RR of `cs.univ.edu`, and DNSKEY RR of the parent zone `univ.edu` (fetched from the parent zone `univ.edu`);
3. DS, RRSIG(DS) corresponding to DNSKEY RR of `univ.edu`, DNSKEY RR of `.edu`;
4. DS, RRSIG(DS) corresponding to DNSKEY RR of `.edu`, from the root zone.

4.3.1 Authenticated Denial

Consider a scenario where the owner of the zone `wesellstuff.com` outsources its DNS operations to `dnsnet.net`. It is indeed conceivable that a competitor `wealsosellstuff.com` could bribe some personnel in `dnsnet.net` (or any entity who has acquired control over the server `dnsnet.net`) to remove the record for `wesellstuff.com` (for example, to drive the competitor out of business).

To ensure that DNS servers and/or their operators need not be trusted, DNSSEC demands a pertinent response from the zone server for *every* query that falls under the zone, where the *response is signed by the zone owner*. If the queried name exists, the ANS should provide a signed RRSet. If the queried name does *not* exist, the ANS is expected to provide *authenticated denial* by providing some information signed by the zone owner, which demonstrates that the queried record does not exist. If the zone server ignores the query, or provides a nonpertinent response, the resolver will send the query again, or will query another zone server for the zone, till it receives a pertinent response.

For example, in response to a query for name `abc.xyz.fgh` the querier expects a signed RRSet by the authority for the zone under which the name `abc.xyz.fgh` falls, or alternately, expects:

1. A signed response from the authority of the root zone that no record for a name `.fgh` exists; or
2. A signed response from the authority of the zone `.fgh` that no record for the name `xyz.fgh` exists; or
3. A signed response from the authority of the zone `xyz.fgh` that no record for the name `abc.xyz.fgh` exists.

As the zone authority is offline, a response denying every possible (as yet unknown) query, regarding the practically unlimited number of possible names and types that can fall under the zone, should somehow be signed by the zone authority and included in the master file provided to all zone servers. This is accomplished cleverly through NSEC records [12]. A signed NSEC record `abc.example.com, NSEC, cat.example.com` indicating two *enclosers* is interpreted as an authenticated denial of all enclosed names: viz., names that fall between `abc.example.com` and `cat.example.com` in the dictionary order. For example, if queried for a record named `cab.example.com`, this NSEC RR signed by the authority of the zone `example.com` (the signature included in a RRSIG(NSEC) RR) is proof (attested by the zone owner) that no such record exists.

4.3.2 DNS-Walk

Even while DNS RRs are not meant to be private they should only be provided when explicitly queried by name and type. NSEC permits one to query random names and learn about unsolicited names of enclosers that *do* exist in the zone master file. For example, a querier may send a query for a random name like `axx.example.com` and get to know the two enclosers `abc.example.com, cat.example.com` that actually exist. The attacker can then query for a random name like `cate.example.com` and obtain its enclosers, say `cat.example.com, data.example.com`, and so on.

The ability to easily enumerate all services under a zone is obviously a useful starting point for any attacker. An attacker wishing to obtain all DNS records for a zone can easily "walk-through" all records in the zone master by simply making a sequence of random queries. Such supercilious queries also have the ill-effect of further burdening the DNS infrastructure.

It is this unintended side effect of providing assurance **A2** that created the need for assurance **A3**. NSEC3 [25], a recent modification to NSEC, employs hashes of names as enclosers instead of using the names themselves as enclosers. For example, an NSEC3 RR indicating a hash encloser (v_l, v_h) indicates that no record with a name y exists, *if* $v_l < h(y) < v_h$. The enclosers v_l and v_h are hashes of real names corresponding to records that do exist in the master file.

Unfortunately, simple dictionary attacks can be used to reveal the names behind hashes like v_l and v_h. Thus, NSEC3 fails in its attempt to provide assurance **A3**.

4.4 MLS Based Alternative to DNSSEC

DNSSEC has seen poor levels of adoption as upgrading a “plain-old” DNS server to support DNSSEC will often necessitate a hardware upgrade due to an order of magnitude increase in the size of DNSSEC records (compared to plain DNS records), and substantial increase in the size of DNS responses [26–28]. In many cases DNSSEC may require more expensive TCP (instead of UDP) as the transport protocol for carrying large DNS responses. LDNSs and stub-resolvers will also need to endure substantial computational burden due to the need to verify multiple digital signatures. Furthermore, the feasibility of zone enumeration also encourages attackers to perform supercilious queries, thus exacerbating the issue of high DNSSEC overhead.

4.4.1 Extending Link-Security Approaches

Cryptographic mechanisms for individually securing each link traversed by DNS records, viz., the links between

1. Offline zone owners and name servers of the zone (for securely conveying master files)
2. LDNSs and zone server
3. Clients and their LDNSs

demand substantially lower overhead compared to the hierarchical PKI-like approach employed by DNSSEC. Unfortunately, link-security approaches implicitly assume that the middle-men are trustworthy: while RRs are protected in transit, there is no protection for RRs while they reside in the DNS servers.

Specifically, symmetric key DNSSEC [21] and DNSCurve [22] simply *assume* that:

1. The keys employed by DNS servers (which are used to compute the link secrets) are well protected from untrustworthy entities (else, any entity with access to the

secrets of a DNS server can impersonate the DNS server to send fake RRs); and that

2. The intermediary DNS servers (i) will not modify RRs, and (ii) will not deny RRs that do exist.

In the proposed alternative approach every DNS server is assumed to house a low complexity trustworthy module (TM). The proposed approach does not make such unjustifiable assumptions regarding the trustworthiness of DNS servers. Instead, the assumptions are:

1. Secrets protected by the TM (which are used to compute link-secrets) cannot be exposed.
2. The trivial functionality of the TM cannot be modified.

As we shall see shortly the functionality inside the TM will merely involve verification and computation if message authentication codes (MAC) using pairwise secrets are facilitated by MLS key distribution. More specifically, we refer to the TM functionality as an "atomic relay" as the TM merely:

1. Accepts a MAC μ for a succinct value v from an "upstream" TM
2. Verifies the MAC
3. Recomputes the MAC for v for verification by a "downstream" TM.

Not withstanding the simplicity of TM functionality, the trust that this functionality can not be modified, and this functionality alone, is bootstrapped to realize substantially improved assurances compared to DNSSEC.

For any system with a desired set of assurances, the trusted computing base (TCB) [29] is a small amount of hardware/software that is trusted to realize the desired assurances. More importantly no component that is not included in the TCB is trusted in order to realize the desired assurances. As only TM functionality is trusted to realize the desired assurances the TMs serve as the TCB for DNS.

4.4.2 Principle of TCB-DNS

In the TCB-DNS protocol [30] every DNS server is equipped with a low-complexity TM. From the perspective of DNS servers, the TMs are black boxes that accept a formatted stream of bits as input, and output a message authentication code (MAC). Such MACs accompany plain DNS responses sent by DNS servers. The operations performed inside the TM (to map the input bits to a MAC) are a fixed sequence of logical and cryptographic hash operations. This simple TM functionality is the TCB of a DNS server, which is leveraged to realize all three assurances **A1**–**A3** with negligible overhead.

4.4.2.1 Atomic Relay

In the path of an RRSet originating from the zone owner Z to the client (a stub-resolver C), is a name server for the zone Z and the LDNS used by the host C. An *atomic relay*, as the name suggests, relays a value from one entity to another, in one atomic step. A TM A in the name server performs an atomic relay of a value V from the zone authority Z to a LDNS TM P, thus eliminating the need to trust the name server in which the TM A is housed. Similarly, the TM P in the LDNS performs an atomic relay of the value V from the ANS TM A to a stub-resolver C, eliminating the need to trust the LDNS.

From the perspective of the TM A, it receives some input bits which specify

1. The identity of the source Z
2. A value V to be relayed
3. A message authentication code (MAC) $M_{V,ZA}$
4. The identity of the entity P to which the value V needs to be relayed.

The TM A does the following:

1. It uses its secrets to compute pair-wise secrets K_{ZA} and K_{AP}.
2. Using the pair-wise secret K_{ZA}, TM A verifies the MAC $M_{V,ZA} = h(V, K_{ZA})$ appended by Z.
3. On successful verification the TM A computes a MAC $M_{V,AP} = h(V, K_{AP})$ using the secret K_{AP}.

The output is the MAC $M_{V,AP}$, which is relayed to P.

The values relayed by TMs are hashes of RRSets. The hashes of RRSets are computed by zone authorities and individually authenticated to each ANS TM using MACs. ANS TMs can atomically relay the hashes to any LDNS TM which can then atomically relay the hash to any stub-resolver. The TMs thus provide a parallel channel for securely conveying hashes of RRSets by leveraging link-secrets (which are computed using secrets protected by the TM).

An atomic relay does not mean that a values received from an upstream entity should be *immediately* relayed. The values can be stored (by the untrusted DNS server) and relayed any number of times, to any number of downstream TMs. The DNS server is free to choose the downstream TM or client.

Note that in both DNSSEC and TCB-DNS end-to-end integrity of an RRSet is realized by securely conveying a preimage resistant hash of the RRSet. In DNSSEC this is achieved by signing the hash. In TCB-DNS the integrity of the hash is assured to the extent we can trust the TMs involved in relaying the hashes.

DNSSEC provides assurance **A1** by signing hashes of regular DNS RRs, and provides assurance **A2** by signing hashes of NSEC/NSEC3 RRs. Obviously, by atomically relaying the hashes of regular RRSets and NSEC/NSEC3 records, TCB-DNS can also provide both assurances provided by DNSSEC. However, a simple addition to the capability of the TMs can provide TCB-DNS with yet another useful feature—the ability to provide assurance **A3**, and thereby eliminate the possibility of DNS-walk.

4.4.2.2 "Intelligent" Atomic Relay

If we merely relay hashes of NSEC/NSEC3 records, then TCB-DNS will also be susceptible to DNS-walk. Fortunately, to realize assurance **A3**, the only additional intelligent feature required of TMs is the ability to recognize that "a value V falls inside an enclosure (V_l, V_h)."

The atomic relay function performed by a TM with identity X, takes the form

$$M_{XD} = \mathcal{A}R_X(S, D, V, V_l, V_h, M_{SX}). \tag{4.2}$$

In executing this TCB function the TM X accepts some fixed-length inputs like:

1. S and D: the identities of a source and destination
2. Cryptographic hashes V, V_l, and V_h
3. A MAC M_{SX} provided by the source S

The TM outputs a MAC for the value V (for verification by the destination D) under two conditions:

1. The MAC M_{SX} is consistent with V; or
2. The MAC M_{SX} is consistent with values $V_l \parallel V_h$, *and* V is enclosed by (V_l, V_h).

In the latter case, the TM interprets a pair of values (V_l, V_h) authenticated by the zone authority as proof that no value enclosed by (V_l, V_h) exists in the master file. If V is enclosed, the TM outputs a MAC for V to inform D that an "enclosure for V was found."

In DNSSEC that a value V is enclosed by (V_l, V_h) is checked by the querier. The need to reveal the enclosures to the querier is the reason that assurance **A3** cannot be provided. In TCB-DNS the enclosure is checked by the ANS TM (**not** provided to the querier). To the extent that the TM can be trusted, the querier trusts that an enclosure exists for the value V (and consequently, is convinced that an RR with the name corresponding to V does not exist).

More specifically, in TCB-DNS,

1. If the queried name and type exists the response includes the desired RRSet in the `ANSWER` section; a MAC for a value V (where V is hash of the RRSet) is also included in the response.
2. To deny a name and type $n_i \parallel t_i$ the `ANSWER` section indicates the name and type $n_i \parallel t_i$; the MAC for the value $V = h(n_i \parallel t_i)$ is included in the response to imply that the indicated name and type does not exist.

Typically, as we shall see in a later section, to provide authenticated denial for a queried name-and-type, a plurality name-and-types will have to be explicitly denied.

4.4.3 Computing Link Secrets

For performing the atomic relay, a TM needs to compute two pairwise secrets—one shared with the sender (the previous hop), which is used to verify the hash V (or the encloser (V_l, V_h) for authenticated denial), and one shared with the destination (the next hop). Specifically,

1. ANS TMs require the ability to establish a pairwise secret with (i) the zone owner for receiving hashes of RRSets, and (ii) all LDNS TMs for securely conveying hashes of RRSets.
2. LDNS TMs require the ability to establish a pairwise secret with (i) all zone server TMs, for receiving hashes, and with (ii) all clients who employ the LDNS, for conveying the hashes.

We can thus interpret all participants in DNS as a network $\mathcal{N}$ where $\mathcal{N}_1 = \mathcal{N}$ includes all participants, and $\mathcal{N}_2$ includes all DNS servers. MLS/IT can be used to facilitate pairwise secret between all (TMs in) DNS servers, between clients and (TMs in) LDNSs and between zone owners and (TMs in) ANSs. We shall assume that MLS is used.

4.4.3.1 Key Distribution for TCB-DNS

In TCB-DNS the KDC can be entity under the control of a regulatory authority (for example, ICANN). The *core* TCB-DNS entities are TMs associated with DNS servers. The *fringe* TCB-DNS entities include zone authorities (who need to securely convey RRs to ANSs) and clients (stub-resolvers) who query DNSs. Pair-wise secrets for TCB-DNS can be (i) between two core entities (between two TMs), or (ii) between a core entity and a fringe entity.

For the former case, a sequence number included in the TM identity specifies which of the two core entities should use the public value to compute the pairwise secret. For the latter (pairwise secret between a core entity and a fringe entity) the fringe entity employs the public value—the core entity does not.

The identity X of a TM (associated with an ANS or an LDNS) is of the form $X = X_t \parallel q_x$ where X_t is a succinct code describing the nature of X and duration of validity; the value q_x is a unique number assigned sequentially to every DNS server TM. To establish a secret between TMs $X = X_t \parallel q_x$ and $Y = Y_t \parallel q_y$ where (say) $q_x < q_y$, Y is required to use the value P_{XY} to compute the pairwise secret K_{XY}; X can compute K_{XY} directly using its secret K_X.

The TCB-DNS identity Z of a zone authority is of the form $Z = Z_t \parallel Z_{name}$ where Z_{name} is a one-way function of the domain name of the zone. To enable Z to compute a pairwise secret K_{ZA} with an ANS TM A the zone authority is issued:

1. A secret $K_{AZ} = h(K_A, Z)$ by the KDC
2. A secret K_Z, along with a public value P_{ZA}
3. A TM with identity Z (with secret K_Z stored inside the TM), along with a public value P_{ZA}.

In the TCB-DNS identity of a client $C = C_t \parallel C'$, C' can be a unique random value. If P is the identity of a TM in an LDNS used by the client C the client C is issued (i) a secret $K_{PC} = h(K_P, C)$ or (ii) a secret K_C and a public value P_{CP}.

Thus, once keys have been distributed to TCB-DNS entities, computing any link-secret will require the TMs to only perform a single hash computation. Periodically, the KDC disseminates signed revocation lists indicating identities of entities revoked.

4.4.3.2 Multiple KDCs and Renewal

At the top of the hierarchy of DNSSEC is a single root CA—which is the authority for the root zone. Though the root zone is expected to sign only public keys for gTLD and ccTLD zones, the all powerful root zone authority has the ability to misrepresent public keys for *any* zone. While ideally we would desire that this power be distributed among multiple independent entities, such an approach can further increase the overhead for DNSSEC.

In the case of MLS, If we use m (for example, $m = 4$) independent KDCs, all that changes is that computation of any pairwise secret will call for $m = 4$ hash operations. We shall use the notation $K_{XY} = F(Y, P_{XY})$ to represent the process of computing the pairwise secret K_{XY} by entity (or TM) X.

For renewal of secrets of a TM $X = X_t \parallel q_x$ the regulatory authority simply issues a new TM with TCB-DNS identity $X' = X'_t \parallel q'_x$, with secrets $K_{X'_1} \cdots K_{X'_m}$. If at the time of renewal, the last issued sequence number was q, the new TM is issued a sequence number $q'_x = q + 1$. The owner of the TM is issued q public values (where each public value is the XOR of m public values). If the secrets of an ANS TM A is renewed, only the zone authorities using the ANS need to be issued new public values for A. If the TM P of a LDNS is renewed, only the clients who use the LDNS are issued with new public values corresponding to P.

More specifically, a node with sequence number q is the q^{th} node to join the network, and is issued one secret and $q - 1$ public values (or m secrets and $q - 1$ public values if we use multiple KDCs). For renewal we simply add a new node. The public values are the same size as the pair-wise keys (say 160-bits). A DNS server with a TM sequence number 10 million will need access to at most 200 MB of storage for public values (which can easily be stored in the hard-disk of the DNS server). There is no practical limit on the number of fringe entities (zone authorities and clients). Each fringe entity requires access only to a small number of public values (as they need to establish a pairwise secret only with a small number of core entities—zone authorities with TMs of all ANSs for the zone, and clients with all its LDNSs).

4.5 The TCB-DNS Protocol

This section begins with a detailed specification of the atomic relay algorithm. This is followed by an outline of the operation of TCB-DNS by describing the steps for creating TCB-DNS master files (in Sect. 4.5.2) and illustrating the sequence of events in typical a query-response process (in Sect. 4.5.3).

4.5.1 The Atomic Relay Algorithm

A TM with identity X stores a secret K_X inside its protected boundary—which is known only to TM X and the KDC. To relay a value from S to D the TM requires to compute secrets K_{XS} and K_{XD}. For this purpose the TM needs two additional inputs—public values P_{XS} and P_{XD}. Thus, the atomic relay function of a TM X takes the form

$$M_{XD} = \mathcal{A}R_X((S, P_{XS}), (D, P_{XD}), V, V_l, V_h, M_{SX}).$$

In a scenario where X does *not* require to use a public value to compute K_{XS}, the input $P_{XS} = 0$ is provided to the TM (as XOR-ing by 0 leaves a value unchanged). It is the responsibility of the (untrusted) DNS server to store and provide appropriate public-values to its TM; if a DNS server provides incorrect public values to its TM the MAC will be rejected by the next-hop[5] which verifies the MAC.

The TM X accepts a formatted stream of bits $\mathbf{b}_i = (S \parallel P_{XS}) \parallel (D \parallel P_{XD}) \parallel V \parallel V_l \parallel V_h \parallel M_{SX}$ as input from the DNS server which houses the TM; the TM performs a simple sequence of logical and cryptographic hash operations, and outputs a MAC M_{XD} or a fixed constant `ERROR`.. An algorithmic description of the sequence of operations is depicted in Fig. 4.2.

As shown in the algorithm in Fig. 4.2, the TM computes the pairwise secret K_{XD} for authenticating TM output to destination D. If $S = X$ (source is indicated as the TM itself), the TM construes this as a request to output a MAC $h(V \parallel K_{XD})$ verifiable by D. This feature, as we shall see soon, permits zone owners to use DNS TMs.

In general (for $S \neq X$) the TM proceeds to compute the pairwise secret K_{XS} required for validating the inputs (V, V_l and V_h authenticated by source S using a MAC M_{SX}:

1. If V_l is zero the TM verifies that the MAC M_{SX} is consistent with V and K_{XS}.
2. If the value V_l is non-zero, the TM verifies that (i) the input MAC M_{SX} is consistent with the two values $(V_l \parallel V_h)$, *and* (ii) that V is *enclosed* by (V_l, V_h). A value V

[5] If the next-hop is a LDNS, when an invalid response is received, the LDNS will send the query again or query another ANS. Similarly if the next-hop is a stub-resolver C, then C will resend the query or query another LDNS.

Fig. 4.2 The atomic relay algorithm $\mathcal{A}R_X()$. K_{XS} and K_{XD} are pair-wise secrets that X shares with TCB-DNS entities S and D respectively

```
A $R_X(S,P_{XS},D,P_{XD},V,V_l,V_h,M_{SX})$ {
$K_{XD} = F(P_{XD},D)$;
IF $(S == X)$;
  RETURN $h(V \| K_{XD})$;
$K_{XS} = F(P_{XS},S)$;
IF $(V_l == 0)$
   $V_i = V$;
ELSE IF $(((V_l < V) \wedge (V < V_h)) \vee ((V > V_l) \wedge (V_l > V_h)))$
   $V_i = h(V_l \| V_h)$;
ELSE RETURN ERROR;
IF $(M_{SX} != h(V_i \| K_{SX}))$;
   RETURN ERROR;
RETURN $M_{XD} = h(h(S \| V) \| K_{XD})$;
}
```

is enclosed by (V_l, V_h) if $V_l < V < V_h$. If $V_l > V_h$ then V is enclosed by the "wrapped around" pair if $V > V_l > V_h$ or $V < V_h < V_l$.

On successful verification the TM outputs a MAC for the value $(S \| V)$ computed using the pairwise secret K_{XD} between X and D.

For ease of following the discussion in the rest of this section, note that

$$\begin{aligned} M_{ZA,V} &= \mathcal{A}R_Z(Z,0,A,P_{ZA},V,0,0,0) \\ &= h(V \| K_{ZA}) \end{aligned} \tag{4.3}$$

is a MAC for a value V computed by a TM Z (for verification by a TM A). We shall see soon that zone authorities can employ TMs in this fashion to authenticate hashes of RRSets for verification by ANS TMs. Also note that

$$\begin{aligned} M_{AP,V_Z} &= \mathcal{A}R_A(Z,0,P,P_{AP},V,0,0,M_{ZA,V}) \\ &= h(h(Z \| V) \| K_{AP}) \end{aligned} \tag{4.4}$$

is a MAC computed by TM A which can be verified by an entity P. The MAC represents A's claim that "a value V was received from Z." If the MAC is verifiable, to the extent P trusts A, P can accept the claim that the value V was provided by Z.

Finally,

$$\begin{aligned} M_{AP,V'_Z} &= \mathcal{A}R_A(Z,0,P,P_{AP},V',V_l,V_h,M_{ZA,V'}) \\ &= h(h(Z \| V) \| K_{AP}) \end{aligned} \tag{4.5}$$

is also a MAC verifiable by an entity P; on successful verification, P concludes that "a value V was received from Z." P does not need to differentiate between the two cases. In the former case, V was explicitly conveyed to A by Z through a MAC $M_{V,A}$. In the latter case, V is any value, not explicitly conveyed by Z, but happens to fall within an enclosure (V_l, V_h) (and the enclosure is authenticated by Z using MAC $M_{V',A}$).

4.5.2 Preparation of TCB-DNS Master File

Consider a zone `example.com`, which employs ANSs with TMs A and B for the zone. The sequence of steps performed by the zone authority to prepare a master file are as follows. Let the TCB-DNS identity of the zone be Z where $Z = Z_t \parallel Z_{name}$, where Z_{name} is the hash of the name of the zone (`example.com`). Recall that Z_t includes a succinct representation of the time of expiry of the secrets assigned to Z.

Step 1 Prepare a regular plain DNS master file. Some of the required additions to plain DNS RRs are as follows:

a) Each RR will indicate an absolute value of time as the time of expiry. This value can be a 32-bit value like UNIX time, and can be different for each RR. In general the time of expiry of any RR should not be later than Z_t.

b) NS-type RRs (which indicate the name of an ANS) includes two additional values:

(i) The TCB-DNS identity of the ANS-TM

(ii) The value Z_t (note that from the name of the zone in the NS RR, one can compute the value Z_{name}; along with the value Z_t the TCB-DNS identity of the zone can be computed as $Z = Z_t \parallel Z_{name}$).

In general, a RRSet **R** has multiple RRs with the same name and type, and each RR indicates its own a time of expiry.

Step 2 Let r be the total number of RRSets. For an RRset **R** with name n_j and type t_j compute (i) the hash of the RRSet $v_j = h(RRSet)$; and (ii) $u'_j = h(n_j \parallel t_j \parallel \tau)$, where τ is the time at which the authentication for all enclosures expire. Repeat for all r RRSets.

Step 3 Sort the hashes $u'_1 \cdots u'_r$ in an ascending order; Let the sorted hashes be $u_1 \cdots u_r$. Now, compute values $d_1 \cdots d_r$ as

$$d_j = \begin{cases} h(u_j \parallel u_{j+1}) & j < r \\ h(u_r \parallel u_1) & j = r \end{cases} \tag{4.6}$$

Note that for the last "wrapped around" enclosure (u_r, u_1) the first value u_r is greater than the second (u_1).

Step 4 For each of the $2r + 1$ values in $\{v_1 \cdots v_r, d_1 \cdots d_r, \tau\}$ compute MACs $M_{ZA,i} = h(v_i \parallel K_{ZA}), 1 \leq i \leq r$, $M'_{ZA,j} = h(d_j \parallel K_{ZA}), 1 \leq j \leq r$, and $M_{ZA,\tau} = h(\tau \parallel K_{ZA})$. If the zone authority Z employs a TM Z then a MAC like $M_{ZA,i} = h(v_i \parallel K_{ZA})$ is computed by using the atomic relay function of the TM as

$$\begin{aligned} M_{ZA,V} &= \mathcal{A}R_Z(Z, 0, A, P_{ZA}, v_i, 0, 0, 0) \\ &= h(v_i \parallel K_{ZA}). \end{aligned} \tag{4.7}$$

Prepare a supplementary master file with

a) The values $(\tau, M_{ZA,\tau})$

b) r rows of the form $(i, M_{ZA,i})$, $1 \leq i \leq r$

c) r rows of the form $((u_j, u_{j+1}), M'_{ZA,j})$, $1 \leq j \leq r$

Step 5 Provide the supplementary master file to ANS with TM A along with the regular master file. The zone authority repeats step 4 for ANS B to create a supplementary master file with values $(\tau, M_{ZB,\tau})$; r rows $(i, M_{ZB,i})$, and r rows $((u_j, u_{j+1}), M'_{ZB,j})$.

4.5.3 Verification of RRSets

We shall consider a scenario where an ANS with TM A is queried for an RRSet `cad.example.com,A` by a LDNS with TM P. Let us further assume that the query was initiated by a stub-resolver C.

4.5.3.1 Events at ANS with TM A

Let the identities A and P of the TMs be $A = A_t \parallel q_a$ and $P = P_t \parallel q_p$. If $q_p < q_a$ (sequence number of P is less than that of A) the ANS fetches P_{AP} from storage (else $P_{AP} = 0$). If the queried name and type (n_i, t_i) exists, or if a suitable delegation exists, the ANS

1. Extracts the RRSet from the plain DNS master file, and computes the hash of the RRSet, v_i
2. Extracts the MAC $M_{ZA,i}$ for v_i from the supplementary master file
3. Requests TM A to compute

$$\begin{aligned} M_{AP,i_1} &= \mathcal{A}R_A((Z,0),(P,P_{AP}),v_i,0,0,M_{ZA,i}) \\ &= h(h(Z \parallel v_i), K_{AP}). \end{aligned} \tag{4.8}$$

In the response sent to the LDNS, the ANS includes the RRSet in the `ANSWER` section along with the value M_{AP,i_1}. If the response is a delegation, the NS RRSet can be included in the `AUTHORITY` section along with the value M_{AP,i_1}. The TM A does not know, or care, if the response is an answer or a delegation.

To deny a name-and-type (n_i, t_i),

1. ANS extracts the values $(\tau, M_{\tau,ZA})$ from the supplementary master file for zone Z.
2. ANS computes $v_i = h(n_i \parallel t_i \parallel \tau)$.
3. ANS finds encloser for v_i (say (u_j, u_{j+1})), and corresponding MAC $M_{ZA,j}$ from the supplementary master file.
4. ANS requests TM A to compute M_{AP,τ_1} and M_{AP,i_1} as

$$\begin{aligned} M_{AP,\tau_1} &= \mathcal{A}R_A((Z,0),(P,P_{AP}),\tau,0,0,M_{ZA,\tau}) \\ &= h(h(Z \parallel \tau), K_{AP}) \\ M_{AP,i_1} &= \mathcal{A}R_A((Z,0),(P,P_{AP}),v_i,u_j,u_{j+1},M_{ZA,j}) \\ &= h(h(Z \parallel v_i), K_{AP}). \end{aligned}$$

For reasons that will be explained later in Sect. 4.6.2, typically the ANS will need to deny multiple name-and-type values in a response. Let us assume that q name-and-type values need to be denied. For each such name-and-type (n_l, t_l) the ANS computes $v_l = h(n_l \parallel t_l \parallel \tau)$, finds an encloser for v_l and the corresponding MAC in the supplementary master file, and requests the TM to compute MACs of the form M_{AP,l_1} (each of the q requests are made independently—each request results in the use of the atomic relay function $\mathcal{A}R_A()$ by the TM A).

In the response sent to the LDNS the ANS includes (in the `ANSWER` section):

1. Values τ and M_{AP,τ_1}
2. q denied name-and-type values $n_i \parallel t_i$
3. q MACs of the form M_{AP,i_1}

4.5.3.2 Events at the LDNS with TM P

Before the LDNS had sent a query to the ANS for a name and type belonging to zone Z, the LDNS would have queried an ANS for the parent zone of Z—say $W = W_t \parallel W_{name}$, and obtained an NS-type RRSet for the name Z_{name} (where $Z = Z_t \parallel Z_{name}$).

Let us further assume that the NS-type RRSet was authenticated by a TM $G = G_t \parallel q_g$ (housed in an ANS for the zone W). In other words, the LDNS would have received a value M_{GP,j_1} to authenticate the NS-type RRSet, where

$$M_{GP,j_1} = h(h(W \parallel v_j), K_{GP}), \tag{4.9}$$

and, v_j is the hash of the NS-type RRSet.

In TCB-DNS, the LDNS is expected to verify the NS RRSet before sending a query to the delegated server. In this case, where the LDNS had chosen to approach the ANS A for the zone Z (based on the information included in the NS-type RRSet authenticated by G) the LDNS computes $v_{j_1} = h(W \parallel v_j)$, and requests its TM P to compute

$$x = \mathcal{A}R_P((G, P_{GP}), (A, P_{PA}), v_{j_1}, 0, 0, M_{GP,j_1}). \tag{4.10}$$

As long as $x \neq ERROR$, the LDNS considers the NS records to be valid.

Similarly, prior to querying G (ANS for W, the parent of Z) the TM would have received a response from an ANS for the parent of W (unless W is the root zone which has no parent—in our case W is the gTLD zone `.com`). Such a response from an ANS of W's parent zone would have also been verified as above before a query was sent to G. Thus, after the response from the parent zone W was verified, the LDNS P had sent a request to A for a name and type under zone Z.

TCB-DNS does not require queries to be authenticated. Queries merely indicate the TCB-DNS identity of the querier.

Now, after the response is received from A, the LDNS P has all the necessary information to send the answer to the stub-resolver C which initiated the query.

Typically, the LDNS will need to include an RRSet in the `ANSWER` section (along with a MAC computed by its TM P). For responses containing authenticated denial for q name-and-types the response will include $q+1$ values authenticated individually using $q + 1$ MACs. For both cases, an NS-type RRSet will be included in the `AUTHORITY` section indicating ANS for the zone, along with a MAC computed by the TM P.

More specifically, the hash of the RRSet in the `ANSWER` section is relayed atomically from A to C, by P. Similarly, for authenticated denial, the q hashes corresponding to multiple nonexisting names, and the value τ, are relayed atomically from A to C by P. The hash of the NS-type RRSet is however relayed atomically by P from G to C.

For example, to relay the RRSet with hash v_i received from A the LDNS first hashes the RRSet to obtain v_i and requests its TM to compute

$$\begin{aligned} M_{PC,i_2} &= \mathcal{A}R_P((A, P_{PA}), (C, 0), v_{i_1}, 0, 0, M_{AP,i_1}) \\ &= h(h(A \parallel v_{i_1}) \parallel K_{PC}) \\ &= h(h(A \parallel h(Z \parallel v_i)) \parallel K_{PC}). \end{aligned} \tag{4.11}$$

If the hash of RRSet v_i computed by the LDNS is not the same as the one authenticated by the zone authority Z, the MAC M_{AP,i_1} will be found inconsistent by the TM P, which will return $ERROR$.

Similarly, to relay the NS-type RRSet received from G along with a value M_{GP,j_1}, the LDNS hashes the RRSet to obtain v_j, and uses its TM P to compute[6]

$$\begin{aligned} M_{PC,j_2} &= \mathcal{A}R_P((G, P_{PG}), (C, 0), v_{j_1}, 0, 0, M_{GP,j_1}) \\ &= h(h(G \parallel v_{j_1}) \parallel K_{PC}) \\ &= h(h(G \parallel h(W \parallel v_j)) \parallel K_{PC}). \end{aligned} \tag{4.12}$$

The response from the LDNS to C thus includes:

1. The NS-type RRSet (with hash v_j) for Z along with the values W, G and M_{PC,j_2} in the `AUTHORITY` section, AND
2. The queried RRSet with hash v_i, along with a MAC M_{PC,i_2}, and the identity A of the ANS, OR
3. Authenticated denial of q name-and-type values (by including $q + 1$ values and $q + 1$ MACs), and the identity A of the ANS.

If the parent zone W does not support TCB-DNS (the ANS for W is not equipped with a TM) then the NS RRSet is relayed without any TCB-DNS authentication.

[6] The value W is obtained from the NS type RRSet for the parent zone W, which was obtained by querying W's parent—the root.

4.5.3.3 Verification by the Stub-Resolver *C*

The stub resolver performs the following steps:

1. It extracts name of zone from the `AUTHORITY` section; hashes name to compute Z_{name} and hence $Z = Z_t \parallel Z_{name.}$
2. If the NS RRSet in the `AUTHORITY` section has TCB-DNS authentication
   a) Client C computes the hash v_j of the NS-type RRSet in the `AUTHORITY` section and verifies that $M_{PC,j_2} = h(h(G \parallel h(W \parallel v_j)) \parallel K_{PC})$.
   b) Parses W as $W = W_t \parallel W_{name}$ and verifies that W_{name} is a legitimate parent of Z_{name}.
3. C verifies that Z_{name} is a legitimate parent of the queried name.
4. C verifies that name of the zone is a parent of the queried name[7] in the `ANSWER` section.
5. If the `ANSWER` is the desired response, hash the RRSet to compute v_i; compute $v_{i_1} = h(Z \parallel v_i)$, $v_{i_2} = h(A \parallel v_{i_1})$, and using key K_{CP} verify that $M_{PC,i_2} = h(v_{i_2} \parallel K_{PC})$.
6. If the `ANSWER` is an authenticated denial indicating q values of the form $n_i \parallel t_i$, for each of the q values compute $v_i = h(n_i \parallel t_i \parallel \tau)$, $v_{i_1} = h(Z \parallel v_i)$, $v_{i_2} = h(A \parallel v_{i_1})$, and verify that $M_{PC,i_2} = h(v_{i_2} \parallel K_{PC})$.

RRs which have expired (as indicated by time-of-expiry field added to each RR in an RRSet) will be ignored. If the `ANSWER` section indicate authenticated denial and the value τ smaller than the current time, the response is ignored. If any of the TMs A and P and G involved in relaying the RRSets has been revoked by the KDC, the RRSet is ignored.

4.5.4 Proof of Correctness

Consider a scenario where the verifier C determines that the set of values $\{Z, A, v_i, M_{PC,i_2}\}$ satisfy

$$M_{PC,i_2} = h(h(A \parallel h(Z \parallel v_i)) \parallel K_{PC}). \tag{4.13}$$

In concluding that the RRSet (with hash v_i) in the `ANSWER` section was indeed created by the zone authority Z (as indicated in the `AUTHORITY` section), TCB-DNS assumes that:

1. The integrity of TMs A and P: more specifically, (i) secrets assigned to TMs are not privy to other entities, and (ii) the atomic relay function cannot be modified.

[7] Just as there is nothing that stops an authority of `example.com` from signing an RRSet for `www.yahoo.com` in DNSSEC, in TCB-DNS a zone authority can authenticate any value. However, resolvers will not accept RRSet as valid as $Z_{name} = h($`example.com`$)$ is not a parent of `www.yahoo.com`.

2. The keys of the zone authority Z (possibly protected by a TM Z) are not privy to anyone else.
3. The hash function $h()$ is preimage resistant.

With these assumptions, it is easy to see that:

1. As the hash function $h()$ is preimage resistant, the value M_{PC,i_2} was computed by an entity with access to the secret K_{PC} (thus the verifier can conclude that the value M_{PC,i_2} was computed by TM P).
2. The TM P can compute M_{PC,i_2} *only* if it was provided values $v_{i_1} = h(Z \parallel v_i)$ and M_{AP,i_1}, satisfying $M_{AP,i_1} = h(v_{i_1} \parallel K_{AP})$.
3. Only TM A could have computed the value M_{AP,i_1} provided to P.
4. TM A can compute M_{AP,i_1} only if it was provided values $\{v_i, M_{ZA,v_i}\}$ satisfying $M_{ZA,v_i} = h(v_i \parallel K_{ZA})$.
5. As only Z has access to secret K_{ZA}, the value v_i was created by Z.

Note that to conclude that "value v_i was indeed created by Z," it is *not* necessary that the parent zone W supports TCB-DNS. However, it is indeed desirable that all zones adopt TCB-DNS. If the parent zone also supports TCB-DNS, then the client can also verify the integrity of the NS RRSet for zone Z, and thereby confirm that A is indeed a TM associated with an ANS for the delegated zone Z.

4.6 Practical Considerations

TCB-DNS can be implemented with minimal modifications to current DNS servers. The specific modifications required to support TCB-DNS are as follows:

1. Every RRSet will indicate an absolute time of expiry (say, 32-bit UNIX time) specified by the zone authority; this value is unrelated to the TTL value[8] specified in each RR.
2. Each NS record will indicate the TCB-DNS identity of the ANS TM (this is similar to the requirement in DNSCurve where the elliptic-curve public key of the ANS is indicated in the NS record).
3. DNS queries will indicate an additional field—the TCB-DNS identity of the querier.

If an NS record for a zone W provided by a parent zone does not indicate the identity of a TM, the implication is that the indicated ANS for the zone W does not support TCB-DNS. It is also possible for a zone to employ as its ANSs, some TCB-DNS aware servers and some plain DNS servers. The NS records corresponding to TCB-DNS compliant ANSs will indicate the TCB-DNS identity of the ANS. NS records corresponding to plain DNS servers will not. Thus, a LDNS which supports TCB-DNS may prefer to query a TCB-DNS compliant ANS for the zone

[8] The TTL value specifies how long an RR can be cached by resolvers.

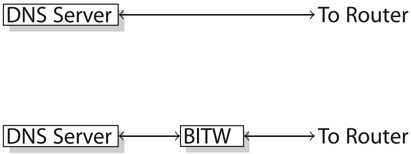


Fig. 4.3 Top: original configuration. Bottom: bump-in-the-wire (BITW) implementation

W. Similarly, a plain DNS LDNS may choose to direct its query to a plain DNS ANS for zone Z.

If a TCB-DNS server receives a query which does not indicate the TCB-DNS identity of the querier, the querier is assumed to be unaware of TCB-DNS. In this case a plain DNS record is sent as a response. If a TCB-DNS unaware server is queried by a TCB-DNS compliant resolver the DNS server will simply ignore the additional field (Fig. 4.3).

TCB-DNS can easily support bump-in-the-wire implementations. Converting a plain DNS server to TCB-DNS server can be as simple as adding an additional BITW unit equipped with a DNS-TM. Only the BITW unit will need to have access to the TCB-DNS supplemental master file. The BITW unit will

1. Verify TCB-DNS authentication appended to incoming DNS packets, strip authentication, and relay plain DNS packets to the DNS server
2. Append TCB-DNS authentication to outgoing DNS packets.

4.6.1 TCB-DNS vs. DNSSEC

The main reasons for the poor adoption of DNSSEC are

1. The significant increase in the size of zone files (over that of plain-old DNS) due to the addition of RRSIG, DS, DNSKEY, and NSEC/NSEC3 records
2. Increased bandwidth overhead for DNSSEC responses
3. The susceptibility of DNSSEC to DNS-walk

Due to the substantial overhead, it is especially expensive for large zones (for example, `.com`) to adopt DNSSEC.

DNSSEC and TCB-DNS have many significant similarities:

1. Both do not require DNS servers and their operators to be trusted. Both protocols achieve this requirement by their ability to securely convey a preimage resistant one-way function of RRSets created by zone owners to end-points (clients), without the need to trust the intermediary servers.
2. In both protocols lifetimes are imposed on the validity of zone keys. Both specify validity periods for the authentication appended for RRs (which can at most be till the expiry of the keys used for validation).
3. Both use a strategy for ordering names (or a one-way function of names) to provide authenticated denial of enclosed names (or hashes of names).

4. Both protocols do *not* possess a mechanism for revoking authentication. Consequently both protocols are susceptible to replay attacks—under some conditions. If the authentication appended for an RRSet is indicated as valid till some time t, and if for some reason, there arises a need to modify the RRSet before time t, then an attacker may be able to replay the old RRSet (with a signature valid till time t) until time t.

The primary differences between DNSSEC and TCB-DNS include:

1. The cryptographic mechanisms employed - DNSSEC relies on digital signatures, while TCB-DNS relies on TMs to atomically relay MACs
2. Unlike TCB-DNS, DNSSEC does not provide assurance **A3**. Some of specific differences in the mechanism for authenticated denial, and the rationale for the choices made in TCB-DNS are outlined in Sect. 4.6.2.
3. DNSSEC demands substantially higher overhead compared to TCB-DNS; Sect. 4.6.3 provides a comparison of the storage bandwidth overheads of DNSSEC and TCB-DNS.
4. DNSSEC is more susceptible to replay attacks compared to TCB-DNS; Sect. 4.6.4 outlines the reasons for this phenomenon.

Since the discovery of the Kaminsky attack [31] the need to secure DNS has attracted renewed attention. Some modifications have been proposed to DNSSEC to address the main reasons for its poor adoption. However, while such efforts reduce some of the overhead for DNSSEC (and thereby reduce the resistance to adoption of DNSSEC), they are at the expense of watering-down some of the originally intended assurances of DNSSEC.

In Sect. 4.6.5 and we discuss such a mechanism, TSIG [32], which can reduce overhead for clients, but has the unfortunate side-effect of requiring to trust the LDNSs. In Sect. 4.6.6 we discuss another modification (NSEC3 opt-out) [33] which is intended to facilitate easier adoption of DNSSEC by large zones like `.com`. This feature has an unfortunate side effect of interfering with the ability to provide authenticated denial. More recently, some attacks that exploit the NSEC3 opt-out feature have also been demonstrated [34].

4.6.2 Authenticated Denial

Consider a scenario where the ANS for the zone `example.com` is queried for a nonexistent record "`a.b.example.com, A`." A negative response indicates that:

1. The queried name does not exist
2. No wild-card name like `*.b.example.com` exists
3. No delegation exists for a zone `b.example.com`
4. No alias (type CNAME) record exists for the name `a.b.example.com`.

In NSEC the enclosures are textual strings indicating names. A single NSEC record [`example.com` [A,MX,NS], NSEC, `cat.example.com`] is adequate for the resolver to verify that all the above conditions are true (all names that have to be denied fall within the single NSEC enclosure).

In NSEC3 the enclosures are hashes of names. Each NSEC3 enclosure can only be used to deny a *specific* name (which hashes to a value inside the encloser). Thus, proof of enclosure of three or four different name-hashes have to be provided to the resolver.

If a record of a type different from NS does exist for `b.example.com` or if a record with name `a.b.example.com` does exist (but not the solicited type A), then the NSEC3 record has to indicate the list of types that *do* exist.[9]

Though intended as an improvement over NSEC, in some ways NSEC3 is actually inferior to NSEC. In response to a query for a nonexistent record, NSEC revealed two unsolicited names; NSEC3 typically reveals six hashes corresponding to six unsolicited name-hashes (which are subject to brute-force attacks). Furthermore, three RRSIG(NSEC3) signatures have to be verified (instead of one RRSIG(NSEC)).

The mechanism in TCB-DNS for authenticated denial is closer to NSEC3 than NSEC. The difference is that in TCB-DNS the name and type are hashed together (in NSEC3 only the name is hashed). As with DNSSEC-NSEC3, multiple name-and-type hashes will have to be denied by the ANS by using different enclosures. At first sight, it may seem that an NSEC-like approach may be preferable for TCB-DNS. After all, if only the TM is privy to the enclosures—viz., textual strings (names) in NSEC and hashes (of names) in NSEC3, there is no need for hashing names. However, checking NSEC enclosures will require TMs to compare variable length text-strings, possibly of different formats (for example, ASCII, Unicode), which can substantially increase the complexity of TMs. With the NSEC3-like approach only fixed-length hashes need to be compared. Thus,

1. In DNSSEC with NSEC3 the purpose of hashing is to "hide" names (albeit ineffectively)
2. In TCB-DNS the purpose of hashing the names is *not* to hide the names; instead, it is intended to lower the TM complexity.

In TCB-DNS, the reason for hashing name-and-type together is to ensure that (unlike NSEC3) names do not have to be disclosed if queried for a nonexistent type. In TCB-DNS the number of encloser pairs equals the number of unique name-and-type values (which is the same as the number of RRSets). In NSEC3 the number of hashes correspond to the number of unique names.

The disadvantage of TCB-DNS is a small increase the number of encloser pairs. However, this is not a problem in practice. The increase in the number of encloser pairs would only be an issue for zones which have a very large number of names, *and* many types corresponding to each name. However, such large zones

[9] Thus, there are two ways in which NSEC3 fails to realize assurance **A3**: (i) by being susceptible to simple dictionary attacks; and (ii) by disclosing unsolicited types for a name.

Table 4.1 Comparison of TCB-DNS and DNSSEC

		Overhead in bytes		
	Bandwidth	Cache per RRSet	Cache per name	Assurances
DNSSEC	2000	200	300	**A1**, **A2**
TCB-DNS	100	24 + 60		**A1**, **A2**, **A3**

(like gTLD `.com`) which have large number of names, have only a single type (type NS) corresponding to almost every names.

Another difference between NSEC3 and TCB-DNS is the mechanism used for hashing names. In TCB-DNS the hash is computed as a function of name, type, and a value τ. The value τ is the time of expiry of the authentication. In DNSSEC-NSEC3 the time of expiry is indicated in the RRSIG record; the name-hash is computed after including a salt to the name. Furthermore, repeated hashing is employed to derive the name-hash. The reason for using the salt is to prevent *precomputed* dictionary attacks. The purpose of repeated hashing is to increase the computational complexity for dictionary attacks. As dictionary attacks are not possible in TCB-DNS (as the enclosers are never sent), TCB-DNS does not need to deliberately increase the computational overhead for generating name-hashes.

4.6.3 Overhead

Table 4.1 provides a quick comparison of TCB-DNS and DNSSEC. The large size of DNSSEC records is due to the fact that public keys and signatures are large (1000 to 2000 bits). This increases the cache memory requirements for name servers. The longer RR sizes, and that multiple RRSIGs, DS and DNSKEY records need to be fetched and verified, results in substantial bandwidth overhead for responses.

For a typical query, the DNSSEC specific bits that accompany the response (over and above the plain DNS records) can easily be of the order of 2000 bytes. As described in Sect. 4.3, the additional DNSSEC specific records required to verify a RRSet include (typically) one RRSIG(RR), 3 DNSKEY records, 3 DS records, and 3 RRSIG(DS) records. Additionally, verification of an RRset requires the computational overhead for verification of 4 signatures.

For TCB-DNS the additional TCB-DNS specific bits that accompany the response (to a typical query) include (i) the TM identity of the ANS and (ii) one MAC in the `ANSWER` section, and two identities (TCB-DNS identity of the parent zone, and identity of an ANS TM of the parent zone) and one MAC in the `AUTHORITY` section. If the identities of TMs are 10 bytes long and identities of zones are 20 bytes long, and all MACs are 20 bytes long, the additional bandwidth overhead is of the order of 70 bytes. If we consider the additional values inserted in NS records the overhead may be close to 100 bytes (compared to 2000 bytes for DNSSEC).

For DNSSEC the increase in cache memory size for any RRSet is due to the addition of one RRSIG for every RRSet (about 200 bytes for every RRSet if 1600-bit RSA modulus is used). Additionally (for authenticated denial), corresponding to every unique name in the master file, one NSEC/NSEC3 record along with an RRSIG(NSEC/NSEC3) record are required: amounting to an overhead of roughly 300 bytes for every unique name in the master file.

In TCB-DNS, corresponding to every RRSet two additional values are required for regular responses - an index (of the RRSet within the master file) and a MAC. For authenticated denial, corresponding to every unique name and type (the total number of which is the same as the number of RRSets) three values are required: two hashes (enclosers), and a MAC for the enclosure. Assuming 20 byte hashes and MACs, the overhead is about 60 bytes for every unique name and type.

4.6.4 Replay Attacks

The fact that anybody can obtain verify a digital signature is a powerful feature of digital signatures. This power can also be abused more easily when no mechanism exists for revocation. A signed packet with prematurely invalidated contents can be more easily abused, compared to a packet authenticated using a MAC.

A DNS RRSet signed by the zone authority can be sent by anyone, from any place, to any place. However, in TCB-DNS, a MAC appended by a zone authority is intended only for the TM of a specific ANS. This implies that only the entity with control of the specific ANS (who has access to the TM) can replay such packets. This substantially reduces the *scope* of possible replay attacks.

For both protocols, reducing the scope of replay attacks requires choice of short life-times for signatures (MACs for TCB-DNS). Unfortunately shorter life-times imply more frequent re-computation of authentication. Due to the substantially lower computational overhead required for TCB-DNS we can can actually afford to recompute MACs more frequently.

4.6.5 DNSSEC with TSIG

As originally intended, DNSSEC provides the end-points with the ability to verify the integrity of RRs. For most clients this is a substantial computational burden. This is especially true for an ever increasing number of battery operated mobile devices. Furthermore, to receive the large number of DNSSEC specific records from the LDNS the clients may have to employ more expensive TCP instead of UDP as the transport layer.

It is for this reason that in most standard installations of DNSSEC the verification of RRs is performed only by the LDNS. Stub-resolvers are expected to establish a secure channel with LDNSs using some light-weight mechanism like TSIG [32], and obtain verified RRSets over the secure channel. TSIG is a protocol which leverages shared symmetric keys (established by other means outside the scope of TSIG) for establishing secure channels. In DNS, TSIG is used by zone authorities to securely send master files to ANSs. This same strategy can also be used for establishing a secure channel between clients and LDNSs.

The implication of using such an approach to lower overhead for clients is that *DNSSEC can no longer claim that end-points do not have to trust the middle-men.* With this approach, clients are required to trust the LDNSs (and consequently their operators).

More specifically,

1. If a DNSSEC enabled LDNS X is compromised, or if the TSIG secret of X is privy to an attacker, then X (or the attacker) can disseminate fake RRSets for *any* zone; such RRSets will be blindly accepted by *all* stub-resolvers which query LDNS X.
2. Similarly, if a DNSCurve enabled LDNS X is compromised, or if the DNSCurve secret of X is privy to an attacker, then X (or the attacker) can disseminate fake RRSets to all stub-resolvers that query LDNS X.
3. On the other hand, in the case of TCB-DNS, if a LDNS X (with DNS-TM P) is compromised, the attacker cannot disseminate fake RRSets. It is only if the secrets of the TM P become privy to the attacker (*and* if the TM P has not been revoked) can the attacker disseminate fake RRSets to the stub-resolvers that query LDNS X.

4.6.6 NSEC3 Opt-Out

For consummate realization of DNSSEC assurances even top level domains should adopt DNSSEC. While authenticated denial is an especially important feature for gTLDs, the overhead for this purpose can be substantial for large zones, and especially for zones where new names are frequently added. More specifically, zones with frequent addition/deletion of names become more susceptible to replay attacks.

Consider a scenario where an RRSet corresponding to a new name (or name-hash) x needs to be added. Before the name is added, a signed encloser (x_l, x_h) will exist for x. However, after inserting x the encloser (x_l, x_h) needs to be revoked. Two new enclosers should be added instead, (x_l, x) and (x, x_h). Similarly, consider a scenario when an existing name y needs to be removed, and two signed enclosers (y_l, y) and (y, y_h) currently exist. In this case, both enclosers (y_l, y) and (y, y_h) need to be revoked and replaced with a new encloser (y_l, y_h).

Due to the fact that it is not possible to foresee which of the currently valid records will need to be revoked due to the addition of an (as yet unknown) name in the future, it is necessary to choose small enough life-times for all NSEC/NSEC3 enclosers. Obviously, for gTLDs like `.com` with several tens of millions of names, this is far from practical.

In TCB-DNS, due to the low overhead for computing MACs even gTLDs can afford to recompute enclosers more frequently. However, frequent reauthentication of NSEC3 records in DNSSEC is expensive for two reasons. The obvious reason is that the computational overhead for digital signatures is high. The other reason is that NSEC3 *deliberately* increases the complexity of hashing names to render dictionary attacks more time-consuming. Due to the substantial overhead involved in re-generation of signed NSEC3 records, DNSSEC is forced to employ larger lifetimes for NSEC3 signatures and consequently become more susceptible to replay attacks.

Recently, NSEC3 with an opt-out specification [33] has been proposed to make it more practical for gTLDs to adopt DNSSEC. Using opt-out NSEC3 can reduce the instances leading to revocation of RRSIG(NSEC3) RRs, thereby permitting longer lifetimes for NSEC3 RRSIGs. An NSEC3 record indicating an encloser (x_l, x_h) with an unset opt-out bit is proof that no enclosing records exist. However, if the opt-out bit is set, the implication is that zero or more unsigned delegations *may* exist, thereby, diluting assurance **A2**. Furthermore, some serious security exploits resulting from using the opt-out approach have been identified recently [34].

4.7 Alternative to IPSec

The atomic relay function can be used for relaying authenticated values or secrets.

Specifically, relaying an authenticated value serves the purpose of broadcasting an authenticated value to any number of receiving processes.

While it does not make much sense to relay a secret to an unlimited number of processes[10] relaying one-way functions of secrets can serve useful purposes. For example, consider a scenario where a secret K created by a process is relayed over a path A, P to C. Specifically, assume that a secret K_C provided to C is a one-way function of the secret K and the path (A, P, C), for example,

$$K_C = h(h(h(K \parallel A) \parallel P) \parallel C) \tag{4.14}$$

As the source can easily compute K_C if the path is known, and as C has knowledge of K_C, K_C can be used as a secret for securing subsequent interactions between the source and C.

Such a functionality to atomically relay secrets can be added to TCB-DNS modules to cater for authenticated end-to-end secrets necessary for protocols like SSL/TLS and IPSec. Any server can relay a secret to a TM associated with any name server for the zone. A secret K chosen by the server can be provided to a zone DNS server module A as $K_A = h(K \parallel A)$. Similarly, a secret $K_B = h(K \parallel B)$ can be provided to another zone server with module B. Zone server modules can relay a function of this secret to any module associated with a local DNS server. The

[10] After all, what is the use of the secret if access to the secret is not controlled?

secret K_A is relayed by name server module A to a local DNS server module P as $K_P = h(K_A \parallel P)$. Similarly name server module B could relay the secret to another preferred name server Q as $K_Q = h(K_B \parallel Q)$, and so on. A client C may then receive a value $K_C = h(K_P \parallel C)$ from module P.

This enables clients and servers to *opportunistically* establish a common secret K_C in the process of making a DNS query. When a client queries the local DNS server (module) P for a DNS record corresponding to a server, the local DNS server queries the zone name server module A for the record. The zone name server module A which already has the secret K_A provided by the server can now relay this value to P as K_P, which is relayed by P to client C as K_C. The client can now establish a secure connection with the server. By merely indicating the path (A, P, C) to the server, the client enables the server to compute K_C which can be used as the shared secret for an end-to-end security protocol like SSL or IPSec.

4.7.1 IPSec Operation

IPSec [35] is a family of end-to-end security protocols between two hosts identified by the source and destination IP address in an IP packet. As IPSec is transparent to the layers above, even IPSec unaware applications can take advantage of the assurances provided by IPSec.

When a host (with IP address) A sends a packet to a host B, IPSec component of A first verifies if a security association (SA) for B exists in A's *SA database* (SAD). If not, the SA is established by exchanging ISAKMP (Internet security association key management protocol) packets [36], with IKE (Internet key exchange) [37] payloads. IKE employs the Oakley key determination protocol [38] which often relies on a variation of Diffie-Helman key establishment protocol for establishment of a shared secret between the two end-points.

In IPSec parlance, an "IPSec SA" is a function of the shared secret, and various security parameters related to the process of sending IP packets from A to B. The security parameters include the specification of a specific IPSEc "mode" to be used, cryptographic mechanisms to be used, and the IP address A of the sender. All such parameters are captured as a security parameter index (SPI), which is a one-way function of the parameters.

IPSec SAs are one-way associations. In other words, an SA established between A and B for packets from A to B can not be used for packets from B to A (for which a new SA with a different SPI will have to be generated).

IPSec can provide authentication-only service using authentication headers (AH) or privacy-only service using encapsulating security payload (ESP) headers, or both privacy and authentication using ESP header with authentication. IPSec provides two modes of operations. In the *transport* mode AH or ESP headers are inserted in IP packets between the IP header and the transport header. In the *tunnel* mode, IPSec packets (with a IP header followed by an IPSec header) can carry entire plain IP packets as payloads. In fact even an IPSec packet can be a payload of another IPSec packet. Any number of such IPSec nesting may exist.

The specific modes and services to be employ are determined by policies specified in a *security policy database* (SPD). Specifically, before A can send an IP packet to B, depending on the IP destination, and possibly transport protocol and port, the SPD may specify an appropriate SPI to be used.

An SA is in the SAD is uniquely identified by the SPI, the IP of the destination, and security protocol (AH or ESP) identifier. Thus, using the SPI information obtained from the SPD, the corresponding SA can be extracted from the SAD.

4.7.2 IPSec Issues

The immense flexibility of IPSec is a two-edged sword. While enabling the use of IPSec in a wide variety of deployment scenarios, the complexity of IPSec options can render deployment of IPSec a difficult task. This is perhaps the main reason for luke-warm adoption of IPSec.

IPSec was also designed originally to be an integral part of IPv6. During the design of IPSEc it was assumed that network address translators (NAT) are a short lived work-around to address the limitation on the number of IPv4 addresses (2^{32} total addresses). Unfortunately, IPv6 adoption took off very slowly, and NATs are still around.

IPSec, by design, cannot be used by hosts behind NATs [39], as the security parameter index in the IPSec header will be unverifiable for the receiver due to two reasons:

1. Change to the source IP address field introduced by the NAT
2. Changes to transport port numbers (the integrity of which is protected by IPSec) by NATs

Currently the only alternative is to establish an IPSec tunnel between the host and the NAT and another tunnel between the NAT and the destination, through which a plain IP packet can be tunneled. Such an approach, obviously can not be considered as true end-to-end security.

Yet another reason that has prevented global use of IPSec is that establishment of shared keys using Diffie–Helman can be expensive, especially considering that servers that will need to establish two SAs for each client.

4.7.3 IPSec Alternative Leveraging TCB-DNS

In the alternative to IPSec, which we shall term as `ipsec`, A-type records created by the zone owner indicate an additional field, a TCB-DNS identity corresponding to the host. A host with TCB-DNS identity X is authorized to receive a secret corresponding

Table 4.2 Port mapping in the receiver in TCB-DNS based alternative to IPSec

S-IP	TCB-DNS ID	S-Port	Actual protocol	Actual source port	Assigned port
R	R_T^1	42000	UDP	u_1	u_1
R	R_T^2	42000	UDP	u_2	u_2
R	R_T^3	42000	TCP	t_1	t_1
R	R_T^4	42000	TCP	t_1	$t_2' \neq t_1$

to identity X from TCB-DNS key distribution centers, and public values necessary to establish a shared secret with each of the name servers for the zone.

The host with identity X can choose a random secret K and supply the secret to an ANS module A. As outlined earlier in this section, this enables the host and the client to establish a shared secret in the process of making a DNS query. Thus, for `ipsec`, a DNS query is also simultaneously serves as the ISAKMP/IKE protocol for establishing `ipsec` SAs.

The `ipsec` layer will receive packets from higher layers (identified by IP protocol numbers) like TCP, UDP, ICMP, IGMP etc. These packets will be encapsulated by an `ipsec` header. The `ipsec` header will indicate the TCB-DNS address of the client and server hosts, and the TCB-DNS path (the identities of an ANS module and a LDNS module) through which the server host received the secret.

The `ipsec` packet is *prepended* with a *dummy* transport header (say, UDP) indicating a special port number (for example, an as yet unassigned number like 799). Thus, the lower layer, viz., the IP layer will simply see `ipsec` packets as UDP packets with sender/destination port numbers 799/799. The main reasons for introducing a dummy UDP layer are:

1. To permit seamless introduction of `ipsec` (no modifications to the IP layer), and
2. To facilitate clients behind NATs to employ `ipsec`

In an `ipsec` enabled host the `ipsec` layer is seen by the IP layer as a process listening at UDP port 799.

For clients behind a NAT, the NAT may replace the source port number (UDP, port 799) with another assigned port number (say, 42000).

Note that from the perspective of the NAT, all `ipsec` packets from a host with a private address R will correspond to a single entry in the NAT table. Consider a scenario where a host with a private IP R and TCB-DNS identity R_T has several active TCP connections with source ports $t_1 \cdots t_n$ and multiple UDP "connections" with source ports $u_1 \cdots u_m$. From the perspective of the NAT all such connections will correspond to a single entry in the NAT (mapping source UDP port 799 and source IP R to assigned port 42000).

When a host receives a packet from a source behind a NAT, if the source port is not 799, the receiver can deduce that the client is behind a NAT. There may be multiple simultaneous connections with R. The TCB-DNS identity in the `ipsec` header will be different for different hosts behind the same NAT. Also note that that the actual port number in the inner transport packet will *not* be modified by the NAT. Thus, it is possible that two inner transport packets with the same source port and protocol, to originate from different computers (different TCB-DNS identities) behind the same NAT with IP address R.

While the ipsec layer can readily differentiate the two connections using the TCB-DNS identities the higher layers can not. For this purpose the ipsec layer may need to modify the inner source port address and maintain a port mapping table as shown in Table 4.2.

Table 4.2 depicts a scenario where the payload of ipsec packets from two different hosts R_T^3 and R_T^4 behind the same NAT (IP address R) carry transport payloads with the same protocol (TCP) and port number (t_1). In order for the TCP layer to differentiate between the two connections (both TCP, same port, and same source address R) the inner port number will need to be mapped to a value $t_2' \neq t_1$. When the response transport packet is received the ipsec layer can map the port number back to t_1. From the table it can also determine that for packets to R the source port in the outer (dummy) UDP header should be set to 42000.

More specifically, the ipsec layer will need to modify the port number only if the source address R, **and** the protocol (TCP), **and** port number t_1, are identical for packets with different TCB-DNS identities.

Chapter 5
Scalable Key Distribution Schemes

While schemes like MLS and IT may be sufficient for most practical networks, the need for storage proportional to the current network size can cramp their utility in some application scenarios. In several emerging application scenarios, billions, or possibly even trillions of resource limited devices may be deployed, where any device may be required to compute a pairwise secret with any other device. Thus, key distribution schemes with no limitation on scalability are called for.

Scalable key distribution schemes can be broadly classified into certificate-based and identity-based schemes.

5.1 Certificates Based Schemes

In certificates-based schemes each entity is associated with an identity, a public key, and a private key. Each entity is free to *choose* their own private key and compute the corresponding public key. An entity with identity A can choose a private key R_A and derive (compute) a public key U_A.

Unfortunately, the public key U_A provides no information about the identity A. Consequently, a trusted third party—in the form of a certificate authority—has to securely specify a *binding* $< A, U_A >$ between the identity and the public key of every entity. This is achieved through a *certificate* issued by the CA. For very large scale networks the CAs themselves may be organized in a hierarchical fashion. In such scenarios, certificate [40] will be required to facilitate validation of public keys.

Certificate based schemes typically rely on asymmetric encryption and signature schemes like RSA, El Gamal (and variants like DSA), and elliptic curve schemes, used in conjunction with a hierarchical organization of CAs in the form of a public key infrastructure (PKI). Such schemes facilitate any two entities with signed public keys to exchange certificate chains to convince each other of the validity of their respective public keys. The verified public key of an entity A can then be used to send a private value to A.

M. Ramkumar, *Symmetric Cryptographic Protocols,* DOI 10.1007/978-3-319-07584-6_5,

5.2 Identity Based Schemes

For identity (ID) based schemes the identity of an entity itself doubles as the public key, thus obviating the very need for certificates. In such schemes, a key distribution center (KDC) chooses public parameters of the system and one or more master secrets. The KDC can then compute the private key(s) corresponding to *any* public key (the identity). The private keys for an entity with identity A are thus *assigned* by the KDC to the entity A.

For convenience, the identity of an entity may be chosen as a one-way function of a descriptive real-life identity. For example, Alice, with a real-life identity described by string $S_A =$ "Alice B Cryptographer, AnyTown, USA" can be assigned an identity $A = h(S_A)$.

Identity-based public key schemes for encryption (IBE) and signatures (IBS), most of which take advantage of pairings in special elliptic curve groups, have attracted substantial attention recently [41–43]. Identity-based schemes are increasingly seen as preferable over certificates based schemes for large scale networks, and especially for many emerging applications. In typical client–server interactions in existing networks, the client and server exchange public key certificates for mutual authentication, at the end of which a shared secret is established. This secret can be used for authentication and encryption of a *large* number of packets exchanged between them subsequently. Thus, the overhead (exchange of certificates or certificate chains and their verification) incurred for establishing a shared secret can be amortized for securing large amounts of data exchanged subsequently between the server and a client.

In several emerging applications however, participants engage in relatively brief interactions with a large number of other participants. Thus, the overhead for exchanging certificates for each interaction may be prohibitive, especially for large scale networks where chains of certificates need to be exchanged and verified. With identity-based schemes two participants A and B can independently and instantaneously (without any handshakes) compute a shared secret K_{AB}.

5.2.1 Identity-Based Key Predistribution Schemes

Identity-based key predistribution schemes (KPS) consist of a KDC and nodes (participants that need to establish shared secrets) with unique identities drawn from an "identity space" $\mathbb{I}$.

The KDC chooses a set of P master secrets $\mathbb{S}$; each node is provided with a set of $k \leq P$ secrets. The set of k secrets $\mathbb{S}_A$ assigned to a node with identity $A \in \mathbb{I}$ is determined by a public function $F()$ which takes two inputs—the identity A, and the set of P master secrets $\mathbb{S}$.

Two nodes A and B (with secrets $\mathbb{S}_A$ and $\mathbb{S}_B$ respectively) can compute a common secret K_{AB} using another public function $G()$. In other words,

$$\left.\begin{array}{l}\mathbb{S}_A = F(\mathbb{S}, A)\\ \mathbb{S}_B = F(\mathbb{S}, B)\end{array}\right\}\text{and}K_{AB} = G(\mathbb{S}_A, B) = G(\mathbb{S}_B, A) \tag{5.1}$$

Apart from A and B who can *legitimately* compute K_{AB} using their respective secrets ($\mathbb{S}_A$ and $\mathbb{S}_B$ respectively), an attacker who has access to secrets of many (say, v) nodes $\{O_1 \cdots O_v\}$ may also be able to *illegitimately* compute K_{AB}. An *n-secure* KPS can resist a collusion of n attackers pooling all their secrets together. In other words, for a n-secure KPS, an attacker who has access to keys of n nodes cannot determine any illegitimate[1] shared secret.

Most scalable KPSs require every node to possess a key chain with $k = \mathbb{O}(n)$ keys to realize n-security. Note that with $\mathbb{O}(n)$ limitation on storage

1. Nonscalable KDSs (like the basic KDS and MLS) can only support networks of size $N = \mathbb{O}(n)$.
2. Scalable KPSs can however, support *any* network size $N \leq |\mathbb{I}|$—as every node requires a unique identity and the number of nodes is limited only by the size of the identity space (for example, $N = 2^{128}$ if 128-bit identities are used)—but can only *tolerate* a collusion of up to n entities.

5.2.2 Blom's Schemes

Blom et. al. [44] proposed the first KPS in the literature. In the n-secure Blom's polynomial scheme, the KDC chooses $P = \binom{k}{2}$ secrets from $\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$, where $n = \lceil \frac{k-1}{2} \rceil$ and $q \geq N$ is a prime.

The $P = \binom{k}{2}$ secrets are interpreted as the coefficients r_{ij} of a symmetric polynomial $f(x, y)$ of order $k-1$,

$$f(x, y) = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} r_{ij} x^i y^j \bmod q. \tag{5.2}$$

For a symmetric polynomial, as $r_{ij} = r_{ji}$, $f(x, y) = f(y, x)$.

Every node is assigned a unique identity—as a number chosen from $\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$ (or $\mathbb{I} = \mathbb{Z}_q$). A node A receives the k coefficients $a_0 \cdots a_{k-1}$ of the polynomial

$$g_A(x) = f(x, A) = \sum_{i=0}^{k-1} a_i x^i \bmod q. \tag{5.3}$$

[1] The attacker can obviously determine all shared secrets that each of the n compromised nodes can legitimately compute.

as its k secrets. Similarly, B receives the k coefficient of the polynomial $g_B(x) = f(x, B)$.

Now, A and B can compute

$$K_{AB} = K_{BA} = f(A, B) = f(B, A) = g_A(B) = g_B(A) \tag{5.4}$$

independently. More specifically, node A with secrets $a_0 \cdots a_{k-1}$ and node B with secrets $b_0 \cdots b_{k-1}$ can readily compute

$$\begin{aligned} K_{AB} &= \sum_{i=0}^{k-1} a_i B^i \bmod q. \\ &= \sum_{i=0}^{k-1} b_i A^i \bmod q. \end{aligned} \tag{5.5}$$

With access to k secrets each from *more* than $n = \lceil \frac{k-1}{2} \rceil$ nodes, an attacker can construct a system of P independent simultaneous equations to solve for *all* P secrets chosen by the KDC. As long as the attacker can only construct less than P simultaneous equations the attacker learns nothing about the P KDC secrets, and consequently the secrets assigned to uncompromised nodes.

5.2.2.1 Blom's SKGS

In the symmetric key generation system (SKGS) scheme based on MDS (maximum distance separation) codes, also proposed by Blom [45], a n-secure scheme requires $k = n + 1$ secrets to be assigned to each entity.

SKGS employs a public primitive element (or generator[2]) $\alpha \in \mathbb{Z}_p$. Corresponding to any $i \in \mathbb{Z}_p$ let us define vectors

$$\mathbf{g_i} = \{\mathbf{g^i_0}, \mathbf{g^i_1}, \ldots \mathbf{g^i_n}\} \text{ where } \mathbf{g^i_j} = \alpha^{\mathbf{(i-1)(j-1)}} \tag{5.6}$$

The KDC chooses a $(n + 1) \times (n + 1)$ symmetric matrix $\mathbf{D}$ with $\binom{n+1}{2}$ independent values chosen randomly from $\mathbb{Z}_q$. The $\binom{P=n+1}{2}$ values are the KDC secrets. Node A is issued $k = n + 1$ values (secrets) computed as $\mathbf{d^A} = \mathbf{Dg_A}$. A and B (with secrets $\mathbf{d^A}$ and $\mathbf{d^B}$ respectively) can now calculate

$$\begin{aligned} K_{AB} &= (\mathbf{d^A})^\mathbf{T}\mathbf{g_B} = (\mathbf{d^B})^\mathbf{T}\mathbf{g_A}. \\ &= \sum_{j=1}^{n+1} d^A_j \alpha^{(j-1)(B-1)} = \sum_{j=1}^{n+1} d^B_j \alpha^{(j-1)(A-1)} \end{aligned} \tag{5.7}$$

An n-secure SKGS is unconditionally secure as long as n or less entities pool their secrets together. It is *completely* compromised if more than n entities do so.

[2] More generally, for any N the $(n+1) \times N$ matrix $\mathbf{G} = [\mathbf{g_0}\, \mathbf{g_1} \cdots \mathbf{g_N}]$ is referred to as the maximum distance separation (MDS) generator matrix.

5.3 Probabilistic KPSs (PKPS)

More generally, KPSs can be regarded as (n, p)-secure, where an attacker, by pooling secrets assigned to n entities can illegitimately compute a pairwise secret with a probability p. In other words, such attacker can only expose a fraction p of all illegitimate pairwise secrets by utilizing secrets pooled from n compromised nodes.

A $(n = 1000, p = 2^{-64})$-secure probabilistic KPS offers a guarantee that an attacker who has access to all secrets of $n = 1000$ nodes can only compute one in 2^{64} illegitimate pairwise secrets (involving nodes that have *not* been compromised). Note that as long as p is low enough (say, 2^{-64}) it is *computationally infeasible* for an attacker to even identify *which* pairwise secrets can be compromised by using the pool of secrets exposed from n entities.

Deterministic n-secure KPSs (like Blom's SKGS) can be seen as special cases of (n, p)-secure KPSs where p takes only binary values (0 or 1). While deterministic KPSs fail catastrophically ($p(n') = 0$ for $n' \leq n$ and $p(n') = 1$ for $n' > n$), for (n, p)-secure probabilistic KPSs p increases gracefully with n.

5.3.1 Allocation of Subsets

Almost every PKPS in the literature is based on the idea of allocation of a subset of keys to every entity, chosen form a larger pool of keys. Two nodes A and B use all the common secrets to derive a pairwise secret K_{AB}.

Early subset allocation schemes [46–47] relied on *deterministic* strategies for allocation of subsets of keys to every node. The matrix scheme [46] was far from efficient in terms of number of stored keys needed in each node (for a network size of N each node needs $O(n\sqrt{N})$ keys in order to be n-secure). Many other strategies for key allocations were proposed subsequently, [47–48], for which $k = O(n \log N)$—most of them were motivated by Erdos et al.'s seminal work on intersections of subsets [49].

However, the complexity of the deterministic key allocation algorithm makes it challenging for two nodes to discover *which* secrets they actually share. Consequently, nodes had to explicitly exchange information regarding the indexes of the keys they possess to identify common secrets. Such schemes cannot be considered as identity based as the identity of nodes are not used for deriving the pairwise secret. Furthermore such schemes can cater only for privacy—not authentication.

Dyer et al. [11] was the first to point out the simplicity and effectiveness of *random subset allocations*. Early in the first decade of this millennium, interest in low complexity key distribution strategies were rekindled by the emergence of new paradigms like ad hoc sensor networks. While several approaches [50–55] based on random subset allocation were proposed, all of them ignored the need for authentication.

5.3.2 Random Preloaded Subsets

The random preloaded subsets (RPS) scheme [56] is an identity based extension of Dyer's scheme. The main difference is that the indexes of secrets assigned to a node are tied to the identity of the node, through a simple one-way function.

In the random preloaded subsets (RPS) scheme, the KDC chooses:

1. An indexed set of P secrets $[K_1 \cdots K_P]$
2. A one-way function $F()$
3. A key-ring size $k < P$; let $a = k/P < 1$

For a node assigned identity $A \in \mathbb{I}$, the key allocation function outputs of the indexes of the keys to be assigned to A.

$$F(A) = \{A_1 \cdots A_k\}, 1 \leq A_i \leq P. \tag{5.8}$$

For example, a random permutation of the numbers $1 \cdots P$ could be generated using the identity A as the seed for $F()$, and the first k numbers may be chosen as values $A_1 \cdots A_k$.

Node A is now assigned a set of k secrets

$$\mathbf{S}_A = \{K_{A_1} \cdots K_{A_k}\}. \tag{5.9}$$

Two nodes A and B can execute $F(A) \cap F(B)$ to discover the indexes of common keys. If $k/P = a$, two nodes will share (on an average) $m = ak = a^2 P$ keys. Let the m common indexes be $s_1 \cdots s_m, 1 \leq s_i \leq P \forall i$. The corresponding shared secrets are then $K_{s_1} \cdots K_{s_m}$. All shared secrets are XORed together to compute K_{AB} as

$$K_{AB} = K_{s_1} \oplus \cdots \oplus K_{s_m}. \tag{5.10}$$

As an example, let $P = 8$ and $k = 4$. Let $K_1 \cdots K_8$ be the keys chosen by the KDC. Let

$$\begin{aligned} F(A) &= \{1, 4, 6, 7\} \text{ and} \\ F(B) &= \{1, 5, 6, 8\}. \end{aligned} \tag{5.11}$$

The shared keys are then K_1 and K_6, and

$$K_{AB} = K_1 \oplus K_6. \tag{5.12}$$

5.3.3 Hash-Chain KPS

In [16] Leighton and Micali also proposed a scalable key predistribution scheme (in addition to the alternative to Kerberos that was discussed in Sect. 3.1.2). In the hash-chain KPS (which was referred to simply as Scheme III in [16]) the KDC chooses:

1. An indexed set of P secrets $[K_1 \cdots K_P]$
2. A value L—the hash chain length
3. A key allocation function $f()$, which generates uniformly randomly distributed numbers between 1 and L

Corresponding to an identity A

$$f(A) = \{a_1 \cdots a_P\}, 0 \leq a_i \leq L - 1 \tag{5.13}$$

determines the hash depth of the $k = P$ secrets to be assigned to node A, viz

$$\mathbf{S}_A = \{K_1^{a_1} \cdots K_P^{a_P}\} \tag{5.14}$$

where

$$K_i^j = h^j(K_i) \tag{5.15}$$

is the result of hashing K_i repeatedly, j times.

Two nodes A and B with secrets $\mathbf{S}_A = \{K_1^{a_1} \cdots K_P^{a_P}\}$ and $\mathbf{S}_B = \{K_1^{b_1} \cdots K_P^{b_P}\}$ can compute P common secrets

$$\mathbf{S}_{AB} = \{K_1^{x_1} \cdots K_P^{x_P}\} \text{ where } x_i = \max(a_i, b_i). \tag{5.16}$$

For example, corresponding to an index 23, let $a_{23} = 4$ and $b_{23} = 6$. Node A has access to K_{23}^4, and node B has access to K_{23}^9. Node A may now hash K_{23}^4 five times $(5 = 9 - 4)$ to compute a common secret K_{23}^9. All P common secrets are then XORed together to yield the shared key K_{AB} as

$$K_{AB} = K_1^{x_1} \oplus \cdots \oplus K_P^{x_P}. \tag{5.17}$$

As a trivial example, let $P = 8$ and $L = 4$. Let $K_1 \cdots K_8$ be the keys chosen by the KDC. Let

$$\begin{aligned} f(A) &= \{4, 2, 1, 3, 3, 4, 2, 1\} \text{ and} \\ f(B) &= \{3, 4, 2, 3, 2, 2, 1, 4\}. \end{aligned} \tag{5.18}$$

The keys assigned to A and B are

$$\begin{aligned} \mathbf{S}_A &= \{K_1^4, K_2^2, K_3^1, K_4^3, K_5^3, K_6^4, K_7^2, K_8^1\} \text{ and} \\ \mathbf{S}_A &= \{K_1^3, K_2^4, K_3^2, K_4^3, K_5^2, K_6^2, K_7^1, K_8^4\} \end{aligned} \tag{5.19}$$

and

$$K_{AB} = K_1^4 \oplus K_2^4 \oplus K_3^2 \oplus K_4^3 \oplus K_5^3 \oplus K_6^4 \oplus K_7^2 \oplus K_8^4. \tag{5.20}$$

5.3.4 Hashed Random Preloaded Subsets (HARPS)

Hashed random preloaded subsets [57] is a combination of RPS and the hash-chain KPS. In HARPS, the KDC chooses:

1. An indexed set of P secrets $[K_1 \cdots K_P]$
2. Hash-chain length L
3. Key-ring size $k = aP, a < 1$
4. A key allocation function $G()$, which generates a set of k indexes between 1 and P, and corresponding to each of the k indexes, a uniformly randomly distributed number between 1 and L.

Corresponding to an identity A

$$G(A) = \{(A_1, a_1) \cdots (A_k, a_k)\}, 1 \le a_i \le L, 1 \le A_i \le P \tag{5.21}$$

determines the k secrets to be assigned to node A, viz

$$\mathbf{S}_A = \{K_{A_1}^{a_1} \cdots K_{A_k}^{a_k}\} \tag{5.22}$$

where $K_i^j = h^j(K_i)$ is obtained by repeatedly hashing K_i, j times.

Two nodes A and B can determine (on an average) $m = a^2 P = ak$ common indexes, say $s_1 \cdots s_m$ where $1 \le s_i \le P$. Corresponding to each shared index the two nodes may however possess a key at different hash depths. Let the hash depths of the secrets assigned to A and B corresponding to a shared index s_i be d_{a_i} and d_{b_i} respectively, and let

$$d_i' = \max(d_{a_i}, d_{b_i}), 1 \le i \le m. \tag{5.23}$$

The m common keys between A and B are then

$$K_{s_1}^{d_1'} \cdots K_{s_m}^{d_m'}, \tag{5.24}$$

and the shared secret K_{AB} is computed as

$$K_{AB} = K_{s_1}^{d_1'} \oplus \cdots \oplus K_{s_m}^{d_m'}. \tag{5.25}$$

As an example, let $P = 8$, $k = 4$ and $L = 4$. Let $K_1 \cdots K_8$ be the keys chosen by the KDC. Let

$$\begin{aligned} G(A) &= \{(1,4),(4,2),(6,1),(7,3)\} \text{ and} \\ G(B) &= \{(1,3),(5,1),(6,3),(8,2)\}. \end{aligned} \tag{5.26}$$

The keys assigned to A and B are

$$\begin{aligned} \mathbf{S}_A &= \{K_1^4, K_4^2, K_6^1, K_7^3\} \text{ and} \\ \mathbf{S}_A &= \{K_1^3, K_5^1, K_6^3, K_8^2\} \end{aligned} \tag{5.27}$$

and using the two shared indexes 1 and 6, A and B compute a common secret

$$K_{AB} = K_1^4 \oplus K_6^3. \tag{5.28}$$

HARPS is a generalization of both RPS and hash chain KPS. Specifically, RPS is HARPS with $L = 1$ as all keys have the same hash depth. The hash chain KPS is HARPS with $k = P$ (or $a = 1$).

5.4 (n, p)-Security of HARPS

An analysis of HARPS [57] involves estimation of the probability p that an attacker who has access to all keys of n nodes can discover an illegitimate pair-wise secret. RPS can be seen as a special case of HARPS with $L = 1$ (as all nodes have keys at the same hash-depth). The hash-chain scheme can be seen as a special case of HARPS with $a = k/P = 1$.

Consider P round game involving $n + 2$ participants—2 participants intending to compute a pairwise secret, and n nodes in the attacker coalition attempting to illegitimately determine the pairwise secret. In the i^{th} round all $n + 2$ nodes choose the key K_i from the key pool with probability a (or, do *not* choose the index K_i with probability $1 - a$). In each round (corresponding to each of P root keys), the two nodes arrive at an *elementary share* of their shared secret if both pick the secret K_i—which occurs with a probability a^2. If both nodes do pick the key, they also pick a random hash depth between 1 and L. Let the maximum of the two hash depths be d. If both nodes happen to choose K_i, the elementary shared secret corresponding to the index i is K_i^d.

This elementary share is "safe" if the attacker coalition *cannot* determine the secret K_i^d. In other words, this elementary share is safe if none of the n attackers choose the index K_i with hash depth less than or equal to d. This may be because

1. None of the attackers chose the key K_i, or
2. Some (say $u \leq n$) of the attackers choose the key, but all of them chose a hash depth greater than d.

The two nodes win the round i if

1. Both nodes pick K_i.
2. The attacker coalition is not able to determine the key K_i^d.

In order to win the match, the two nodes *have to win only one* of the P rounds. Note that even if they win one round, they arrive at a shared secret which is not available to the attacker coalition. In other words, the attacker coalition has to win all P rounds.

We shall denote by ϵ, the probability that the two nodes win any round i. The probability $p(n)$ that the attacker coalition wins *all* P rounds is then

$$p(n) = (1 - \epsilon)^P. \tag{5.29}$$

5.4.1 Probability of Winning a Round

The probability that two nodes pick the ith key is a^2. In other words, a^2 is the probability that the two nodes arrive at an elementary share. Out of the n nodes in the attacker's coalition, zero or more nodes may also pick the ith key.

Let us consider a scenario where u of the n attacker nodes pick the ith key. The probability that *exactly* u $(0 \leq u \leq n)$ of n nodes pick the ith key is the binomial probability $B_a(n,u)$, where

$$B_a(n,u) = \binom{n}{u} a^u (1-a)^{n-u}. \tag{5.30}$$

Let α_1, α_2 be the hash depths corresponding to the ith root key in the two nodes, and let $\alpha = \max(\alpha_1, \alpha_2)$. Let $\beta_1 \cdots \beta_u$ be the corresponding hash depths of the u keys, in the attacker's nodes. As long as $\beta = \min(\beta_1 \cdots \beta_u) > \alpha = \max(\alpha_1, \alpha_2)$, the attacker coalition *cannot* discover the elementary shared secret.

The probability $\Pr\{\beta > \alpha\}$ is

$$Q(L,u) = \sum_{i=1}^{L} \frac{2i-1}{L^2} \left(\frac{L-i}{L}\right)^u, \tag{5.31}$$

where $1 \leq u \leq n$. Thus, the probability that the elementary share is safe is

$$\begin{aligned} \epsilon &= \sum_{u=0}^{n} a^2 B_a(n,u) Q(L,u) \\ &= \sum_{u=0}^{n} \binom{n}{u} a^{u+2} (1-a)^{n-u} \sum_{i=1}^{L} \frac{2i-1}{L^2} \left(\frac{L-i}{L}\right)^u. \end{aligned} \tag{5.32}$$

The probability that the attacker wins all P rounds is then

$$p(n) = (1-\epsilon)^P = \left\{(1-\epsilon)^{\frac{1}{a}}\right\}^k. \tag{5.33}$$

For the special cases of RPS ($L = 1$) and LM ($a = 1$), the expressions for ϵ are respectively

$$\epsilon = \left\{ \begin{array}{ll} \epsilon_R = a^2 B_a(n,0) = a^2(1-a)^n & L = 0, a < 1 \\ \epsilon_L = Q(L,n) & a = 1, L > 0 \end{array} \right. . \tag{5.34}$$

Note that the expression for ϵ for HARPS has $n+1$ terms corresponding to $u = 0 \cdots n$. For LM (with $a = 1$), only the term corresponding to $u = n$ is non-zero. For RPS (with $L = 1$) only the term corresponding to $u = 0$ is non-zero. For the same value of a obviously $\epsilon_R \leq \epsilon$, indicating that the probability that each elemental share is safe (or the probability that the two nodes win any round) is higher for HARPS. Alternately, for the same P, k, $p(n)$ for HARPS ($L \geq 0$) is less, or at worst equal, to that of RPS (with $L = 1$). As L increases, $p(n)$ decreases.

5.4.2 Optimization of Parameters

For all three PKPSs the KDC can generate P secrets by choosing a master secret S and realizing any secret as $K_i = h(S \parallel i)$. Thus, the total number of secrets P is not an issue. However, as each node needs to store k secrets, it is desirable to minimize k.

Let us assume that it is desired that $p(n) < p^*$ for all $n \leq n^*$. For the hash chain KPS,[3] with $P = k$ there is no scope for optimization. For RPS, for a desired $p(n)$, the KDC can choose an optimal value of $a = k/P$ that minimizes k. For HARPS, the optimal value of a will also be a function of the maximum hash depth L.

For RPS, with $p_R(n) = (1 - a^2(1 - a)^n)^{k/a}$, k is minimized when $a = 1/(n^* + 1)$. Corresponding to this optimal (k-minimizing) choice of a,

$$\begin{aligned} k &= n^* e \log(1/p^*) \\ P &= n^*(n^* + 1) e \log(1/p^*) \text{ and} \\ m &= \frac{k^2}{P} \approx e \log(1/p^*) \end{aligned} \tag{5.35}$$

where m is the average number of secrets shared between any two nodes.

For HARPS, it is far from trivial to derive a simple closed form expression relating optimal choice of parameters k and P to a desired $p(n^*) \leq p^*$. However, from numerical computations using Eqs. (5.32) and (5.33) we can readily compute the optimal value of k/P for different values of L.

In general, for HARPS

$$\begin{aligned} a &= c_a/n^* \\ k &= c_k n^* \log(1/p^*) \\ P &= c_P (n^*)^2 \log(1/p^*) \text{and} \\ m &= c_m \log(1/p^*) \end{aligned} \tag{5.36}$$

where the constants c_a, c_k, c_P and c_m depend on L. We already know that for RPS (or HARPS with $L = 1$), we have $c_a \approx 1$, and $c_k \approx c_P \approx c_m = e \approx 2.72$.

For increasing L it can be seen that

1. c_k reduces from its maximum value of e for $L = 1$; in other words HARPS requires a lower number of keys to be stored by every node to achieve a desired (n^*, p^*)-secure scheme.
2. c_P also reduces—at a faster rate than c_k.
3. c_m increases marginally; in other words, even while the reduction in c_P is faster than the reduction in c_k, it does not reduce fast enough to prevent increase of c_k^2/c_P.

[3] The hash chain KPS by itself is inefficient as it requires more that $k = \mathbb{O}(n^2)$ keys to be assigned to each node. We shall not discuss this KPS further.

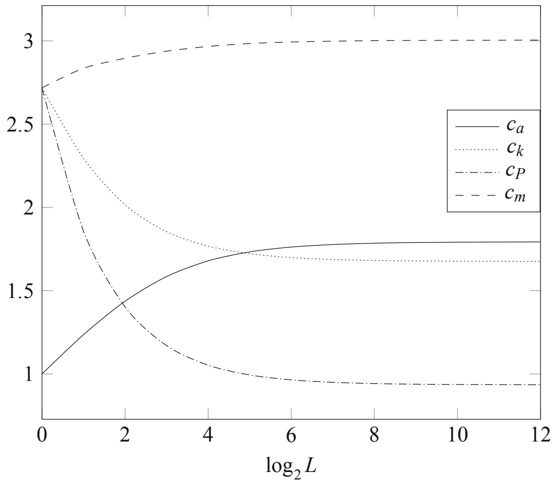


Fig. 5.1 Variation of HARPS parameters with maximum hash depth L

The four values are plotted as a function of L in Fig. 5.1. For ready reference, they are also tabulated in Table 5.1.

Thus, for a desired $p(n) \leq p^*$, if we choose $L = 64$, the number of keys to be stored by each node reduces by a factor of $e/1.7 \approx 1.6$ compared to RPS. Note that the advantage of choosing even larger L rapidly diminishes as L increases. Given that larger L will increase the computational overhead[4] for computing the shared secret, it does not seem productive to increase L beyond (for example) 64.

Finally, as a concrete example, if we desire $p(n) \leq 10^{-20} \forall n \leq 1000$ the optimal choice of parameters for RPS are $P = 125166005$ and $k = 125041$. For HARPS with $L = 64$, $P = 44405682$ and $k = 78154$. Figure 5.2 depicts the performance of HARPS for various values of L—$L = 1$ (RPS) and $L = 2, 4, 8, 16, 32, 64$ and 128. All eight schemes are optimized to realize $p(n) \leq 10^{-20} \forall n \leq 1000$.

As expected the performance curves of all schemes are close together for n close to the design value $n^* = 1000$. Note that unlike deterministic schemes which fail catastrophically for n greater than the design value, (n, p)-secure schemes deteriorate gracefully. Notwithstanding the fact that HARPS requires a smaller number of keys per node, the performance of HARPS the performance of HARPS also deteriorates more gracefully than RPS. The plot on the right in Fig. 5.2 illustrates the performance of RPS and HARPS for values of n substantially higher than the design value of $n^* = 1000$.

[4] For each of the k^2/P shared secrets one node has to perform $L/3$ hashes on an average to arrive at a key with the same hash depth as the other node.

Table 5.1 Optimal values of $a = k/P$ and k chosen to minimize k to realize a (n^*, p^*)-secure HARPS for various values of L. The corresponding values of key pool size P and the average number of shared secrets m are also shown

L	a	k	P	$m = k^2/P$
1	$\frac{1}{n^*}$	$2.718n^* \log(1/p^*)$	$2.718n^{*2} \log(1/p^*)$	$2.718 \log(1/p^*)$
2	$\frac{1.236}{n^*}$	$2.294n^* \log(1/p^*)$	$1.856n^{*2} \log(1/p^*)$	$2.836 \log(1/p^*)$
4	$\frac{1.437}{n^*}$	$2.015n^* \log(1/p^*)$	$1.402n^{*2} \log(1/p^*)$	$2.896 \log(1/p^*)$
8	$\frac{1.585}{n^*}$	$1.854n^* \log(1/p^*)$	$1.17n^{*2} \log(1/p^*)$	$2.938 \log(1/p^*)$
16	$\frac{1.679}{n^*}$	$1.767n^* \log(1/p^*)$	$1.052n^{*2} \log(1/p^*)$	$2.967 \log(1/p^*)$
32	$\frac{1.733}{n^*}$	$1.721n^* \log(1/p^*)$	$0.994n^{*2} \log(1/p^*)$	$2.984 \log(1/p^*)$
64	$\frac{1.762}{n^*}$	$1.699n^* \log(1/p^*)$	$0.964n^{*2} \log(1/p^*)$	$2.994 \log(1/p^*)$
128	$\frac{1.777}{n^*}$	$1.687n^* \log(1/p^*)$	$0.950n^{*2} \log(1/p^*)$	$2.998 \log(1/p^*)$
256	$\frac{1.785}{n^*}$	$1.681n^* \log(1/p^*)$	$0.942n^{*2} \log(1/p^*)$	$3.001 \log(1/p^*)$
512	$\frac{1.789}{n^*}$	$1.679n^* \log(1/p^*)$	$0.938n^{*2} \log(1/p^*)$	$3.003 \log(1/p^*)$
1024	$\frac{1.791}{n^*}$	$1.677n^* \log(1/p^*)$	$0.936n^{*2} \log(1/p^*)$	$3.004 \log(1/p^*)$

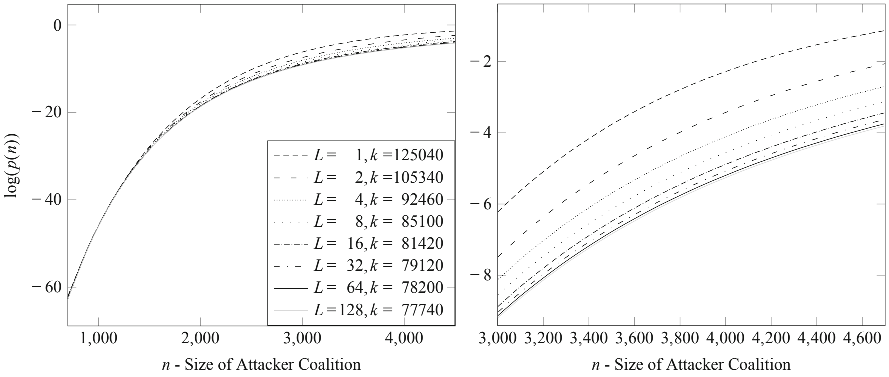


Fig. 5.2 Comparison of HARPS and RPS designed to meet the same specification, viz., $p(n) \leq 10^{-20} \forall n \leq 1000$. Even while HARPS requires a smaller number of stored keys per node, $p(n)$ for HARPS increases in a slower manner as n increases

5.5 Deterministic Versus Probabilistic KPSs

Renewed interest in KPSs since the beginning of this millennium stems from the emergence new application paradigms involving resource limited devices that may be deployed in very large numbers. In emerging ubiquitous computing applications [58], billions of unattended devices may need to interact among themselves, and co-operatively perform tasks to realize synergistic benefits.

Inexpensive resource limited sensors deployed in an unattended manner may sense various environmental conditions like pressure, humidity, temperature, pollen count, toxicity, etc., and relay such measurements to locations where such information is required. Devices with limited wireless transmission range may themselves form multi-hop ad hoc networks to relay data packets, thereby making it possible to quickly deploy vital communication infrastructure even in remote locations. Sensors are also expected to play a large role in reducing the cost of health-care in the future, where inexpensive sensors may continuously monitor the functioning of crucial organs and provide early warnings of abnormalities.

Not withstanding the fact that the devices are intended to be inexpensive, and resource limited (as most devices may be battery powered, and their useful lifetime may be limited by their battery life), it is important to assure the integrity of data emanating from or relayed by such devices. Unless there is some confidence in the integrity of reports from sensors—regarding the state of vital organs, or structural weaknesses in bridges, or the presence of dangerous contaminants in the atmosphere/potable water—the very utility of such emerging applications becomes questionable.

One essential prerequisite for emerging applications is thus an effective cryptographic strategy for protecting the integrity of data in transit. Useful cryptographic strategies for such applications are unfortunately constrained by several factors.

1. Ultimately, the strength of any cryptographic mechanism is limited by the extent of protection offered to secrets of the device. Thus, notwithstanding their low cost, unattended devices require well protected storage for secrets. Without such protection mechanisms, attackers can readily gain access to the secrets of a device to impersonate the device, to send deliberately misleading information.
2. For any device X, it may not be possible to identify a priori, a restricted set of specific other devices with which the device X may need to interact. Consequently, any device should have the capability to establish a shared secret with every (possibly billions or even trillions) device.
3. Devices in remote locations may not have access to a trusted server for mediation. Consequently, schemes that support ad hoc establishment of shared secrets are necessary.

While certificates based asymmetric cryptographic schemes can easily cater for ad hoc establishment of secrets, they may be prohibitively expensive for several devices that are deliberately resource limited to conserve battery life. In large-scale networks requiring ad hoc establishment of shared secrets where proactive mechanisms to protect secrets *are mandatory in any case*, KPSs do merit consideration as an alternative to conventional approaches based on asymmetric cryptography. This is especially true if the proactive strategies to protect secrets can ensure that it is impractical for attackers to compromise secrets from a large number of devices.

An ideal KPS should demand low computational, bandwidth and storage overhead, and simultaneously resist large coalitions of colluders. More specifically

1. Low computational and storage requirements are especially crucial inside the protected boundary within every device where secrets are stored and computations using such secrets are performed—smaller the protected boundary, better the extent of protection that can be afforded to components inside the protected boundary.
2. Low bandwidth for authentication of data is important as transmission overhead has a bearing on the battery life.

3. Storage overhead for nonsecret values is increasingly less of an issue. With rapidly decreasing costs of storage any conceivable device could support flash storage of several MBs, or even GBs.

5.5.1 KPS Complexity

The complexity of any KPS can be seen as composed of the following facets:

1. KDC complexity for issuing secrets to nodes
2. Complexity for any two nodes to compute a pairwise secret

The complexity for computing a pairwise key which can in turn be seen as consisting of three factors:

1. *Storage* for keys
2. *Public function* complexity—for operations that do *not* need access to KPS secrets
3. *Private function* complexity—for operations performed using KPS secrets

The reason to distinguish between public and private functions is that private functions will need to be executed inside a well-protected environment.

KDC Complexity

In SKGS, the KDC chooses a public generator $\alpha \in \mathbb{Z}_q$ and computes $k = n + 1$ secrets $\mathbf{d^A} = \{\mathbf{d_0^A}, \mathbf{d_1^A}, \dots \mathbf{d_n^A}\}$ where

$$d_i^A = \sum_{j=0}^{n-1} D(i,j)\alpha^{jA} \bmod q, 0 \leq i \leq n, \tag{5.37}$$

$D(i,j) = D(j,i)$ are the $k^2 = (n+1)^2/2$ KDC secrets. The overhead for the KDC for computing the secrets $\mathbf{d^A}$ to be assigned to node A amounts to $\mathbb{O}(n+1)^2$ finite-field multiplications. On the other hand, for PKPSs, the KDC complexity for computing $k = \mathbb{O}(n \log(1/p))$ secrets increases linearly with n (and k).

Storage for Key-Ring

For the deterministic n^*-secure SKGS the size of the key ring is $k = n^* + 1$. For (n^*, p^*)-secure RPS the storage required is $k \approx \mathrm{e}n^* \log(1/\mathrm{p}^*)$. For HARPS the storage required $k \approx c_k n^* \log(1/p^*)$ where $c_k \leq \mathrm{e}$.

Public Function Complexity

In SKGS, to ultimately compute K_{AB}, a node A first needs to compute $\alpha^{iB}, 0 \leq i \leq n$. This amounts to about $\log_2 n + n - 1$ finite-field multiplications.[5] In PKPSs like RPS

[5] $\log_2 n$ multiplications for computing α^B and $n-1$ multiplications for computing $\alpha^{iB}, 2 \leq i \leq n$ values.

and HARPS A has to compute $F(A)$ and $F(B)$ (to generate $2k$ pseudo-random numbers), and determine the m common indexes.

Private Function Complexity

In SKGS A will need to perform $n+1$ finite-field multiplications $x_i s_i^A, 0 \leq i \leq k-1$ (where s_i^A are the k secrets assigned to A). In PKPSs a relatively small number of secrets (m) to derive the shared secret K_{AB}. For (n^*, p^*)-secure RPS $m = \mathrm{e}\log(1/\mathrm{p}^*)$ block-cipher/hash operations will be required. In HARPS, where $m \approx 3\log(1/p^*)$, as some secrets will need to be hashed further, the private function complexity is $3\log(1/p^*)(L/6)$ (on an average, half of the m secrets will need to hashed forward by an average of $L/3$ hashes).

5.5.2 Complexity Versus Desired Collusion Resistance *n*

For deterministic schemes like SKGS, all facets of complexity increase linearly with n. For SKGS the private function complexity can become comparable to public key schemes even for n of the order of hundreds. The KDC complexity also increases as the square of the desired collusion resistance n. Thus, deterministic KPSs are ill suited for scenarios where we desire large n.

The main advantage of PKPSs is that the private function complexity ($m \propto \log(1/p^*)$) is *independent* of the desired collusion resistance n. Furthermore, the only two facets of complexity that are proportional to the desired collusion resistance n—storage and public function complexity may even be offloaded to external resources.

5.5.3 Using External Resources

Offloading storage is trivial as all keys can be encrypted and stored in an untrusted location to which the node has ready access (for example, a flash card plugged into the device). Only the single secret used for encrypting all secrets need to be stored inside the protected boundary.

Offloading public function $F()$ to external resources is also easy in scenarios where the module is merely used a trusted session key generator. In such a scenario, an external resource employed by A computes $F(A)$ and $F(B)$ to identify the indexes of m secrets shared with B, fetches m encrypted secrets from storage, and supplies the m encrypted secrets to the protected module in A. The module A, decrypts the m secrets to compute K_{AB}, chooses a random nonce N_{AB}, and outputs the nonce N_{AB} and a session key computed as

$$K_S = h(K_{AB} \parallel N_{AB}). \tag{5.38}$$

However, if the protected module is expected to verify the authenticity of message authentication codes computed using the shared secret or convey/receive authenticated secrets from other nodes, it is *not* sufficient for the module to restrict it's scope to operations with secrets. The module also needs to *verify* that the indexes of the encrypted secrets supplied by the external resource are indeed consistent with $F()$. Note that unless the module executes $F()$ by itself, it cannot verify the binding between the indexes of keys and the identities.

Fortunately, if the key allocation function is redefined as

$$a_i = F(A,i), 0 \leq i \leq k-1, 0 \leq a_i \leq P/k, \tag{5.39}$$

then any index can be computed at random. In this scenario, the module only needs to be provided with the $m \propto \log(1/p^*)$ shared secrets, along with shared indexes $s_1 \cdots s_m$. The module A can then simply *verify* that $s_i = F(A,i) = F(B,i)$ before using K_{AB} to verify the consistency of the message authentication code from B, or preparing an encrypted secret to be conveyed to B.

5.5.4 Low Complexity Hardware

Probabilistic KPSs are especially well suited for application scenarios where computations with secrets are performed inside a low-complexity tamper-responsive boundary. PKPSs requires only a secure PRF inside the module, which can be used for encryption, hashing, and public function operations. Note that a PRF is required *in any case* to perform computations (encryption of messages and computing message authentication codes) *using* pairwise secrets—irrespective of whether we use PKPSs or deterministic KPSs. Implementation of finite-field arithmetic required for Blom's SKGS can be more expensive as *additional* circuitry will be required for this purpose.

5.5.5 Multiple KDCs and Renewal

Probabilistic KPSs lend themselves readily to seamless renewal of keys, and simple strategies for employing multiple independent KDCs. Two (n, p)-secure PKPSs can be combined to yield an (n, p^2)-secure PKPS. An (n, p)-secure RPS/HARPS scheme with parameters (P, k) can actually be s parallel deployments of (n, p_i)-secure schemes with parameters (P, k_i) where $\prod_{i=1}^{s} p_i = p$, and $\sum_{i=1}^{s} k_i = k$. Thus, parallel deployments (controlled by independent KDCs) can be realized without any loss in efficiency. Furthermore, to facilitate seamless renewal, $s-1$ of the s systems can be used during the finite period required for renewing the secrets of one of the s systems.

In deterministic Blom's SKGS changing even one of the P secrets chosen by the KDC (the symmetric matrix $\mathbf{D}$) will result in modification of *every* secret assigned to *every* node. Thus, renewal of keys involves replacing *all* secrets assigned to *every* node. This may be very difficult to achieve seamlessly (without interrupting the operation of the deployment). Also note that if two n-secure deterministic schemes are deployed in parallel (controlled by two independent KDCs; each node receives $\mathcal{O}(n)$ secrets from each KDC), the resulting KPS, where shared secrets from both KPSs are used to establish pairwise secrets, is still only n-secure.

5.5.6 Exploiting Multi-path Diversity

Yet another advantage of PKPSs (compared to deterministic KPSs) is their ability to exploit "multi-path diversity" [51]. In a network where A and B have multiple independent communication paths—say $A \rightarrow B$, $A \rightarrow C \rightarrow B$ and $A \rightarrow D \rightarrow B$, A can send (to B) three independent components of a session secret over the three paths. In order to gain access to the session secret, the attacker has to compromise K_{AB}; K_{AC} *or* K_{CB}; and K_{AD} *or* K_{DB}.

For an (n, p)-secure scheme where the probability that the attacker can compute a specific secret is $p << 1$, the probability that the attacker can compute s specific secrets is $\mathbb{O}(p^s) << p$. Thus, at the cost of some bandwidth overhead, a (n, p)-secure PKPSs can be rendered (n, p^s)-secure. For deterministic schemes, due to the catastrophic onset of failure, the attacker gains access to *all* secrets or *none*. Consequently, deterministic schemes cannot take advantage of multi-path diversity.

5.5.7 Conclusions

Probabilistic KPSs have several compelling advantages over deterministic KPSs. The most important advantage is the need for lower overhead inside the protected environment in which security sensitive operations like performing computations with secrets need to be performed. This feature, coupled with the fact that the devices can easily take advantage of external untrusted resources for performing more resource intensive tasks, renders PKPSs even more appealing.

Chapter 6
Scalable Extensions of Nonscalable Schemes

Ultimately, even if probabilistic key predistribution schemes (PKPS) permit offloading of complexity to eternal resources, the achievable security is still restricted by the computational overhead for the public function $F()$ (which increases linearly with $k \propto n$). While storage for even several million encrypted keys (16 MB for a million encrypted 128-bit keys) is unlikely to be prohibitive for almost any conceivable device, the overhead for generating millions of indexes and identifying common indexes is far from acceptable.

A new class of PKPSs based on multiple parallel instances of nonscalable schemes overcome this limitation. In this chapter we discuss three such (n, p)-secure scalable key predistribution schemes (KPS). Specifically,

1. m parallel instances of the nonscalable basic key distribution scheme (KDS), where each instance supports a network size of l, yields the parallel basic key predistribution scheme (PBK) scheme;
2. m instances of the Leigton–Micali KDS, where each instance supports a network size of l, yields the parallel Leighton–Micali (PLM) scheme.
3. m instances of the identity ticket scheme, where each instance supports a network size of l, yields subset keys and identity tickets (SKIT) scheme.

6.1 Parallel Basic KPS

In the "basic" KDS, for a network of N nodes the KDC chooses $\binom{N}{2}$ secrets and each node is provided with $N - 1$ secrets. PBK is a scalable extension of the basic KDS where m independent deployments of basic KDS, each catering only for a network size of l, are used in parallel. The m systems cater for practically unrestricted network sizes.

In the PBK scheme [59, 60], the KDC chooses m sets of secrets $\mathbb{S}^1 \cdots \mathbb{S}^m$, where *each* set consists of $\binom{l}{2}$ secrets:

$$\mathbb{S}^i = \{K_i(j_1, j_2)\}, 0 \le i \le m, 0 \le j_1, j_2 \le l - 1, \tag{6.1}$$

where $K_i(j_1, j_2) = K_i(j_2, j_1)$.

M. Ramkumar, *Symmetric Cryptographic Protocols,* DOI 10.1007/978-3-319-07584-6_6,

A node with identity A is assigned m "short-IDs," one in each of the m parallel systems. Each short-ID is $\log_2(l)$-bit long (for $l = 1024$ each short-ID will be 10 bits). A simple one-way function

$$f(A) = \{a_0 \cdots a_{m-1}\}, 0 \leq i \leq m-1, \tag{6.2}$$

where a_i's are uniformly distributed between 0 and $M - 1$ is employed to assign such short-IDs. Note that the function $f()$ merely needs to produce a pseudo-random sequence of $m \log_2 l$ bits. In practice, if a 160-bit PRF $h()$ is also used to realize $f()$, A could be repeated hashed $m \log_2 l/160$ times to generate the m short identities. For example, if $m = 60$ and $\log_2 l = 10$, then a pseudo random sequence $\{a_0 \cdots a_{59}\}$ of 600 bits can be generated by hashing the identity A four times.

Node A assigned short-IDs $a_0 \cdots a_{m-1}$ is issued l secrets from *each* of the m parallel systems—corresponding to the short-ID a_i in each system. Specifically, node A with short-IDs $a_0 \cdots a_{m-1}$ is issued $k = m \times l$ secrets

$$\mathbb{S}_A = \{K_i(a_i, j)\}, 0 \leq i \leq m-1, 0 \leq j \leq l-1 \tag{6.3}$$

Node B, like wise, issued $k = lm$ secrets $\mathbb{S}_B = \{K_i(b_i, j)\} 0 \leq i \leq m-1, 0 \leq j \leq l-1$.

In practice the KDC can choose m master secrets $\mu_0 \cdots \mu_{m-1}$ and compute any $K_i(j_1, j_2)$ as

$$K_i(j_1, j_2) = K_i(j_2, j_1) = \begin{cases} h(\mu_i \parallel j_1 \parallel j_2) & j_1 \geq j_2 \\ h(\mu_i \parallel j_2 \parallel j_1) & j_1 < j_2 \end{cases}$$
$$\text{where} 0 \leq i \leq m-1, 0 \leq j_1, j_2 \leq l-1 \tag{6.4}$$

Nodes A and B possess m shared secrets

$$S_i(A, B) = K_i(a_i, b_i) = K_i(b_i, a_i), 0 \leq i \leq m-1 \tag{6.5}$$

Specifically, by executing $f(A)$ and $f(B)$ both nodes can determine their respective short-IDs $\{(a_i, b_i)\}$ in each of the m parallel deployments, and the corresponding shared key in each deployment. All m shared secrets are used to derive the pairwise secret K_{AB}.

6.2 Parallel Leighton–Micali Scheme (PLM)

Recall that in the Leighton–Micali KDS for a small network consisting of l nodes, each node receives one secret. In addition, there exist $\binom{l}{2}$ public values corresponding to each possible pair of nodes.

PLM [61] employs m parallel instances of the nonscalable LM scheme, where every node receives one secret from each of the m parallel deployments, corresponding to a short-ID in each deployment.

Similar to PBK, PLM employs a public function $f()$, which when seeded by the identity of a node yields m short-IDs—each $\log_2 l$-bits long.

The KDC chooses m master secrets $\mu_0 \cdots \mu_{m-1}$. A node with identity A is assigned m short-IDs $[a_0 \cdots a_{m-1}] = f(A)$, and receives m secrets,

$$\mathbb{S}_A = \{K_{a_i}(i) = h(\mu(i) \parallel a_i)\}, 0 \leq i \leq m-1, \tag{6.6}$$

one corresponding to each deployment.

Corresponding to any two secrets $K_{j_1}(i)$ and $K_{j_2}(i)$ is a public value

$$P^i_{j_1,j_2} = P^i_{j_2,j_1} = h(K_{j_1}(i) \parallel j_2) \oplus h(K_{j_2}(i) \parallel j_1). \tag{6.7}$$

Two nodes A and B can compute m common secrets—one in each of the m deployments. Specifically, A and B determine their respective short IDs $[a_0 \cdots a_{m-1}]$ and $[b_0 \cdots b_{m-1}]$ in each deployment, and proceed to compute a secret

$$S_{A,B}(i), 0 \leq i \leq m-1 \tag{6.8}$$

in each deployment. For computing the common secret only one node uses the public value. For example, if A with short-ID a_i has access to the public value $P^i_{a_i,b_i}$ the common secret is

$$\begin{aligned} S_{A,B}(i) &= h(K_{b_i}(i) \parallel a_i) \\ &= h(K_{a_i}(i) \parallel b_i) \oplus P^i_{a_i,b_i} \end{aligned} \tag{6.9}$$

Instead, if B with short-ID b_i has access to the public value $P^i_{a_i,b_i}$ the common secret is

$$\begin{aligned} S_{A,B}(i) &= h(K_{a_i}(i) \parallel b_i) \\ &= h(K_{b_i}(i) \parallel a_i) \oplus P^i_{a_i,b_i} \end{aligned} \tag{6.10}$$

To unambiguously determine which of the two should use the public value, the KDC specifies a rule $pv()$ based on their short-IDs. Thus, if $pv(a_i, b_i) = a_i$ the entity A with secret $K_{a_i}(i)$ should have access to the public value $P_{a_i,b_i}(i)$. Similarly, if $pv(a_i, b_i) = b_i$, B requires access to the corresponding public value and A does not. The rule can be expressed as

$$pv(j_1, j_2) = \begin{cases} \min(j_1, j_2) & (j_1 + j_2) \equiv 1 \pmod 2 \\ \max(j_1, j_2) & (j_1 + j_2) \equiv 0 \pmod 2 \end{cases} \tag{6.11}$$

For example, $pv(6,3) = 3$, $pv(4,8) = 8$, $pv(5,7) = 7$. In plain English the rule $pv()$ translates to "if sum of the two inputs is odd choose the minimum of the two; if then sum is even choose the maximum." According to this rule, the shared secret

$$S_{A,B}(i) = \begin{cases} h(K_{b_i}(i) \parallel a_i) & pv(a_i, b_i) = a_i \quad \text{A uses pub. value} \\ h(K_{a_i}(i) \parallel b_i) & pv(a_i, b_i) = b_i \quad \text{B uses pub. value} \\ \end{cases} \tag{6.12}$$

When public values are allocated according to this rule, every node will need to store exactly $l/2$ public values corresponding to each deployment (or a total of $ml/2$ public vales). More specifically, in a deployment i, a node with short-ID a_i stores the following $l/2$ public values:

1. If a_i is even, A stores public values $P^i_{a_i,j}$ for all *even* js *less than* a_i, and all odd js greater than a_i.
2. Corresponding to an odd a_i, A stores public values $P^i_{a_i,j}$ for all *odd* js *less than* a_i, and all even js greater than a_i.

6.3 (n, p)-Security of PBK and PLM

For PBK and PLM, a secret $S_{A,B}(i)$ is also computable by a node C if $[c_0 \cdots c_{m-1} = f(C)$ is such that $c_i = a_i$ or $c_i = b_i$ (or $c_i = a_i = b_i$).

The probability that some node C can compute $S_{A,B}(i)$ is thus

$$\gamma = \frac{2\,l}{l^2 - 1} \approx \frac{2}{l}, \tag{6.13}$$

where the approximation holds for large l. The probability that an entity which has access to secrets of n nodes *can not* compute $S_{A,B}(i)$ (or the key $S_{A,B}(i)$ can be safely used by A and B) is thus

$$s_i(n) = (1 - \gamma)^n \approx \mathrm{e}^{-\mathrm{n}\gamma}. \tag{6.14}$$

The probability that the entity (with access to secrets of n nodes *can* compute *all* $S_{A,B}(i), 0 \leq i \leq m - 1$ (or all m keys are unsafe) is

$$p(n) \approx (1 - \mathrm{e}^{-\mathrm{n}\gamma})^{\mathrm{m}} \approx (1 - \mathrm{e}^{-2\mathrm{n}/\mathrm{l}})^{\mathrm{m}}. \tag{6.15}$$

where the first approximation follows from the well known identity $(1-1/x)^x \approx \mathrm{e}^{-1}$ for large x.

6.3.1 *Optimal Choice of Parameters* m *and* M

For a target $p(n) \leq p^* \forall n \leq n^*$ it is beneficial to choose m and l so as to minimize $k = ml$. Note that in PBK, every node needs to store ml secrets. In PLM every node needs to store m secrets and $ml/2$ public values. As we shall see shortly, typically $m << l$. Thus, for both PBK and PLM it is advantageous to minimize $k = ml$.

From Eq. (6.15), replacing m with k/l we have,

$$k = \frac{2n \log{(1/p)}}{-x \log{(1 - \mathrm{e}^{-\mathrm{x}}}}) \text{ where } x = \frac{2n}{l}. \tag{6.16}$$

Table 6.1 Reducing m

b	2	3	4	5	8
l'/l	2.409	5.191	10.74	21.83	177.1
k'/k	1.204	1.730	2.685	4.366	22.137

Minimizing k calls for maximizing $-x\log(1-\mathrm{e}^{-\mathrm{x}})$, which occurs when $\mathrm{e}^{-\mathrm{x}} = 0.5$, or $x = \log(2)$. Substituting $\mathrm{e}^{-\frac{2\mathrm{n}}{\mathrm{l}}} = 1/2$ in Eq. (6.15), we have

$$\begin{aligned} p(n) &= \left(\frac{1}{1-1/2}\right)^m = \frac{1}{2^m} \\ m &= \log_2(1/p) = \log(1/p)/\log(2) \\ l &= 2n/\log(2) \approx 2.885n \\ k &= ml = \frac{2}{(\log(2))^2} n\log(1/p) \approx 4.16n\log(1/p) \end{aligned} \tag{6.17}$$

If we desire to reduce m by a factor b (to $m' = \frac{m}{b} < m$) but still realize the same $p(n)$ we can choose $l' > l$ such that

$$\begin{aligned} p(n) &= (1-\mathrm{e}^{-\frac{2\mathrm{n}}{\mathrm{l}}})^{\mathrm{m}} = (1-\mathrm{e}^{-\frac{2\mathrm{n}}{\mathrm{l}'}})^{\mathrm{m}'} \Rightarrow \\ \mathrm{e}^{-\frac{2\mathrm{n}}{\mathrm{l}'}} &= 1 - 1/2^b \end{aligned} \tag{6.18}$$

Thus, it is required to increase l' (and $k' = l'm'/2$) to

$$l' = -2n/\log(1-1/2^b) \quad k' = k\frac{\log(1-1/2)}{b\log(1-1/2^b)} \tag{6.19}$$

Table 6.1 depicts the trade-off involved in reducing m (by a factor b) vs. the corresponding increases in M and the total storage $k = ml$ (or $k/2 = lm/2$ for PLM).

Decreasing m by a factor of 3 ($b = 3$) calls for increasing storage k by a factor 1.73 (and increasing l by a factor 3 × 1.73). As an example, PBK/PLM scheme with parameters ($m = 64, l = 2^{16}$) is the storage efficient choice to meet the requirement $p(22500) < 2^{-64}$. The choice of ($m = 24, l = 2^{18}$) instead meets the requirement $p(22500) < 2^{-64}$ with a 1.5 fold increase in storage requirement.

6.4 Subset Keys and Identity Tickets (SKIT)

Similar to PBK and PLM, the key subset and identity tickets (SKIT) scheme [62] is defined by two parameters (m, l), where typically $l >> m$. Similar to PLM and PBK, the KDC chooses

1. m master keys $\mu(i), 0 \le i \le m-1$
2. One-way function $f()$ which when seeded by an identity generates m $\log_2 l$-bit short-IDs.

A node with identity A receives a set m secrets

$$\mathbb{S}_A = \{K_{a_i}(i) = h(\mu(i) \parallel a_i)\}, 0 \leq i \leq m-1, \tag{6.20}$$

In addition, A is issued *ml tickets* corresponding to its identity

$$\mathbb{T}_A = \{h(K_j(i) \parallel A)\}, \tag{6.21}$$

for all $0 \leq i \leq m-1$ and $0 \leq j \leq l-1$.

Two nodes A and B share 2 m tickets

1. Node A can compute m of B's stored tickets using its m secrets.
2. Node B can compute m of A's stored tickets using its (B's) m secrets.

The 2 m common tickets are

$$S_{A,B}(i) = h(K_{a_i}(i) \parallel B), 0 \leq i \leq m-1 \tag{6.22}$$

$$S_{B,A}(i) = h(K_{b_i}(i) \parallel A), 0 \leq i \leq m-1 \tag{6.23}$$

The 2 m common tickets are XORed together realize the shared secret K_{AB}. To compute K_{AB}, A evaluates $f(B)$ and fetches m stored tickets. A then computes m additional tickets (stored by B).

6.4.1 (n, p)-Security of SKIT

A node C can compute $S_{A,B}(i)$ if $c_i = a_i$ (probability $1/l$). The probability that an entity who has access to secrets of n nodes *cannot* compute $S_{A,B}(i)$ (or the probability that $S_{A,B}(i)$ is safe) is

$$s_i(n) = (1 - 1/l)^n \approx \mathrm{e}^{-\mathrm{n}/\mathrm{l}}. \tag{6.24}$$

Similarly, C can compute $S_{B,A}(i)$ only if $c_i = b_i$. The probability that an entity which has access to secrets of n nodes *cannot* compute $S_{B,A}(i)$ is also $s_i(n) \approx \mathrm{e}^{-\mathrm{n}/\mathrm{l}}$.

Thus, the probability that the n-attacker collusion *can* compute a specific ticket (of the $2m$ tickets common to A and B) is

$$\gamma = (1 - \mathrm{e}^{-\mathrm{n}/\mathrm{l}}) \tag{6.25}$$

An entity with access to secrets of n nodes can compute *all* 2 m tickets (and thus compute K_{AB}) with a probability

$$p(n) = \gamma^{2\,m} \approx (1 - \mathrm{e}^{-\mathrm{n}/\mathrm{l}})^{2\,\mathrm{m}}. \tag{6.26}$$

6.4.2 Optimal Choice of Parameters

As m is typically small, the number of secrets m is not an issue. However, as every node needs to store $k = ml$ tickets, it is advantageous to minimize $k = ml$ to achieve the desired (n^*, p^*)-security.

Comparing Eq. (6.15) for $p(n)$ for PBK/PLM, with Eq. (6.26)

$$\begin{aligned} p(n) &\approx (1 - \mathrm{e}^{-2\mathrm{n}/\mathrm{l}})^{\mathrm{m}} \text{ PLM/PBK} \\ p(n) &\approx (1 - \mathrm{e}^{-\mathrm{n}/\mathrm{l}})^{2\mathrm{m}}. \text{ SKIT} \end{aligned} \tag{6.27}$$

we can see that using SKIT we can afford to

1. Reduce l by a factor 2
2. Reduce m by a factor 2

to achieve the same $p(n)$ characteristics as PBK/PLM!

The choice of parameters that minimizes $k = ml$ are thus

$$\left.\begin{aligned} m = & \quad \tfrac{\log(1/p^*)}{2\log 2} = \log(1/p^*)/2 \\ l = & \quad \tfrac{n^*}{\log 2} = \end{aligned}\right\} \Rightarrow k = \frac{1}{2\log^2 2} n^* \log(1/p^*) \tag{6.28}$$

As a concrete example, SKIT with parameters $m = 32$ and $l = 2^{11}$ will have identical $p(n)$ characteristics as PBK/PLM with $m = 64$ and $l = 2^{12}$. All schemes satisfy $p(n) \leq 2^{-64} \forall n < 1420$. SKIT requires only one fourth the storage of PBK (where each node needs to store ml secrets) and half of that of PLM (where each node requires access to $ml/2$ public values).

As with PBK and PLM it is possible to reduce m by increasing l to achieve the desired $p(n)$. As with PBK/PLM decreasing m by a factor of 3 calls for increasing storage k by a factor 1.73 (and increasing l by a factor 3×1.73). Decreasing m from 32 to 12 calls for a four-fold increase in l, and a 1.5 fold increase in k. In other words, SKIT with $(m = 32, l = 2^{11})$ and $(m = 12, l = 2^{13})$ meet the requirement $p(1420) < 2^{-64}$.

6.5 Comparison of KPSs

Consider PBK/PLM with parameters $m = 64$ and $l = 2^{12}$ satisfying $p(n = l/2.885 \approx 1420) \approx 2^{-64}$. For PBK, each node will need to store $ml = 2^{18}$ secrets. If each secret is 128-bits long, the total storage required is about 5 MB. For PLM each node will need to store $m = 64$ secrets and $ml/2 = 2^{17}$ public values—about 2.5 MB of storage per node.

For SKIT, the same $p(n)$ characteristics can be obtained by choosing $m = 32$ and $l = 2^{11}$. Each node will need to store 32 secrets and $ml = 2^{16}$ identity tickets. If each ticket is 128-bits long, the total storage required is about 1.25 MB.

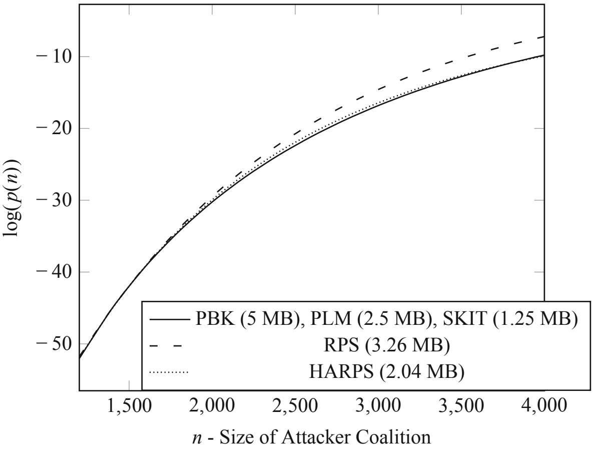


Fig. 6.1 Comparison of 5 (n, p) secure KPSs. PBK, PLM, and SKIT have identical $p(n)$ characteristics. The storage per node required for the different schemes are PBK—5 MB, PLM—2.5 MB, SKIT—1.25 MB, RPS—3.26 MB, and HARPS—2.04 MB

To meet the same requirement, viz., $p(n) \leq 2^{-64} \forall n \leq 1420$, the choice of parameters for random preloaded subsets (RPS) (with $L = 1$) and Hashed random preloaded subsets (HARPS) (with $L = 64$) will be

$$\begin{aligned} k_r &= n^* e \log(1/p^*) \approx 171233 \\ P_r &= (n^* + 1)k_r \approx 243322093 \\ k &= 1.70 n^* \log(1/p^*) \approx 107088 \\ P &= 0.96(n^*)^2 \log(1/p^*) \approx 85872352. \end{aligned} \tag{6.29}$$

For 20 byte (128-bits) keys the storage (for each node) required for RPS and HARPS are 3.26 and 2.04 MB respectively.

The public function complexity for PBK, PLM, and SKIT is trivial compared to RPS and HARPS. The public function calls for pseudo random generation of m, $\log_2 M$ values (and not k values as in RPS/HARPS).

In PBK $m = 64$ secrets need to be fetched from storage and used for computing any pairwise secret. For PLM, on an average, $m/2 = 32$ values need to be fetched from storage. For SKIT, $m = 32$ identity tickets need to be fetched. For RPS the average number of secrets needed for computing a pairwise secret is $m = 121$. For HARPS, $m = 133$. The operations with secrets are also less complex in PBK, PLM, and SKIT ($m = 64/m = 32$ instead of $m = 121$ or $m = 133$).

To summarize, for RPS and HARPS both storage overhead and public function complexity were proportional to n. For PBK, PLM, and SKIT only the storage complexity is linear in n. All other facets of complexity are proportional to $m \propto \log(1/p)$.

The three plots in Fig. 6.1 depict the $p(n)$ characteristics of the five schemes (PBK, PLM, and SKIT have identical $p(n)$ characteristics). Apart from substantially lower complexity, SKIT, PLM, and PBK also boast slower degradation of security with increasing n, compared to RPS (and similar to HARPS in this respect).

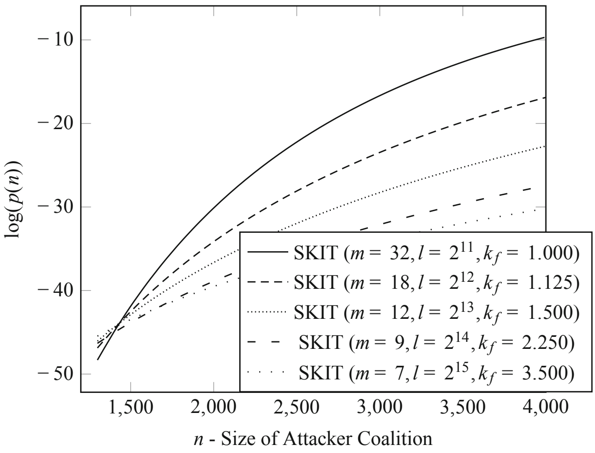


Fig. 6.2 Comparison of five realizations (with different parameters m, l) of SKIT designed to meet the same (n, p)-security. The increase in storage (factor k_f) compared to the scheme with storage optimized choice of parameters (or $k_f = 1$) is indicated for each plot

If storage is not an expensive resource, and as all other facets of complexity is proportional to m, it may be advantageous to lower m (at the cost of increasing storage mM).

Figure 6.2 depicts the performance (n vs $\log(p)$ curves) of SKIT schemes designed to meet $p(n) \leq 2^{-64} \forall n \leq 1420$. The solid line represents the performance of schemes designed to minimize $k = ml$. The dashed and dotted lines represent four different choices for m, l that reduce m and increase l, and consequently, the storage $k = ml$ by a factor $K_f > 1$. Note that the plots are applicable even for PLM/PBK with double the value of m and l.

Apart from reduced public and private function complexity, schemes with higher k_f (and smaller m) enjoy slower degradation of security. We already saw in Fig. 6.1 that even the choice of $k_f = 1$ results is slower degradation of PBK/PLM/SKIT security compared to RPS. Choosing SKIT/PBK/PLM with $k_f > 1$ further slows down the degradation.

At first sight it may appear that such trade-offs, viz., reducing $m = ak$—the number of shared secrets—by increasing k beyond the optimal (k-minimizing) choice, and reducing $a = k/P$ (to a value less than $1/n$) — can also be performed for subset allocation schemes like RPS and HARPS. However, for RPS and HARPS m can never be reduced below $\log(1/p)$. To see this note that for small $a = k/P$ (as we need to reduce a by a larger factor for reducing $m = ak$)

$$\begin{aligned} p(n) &= (1 - a(1-a)^n)^k \approx (1 - \xi \mathrm{e}^{-\mathrm{na}})^{\mathrm{m/a}} \Rightarrow \\ m &\approx \frac{a \log(p)}{\log(1 - \xi \mathrm{e}^{-\mathrm{na}}} \approx \frac{a \log(1/p)}{-a \mathrm{e}^{-\mathrm{na}}} \\ &= \log(1/p) \mathrm{e}^{\mathrm{na}} \end{aligned} \tag{6.30}$$

Thus, even for $a \to 0$, m cannot be reduced below $\log(1/p)$. Reducing m by a factor $b < \mathrm{e}$ will require $a = \frac{1 - \log b}{n}$ calling for increase in k by a factor $\frac{1}{b(1 - \log b)}$. For example, for $b = 2$ (say, reducing m from 121 to about 60) calls for increasing k by a factor 1.66 (for PBK/PLM/SKIT reducing m by a factor 2 increases k only by a factor 1.2). Reducing m to 45 will necessitate a 35 fold increase in the value of k.

6.6 Beyond (n, p)-Security

Thus so far we have used the simple (n, p)-security metric to evaluate the strength of probabilistic KPSs. For a P-KPS designed to be (n^*, p^*)-secure (or $p(n^*) \leq p^* \forall n < n^*$) an attacker who has compromised all secrets from n nodes can compute any specific shared secret with a probability $p(n)$.

For example, if such an attacker with access to secrets from n nodes desires to eavesdrop on exchanges between two specific nodes A and B, there is a probability $p(n)$ that the attacker will be successful in discovering the shared secret K_{AB}. Alternately, as the attacker can expose a fraction $p(n)$ of illegitimate pairwise secrets, a nonpicky attacker may have the freedom to search for identities X and Y such that K_{XY} is computable by the attacker. In other words, an attacker seeking any illegitimate key has two avenues:

1. Compromise nodes and expose secrets (to increase the probability p of determining any shared secret), and/or
2. Perform brute-force search to identity pairs that can be compromised using the exposed secrets.

In identity based schemes the identity space $\mathbb{I}$ is typically substantially larger than the actual number of nodes that may be deployed. Specifically, it is common practice in identity based schemes to choose descriptive identities (for example, "Alice B. Cryptographer, Anytown, U.S.A"), or the (128/160 bit) cryptographic hash of a descriptive identity as the identity. If 128-bit identities are used, and if only billion nodes ($\approx 2^{30}$) actually exist in the network, only one in 2^{98} identities correspond to nodes that actually exist. Obviously, most pairwise secrets that the attacker may be able to compute (using secrets exposed from other nodes) will correspond to nonexistent nodes.

In a scenario where an attacker has successfully (through brute-force search) determined such an X and Y, there are thus three possibilities:

1. Both X and Y are nonexistent nodes
2. One node (say, X) is nonexistent while the node Y is a real node
3. Both X and Y are real nodes

The first scenario, where the attacker has unrestricted freedom to search for X and Y, is not of much use to the attacker. In the third scenario, which is obviously useful for the attacker, the attacker's freedom to search is fortunately limited by the number of actual nodes. However, the second scenario may also be useful for the attacker to carry out some attacks.

As an example, consider a scenario where an attacker desires to impersonate a (for example, temperature) sensor to provide misleading information to one or many target nodes. The attacker has practically unrestricted freedom to generate identities that will be recognized by potential targets as a "temperature sensor." The attacker is not constrained to impersonate a real sensor. Leveraging this freedom, the attacker may be able to search for a suitable identity X, such that K_{XY} is computable for a substantially larger fraction of identities in $Y \in \mathbb{I}$ (using the secrets he has exposed by tampering with several nodes). With the increased probability of finding a suitable Y, the attacker has a better chance of identifying an actually exploitable target Y.

More formally,

1. Let $\mathbb{I}$ represent the large space of identities assigned to nodes.
2. Let $\mathbb{A} = \{A_1 \cdots A_r\}$ represent the identities of a set of r nodes from which an attacker has exposed all secrets.
3. Let $\mathbb{T} = \{T_1 \cdots T_s\}$ represent the identities of a set of s nodes—the potential *target* set for the attacker.
4. Let $\mathbb{X} = \mathbb{I} \setminus \mathbb{A}$ represent the set of identities of nodes that the attacker may attempt to illegitimately impersonate.

It is important to remember that while $\mathbb{A}$ and $\mathbb{T}$ represent nodes that physically exist, most of the identities in $\mathbb{X}$ will correspond to nodes that do *not* actually exist. The attacker can however *search for a subject* $X \in \mathbb{X}$ that he can impersonate, and an *object* $T_i \in \mathbb{T}$, such that K_{XT_i} is computable using the secrets pooled from r nodes. The only constraint on the ability of the attacker in choosing a suitable subject for the attack is the *brute-force search complexity* that can be borne by the attacker. The target (or object) of the attacks are however constrained to be nodes that actually exist, and to which the attacker can physically send messages (either directly or over multiple hops). We shall henceforth refer to such attacks as message injection attacks.

To characterize the resistance of PKPSs to message injection attacks let us consider a (n, ϕ, p_a) security model where $p_a(n, \phi)$ represents the "capability of an attacker" who has

1. Access to secrets of n randomly chosen nodes
2. Perform $1/\phi$ brute-force searches

Thus, for example, $p_a(1000, 2^{-40})$ is the probability of a successful message injection attack by an attacker who is willing to perform a brute force search of up to 2^{40} different identities from $\mathbb{X}$ in order to discover a $X \in \mathbb{X}$ such that K_{XY} is computable for a fraction p_a of all nodes.

6.6.1 (n, ϕ, p_a)-Security of RPS

For carrying out a message injection attack the attacker seeks some $X \in \mathbb{I}$ such that all/most keys assigned to X are available from the r compromised nodes. Specifically, if the set of k keys assigned to $X \in \mathbb{I}$ is $\mathbb{S}_X$, and if $\mathbb{S}_r^A$ represents set of all secrets accumulated from r compromised nodes, the attacker can search for a suitable X such that $\mathbb{S}_r^A \cap \mathbb{S}_X$ is maximized.

Consider RPS with parameters (P,k), where $t = 1/a = P/k \approx n^*$ and $k \approx n^* \mathrm{e} \log(1/\mathrm{p}^*)$. The probability that a particular key of X is *not* found in key ring of a specific node A_1 in the attacker pool is $(1 - 1/t)$. Thus, the probability that none of the r nodes (in the attacker pool) possess the specific key (or the probability that the key is not found in the set $\mathbb{S}_r^A$) is

$$\epsilon = (1 - 1/t)^r \approx \mathrm{e}^{-\mathrm{r/t}} \tag{6.31}$$

The probability that at least k_s of the k keys of X are included in the attacker pool is then

$$\phi(r, k_s) = \sum_{i=k_s}^{k} \binom{k}{i} (1-\epsilon)^i \epsilon^{k-i}. \tag{6.32}$$

In other words, by performing $1/\phi$ brute-force searches, the attacker can expect to find some $X \in \mathbb{X}$ such that at least k_s of the k secrets of X are available to the attacker.

In such a scenario, let $\delta = \frac{k-k_s}{k}$ be the fraction of missing keys. If $\delta = 0$ (or $k_s = k$) the attacker can obviously compute *all* pairwise secrets of the form K_{XY} for any Y (or $p_a = 1$). On the other hand, if $\delta > 0$ the probability $p_a < 1$ can be evaluated by considering a k-round game between the attacker and Y.

Let us assume that the index of the keys assigned to X are $x_0 \cdots x_{k-1}$. Each round of this k-round game corresponds to a one of the k keys of X. In each round (say, round (i) the attacker is successful in drawing the index x_i with a probability $1 - \delta$. However, Y succeeds only with a probability $1/t$. Thus, the probability that the attacker looses any round is δ/t.

However, the attacker has to win *all* k rounds to win the game (and compute K_{XY}), for if he looses even one round, one of the keys required for computing K_{XY} is not available to the attacker. The probability that the attacker will win all k rounds (and be able to compute a specific K_{XY}) is

$$p_a(\delta) = (1 - \delta/t)^k \tag{6.33}$$

As $k \approx t \mathrm{e} \log(1/\mathrm{p}^*)$, and as $\mathrm{e} \approx (1 - 1/\mathrm{t})^{\mathrm{t}}$ for large t, we have

$$p_a(\delta) \approx \mathrm{e}^{\delta \mathrm{e} \log(\mathrm{p}^*)}. \tag{6.34}$$

If the design value $p^* = 2^{-64}$, we have $\mathrm{e} \log(2^{-64}) \approx -120.59$. For $120.59\delta \approx 0.693$ (or $\delta \approx 0.00575$) we have $p_a = 0.5$. Thus, the attacker needs about $k_s = 0.99425k$ of the k keys of node X in order to impersonate X successfully for its interactions with half ($p_a = 0.5$) the nodes $Y \in \mathbb{T}$. To be able to inject a message to at least one of the nodes in the set $\mathbb{T}$ (with s nodes) the attacker requires $p_a = 1/s$. For $s \approx 2^{30}$ the attacker requires $\delta \leq 0.1718$. For $s \approx 2^{20}$ the attacker requires $\delta \leq 0.1145$. For $s \approx 2^{10}$ the attacker requires $\delta \leq 0.057$.

Plots of $\log(\phi(r, p_a))$ vs r for various values of p_a are shown in Fig. 6.3 for RPS with parameters $k = 7,902,779, a = 2^{-16}$ (designed to meet $p(n^* = 2^{16}) = p^* = 2^{-64}$). For achieving a desired p_a the attacker can either compromise more nodes (by increasing r) or perform more brute-force searches (reduce ϕ).

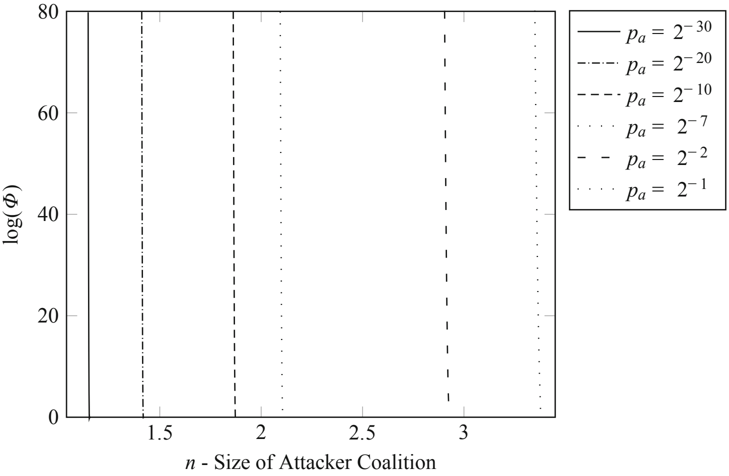


Fig. 6.3 (r, ϕ, p_a)-security characteristics of RPS. Note that for a desired p_a, ϕ increases rapidly with a small increase in the attacker pool size r. However increasing p_a requires a substantial increase in r

For a brute-force complexity of 2^{-50}, an attacker with access to secrets from one about $r = 114,000$ nodes can inject a message to one in a billion targets ($p_a = 2^{-30}$); for $r \approx 140,000$ the attacker can inject a message to one in a million targets ($p_a = 2^{-20}$); with $r \approx 184,000$ the attacker can inject a message to one in a thousand targets ($p_a = 2^{-10}$); with $r = 335,500$ the attacker can inject messages to every other node ($p_a = 1/2$).

The fact that the plots for $\phi(r, p_a)$ vs r are almost *vertical* indicates that even a small increase in r can significantly reduce the brute-force complexity for the attacker. In other words, the attacker *cannot* use brute-force capability productively to reduce r (the number of nodes from which secrets need to be exposed).

6.6.2 (n, ϕ, p_a)-Security of PBK/PLM

As the conditions necessary for an attacker to determine K_{XY} are identical for PBK and PLM, let the short-identities assigned to X be $x_0 \cdots x_{m-1}$. If any of the r nodes in the attacker pool is also assigned x_0 as its first short identity, then the attacker has access to all the l secrets (in PBK) or the 0th deployment secret corresponding to short-identity x_0 (in PLM). More generally, if any of the r nodes are assigned the same short-ID x_i as X in the ith deployment, then the attacker has all keys of X corresponding the ith deployment.

The probability that a specific short identity x_i is also assigned to a node $A_1 \in \mathbb{A}$ is $1/l$. The probability that none of the r nodes in the attacker's pool is assigned the

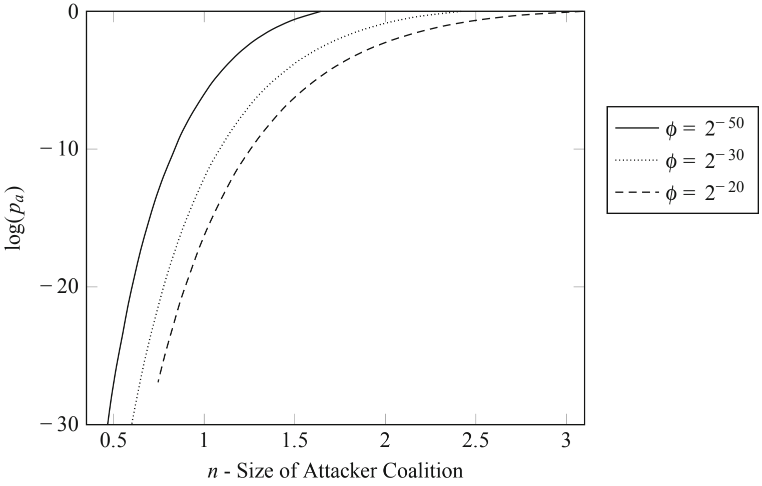


Fig. 6.4 (r, ϕ, p_a)-security characteristics of PBK/PLM

short identity x_i in the ith deployment is

$$\epsilon = (1 - 1/l)^r \approx \mathrm{e}^{-\mathrm{r}/\mathrm{l}}. \tag{6.35}$$

The probability that at least $m_s \leq m$ short identities are assigned to the nodes in the attacker pool is thus

$$\phi(r, m_s) = \sum_{i=m_s}^{m} \binom{m}{i} (1 - \epsilon)^i \epsilon^{m-i}. \tag{6.36}$$

In a scenario where the attacker has access to m_s of the m sets of keys assigned to X, the key K_{XY} can still be computed if for the other $m - m_s$ deployments, at least one of the r nodes in $\mathbb{A}$ is assigned the same short identity as Y.

The probability that a specific node in $\mathbb{A}$ is assigned the same short ID as Y in a specific deployment is $1/M$. The probability that none of the r nodes are assigned the same short-ID as Y in the ith deployment is $(1 - 1/M)^r = \mathrm{e}^{\mathrm{r}/\mathrm{l}}$. Thus, given that the attacker has found an X such that m_s sets of secrets of X are available to the attacker, the probability p_a that the remaining $m - m_s$ keys can be computed is

$$p_a(r, m_s) = \left(1 - \mathrm{e}^{-\mathrm{r}/\mathrm{l}}\right)^{m-m_s}. \tag{6.37}$$

Figure 6.4 depicts plots of p_a vs. r for three different values of ϕ for PBK/PLM with parameters $\tilde{m} = 64$ and $\tilde{l} = 189{,}097$ (designed to meet the requirement $p(n^* = 2^{16}) = p^* = 2^{-64}$). Note that (unlike RPS), for PBK/PLM ϕ decreases gracefully with increasing r. Furthermore, realizing large p_a (close to one) is easier in PBK/PLM.

An attacker capable of performing 2^{50} brute-force searches ($\phi = 2^{-50}$) can achieve $p_a = 2^{-30}, 2^{-20}, 2^{-10}$ respectively with pool sizes of $r = 54{,}000$, $r = 73{,}000$ and

$r = 96,000$ respectively. With $r = 165,000$ nodes the attacker can realize $p_a = 1$ (plot labeled $\phi = 2^{-50}$ in the figure). However, if the brute-force capability is 2^{30}, the attacker requires substantially higher $r \approx 240,000$ to realize $p_a \approx 1$, thus, clearly demonstrating that the attacker can indeed take advantage of brute-force search capability.

6.6.3 (n, ϕ, p_a)-Security of SKIT

For SKIT with parameters m and l, to find a shared secret K_{XY}, the attacker needs access to the m secrets corresponding to the identity X *and* the m secrets corresponding to the identity Y. The probability that the attacker's pool of secrets will include all secrets of a specific node X is

$$\phi(r) \approx (1 - \mathrm{e}^{-\mathrm{n}/\mathrm{l}})^{\mathrm{m}} \tag{6.38}$$

Also the fraction of $Y \in \mathbb{X}$ for which all m secrets are included in the attacker's pool is

$$p_a(r) = \phi(r). \tag{6.39}$$

In other words, the attacker *cannot* take advantage of brute-force search capability (or low ϕ) to choose a subject, as a low ϕ also implies low p_a.

For $(n^* = 2^{16}, p^* = 2^{-64})$-secure SKIT with parameters $m = 32$ and $l = 94,548$, for target pool sizes of $s = 10^3$, $s = 10^6$, and $s = 10^9$ (or $p_a = 2^{-10}, 2^{-20}, 2^{-30}$ respectively) the size of the required attacker pool size is $r = 155,000$ (for $p_a = 2^{-10}$), $r = 99,000$ (for $p_a = 2^{-20}$) and $r = 70,500$ (for $p_a = 2^{-30}$). For $\phi = p_a \approx 1/2$ the attacker requires $r = 363,000$.

A comparison of the resiliency of three P-KPSs: RPS, PBK/PLM and SKIT designed to meet the same (n, p)-security criterion are tabulated in Table 6.2. The table depicts the number of nodes r required in the attacker pool to inflict message injection attacks with a probability p_a (for $p_a = 2^{-30}, 2^{-20}, 2^{-10}, 1/2$). For PBK/PLM (where the attacker *can* take advantage of brute-force search capability) the value r is based on the assumption that a brute-force search capability of 2^{50} is possible for the attacker.

In general, (n, p)-secure RPS exhibits substantially higher resistance to message injection attacks compared to (n, p)-secure PBK/PLM. However, while RPS exhibits better resistance than SKIT for small values of p_a, SKIT offers better resistance for higher values of p_a (close to 1). In the next section we argue why it is *especially* desirable for a PKPS to offer high resistance (larger r) for values of p_a close to one.

6.6.4 Addressing Message Injection Attacks

There are two broad strategies to address message injection attacks. The first is to reduce the search freedom for the attacker by limiting the size of the identity space. This is not desirable as it reduces the inherent advantages of identity based schemes.

Table 6.2 Number of nodes required in the attacker pool for an attacker to carry out message injection attacks

p_a	SI	PBK/PLM	SKIT-1
2^{-30}	114,000	54,000	70,500
2^{-20}	140,000	73,000	99,000
2^{-10}	184,000	96,000	155,000
1/2	337,600	149,000	363,000

The second is by employing multiple "check points." If the multiple check-point defense (MCD) [62] is enforced, to send a message to B, A will need to authenticate itself to B, and possibly some nodes (randomly) designated by B (say C, D). While this is not an issue for a legitimate A (which can compute K_{AC} and K_{AD}), MCD makes it more difficult for an *attacker* to impersonate A, as the attacker will also need access to K_{AC} and K_{AD} (apart from K_{AB}).

In a scenario where an attacker has found (through a brute-force search) some identity A with $p_a = 10^{-3}$ (or he can compute K_{Ay} for one in a thousand $y \in \mathbb{X}$), the probability that he can compute K_{AC}, K_{AD}, and K_{AB} is substantially lower (10^{-9}). More generally, when t check points are used, the probability of attacker success reduces from p_a to ${p_a}^t$.

Obviously, if the attacker can achieve $p_a = 1$ (or very close to one), MCD is rendered useless. Thus, even while we desire that both ϕ and p_a should be low for an attacker who has access to the key-rings or r nodes, *it is especially important to ensure that achieving p_a close to one is impractical* for attackers.

One of the main weaknesses of PBK/PLM is that it is easier (relatively small r required) for the attacker to achieve $p_a = 1$. RPS can take better advantage of MCD as it better resists realization of high p_a. SKIT offers an even higher resistance to "high p_a" (p_a closer to 1) attacks, and thus takes best advantage of MCD.

6.7 PLM for Sensor Networks

Clearly, SKIT appears to have a substantial edge over all other PKPSs. The achievable security for RPS/HARPS is limited by the complexity of the public function $F()$. For PBK, PLM, and SKIT the achievable n is limited only by available storage. The only difference between a $(n = 1000, p)$-secure (PBK/PLM/SKIT) scheme and a $(n = 100000, p)$-secure scheme is that the latter will need 100-fold higher storage. All other facets of complexity are identical for both schemes, and more importantly, very low.

All other facets of complexity, which are proportional to $\log(1/p)$, can further be lowered if desired, by increasing storage complexity. As unprotected storage (a plentiful resource) is the only bottle-neck, and as SKIT is the most storage efficient, SKIT appears to be the best choice. The appeal of SKIT is further improved by its substantially higher resistance to message injection attacks (better (n, ϕ, p_a)-security compared to PBK/PLM).

However, there are still specific application models where PBK or PLM may be better suited than SKIT. Specifically, as we shall see later in Sect. 7.2.5, PBK has some unique advantages that will become apparent when we consider issues specific to tamper-responsive hardware modules which may be used to protect secrets and perform private computations. Similarly, as we shall soon see, PLM is better suited for a broad class of classical sensor network applications.

6.7.1 Classical Sensor Network Model

Ad hoc networks of spatially distributed battery powered wireless sensors are useful for many application scenarios involving monitoring of environmental conditions like temperature, pressure, humidity, pollution levels, etc. Such networks are typically constituted by many inexpensive wireless sensors with limited battery resources and limited transmission range (to reduce their battery consumption) [63]. Sensors will relay measurements over multiple hops to one (or more) more capable proxy devices. The proxies may have intermittent access to satellite channels for relaying measurements to a remote location.

Typically, a large number sensors and a small number of proxies may be aerially dropped over a region to be monitored. The more capable proxy devices may be equipped with GPS capabilities. The sensors and proxy devices may then engage in protocols to facilitate geographic localization of sensors [64]. After all, without the knowledge of the geographic positions of the sensors the measurements may not be very useful. Sensors may also exchange messages with each other to determine optimal paths for relaying measurements to the closest proxy. In some application scenarios the proxy devices may also be directed (from a remote location) to send specific queries to specific sensors. Sensors may also exchange measurements amongst themselves, possibly for more efficient relaying of sensed measurements.

One important requirement for securing interactions between sensors is the ability to establish pairwise secrets between sensors. For this purpose every sensor device will be initialized with one or more secrets using which the required pairwise secrets can be computed.

6.7.2 Assumptions

Let us assume that sensor devices are mass produced as chips. The total number of manufactured sensors may be billions. Every sensor is assigned a unique identity, and preloaded with a small set of secrets by a key distribution center in the factory floor. Every sensor is equipped with modest hardware for performing symmetric cryptographic computations, and a few tens of kilobytes of storage for keys and sensor-specific software.

Proxy devices may be general purpose devices equipped with GPS and satellite communication capabilities. The proxy devices are assumed to be much more capable, say comparable in capability to a modern mobile phone/PDA. It is assumed that proxy devices can support several GBs of storage using flash storage cards.

An entity desiring to deploy a sensor network may acquire a random set of sensor chips along with a few proxy devices. The sensors and proxy devices may then be aerially dropped over some region to be monitored. Periodically, sensors may be replenished by dropping a fresh batch of sensors over the region, say when many of the earlier batches of sensors have been depleted of their battery power. Likewise, proxies may also be replenished periodically.

Once the sensors (along with proxies) are dropped over a remote region, every sensor will be required to establish pairwise secrets with a few other sensors. Most often, sensors will be required to establish a pairwise secret with all the "accidental neighbors" that happen to fall close to them. Thus, even while a sensor may have to establish shared secrets only with a small set of sensors (e.g, neighbors), no information may be available regarding the potential neighbors of a sensor A at the time the sensor A was provided with secrets (in the factory floor). Thus, any two sensors should have the ability to establish a pairwise secret.

6.7.3 Key Distribution for Sensor Networks

If the total number of proxy devices is small, it is easy to provide every sensor with a secret corresponding to every proxy device. For example if we have L proxy devices identified as $1 \cdots i \leq L$ the KDC could assign a secret K_i^R to the ith proxy device. Every sensor can be given a ticket for every proxy device. For example, a sensor with ID A can be issued L identity tickets $K_{i,A}^R = h(K_i^R, A), 1 \leq i \leq L$. Sensors to use the proxy devices act as mediators to establish a secret session with other sensors.

This approach is not desirable due to three reasons. Firstly, the general purpose proxy devices are trusted only for relaying measurements: only the sensors are trusted for the measurements reported by them. Thus, keys assigned to sensors for purposes of authentication of sensor data (either data sent to remote locations or data exchanged between sensors) should not be privy to the proxies. Secondly, it is undesirable for an attacker who has compromised a proxy device to be able to impersonate a large number sensors. While an attacker who has compromised a proxy device may be able to stop the network (or a part of the network) from being useful, it is not desirable for the attacker to be able to impersonate sensors to send misleading information—"no information" is acceptable; deliberately misleading information is not. Thirdly, even the number of proxies L may be large.

If a scalable KPS like SKIT is used instead, each sensor is issued m secrets and $k = mM$ tickets. We cannot realize high levels of collusion resistance n due to limited storage available to sensors. One possibility is to utilize the storage capabilities of the proxy device to store encrypted secrets on behalf of the sensors. In other words, the k tickets assigned to every sensor can be stored encrypted (using a secret privy only to the sensor to which the tickets were issued) in a proxy device reachable by the sensor. In such a scenario, a proxy device catering for N sensors will need to store kN encrypted tickets. Once deployed, the sensors may establish multi-hop link to the proxy and fetch only the tickets they requires to establish shared secrets with neighboring sensors.

Unfortunately, as it may not be possible to a priori associate every proxy with a specific set of sensors, we desire that any proxy should be able to store and serve encrypted tickets to *every* possible sensor. If the total number of sensors is practically unlimited, then even with generous storage capabilities, the number of keys per sensor (that can be stored by proxy devices) will be small.

For example, assume that every proxy device is equipped with a 64 GB flash card, and a (m, M)-SKIT scheme is used for the deployment. The proxy can store 2^{30} 128-bit tickets. If the proxy is designed to support a N sensors then we can use a SKIT scheme with $mM \leq 2^{30}/N$. For example, if $N = 2^{20}$, then the proxy can store 2^{10} tickets per sensor. In such a scenario a SKIT scheme with $m = 16 = 2^4$ and $l = 2^6 = 64$ can be used.

Perhaps the most efficient way to use a proxy device as an untrusted storage resource is to use the proxy for storing PLM public values. If (m, l) PLM is used, then all proxies store all $m\binom{l}{2}$ public values. Sensors will need to store only m secrets each. Note that the total number of sensors N does not affect the choice of ml. The $m\binom{l}{2}$ public values stored by any proxy device can be utilized by any sensor.

Unlike the scenario in Sect. 6.3.1 where it was desirable to minimize the total storage every node, for our current application of interest, the main constraint on achievable security is imposed by the total storage for public values in the proxies.

For proxy devices capable of storing 2^{30} public values some of the possible choice of m and l are (for example) $(m = 64, l = 5793)$, $(m = 96, l = 4730)$, $(m = 128, l = 4096)$, $(m = 160, l = 3664)$, $(m = 224, l = 3096)$, $(m = 256, l = 2896)$ etc.

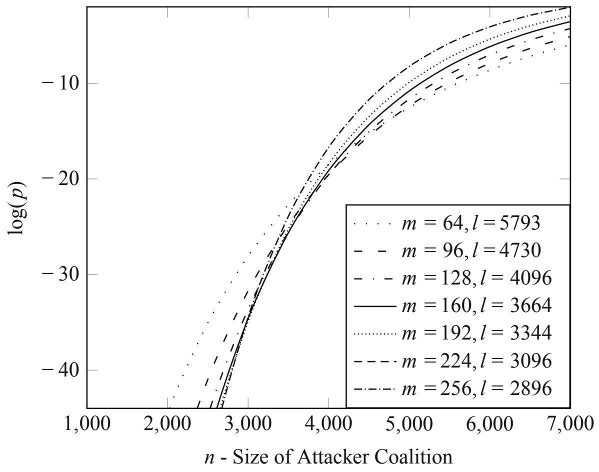
6.7.4 Key Establishment

Similar to sensors, every proxy device is also equipped with a chip with a special identity, and issued m secrets, and $ml/2$ public values.

After deployment, a proxy with identity R broadcasts a message $[R, 0]$ to all neighboring sensors. A neighbor F may then rebroadcast the packet onwards as $[R, F, 1]$. A neighbor G of F two-hops away from the proxy R similarly broadcasts $[R, G, 2]$ and so on. Every sensor will broadcast as many packets as the number of proxies. Simultaneously this facilitates a sensor A to identify the proxy closest to it and the identities of all its neighbors.

Assume that A is within the range of r neighbors $N_1 \cdots N_r$, and the proxy with ID R is the proxy closest to A. The sensor A relays the list of its neighbors to the closest proxy. This list can be authenticated by A using the secret shared between

Fig. 6.5 Performance of PLM for a classical sensor network. All schemes require the same storage for proxies (to store $m\binom{l}{2}$ public values)



sensor A and proxy R, viz.,

$$K_{AR} = h(K_{0,a_0}, r_0) \oplus \cdots \oplus h(K_{m-1,a_{m-1}}, r_{m-1} \tag{6.40}$$

The proxy which has access to all public values computes the same secret by XORing m values of the form

$$S_i = h(K_{i,r_i}, a_i) \oplus P(a_i, r_i), 0 \leq i \leq m-1. \tag{6.41}$$

The proxy relays r values to A—one public value corresponding to each neighbor. The public value corresponding to a neighbor B of A is

$$P_{A,B} = P_0(a_0, b_0) \oplus \cdots \oplus P_{m-1}(a_m, b_m) \tag{6.42}$$

The shared secret K_{AB} is computed by A as follows

$$K_{A,B} = h(K_{0,a_1}, b_1) \oplus \cdots \oplus h(K_{m-1,a_{m-1}}) \oplus P'_{A,B} \tag{6.43}$$

Sensor B can simply compute K_{AB} as

$$K_{AB} = h(K_{0,b_0}, a_0) \oplus \cdots \oplus h(K_{m-1,b_{m-1}}, a_{m-1}). \tag{6.44}$$

6.7.5 Performance and Overhead

The bandwidth overhead for relaying public values to each sensor is proportional to the number of neighbors. Specifically, as only one of the two neighbors will need access to the public value, a sensor with r neighbors will require $r/2$ public values provided by the proxy.

For sensors the computational overhead is m—as m PRF operations are required to compute a pairwise secret. The storage overhead is also m.

Minimizing ml^2 (instead of ml, when every node stores public values) will in general result in a larger choice of m. As m does not affect the bandwidth overhead, and as pairwise secrets will need to be computed infrequently in stationary sensor networks (one a secret for a neighbor has been computed, it can be stored) it may be advantageous to increase m to the extent permitted by the limited storage available to sensors.

Figure 6.5 depicts the performance of PLM ($p(n)$ vs n) for classical sensor networks for various choice of m and M constrained to meet the requirement $2^{30} = ml^2/2$.

6.8 Conclusions

Scalable KPSs like PLM, PBK, and SKIT, derived as extensions of nonscalable KPSs, have some clear advantages over PKPSs like RPS and HARPS which demand substantial public function complexity. While the achievable security for RPS/HARPS is limited by the complexity of the public function $F()$. For PBK, PLM, and SKIT the achievable n is limited only by available storage. The only difference between a $(n = 1000, p)$-secure (PBK/PLM/SKIT) scheme and a $(n = 100000, p)$-secure scheme is that the latter will need 100-fold higher storage. All other facets of complexity are identical for both schemes, and more importantly, very low.

All other facets of complexity, which are proportional to $\log(1/p)$, can further be lowered if desired, by increasing storage complexity. As unprotected storage (a plentiful resource) is the only bottle-neck, and as SKIT is the most storage efficient, SKIT appears to be the best choice. The appeal of SKIT is further improved by its substantially higher resistance to message injection attacks (better (n, ϕ, p_a)-security compared to PBK/PLM).

However, there are still specific application models where PBK or PLM may be better suited than SKIT. PLM is better suited for a broad class of classical sensor network applications. As we shall see in the next chapter, PBK has some unique advantages that will become apparent when we consider issues specific to tamper-responsive hardware modules that may be used to protect PKPS secrets and perform private computations using such secrets.

Chapter 7
Using PKPSs with Tamper-Responsive Modules

Tamper-responsive modules are expected to provide two broad assurances—write-proofing of values stored inside the module and read-proofing of secrets protected by the module. The two requirements are however *not* independent [65]. For instance, with the ability to tamper with the module functionality (software executed by the module), an attacker could direct the module to "spit out" its secrets. On the other hand, secrets that are protected can be used to authenticate software that will be executed by the computer, using (for example) key based hashed message authentication codes (HMACs). Unless the secret used for computing the HMAC is known, the attacker cannot modify the software.

In practice read-proofing is a stepping stone to the more elusive goal of tamper-proofing of software. Attacks aimed at modifying software *to reveal secrets* can be prevented by ensuring that software does not have access to at least some of the secrets that are protected. Some secrets may be *generated, stored, and used* by dedicated hardware [66, 67]. However, authenticating software with the secrets provides a bootstrapping problem [68]. After all, some software should be loaded (typically the BIOS) which includes instructions to load the secret and perform the authentication. Recently Gennaro et al. [65] have argued that providing assurances that software cannot be modified entails assurances of read-proofing *and* the additional assurance of a write protected nonvolatile counter.

Execution of an algorithm inside a tamper-responsive boundary may have some additional constraints compared to the execution of the same algorithm in a general-purpose computing environment. Thus a fair evaluation of different probabilistic key predistribution schemes (PKPSs) will need to take such special considerations into account.

7.1 Core Principles

The problem of practical realization of sufficiently trustworthy tamper-responsive modules has received significant attention since the development of the ABYSS coprocessor [69] in the late 1980s. Even with substantial changes in semiconductor technology and the capabilities of tools that can be utilized by attackers since then,

M. Ramkumar, *Symmetric Cryptographic Protocols,* DOI 10.1007/978-3-319-07584-6_7,

the core principles behind possible attacks and countermeasures have not changed significantly.

Most realizations of tamper-responsive modules like ABYSS [69], Citadel [70], Dyad [71], IBM 4758 [66], and Cerium [72] consist of a tamper-resistant package that includes the CPU, DRAM, battery-backed RAM (BBRAM), and flash ROM. Tamper attempts will result in *zeroizing*, or erasure of secrets stored. In almost all approaches a secure public–private key pair is generated inside the module, and only the public key is exported. The private key (typically a private RSA exponent) is stored in BBRAM and is protected at all times—even when the CPU in the module is off.

Most modules will also generate a private symmetric *master* secret which can be used to encrypt all other secrets that need to be protected. The master secret can be used to encrypt even the private RSA key when the CPU is off. Encrypted secrets could then be stored in nonvolatile memory (NVM) that is not afforded any protection. Only the master secret stored in BBRAM needs to be protected when the device is off. However, when the device is in the on state, other physical areas of the module (like DRAM, special cache memories, etc.) will also be extended protection.

7.1.1 Active and Passive Shields

Such protection measures take the form of active and passive shields, and circuitry that execute countermeasures for zeroizing when active sensors are triggered by intrusions.

Passive shields block inbound and outbound electromagnetic radiations. Outbound radiations (emanating from inside the chip) can be used to reveal some information about the secrets used. Inbound radiations can be used for inducing faults, which can in turn lead to compromise of cryptographic keys [73, 74]. For example, differential power analysis (DPA) [75] can be used for gaining clues about secrets based on instantaneous power consumption by the processor. Countermeasures against DPA include introducing redundant steps in cryptographic computations [76] and the use of self-timed circuits [77].

Active shields attempt to identify intrusions and activate circuitry for zeroizing. Thus active shields are also sensors which can trigger various countermeasures. For instance, sophisticated attacks involving focused ion beam (FIB) techniques [78] can permit an attacker to drill fine holes and establish connections with the computer buses. With such taps the attacker can gain access to the bits that pass through the buses when the CPU is functioning. The active shields used as countermeasures typically take the form of a mesh (or many layers of meshes) of nonintersecting conductors [66, 71]. They can prevent microprobes and picoprobes [79] from gaining line-of-sight access to the buses. Even if one line of the mesh is cut, the resulting open circuit will trigger the circuitry for zeroization. In addition, even if we can ensure that only a fraction of the lines can be tapped, it may be possible to use *private circuits* [80] to ensure that the attacker gains no knowledge (by tapping a few lines).

A side effect of shielding is that it may make it difficult to dissipate heat generated inside the shielded boundary. An obvious strategy to minimize this issue is simply to reduce to the extent possible the computational overhead inside the protected boundary. For example, limiting operations inside the boundary to symmetric pseudo random functions (PRF) operations (or avoiding more expensive asymmetric operations) can be a useful strategy.

7.1.2 State Transitions

Any tamper-responsive module can be at one of two broad states—*in use* (or “on”) or *at rest* (or “off”). Secrets have to be protected during both states.

When the module is off there is no need to extend protection to all regions—only the privacy of a single master secret and the integrity of a single monotonic counter needs to be assured. However, when the module is on, the scope of protection will need to be wider. Various registers, scratch pad memory, and buses inside the module may also need to be extended protection.

Physical unclonable functions (PUF) [81] provide a satisfactory solution to protect secrets of a device while the device is *at rest*. Silicon PUFs [82] exploit uncontrollable statistical delay variations of connections and transistors etched on substrates in each manufactured chip to provide an “uncharacterizable” and therefore *unclonable* unique physical one-way function (POWF) [83] which can serve as a “random oracle.” The response of the random oracle (PUF) to a randomly chosen challenge could be used to encrypt the master secret at rest. The challenge itself could be stored in the clear in NVM. When the device is at rest, there is no way for the attacker to challenge the PUF to determine the response used for encrypting the secrets. When the device is powered on for this purpose, the tamper detection sensors kick in. Thus, PUFs eliminate the need for active shielding of modules at rest.

7.1.2.1 Remnance

One of the hidden problems associated with the dynamic scope of countermeasures takes the form of *remnance* in volatile memory [84, 85]. Most protected volatile memory regions during the on state are not protected when the module is in the off state. Unfortunately, bits stored in volatile memory (especially for extended periods) can leave decipherable “footprints.” The footprints can be scavenged *even after the power supply is removed*. The ability of the attacker to scavenge bits from footprints can be improved by cooling the chip, for example, by immersing the chip it in liquid nitrogen. Thus, even after volatile memory regions like cache/RAM have been powered off (in the off state, where only the BBRAM is extended protection), secrets that *were* stored in RAM/cache regions (while the device was on) may be revealed.

Safe deletion [84] of contents in magnetic and solid state memories require many *repeated* overwriting operations. Some of the usual countermeasures against attacks that exploit this weakness include:

1. Clean erasure (by repeated overwriting) of contents of volatile memory (like RAM/cache memory, except contents of BBRAM) before powering off, and as part of zeroization
2. Ensuring that secrets are not stored for long durations in RAM
3. Use of special sensors that respond to sudden changes in temperature and trigger clean erasure [71] of volatile memory regions
4. Increasing the mass of the modules to inhibit *rapid* cooling [71] to provide an adequate response time to execute countermeasures
5. Periodic ones-complementing of some highly sensitive secrets [66, 84], (for example, the master secret stored in the BBRAM) with dedicated circuitry for this purpose, to ensure that no footprints are left behind.

Increasing the mass of modules may simply not be a viable option for various applications scenarios of interest to us. Furthermore, while periodic ones-complementing may be possible for a limited number of secrets (for example, contents of the BBRAM), extending such protection to *all* volatile memory regions like cache memory and RAM, where the values stored may be *actively* used in computations, is far from practical.

Even storing secrets for a fleeting duration in RAM can be risky, as simple attacks that induce faults in memory [85] that could cause the CPU to hang. As a result, a key that was intended to be stored only for a fleeting moment at the time the fault was induced may end up being stored for a long time. The process of inducing a fault could be as easy as shining a powerful beam of light [78], if proper shielding is not provided.

Even with good shielding to ensure that the risks of such attacks are minimal, there may be numerous other reasons, including hardware/software bugs, which may result in the CPU hanging. If the CPU hangs *while* a sensitive secret is stored in the RAM, an attacker can wait for some duration to ensure that the secret leaves a deep footprint before plunging the module in liquid nitrogen.

While sensors that can detect rapid changes in temperature (which obviously should work independent of the CPU and are well protected by active shields) can erase contents of the RAM, the repeated overwriting operations mandated for *clean erasure* of all sensitive information in the RAM may not be possible. Even repeated overwriting may not be a satisfactory solution for DRAMs [84] for which the only option may be to ensure that sensitive values are not stored for extended durations[1] (even a few tens of seconds).

[1] For clean erasure of contents stored for long durations in DRAM the only option (apart from heating) may be to store some random value for a long duration to “dilute the stress” [84] imposed on the oxide layer by the old data.

7.1.3 *Single-Step Countermeasures*

Countermeasures that involve more than a single step to be effective are *inherently vulnerable*. With complete knowledge of the layout of the components (which attackers can easily determine by tampering with a few chips/modules [79]), attackers can "force their way in" using FIBs to cut off circuitry (or power supply) responsible for undertaking the countermeasures. Even with good shielding and assurances that it is not possible for intrusive attempts to evade active shields, with such attacks the attacker does *not* have to worry about triggering active shields (which leads to erasure of the master secret). As long as the circuitry for taking *additional* countermeasures (like clean erasure of contents of RAM) can be cut off, the attacker can still scavenge bits from RAM.

The end result of a successful attack is that the attacker gets a *snapshot* of all contents of all volatile memory regions (except the BBRAM) at the instant the attack was launched. However, the attacker is limited to a *single* snapshot as the module is irrevocably destroyed in this process, and the master secret used for encrypting all other secrets is erased. Nevertheless, the attacker gains knowledge of secrets/data that may have been stored unencrypted in RAM, even temporarily, at the instant the attack was carried out.

7.2 The DOWN Policy

If contents of the volatile RAM cannot be well protected following abnormal state transitions that can be induced by the attacker, a solution is to make sure that the RAM has very minimal information at *any point in time*. A simple security policy—decrypt only when necessary (DOWN) [60], recognizes the fact that most cryptographic operations have some inherent *atomicity*. At any point in time only one, or maybe even a small part of a secret may be necessary for cryptographic computations.

For instance, if the secret to be protected is an RSA private exponent r (say, of size 1024 b), and n is the RSA modulus, decryption of a cipher text C involves modular exponentiation of C with r as $P = C^r \bmod n$. However, to perform the exponentiation, *only one bit of r is needed at any point in time* (for example, exponentiation using the square and multiply algorithm). We could thus keep r encrypted at all times, and decrypt each bit *as and when necessary*.

When the DOWN policy is used, the single master secret χ in the module is used for encrypting all secrets—which may then be stored in unprotected NVM. A protected PRF block inside the module can generate any number secrets of the form $S_i = h(\chi \parallel i)$, where S_i is used for encrypting the ith secret K_i stored in NVM. At any point in time, at most one S_i can be stored in RAM—or no footprint will reveal more than one S_i.

The master secret can indeed be afforded a high level of protection as:

1. This secret is not shared with any other entity
2. The secret is never used by any software (never transferred to RAM), and
3. Does not leave footprints

Without the use of the DOWN policy an attacker may be able to scavenge the entire RSA private key from footprints in RAM. With the DOWN policy in effect, the attacker can discover no more than 1 b of the RSA secret. As in this process the master secret is erased, the attack can not be repeated to determine other bits.

With the DOWN policy in place, operations like computing a signature using a private key, or decrypting a value using a private key, or using multiple keys to compute a shared secret are broken down into several DOWN cycles, wherein each cycle secret (or a part of the secret) is decrypted using a secret S_i generated from the master secret, and is used as an input to some algorithm which performs application specific operations using the decrypted secret. During each DOWN cycle the decrypted secret overwrites the secret employed in the previous DOWN cycle.

7.2.1 DOWN-Enabled Modules

For DOWN-enabled modules the master secret χ generated spontaneously inside the module is the only secret that is *directly* protected. The master secret is stored in special volatile register (a BBRAM). The CPU in the module has has exclusive access to a hardware PRF. The processor exposes dedicated CPU instructions for using the hardware PRF in conjunction with the master secret χ for generating secrets like S_i. As special protection may be in place when the master secret is used, we shall represent by **D** the special operations that involve the master secret, or

$$S_i = \mathbf{D}(i) \tag{7.1}$$

The secrets like S_i are used to encrypt all other secrets *indirectly* protected by the module.

The module draws power from external devices during its operation. While the CPU is off battery backup power the BBRAM and minimal active circuitry required for protecting the master secret. The active circuits include mechanisms for periodically ones complementing the master key so that even if the battery backups fail, no decipherable footprints of the master secret are left behind. The battery or power lines from the battery do *not* need to be protected. For any countermeasure the *only* step is erasure of the master secret by removing power supply to the BBRAM.

7.2.1.1 DOWN with RSA

In RSA two large secret primes p and q are chosen and the RSA domain $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ is computed as $n = pq$. A value $e, 3 \leq e \leq \Phi(n) = (p-1)(q-1)$ is chosen as the *public exponent* subject to the constraint that e and $\Phi(n)$ are relatively prime. The inverse d of e in the modular domain of $\Phi(n)$, or $d \equiv e^{-1} \bmod \Phi(n)$ is the *private* exponent. The values p, q, and $\Phi(n)$ are then destroyed.

For encryption the cipher-text $C \in \mathbb{Z}_n$ corresponding a plain-text $P \in \mathbb{Z}_n$ is computed as $C \equiv P^e \bmod n$. The decryption of C is performed as $P \equiv C^d \bmod n$. Similarly signing a hash H of a message is performed as $\Sigma \equiv H^d \bmod n$, and verification of the signature Σ is achieved by computing $H = \Sigma^e \bmod n$.

The goal of the DOWN policy is to protect the *private* exponent d from being scavenged from memory. Computations performed for encryption and verification of signatures (which do *not* use the private key) are unaffected by the DOWN policy. The private exponent d is used for decryption and signing. The private exponent needs to be stored in RAM for performing computations like $P \equiv C^d \bmod n$ (decryption) and $\Sigma \equiv H^d \bmod n$ (signing).

Modular exponentiation is often performed using the square and multiply ([1], Chap. 5) algorithm. Let the binary representation of d be $\delta_1\delta_2\ldots\delta_b$ (or $\delta_i, i = 1 \leq i \leq b$ are the b bits of d, where δ_1 represents the most significant bit (MSB) and δ_b the least significant bit (LSB)). The evaluation of $P = C^d \bmod n$ with the square-and-multiply algorithm proceeds in b steps

$$z_i = \begin{cases} z_{i-1}^2 \bmod n & \text{if } \delta_i = 0 \\ z_{i-1}^2\ C \bmod n & \text{if } \delta_i = 1 \end{cases} \tag{7.2}$$

with z_0 initialized to 1 and $z_b = P$. Note that in each step (loop) *only one bit of the private key d* is required.

Thus the private exponent d can be stored as b independent encryptions of each bit. For each step in the evaluation of the square-and-multiply algorithm one encrypted bit is fetched, decrypted, and used in modular computations. Thus no snapshot will reveal more than one bit of the private key. Recall that *without* the DOWN policy the *entire* private key could be exposed by a snapshot.

7.2.2 DOWN with Other Asymmetric Schemes

The effectiveness of the DOWN policy is intricately tied to the nature of cryptographic computations that have to be performed *using the private key*. Such computations may involve different types of finite field (or ring or group) operations like exponentiation, multiplication, and computation of multiplicative inverses. As seen earlier, modular exponentiation (where the exponent is a secret to be protected) is naturally facilitated. Modular multiplication of two quantities (one of which is a secret to be protected) can also be facilitated in the same way. Just as exponentiation involves "squaring" or "squaring and multiplication" in every loop (depending on whether the particular bit of the private key is a 0 or a 1), for multiplication each loop involves "doubling" (left shift) or "doubling and addition," (left shift followed by addition) depending on the particular bit of the private key.

7.2.2.1 DOWN with Exponential Ciphers

For example, in the El Gamal cryptosystem ([1], Sect. 6) over $\mathbb{Z}_p$, using a primitive element $g \in \mathbb{Z}_p$, private key $a \in \mathbb{Z}_p$, and public key $\alpha = g^a \bmod p$. Encryption of a message x using the public key and decryption using the private key are carried out as follows:

$$\begin{aligned} e_K(x,k) =& (y_1, y_2), \begin{cases} y_1 = g^k \bmod p \\ y_2 = x\alpha^k \bmod p \end{cases} \\ d_K(y_1, y_2) =& x = y_2(\{y_1^a\})^{-1} \bmod p. \end{aligned} \tag{7.3}$$

where $k \in \mathbb{Z}_{p-1}$ is randomly chosen by the entity encrypting the message. The scope of the DOWN policy in this case is to ensure that no more than a small fraction of the private key a is decrypted and stored in RAM at any point in time during the computation of y_1^a. As in the case of RSA, exponentiation with a requires only one bit of a in each loop of the square and multiply algorithm.

In the El Gamal signature scheme ([1], Sect. 7.3) $\{p, g, a, \alpha\}$ over $\mathbb{Z}_p$, where $g \in \mathbb{Z}_p$ is a primitive element, and $a \in \mathbb{Z}_p$ is the private key of the signer, and $\alpha = g^a \bmod p$ is the corresponding public key, the signature $\mathbf{sig}(x, k)$ for a value $x \in \mathbb{Z}_p$, and a random $k \in \mathbb{Z}_{p-1}$ is

$$\mathbf{sig}(x,k) = (\gamma, \delta), \begin{cases} \gamma = g^k \bmod p \\ \delta = (x - \{a\gamma\})k^{-1} \bmod p - 1 \end{cases} \tag{7.4}$$

The operation performed with the secret a (during signing of a message) is evaluation of $a\gamma \bmod p$. Multiplication with a can also be trivially performed using only one bit of a in each loop.

The popular Diffie–Helman (DH) key exchange algorithm also lends itself readily to DOWN. In the DH key exchange algorithm two nodes A and B agree on some prime p and $g\mathbb{Z}_p$, choose secrets a and b respectively and make public $\alpha = g^a \bmod p$ and $\beta = g^b \bmod p$ respectively, to establish a shared secret $K_{AB} = \alpha^b = \beta^a$.

7.2.2.2 Elliptic Curves

DOWN also readily extends itself to protecting the private keys of elliptic curve (ECC, [1], Sect. 6.5) based systems. Elliptic curves form an additive group $\mathcal{G} \in \mathbb{Z}_p \times \mathbb{Z}_p$ defined over a finite field $\mathbb{Z}_p$. For example

$$\mathcal{G} = \{(x, y)\} : y^2 \equiv x^3 + a_0 x + a_1 \bmod p \tag{7.5}$$

is an elliptic curve.

Consider points $P = (x_P, y_P), Q = (x_Q, y_Q) \in \mathcal{G}$ that lie on the elliptic curve, where and $x_p, y_P, x_Q, y_Q \in \mathbb{Z}_p$. The operation $R = P + Q$ (addition of two points on the curve) results in another point R which also lies in the same curve.

$$\begin{aligned} \mathcal{G} =& \{(x, y)\} : y^2 \equiv x^3 + a_0 x + a_1 \bmod p \\ R =& P + Q = (x_R, y_R), \begin{cases} x_R = \lambda^2 - x_P - x_Q \\ y_R = \lambda(x_P - X_R) - y_P \\ \text{where } \lambda = \begin{cases} (y_Q - y_P)(x_P - x_Q)^{-1} & P \neq Q. \\ (3x_P^2 + a_0)(2y_P)^{-1} & P = Q \end{cases} \end{cases} \end{aligned} \tag{7.6}$$

For ECC schemes the private key is a randomly chosen value $a \in \mathbb{Z}_p$. The security of ECC schemes rely on the assumption that if $P' = aP$, where $P, P' \in \mathcal{G}$, even with the knowledge of P *and* P', it is infeasible to evaluate a. The operation performed with the secret (private key) a in all ECC schemes involves computation of a value aP. Multiplication of a point $P \in \mathcal{G}$ by a value $a \in \mathbb{Z}_p$ is carried out as $\log_2(a)$ doubling /"doubling and addition" group operations where only one bit of a need to be used at any time. Thus, ECC schemes also "DOWN compatible."

7.2.2.3 Generation of Private Keys

A *strict* implementation of the DOWN policy mandates that the DOWN policy should be observed *throughout the life cycle* of the module. While for most public key schemes observing the DOWN policy for *using* the secrets is trivial, for RSA it may be very difficult to observe the DOWN policy during *generation* of primes p and q. Furthermore, many of the optimizations employed for speeding up exponentiation [86] in RSA can render observing the DOWN policy even for using[2] the private key difficult.

However, generation of secrets is not an issue for El Gamal (and variants) and ECC schemes, where the private key can be *randomly* chosen from $\mathbb{Z}_p$. Thus for such schemes it is possible to generate each *bit* of the private key a independently, encrypt, store them.

7.2.3 DOWN With ID-Based Schemes

Recall that for ID-based schemes (Sect. 5.2) the ID of the node itself doubles as the public key, thus obviating the very need for certificates. In ID-based schemes a key distribution center (KDC) chooses public parameters of the system and one or more master secrets. Using the secrets the KDC can compute the private key(s) corresponding to *any* public key (ID). The private keys for a node with identity A are thus *assigned* by the KDC to the node A.

We saw several advantages of ID-based schemes (over certificate-based schemes) in Sect. 5.2. Another desirable feature of ID-based schemes, especially for their use in conjunction with *tamper-responsive* devices, comes from the fact that the keys are implicitly escrowed (by the KDC). In tamper-responsive devices, false alarms leading to unintended zeroization can never be ruled out. Without key escrow, functional devices will be rendered useless under such circumstances. This could even lead to scenarios where an unfortunate end user may be locked out of all data encrypted using a secret protected by the module. With escrowed ID-based schemes such devices can be easily reinstated into the network.

[2] Most such optimizations involve exponentiating with the private key in $\mathbb{Z}_p$ and $\mathbb{Z}_q$ where $n = pq$ is the RSA modulus. Thus *both* the exponent and the modulus (p and q) have to be protected, which may not be feasible.

While identity-based schemes based on symmetric primitives require multiple keys to be stored by each participant, identity-based schemes based on asymmetric primitives require only one key to be stored by each participant.

7.2.3.1 Shamir's IBS Scheme

The first identity-based *signature* (IBS) scheme, was proposed by Shamir [43]. The KDC chooses:

1. Two large primes p and q, where $n = pq$
2. $e \in \mathbb{Z}_n$ relatively prime to $\Phi(n) = (p-1)(q-1)$ (and e is preferably a large prime)
3. A one-way function $f()$.

The KDC makes n, e, and $f()$ public. A node with ID ID_i is issued a secret g_i where $g_i^e \equiv ID_i \text{mod } n$, or g_i is a eth root of ID_i in $\mathbb{Z}_n$ (which can be easily computed by the KDC as the KDC knows the factors of n). To sign a message M the signer:

1. Chooses a random $r \in \mathbb{Z}_n$
2. Computes $t \equiv r^e \bmod n$
3. Computes $\alpha = r^{f(t,M)}$. The signature for a message M is (s,t), where

$$s \equiv \{g_i \alpha\} \bmod n. \tag{7.7}$$

The verification condition is $s^e \equiv ID_i t^{f(t,M)} \bmod n$.

Note that the operations with the secret g_i by the signer only involve *multiplication* ($\{g_i\alpha\}$), which poses no problems with DOWN implementation.

7.2.3.2 Pairing-Based Schemes

Shamir's identity-based scheme does not support encryption. Boneh and Franklin [41] responded to Shamir's challenge to develop the first ID-based scheme which could support both encryption and signatures. Such ID-based encryption (IBE) and signature (IBS) schemes [41, 42] rely on a bilinear mapping $e : \mathcal{G}_1 \times \mathcal{G}_1 \rightarrow \mathcal{G}_2$, where $\mathcal{G}_1$ is an additive group and $\mathcal{G}_2$ is a multiplicative group. Typically $\mathcal{G}_1$ is a special elliptic curve and the mapping e represents a class of Weil pairings [41]. For pairing-based IBE/IBS schemes the private key assigned to each node is a point in the elliptic curve $\mathcal{G}_1$.

The pairing operation requires group additions involving a secret in $\mathcal{G}_1$, which in turn calls for computation of *multiplicative inverses* using the secret. Computation of multiplicative inverses, say $b = a^{-1} \text{mod } m$, where only one part of the secret a can be revealed at any time, does not appear to be trivial.

Note that while ECC schemes call for group addition (which requires computation of multiplicative inverses), the operation is performed on points on the elliptic curve which *do not reveal any information about the private key.* For ECC schemes points on the elliptic curve may be publicly known generators or intermediate values of computations. For pairing-based IBE schemes on the other hand, the *private key is itself a point on the elliptic curve*. Thus the more versatile pairing based IBE/IBS schemes do not seem to support DOWN implementations.

7.2.4 DOWN Assurance and Complexity

The *DOWN assurance* provides a guarantee that an attacker can expose no more than one elementary fraction of the secret (private key) by tampering with a tamper-responsive module, *as long as the master secret cannot be compromised*. In other words, the DOWN assurance relies *only* on the *first*-step countermeasure (erasing the master secret by removing power supply to the BBRAM). By simply *tolerating* the fact that the attacker cannot expose more than 1 b (or fraction) of the private key, the DOWN policy eliminates the need for the expensive and inherently vulnerable multi-step measures.

Without the DOWN policy we saw that several *additional* countermeasures are mandated to address the problem of remnance. Specifically, such countermeasures mandated:

1. Sensors for detecting rapid changes in temperature
2. Exclusive circuitry for erasing footprints (when active shields or temperature sensors are triggered)
3. Increasing the mass of modules

Furthermore such expensive countermeasures are *still* vulnerable as an attacker with complete knowledge of the layout of a module can still expose an entire private key from RAM. With the DOWN policy the attacker is restricted to exposing at most 1 b (or a fraction) of the private key. The complexity imposed by DOWN depends on the number of elementary DOWN operations into which the process of decryption/signing is split. For a 1024-b private key d the DOWN complexity is 1024 DOWN operations. However, for protecting the 1024-b private exponent we do *not* need to employ 1024 DOWN operations. If we employ only two (where in each DOWN operation 512 b of the private key are decrypted) no more than 512 b of the private key can be revealed by a snapshot. In practice, if the PRF used by the module produces (say) 128-b outputs, it may be more efficient to store the private key as multiple (encrypted) 128-b chunks. Each DOWN operation may require mode switching if the dedicated CPU instruction for encrypting/decrypting secrets is permitted only in a special secure kernel mode [67] (or a *concealed execution mode* [87]).

The DOWN policy readily lends itself to asymmetric schemes as long as the operations that employ the private key is restricted to modular exponentiation (RSA, DH key exchange, El Gamal encryption schemes) or multiplication (El Gamal signature scheme and variants, ECC). However the DOWN policy is *better suited for El Gamal and ECC schemes* where there are *no restrictions on the choice of the private key* (unlike RSA where it is required to verify that the private keys p and q are indeed primes).

7.2.5 DOWN with PKPSs

When DOWN is used with PKPSs it can provide the assurance that not more than one of the multiple secrets can be revealed. The effect of this assurance is, however, different for different PKPSs.

In general, for all PKPSs $\mathbb{O}(\log(1/p))$ DOWN cycles will be necessary. Random preloaded subsets (RPS), hashed random preloaded subsets (HARPS), and parallel basic key predistribution scheme (PBK) can take good advantage of the DOWN assurance. As only one of the $k = \mathbb{O}(n \log(1/p)$ secrets can be exposed by successfully tampering with any device, to obtain the equivalent of "secrets from n nodes," an attacker has to successfully tamper with nk devices. In other words, an (n, p)-secure PKPS is rendered (nk, p)-secure with the DOWN assurance.

Unfortunately, parallel Leighton–Micali scheme (PLM) and subset key and identiity tickets (SKIT) cannot take very good advantage of the DOWN assurance. In PLM/SKIT, if an attacker can able to expose one of the m secrets from a snapshot, the DOWN assurance merely improves the collusion resistance by a factor $m = \mathbb{O}(\log(1/p))$.

Without the DOWN assurance all schemes below:

1. PBK/PLM with parameters $m = 64$ and $l = 2^{12}$
2. SKIT with $m = 32$ and $l = 2^{11}$
3. RPS with $k_r = 171,233$
4. HARPS with $k = 107,088$

satisfy $p(n) \leq 2^{-64} \forall n \leq 1420$. With the DOWN assurance, however,

1. PBK is rendered $(372,244,480, 2^{-64})$-secure $(nlm = 372,244,480)$
2. RPS is rendered $(243,150,860, 2^{-64})$-secure $(nk_r = 243,150,860)$
3. HARPS is rendered $(152,064,960, 2^{-64})$-secure $(nk_r = 152,064,960)$
4. PLM is rendered $(90,880, 2^{-64})$-secure $(nm = 90,880)$, and
5. SKIT is rendered $(45,440, 2^{-64})$-secure $(nm = 45,440)$

7.3 A Second Look at Key Predistribution Scheme (KPS) Complexity

That KPSs will be used in conjunction with a tamper-responsive module influences not only the security of KPSs but also the overhead. In order to get a better understanding of various facets of complexity associated with practical deployment of KPSs, consider the following generic device model suitable for a broad range of applications.

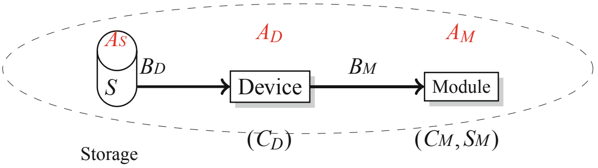


Fig. 7.1 Generic device model. A node A includes the device A_D, module A_M, and an external resource A_S. The external resource may be a passive storage resource, or an active resource which may also perform some computations on behalf of A_D

7.3.1 Generic Device Model

Fig. 7.1 depicts a broad device model where a "node" A is seen as consisting of three components:

1. A device A_D,
2. A (possibly external) resource A_S employed by device A_D
3. A tamper-resistant module associated with device A_D

While possible, it is not necessary that the three distinct components are physically linked together. The only assumption is that there exists a channel—a possibly open channel $A_D \leftrightarrow A_S$ and $A_D \leftrightarrow A_M$. At one extreme, all three components may be housed in a single physical unit. At the other extreme, the links $A_S \leftrightarrow A_D$ and $A_D \leftrightarrow A_M$ may be over long-range networks.

Some practical examples of nodes under this model are as follows:

1. A_D may be a wireless sensor device. A_M is a chip housed in the device; A_S refers to a proxy device that may even be shared by several nodes.
2. The device A_D is a mobile phone; A_M is a chip/SIM card in the mobile phone; A_S is a flash memory card in the phone.
3. A_D is a general purpose server; A_S is a remote database, and A_M is a chip possibly housed in a remote secure location.

Under this device model, the process employed by A for computing a secret K_{AB} shared with node B can be seen as consisting of the following steps:

Step 1: Computation of Public Function This step is necessary to determine the indexes of a small fraction of KPS secrets necessary for computing K_{AB}. While this operation will most likely be the responsibility of the device A_D, it may also be performed by an active external resource A_S. In Fig. 7.1 the complexity of this step is represented as $\mathcal{C}_D$.

Step 2: Retrieving Secrets from Storage In general, PKPSs will require the resource A_S to store $k = \mathbb{O}(n \log(1/p))$ secrets/tickets/public values. Out of the k secrets, only a small fraction $m = \mathbb{O}(\log(1/p))$ values will need to be retrieved, for computing a specific K_{AB}. Let us represent the storage overhead as $\mathcal{S}_S$. Let $\mathcal{B}_D = m$ be the bandwidth overhead incurred for retrieving the m values.

Step 3: Providing Inputs to Module A_M The m values retrieved from storage may be combined before they are presented as inputs to the module. For most PKPSs the m values may be XORed together into a single value. We shall represent the number of values that need to be provided to the module A_M as $\mathcal{B}_M$. For most PKPSs $\mathcal{B}_M = 1$. For some PKPSs, however, the m values will have to be provided as inputs to the module A_M (or $\mathcal{B}_D = \mathcal{B}_M = m$).

In scenarios where the external resource is active, the overhead $\mathcal{C}_D$ may be borne by the external resource. The external resource retrieves m values, XORs them together and may provide only a single value to device A_D, which is passed on by device A_D to module A_M. In this scenario $\mathcal{B}_D = \mathcal{B}_M = m$. If the external resource is a passive storage device the overhead $\mathcal{C}_D$ is borne by the device A_D. The device may XOR m values together and submit a single value to the module A_M. In this case $\mathcal{B}_D = m$ and $\mathcal{B}_M = 1$.

Step 4: Operations with Secrets The module A_M performs some operations using secrets stored inside the module, and values presented as inputs to the module. The secrets that need to be stored inside the module and number of values that need to be presented to the module will both influence the storage $\mathcal{S}_M$ required inside the module.

For most PKPSs $\mathbb{O}(m)$ PRF operations will need to be performed inside the tamper-responsive boundary of the module. Let the number of PRF operations that need to be performed inside the module be $\mathcal{C}_M$. In addition to PRF operations (if the DOWN policy is used) then the number of DOWN cycles will also need to be considered as a source of complexity (as each cycle may require mode switching). Let $\mathcal{D}_M$ represent the DOWN complexity for module A_M.

Ideally, while we would like to minimize *all* facets of complexity, it is especially important to minimize the complexity of the module A_M, viz., $\mathcal{S}_M$ (memory requirement inside the module), $\mathcal{C}_M$ (computations performed by the module), and $\mathcal{D}_M$ (DOWN complexity).

In addition, it is also necessary to consider the overhead mandated for KDCs. For any KPS, the KDC can choose a single master secret χ and derive all other values from χ. Thus, storage for secrets is not an issue. We shall also assume that every node already possesses a secret it shares with the KDC, which can also be readily computed by the KDC using the master secret χ. For example, let us assume that the secret shared between module A and the KDC is K_A. The secret K_A will then be used for encrypting all other secrets conveyed to A.

It is not sufficient to enforce the DOWN policy merely for computation of pairwise secrets. The DOWN policy should also be enforced during the process of receiving secrets from the KDC. Note that if the secret K_A is revealed from a snapshot, all secrets assigned to A are compromised. A simple way around this is for the KDC to issue two secrets to every node. Let the two secrets shared with A be K_A and K'_A. For example, the KDC may compute the two secrets as

$$\begin{aligned} K_A &= h(\chi \parallel A \parallel 0) \text{ and} \\ K'_A &= h(\chi \parallel A \parallel 1) \end{aligned} \tag{7.8}$$

A secret S_{ij} (associated with an index (i, j)) may then be conveyed to module A as

$$S_{ij}^{A} = h(K_A \parallel i \parallel j) \oplus h(K'_A \parallel i \parallel j) \oplus S_{ij} \tag{7.9}$$

Module A decrypts S_{ij}^{A} in two DOWN operations—one in which only K_A is revealed, and the other in which only K'_A is revealed. This ensures that only one of the two secrets K_A or $K_{A'}$ can be salvaged from a snapshot. After obtaining S_{ij}, it is re-encrypted using module A's master secret (say, χ_A) as

$$S'_{ij} = h(\chi_A \parallel i \parallel j) \oplus S_{ij} \tag{7.10}$$

Overall, two DOWN operations will be required for the KDC to convey a secret to a module; three DOWN operations will be required for a module to accept each secret from the KDC. However, this is a one-time operation.

7.4 Comparison of KPSs

We are now ready to evaluate the seven facets of complexity $(\mathcal{C}_T, \mathcal{S}_S, \mathcal{B}_D, \mathcal{C}_D, \mathcal{B}_M, \mathcal{C}_M, \mathcal{S}_M$ for various KPSs we have discussed thus far. For a fair comparison, we shall evaluate the complexity of all five KPS for the same (n, p)-security: $n = 1420$ and $p = 2^{-64}$. The parameters necessary for the five KPSs to achieve this requirement are as follows:

1. PBK/PLM: $m = 64$ and $l = 2^{12}$;
2. SKIT: $m = 32, l = 2^{11}$;
3. RPS: $P = 243{,}322{,}093, k = 171{,}233$;
4. HARPS: $L = 64, P = 85{,}872{,}352, K = 107{,}088$.

7.4.1 Deployment Complexity

For all KPSs the KDC starts with a single secret μ.

7.4.1.1 SKIT

The m SKIT secrets $\chi_0 \ldots \chi_{m-1}$ can be computed as

$$\chi_i = h(\chi \parallel i). \tag{7.11}$$

Corresponding to each μ_i the KDC can readily compute l secrets as

$$K_j^i = h(\chi_i \parallel j), 0 \leq j \leq M - 1. \tag{7.12}$$

Let the two shared secrets between the KDC and node A be $K_A = h(\chi \parallel A \parallel 0)$ and $K'_A = h(\chi \parallel A \parallel 1)$. As $l = 2^{11}$, the m short IDs assigned to A are 11-b values. Thus, the public function $f(A) = [a_0 \dots a_{m-1}]$ outputs a sequence of $m = 32$ 11-b values. The sequence of $32 \times 11 = 352$ b can be readily obtained by hashing A repeatedly. If a 160-b hash function $h()$ is used, then three evaluations of $h()$ can be used for realizing $f(A) = [a_0 \dots a_{m-1}]$.

For providing m secrets to A the KDC has to compute

$$K_i^{a_i} = h(h(\chi \parallel i) \parallel a_i), 0 \le i \le m-1 \tag{7.13}$$

and for encrypting each secret the KDC has to compute $K_A = h(\mu \parallel A \parallel 0)$ and $K'_A = h(\mu \parallel A \parallel 1)$, and

$$K_i^{a_i\,\prime} = K_i^{a_i} \oplus K_A \oplus K'_A \tag{7.14}$$

This calls for four hash operations and three DOWN cycles for generating and encrypting each of the $m = 32$ secrets to be supplied to A.

Similarly, the ml tickets to be conveyed to A are computed as

$$T_i^j(A) = h(h(h(\chi \parallel i) \parallel j) \parallel A), 0 \le i \le m-1, 0 \le j \le M-1. \tag{7.15}$$

As tickets assigned to A do not reveal secrets assigned to other devices, it may be an overkill to extend DOWN protection to every ticket. The KDC may thus convey a single secret K_A^T to A which is used for encrypting every ticket. For example, the secret K_A^T may be generated as $h(\chi \parallel A \parallel A)$. The ml tickets are encrypted as

$$T_i^j(A)' = T_i^j(A) \oplus h(K_A^T \parallel i \parallel j), 0 \le i \le m-1, 0 \le j \le M-1. \tag{7.16}$$

Thus, for generating any encrypted ticket two DOWN cycles and four hash operations will be required. As $ml >> m$, the computational overhead for the KDC can be seen as $2\,ml$ DOWN operations and $4\,ml$ hash operations. As $ml = 2^{16}$ the KDC overhead is 2^{17} DOWN operations and 2^{18} hash operations.

For accepting the $m+1$ secrets (decrypt the secrets using K_A and K'_A and reencrypting them using the master secret μ_A) module A will require three hashes in three DOWN cycles. For accepting the ml tickets and reencrypting them two DOWN cycles and two hashes will be required per ticket.

7.4.1.2 PLM

For PLM the leading factor in KDC complexity is the cost for computing $ml/2$ public values per node. Computing each $K_i^{a_i}$ will require two hash operations and one DOWN operation. As two such values and two additional hashes will be needed for computing a pairwise public value ($K_i^{a_i}$ and K_i^j required for computing $P^i_{a_i,j}$), each public values requires four hash operations and two DOWN operations. The total computational overhead for the KDC is thus $2\ ml/2$ DOWN operations and

4 $ml/2$ hash operations. As $ml = 2^{18}$ the total overhead is 2^{18} DOWN operations and 2^{19} hash operations.

For accepting m secrets (decrypt the secrets using K_A and K'_A and reencrypting them using the master secret μ_A) module A will require three hashes in three DOWN cycles. No operations with public values are required.

7.4.1.3 PBK

In PBK the KDC has to compute ml secrets to be assigned to every node. Computing any $K_i^{a_i}$ will require one DOWN operation and two hash operations. Computing each of the l secrets derived from $K_i^{a_i}$ will require another hash operation. In addition, two DOWN operations and two hash operations will be required for generating the secrets $K_A = h(\chi \parallel A \parallel 0)$ and $K'_A = h(\chi \parallel A \parallel 1)$ to be used for encrypting each of the ml secrets. Two more hash operations will be required for encrypting each of the ml secrets using K_A and K'_A. Thus, a total of $(1 + 2)ml$ DOWN operations and $(2 + 2 + 2)ml$ hash operations are called for. As $ml = 2^{18}$ the total overhead is 3×2^{18} DOWN operations and 6×2^{18} hash operations.

For accepting ml secrets and rencrypting them 3 ml DOWN cycles and 3 ml hash operations will be required.

7.4.1.4 RPS and HARPS

For RPS and HARPS with parameters (P, k) (and L for HARPS) it is beneficial to choose P/k as a power of two. In other words, the P keys in the KDC's pool can be seen as k sets of keys each with P/k keys. Every node receives one key from each set. The index of the ith key assigned to a node A can be readily computed as $a_i^j = h(A \parallel i)$ where $0 \leq a_i^j \leq P/k - 1$ is truncated to $\log_2(P/k)$ bits. The value a_i^j identifies the index of the secret (within the subpool i) to be assigned to A. In HARPS the index *and* the hash depth can be obtained as $a_i^j \parallel d_i^j = h(A \parallel i)$ where d_i^j is a $\log_2 L$ bit value.

Once an index a_i is identified using one hash operation, the KDC can compute the ith secret to be assigned to A as as $K_i = h(\chi \parallel i \parallel a_i)$ (one DOWN operation and one hash operation). For encrypting each secret two DOWN operations and two hash operations will be called for (a total three DOWN operations and four hashes per issued secret). For HARPS apart the secret K_i will need to hashed (on an average $L/2$ times) before it is issued. Thus, the total number of operations per issued secret is three DOWN operations and $4 + L/2$ hash operations.

For RPS with $k = 171,233$ the overall KDC complexity is about 1.95×2^{18} DOWN operations and 2.6×2^{18} hash operations. For HARPS with $k = 107,088$ and $L = 64$ the overall KDC complexity is about 1.22×2^{18} DOWN operations and 14.7×2^{18} hash operations.

For accepting k secrets $3k$ hash operations and $3k$ DOWN cycles will be called for.

Table 7.1 Complexity of PKPSs designed to meet the requirement $p(n = 1420) = 2^{-64}$. Deployment complexity

	KDC (Generating secrets)		Node (Receiving secrets)	
PKPS	$\mathcal{C}_T$	$\mathcal{D}_T$	$\mathcal{C}'_M$	$\mathcal{D}'_M$
SKIT	2^{18}	2^{17}	2^{17}	2^{17}
PBK	6×2^{18}	3×2^{18}	3×2^{18}	3×2^{18}
PLM	2^{18}	2^{19}	192	192
RPS	2.6×2^{18}	1.95×2^{18}	1.96×2^{18}	1.96×2^{18}
HARPS	14.7×2^{18}	1.22×2^{18}	1.23×2^{18}	1.23×2^{18}

Table 7.1 summarizes various facets of complexity associated with *deployment* of probabilistic KPSs—$\mathcal{C}_T$ is the computational overhead to be borne by the KDC for generating secrets for a node, performed in $\mathcal{D}_T$ DOWN cycles. $\mathcal{C}'_M$ is the computational overhead for a node to accept the secrets issued by the KDC, where the process calls for $\mathcal{D}'_M$ DOWN cycles. From the perspective of complexity of deployment of the KPS all schemes are comparable. The only exception is PLM, where the complexity for receiving secrets is substantially lower due to the fact that public values generated by the KDC can be directly stored without additional processing.

7.4.2 *Complexity During Regular Operation*

During regular operation the the three components of every device will need to perform some tasks in order to compute any pairwise secret. For a node A, $\mathcal{S}_S$ values will need to be stored by S_A. For computing any pairwise secret $\mathcal{C}_A$ computations will have to be performed to identify $\mathcal{B}_M$ encrypted secrets/public values to be fetched from storage. This will be followed by supplying $\mathcal{B}_M$ values to mode A_M. Module A_M will require an internal storage of $\mathcal{S}_M$ to store secrets and inputs provided by A_D. Module A_M will then have to perform $\mathcal{C}_M$ PRF operations in $\mathcal{D}_M$ DOWN cycles to compute the pairwise secret.

7.4.2.1 SKIT

At node A, the ml encrypted tickets can be stored in storage S_A. Thus, $\mathcal{S}_S = ml = 2^{16}$. The $m = 32$ secrets $K_i^{a_i}$ and the secret K_A^T used for encrypting all secrets are stored inside the module (encrypted using module A's master secrets μ_A).

To compute a pairwise secret K_{AB}, the device A_D:

1. Computes $[a_0 \dots a_{m-1}] = f(A)$ to identify the 32 encrypted tickets that need to be fetched from storage A_S
2. Fetches 32 tickets $T_i^{b_i}(A)'$ from A_S (or $\mathcal{B}_D = 32$)
3. XORs all 32 (encrypted) tickets together to obtain

$$T_B(A)' = T_0^{b_0}(A)' \oplus T_1^{b_1}(A)' \dots \oplus T_{31}^{b_{31}}(A)' \tag{7.17}$$

4. Provides $T_B(A)'$ (along with the identity B) to the module A_M ($\mathcal{B}_M = 2$)

Note that each of the 32 encrypted tickets is of the form

$$\begin{aligned} T_i^{b_i}{}' =& T_i^{b_i} \oplus E_i^{b_i} \text{ where} \\ E_i^{b_i} =& h(K_A^T \parallel i \parallel b_i) \end{aligned} \tag{7.18}$$

Thus, the XOR of the 32 unencrypted tickets, viz.,

$$\begin{aligned} T_B(A) =& T_0^{b_0}(A) \oplus \ldots \oplus T_{31}^{b_{31}}(A) \\ =& (T_0^{b_0}(A)' \oplus E_0^{b_0}) \oplus \ldots \oplus (T_{31}^{b_{31}}(A)' \oplus E_{31}^{b_{31}}) \\ =& T_B(A)' \oplus E^B \text{ where} \\ E^B =& E_0^{b_0} \oplus \ldots \oplus E_{31}^{b_{31}}. \end{aligned} \tag{7.19}$$

The module A_M:

1. Computes $[a_o \ldots a_{m-1}] = f(A)$ and $[b_o \ldots b_{m-1}] = f(B)$
2. Computes $m = 32$ tickets

   $$T_i^{a_i}(B) = h(K_i^{a_i} \parallel B), 0 \le i \le 31 \tag{7.20}$$

   that B has access to, and XORs all of them together as

   $$T_A(B) = T_0^{a_0}(B) \oplus \ldots \oplus T_{31}^{a_{31}}(B) \tag{7.21}$$

   For decrypting the m keys $K_i^{a_i}$ m DOWN operations and m hash function operations will be required. For computing m tickets, m additional hash operations will be required (a total of 2 m hash operations in m DOWN cycles).
3. Computes $E_i^{b_i} = h(K_A^T \parallel i \parallel b_i), 0 \le i \le m-1$ to determine the secrets used for encrypting the tickets $T_0^{b_0}(A) \ldots T_{31}0^{b_{31}}(A)$ fetched from storage, and XORs them together as

   $$E^B = E_0^{b_0} \oplus \ldots \oplus E_{31}^{b_{31}}. \tag{7.22}$$

   For decrypting K_A^T one DOWN operation and one hash operation is required. For computing the m E_is, m hash operations are called for ($m + 1$ hash operations and 1 DOWN cycle for computing E^B).
4. Finally, module A_M computes the pairwise secret as

   $$K_{AB} = T_B(A) \oplus E^B \oplus T_A(B) \tag{7.23}$$

Thus, for module $A_M, \mathcal{C}_M = 2m+m+1+6 = 103$—$2\,m$ $h()$ operations are required to compute $T_A(B)$, $m + 1$ operations for computing E^B, and six hashes to compute $f(A)$ and $f(B)$. The number of DOWN cycles required is $\mathcal{D}_M = m + 1 = 33$. The storage complexity for A_M is $\mathcal{S}_M = m + 1 + 2 = 35$ (32 secrets of the form $K_i^{a_i}, 0 \le i \le 31$, the secret K_A^T, and two values—the XORed tickets and the identity of the peer B—provided as inputs).

7.4.3 PLM

For PLM with $m = 64, l = 2^{12}$ ($ml/2 = 2^{17}$), the $ml/2$ public values can be stored in storage S_A. Thus, $\mathcal{S}_S = ml/2 = 2^{17}$. The $m = 64$ secrets are decrypted and stored inside the module.

For computing a pairwise secret, the device A_D:

1. Computes $f(B)$ and $f(A)$ ($\mathcal{C}_D = 2 \times 5 = 10$)
2. Fetches (on an average) $m/2$ public values from storage ($\mathcal{B}_D = m/2 = 32$)
3. XORs them together to obtain P_{AB}
4. Provides P_{AB} to A_M (or $\mathcal{B}_M = 1$)

Thus, $\mathcal{B}_D = m/2 = 32$, $\mathcal{C}_D = 10$, and $\mathcal{B}_M = 2$.

The module A_M stores $m = 64$ secrets. The module

1. Computes $f(B)$
2. Computes m secrets $S_i = h(K_i^{a_i} \parallel B)$ and XORs them together, and computes the pairwise secret as

$$K_{AB} = S_0 \oplus \ldots \oplus S_{63} \oplus P_{AB} \tag{7.24}$$

This calls for m DOWN cycles in which each $K_i^{a_i}$ is decrypted, and m hash operations: $\mathcal{C}_M = m + 5 = 69$, and $\mathcal{S}_M = m + 2 = 66$, and $\mathcal{D}_M = m = 64$.

7.4.4 PBK

For PBK with $m = 64, l = 2^{12}$ ($ml = 2^{18}$) the ml encrypted secrets can be stored in S_A—$\mathcal{S}_S = ml = 2^{18}$.

To enable A_M to compute a pairwise secret K_{AB}, the device A_D:

1. Computes $f(A) = [a_0 \ldots a_{63}$ and $f(B) = b_0 \ldots b_{63}$.
2. Fetches $m = 64$ encrypted keys $K_{a_i}^{b_i}{}'$ from storage ($\mathcal{B}_D = m = 64$)
3. Computes

$$S' = K_{a_0}^{b_0}{}' \oplus \ldots \oplus K_{a_{63}}^{b_{63}}{}' \tag{7.25}$$

4. Supplies two values—S' and the identity B to A_M

Thus, $\mathcal{B}_D = m = 64$, $\mathcal{C}_D = 10$, and $\mathcal{B}_M = 2$.

To compute the pairwise secret the module:

1. Computes $f(A) = [a_0 \ldots a_{63}$ and $f(B) = b_0 \ldots b_{63}$
2. Computes m values of the form $E_i = h(\mu_A \parallel a_i \parallel b_i)$—the values used to encrypt the secrets stored in storage—and computes

$$K_{AB} = E_0 \oplus \ldots \oplus E_{63} \oplus S' \tag{7.26}$$

Thus, $\mathcal{C}_M = m + 10 = 74$, $\mathcal{D}_M = m = 64$ and $\mathcal{S}_M = 2$.

7.4.5 RPS and HARPS

For RPS with parameters P and k the k encrypted secrets can be stored in in A_S (or $\mathcal{S} = 171,233$. To determine the shared secret, device A_D

1. Computes $F(A)$ and $F(B)$ (or $\mathcal{C}_D = 2k$.
2. The $m \approx 121$ encrypted secrets $K'_{j_1} \ldots K'_{j_{121}}$ are fetched from storage: $\mathcal{B}_D = 121$
3. The 121 encrypted secrets are XORed together as

$$K'_{AB} = K'_{j_1} \oplus \ldots \oplus K'_{j_{121}} \tag{7.27}$$

4. Values K'_{AB} and B are inputs to module A_M, along with the m common indices. Let us assume that 121 (for example, 16-b) indexes is equivalent to 12 keys ($\mathcal{B}_M = 2 + 12 = 14$).

The module A_M now needs to:

1. Verify the m indexes $j_1 \ldots j_m$. To verify that $f(A,i) = f(B,i) = j_i$ two hash operations will be required.
2. Compute $E_i = (\mu_A \parallel i \parallel j_i)$ for each shared index (121 PRF operations in 121 DOWN cycles) and compute

$$K_{AB} = E_1 \oplus \ldots \oplus E_{121} \oplus K'_{AB} \tag{7.28}$$

Thus $\mathcal{C}_M = 3\,m = 363$ and $\mathcal{D}_m = m = 121$.

For HARPS with parameters the k encrypted secrets can be stored in in A_S (or $\mathcal{S} = 107,088$. To determine the shared secret, device A_D:

1. Computes $F(A)$ and $F(B)$ to determine the $m = k^2/P \approx 133$ (on an average) shared indexes $j_1 \ldots j_m$. Thus, $\mathcal{C}_D = 2k$.
2. The $m \approx 133$ encrypted secrets $K'_{j_1} \ldots K'_{j_{133}}$ are fetched from storage: $\mathcal{B}_D = 133$
3. The 133 values $K'_{j_1} \ldots K'_{j_{133}}$ along with 133 shared indexes are inputs to module A_M ($\mathcal{B}_M = 133 + 13$).

Module A_M:

1. Verifies $F(A)$ and $F(B)$ for the shared indexes $j_1 \ldots j_m$ (about $2\,m = 266$ $h()$ operations).
2. For each shared index j_i compute S_i as follows:
   (a) Compute $E_i = (\chi_A \parallel i \parallel j_i)$, and $S'_i = E_i \oplus K'_{j_i}$ to decrypt the stored secret
   (b) Depending on the hash depths of A and B corresponding to the index j_i the secret S'_i may need to be hashed to reach the same depth as the other node. As the average difference between any two uniformly distributed numbers between one and L is $L/3$, and as only for half the keys a node has to hash forward, the average number of additional hashes per shared key is $L/6$.
3. All 133 hashes secrets are XORed together to yield K_{AB}.

Thus $\mathcal{C}_M = 2\,m + m + mL/6 \approx 1818$, and $\mathcal{D}_M = m = 133$.

Table 7.2 Complexity of PKPSs designed to meet the requirement $p(n = 1420) = 2^{-64}$ without the DOWN assurance, and $p(n \times d_b) = 2^{-64}$ with the DOWN assurance

	Storage	Device		Module			DOWN benefit
PKPS	$\mathcal{S}_S$	$\mathcal{C}_D$	$\mathcal{B}_D$	$\mathcal{S}_M$	$\mathcal{C}_M$	$\mathcal{D}_M$	d_b
SKIT	65,536	3	32	35	103	33	32
PBK	262,144	10	64	2	74	64	262,144
PBK′	393,216	9	24	2	33	24	393,216
PLM	131,072	10	32	65	69	64	64
RPS	171,233	171,233	121	14	363	121	171,233
HARPS	107,088	107,088	133	16	1818	133	107,088

Table 7.2 summarizes various facets of complexity associated with regular day-to-day use of probabilistic KPSs. RPS and HARPS can be immediately rejected as unsuitable due to the considerable overhead for devices—for computation of the public function to determine common indexes.

Among the three KPSs with low public function complexity, viz., SKIT, PLM, and PBK, PBK is clearly advantageous due to the fact that it requires the least overhead inside the module, and that it can take good advantage of the DOWN assurance. For PLM and SKIT taking advantage of the DOWN assurance will require a larger choice of parameter m, which will unfortunately increase both the computational overhead $\mathcal{C}_M$ and DOWN complexity $\mathcal{D}_M$. On the other hand, for PBK we can actually improve the DOWN assurance while lowering computational overhead $\mathcal{C}_M$ and DOWN complexity $\mathcal{D}_M$. For example, instead of $m = 64$ and $l = 2^{12}$, we can choose the $m = 24$ and $l = 2^{14}$ to achieve the same $p(n)$ security. However for such a scheme:

1. Computational overhead $\mathcal{C}_M$ and DOWN complexity $\mathcal{D}_M$ are lower by a factor $24/64 = 0.375$.
2. As ml increases by a factor 1.5, so does the benefit d_b accrued from the DOWN policy.

In Table 7.2 the overhead for PLM with lower m (and higher l and storage ml) is labeled PBK′.

7.5 KPS Algorithms

Irrespective of the specific nature of the KPS, the algorithm executed by two modules M_i and M_j for computing a pairwise secret K_{ij} can be represented as

$$K_{ij} = f_{pw}(\langle \mathbf{K}_i \rangle, M_j, P_{ij}) = K_{ji} = f_{pw}(\langle \mathbf{K}_j \rangle, M_i, P_{ji}) \tag{7.29}$$

where

1. $\mathbf{K}_i$ represents the secrets stored inside M_i
2. $\mathbf{K}_j$ represents the secrets stored inside M_j
3. P_{ij} and P_{ji} are pairwise non secret values

As secrets like $\mathbf{K}_i$ and $\mathbf{K}_j$ are always stored inside the module, they need not be provided as inputs to the module every time the function $f_{pw}()$ is invoked (this is the reason they are enclosed in angled brackets). Values like P_{ij} and P_{ji} can either be public values (in the case of MLS or PLM) or encrypted secrets (for SKIT and PBK).

The difference between the nonscalable MLS and scalable schemes (SKIT, PBK, and PLM) is that in the former every P_{ij} is independently stored. For scalable schemes the P_{ij} is obtained by XORing multiple values together.

Specifically, to obtain the value P_{ij} required for M_i to compute K_{ij}:

1. In MLS the public value P_{ij} is fetched from storage.
2. In SKIT m encrypted tickets are XORed together to obtain the input P_{ij}.
3. In PLM $m/2$ public values (on an average) are XORed together to obtain P_{ij}.
4. In PBK m encrypted secrets are XORed together to obtain P_{ij}.

Scalable KPSs also require evaluation of short IDs. Specifically,

1. In SKIT and PLM M_i requires to compute m the short IDs corresponding to M_j.
2. In PBK M_i requires to compute the short-IDs itself (M_i) and M_j.

Let us represent by $[j_1 \dots j_m] = f_{sid}(m, n, M_j)$ a function which generates m short IDs of M_j, each of length n bits, where $2^n = l$ for a (m, l) PKPS. As mentioned earlier, if the PRF outputs L bits hashes then ceil(mn/L) hashes will be computed to generated to compute m n-b short-IDs. The function $f_{sid}()$ is as follows:

```
$[j_1 \| \cdots \| j_m] := f_{sid}(m,n,M_j)\{$
    $t \leftarrow$ceil(mn/L)
    $tmp \leftarrow h(M_j)$
    FOR $c = 2 : t$
        $tmp \leftarrow tmp \| h(tmp)$;
    TRUNCATE $tmp$ to $mn$ bits
$\}$
```

Irrespective of the KPS, if DOWN policy is enforced, every module possesses a master secret. Let χ_i be the master secret in module M_i. Let us represent the DOWN operation as $\mathbf{D}()$. When invoked with a value X

$$\mathbf{D}(x) = h(\chi_i \parallel X) \tag{7.30}$$

returns as secret used to encrypt another secret with an index X.

However, the number of secrets $\mathbf{K}_i$ are different for different schemes. Specifically,

1. In MLS every module will need to store as many secrets as the number of independent KDCs. At least two KDCs should be used to take advantage of the DOWN assurance (at most one of the two secrets can be exposed from a snapshot). Let the encrypted secrets be K_i^1 and K_i^2. The secrets can be decrypted using a DOWN operation as $\mathbf{D}(1) \oplus K_i^1$ and $\mathbf{D}(1) \oplus K_i^2$ respectively.

2. In PBK no additional secret is necessary as all mM secrets are encrypted using the master secret and stored outside. The decryption secret for a secret with index (c_i, c_j) stored outside is $\mathbf{D}(c_i \parallel c_j)$.
3. In PLM m secrets (say) $K_i^1 \dots K_i^m$ are stored inside the module. Specifically, even inside the module they are stored encrypted using the master secret; only one of the m secrets will be revealed by any snapshot. The secret K_i^c can be decrypted using a DOWN operation $\mathbf{D}(c)$.
4. In SKIT the module M_i stores $m + 1$ secrets used to compute tickets, and one secret used to decrypt stored tickets. Let the $m + 1$ secrets be $K_i^1 \dots K_i^m$, and K_i^t respectively. The secret K_i^c where $c = 1 \dots m$ can be decrypted using a DOWN operation $\mathbf{D}(c)$. The secret K_i^t is decrypted using $\mathbf{D}(m + 1)$.

The function $f_{pw}()$ executed inside module for various KPSs can be algorithmically represented as follows.

7.5.1 MLS

Two DOWN operations are required to decrypt the two secrets assigned to M_i. Unlike scalable KPSs function $f_{sid}()$ is not required.

```
$K_{ij} := f_{pw}^{mls}(M_j, P_{ij})$ {
    RETURN $h((\mathbf{D}(1) \oplus K_i^1) \parallel M_j) \oplus h((\mathbf{D}(2) \oplus K_i^2) \parallel M_j) \oplus P_{ij}$;
}
```

7.5.2 Scalable KPSs

In PBK M_i will need to compute $[i_1 \dots i_m]$ and $[j_1 \dots j_m]$ to determine the index (i_c, j_c) (where $1 \leq c \leq m$) required for performing the DOWN operation.

```
$K_{ij} := f_{pw}^{PBK}(M_j, P_{ij})$ {
    $[i_1 \cdots i_m] = f_{sid}(m, n, M_i)$;
    $[j_1 \cdots j_m] = f_{sid}(m, n, M_j)$;
    $K \leftarrow P_{ij}$;
    FOR ($c = 1 : m$)
        $K \leftarrow K \oplus \mathbf{D}(i_c, \mathrm{j}_c)$;
    RETURN $K$;
}
```

In PLM the m secrets stored inside the module can be decrypted using DOWN operations $\mathbf{D}(1) \dots \mathbf{D}(m)$. Each secret then needs to be hash extended using a short ID of M_j.

```
$K_{ij} := f_{pw}^{plm}(M_j, P_{ij})$ {
  $[j_1 \cdots j_m] = f_{sid}(m, n, M_j)$;
  $K \leftarrow P_{ij}$;
  FOR ($c = 1 : m$)
    $tmp \leftarrow \mathbf{D}(c) \oplus K_i^c$;
    $K \leftarrow K \oplus h(tmp \,||\, j_c)$;
  RETURN $K$;
}
```

In SKIT the $m + 1$ secrets stored inside the module can be decrypted using DOWN operations $\mathbf{D}(1) \ldots \mathbf{D}(m)$ and $\mathbf{D}(m+1)$. After decryption each secret is used to compute a ticket by hash extending the secret using the identity M_j. The short IDs of M_j are required to decrypt the encrypted tickets (XORed together as P_{ij}).

```
$K_{ij} = f_{pw}^{skit}(M_j, P_{ij})$ {
  $[j_1 \cdots j_m] = f_{sid}(m, \mathrm{n}, M_j)$;
  $K \leftarrow P_{ij}$;
  FOR ($c = 1 : m$)
    $tmp \leftarrow \mathbf{D}(c) \oplus K_i^c$;
    $K \leftarrow K \oplus h(tmp \,||\, M_j)$;
  $tmp \leftarrow \mathbf{D}(m+1) \oplus K_i^t$;
  FOR ($c = 1 : m$)
    $K \leftarrow K \oplus h(tmp \,||\, j_c)$;
  RETURN $K$;
}
```

7.6 Security Protocols Utilizing $f_{pw}()$

The shared key K_{ij} is typically used for two broad purposes:

1. Authenticating a value, or
2. Encrypting a secret

Specifically, a process U_i with access to module M_i can utilize the module to convey an authenticated value/secret to a process U_j with access to module M_j.

The authenticator for a value v emanating from module M_i for purposes of verification by a module M_j, will take the form of a message authentication code (MAC) computed using the secret K_{ij}. Specifically, the MAC for value v may be computed using a PRF $h()$ as

$$A_{ij} = h(v \parallel K_{ij}). \tag{7.31}$$

For conveying a secret K to module M_j, module M_i can encrypt secret K using K_{ij}. More specifically, the M_i chooses a random nonce N and computes

$$K' = K \oplus h(K_{ij} \parallel N) \tag{7.32}$$

as the encrypted version of K. On receipt of the encrypted secret and the nonce N, the receiver may now decrypt the secret as

$$K = K \oplus h(K_{ij} \parallel N) \tag{7.33}$$

If the simple operations above for computing/verifying MACs or for encrypting/decrypting secrets are also performed inside the trusted boundary of the modules, there is no reason for the processes like U_i and U_j that employ the modules to gain access to shared secrets like K_{ij}.

More generally, authenticated values and encrypted secrets may also be atomically relayed by modules. We already saw the utility of relaying authenticated values in the trusted computing base–domain name server (TCB–DNS) protocol outlined in Sect. 4.5. In the rest of this section we provide a more useful generalization of such atomic relay protocols, where the module functionality $f_{pw}()$ is utilized to compute pairwise secrets between modules.

7.6.1 Atomic Relay Protocols

An atomic relay function executed by a module M_i receives as inputs authenticated values from an upstream module M_u. Relayed values may be modified in a specific manner to provide a context. For example, to indicate that a value v was received from M_u, the module M_i may relay a value $v' = h(v \parallel M_u)$ to the downstream module M_d.

The atomic relay function can be used for relaying authenticated values or secrets. Specifically, relaying an authenticated value serves the purpose of broadcasting an authenticated value to any number of receiving processes. Recall that such a function was used in TCB–DNS (Sect. 4.5) to relay authenticated values corresponding to DNS records created by zone owners, to any number of DNS clients who may desire a specific record created by the zone owner. In TCB–DNS the authenticated values were relayed over a fixed number of hops. A value created by a zone owner was relayed over two hops to DNS clients. At the first hop is the module associated with a name server for the zone, and at the second hop was the preferred name server (or the local DNS server) employed by the client to perform iterative queries. Similarly, the atomic relay function can also be used to relay secrets over multiple hops to result in unique path based secrets. In this section we take a deeper look at atomic relay protocols.

7.6.2 Atomic Authentication Relay Algorithm

The inputs to an atomic relay algorithm $F_{ra}()$ can be classified into the following categories:

1. $[M_u, P_u, M_d, P_d]$—Values needed for $f_{pw}()$ to compute pairwise secrets with the upstream module M_u and downstream module M_d.
2. $[v, \alpha]$—Authenticated values provided by the upstream module M_u. The value α is an authenticator (a MAC) for the value v.
3. A value o for representing path length.

The output of $F_{ar}()$ is an authenticator α_i (which is the input α for the downstream module).

Using the inputs $[M_u, P_u, M_d, P_d]$ the module M_i executing $F_{ar}()$ can readily compute the shared pairwise secrets K_{iu} (between M_i and M_u) and K_{id} (between M_i and M_d) as

$$K_{iu} = f_{pw}(M_u, P_u)\text{and}K_{id} = f_{pw}(M_d, P_d) \tag{7.34}$$

If the input $M_u = 0$, the implication is the inputs $[v, \alpha]$ to F_{ar} were *not* provided by an upstream module. Specifically, the v has been provided by the untrusted process U_i with access to M_i.

When $M_u = 0$ the authenticator for v is computed by M_i as

$$\alpha_i = h(v \parallel o = 1 \parallel K_{id}) \tag{7.35}$$

The value $o = 1$ conveys to the downstream module that M_i is the initiator of the relay.

If $M_u \neq 0$ the implication is that the values $[v, \alpha]$ provided as inputs to M_i were outputs of $F_{ar}()$ executed by M_u. The integrity of the MAC α is verified as

$$\alpha = h(v \parallel o \parallel K_{iu}) \tag{7.36}$$

where o is the hop count of the upstream module.

On successful verification, the verified value v can be relayed to the downstream module M_d as $v_i = h(v \parallel M_u)$ indicating that v was received from M_u. The authenticator α_i is computed as

$$\alpha_i = h(v_i \parallel o + 1 \parallel K_{id}). \tag{7.37}$$

Note that the authenticator is bound to the incremented hop count.

The process U_i employing M_i is free to decide if the inputs $[v, \alpha]$ should merely be verified, or if the values should be relayed onward. For the former scenario $F_{ar}()$ is invoked with $M_d = 0$ (no downstream module). For the latter scenario, $M_d \neq 0$.

The algorithm for $F_{ar}()$ can be represented as follows:

```
$A_d := F_{ar}([M_u, P_u, M_d, P_d], [v, \alpha], o)\{$
  $K_{iu} \leftarrow f_{pw}(M_u, P_u); K_{id} \leftarrow f_{pw}(M_d, P_d);$//Compute Keys Shared with $M_u$ and $M_d$
  IF ($M_u = 0$)
    RETURN $\alpha_i \leftarrow h(v \parallel 1 \parallel K_{id});$
  ELSE IF($M_u \neq 0$)
    IF ($\alpha \neq h(v \parallel o \parallel K_{iu})$) RETURN 0;
    IF ($M_d > 0$) RETURN $\alpha_i = h(h(v \parallel M_u) \parallel o + 1 \parallel K_{id});$
    ELSE RETURN $v$;
$\}$
```

A value v initiated by a process with access to module M_A, relayed over a path that includes M_B and M_C will be received any module M_D downstream of M_C as

$$v_c = h(h(h(v \parallel M_A) \parallel M_B) \parallel M_C) \tag{7.38}$$

Accompanying this will be an authenticator for hop count $o = 3$. the hop count prevents the process U_A associated with M_A to choose (for example) $v = h(v' \parallel M_X)$ to effectively implicate M_X as the originator of a value v'.

The main difference between the function $F_{ar}()$ and the atomic relay function used in TCB–DNS is that the latter did not require a hop count, as the path length was always the same. In TCB–DNS values like v are created by zone owners and relayed by two intermediate modules—one associated with an zone name server for the zone and the second associated with a preferred name server employed by the querying stub resolver.

Note that from the perspective of a receiver, there in no way to differentiate between a value $v = h(v' \parallel M_k)$ initiated by M_i and a value v' initiated by M_k. If path lengths can be variable, then the hop count is required to eliminate such ambiguities. Specifically, as M_i did not receive the value v' from M_k it will set the hop count to $o = 1$ in the authenticator relayed downstream. On the other hand, if M_i had actually received a value v' from M_k then the hop count in the authenticator relayed downstream will be $o = 2$.

7.6.3 Atomic Path Secret Relay Algorithm

The inputs to an atomic relay algorithm $F_{ras}()$ are

1. $[M_u, P_u, M_d, P_d]$—Values needed for $f_{pw}()$ to compute pairwise secrets with the upstream module M_u and downstream module M_d.
2. $[v, \alpha, N]$—Authenticated values provided by the upstream module M_u. N is a nonce necessary to decrypt the encrypted secret v, and the MAC α enables verification of the integrity of the decrypted secret.
3. A flag r which specifies an instruction by the previous hop indicating if the secret should be relayed onward.

The outputs of $F_{ars}()$ include a value v_i (which is the input v to the downstream module), an authenticator α_i (which is the input α for the downstream module) and a nonce N_i.

Using the inputs $[M_u, P_u, M_d, P_d]$ the module M_i executing $F_{ars}()$ can easily compute the shared pairwise secrets K_{iu} (between M_i and M_u) and K_{id} (between M_i and M_d).

If the input $M_u = 0$, the implication is the inputs $[v, \alpha, N]$ to F_{ars} were not provided by an upstream module. Instead, the v has been provided by the untrusted process U_i with access to M_i. In such a scenario the module M_i generates a nonce N_i. The secret v is first encrypted as to

$$v_d = h(v \parallel r \parallel M_d) \tag{7.39}$$

where r is an instruction to M_d.

The secret v_d is then encrypted using a generated nonce N_i and the pairwise secret K_{id} as

$$v_i = h(K_{id} \parallel N_i) \oplus v_d \tag{7.40}$$

to obtain the output v_i.

The value r provided as input serves as an instruction to the downstream module.

1. If $r = 0$ the next hop M_d is instructed to *not* forward the secret onward. M_d is required to return the secret to the process invoking $F_{ars}()$ in M_d.
2. If $r = 1$ the secret should only be relayed onward, and can not be returned to the process.

The authenticator α_i is computed as a function of the secret v_d and the instruction r as

$$\alpha_i = h(v_d \parallel r \parallel K_{id}) \tag{7.41}$$

7.6.4 Accepting Relays

If $M_u \neq 0$ the implication is that the values $[v, \alpha, N]$ provided as inputs to M_i were outputs of $F_{ars}()$ executed by M_u.

The secret is first decrypted as

$$v' = h(K_{id} \parallel N) \oplus v \tag{7.42}$$

Unlike the scenario for relaying authenticated values, the process U_i does not have the freedom decide if the secret should be returned to the invoking process U_i or relayed onward. Whether M_i will relay the secret or return the secret to U_i depends on the value r bound to the authenticator α provided by the upstream module.

If the upstream module had set $r = 0$ for purposes of computing the authenticator, then the function $F_{ars}()$ should be invoked with $M_d = 0$ (no downstream module should be specified). On the other hand, if the upstream module had specified $r = 1$ then M_d should not be zero.

If $M_d = 0$ the authenticator for the received secret is verified as

$$\alpha = h(v' \parallel 0 \parallel K_{iu}) \tag{7.43}$$

On successful verification the value v' is returned to the process U_i. If $M_d \neq 0$ the authenticator for the received secret is verified as

$$\alpha = h(v' \parallel 0 \parallel K_{iu}) \tag{7.44}$$

The secret is then used to generate the secret

$$v_d = h(v' \parallel r \parallel M_d) \tag{7.45}$$

to be conveyed to the downstream module M_d, along with the instruction to relay ($r = 1$) or to not relay ($r = 0$).

The secret v_d is encrypted as

$$v_i = h(K_{id} \parallel N_i) \oplus v_d \tag{7.46}$$

to obtain the output v_i, where N_i is also provided as an output. The authenticator for v_i and the instruction r bound to v_i is

$$\alpha_i = h(v_d \parallel r \parallel K_{id}). \tag{7.47}$$

The algorithm for $F_{ars}()$ can be represented as follows:

```
$A_d := F_{ars}([M_u, P_u, M_d, P_d], [v, \alpha, N], r)\{$
  $K_{iu} \leftarrow f_{pw}(M_u, P_u)$;$K_{id} \leftarrow f_{pw}(M_d, P_d)$;//Compute Keys Shared with $M_u$ and $M_d$
  ELSE IF($M_u = 0$)
    $N_i = f_{rsg}()$;
    $v_d \leftarrow h(v \parallel r \parallel M_d)$;
    $v_i \leftarrow h(K_{id} \parallel N_i) \oplus v_d$;
    $\alpha_i \leftarrow h(v_d \parallel r \parallel K_{id})$;
    RETURN $N_i, v_i, \alpha_i$;
  ELSE IF($(M_u \neq 0) \wedge (M_d = 0)$)
    $v_i \leftarrow h(K_{iu} \parallel N) \oplus v$;
    IF ($\alpha = h(v_i \parallel 0 \parallel K_{iu})$) RETURN $v_i$;
  ELSE IF($(M_u \neq 0) \wedge (M_d > 0)$)
    $v' \leftarrow h(K_{iu} \parallel N) \oplus v$;
    IF ($\alpha \neq h(v' \parallel 1 \parallel K_{iu})$) RETURN 0;
    $v_d = h(v' \parallel r \parallel M_d)$;$N_i = f_{rsg}()$;
    $v_i \leftarrow h(K_{id} \parallel N_i) \oplus v_d$;
    $\alpha_i \leftarrow h(v_d \parallel r \parallel K_{id})$;
    RETURN $N_i, v_i, \alpha_i$;
  }
```

For purposes of relaying a secret, a secret v relayed over a path M_B and M_C for consumption by the process associated with module M_D will take the form

$$\begin{aligned} v_d &= h(v_c \parallel 0 \parallel M_D) \text{ where} \\ v_c &= h(v_b \parallel 1 \parallel M_C) \text{ where} \\ v_b &= h(v \parallel 1 \parallel M_B) \end{aligned} \tag{7.48}$$

Note that all relays should be associated with a value $r = 1$ and the terminal (M_D in this case) with $r = 0$.

In order to permit multiple independent processes to utilize a trusted module, untrusted processes may themselves be issued "module" identities and secrets corresponding to such identities. Some bits in the identity of every "module" can specify if the module is trusted or untrusted. Thus, for example, all DNS clients employing a local DNS server associated with module M_p can be seen by M_p as "modules" to which authenticated values/secrets can be relayed.

As module identities of all relaying modules are bound to authenticated values, authentication information that includes untrusted modules in the path can be ignored.

Untrusted modules can however be terminal points for authenticated values and path secrets.

Once again the difference between the atomic relay strategy discussed in Sect. 4.7 and the more generic version outlined in this section is that the instruction *r* was not used in the former. In the former scenario, all intermediaries (modules associated with a zone name server and a preferred name server) can only relay decrypted secrets, and not return secrets to the calling process.

7.7 Conclusions

The choice of an appropriate key distribution strategy is influenced by several factors. Ultimately, algorithms for computing pairwise secrets will need to be executed inside trustworthy tamper-responsive boundaries. From the perspective of lowering overhead inside the trusted boundary, MLS is the ideal choice as it requires the least overhead inside the trusted boundary, and that it is not susceptible to collusion.

While MLS may be sufficient for a wide range of applications, the limit on its scalability, albeit a soft limit, can render MLS unsuitable for some application scenarios. Until we develop substantial confidence in the ability to offer close to absolute guarantee of the integrity of the master secret in tamper-responsive devices (as this is a prerequisite for the DOWN assurances) SKIT may be a wiser choice for scenarios demanding unlimited scalability (due to the fact that SKIT offers the best (n, p)-security for a fixed amount of storage, and that SKIT also offers the best resistance to attacks that can leverage brute-force search capabilities. Thereafter, PBK is a better choice.

Chapter 8
Broadcast Authentication and Broadcast Encryption

A broadcast authentication (BA) scheme permits any receiver to verify the authenticity of the source and the integrity of the contents of the broadcast message. This can be achieved using digital signatures if public key cryptography is used. However, in many application scenarios, resource constraints may prohibit the use of asymmetric cryptographic primitives. BA schemes that employ only symmetric cryptographic primitives include one-time signature (OTS) schemes, schemes like TESLA, based on asymmetry of time, and schemes based on probabilistic key predistribution.

Broadcast encryption (BE) [88] provides a means of establishing shared secret between g privileged devices, among a set of $G = g + r$ devices, where the r devices which are not provided with the secret are usually referred to as *revoked* devices. BE schemes can be broadly classified into tree-based schemes and "flat" schemes based on probabilistic key predistribution.

8.1 Certificates-Based Broadcast Authentication (BA)

BA schemes can be broadly classified into certificates-based schemes and identity-based schemes. In certificates-based schemes, the signer has complete freedom to choose the private key R and compute the corresponding public key U. Consequently, the public key U reveals no information about the identity of the signer. This is the reason that a certificate authority is required to certify the binding between a public key and an identity/credential. Recall that in identity-based schemes, the private key is assigned by a key distribution center (KDC)—corresponding to a public value (identity) assigned by the KDC. In identity based schemes, no certificate is required.

Certificates-based schemes employing only symmetric cryptographic primitives include one-time signature schemes and TESLA.

8.1.1 One-Time Signatures (OTS)

In public key schemes, a message signed using a private key R can be verified using the corresponding public key U. Unlike an asymmetric key pair (R, U), which can be

M. Ramkumar, *Symmetric Cryptographic Protocols,* DOI 10.1007/978-3-319-07584-6_8,

used for signing any number of messages, one-time-signature (OTS) key pair (r, u) can be used for signing only a single message (or equivalently, a b-bit hash).

An OTS key pair (r, u) is generated by choosing a private key r and computing the public key as

$$u = g_{ots}(r) \tag{8.1}$$

where the OTS key generation function $g_{ots}()$ involves several applications of a PRF $h()$.

8.1.1.1 One-Time Signatures (OTS) Key Pair Generation

If the hash to be signed is b-bits long, the signer chooses:

1. values n_p, l_p such that $n_p \log_2 l_p \geq b$; and
2. values n_s and l_s such that $l_s^{n_s} \geq n_p(l_p - 1)$.

For example, for $b = 80$, the signer can choose $n_p = 10, l_p = 256, n_s = 2, l_s = 51$ $(n_p \log_2 l_p = 10 \times 8 = 80, l_s^{n^s} = 51^2 = 2601 > n_p(l_p - 1) = 10 \times 255 = 2550)$. As another example, the signer can choose $n_p = 8$, $l_p = 1024$, $n_s = 2$ and $l_s = 91$ $(n_p \log_2 l_p = 8 \times 10 = 80, l_s^{n^s} = 91^2 = 8281 > n_p(l_p - 1) = 8 \times 1023 = 8184)$.

The private key r is used to generate:

1. n_p "primary" hash chains, each of length l_p, and
2. n_s "secondary" hash chains, each of length n_s.

The *head* values of the primary chains can be readily computed from r as

$$P_i^0 = h(r, i, 0), 1 \leq i \leq n_p. \tag{8.2}$$

Each head value is hashed repeatedly to create the respective tail values

$$P_i^{l_p - 1} = h^{l_p - 1}(P_i^0), 1 \leq i \leq n_p. \tag{8.3}$$

Similarly, the head values of the n_s secondary hash chains are computed as $S_j^0 = h(r, j, 1)$. Let their respective tails be $S_j^{l_s - 1}, 1 \leq j \leq n_s$. The $n_p + n_s$ tails

$$P_1^{l_p - 1} \cdots P_{n_p}^{l_p - 1} \text{ and } S_1^{l_s - 1} \cdots S_{n_s}^{l_s - 1} \tag{8.4}$$

are hashed together to yield the public key u.

To sign a b-bit value M using the key pair (r, u), M is interpreted as $n_p \log_2 l_p$-bit values $m_1 \parallel \cdots \parallel m_{n_p}$. For example, if $b = 80, n_p = 10, l_p = 256$, then the 80-bit M is seen as 108-bit values $0 \leq m_i \leq 255$.

Let $X = \sum_{i=1}^{n_p} m_i$. Now let $x_j, 1 \leq j \leq n_s$ represent the digits of a base-l_s representation of X, or

$$X = \sum_{j=1} n_s x_i^{j-1} \tag{8.5}$$

Let u_j be the complement of x_j (or $u_j = l_s - 1 - x_j, 1 \leq j \leq n_s$). The signature for M is then $n_p + n_s$ hashes, $P_i^{m_i}, 1 \leq i \leq n_p$ and $S_j^{u_j}, 1 \leq j \leq n_s$ (one value from each chain).

Verifying the signature is performed by:

1. repeatedly hashing the values $P_i^{m_i}, 1 \leq i \leq n_p$, $l_p - 1 - m_i$ times (to arrive at the tail values of each primary chain);
2. computing X and thus the x_j values;
3. hashing values $S_j^{u_j}, 1 \leq j \leq n_s$, x_j times (to arrive at the tail values of the secondary chains); and
4. hashing all tails together to arrive at the public key u.

If the result is indeed the public key u, the verifier can conclude that only an entity with access to the corresponding private key r could have computed the signature.

Note that given a value $P_i^{m_i}, 1 \leq i \leq n_p$ and $S_j^{u_j}, 1 \leq j \leq n_s$, only entities with access to the private key r can compute values $P_i^{m'_i}$ for $m'_i < m_i$ and $S_j^{u'_j}$ for $u'_j < u_j$. While any one can compute $P_i^{m'_i}$ for $m'_i > m_i$, in such cases $X' = \sum_{i=1}^{n_p} m'_i = \sum_{j=1}^{n_s} x'_j l_s^{j-1}$ will be such that at least one $x'_j > x_j$ (or at least one $u'_j < u_j$). Thus, without access to the private key r, computing a valid signature for any message $M' \neq M$ will require inverting the one-way function $h()$.

8.1.2 Timed Efficient Stream Loss Tolerant Authentication (TESLA)

In "timed efficient stream loss tolerant authentication" (TESLA) [12], the signer A generates a hash chain $\{K_A^0, K_A^1, \ldots, K_A^{L-1}, K_L^A\}$ by choosing a random secret (head of the chain) K_A^0, and repeatedly hashing the value to generate the remaining values in the chain. The last value in the chain, K_A^L is made public.

Each value in the chain is associated with an absolute value of time. A TESLA key associated with time t_i should be kept a secret until time t_i. The key can be used to compute an hashed message authentication code (HMAC). Any verifier who receives the HMAC *before* time t_i can verify the HMAC *after* time t_i, when the TESLA key used for computing the HMAC is made public.

Unlike digital signatures and OTS schemes, TESLA-based authentication is delay sensitive, and does not facilitate instantaneous verification. In a TESLA hash chain

$$\mathcal{H}_{t_0,\Delta}^A = \{K_A^0, K_A^1, \ldots, K_A^{L-1}, K_L^A\} \tag{8.6}$$

with public key (or commitment) K_L^A is associated with an absolute value of time t_0 and a period (or TESLA interval) Δ, for example, $\Delta = 1$ s, and t_i is December 1 2014 0600:00 GMT. The signer and all verifiers are assumed to agree on the current time within a small margin of error (less than the interval Δ).

A is required to keep the key K_A^{L-i} private till time $t_i = t_0 + i\Delta$. The key K_A^{L-i} can be used for authenticating any arbitrary value M by appending an HMAC

$h(M, K_A^{L-i})$, provided the HMAC reaches potential verifiers *before* time t_i (when K_A^{L-i} is still a secret known only to A). Once K_A^{L-i} is made public (*after* time t_i), the verifier can (1) verify the TESLA HMAC using K_A^{L-i}; and (if consistent) (2) repeatedly (i times) hash K_A^{L-i} and verify that the result is indeed K_L^A.

8.2 Identity-Based Broadcast Authentication (BA) Using Key Predistribution

For broadcast authentication using key predistribution, the KDC chooses a set of ml indexed secrets $\{K_{ij}\}, 0 \leq i \leq m-1, 0 \leq j \leq l-1$, and a one-way function $f()$, which generates m values between 0 and $l-1$.

Corresponding to a node assigned identity A, where $f(A) = a_0 \cdots a_{m-1}$, node A is assigned m secrets,

$$\mathbb{S}_A = K_{0a_0} \cdots K_{(m-1)(a_{m-1})} \tag{8.7}$$

and ml identity tickets

$$\mathbb{T}_A = \{T_{i,j}^A = h(K_{ij} \parallel A)\}, 0 \leq i \leq m-1, 0 \leq j \leq l-1. \tag{8.8}$$

To sign a message (or equivalently, the hash M of a message), the signer computes ml message authentication codes (MAC) using each of its $k = ml$ tickets. For example, the MAC corresponding to a ticket T_{ij}^A is

$$\mu_{ij}^M = h(T_{ij}^A \parallel M). \tag{8.9}$$

A node B assigned keys defined by indexes $f(B) = b_0 \cdots b_{m-1}$ can verify m of the ml MACs. Specifically, the m MACs verifiable by B are $\mu_{0b_0}^M \cdots \mu_{(m-1)(b_{m-1})}^M$.

For an (n, p_f)-secure broadcast authentication scheme, an attacker who has secrets of n nodes can forge the signature of an arbitrary source (not included in the n compromised nodes) for the purpose of fooling a specific verifier, with a probability p_f.

A node C can compute $\mu_{ib_i}^M$ if $c_i = a_i$ (probability $1/l$). The probability that an entity who has access to secrets of n nodes *cannot* compute a specific MAC, $\mu_{ib_i}^M$ is

$$\epsilon = (1 - 1/l)^n \approx \mathrm{e}^{-\mathrm{n/l}}. \tag{8.10}$$

Thus, the probability that the n-attacker collusion can compute *all* m MACs verifiable by B (and thus impersonate A for purposes of deceiving B) is

$$p_f(n) = (1 - \epsilon)^m = (1 - \mathrm{e}^{-\mathrm{n/l}})^{\mathrm{m}}. \tag{8.11}$$

For a desired level of security, say $p_f(n) < p^* \forall n < n^*$, if we desire to limit the size of the signature (which is proportional to $k = ml$, the number of MACs appended by the signer), the choice of m and l that minimizes ml are

$$\left.\begin{array}{ll} m = & \frac{\log(1/p^*)}{\log 2} = \log(1/p^*) \\ l = & \frac{n^*}{\log 2} = \end{array}\right\} \Rightarrow k = \frac{1}{\log^2 2} n^* \log(1/p^*). \tag{8.12}$$

For example, for $p_f = 2^{-64}$, the bandwidth minimizing choice of parameters is $m = 64$ and $M \approx 1.44n$.

8.2.1 Reducing Signature Size

The bandwidth required for $k = ml$ MACs is $\beta = mlb$, where b is the size of each MAC. To conserve bandwidth, each of the ml MACs may be truncated. Each MAC may even be truncated to 1 bit. This may however open up another line of attack for the attacker—just "guessing" the signature bits that cannot be obtained by using the exposed secrets. Fortunately, success by guessing is not as favorable for the attacker as the ability to actually compromise the keys and compute the signature bits.

In situations where the attacker has to resort to guessing a few MACs, the attacker has no way of knowing a priori if the impersonation attempt is going to succeed. Furthermore, while the attacker may succeed in impersonating A for fooling B for a specific message, success is not guaranteed for the next message—as the attacker will have to guess all over again. In scenarios where the attacker may need to carefully orchestrate attacks where a series of messages (with authenticated misleading information) need to be propagated, blindly guessing the MAC—even if the attacker has to guess just one bit in each attempt—certainly cramps the attacker's ability.

When we truncate MAC sizes, we need to modify the metric used for defining the security of such schemes. An (n, p_f, w)-secure scheme implies that attacker desiring to forge the signature of a node using secrets exposed from n other nodes can identify only 1 in $1/p_f$ potential receivers such that for fooling such a receiver the attacker will only have to guess w or less bits (and thus, succeed with a probability $1/2^w$). In other words, while we had earlier defined the probability of attacker failure $(1 - p_f)$ as the probability that at least one of the m verifiable MACs should be safe, according to the new metric "at least $u = w/b$ of the m MACs should be safe," in a scenario where each MAC is truncated to b bits.

As ϵ is the probability that any MAC is safe, the probability less than u MACs (among the m MACs that can be verified by a specific verifier) are safe, or the probability that the attacker has to guess at least w or less bits) is

$$p'_f(n) = \sum_{0}^{u-1} \binom{m}{u} \epsilon^u (1-\epsilon)^{m-u}. \tag{8.13}$$

If we choose (say) $w = 8$, and if we desire to limit the signature size to (say) 1024 bits some reasonable options for the choice of parameters are as follows:

1. $ml = 1024$, 1-bit MACs
2. $ml = 512$, 2-bit MACs,
3. or more generally, $ml = 1024/b$, b-bit MACs.

In a scenario where an attacker has exposed secrets from n nodes, if we choose $b = 1$, we desire that at least $w/b = w = 8$ of the m MACs that can be verified by a specific

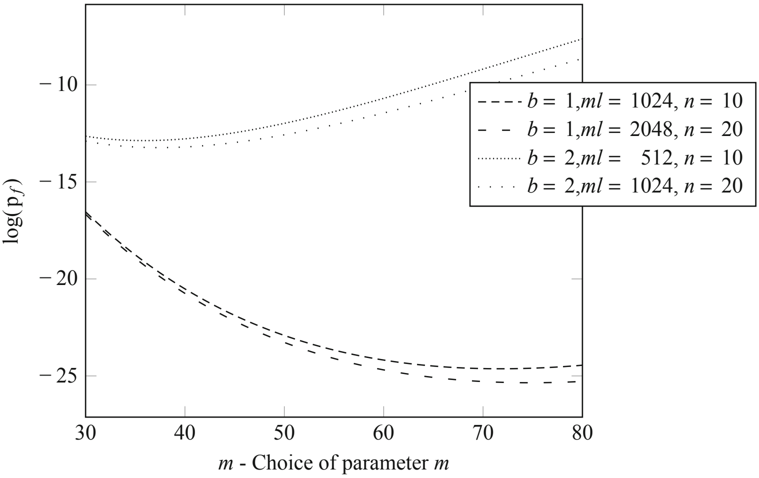


Fig. 8.1 Plots of probability of forgery p_f for various choice of parameter m (x-axis). For all four plots mlb/n—the ratio of the total bandwidth mlb for the signature and the number of nodes n in the attacker collusion is the same. The *dashed plots* are for the choice of 1-bit message authentication codes (*MACs*). The *dotted plots* are for 2-bit MACs

verifier cannot be obtained from keys exposed by the attacker (by tampering with n nodes). If $b = 2$ instead, we desire that at least four of the m MACs that can be verified by a specific verifier cannot be obtained from keys exposed by the attacker.

As for any p_f the signature size will need to increase linearly with n, we shall compare the (n, p_f, p_g) security of schemes constrained to demand the same signature bandwidth $\beta = mlb$. Figure 8.1 depicts the probability of forgery p_f for an attacker who has access to secrets from n nodes. For all four plots, the parameters m (the number of MACs verified by any node), l and b (ml is the total number of b-bit MACs in the signature) have been chosen such that the ratio of the total bandwidth (mlb) to the number of colluding nodes (n) is the same. The x-axis depicts various choices of m (higher m implies lower l to keep ml constant). The y-axis is the logarithm of the probability of forgery p_f.

From Fig. 8.1, it is clear that for any n, and any choice of m and l, choosing $b = 1$ leads to lower p_f (the dashed lines are clearly better as they correspond to substantially lower p_f). For example, when the signature is constrained to be 1024 bits long 1024 1-bit MACs) $m = 72$ (and $l = 1024/72 \approx 14$) is the best choice of parameters against an attacker who has secrets from $n = 10$ nodes.

Table 8.1 depicts the bandwidth efficient choice of parameters (in terms of p_f and n) to realize an (n, p_f, w)-secure BA scheme (with 1-bit MACs) for various values of w.

If we desire an $(n = 10, p_f = 2^{-32}, w = 8)$-secure scheme, then the optimal choice of parameters is $m = 2.42 \log p_f \approx 54$ and $l = 1.35 \times 10 \approx 14$. The signature will consist of $ml = 756$ bits. Any verifier can verify $m = 54$ bits.

Table 8.1 Bandwidth efficient choice of parameters m and l for a desired (n, p_f, w)-security. The parameters are indicated in terms of n and p_f. Note that $l \propto n$ and $m \propto \log p_f$, and the bandwidth $mlb = ml \propto n \log p_f$

w	$\frac{l}{n}$	$\frac{-m}{\log p_f}$	$\frac{\beta}{(-n \log p_f)}$
32	1.14	12.39	14.19
16	1.27	3.96	5.04
8	1.35	2.42	3.27
6	1.37	2.14	2.95
4	1.39	1.90	2.65
3	1.40	1.79	2.50
2	1.41	1.67	2.36
1	1.43	1.56	2.23

8.2.2 *Effect of Decrypt Only When Necessary (DOWN) Assurance*

With the DOWN assurance, as only one of the m secrets of a node can be exposed by tampering with a node, the attacker has to expose one secret from nm nodes in order to expose an equivalent of "all m secrets from n nodes." Thus, an $(n = 10, p_f = 2^{-32}, w = 8)$-secure BA scheme with $(m = 54, l = 14)$ is rendered $(n_d = mn = 540, p_f = 2^{-32}, w = 8)$-secure with the DOWN assurance.

For the $(n_d = 540, p_f = 2^{-32}, w = 8)$-secure scheme every node is assigned $ml = 756$ tickets. The signer requires 756 hashes to compute the MACs, and outputs the last bit of each MAC. Verifiers will be required to compute $m = 54$ hashes. The storage required for each node, for $ml = 756$ tickets and $m = 54$ secrets, is about 16 KB (assuming 128-bit secrets and tickets).

While it is not desirable to increase the computational overhead or bandwidth overhead to improve the security, we can afford to increase the storage overhead. The DOWN assurance allows us to take advantage of plentiful storage. For example, let us assume that each node can afford an s-fold increase in storage. For example, $s = 1024$ implies a storage requirement of 16 MB per node instead of 16 KB. In this case, each node can be issued sml tickets and sm secrets. For computing MACs for authenticating a value M, only ml of the sml tickets will be chosen. The specific ml tickets can be dictated by a simple one way function of the message M to be authenticated. For example, if $u = \log 2(sml)$ then a function

$$f(M) = m_0 \cdots m_{ml-1} \tag{8.14}$$

generates ml u-bit values $m_0 \cdots m_{ml-1}$, which dictates the indexes of the tickets to be used for computing MACs. Thus, as earlier, the signer has to compute only ml MACs, and each verifier verifies (on an average) m MACs. The bandwidth overhead still remains ml.

Compared the $(n_d = 540, p_f = 2^{-32}, w = 8)$-secure scheme which required 16 KB, the modified scheme which requires the same computational and bandwidth overhead, but an $s = 1024$ increases in storage overhead (16 MB), is $(n_d = 552, 960, p_f = 2^{-32}, w = 8)$-secure. If 160 MB storage per node is acceptable, then the scheme is rendered $(n_d = 5, 529, 600, p_f = 2^{-32}, w = 8)$-secure.

8.3 Broadcast Encryption

Broadcast encryption (BE) [88] provides a means of establishing a shared secret between g privileged devices, among a set of $G = g + r$ devices, where the r devices which are not provided with the secret are referred to as *revoked* devices.

BE schemes involve a set-up phase where secrets are distributed to all devices in the network. To disseminate a broadcast secret K_b to all devices (except r explicitly excluded devices), the source

1. encrypts K_b using n keys $K_{e1} \cdots K_{en}$, and
2. broadcasts n values[1] $K_{ei}(K_b), 1 \leq i \leq n$.

The secrets $K_{e1} \cdots K_{en}$ are chosen in such a way that none of the $r = G - g$ revoked devices can (using the secrets they possess) determine *any* of the keys $K_{e1} \cdots K_{en}$. On the other hand, each of the $G - r$ privileged devices should have access to *at least one* of the secrets $K_{e1} \cdots K_{en}$, and consequently, the broadcast secret K_b.

The capability to establish and control access to group secrets has a wide variety of applications like digital rights management (DRM) [89], publish–subscribe systems [90], and multicast communications [91]. For instance, in DRM applications, regulating access to content C is realized by encrypting content with a *content encryption key* K_C. The content encryption key is encrypted with the group secret K_G (and the key $K_G(K_C)$ may be distributed along with the encrypted content) to ensure that only members of the group can gain access to K_C, and hence, the content.

The ability to protect any secret depends on the number of members that have access to the secret. Obviously, the higher the number of members with access to a secret, the higher is the susceptibility of the secret to exposure. With BE, *all* except a few revoked members share the group secret. In practical application scenarios involving group secrets, some proactive measures are required to protect the group secrets. For instance, the group secrets, and the predistributed secrets that are leveraged for disseminating the group secrets, should be protected by a tamper-responsive module. Limiting such modules to perform only symmetric cryptographic primitives is obviously a useful strategy.

8.3.1 Tree-Based Broadcast Encryption (BE) Schemes

Many efficient BE schemes that utilize only symmetric cryptographic primitives have been proposed in the literature. Most solutions [92, 93] are tree-based, where the KDC keys are maintained in a binary tree-like structure. A small set keys from the tree are assigned to every intended receiver. The receiver utilize their keys to decrypt broadcast secrets.

[1] the notation $K(M)$ represents encryption of a quantity M using a secret K, using a standard symmetric cipher.

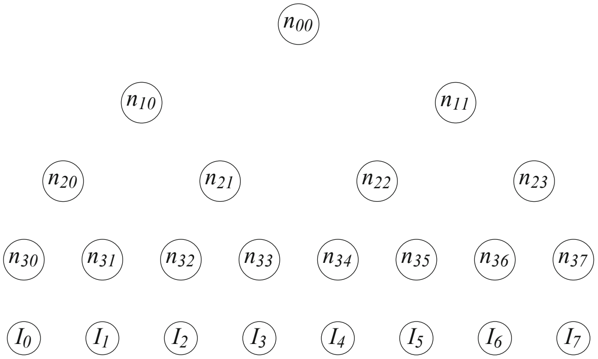


Fig. 8.2 Complete subtree broadcast encryption (*BE*) scheme

In the complete subtree scheme for a system with $N = 2^L$ devices, the N devices are assumed to correspond to the leaf nodes of a binary tree of depth L. Each of the $2N - 1 = \sum_{i=0}^{L} 2^i$ nodes in the binary tree, $n_{ij}, 0 \leq i \leq L, 0 \leq j \leq 2^i - 1$ is associated with a secret K_{ij} chosen by the KDC.

Each device is also associated with L direct ancestor nodes, one in each level of the tree. A device I_l receives $L + 1$ secrets, L secrets associated with its L ancestor nodes, and the secret K_{Ll} corresponding to the leaf node n_{Ll}.

Figure 8.2 depicts such a tree for $L = 3$ (or $N = 8$). Device I_3 corresponds to the node n_{33} with ancestors n_{21}, n_{10} and n_{00}, and thus receives secrets K_{33}, K_{21}, K_{10}, and K_{00}. All devices receive K_{00}, half the devices receive K_{10}, and so on.

To revoke I_5 the KDC encrypts a broadcast secret K_b with three secrets—K_{34}, K_{23}, and K_{10}, and broadcasts

$$\mathcal{B} = [I_5 \parallel (K_{34}(K_b), K_{23}(K_b), K_{10}(K_b))]. \tag{8.15}$$

Any device receiving the broadcast $\mathcal{B}$ can determine which secrets have been used by the source (as the revoked node is explicitly specified). Devices $I_0 \cdots I_3$ can decrypt $K_{10}(K_b)$. I_6 and I_7 can decrypt $K_{23}(K_b)$. I_4 can decrypt $K_{34}(K_b)$. In general, revoking any device will call for L encryptions of the broadcast secret. Note that any number of devices can be revoked together, in one broadcast. Further, while revoking one device calls for using $\log_2 N$ encryptions, revoking more than one, say $r > 1$ devices, will require *less* than $r \log_2 N$ encryptions. Even in the worst case scenario only $r \log_2 (N/r)$ encryptions are called for [92].

8.3.1.1 Multisource Extensions

In tree-based schemes, the source of the broadcast is assumed to be the KDC, who distributes the secrets in the first place. However, BE schemes that cater for broadcasts by multiple sources have some very useful applications.

In multisource BE schemes, the keys distributed by the KDC can be used to securely receive broadcast secrets from any number of broadcast sources. Many multisource BE schemes employing public key primitives exist [94, 95]. Some tree-based schemes can also be readily extended to cater for multisource BE *if* asymmetric cryptographic primitives are employed.

Extending a tree-based BE scheme to facilitate broadcasts by multiple sources involves interpreting the value assigned to each of the $2N - 1$ nodes in the binary tree as a *public* value, corresponding to which secrets (or private keys) are assigned to every device. Thus, corresponding to each of the $2N - 1$ nodes n_{ij}, the KDC generates public–private key pairs $\{(U_{ij}, R_{ij})\}$.

Each device stores $L + 1$ private keys. In this case, device I_3 stores private keys R_{33}, R_{21}, R_{10}, and R_{00}. The public values U_{ij} of all $2N - 1$ nodes are made public (provided to all potential sources of broadcasts). Any source with the knowledge of the public keys can encrypt the broadcast secret using the public keys, which only devices with the corresponding private keys can decrypt.

8.3.2 Broadcast Encryption (BE) Using Probabilistic Key Distribution

Probabilistic key predistribution schemes (PKPSs) like hashed random preloaded subsets (HARPS) and random preloaded subsets (RPS) can also be used for encrypting broadcast secrets. Unlike tree-based schemes, BE schemes based on PKPSs can also be used for broadcasts by multiple sources. As HARPS, defined by parameters (P, k, L) is a generalization of RPS (or HARPS with $L = 1$), we shall begin by illustrating the principle behind BE using HARPS. We shall initially restrict ourselves to BE by the KDC, and then discuss simple extensions to facilitate multisource BE.

Consider, for example, HARPS with $P = 8, k = 4, L = 4$ depicted below. The KDC chooses $P = 8$ keys $K_1 \cdots K_8$. Device A has been issued four keys with indexes $i = 1, 2, 4, 6$ at hash depths 4, 2, 1, and 3, respectively (keys K_1^4, K_2^2, K_4^1, and K_6^3). The row marked d_i is the hash depth, the KDC can *safely* employ for each $1 \leq i \leq P$ for encrypting K_b. For example, for revoking A and B, the KDC can use keys K_1^3, K_2^1, K_5^2, K_6^1, K_7^4, and K_7^4 as none of the secrets can be determined by A or B, or even by A and B pooling their secrets together. Device C can determine K_7^4 by hashing its secret K_7^2 twice, and thus decrypt K_b. Device D can determine K_1^3 or K_6^1 or K_8^4 (Table 8.2).

In the case of RPS, where all secrets have the same hash depth, each device either has a key corresponding to some index, or does not. In this case, the KDC can choose the key corresponding to indexes 7 and 8 to encrypt the broadcast secret (to revoke A and B). As it turns out in this (overly simplified) example, both C and D can employ the key with index 7.

Table 8.2 Example illustrating hashed random preloaded subsets broadcast encryption (HARPS BE). The table depicts the hash depths (or x if the key is not assigned) of keys corresponding to each index assigned to four devices

i	1	2	3	4	5	6	7	8
A	4	2	x	1	x	3	x	x
B	x	3	1	x	3	2	x	x
d_i	3	1	x	x	2	1	4	4
C	x	3	4	1	x	x	2	x
D	2	x	x	x	3	1	4	x

8.3.3 Broadcast Encryption (BE) by Sources Other Than Key Distribution Center (KDC)

One of the main advantages of PKPS-BE schemes comes from the fact that they trivially cater for BE by any source. The KDC can authorize any source, say a content distributor Θ, to perform broadcast encryption by providing the distributor Θ with PL identity tickets

$$\mathfrak{S}_\Theta = \{T_\Theta^{i,j} = h(K_i^j \parallel \Theta)\}, 1 \leq i \leq P, 1 \leq j \leq L. \tag{8.16}$$

In the case of RPS, the broadcast source Θ is issued just P tickets

$$\mathfrak{S}_\Theta = \{K_\Theta^i = h(K_i \parallel \Theta)\}. \tag{8.17}$$

To revoke r device, just as the KDC can use a subset (or hashed subset) of the secrets $K_1 \cdots K_P$—say $K_{I_i}^{d_1} \cdots K_{I_n}^{d_n}$, to encrypt the broadcast secret K_b, the external source Θ can use $T_\Theta^{I_1,d1} \cdots T_\Theta^{I_n,d_n}$ to encrypt K_b. A device that has a secret K_i^d can still compute tickets $K_\Theta^{i,d'} = h(K_i^{d'} \parallel \Theta)$ for any $d' \geq d$.

8.4 Performance of Probabilistic Key Predistribution Scheme Broadcast Encryption (PKPS BE)

In order to broadcast the secret to all but r devices, none of the tickets that can be computed using any of the keys of the r revoked devices may be used for encrypting the broadcast secret.

A ticket $T_{i,j}$ is safe for purposes of conveying a broadcast secret only if the node of the r revoked devices is assigned a key with index high at a hash depth j or lower. Consider a key indexed i, which say u of the r revoked devices possess. Let the hash depths of those u keys be $d_1 \cdots d_u$, with $d' = \min(d_1 \cdots d_u)$. The ticket $T_{i,j}$ can be safely used as long as $j < d'$.

Let n_j be the average number of such safe tickets at hash depth $1 \leq j \leq L$. With these $n = \sum_{j=1}^{L} n_j$ encryptions of the broadcast secret, the KDC hopes to "reach" (or convey to secret K_b to) *every* privileged device. In order to decrypt a secret encrypted with a ticket index i at depth j, a device should have a secret indexed i at depth $d \leq j$, which will occur with a probability $\frac{\xi j}{L}$. Obviously, encryption keys corresponding to higher hash depths will be more effective in conveying the secret to more privileged nodes.

The value n_L corresponds to the keys that none of the r devices have (at *any* hash depth). The probability that any device has a key indexed i is $a = \frac{k}{P}$. The probability that none of the r devices have key i is $(1 - a)^r$. In other words,

$$n_L = P(1 - a)^r. \tag{8.18}$$

For RPS (with $L = 1$), $n = n_L = P(1 - a)^r$ is the *total* number of safe keys. For HARPS, n_j may be nonzero even for hash depth less than L. Specifically, the expected number of safe keys at a hash depth j is

$$n_j = P \sum_{u=1}^{r} \binom{r}{u} a^u (1 - a)^{(r-u)} \frac{(L - j)^u - (L - j - 1)^u}{L^u}. \tag{8.19}$$

As any key is assigned to any node with probability $a = k/P$, the probability that exactly u of r nodes have a secret corresponding to some index i is the binomial probability $B_a(r, u) = \binom{r}{u} a^u (1 - a)^{(r-u)}$. Let the u keys be $K_i^{d_1} \cdots K_i^{d_u}$. The probability $j + 1 = d' = \min(d_1 \cdots d_u)$ is

$$\begin{aligned}\Pr\{d' = j + 1\} =& \Pr\{d' > j\} - \Pr\{d' > j + 1\} \\ =& \frac{(L - j)^u - (L - j - 1)^u}{L^u}.\end{aligned}$$

While it is guaranteed that none of the r devices can decipher the broadcast secret, there is a possibility that some of the $g = G - r$ privileged devices may not be able to decrypt *any* of the $n = \sum_{i=1}^{L} n_j$ encryptions. Let p_o be the "outage probability"—the probability that an arbitrary device among the group of g privileged devices cannot decrypt *any* of the n encryptions.

For a particular encryption key at hash depth j (or any one of the n_j keys), the probability of outage is $p_{o_j} = (1 - a\frac{j}{L})$. In general, broadcast source may not have to use all the $n = \sum_{j=1}^{L} n_j$ possible safe keys. Only a subset $n_e = \sum_{j=q}^{L} n_j$ keys may be used to achieve a target outage probability of p_o^*. For instance, the source will first try to use only keys at depth L (as they will be more useful for more privileged nodes), and if necessary consider using keys at depth $L - 1$, and so on, till a target outage probability is reached.

Assume that the source uses all possible safe keys at depth greater than q, and $n_q{}' \leq n_q$ keys at depth q. In this case, the probability of outage for any device, and hence the total number of encryptions n_e^* required to convey the broadcast secret to all privileged nodes are

$$p_o^* = (1 - \xi \frac{q}{L})^{n'_q} \prod_{j=q+1}^{L} (1 - \xi \frac{j}{L})^{n_j}, \tag{8.20}$$

$$n_e^* = gp_o^* + n_q' + \sum_{j=q+1}^{L} n_j, \tag{8.21}$$

where the term gp_o^* accounts for the accidentally missed devices. For instance, if source chooses a target of $p_o^* \approx 1/g \approx 1/G$, one of the g devices will be accidentally missed *on an average*, for every revocation. To reach the missed devices, either an additional safe key can be added for each missed device (as the source does not typically use all possible safe keys to achieve the target of p_o), or they can be conveyed by encrypting the broadcast secret with a unique key provided to each device.

Apart from broadcasting the several encryptions of the secret, recall that for tree-based schemes the broadcast should indicate the identities of the revoked nodes (for example, I_5 in Eq (8.15)). For PKPS-BE, while this is possible, it is more efficient, both in terms of bandwidth needed and computational complexity at the receiver, to instead provide the *indexes* and the hash depths of the keys used to encrypt the broadcast secret.

8.4.1 Performance Bounds

The exact analytical expressions for the relationship between P, k, L and the number of encryptions n_e^* (Eqs. (8.19)–(8.21)) necessary to revoke r devices, can be used readily for evaluating the performance of PKPS-BE for various choices of P, k, L, and r. However, they provide very little intuition regarding the bounds of performance. To gain some more insight we shall look more closely at the simpler case (RPS) where no hashing employed.

With P secrets chosen by the KDC, and $k = aP$ provided to each device, as the total number of safe tickets is $n = P(1-a)^r$ the minimum achievable probability of outage for revoking r devices, is

$$p_{min} = (1-a)^n = (1-a)^{P(1-a)^r} \approx 1/G_{max}, \tag{8.22}$$

where G_{max} is the maximum possible group size. Obviously p_{min} can be reduced to any extent (or G_{max} increased to any extent) by increasing P. The question now is what is the “optimal” choice of a for some G, r?

If we wish to minimize P, the optimal choice of a should minimize $\mathbf{C} = (1-a)^{(1-a)^r}$, which occurs when $a \approx 1/r$ (for large r). Also, for large r

$$(1-1/r)^r \to \mathrm{e}^{-1} \Rightarrow \mathrm{p_{min}} = (1-\mathrm{a})^{\mathrm{P/e}}. \tag{8.23}$$

As $log(1-a) \approx -a$ for small a (or large $r \approx 1/a$), we have

$$P \approx \mathrm{er}\log \mathrm{G_{max}} \quad k \approx \mathrm{e}\log \mathrm{G_{max}} \quad n_e \approx r \log G_{max}. \tag{8.24}$$

where $\lambda = p_{min}^{-1} \approx G_{max}$. For a network size of one billion (2^{30}), where we would desire $p_{min} \approx 2^{-30}$, for $r = 128$, we require $k \approx 57$, $P = rk = 7296$, calling for $n_e \approx 2661$ encryptions of the broadcast secret.

A better measure is the number of encryptions n_e required for revoking r devices. Now instead of choosing $\xi = 1/r$, let us instead choose $\xi = b/r, b > 1$. In this case, $p_{min} = (1 - b/r)^{P(1-b/r)^r} \approx (1 - b/r)^{P\mathrm{e}^{-\mathrm{b}}}$, or

$$P \approx r\frac{\mathrm{e}^{\mathrm{b}}}{b} \log(1/p_{min}) \quad k \approx \mathrm{e}^{\mathrm{b}} \log(1/\mathrm{p}_{\mathrm{min}}) \quad n_e \approx \frac{r}{b} \log(1/p_{min}). \tag{8.25}$$

In other words, if we increase P by a factor $\mathrm{e}^{\mathrm{b}-1}/\mathrm{b}$, and k by a factor e^{b-1}, we can reduce the bandwidth needed for conveying the broadcast secret by a factor b. For $b = 4$ for instance, $P = 36636, k = 1145$, but n_e reduces to 665 encryptions for revoking 128 devices (for a group size of 1 billion).

8.4.2 Over-Provisioning Keys

For a group size of $G = 2^{30}$ (or $p_o = 1/G$), RPS with parameters $P = r\mathrm{e}\log \mathrm{G} = 7296, \mathrm{k} = \mathrm{e}\log(\mathrm{G}) = 57$ can revoke $r = 128 = P/k$ device, using $r \log G$ encryptions of a broadcast secret. With this choice of parameters

1. of the $P = r\mathrm{e}\log \mathrm{G}$ possible secrets of the KDC, only a fraction P/e are "safe" (on an average), when r devices have to be revoked;
2. all $n_e = r \log G$ safe secrets are *required* to achieve the target outage probability of $p_o = G^{-1}$. Thus, even if r is less than 128, the KDC will still need to transmit $n_e = 1/a \log G$ encryptions in order to convey the secrets to the $G - r \approx G$ privileged nodes (or achieve outage probability $p_o \approx G^{-1}$).

In other words, for $r < 128$, the bandwidth efficiency per revoked node reduces (or n_e/r increases, as n_e remains the same), and for r much larger than 128, the system is unusable.

However, now consider a scenario where the same $P = 7296, k = 57$ scheme is used for a group size of $G = 2^{10}$ (thousand, instead of a billion). In this case, we are actually employing P and k three times larger than what is required to revoke 128 devices[2]. Alternately, we can interpret this approach as a scheme corresponding to the choice of $b = 2.11$ (as $\mathrm{e}^{\mathrm{b}} \log(2^{10}) = 57$) to reduce bandwidth by a factor b (see Eq (8.25)), and designed for $r^* = b * 128 \approx 270$ and $G = 2^{10}$.

Thus, for a group size of $G = 2^{10}$ the ($P = 7296, k = 57$), scheme can revoke upto 270 devices with $n_e/r = (\log G)/b = 3.29$ encryptions per revoked device. At the same time, 128 devices can be revoked with an efficiency of $n_e/r = \log G = 6.93$, as only a fraction of the P/e safe secrets need to be employed for ensuring outage probability $p_o \approx \log(G^{-1})$.

[2] Choosing $k = 19, P = 2432$ would have sufficed.

While a system designed to minimize P, k for some r is efficient only for a narrow range of r, *by over-provisioning keys, we can realize efficient operation over a wider range* of r. As an other example, RPS with $P = 200 \times 128, k = 200$ can cater for efficient operation for a range of r between 128 and 340 for $G = 2^{30}$, and a range of 128 to 460 for $G = 2^{20}$ (a million). For larger ranges, we could increase k further, or alternately, employ *parallel deployments* of RPS with different values of $a = k/P$, so that together, they can be used efficiently for a wide range of r.

8.4.3 Hashed Random Preloaded Subsets (HARPS) vs. Random Preloaded Subsets (RPS)

Without over-provisioning keys, the KDC cannot revoke much more than P/k devices in a batch as the KDC *runs out* of the $n = P(1-a)^r$ safe secrets that can be used to convey the broadcast secret to the privileged devices in the group. However, in the case of HARPS, in addition to $n_L = P(1-a)^r$ safe secrets (corresponding to which none of the r nodes have a secret assigned) other $n_{L-1}, \dots n_1$ safe secrets are available.

While closed form expressions for the performance bounds of HARPS (akin to Eqs. (8.24) and (8.25) for RPS) are not readily tractable, the performance of HARPS can still be evaluated using the analytical expressions derived in the previous section (see Eqs. (8.19)–(8.21)), for the relationships between $n_e, r, (P, k, L)$ and $G \approx 1/p_o$.

Figure 8.3 depicts the performance of RPS and HARPS with the same $P = 7296, k = 57$ (and $L = 64$ for HARPS) in terms of n_e/r (y-axis) and the number of revoked devices, r (x-axis), for a group size of $G = 2^{30}$. Note that till the point r is not much larger than 128, both RPS and HARPS perform identically, as only keys at hash depth L are used for HARPS (keys at lower hash depths are not needed *yet*). But for larger r while RPS runs out of safe secrets/tickets, HARPS can begin using keys at lower hash depths, and thus continue to operate efficiently (even for $r > 500$ as can be seen from the figure).

Just as over-provisioning helped improve the usable range of r for RPS, it can also help to *further* improve the usable range of HARPS. For example, for $P = 25{,}600, k = 200$, for values of $L = 1, 2, 4, 6, 8, 16, 32$, and 64, respectively, the range of usable[3] r is between 128 and r_L, where r_L for different L is shown in the table below (the case $L = 1$ corresponds to RPS):

L	1	2	6	8	16	64
r_L	340	565	1040	1185	1520	2000

[3] We define the usable range $r_0 = 1\xi$ to r_L, where for revoking r_L devices the efficiency is the same as revoking $r_0 = 1/a$ devices. The maximum efficiency (or minimum n_e/r) occurs at some r between $r_0 = 1/a$ and r_L.

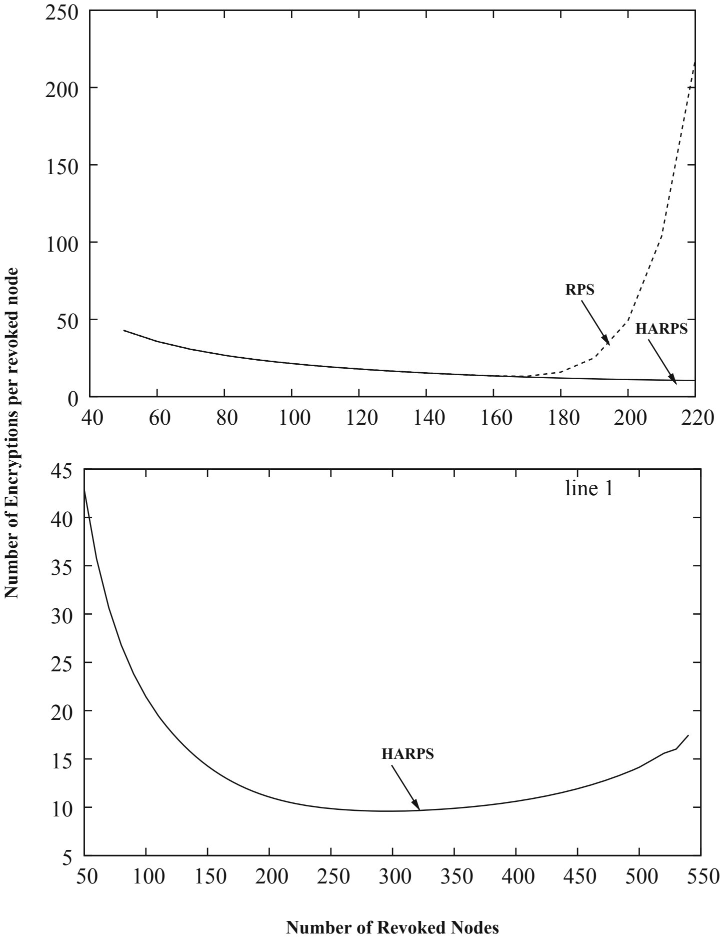


Fig. 8.3 Comparison of hashed random preloaded subsets (*HARPS*) and random preloaded subsets (*RPS*) for $P = 7296, k = 57$. For HARPS $L = 64$

Now consider a scenario where we desire to cater for efficient revocation for a range of r from 128 to say 1000, for a group size of 1 billion. With RPS, we can realize this by using two schemes, one with ($k = 200, a = 1/128$), and the second with ($k = 200, a = 1/360$). However (as can be seen from the table above), a single deployment of HARPS with $k = 200, a = 1/128, L = 6$ can meet this requirement (usable range of r).

In Fig. 8.4, the plot-labeled RPS depicts the performance (in terms of n_e/r for different r) when the two RPS schemes are used in parallel (the first is used for $r \leq 155$, and second for $r > 155$). The plot-labeled HARPS, $L = 6$ is for the case with a single HARPS deployment ($P = 25,600, k = 200, L = 6$), caters for a slightly larger range of r than two deployments of RPS used in parallel. As a quick comparison of the two approaches,

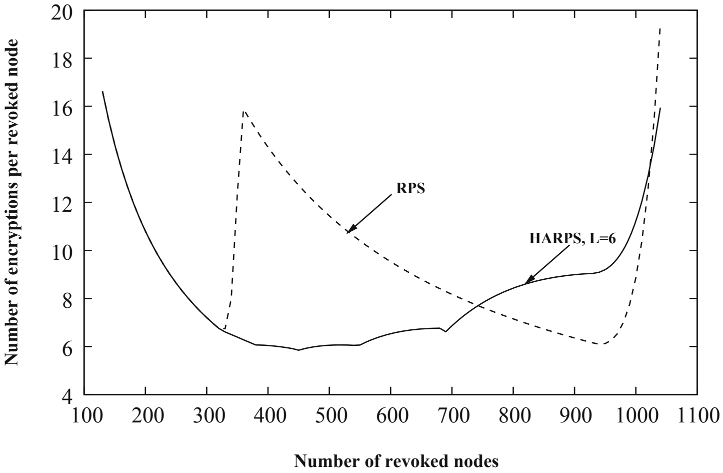


Fig. 8.4 Comparison of two approaches—one using two parallel deployments of random preloaded subsets (*RPS*) and the second using one deployment of hashed random preloaded subsets (*HARPS*) with $L = 6$

1. HARPS requires 200 secrets to be assigned to each device, RPS requires 400;
2. the KDC requires $P = 25600$ secrets for HARPS, and $25600+360*200 = 97600$ for RPS;
3. to facilitate BE by external sources, each external source requires $25600 \times L = 153600$ ($L = 6$) encryption secrets for HARPS, and $P = 97600$ (the same as the KDC) for RPS.

8.4.3.1 Choice of P, k, L for Practical Deployments

A reasonable approach then, to cater for efficient revocation for a wide range of r, may be to use say four independent deployments, (1) $a = 1/4, k = 100$, (2) $a = 1/16, k = 100$, to cater for small r, (3) $a = 128$ for r between 128 and 1050, and (4) $a = 1024, k = 200$ for larger r, for a total of 600 keys per device.

Further, the scheme supporting large r ($a = 1/1024$) could use large L (say 512) to facilitate batch sizes even upto 30,000. However, depending on the storage ability of the external source, the source does *not* have to store all $L \times 1024 \times 200$ encryption secrets for the scheme with $a = 1/1024$. For instance, if the external source decides to acquire only 1024×200 secrets at hash depth L, the deployment of HARPS can still be used with the same efficiency as RPS by the external source (while at the same time the KDC can employ much larger batch sizes). Note that in this case the external source has, and can thus use, only the $n_L = P(1 - a)^r$ safe secrets. Or the situation is no different from using RPS instead.

Thus, HARPS can simultaneously be used in the "RPS mode" by external sources that cannot afford to store a large number of secrets. Such a scheme (HARPS used in "RPS mode") can still be used by external sources for efficient revocation of up to 3000 devices (while the KDC can revoke upto 30,000 devices using $\sum_{j=1}^{L} n_L$ safe secrets). Furthermore, if the source can store $2 \times 1024 \times 200$ encryption secrets (say corresponding to hash depths L and $L/2$), the external source can support batch sizes upto 5000 nodes (in this case, for the external sources the scheme is equivalent to HARPS with $L = 2$).

By providing 200 more keys to each node of a $P = 25000 \times 200, k = 200, L = 1024$ (or $1/a = 25000$) HARPS scheme, the KDC can support batch sizes of up to a *million*. While this calls for a storage of 5 million keys by the KDC, in practice, the KDC does not have to actually *store* all the secrets—it could simply generate any secret *on demand* using a single (or a few) highly protected secrets using strong hash functions.

Thus, with HARPS, there is *almost no practical limitation* on the maximum batch size for the KDC. External sources, can also support sufficiently large batch sizes r with storage of $\mathbb{O}(r \log G)$. In most practical scenarios, even several GBs of storage is an inconsequential requirement for external sources (for example, distributors of digital content who may have to deal with thousands of TB of content). Furthermore, as we shall see in the next section, while it is desirable for the KDC to support large batch sizes, it is strictly *not* necessary for revocation by external sources to support large batch sizes.

8.5 Models for Broadcast Encryption (BE)

From the perspective of the KDC, the "network size" N is the number of devices that are assigned (or *could* be assigned) secrets. Most conventional models for BE assume that the group size G is the same as the network size N. Furthermore, the devices taking part in such deployments are also assumed to be *stateless* devices [92]. In other words, once keys are distributed to such devices, there is no way to provide them with new secrets. In most cases, the source of the broadcast (the entity which revokes devices) is also the entity that distributes the secrets in the first place—the KDC. On the other hand, there are many application scenarios calling for BE, where $N >> G$. In such scenarios, many independent sources (apart from the KDC) will be able to control group secrets for *their* specific interest groups consisting of perhaps $G << N$ users/devices.

8.5.1 $G = N$ *Models*

A practical example of a $G = N$ stateless model is the case of DVD content protection, where each DVD player is provided with a set of secrets (that cannot be modified during the lifetime of the device). By default, all DVD players can render all DVDs, unless explicitly revoked. The content in a DVD is encrypted with a content encryption key K_C, and the secret K_C is encrypted with a secret K_N. The secret

K_N is disseminated using BE (included in every DVD) so that only nonrevoked devices can gain access to K_N, and hence K_C, and thus decrypt the content. Typically, DVD players that are suspected to have been compromised by attackers are revoked. More specifically, a player is "compromised" when an attacker has exposed (or is suspected to have exposed) secrets from the player.

By using secrets exposed from one or more compromised players, say $D_1 \cdots D_n$, an attacker can *synthesize* any number of illegitimate players. If such illegitimate players are discovered, it may be possible to employ traitor tracing [92] schemes to determine the original DVD players $D_1 \cdots D_n$, whose secrets were employed for constructing the illegitimate players. Thus, revoking $D_1 \cdots D_n$ will simultaneously revoke *all* such illegitimate DVD players in addition to the $D_1 \cdots D_n$.

8.5.1.1 Revocation in $G = N$ Stateless Models

Assume that a month after such a system is deployed, n_1 devices have been identified as compromised. For all content distributed from this point onwards, a new group secret K_{N_1} is chosen and conveyed to all devices except the n_1 devices. At the end of the second month, say n_2 more compromised devices are identified. Now a new group secret K_{N_2} is chosen and conveyed to all but $n_1 + n_2$ devices (in all DVDs pressed after this point; revoked devices can still play older DVDs).

8.5.1.2 $G = N$ Stateful Models

In $G = N$ *stateful* models, the DVD players can remember (store) changing group secrets. In this case, it may appear at first sight that revocations can be performed in batches. For example, all devices share a secret K_{N_0} initially. At the end of the first month, a broadcast revokes n_1 devices by providing a secret K'_{N_1} to all other devices. Thus, the new group secret shared (and stored) by all $N - n_1$ legitimate devices is $K_{N_1} = K_{N_0} \oplus K'_{N_1}$. At the end of the second month a revocation broadcast revokes n_2 devices by broadcasting K'_{N_2} that the n_2 devices cannot decrypt. The group secret after the second revocation is then $K_{N_2} = K_{N_1} \oplus K'_{N_2}$.

Note that while the n_1 devices revoked in the first batch can still gain access to K'_{N_2}, they cannot gain access to the new group secret $K_{N_2} = K_{N_1} \oplus K'_{N_2} = K_{N_0} \oplus K'_{N_1} \oplus K'_{N_2}$ as they do not have access to K'_{N_1}. Unfortunately, a pirate with access to secrets from one device in the first batch and one device in the second batch can still gain access to the new group secret K_{N_2}. Thus, *if* the purpose of revocation is for excluding devices suspected of key compromises, revocation should not be performed in batches.

8.5.2 $N >> G$ Models

In many application scenarios, where BE can be performed by multiple sources, the network size N can be substantially larger than the group size G. Consider a scenario where a maximum of $N = 2^{32}$ (about 4 billion) DRM enabled set-top boxes (STB) could be deployed for playing protected video content. Every end user owns one such STB. A content distributor D may have $G << N$ paying *subscribers* (say $G = 2^{20}$, or a million). From the perspective of the distributor, the group size is a million. Each member of the group (or the STB's belonging to the G users) may be provided with a secret K_{G_0} as part of the subscription process. Later, when the distributor desires to cancel the membership of r (say 1000) of his G subscribers, the distributor can broadcast a new secret K_{G_1} to the $G - r$ continuing subscribers, that explicitly revokes r subscribers.

8.5.2.1 Revocation by Sources Other than the Key Distribution Center (KDC)

Thus, the new secret that is broadcast, is protected only from the r explicitly revoked subscribers. However, Both the $G - r$ continuing subscribers and the $N - G$ *non-members* can gain access to the secret K_{G_1}. To prevent any of the $N - G$ nonmembers from gaining access to the broadcast secret the entire broadcast may be encrypted with the group secret K_{G_0}. Nevertheless, it still does not prevent a revoked user from colluding with one of the $N - G$ users outside the group G to determine the secret K_{G_1}.

In other words, ideally $N - (G - r) \approx N$ users/STBs will have to be revoked, which is obviously impractical (it is far more efficient to unicast the broadcast secret independently to each of the $G - r$ nodes when $N >> G$). However, mandating that all $N - (G - r)$ nodes be revoked by D, while impractical, is also *unnecessary*.

In the $N = G$ scenario, *and* for revocation broadcasts by the KDC for $N >> G$ models, revocation will occur when secrets are (or suspected of being) compromised. However, the purpose of revocation by external sources in $N >> G$ models is to control access to group secrets to paying customers. *A revoked user is not necessarily more malicious than a user who is not revoked, or some user outside the group.* Given the fact that it is impractical to revoke all $N - (G - r)$ devices in any case, mandating that revocation broadcasts by distributors like D should not be batched, does not help much.

8.5.2.2 Revocation by Key Distribution Center (KDC)

However, revocation by KDC in $N >> G$ models will still be for the same purpose as $N = G$ models—ejecting devices that are suspected of compromise of secrets. While ideally, revocation broadcasts by the KDC should be able to support unlimited r, in practice this is not an essential requirement if devices taking part in the deployment are *not stateless*.

Note that stateless devices are not well suited for $N >> G$ scenarios in any case, as external sources will need to provide dynamic group secrets to members of the group. Consider a scenario where broadcasts by KDC supports batch sizes upto $r_{max} = 100,000$. Arguably, irrespective of the network size N, a scenario where such a large number of devices are suspected of being compromised, is a crying need for *renewal* of secrets. The devices revoked by the KDC will not be allowed to take part in renewal.

In other words, for systems that are *not* stateless, r_{max} is just the number of devices that *trigger renewal* of the system. While we would still like r_{max} to be high (as the process of renewal may be expensive), it is sufficient if r_{max} is "high enough." In other words, while the BE by KDC *should* support large r, it does *not* have to cater for *unbounded* r.

8.5.3 Batch Sizes for External Sources

For revocation by sources other than the KDC, where revocation *can* be performed in batches, at first sight it might seem that we could simply employ a scheme optimized to revoke *one* device in each batch, in that case *any* number of devices can be revoked efficiently. However, there are two very important reasons as to why this is not a good approach:

1. Schemes optimized for low r will employ $a = k/P$ very close[4] to one (or almost every node has almost every KPS secret), and are thus less secure.
2. the efficiency increases (or n_e/r reduces) for schemes optimized for larger r (small ξ).

8.5.3.1 Resistance to Synthesis Attacks

By compromising secrets from a certain number of devices, an attacker can determine a large fraction of the secrets[5] of the system.

One measure of the security of any KDS is their resilience to "synthesis attacks." More specifically, if an attacker needs to compromise secrets from n_s devices to expose *all* secrets of a fraction p of the devices (or pN devices that are not part of the n_s compromised devices), the resistance of the KDS to synthesis attacks is $p(n_s)$. For RPS with $\xi = 1/128$ and $k = 200$, by compromising all secrets from $n_s \approx 340$ devices, an attacker can synthesize one in a million devices (or $p(340) \approx 10^{-6}$. HARPS performs significantly better under this metric. For $\xi = 1/128$ and $k = 200, L = 64$, for HARPS $p(1650) \approx 10^{-6}$. On the other hand, if $\xi = 200/206$ (RPS optimized for batch size of 1), even by compromising secrets of one node (or $n_s = 1$), the attacker has access to all secrets of one in every two thousand devices (or $p(1) = 1/2000$).

[4] For example, for $G = 2^{30}$, and a batch size of one, with a limit of 200 keys assigned to each device, the best choice of parameters for RPS is $P = 206, k = 200$, and $P = 201, k = 200, L = 64$ for HARPS.

[5] Even while each device is provided with a unique secret (which will be used under conditions of outage), note that such secrets are meant to be used rarely.

8.5.3.2 Bandwidth Efficiency

In practical application scenarios, it is only the efficiency for large batch sizes that really matters. If a PKPS-BE scheme that is optimized for a batch size of 100 is used for revoking two devices, the overhead may be the same as the case of revoking 100 devices. The fact that the overhead is say 20 KB instead of 200 bytes may not, however, be a serious limiting factor. However, we would certainly desire that the overhead for revoking say 100,000 devices is not prohibitively high.

Let us consider two scenarios (1) HARPS with $k = 200$ optimized for batch size of one, and (2) HARPS optimized for $r >> 1$, for revoking r nodes. In the first case, the r independent broadcasts (even though they can actually be sent together) revoke one device each. In other words, each privileged device will receive r secrets, while the revoked devices will receive only $r - 1$ secrets. The final group secret then is derived from all r secrets, thus shielding the secret from the r revoked devices. However, in this case, the overall outage probability for the privileged devices increases, as outage can happen even if *one* of the r secrets "evade" a privileged device.

Thus, instead of aiming for an outage probability of $p_o = G^{-1}$, with batched revocation (with batch size of one) our target is to ensure outage probability less than p'_o, for each batch, where $(1 - p'_o)^r \approx G^{-1}$, or $p'_o \approx (rG)^{-1}$) instead. In other words, effectively the scenario is equivalent to increasing the group size G by a factor r. We already know that PKPS can take advantage of reduced group sizes G to improve their efficiency (irrespective of N). Obviously, the effective increase in group size for batched approaches will make them even less efficient.

The table below compares the achievable n_e/r for two schemes—one with $a = 200/201 \approx 1, L = 64$ for $r = 1$, and the other with $a \approx 1/128, L = 64$ for $r = 500$, for three different group sizes G (a billion, million, and thousand). For the batched scheme with $a \approx 1$, the table also indicates the reduction in efficiency due to the need to cater for reduced outage probability. In other words, row 2 in the table is n_e for revoking one device *without* taking the need to reduce outage probability into account. Row 3 (labeled $a \approx 1$*), however, takes this into account.

	$G = 2^{30}$	$G = 2^{20}$	$G = 2^{10}$
$a = 1/128$	5.39	2.71	0.62
$a \approx 1$	5.8	3.81	2.02
$a \approx 1$*	7.68	5.58	3.62

8.6 Application of Probabilistic Key Predistribution Scheme Broadcast Encryption (PKPS BE) in Publish–Subscribe Systems

Users of a publish–subscribe system [90] can assume the role of a publisher (of content), or subscriber, or both. Publishers regulate access to their content by issuing a group secret to their subscribers.

In a practical publish subscribe system, every user may employ a smart card, which protects:

1. the core predistributed secrets used for disseminating group secrets, and
2. the group secrets themselves, from the end users.

Content published by publishers may be encrypted directly or indirectly with the group secret. For example, a content encryption secret K_C for a specific content may be encrypted using the group secret K_G, and distributed with the content.

In all scenarios, the core secrets and group secrets have to be protected from the subscribers—only their smart cards will have access to the secrets. In some application scenarios, even the content encryption secret K_C may have to be protected. For example, the secret K_C will be handed over only to a trusted DRM enabled device.

The role of the key distribution center for a publish–subscribe systems is to identify compromised smart cards and revoke such smart cards from the deployment, by providing all nonrevoked smart cards with a time varying universal secret U_i. Apart from encrypting session secrets (or content encryption secrets) with group secrets, all messages will also be encrypted using the secret U_i to ensure that revoked devices cannot take part in the deployment. When the number of revoked devices crosses a threshold, the KDC could renew the predistributed secrets and provide new secrets to every nonrevoked smart card in the system.

The publishers on the other hand do not have to concern themselves with the possibility of compromised devices. Furthermore, the system should also support mutual authentication of a publisher A and potential subscriber B (by facilitating discovery of a secret K_{AB}), to facilitate initial dissemination of the publishers (A) group secret to the newly inducted subscriber B. Thereafter, revoking privileges of node B (once B cancels his subscription) can be realized by employing BE.

In addition, the system also needs to cater for authentication of broadcasts by the publisher—both content and revocation messages.

8.6.1 Desirable Features

A very desirable feature in such large scale application scenarios is practically unlimited scalability.

1. *Identity-Based Deployment:* Even while the total number of users (say N') in the system may never exceed a few billions, it is still desirable to assign large IDs to each user (for example 160-bit IDs) to facilitate ID-based naming.
2. *Dynamic Group Sizes:* The group sizes can have a wide range. Even very large distributors may have only millions of subscribers. Thus, it is desirable that the predistributed secrets can be leveraged for efficient broadcast for any number of revoked nodes, for any network size.

   Privacy: As revocation broadcasts are public, users may wish to maintain information regarding memberships private. Thus, revocation broadcasts should not explicitly indicate the identities of revoked nodes.

8.6.2 PKPS-BE vs. T-BE for Pub–Sub Systems

Some of the considerations for the choice of an appropriate BE scheme include the group size of dynamic groups, scalability, privacy of participants, and storage required for secrets.

8.6.2.1 Dynamic Group Size

For the use of T-BE schemes for scenarios where $N >> G$, one possibility (though not very desirable) is to let the KDC control memberships of every group within the network (if we desire to eliminate the use of asymmetric primitives). Even in this case, irrespective of the group size G, the efficiency of revocation will still depend on the network size N. For example, for a T-BE scheme that caters for a network size of 2^{30} (a billion), where say $G = 2^{10}$ (a thousand) of the $N = 2^{30}$ devices belong to a group, to revoke r of the G devices, the number of encryptions required is still $30 \times r$ - or $\log_2(N) = 30$ encryptions per revoked device. Unlike T-BE schemes, PKPS-BE can take advantage of reduced group sizes to increase their efficiency. Recall that for small group sizes (say 1000), the number of encryptions required per revoked device can be substantially smaller than one.

8.6.2.2 Scalability

Even if we ignore the primary disadvantage of T-BE schemes for multisource BE, the need for asymmetric cryptographic primitives, T-BE does not scale as well as PKPSs, due to the storage complexity to be borne by external sources. For multisource T-BE schemes, sources other than the KDC need to store $\mathbb{O}(N)$ public values (more specifically $2N - 1$ public values corresponding to the $2N - 1$ nodes in the binary tree). This may not be a problem in practice even for network sizes of billions. After all, storing billions of public keys will call for a (mere) few hundreds of GBs of storage, a trivial requirement for any content distributor. Nevertheless, calling for storage proportional to N certainly certainly cramps the scalability of the network (for example, making the use of identity based approaches impractical). Further, for T-BE schemes we have to take future scalability into account before we decide N.

For PKPSs, the storage required for external sources[6] is $PL \propto r \log(N')$, where r is the maximum number of nodes that can be revoked in a single batch. As far as the KDC is concerned, revocation broadcasts by the KDC should reach all nodes. If the system actually has N' nodes at some point in time, the KDC has to ensure outage probability $p_o \approx 1/N'$ to convey the secret to every node with a high probability. Similarly, the publishers (with group size G) only have to cater for $p_o \approx 1/G'$. Thus, irrespective of the theoretical maximum network size N (for example, $N \approx 2^{160}$ if identities are 160-bit long) the KDS just has to cater only for the maximum number of users currently in the system.

8.6.2.3 Privacy

In T-BE schemes, the revoked devices have to be explicitly specified in the revocation broadcast. In PKPS-BE, we only need to specify the indexes (and hash depths) of the n_e secrets used in the broadcast (to encrypt the broadcast secret). Apart from protecting privacy of group membership information, we can also afford to use large IDs *without* adding to the bandwidth overheads.

8.6.2.4 Storage for Secrets

The advantages of PKPS-BE over schemes are achieved primarily by increasing the number of secrets assigned to every device (smart card). Even for network size of 2^{60}, T-BE schemes like the complete subtree scheme require only storage for 60 secrets per node. However, a PKPS scheme requiring n parallel deployments defined by parameters $(P_i, k_i, L_i), 1 \leq i \leq n$ calls for $\sum_{i=1}^{n} k_i$ secrets to be stored in each device (for example 800 secrets if $n = 4$ and each deployment has 200 secrets).

However, in any scenario calling for protection of multiple secrets, a very common approach (dating back to at least 1978 [96]) is to employ a single host master secret to encrypt all other secrets. The smart card A belonging to a user Alice could store just one master secret M_A, and all other $\sum_{i=1}^{n} k_i$ decryption secrets assigned to A could be encrypted using M_A and stored outside the smart card, for example, in the hard-disk of Alice's desktop/laptop (or even a SD card that can be plugged into a PDA). Obviously, the storage complexity for the decryption secrets is not an issue. In other words, we can increase the number of decryption secrets substantially to facilitate bandwidth efficient revocation (for example increasing k by a factor $50 = e^{a-1}$ to reduce bandwidth requirement by a factor five.

[6] typically of the order of tens or hundreds of MB.

It is pertinent to point out that tree-based schemes substantially more efficient ($n_e/r \approx 1.25$) [92] have also been proposed which call for a storage complexity of $\log_2(N)^2/2$ keys (about 512 keys per device for $N = 2^{32}$, and 1800 keys for $N = 2^{60}$). However, such schemes extend less readily to BE by external sources.

8.6.3 Pub–Sub Operation

A pub–sub system employing PKPS-BE will consist of a KDC who chooses a set of n HARPS systems and public functions $F_i()$ and $f_i()$. Every participant in the system employs a smart card, which protects the $\sum_{i=1}^{n} k_i$ PKPS decryption secrets and group secrets on behalf of the participant. The smart card is assumed to be plugged into a general purpose computer, for example, desktop, laptop, or a mobile phone. Each smart card, associated with a user is assigned a 160-bit identity, based on the identity of the owner.

All users who desire to be publishers can be provided with a maximum of $\sum_{i=1}^{n} P_i L_i$ tickets. However, operations involving tickets are *not* performed by the smart card as the tickets issued to Alice need not be protected from Alice. The secrets that are protected (hidden from the owner of the smart card) by the smart cards are (1) predistributed secrets used for decrypting the broadcast secret, (2) the group secrets, and (3) the broadcast secrets (which may be used to modify the group secrets).

Apart from providing encryption secrets to the publishers, the tasks performed by the KDC include

1. proactive measure to identify compromised nodes, and revoke them from the system, and
2. renew secrets of the system periodically when a substantial number of nodes have been revoked.

8.6.3.1 Establishing Group Secrets

Consider a scenario where a publisher Alice, with smart card A inducts a member Bob with smart card B. Apart from catering for BE, HARPS also facilitates establishment of shared secrets between any publisher (or any user with PL tickets) with any user with decryption secrets. For example, for using the system with $P = 204,800, k = 200, L = 64$ for mutual authentication, Alice determines the k indexes and the hash depths of the decryption secrets $\mathbb{S}_B$. Using k of PL encryption secrets $\mathfrak{S}^A$, Alice can encrypt a session secret K_S that only Bob's smart card can decrypt [57].

The publisher Alice chooses a group secret K_{GA_0} and supplies the group secret to her smart card A, which encrypts the secret K_{GA_0} with the universal secret U_i. The secret $U_i(K_{GA_0})$ is now provided to the newly inducted member, over a channel secured using the established session secret K_S. Note that smart cards revoked by the KDC (that do not have access to the secret U_i) cannot decrypt the group secret, and thus cannot become members of *any* group. By using the secret shared between the publisher and the subscriber to authenticate a commitment, hash-chain-based approaches [12,13], could be used for authenticating subsequent revocation broadcasts by the publisher.

8.6.3.2 Revoking Users from Publishers Groups

For revoking a set of r users who have have access to the current group secret K_{GA_j} of the publisher Alice, Alice chooses a new group secret $K_{GA_{j+1}}$, and encrypts it with the current universal revocation secret U_i (by providing $K_{GA_{j+1}}$ to her smart card A).

The secret $K_b = U_i(K_{GA_{j+1}})$ is broadcast by encrypting it with n_e encryption secrets. Note that the primary complexity associated with PKPS-BE lies in the determining the indexes and hash depths of the n_e secrets to use, for revoking a specific set of r nodes (by evaluating the public functions). However, for purposes of creating the revocation broadcast, the only operation performed by the smart-card is encrypting the secret $K_{GA_{j+1}}$ to provide the publisher with the secret $U_i(K_{GA_{j+1}})$. All other operations are performed by Alice's desktop computer.

8.6.3.3 Decryption of Group Secrets

Depending on the nature of content distributed by the publisher, the revocation broadcast can be posted in the Website of the publisher or distributed with the content. With PKPS-BE, the distributor only indicates the indexes and the hash depths of the keys used.

At the other end, a subscriber Bob, accesses the broadcast with n_e encrypted versions of the broadcast secret, and a header indicating the indexes of the PKPS secrets. Bob's computer can evaluate public functions $F(B)$ (and $f()$) to determine an index of the secret that can be used by his smart card B to decrypt the broadcast secret K_b.

For instance, assume that (1) the broadcast includes $K_e(K_b)$ where $K_e = h(K_{46}^{63} \parallel A)$, (2) $F(B)$ includes the index 46, and (3) $f(B, 46) = 42 < 63$. In other words, one of B's decryption secrets $\mathbb{S}_B$ is ${}^{42}K_{46}$, which is stored in Bob's desktop computer as $M_B(K_{46}^{42})$ (or encrypted with the master secret M_B stored inside the the smart card B). In such a scenario, Bob provides his smart card with the values

$$[M_B(K_{46}^{42}) \parallel K_e(K_b) \parallel 21 \parallel A]. \tag{8.26}$$

The sequence of operations to be performed by the smart card to gain access to the group secret are as follows:

1. perform one decryption to determine K_{46}^{42}, and $21 = 63 - 42$ repeated hashes to evaluate K_{46}^{63}.
2. compute $K_e = h(K_{46}^{63} \parallel A)$
3. decrypt $K_e(K_b)$ to determine K_b
4. decrypt K_b with U_i to determine new group secret $K_{GA_{j+1}}$

5. encrypt $K_{GA_{j+1}}$ with M_B, and
6. hand $M_B(K_{GA_{j+1}})$ back to Bob's desktop for storage

Note that the smart card stores only the master secret M_B and possibly the current universal secret U_i (controlled by revocation broadcasts by KDC).

Any content distributed by the publisher Alice (with smart card A), meant exclusively for her subscribers, is encrypted with a secret K_C. The secret K_C is then encrypted with the group secret $K_{GA_{j+1}}$, and distributed along with the encrypted content. Thus, Bob provides his smart card B with $M_B(K_{GA_{j+1}})$ and $K_{GA_{j+1}}(K_C)$. The smart card performs two decryption operations to evaluate K_C. Depending on the nature of the specific application scenario and type of content, the secret K_C may be handed over to Bob, or handed over to a trusted DRM-enabled device.

Chapter 9
Authenticated Data Structures

Authenticated data structures (ADS) [97–103] are useful constructions in scenarios where databases are maintained by an untrusted database server. In typical applications employing ADSes, clients who query a server trust only the originator/provider of the data, and not the middle-man—the untrusted server maintaining the database. Specifically, the clients trust only a compact ADS digest duly authenticated by the source of the data.

An ADS is characterized by a construction algorithm $f_c()$ and a verification algorithm $f_v()$, where both $f_c()$ and $f_v()$ typically involve simple sequences of pseudo random function (PRF) (for example, $h()$) operations.

Using an ADS, a database of records $\mathbf{D}$ may be summarized as a succinct value

$$r = f_c(\mathbf{D}), \tag{9.1}$$

such that for any record $R \in \mathbf{D}$ it is possible to determine a *verification object* (VO) $\mathbf{v}_R$ satisfying

$$r = f_v(R, \mathbf{v}_R). \tag{9.2}$$

The provider A of a set of records $\mathcal{D}$ computes a static summary $r = f_c(\mathcal{D})$. The database records $\mathcal{D}$ are hosted by an untrusted repository/server. Along with a response R to a query by a client, the server is expected to send a VO $\mathbf{v}_R$ satisfying $r = f_v(R, \mathbf{v}_R)$, and the signature of the originator A for the summary r. The client is now convinced that the response R is identical to the one that would have been provided by the originator A, *if* the client had directly queried A—thereby, rendering the middleman transparent from a security perspective.

Some limitations of the above approach, where ADSes are constructed by the data source and verified by clients, render it unsuitable for several practical services with any of the following characteristics:

1. The legitimate response to a query is a function of records submitted by *multiple independent* providers. For example, in a remote file storage system, the database maintained by a server can be seen as hashes of different versions of files. A file hash for a file may be provided by any possible multiple entities who may have write access to the file. As another example, the database may be required to store

M. Ramkumar, *Symmetric Cryptographic Protocols,* DOI 10.1007/978-3-319-07584-6_9,

quotes for prices for different widgets from different providers. When a querier queries the database for the price of a widget, the database may be expected to provide the least price. Such a database may even be a routing database where different routers provide their best path to different destinations.

2. Data provided by originators are dynamic and/or provided in an asynchronous manner. Assume that the summary r is signed by the source indicating an expiry time of t. If the summary needs to be modified at any time before t (for example, if there is a need to update any record in the database before time t), then the source can provide a fresh signature for the updated summary. However, at a time $t' < t$, there is nothing that prevents the untrusted middleman from replaying the old record consistent with the previous (but still valid till time t) signature.
3. In providing the VO to the clients, it is desired that servers should not need to reveal any information that is not explicitly queried. For example, in a scenario, where the database contains DNS records for a zone, NSEC records provided to demonstrate nonexistence of the queried names can reveal unsolicited information.

All such inadequacies can be overcome if ADSs are constructed and verified by a *trusted third party* (TTP). As typical ADS construction and verification algorithms involve simple sequences of cryptographic hashing operations, the TTP can be a low complexity trustworthy module **T**.

Irrespective of whether ADSs are verified by the querier, or a TTP, the ADS-based protocols can be seen as an interaction between two parties

1. an untrusted prover who maintains the database, and identifies the appropriate VOs for any record, and
2. the verifier who has access only to the summary.

In the former scenario (where the verifier is a client), the summary is signed by the originator of the data. In the latter scenario (where the verifier is module **T**), the summary r is stored inside the module **T**.

In the rest of this chapter, our focus is on ADS models where verification is performed by a resource-limited trusted module.

9.1 Merkle Tree as an ADS

The Merkle hash tree discussed in Sect. 9.1.2 is one example of an ADS. Recall that the Merkle hash tree can be used to obtain a commitment for a set of N leaf nodes $v_0 \cdots v_{N-1}$ (the root of the tree is the commitment to all leaf nodes). Each leaf node can in turn be seen as a commitment to a record, in a database with N records. For a leaf-node v_i at position i (where $0 \leq i \leq N-1$), a set of L complementary nodes $\mathbf{c}_i = \{c_0 \cdots c_{L-1}\}$, is a VO satisfying

$$r = f_{bt}(v_i, i, \mathbf{c}_i), \tag{9.3}$$

where r is the root of the tree, and the function $f_{bt}()$ was outlined in Sect. 2.1.2.

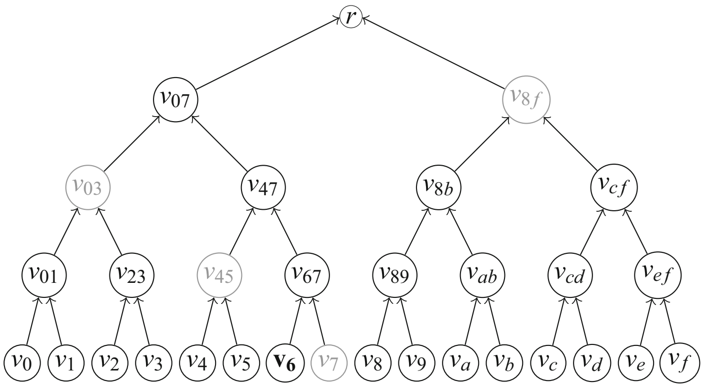


Fig. 9.1 Merkle hash tree

If $v_i = h(R_i)$ (or v_i is the commitment to the ith record R_i in the database), then

$$r = f_{bt}(v_i, i, \mathbf{c}_i) = f_v(R = h(v_i), \mathbf{v}_R = \mathbf{c}_i). \tag{9.4}$$

As long as the PRF $h()$ used for realizing the ADS construction and verification algorithms are guaranteed to be preimage resistant, it is infeasible to determine

1. $R' \neq R$ satisfying $r = f_v(R', \mathbf{v}_R)$; or
2. a VO $\mathbf{v}'_R \neq \mathbf{v}_R$ satisfying $r = f_v(R, \mathbf{v}'_R)$.

In a database represented as a Merkle hash tree, each leaf can be independently verified against the root r (using $f_v()$). Each leaf can also be independently modified. For example, if there is a legitimate requirement to modify record R to R', we need to update $v_i = h(R_i)$ to $v'_i = h(R'_i)$. A modification to v_i will require a corresponding modification to the root d. Specifically, the old root r and the new root r' are related as

$$\begin{aligned} r =& f_{bt}(v_i, i, \mathbf{c}_i) \\ r' =& f_{bt}(v'_i, i, \mathbf{c}_i). \end{aligned} \tag{9.5}$$

Recall that the VO $\mathbf{c}_i$ can be seen as a set of commitments to all leaf nodes *except* v_i. As the same VO, used to verify the integrity of the old v_i against root r, is also used to compute the new root, the VOs themselves are unaffected due to the change. As the VOs are commitments to all other leaves, it is guaranteed that they are also not affected when the root is changed to r' (Fig. 9.1).

9.1.1 Merkle Tree Protocols

Protocols employing Merkle hash trees can be seen as an interaction between two parties

1. an untrusted *prover* **U** and
2. a trusted *verifier* **T**.

Typically the prover **U** has plentiful storage resources. **U** maintains all records and nodes (or the entire Merkle tree). The verifier **T** is, however, resource limited, and maintains a single value—the root of the tree. As a practical example, the prover may be database server. The verifier may be a trusted module that is entrusted with the integrity of the root r.

The verification protocol can be represented as

$$\begin{array}{ll} \mathbf{U} \rightarrow \mathbf{T}: & v, i, \mathbf{v} \\ \mathbf{T}: & \text{IF } (r = f_{bt}(v, i, \mathbf{v})) \text{ VERIFICATION SUCCESSFUL} \end{array} \tag{9.6}$$

On successful verification, the verifier is assured that v is an authentic leaf-node of a Merkle tree with root r.

To update the ith leaf-node v, the prover provides the current value v, the position i, the VO for the leaf node, the new value v', and a justification J as to why the update is warranted. Obviously the nature of the justification J and the mechanism for verifying the justification will be closely tied to the nature of the application. In scenarios involving multiple independent providers, the application specific justification may indicate which sources are permitted to modify which records, and under what specific circumstances.

The verifier verifies that the current leaf-node is indeed consistent with root r before updating v to v'. Once the update is successful, the prover may now update the tree to modify all direct ancestors of v (as a change in v will affect all ancestors of v including the root).

$$\begin{array}{ll} \mathbf{U} \rightarrow \mathbf{T}: & v, v', i, \mathbf{v}, J \\ \mathbf{T}: & \text{IF } (J \text{ not satisfactory) RETURN} \\ & \text{ELSE IF } r = f_{bt}(v, i, \mathbf{v}) \\ & \quad \text{RETURN } r = f_{bt}(v', i, \mathbf{v}). \\ \mathbf{U}: & \text{Update } v \rightarrow v'. \text{ Update all ancestors of } v \end{array} \tag{9.7}$$

The Merkle tree by itself is not associated with a protocol for insertion or deletion of records. In other words, in practical applications utilizing Merkle tree protocols, the verifier—for example, a trusted module **T**—is assumed to be initialized (by an entity trusted by the prover and the verifier) with a root r_0 corresponding to the initial snapshot of the database with N records.

From this point onwards, the prover **U** can utilize the update protocol to update records. The verification protocol is typically used in scenarios where the prover is required to advertise a record R from the database to a third party. For example, the third party could be a client that has queried the database for a specific record. As the third party may only trust the verifier **T** (and not the prover), the prover has two options:

1. Request the verifier to certify the current value of the root r; the prover can then submit the record R, the VO for R, along with the certificate authenticated by the verifier **T** to the third party.
2. Request the verifier to verify and certify that the "record R is indeed consistent with the current root." This certificate can then be sent to the third party along with the record R.

The second option is useful in scenarios where we desire to minimize the bandwidth overhead for the exchange between the third party and the prover, or in scenarios where the database server does not desire to leak any additional information regarding the database. Note that if the first option is used, the client will get to know "other information" regarding the database like the (approximate) total number of records (from the size of the VO) and the position i of the record in the database.

9.2 Ordered Merkle Tree

While the fact that each record is independent (each record is verified/updated independent of other records) certainly eases the task of the resource-limited verifier **T**, it does not, unfortunately, permit the verifier to infer some useful holistic properties regarding the database of records. Furthermore, that records cannot be inserted/deleted cramps the utility of such an approach for scenarios involving dynamic number of records.

If the Merkle tree is enhanced to include a well-defined protocol for insertion and deletion of the leaves, and if the protocol for insertion/deletion guarantees that all records will be ordered in a specific manner, then it is possible for the verifier **T** to readily infer some useful properties regarding the database.

For example, if every record is associated with a unique index (which may be different from the actual position i of a leaf in the database), and if records are constrained to be ordered by the index, then the verifier **T** can readily verify the nonexistence of records with a specific index. If a record for index F and a record for index X can be verified to be adjacent (for example, positions i and $i + 1$), **T** can conclude the nonexistence of all records with indexes between F and X. Such a conclusion is not possible if a plain Merkle tree is used, as the verifier will need to verify *all* the leaves before concluding that the desired index does not exist. As another example, if all records are constrained to be ordered by another field (for example, cost) in each record, then it becomes trivial for the verifier to readily identify the record with the least or highest cost.

The ordered Merkle tree (OMT) is such an extension of the Merkle tree.

9.2.1 OMT Leaves

The main differences between an OMT and a plain Merkle tree are as follows:

1. The OMT specifies a rigid structure for the leaves of the tree. An OMT leaf is of the form

$$\mathbf{L} = (a, a_n, \omega_a). \tag{9.8}$$

where a is an index, a_n is the next index, and ω_a is a value associated with index a. Alternately, a could represent the beginning of an interval, and a_n the beginning of the next interval. In the latter case, ω_a is a value associated with the half-open interval[1] $[a, a_n)$. For all but one leaf in the tree, $a_n > a$. For the leaf with $(a = x > a_n = y)$ indicates that $a = x$ is the highest index—the next index $y = a_n$ is lowest index in the tree.
2. The OMT specifies a protocol for insertion and deletion of leaves.
3. The OMT has a special interpretation of the value 0.

Corresponding to an OMT leaf (a, a_n, ω_a) is a leaf node computed as

$$\begin{aligned} v_a =& H_L(a, a_n, \omega_a) \\ =& \begin{cases} 0 & a = 0, \\ h(a \parallel a_n \parallel \omega_a) & a \neq 0. \end{cases} \end{aligned} \tag{9.9}$$

The first field a is constrained to be unique for every record in the tree. Together the first and second fields (a, a_n) indicate that no record exists in the tree for which the first field is *circularly enclosed* by (a, a_n). The tuple (a, a_n) is a circular enclosure for b only if

$$(a < b < a_n) \text{ OR } (a_n < a < b) \text{ OR } (b < a_n < a) \tag{9.10}$$

For example, $(2, 42)$ circularly encloses 4; 48 is enclosed by $(46, 23)$; 21 is enclosed by $(43, 26)$; 21 is *not* enclosed by $(26, 43)$.

In the rest of this chapter, we shall represent as $f_{encl}((a, a_n), b)$, the condition required for (a, a_n) to be circular enclosure for b. $f_{encl}((a, a_n), b) =$ TRUE implies (a, a_n) does enclose b.

The third field ω_a is a value "associated" with index a (or the interval $[a, a_n)$). It is also possible that ω_a is itself the root of an OMT nested in the leaf (a, a_n, ω_a) of the "outer" OMT.

An OMT can also be interpreted as a circular list, where each element in the list is bound to a value. An example of the leaves of an OMT (with four leaves) is

$$[(2, 4, \omega_2), (4, 98, \omega_4), (98, 101, \omega_{98}), (101, 2, \omega_{101})]. \tag{9.11}$$

All leaves together can be interpreted as a representation of a circular list 2→4→98→101→2, where each element in the list is associated with a value—2 is associated with ω_2, 98 with ω_{98}, and so on.

[1] An interval that includes all x satisfying $a \leq x < a_n$.

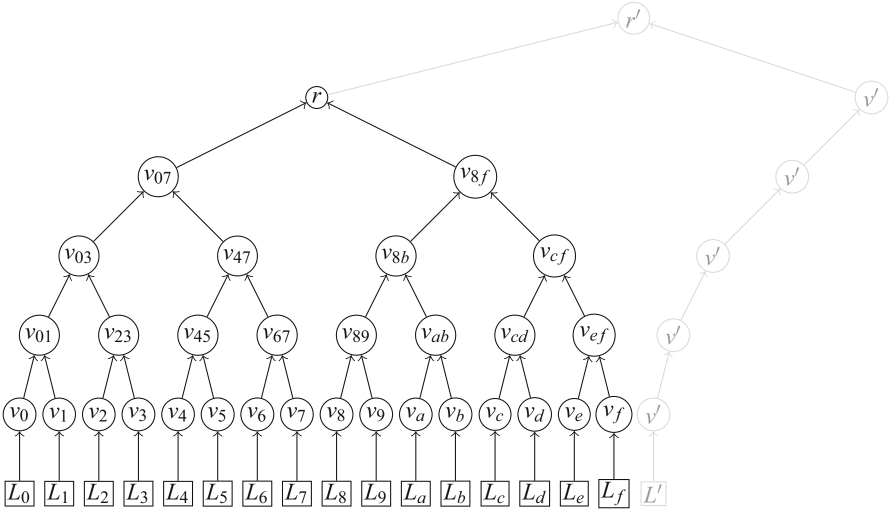


Fig. 9.2 Ordered Merkle tree

9.2.2 OMT Nodes

In a plain merkle tree, two siblings u and v (where u is the left sibling and v is the right sibling), and their immediate parent p are related as

$$p = h(u \parallel v). \tag{9.12}$$

In an OMT, the relationship between a parent and its two children is defined instead as follows:

$$p = H_V(u, v) = \begin{cases} u & \text{if } v = 0 \\ v & \text{if } u = 0 \\ h(u \parallel v) & \text{if } u \neq 0, v \neq 0 \end{cases} \tag{9.13}$$

Note that the value 0 has a special significance in OMTs.

1. The parent of two nodes is the hash of the two child nodes *only if both* the children are nonzero. If any child is zero, the parent is the same as the other child. The parent of $u = v = 0$ is $p = 0$.
2. An OMT leaf with the index set to zero is an *empty* leaf, represented as Φ. The leaf hash corresponding to an empty leaf is 0. Introducing an empty leaf node (corresponding to an empty leaf) does not affect any node of the tree due to the way $H_V()$ is defined. Consequently, any OMT implicitly includes an unlimited number of empty leaves. A tree with root 0 has no nonempty leaf.

Figure 9.2 depicts an OMT with 16 leaves $L_0 \cdots L_f$. The function $H_L()$ is used to derive the leaf node corresponding to the leaf. The function $H_V()$ is used to combine two siblings to derive a parent at the next level.

Similar to the Merkle tree, a tree with $N = 2^L$ leaves has a height of L and includes

$$\sum_{i=0}^{L} 2^{L-i} = \sum_{i=0}^{L} \frac{N}{2^i} = 2N - 1 \tag{9.14}$$

internal nodes (N nodes at level 0, $N/2$ at level 1, ... a lone node at level L). However, this relationship between total number of leaves N and the total number of internal nodes $2N - 1$ does not hold in Merkle tree if N is not a power of two. On the other hand, in an OMT, this relationship holds even when N is not a power of 2. To see this, consider a scenario where a 17th node is added (represented by faint lines in Fig. 9.2). Corresponding to the 17th node L', let the leaf-node be $v' = H_L(L')$. The ancestor nodes of v' at various levels remain v' as all other nodes to the right of v' are zero. The new root is then $r' = H_V(r \parallel v')$.

Note that only two new values—v' and r' have to be maintained to account for an additional node. It can be easily seen that adding an 18th node will add two more values, the leaf-node (say) v'' corresponding to the 18th node, and the common parent $v''' = H_V(v' \parallel v'')$ which will replace v' in levels 1 to L. Thus, irrespective of whether N is a power of 2, the total number of distinct nodes that have to be maintained by the prover is always $2N - 1$.

9.2.3 Verification and Update Protocols

The OMT verification and update protocols are very similar to that of the corresponding Merkle tree protocols. To demonstrate that a leaf (a, a_n, ω_a) does indeed belong to the tree, the prover provides the following values to the verifier **T**:

1. the leaf (a, a_n, ω_a), along with its position $i = b_{L-1}b_{L-2}\cdots b_0$ (where b_i is a bit of i) in the tree, and $L = \log_2 N$;
2. a verification object (VO) consisting of L intermediate nodes, say $(x_0 \cdots x_{L-1})$.

The verifier first computes $v = H_L(a, a_n, \omega_a)$, and proceeds to compute

```
FOR $i = 0$ TO $L - 1$
    IF ($b_i = 0$)
        $v = H_v(v, x_i)$
    ELSE IF($b_i = 1$)
        $v = H_v(x_i, v)$
```
(9.15)

If the final result v after L iterations is the same as the root of the tree the verifier concludes that (a, a_n, ω_a) is indeed a part of the tree.

Henceforth, we shall represent the process of iterating through the values $\mathbf{x}$ in the VO starting with a value v as

$$r = f_{omt}(v, i, \mathbf{x}) \tag{9.16}$$

where, if the number of VOs is l, then

1. r is a root of a subtree of depth l
2. v is a leaf-node in the subtree at position i

Thus, the verification protocol can be represented as

$$\begin{array}{ll} \mathbf{U} \rightarrow \mathbf{T} : & (a, a_n, \omega_a), i, \mathbf{x} \\ \mathbf{T} : & v = f_{omt}(H_L(a, a_n, \omega_a), i, \mathbf{x}), \end{array} \tag{9.17}$$

where verification is deemed successful if $v = r$, the OMT root stored by the verifier.

In a scenario, where the prover **U** has successfully demonstrated to the verifier **T** that (for example) $(4, 98, \omega)$ is a leaf of the tree, the verifier can make the following conclusions:

1. a value ω is bound to $(4, 98)$; and
2. no record exists with index $4 < x < 98$.

On the other hand, in a scenario where the verifier is convinced that $(101, 2, \omega')$ is a leaf in the tree (note that the second field 2 is smaller than the first field 101), the verifier can conclude that

1. a value ω' is bound to $(101, 2)$; and
2. no record exists with index $x < 2$, and no record exists with index $x > 101$.

The OMT update protocol is used to modify the third field in leaf—for example, modifying (a, a_n, ω_a) to (a, a_n, ω'_a). To update a leaf, the prover will need to provide a (application specific) justification J for the update. The update protocol can be represented as:

$$\begin{array}{ll} \mathbf{U} \rightarrow \mathbf{T} : & (a, a_n, \omega_a), i, \mathbf{x}, \omega'_a, J \\ & \text{IF } (J \text{ not satisfactory}) \text{ RETURN;} \\ & \text{ELSE IF } r = f_{omt}(H_L(a, a_n, \omega_a), i, \mathbf{x}) \\ & \quad r = f_{omt}(H_L(a, a_n, \omega'_a), i, \mathbf{x}) // \text{ update root } r \end{array} \tag{9.18}$$

If the justification is acceptable, and if the current leaf belongs to an OMT with root r, then the root is updated.

9.2.4 Insertion of OMT Leaves

A leaf with index a can be inserted into an OMT only if no leaf with index a currently exists. To prove to the verifier that a does not exist currently in the tree, the prover has to demonstrate the existence of an *encloser* leaf with first two fields (b, b_n) such that $f_{encl}((b, b_n), a) = \text{TRUE}$.

After insertion, the newly inserted leaf will have the first two fields set to (a, b_n) and the encloser will be modified to (b, a). Note that insertion of a leaf with index a will modify two leaves:

1. an empty leaf is modified to (a, b_n, ω_a); and
2. the encloser (b, b_n, ω_b) is modified to (b, a, ω_b).

As we shall see in the next section when a leaf with index a is inserted, the value ω_a will be set to one of the two values:

1. if the OMT is an index ordered Merkle tree (IOMT) then $\omega_a = 0$;
2. if the OMT is a domain ordered Merkle tree (DOMT) then $\omega_a = \omega_b$ (same as the value in the encloser).

For now we shall ignore the precise differences between an IOMT and a DOMT. For reasons that will become apparent later, inserting a leaf subject to the rules mentioned above (existence of an encloser, initial value for inserted leaf) will not modify the actual database represented by the OMT. A newly inserted leaf is merely a "placeholder," possibly for an anticipated change to the database. Thus, the prover does *not* have to provide an application-dependent justification J for doing so. A justification is required only for updating the value (third field) in a leaf.

To update the root to reflect the newly inserted leaf, two leaf nodes have to be updated simultaneously:

1. a leaf node $v_1 = 0$ (corresponding to an empty leaf) should be updated to $v_1' = H_L(a, b_n, \omega_a)$ (where $\omega_a = \{0, \omega_b\}$ depending on whether the OMT is an IOMT or a DOMT).
2. a leaf node (corresponding to the encloser) should be modified from $v_2 = H_L(b, b_n, \omega_b) \rightarrow v_2' = H_L(b, a, \omega_b)$.

One exception from the general rule is for insertion of the first leaf. Before the first leaf is inserted, the root of the OMT (with no nonzero leaves) is $r = 0$. The first inserted leaf for index a will take the form $(a, a, 0)$ (both index and next index are the same), corresponding to which the root will be $H_L(a, a, 0)$. Note that (a, a) is a circular enclosure for *all* values *except* a. Following this, if a second leaf is inserted for an index b, then the resulting two leaves will be $(a, b, 0)$ and $(b, a, 0)$.

The protocol for inserting a leaf can be represented as follows:

```
$\mathbf{U} \rightarrow \mathbf{T}: a, i_a, \mathbf{x}_a, (b, b_n, \boldsymbol{\omega}_b), i_b, \mathbf{x}_b$
$\mathbf{T}:$  IF $(r = 0)$//insertion of first leaf
                 $r = H_L(a, a, 0)$;
             ELSE IF $(f_{encl}((b, b_n), a))$
                 $v_a = 0; v_b = H_L(b, b_n, \boldsymbol{\omega}_b); v_b' = H_L(b, a, \boldsymbol{\omega}_b)$;
                 IF (IOMT) $v_a' = H_L(a, b_n, 0)$;
                 ELSE IF (DOMT) $v_a' = H_L(a, b_n, \boldsymbol{\omega}_b)$;
                 IF $(r = f_{omt}(v_b, i_b, \mathbf{x}_b))$
                     IF $(f_{omt}(0, i_a, \mathbf{x}_\mathrm{a}) = f_{omt}(v_b', i_b, \mathbf{x}_b))$
                         $r = f_{omt}(v_a', i_a, \mathbf{x}_a)$// update root $r$
                                                                      (9.19)
```

If the root r is not 0 then $\mathbf{T}$ verifies that the index a to be inserted is enclosed by (b, b_n). If (b, b_n, ω_b) is a valid leaf in the tree with root r then after insertion, the leaf will be modified to (b, a, ω_b), and the newly inserted leaf will be $(a, b, 0)$ (if the OMT is an IOMT) or (a, b, ω_b) (if the OMT is a DOMT). To update two leaf nodes from $0 \rightarrow v'_a = H_L(a, b, \omega_b)$ and $v_b = H_L(b, b_n, \omega_b) \rightarrow v'_b = H_L(b, a, \omega_b)$, the verifier **T**

1. verifies that v_b is a leaf node of r
2. computes the leaf node r' that will result *if* v_b is updated to v'_b, or $r' = f_{omt}(0, i_a, \mathbf{x}_a)$
3. verifies that $\mathbf{x}_a$ is the VO for an empty leaf with root r', (or verify $r' = f_{omt}(0, i_a, \mathbf{x}_a))$
4. compute r'' necessary to modify leaf node $v_a = 0 \rightarrow v'_a$ (or $r'' = f_{omt}(v'_a, i_a, \mathbf{x}_a)$), and
5. update root to r''.

9.2.5 Reordering OMT Leaves

The physical ordering of the OMT leaves has no bearing on the interpretation of the database represented by the OMT. For example, the two OMTs below (each with four leaves)

$$\begin{aligned} &[(2,4,\omega_2),(4,98,\omega_4),(98,101,\omega_{98}),(101,2,\omega_{101})] \text{ and} \\ &[(98,101,\omega_{98}),(4,98,\omega_4),(2,4,\omega_2),(101,2,\omega_{101})] \end{aligned} \tag{9.20}$$

represent the same database of records. Note that the second OMT can be obtained by swapping the first and third leaves of the first OMT.

Even while swapping the position of two leaves does not modify the database represented by the OMT, it changes the OMT root. In a scenario where the prover **U** desires to swap two leaves[2] of the OMT, the protocol to be adopted is as follows:

$$\begin{aligned} &\mathbf{U} \rightarrow \mathbf{T} : v_a, i_a, \mathbf{x}_a, v_b, i_b, \mathbf{x}_b \\ &\mathbf{T} : \quad v'_b = v_a; v'_a = v_b; \\ &\qquad \text{IF } (r = f_{omt}(v_b, i_b, \mathbf{x}_b)) \\ &\qquad\quad \text{IF } (f_{omt}(v_a, i_a, \mathbf{x}_a) = f_{omt}(v'_b, i_b, \mathbf{x}_b)) \\ &\qquad\qquad r = \mathrm{f}_{\mathrm{omt}}(v'_a, i_a, \mathbf{x}_a) // \text{update root } r \end{aligned} \tag{9.21}$$

To swap two leaf nodes v_a and v_b (which can be seen as simultaneously updating $v_a \rightarrow v_b$ and $v_b \rightarrow v_a$), the protocol is similar to the insertion protocol. The verifier **T**

1. verifies that v_b is indeed a leaf node of a tree with r at position i_b
2. compute r' necessary to update $v_b \rightarrow v_a$
3. verifies that $\mathbf{x}_a$ is also a VO for v_a at position i_a for a tree with root r'
4. compute $r' \rightarrow r''$ to modify v_a at position i_a to v_b, and
5. sets the root to r''.

[2] The practical utility of the prover's ability to swap leaves is explained later in this chapter.

The protocol for swapping leaves can actually be used for swapping even positions of two subtrees (and thus all leaves under the two subtrees) within an OMT. For example, to swap the position of all the four leaves $L_0 \cdots L_3$ (all of which have a common ancestor v_{03}) with the positions of the four leaves $L_8 \cdots L_b$ (with a common ancestor v_{8b}, it is sufficient to swap v_{03} and v_{8b}. More specifically, in a subtree with depth 2 rooted at r, as the leaf nodes are v_{03}, v_{47}, v_{8b}, and v_{bf}, the leaf-node v_{03} has an index 0, and the leaf node v_{8b} has an index 2. The VO for v_{03} is $\{v_{47}, v_{8f}\}$. The VO for v_{8b} is $\{v_{cf}, v_{07}\}$.

9.2.6 Index Ordered Merkle Tree

In an index ordered Merkle tree (IOMT) leaf (a, a_n, ω_a), the first field a is interpreted as the index of a record. The second field is the next higher index in the tree. The third field ω_a provides some information regarding index a (for example, hash of the record for index a). A leaf with $\omega_a = 0$ is a placeholder indicating that "no information is available" regarding index a. Some examples of databases that can be represented using an IOMT are as follows:

1. The database can be a routing information database (RIDB). In a leaf (a, a_n, ω_a), the index a is the destination, and ω_a is the hash of the record for the destination a. The root of the tree is a succinct representation of the entire RIDB stored by a router. $\omega_a = 0$ implies no record exists for a.
2. The database can be an access control list (ACL) for a specific resource. The value ω_a specifies the type of access granted to a. For example, $\omega_a = 1$ may imply read-only access, $\omega_a = 2$ may imply read-write access, $\omega_a = 3$ may imply the permission to even modify the ACL, etc. $\omega_a = 0$ implies no access.
3. The database can represent a list of items some of which may occur multiple time. A leaf (a, a_n, ω_a) indicates that the item a appears ω_a times in the list. For example, a list $(2, 2, 3, 3, 4, 6, 6, 6, 6)$ can be represented by an OMT with four leaves

$$(2, 3, 2), (3, 4, 2), (4, 6, 1), (6, 2, 4). \tag{9.22}$$

   It is easy to see that inserting a placeholder for an arbitrary index (say) 245 does not change the database. After insertion of the placeholder for index 245, the five leaves of the IOMT will be

$$(2, 3, 2), (3, 4, 2), (4, 6, 1), (6, 245, 4), (245, 2, 0). \tag{9.23}$$

4. The database can represent credential associations. For example, an entity a and a public key ω_a, or an entity a and the domain name ω_a owned by the entity, or the IP address ω_a assigned to a network interface with message authentication code (MAC) address a (or vice-versa). $\omega_a = 0$ implies no information is available regarding address a.

5. More generally, the third field in each leaf can even be a root of another OMT. For example, an OMT leaf (a, a_n, ω_a) may indicate that ω_a is a root of an ACL IOMT for a resource identified as a. $\omega_a = 0$ implies no ACL exists for resource a.

Note that in each scenario the act of inserting a placeholder (with value 0) does not affect the integrity of the database. Consider the following two scenarios where for an IOMT

1. No leaf exists for an index a.
2. A placeholder $(a, a_n, 0)$ exists.

In the first scenario, a leaf (b, b_n, ω_b) should exist in such a tree such that $f_{encl}((b, b_n), a)$ is TRUE. By demonstrating the existence of the leaf (b, b_n, ω_b), the prover can convince the verifier that "no information exists regarding index a." In the second scenario, the prover can demonstrate the existence of the leaf $(a, a_n, 0)$ to convince the receiver of the same thing—that "no information exists regarding a."

9.2.7 Domain Ordered Merkle Tree

The DOMT can be seen as a *look up table* (LUT) for a step-wise approximation of any function $y = f(x)$. In a DOMT, each leaf (say, (a, a_n, ω_a)) indicates the half-open domain $[a, a_n)$ of the independent variable x (or $a \leq x < a_n$), corresponding to which the function evaluates the dependent variable $y = \omega_a$. In general, a DOMT leaf (a, a_n, ω_a) indicates that the interval $[a, a_n)$ is "associated" with a value ω_a.

The following are some examples of databases that can be represented using a DOMT:

1. IP registry database. A leaf (a, a_n, ω_a) indicates that the IP addresses in the range $a \leq x < a_n$ are assigned to an entity ω_a. $\omega_a = 0$ implies unallocated addresses $[a, a_n)$.
2. An LUT for a possibly complex function $y = f(x)$. $\omega_a = 0$ implies that the function is undefined for the range $[a, a_n)$.
3. A two dimensional LUT for a function $y = f(x_1, x_2)$. In this case, in a leaf (a, a_n, ω_a), $[a, a_n)$ is the domain of x_1 and ω_a is the root of a DOMT. A leaf (b, b_n, ω_b) in the DOMT with root ω_a conveys the domain $[b, b_n)$ of the independent variable x_2, and the range $y = f(a \leq x_1 < a_n, b \leq x_2 < b_n) = \omega_b$.

While a nested DOMT can be used literally as an LUT of two variables, it can also be used for representing real-life databases. For example, such a nested DOMT could be used in geographic information systems to represent latitude and longitude enclosures for tessellations of geographical regions owned by/delegated to various entities. In such a scenario, a leaf (a, a_n, ω_a) and a leaf (b, b_n, ω_b) in the DOMT with root ω_a conveys that ω_b is the owner of the tessellation demarcated by longitudes $[a, a_n)$ and latitudes $[b, b_n)$.

In such a DOMT, $\omega_a = 0$ implies that no entity has been assigned any area between longitudes $[a, a_n)$. $\omega_b = 0$ (in a leaf in the nested OMT with root $\omega_a \neq 0$) implies that the tessellation $([a, a_n), [b, b_n))$ is unassigned.

Recall that in a DOMT (as in any OMT), for inserting a leaf with first field c, a leaf (a, a_n, ω_a) should exist such that (a, a_n) encloses c. In a DOMT, after insertion the enclosing leaf will become (a, c, ω_a), and the newly inserted leaf will be $(c, a_n, \omega_c = \omega_a)$. Such an operation does not change the integrity of the database, as all that insertion accomplishes is the split domain into two intervals and assigns the same range (third field) to both intervals, before the split interval $[a, a_n)$ was associated with ω_a. After insertion of a placeholder, the two leaves indicate that (1) interval $[a, c)$ is associated with ω_a; and (2) interval $[c, a')$ is also associated with ω_a.

For example, a DOMT leaf $(10, 45, \omega)$ implies that values in the interval $[10, 45)$ are associated with ω. After insertion of a new leaf, say with first field 22, the two leaves will become $(10, 22, \omega)$ and $(22, 45, \omega)$, implying values in the range interval $[10, 22)$ are associated with ω and values in the range interval $[22, 45)$ are associated with ω.

In a 2D OMT, where the outer OMT provides horizontal coordinates and the nested OMTs correspond to vertical coordinates, inserting a leaf in the outer OMT splits a rectangle into two rectangles with the same vertical dimensions. Inserting a leaf in an inner OMT splits a rectangle into two rectangles with the same horizontal dimensions.

9.2.8 Summary of OMT Properties

Some of the important properties of OMTs are as follows:

1. The leaf hash corresponding to an empty leaf Φ is zero.
2. An OMT with root 0 can be seen as a tree with *any* number of empty leaves.
3. For a tree with a single nonempty leaf, the leaf node is the same as the root of the tree.
4. The existence of a leaf (a, a, ω_a) (the first two fields are the same) in an OMT indicates that the leaf is the sole leaf in the tree (the root of the tree will be the same as the leaf node $H_L(a, a, \omega_a)$).
5. Existence of a leaf like $(1, 3, \omega_1)$ is proof that no leaf exists with first field in-between 1 and 3. Existence of a leaf like $(7, 1, \omega_7)$ is a proof that no leaf exists with first field less than 1 *and* that no leaf exists with first field greater than 7.
6. Swapping leaves of an OMT does not affect the integrity of the database represented by the OMT. For example, both

$$\begin{array}{l}(1, 3, \omega_1), (3, 4, \omega_3), (4, 7, \omega_4), (7, 1, \omega_7) \text{ and} \\ (3, 4, \omega_3), (1, 3, \omega_1), (4, 7, \omega_4), (7, 1, \omega_7)\end{array} \tag{9.24}$$

   represent an identical database with four records—either an IOMT corresponding to four indexes 1, 3, 4, and 7, or a DOMT for four intervals $[1, 3)$, $[3, 4)$, $[4, 7)$, and $[7, 1)$. The interval represents all values greater than or equal to seven, and all values less than one).

7. A leaf with a first field a can be inserted only if an encloser exists. that *circularly encloses* a exists.
8. A placeholder is a nonempty leaf whose insertion does not change the interpretation of the database. For an IOMT, a placeholder is of the form $(a, a', 0)$. Introduction of a placeholder for an index A does not change the database in any way, as both the existence of placeholder for index a and the nonexistence of a leaf for index a implies that "no record exists for index a." Thus,

$$\begin{array}{c}(3,4,\omega_3),(1,3,\omega_1),(4,7,\omega_4),(7,1,\omega_7) \text{ and} \\ (3,4,\omega_3),(1,3,\omega_1),(4,5,\omega_4),(5,7,0),(7,1,\omega_7)\end{array} \tag{9.25}$$

   which correspond to before and after insertion of a placeholder for an index 5, represent an identical database.
9. For a DOMT, a placeholder is a leaf with third value, the same as the third value of the encloser. Specifically, inserting a leaf can be seen as a process of splitting a leaf (for example), $(4, 7, \omega_4)$ into two leaves (for example) $(4, 5, \omega_4)$ and $(5, 7, \omega_4)$. Thus, both

$$\begin{array}{c}(1,3,a),(3,4,b),(4,7,c),(7,1,d) \text{ and} \\ (1,3,a),(3,4,b),(4,5,c),(5,7,c),(7,1,d)\end{array} \tag{9.26}$$

   represent an identical database. Before insertion, the leaf $(4, 7, c)$ indicated that values $4 \leq x < 7$ are associated with c. Nothing has changed after the range is split into two, as values $(4 \leq x < 5)$ *and* values $(5 \leq x < 7)$ are associated with the same quantity c.
10. While operations like swapping leaves in any OMT or insertion/deletion of a placeholder do not change the contents of the database, they will result in a change in the root of the tree—say from r to r'. Such roots are considered as *equivalent* roots.

9.3 OMT Algorithms in Trusted Resource Limited Boundaries

In practice, the verifier **T** will often be trustworthy tamper-responsive module. Such a module **T** is expected to be write-proof—to protect the integrity of the dynamic OMT root stored inside, and to preserve the integrity of OMT algorithms (necessary for protocols like verification, update, insertion, and swapping of OMT leaves) executed inside the boundary. As the module **T** will be required to certify the integrity of records to third parties, the secrets used for authentication of outputs need to be protected to ensure that the module cannot be impersonated. In other words, the module also needs to be read-proof.

One obvious goal is to minimize the computational and memory requirements inside the module **T** for executing the OMT protocols. One challenge in implementation of various OMT algorithms for verification, insertion/deletion of leaves, and swapping of leaves is that for large databases the VOs themselves ($\log_2 N$ values for a tree with N leaves) may demand non-negligible memory requirements inside the module. Furthermore, the fact that VOs can be of any size (as the database can be of any size) can also complicate algorithms executed inside **T**.

9.3.1 Self-Certificates

One useful strategy to address this challenge is that of employing self-certificates. A self-certificate is a memorandum issued to oneself. For example, an entity A may issue a memorandum to itself to the effect that "a value V was received from Y at time t" by computing a MAC

$$\mu = h(\theta \parallel \chi), \text{where}\, \theta = V \parallel Y \parallel t, \tag{9.27}$$

and χ is a secret known only to A. The certificate (or MAC) μ can be handed over by the "memory-constrained" entity A (which stores only the secret χ) to an untrusted entity (prover) for storage. A may prepare any number of certificates in this manner (number of certificates limited only by the storage available for the prover). The prover can at any time convince A (by providing values X, Y, t, and μ) that "A received a value V from Y at time t."

As one example of how such an approach can reduce the memory requirements inside the module consider a scenario where a database whose integrity is assured by **T** has close to 2^{26} (about 64 million) records. To verify the integrity of any leaf, the module will require temporary storage for a VO with 26 values. However, using a symmetric certificate enables the module to safely split the computations required into multiple steps.

For example, assume that the module can support only VO sizes up to eight. Let v be a leaf node of a tree with root r, with depth 26 (between 2^{25}and2^{26} leaves). In such a scenario, the prover can provide a VO with eight values to map v to an ancestor y_1, eight levels higher. After verifying that y_1 is indeed an ancestor of v, the module **T** issues a self-memoranda to the effect that "it has been verified that y_1 is an ancestor of v." For example, such a certificate could be of the form

$$\mu = h(y_1 \parallel v \parallel \chi) \tag{9.28}$$

A simple function $y = f_{omt}(v, i, \mathbf{v})$ that maps a value v to a value y up to eight levels higher can be reused for mapping the ancestor y_1 to another ancestor y_2 at a higher level, and so on. Thus, the prover can obtain certificates relating y_1 to an ancestor y_2 at level 16, and a certificate relating y_2 to an ancestor y_3 at level 24.

Finally, to obtain a certificate relating y_3 to the root that is six levels higher, only six VOs are needed. To simplify $f_{omt}()$, assume that it does not even have to check the size of the VO (it is always assumed to be eight). This is not an issue, as a VO of length eight with last two values set to zero is the same as a VO of length six. More specifically

$$f_{omt}(v,i, \mathbf{x} = [x_0 x_1 x_2 x_3 x_4 x_5]) = f_{omt}(v, i, [x_0 x_1 x_2 x_3 x_4 x_5 00]. \tag{9.29}$$

Note that this is again a consequence of the special way in which $H_V()$—the function that maps two child nodes to the common parent—is defined.

A simple function that utilizes $f_{omt}()$ to map any node to a node, a few levels (for example, eight) higher and issues a self-certificate to this effect, along with simple functions that combine one or more such self-certificates, can eliminate the need for complex functions—for simultaneously verifying/updating a plurality of leaves, especially for large databases.

9.3.2 Core OMT Functions

OMT related functions executed inside **T** can be broadly classified into internal functions (that are not exposed) and exposed functions that can be invoked by any entity **U** with access to **T**.

The internal functions include:

1. $f_l()$ to compute a leaf-node corresponding to a leaf (same as $H_L()$ discussed earlier)
2. $f_v()$ to map two siblings to their parent (same as $H_V()$) and
3. $f_m()$ that maps a node to an ancestor eight (or less) levels above

```
$v = f_l(a, a_n, \omega_a)\{$
    IF ($a =$ 0) RETURN 0
    ELSE RETURN h($a \| a_n \| \omega_a$);
}
$p = f_v(v_1, v_2, b)\{$
    IF ($v_1 =$ 0) RETURN $v_2$;
    ELSE IF ($v_2 =$ 0) RETURN $v_1$;
    ELSE IF ($b =$ 0) RETURN $h(v_1 \| v_2)$
    ELSE IF ($b =$ 1) RETURN $h(v_2 \| v_1)$
}
$y = f_m(v, i, \mathbf{x})\{$// $\mathbf{x} = \{x[0] \cdots x[7]\}$
    tmp $\leftarrow v; j \leftarrow i$;
    FOR ($i =$ 0 TO 7)
        $tmp \leftarrow f_v(tmp, x[i],$ LSB(j));
        $j \leftarrow j >>$ 1;// right shift $j$
    RETURN $tmp$;
}                                                    (9.30)
```

The function $f_l()$ returns 0 if the first field is zero. Else it returns the hash of the leaf. If any of the two siblings is zero, function $f_v()$ returns the other sibling (which can also be zero, in which case $f_v()$ returns 0). If both siblings are nonzero, the third (single-bit) input b specifies the orientation. Depending on the value of b, $f_v()$ returns $h(v_1 \| v_2)$ or $h(v_2 \| v_1)$.

The function $f_m()$ performs eight repeated applications of $f_v()$ using the eight values in the VO $\mathbf{x}$. The input i is a 8-bit (or one byte) value. In each iteration, the LSB provides the value b required for $f_v()$. In each iteration, the byte is right shifted.

9.3.3 *OMT Functions Exposed by* T

The exposed OMT functions can be used to create self-certificates authenticated by the module. The various types of certificates created by **T** include:

1. "Node Verify" certificates (type *NV*)
2. "Two-Node Verify" certificates (type *NV*2)
3. "Node Update" certificates (type *NU*)
4. "Two-Node Update certificates, (type *NU*2) and
5. "Equivalent Root" certificates (type *EQ*)

All certificates are message authentication codes computed using a secret χ known only to the module (verifier) **T**.

A certificate of type *NV* is computed as

$$\rho_{nv} = h(NV \parallel x \parallel y \parallel \chi). \tag{9.31}$$

Such a certificate is issued by **T** after verifying the existence of a VO $\mathbf{v}$ satisfying $y = f_m(x, i, \mathbf{v})$. The values x, y, and ρ_{nv} can be presented to the module at any time to convince the module that "y is an ancestor of x."

A certificate of type *NV*2 is computed as

$$\rho_{nv2} = h(NV2 \parallel x_1 \parallel x_2 \parallel y \parallel \chi). \tag{9.32}$$

The existence of such a certificate implies that **T** has verified that "y is a common ancestor of nodes x_1 and x_2."

A certificate of type *NU* is computed as

$$\rho_{nu} = h(NU \parallel [x \parallel y] \parallel [x' \parallel y'] \parallel \chi). \tag{9.33}$$

The existence of such a certificate implies that **T** has verified that "y is a an ancestor of x, and modifying $x \rightarrow x'$ will require $y \rightarrow y'$."

Certificates of type *NU*2 are computed as

$$\rho_{nu2} = h(NU2 \parallel [x_1 \parallel x_2 \parallel y] \parallel [x'_1 \parallel x'_2 \parallel y'] \parallel \chi). \tag{9.34}$$

The existence of such a certificate implies that **T** has verified that "y is a common ancestor of both x_1 and x_2 and modifying $(x_1 \rightarrow x'_1)$ and $(x_2 \rightarrow x'_2)$ will result in $y \rightarrow y'$."

Certificates of type *EQ* are computed as

$$\rho_{eq} = h(EQ \parallel x \parallel y \parallel \chi). \tag{9.35}$$

The implication of such a certificate is that **T** has verified that “an OMT root x can be modified to y, or an OMT root y can be modified to x.” More specifically, such certificates are issued after **T** verifies that such transformations correspond to insertion/deletion of placeholders, or swapping of leaves. Thus, changing the root in accordance with an equivalence certificate does not affect the integrity of the database assured by **T**.

9.3.3.1 Function $F_{v1}()$

Function $F_{v1}()$ issues a certificate of type *NV*.

```
$\mathrm{F}_{v1}(v, i, \mathbf{v})\{$
$y \leftarrow f_m(v, i, \mathbf{v});$
RETURN $h(NV \parallel v \parallel y \parallel \chi);\}$
```

(9.36)

9.3.3.2 Concatenation of *NV* Certificates

Function $F_{vc}()$ concatenates two *NV* certificates—one stating that “y is an ancestor of x” and another to the effect that “z is the ancestor of y” to issue a certificate stating that “z is an ancestor of x.”

```
$F_{vc}(x, y, \rho_1, z, \rho_2)\{$
    IF $(\rho_1 \neq h(NV \parallel x \parallel y \parallel \chi))$ RETURN;
    IF $(\rho_2 \neq h(NV \parallel y \parallel z \parallel \chi))$ RETURN;
    RETURN $h(NV \parallel x \parallel z \parallel \chi);$
$\}$
```

(9.37)

Function $F_{vv}()$ concatenates two *NV* certificates

$$\begin{aligned} \rho_1 &= h(NV \parallel x_1 \parallel y \parallel \chi) \text{ and} \\ \rho_2 &= h(NV \parallel x_2 \parallel z \parallel \chi) \end{aligned} \tag{9.38}$$

where $y \neq x_2$. In this case, the ancestors y (of x_1) and z (of x_2) are assumed to be siblings with parent $p = f_v(y, z, 0)$ (y is assumed to be the left of its sibling z). From the two certificates, the verifier **T** can conclude that both x_1 and x_2 have a common ancestor p.

```
$F_{vv}(x_1, y, \rho_1, x_2, z, \rho_2)\{$
    IF $(\rho_1 \neq h(NV \parallel x_1 \parallel y \parallel \chi))$ RETURN;
    IF $(\rho_2 \neq h(NV \parallel x_2 \parallel z \parallel \chi))$ RETURN;
    RETURN $h(NV2 \parallel x_1 \parallel x_2 \parallel f_v(y, z, 0) \parallel \chi);$
$\}$
```

(9.39)

Note that the certificate issued by function F_{vv} binds two nodes to their lowest common ancestor. In scenarios where it may be required to demonstrate that x_1 and x_2 are nodes in a tree with root z, an $NV2$ certificate binding x_1 and x_2 to a common ancestor p can be combined with an NV certificate binding p to an ancestor z of p to issue another $NV2$ certificate binding x_1 and x_2 to ancestor p's ancestor z. Function $F_{vv'}$ accomplished this.

```
    $F_{vv'}(x_1,x_2,p,\rho_1,z,\rho_2)\{$
    IF $(\rho_1 \neq h(NV2 \parallel x_1 \parallel x_2 \parallel p \parallel \chi))$ RETURN;
    IF $(\rho_2 \neq h(NV \parallel p \parallel z \parallel \chi))$ RETURN;
    RETURN $h(NV2 \parallel x_1 \parallel x_2 \parallel z \parallel \chi)$;
$\}$                                                        (9.40)
```

9.3.3.3 Update Certificates

Most often, verification of a node against an ancestor is performed for updating the node and it's ancestor. Function $F_{u1}()$ issues a certificate of type NU which states that "y is an ancestor of x, and if $x \rightarrow x'$ then $y \rightarrow y'$."

```
    $F_{u1}(x,x',i,\mathbf{x})\{$
    $y \leftarrow f_m(x,i,\mathbf{x}); y' \leftarrow f_m(x',i,\mathbf{x})$;
    RETURN $h(NU \parallel [x \parallel y] \parallel [x' \parallel y'] \parallel \chi)$;
$\}$                                                        (9.41)
```

Note that two NV certificates cannot be combined to produce an NU certificate, as in the NU certificate, the *same* VO is used for computing both the old and the updated ancestor.

Function F_{uc} combines two NU certificates to issue another NU certificate.

```
    $F_{uc}(x,y,x',y',\rho_1,z,z',\rho_2)\{$
    IF $(\rho_1 \neq h(NU \parallel [x \parallel y] \parallel [x' \parallel y'] \parallel \chi))$ RETURN;
    IF $(\rho_2 \neq h(NU \parallel [y \parallel z] \parallel [y' \parallel z'] \parallel \chi))$ RETURN;
    RETURN $h(NU \parallel [x \parallel z] \parallel [x' \parallel z'] \parallel \chi)$;
$\}$                                                        (9.42)
```

Function F_{uu} combines two NU certificates

$$\begin{aligned} \rho_1 &= h(NU \parallel [x_1 \parallel y_1] \parallel [x_1' \parallel y_1'] \parallel \chi) \text{ and} \\ \rho_2 &= h(NU \parallel [x_2 \parallel y_2] \parallel [x_2' \parallel y_2'] \parallel \chi) \end{aligned} \tag{9.43}$$

assuming that y and z are siblings, and thus making $p = h_v(y_1, y_2, 0)$ a common ancestor of x_1 and x_2. If $x_1 \rightarrow x_1'$ and $x_2 \rightarrow x_2'$, then $y_1 \rightarrow y_1'$ and $y_2 \rightarrow y_2$, and consequently,

$p \rightarrow p' = f_v(y_1', y_2', 0)$.

```
    $F_{uu}(x_1, y_1, x_1', y_1', \rho_1, x_2, y_2, x_2', y_2', \rho_2)\{$
    IF $(\rho_1 \neq h(NU \parallel [x_1 \parallel y_1] \parallel [x_1' \parallel y_1'] \parallel \chi))$ RETURN;
    IF $(\rho_2 \neq h(NU \parallel [x_2 \parallel y_2] \parallel [x_2' \parallel y_2'] \parallel \chi))$ RETURN;
    $p \leftarrow f_v(y_1, y_2, 0); p' \leftarrow f_v(y_1', y_2', 0)$;
    RETURN $h(NU2 \parallel [x_1 \parallel x_2 \parallel p] \parallel [x_1' \parallel x_2' \parallel p'] \parallel \chi)$;
}                                                                    (9.44)
```

Function $F_{uu'}$ combines an $NU2$ certificate binding two nodes to a common ancestor p to an ancestor w of p (conveyed in a certificate of type NU)

```
    $F_{uu'}(x_1, x_2, p, x_1', x_2', p', \rho_1, z, z' \rho_2)\{$
    IF $(\rho_1 \neq h(NU2 \parallel [x_1 \parallel x_2 \parallel p] \parallel [x_1' \parallel x_2' \parallel p] \parallel \chi))$ RETURN;
    IF $(\rho_2 \neq h(NU \parallel [p \parallel z] \parallel [p' \parallel z'] \parallel \chi))$ RETURN;
    RETURN $h(NU2 \parallel [x_1 \parallel x_2 \parallel z] \parallel [x_1' \parallel x_2' \parallel z'] \parallel \chi)$;
}                                                                    (9.45)
```

9.3.4 Root Equivalence Certificates

Recall that operations like swapping leaves of an OMT or inserting a placeholder does not modify the database captured by the OMT (even while root changes from x to y). Such roots are equivalent roots.

9.3.4.1 Equivalence Due to Swapping Nodes

A function $F_{sw}()$ generates equivalence certificates after verifying that the transformation of the root corresponds to swapping two nodes in the tree. Swapping two nodes x_1 and y_1 is the same as simultaneously updating two nodes—$x_1 \rightarrow x_2$ and $x_2 \rightarrow x_1$. Thus, an $NU2$ certificate of the form

$$\rho \neq h(NU2 \parallel [x_1 \parallel x_2 \parallel y] \parallel [x_2 \parallel x_1 \parallel y'] \parallel \chi) \tag{9.46}$$

is proof that y and y' are equivalent roots. Function $F_{sw}()$ simply verifies an $NU2$ certificate with the special structure to issue an EQ certificate.

```
    $F_{sw}(x_1, x_2, y, y', \rho)\{$
    IF $(\rho \neq h(NU2 \parallel [x_1 \parallel x_2 \parallel y] \parallel [x_2 \parallel x_1 \parallel y'] \parallel \chi))$ RETURN;
    RETURN $h(EQ \parallel y \parallel y' \parallel \chi)$;
}                                                                    (9.47)
```

Equivalent certificates can be flipped or concatenated. Flipping a certificate $h(EQ \parallel x \parallel y \parallel \chi)$ modifies the certificate to $h(EQ \parallel y \parallel x \parallel \chi)$. Obviously, any equivalent transformation that was performed can also be undone readily.

Concatenation of two equivalent certificates for (x, y) and (y, z) produces a certificate that (x, z) are equivalent root. Function $F_{ce}()$ can be used for flipping or concatenating *EQ* certificates.

```
$F_{ce}(x,y,z,\rho_1,\rho_2)\{$
    IF $(\rho_1 \neq h(EQ \,\|\, x \,\|\, y \,\|\, \chi))$ RETURN;
    IF $(\rho_2 = h(EQ \,\|\, y \,\|\, z \,\|\, \chi))$
        RETURN $h(EQ \,\|\, x \,\|\, z \,\|\, \chi)$;
    ELSE RETURN $h(EQ \,\|\, y \,\|\, x \,\|\, \chi)$;
$\}$                                                        (9.48)
```

9.3.4.2 Equivalence Due to Insertion of Placeholders

For enforcing rules for the purpose of inserting a placeholder, the module needs to know if the OMT is of type IOMT or DOMT. A simple strategy to mark an OMT as an IOMT or DOMT is to reserve a bit in the first field in every leaf for this purpose. For reasons that will become apparent soon, we reserve n MSBs (for example, $n = 16$) in the first field of every OMT leaf to identity the specific nature of the OMT. For all leaves in the OMT, both the first and second fields will have n identical MSBs.

1. For an IOMT, the MSB is zero. For a DOMT, it is one.
2. The second MSB is used to identify that an OMT leaf has a nested OMT. If the second MSB is one then the third field in every leaf in the OMT is actually the root of an OMT. If it is zero the third filed is not an OMT.
3. The remaining $n - 2$ bits are reserved for reasons to be explained later.

More specifically, the specific type of the OMT is decided by the first placeholder inserted into the OMT, say $(x, x, 0)$. From this point onward, any placeholder inserted into the OMT should have the first n bits the same as those of x.

We shall represent by *MSB1(a), MSB2(a)*, and *MSBn(a)*, functions that return the first, second, and first n bits of the MSB of a value a.

The algorithm $F_{ph}()$ for generating an *EQ* certificate is as follows:

```
$F_{ph}(a, a_n, \omega_a, b, y, y', \rho)\{$
    IF $(\rho = 0)$ RETURN $h(EQ \,\|\, 0 \,\|\, f_l(b, b, 0) \,\|\, \chi)$
    IF $(MSBn(a) \neq MSBn(b))$ RETURN;
    IF $(!f_{encl}((a, a_n), b))$ RETURN;
    $v_a \leftarrow f_l(a, a_n, \omega_a); v'_a \leftarrow f_l(a, b, \omega_a); v_b = 0;$
    IF $(MSB1(b) = 0)$ $v'_b \leftarrow f_l(b, a_n, 0)$;
    ELSE $v'_b \leftarrow f_l(b, a_n, \omega_b)$;
    $\rho_1 \leftarrow h(NU2 \,\|\, [v_a \,\|\, v_b \,\|\, y] \,\|\, [v'_a \,\|\, v'_b \,\|\, y'] \,\|\, \chi)$
    $\mathrm{\rho}_2 \leftarrow \mathrm{h}(NU2 \,\|\, [v_b \,\|\, v_b \,\|\, y] \,\|\, [v'_b \,\|\, v'_b \,\|\, y'] \,\|\, \chi)$
    IF $(\rho \notin \{\rho_1, \rho_2\})$ RETURN;
    RETURN $h(EQ \,\|\, y \,\|\, y' \,\|\, \chi)$;
$\}$                                                        (9.49)
```

In general, as two nodes will need to be simultaneously modified to insert a placeholder, an *NU*2 certificate is required to generate an *EQ* certificate for inserting placeholders. An exception is for the first placeholder. If the first placeholder to be included is for an index b then the root before and after insertion will be 0 (corresponding to an empty tree) and $f_l(b, b, 0)$ (the root corresponding to the tree with a single placeholder). Thus, 0 and $f_l(b, b, 0)$ are equivalent for all b's.

For all other cases, insertion of a placeholder for b requires existence of a cover leaf (a, a_n, ω_a). Yet another condition to be satisfied is that the two MSBs of a and b should be the same. After insertion, the leaf node $v_a = f_l(a, a_n, \omega_a)$ will be modified to $v'_a = f_l(a, b, \omega_a)$ and corresponding to a newly inserted node a leaf node $v_b = 0$ will be modified to $v'_b = f_l(b, a, 0)$ (if $MSB1(b) = 0$) or $v'_b = f_l(b, a_n, \omega_a)$ (if $MSB1(b) = 1$).

The *NU*2 certificate relating the two nodes v_a, v_b and the current root y and v'_a, v'_b with modified root y' conveys that y and y' are equivalent.

Note that there are two possibilities for such an *NU*2 certificate, depending on whether the placeholder for b is inserted to the right/left of the encloser a.

$$\begin{aligned} \rho =& h(NU2 \parallel [v_a \parallel v_b \parallel y] \parallel [v'_a \parallel v'_b \parallel y'] \parallel \chi) \text{ or} \\ \rho =& h(NU2 \parallel [v_b \parallel v_b \parallel y] \parallel [v'_b \parallel v'_b \parallel y'] \parallel \chi) \end{aligned} \tag{9.50}$$

9.3.4.3 Equivalence Transformations to Nested OMTs

Consider a scenario where an OMT is nested. In such a scenario, in a leaf (a, a_n, ω_a), the third field ω_a is the root of another OMT. To insert a placeholder in the inner OMT (with root ω_a), or to swap leaves in the inner OMT, an *EQ* certificate of the form $h(EQ \parallel \omega_a \parallel \omega'_a \parallel \chi)$ can be readily obtained.

In general, modifying the third value in any OMT leaf can be performed only by providing an application specific justification J. However, we make an exception for changes in the third field due to equivalent changes in the nested OMT.

Thus, in addition to the equivalent certificate, an *NU* certificate is needed to update the root y of the outer tree to y'. Specifically,

1. the root y should be consistent with leaf node $v = f_l(a, a_n, \omega_a)$,
2. the new root y' should be consistent with $v = f_l(a, a_n, \omega'_a)$, and
3. ω_a and ω'_a should be equivalent.

Function $F_{nst}()$ issues a certificate to enable equivalence transformations in nested OMTs. This function can be used only if the second MSB is set in the OMT leaf index (indicating the presence of a nested OMT).

```
$F_{nst}(a, a_n, \omega_a, \omega'_a, y, y', \rho, \rho_e)\{$
    IF $(MSB2(a) \neq 1)$ RETURN;
    IF $(\rho_e \neq h(EQ \parallel \omega_{\mathrm{a}} \parallel \omega'_{\mathrm{a}} \parallel \chi))$ RETURN;
    $v \leftarrow f_1(a, a_n, \omega_{\mathrm{a}}); v' \leftarrow f_1(a, a_n, \omega'_a)$;
    IF $(\rho \neq h(NU \parallel [v \parallel y] \parallel [v' \parallel y'] \parallel \chi)$ RETURN;
    RETURN $h(EQ \parallel y \parallel y' \parallel \chi)$;
}                                                            (9.51)
```

9.3.5 *Module* **T** *State*

The state of the module (verifier) **T** is captured succinctly by a single OMT root r. The module functionality outlined thus far, viz.,

1. the three core (or internal) functions
   a) $f_l()$: for obtaining the leaf node corresponding to an OMT leaf,
   b) $f_v()$: for computing the parent of two sibling nodes; and
   c) $f_m()$: for determining an ancestor of a node at higher level (upto eight levels higher);
2. and the 12 exposed functions
   a) $F_{v1}()$: for generating a certificate of type *NV*
   b) $F_{vc}()$: for concatenating two *NV* certificates to create a new *NV* certificate
   c) $F_{vv}()$: for concatenating two *NV* certificates to create an *NV*2 certificate
   d) $F_{vv'}()$: for concatenating an *NV*2 and an *NV* certificate to create an *NV*2 certificate;
   e) $F_{u1}()$: for generating a *NU* certificate
   f) $F_{uc}()$: for concatenating two *NU* certificates to create a new *NU* certificate
   g) $F_{uu}()$: for concatenating two *NU* certificates to create a *NU*2 certificate
   h) $F_{uu'}()$: for concatenating an *NU*2 and an *NU* certificate to create an *NU*2 certificate
   i) $F_{sw}()$: for generating an *EQ* certificate for swapping two leaves of an OMT
   j) $F_{ec}()$: for concatenating and flipping *EQ* certificates
   k) $F_{ph}()$: for generating an *EQ* certificate for inserting a placeholder
   l) $F_{nst}()$: for generating an *EQ* certificate for performing an equivalence transformation in a nested OMT

can all be considered as *state independent* functions. All such functions operate the same way irrespective of the current state r of the module **T**.

State dependent functions, as the name suggests, either change the state, and/or are influenced by the state r. State dependent functions can be broadly classified into two categories:

1. functions that perform state changes independent of the application context; and
2. functions that are influenced or change state according to application-specific rules.

The context independent functions in the first category merely change the state r to an equivalent root r' if an equivalence certificate relating r and r' can be provided.

We need only one such context independent (but state-dependent) function $F_{ci}()$ outlined below. This function can be used to request **T** to modify its state by changing the OMT root r to an equivalent root. For this purpose, a self-certificate of type *EQ* is supplied to **T**.

```
$F_{ci}(r', \rho)$ {
    IF $(h(EQ \| r \| r' \| \chi) = \rho)$ $r = r'$;
    ELSE IF $(h(EQ \| r' \| r \| \chi) = \rho)$ $r = r'$;
    ELSE RETURN ERROR;
}                                                        (9.52)
```

As the equivalence certificate states that x and y are equivalent either by issuing a certificate $h(EQ \| x \| y \| \chi)$ or $h(EQ \| y \| x \| \chi)$. In other words, toggling an OMT root between equivalent roots x and y does not affect the integrity of the database. This is the reason why we do not need equivalence certificates for *deleting* OMT leaves. If inserting a node modifies a root from $x \rightarrow y$ then deleting the inserted root will modify the root from $y \rightarrow x$.

Consider a scenario where the OMT root is y. Also assume that the tree has a leaf that encloses the index a for the placeholder the prover desires to insert into the tree. Let the root after insertion be y'. The prover can readily obtain VOs from the OMT to demonstrate to module that y and y' are equivalent. Specifically, the VOs will be supplied to the **T** using two or more calls to $f_{u1}()$, to obtain two or more *NU* certificates. *NU* certificates can be provided as inputs to functions like $F_{uu}()$ and $F_{uu'}$ to receive an *NU*2 certificate. An *NU*2 certificate indicating old and updated ancestors y to y' can be submitted to **T** as a request to issue an *EQ* certificate binding (y, y'). After the certificate is issued, the prover can update its database (by inserting the placeholder) and request the module to update the root to y'.

To delete a placeholder, the prover can first go ahead and remove the placeholder from its OMT. Let the OMT root be x before deletion and x' after deletion. After deletion of the placeholder, the prover can readily identify VOs necessary to convince the module that (x', x) are equivalent. More specifically, the database with a placeholder already removed provides the VO necessary for convincing the module of the equivalence of (x, x'). As the prover has already changed its root to x', the prover requests the module to change its root $x \rightarrow x'$.

Recall that *EQ* certificates can also be generated for purposes of swapping leaves. As an example of the utility of the ability of the prover to swap (and thereby, rearrange OMT leaves), consider a scenario where an OMT is used to represent a dynamic database of files stored by a file server, where each leaf corresponds to a file hash. Popular file services may support millions of clients, where each client may upload thousands of files. Even while the file server is required to track billions of file hashes, during small interval only a few thousands of the billions of files may undergo active editing. Thus, to reduce the overhead for maintaining the OMT, it is advantageous to reorder the OMT leaves such that all active files are grouped together. In this case, only a small part of the OMT used to represent the files will need to be cached for faster access.

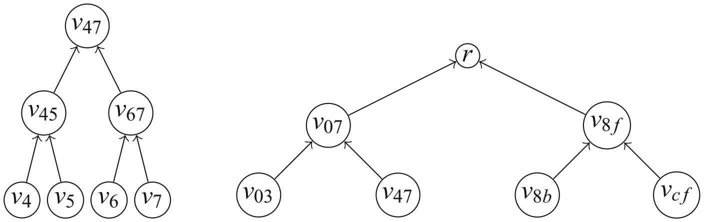


Fig. 9.3 The subtree rooted at v_{47} includes both the leaf nodes v_4 and v_7 to be updated. Updating the two nodes will require an update to v_{47}. The subtree to the right—with depth two, four leaf nodes $v_{03}, v_{47}, v_{8b}, v_{cf}$, can be used to update root r corresponding to the update to v_{47}

9.3.6 Using Module Functions

Consider a scenario where the prover maintains an IOMT shown in Fig. 9.2 and that she desires to insert a placeholder for an index 110 at position L_4 (L_4 is currently an empty leaf with corresponding leaf node $v_4 = 0$). Also assume that $L_7 = (101, 115, \omega)$ is the encloser for the leaf to be inserted.

After insertion, the leaf-node v_4 changes from $v_4 = 0 \rightarrow v_4' = f_l(110, 115, 0)$. The leaf node corresponding to the encloser L_7 will need to be changed from $v_7 = f_l(101, 115, \omega) \rightarrow v_7' = f_l(101, 110, \omega)$. Figure 9.3 depicts only the relevant subtrees affected by the update.

Note that the earliest common parent of v_4 and v_7 is v_{47}, with left child v_{45} and right child v_{67}. Now,

1. The VO that maps v_4 to the left child v_{45} of the common parent is $\{v_5\}$; the index of v_4 in the subtree rooted at v_{45} is 0.
2. The VO that maps v_7 to the right child v_{67} of the common parent is $\{v_6\}$; the index of v_4 in the subtree rooted at v_{67} is 1.
3. The VO that maps the common parent v_{47} to the root r is $\{v_{03}, v_{8\,f}\}$; the index of v_{47} in the subtree (with four leaf nodes $v_{03}, v_{47}, v_{8b}, v_{cf}$) rooted at r is 1.

Invoking function $F_{u1}(v_4 = 0, v_4', i = 0, \mathbf{x} = \{v_5\})$ will result in an *NU* certificate of the form

$$\rho_1 = h(NU \parallel 0 \parallel v_{45} \parallel v_4' \parallel v_{45}' \parallel \chi) \tag{9.53}$$

instructing that if $0 \rightarrow v_4'$ then $v_{45} \rightarrow v_{45}'$.

Invoking function $F_{u1}(v_7, v_7', i = 1, \mathbf{x} = \{v_6\})$ will result in an *NU* certificate of the form

$$\rho_2 = h(NU \parallel v_7 \parallel v_{67} \parallel v_7' \parallel v_{67}' \parallel \chi) \tag{9.54}$$

instructing that if $v_7 \rightarrow v_7'$ then $v_{67} \rightarrow v_{67}'$.

Recall that $F_{uu}()$ can be used to bind two *NU* certificates that binds two leaves to the earliest common parent. In this case, to bind v_4 and v_7 to common parent v_{47}, invoking $F_{uu}(0, v_{45}, v'_4, v'_{45}, \rho_1, v_7, v_{67}, v'_7, v'_{67}, \rho_2)$ will result in an $NU2$ certificate

$$\rho_3 = h(NU2 \parallel [0 \parallel v_7 \parallel v_{47}] \parallel [v'_4 \parallel v'_7 \parallel v'_{47}] \qquad (9.55)$$

where $v_{47} = f_v(v_{45}, v_{67}, 0)$ and $v'_{47} = f_v(v'_{45}, v'_{67}, 0)$. This certificate is an instruction if $0 \rightarrow v'_2$ and $v_7 \rightarrow v'_7$, then the common parent changes from $v_{47} \rightarrow v'_{47}$.

The common parent v_{47} can be mapped to root r by invoking $F_{u1}(v_{47}, v'_{47}, i = 1, \mathbf{x} = \{v_{03}, v_{8\ f}\})$, resulting in a certificate

$$\rho_4 = h(NU \parallel [v_{47} \parallel r] \parallel [v'_{47} \parallel r'] \parallel \chi) \qquad (9.56)$$

where r' should be the root resulting from insertion of the leaf.

Recall that F'_{uu} can be used to combine an $NU2$ certificate binding two nodes with a common ancestor to an ancestor at a higher level. In this case, we desire to bind nodes v_2 and v_7 to ancestor r. Invoking $F_{uu'}(0, v_7, v_{47}, v'_2, v'_7, v'_{47}, \rho_3, r, r'\rho_4)$ will result in another certificate of type $NU2$

$$\rho_5 = h(NU2 \parallel [0 \parallel v_7 \parallel r] \parallel [v'_2 \parallel v'_7 \parallel r'] \parallel \chi). \qquad (9.57)$$

This certificate is an instruction if $0 \rightarrow v'_2$ and $v_7 \rightarrow v'_7$, then an ancestor $r \rightarrow r'$.

The prover can now invoke the function $F_{ph}(a = 101, a_n = 115, \omega_a = \omega, b = 110, r, r', \rho_5)$ that will simply produce an equivalence certificate

$$\rho_6 = h(EQ \parallel r \parallel r' \parallel \chi), \qquad (9.58)$$

declaring the equivalence of roots r and r'.

For executing all the functions above the module did not bother to refer to the root r stored inside. As long as the prover is maintaining a tree consistent with the root r stored by the module, it is trivial for the prover to identify VOs necessary to obtain the necessary equivalence certificate.

Finally, in order to actually request the module to change the root to r', the prover invokes $F_{ci}(r', \rho_6)$.

9.3.7 Context/Application Dependent Functions

The context dependent functions have two main purposes:

1. specify rules for initialization of the OMTs and
2. specify rules for updating the value field in any OMT leaf. Thus far, we have loosely denoted such rules as a *justification* J for requesting an update.

In applications secured by leveraging trusted modules, the broad purpose of a module **T** is to assure the integrity of the information obtained from a database maintained by an untrusted entity **U**. Specifically, entities that receive information from **U** expect the information to be authenticated by the trusted module **T** associated with **U**. In order to ensure the integrity of the database, the module is required to ensure that the database is initialized *and* updated only according to application-specific rules.

As a practical example, consider a scenario where **U** is an untrusted remote file service. The users of the file service upload files to the file server. The owner/creator of the file may specify access control lists (ACL) for each file by explicitly identifying the users that have access to the file and the type of access (for example, read-only, read-write, etc.).

In such an application, when a user requests a file from the file server, the file server is expected to provide the latest version of the file. To be convinced of the integrity of the file provided by the file server, the users expect the module **T** to authenticate the file hash corresponding to the latest version of the file. Only the owner of the file, and the users with read-write-access (in the ACL specified by the owner) may supply newer versions of the file.

As the server **U** is untrusted, all desired assurances sought by the users should be realized by leveraging only the trust in **T**. It is very important for the module to provide an authenticated response (even if it is negative response) to every query from every user.

Specifically, as any user is free to request any file (even nonexistent ones), it is necessary for **T** to be able to readily determine nonexistence of files so that the querier can be informed accordingly. Note that if **T** is not able to readily infer nonexistence of a file, then the querier will be required to trust the file service **U** when the file server claims that the queried file does not exist—a trust that can be abused by **U** (by denying the presence of files that actually exist).

Similarly, if the file exists, but the querier does not have access to the file, once again **T** should be able to unambiguously determine that the querier does not have access (or the querier is not included in the ACL for the file). If the module **T** is not able to determine lack of access, then the server may abuse its privilege and deny access to an authorized user.

One strategy to secure such an application is for the module to maintain two IOMT roots—say r_1 and r_2. The corresponding two IOMTs maintained by **U** are:

1. File hash IOMT: The file hash IOMT is indexed as a function of the file name and name of the owner; for example a file with name f created by U_i will be represented as $f_i = h(f \parallel U_i)$. The value field is the file hash for the latest version of the file.
2. ACL IOMT: The ACL IOMT has the same index as the first IOMT. For this IOMT, the value field is the root an IOMT which serves as the ACL for the file.

Let us assume that every user has a unique identity, and that the module can easily compute the secret shared with each entity. Let K_i be the secret shared between **T** and user U_i.

The rules for updating the IOMTs are as follows:

1. In both IOMTs, the value field in leaf with index f_i can be updated from 0 to a nonzero value or from a nonzero value to 0, only by the owner of the file. For this purpose, **U** is required to provide

   a) a user name U_i and file name f satisfying $f_i = h(f \parallel U_i)$
   b) values $gamma_{f_i}$ and ACL IOMT root α_{f_i} authenticated using secret K_i known only to the user U_i (and **T**)
   c) *NU* certificates for updating the two roots.
2. In a leaf for index f_i in the first IOMT, the value can be updated from $\omega \neq 0 \rightarrow \omega' \neq 0$ by the owner, or by a user in the ACL for the file. For demonstrating that a user, U_x (who has requested the update), can indeed update the file, the server **U** is required to provide the following inputs to **T**
   a) Values $(U_x, U_x', a = 2)$ (assume $a = 2$ implies read-write access for U_x) and an *NV* certificate relating a node $v = f_l(U_x, U_x', 2)$ to a value y.
   b) A leaf (f_i, f_i', y) from the ACL IOMT along with an *NV* certificate relating a node $v_o = f_l(f_i, f_i', y)$ to a value r_2, where r_2 is the root of the second IOMT.
   c) Values x and x' corresponding to old and new file hashes for the file f_i, duly authenticated by user U_x.
   d) An *NU* certificate to update the root r_1 to r_1'. As in the first IOMT, a leaf $(f_i, f_i', x) \rightarrow (f_i, f_i', x)$, a leaf node in the first IOMT needs to be modified from $v_1 = f_l(f_i, f_i', x) \rightarrow v_1' = f_l(f_i, f_i', x')$. The *NU* certificate should bind $v_1 \rightarrow v_1'$ to $r_1 \rightarrow r_1'$.
3. If the user does not have access to the file, then an encloser (U_y, U_y', a) should be demonstrated to be consistent with y in the ACL leaf (f_i, f_i', y), by providing an *NV* certificate. Another *NV* certificate should be provided binding the leaf node $v_o = f_l(f_i, f_i', y)$ with the root r_2 of the ACL IOMT. Alternately, a placeholder for U_x can be inserted into the tree with root y by providing an equivalent certificate. The third value of zero in the placeholder will convince the module **T** that U_x does not have write access.

As long as such simple update rules can be encoded in a manner that is easily enforceable by resource limited **T**, such modules can be leveraged to assure the integrity of even complex databases.

9.4 Infrastructural Requirements

Assume that a trusted infrastructure $\mathcal{T}$ exists to mass produce identical tamper-responsive modules **T**. We have already discussed several components of useful algorithms that can be executed inside resource limited boundaries. Apart from the OMT functions discussed in this chapter, module functionality will include functionality $f_{pw}()$ for computing pairwise secrets (using MLS/PBK/SKIT, etc.), and atomic relay function $F_{ar}()$ which leverages $f_{pw}()$.

While all modules will have identical functionality their differences stem from two aspects:

1. Every module has a unique identity, and unique secrets corresponding to the identity.
2. Every module is initialized with some parameters depending on the expected *role* of the module.

Assume that all module identities are 160-bits long, and the first 32 bits are used to convey information regarding the nature of the module. For example, some bits may indicate the extent to which the module can be trusted—depending on the extent of tamper-resistance features and the level of rigorousness of integrity verification performed before the module was made available for use. Other bits may be used to identify the specific role of the module. The exact nature of the tasks performed by any module will depend on the nature of the databases—the integrity is assured by the module. The databases tracked by the module will in turn depend on the intended role of the module.

We already saw that for a module with a role of assuring the integrity of a remote file service, one possible approach was to maintain two IOMTs. In general different roles will call for different number and types of OMTs. As some practical examples of various possible roles:

1. module for protecting integrity of IP registry database:
2. module for border gateway protocol (BGP) router
3. module for AODV (an ad hoc routing protocol) router
4. module acting as a key distribution center
5. module associated with a certificate authority
6. module for a general look-up server
7. module for zone owner database
8. module for zone server database
9. module for caching-only domain name system (DNS) server database
10. module for IMAP server
11. module for POP3 server,
12. module for maintaining GIS database
13. module for a publish–subscribe system
14. module for protecting group secrets in a pub–sub system, etc.

The remaining 128 bits may be used to specify a unique identity within the role. For example, for a DNS zone owner module, the name may be obtained as a one-way function (hash) of the owned name. For any server, the identity could be the hash of the DNS name. More generally, 128 bits of the identity could be obtained by hashing a descriptive identity.

All modules with the same role will be initialized with the same initial state (OMT roots). The main purpose of initialing the OMT roots is to convey the specific structure of OMTs used. Note that inserting one leaf into any OMT is enough to completely characterize the OMT (whether it is an IOMT or DOMT, if it has nested OMTs, etc.) as the first n bits of every leaf in an OMT are constrained to be the same. If an OMT has nested OMTs then just one leaf is required in the nested OMT to characterize that OMT. If the nested OMT has another level of nesting then one leaf will be required in the innermost level too.

For example, if the first OMT is an IOMT and has a nested DOMT, all that is needed to convey this is to create one leaf (a, a', ω_a) where the first two bits of a convey the type of OMT (0 and 1 in this case as the OMT is an IOMT and it has a nested OMT). The root ω_a of the nested OMT may correspond to a single leaf

$(b, b', \omega_b \neq 0)$ where the first two bits are 1 (DOMT) and 0 (ω_b is not an OMT root). If less than four OMTs need to be maintained for a particular role, the unwanted OMTs may be initialized to zero.

Untrusted provers **U** (for example, a server) that utilize the module will be expected to initialize their databases to be consistent with the initial roots. From this point onwards, they can perform operations like inserting any number of placeholders, swapping positions of leaves if required etc. Each time to ensure that the roots in the module stay in sync with the database maintained by the prover **U**, the prover can utilize exposed module functionality to create equivalent certificates and use such certificates along with function $F_{ci}()$ to modify the roots stored by the module.

The reason that the database (OMT) maintained by the prover and the root of the module needs to be in sync is that no one will accept information from **U** unless it is authenticated by **T**. However, **T** will authenticate only information consistent with the roots stored inside.

Ideally, we would like to identify a small number of simple rules for any application/role. However, the number of rules is not really a practical concern. Using a large number of simple rules does not affect the complexity of operations that need to be performed inside the trusted boundary, as the rules themselves can be stored outside as leaves of an OMT. Only the root of the tree will need to be stored inside. Thus, complex application-specific rules can be broken into multiple simple rules.

From the perspective of functions required to perform equivalence transformations on OMT roots, the first two bits of an OMT leaf provide all necessary information (IOMT or DOMT, nested or not). However, from the perspective of an application rules, an IOMT indexed by (say) file names may have to be treated differently from an IOMT indexed by (say) file hashes. Thus, the application-specific rules can benefit from using the $n - 2$ reserved bits in each leaf to identify different (application specific) types of OMTs.

To summarize, the responsibilities of the trusted infrastructure $\mathcal{T}$ are as follows:

1. certify modules **T**
2. assign identities to module
3. deploy key distribution centers for facilitating pairwise secrets between modules
4. identify different roles, and static rules associated with each role.

The role dependent static rules will need to provide an unambiguous specification of conditions required to update different types of OMT leaves. The root of the OMT that includes static rules and parameters as leaves becomes the unambiguous definition of the security protocol for the role.

With the existence of such an infrastructure modules, **T**, with fixed functionality, can be leveraged to secure possibly any application. More importantly, for such applications **T** is the trusted computing base. No component/personnel/organization (except the infrastructure $\mathcal{T}$) needs to be trusted to realize the desired assurances.

However, it is obviously desirable to reduce the scope of the involvement of the trusted infrastructure $\mathcal{T}$. In the next chapter, we shall discuss broad strategies to limit the involvement of the trusted infrastructure to merely certifying the integrity of modules.

Chapter 10
Universal Trusted Computing Bases

For any system with a desired set of assurances, the trusted computing base (TCB) includes every component that needs to be trusted to realize the desired assurances [29]. In other words, the assumptions regarding the integrity of the TCB components, and more importantly, *only* the assumption of integrity of the TCB is required to realize all desired assurances for the system.

Unfortunately, for almost every practical system in use today, we either do not explicitly understand what exactly is included in the TCB, or make unjustifiable assumptions regarding certain components in the TCB. Obviously, the incorrectness of such assumptions can lead to the failure to realize the desired assurances.

10.1 Practical Systems

When Average Joe sends an e-mail, he implicitly expects the contents of the mail to be privy only to the addressee. He does not pause to ponder if the e-mail could be accessed by personnel with access to the e-mail server, or computers in the local area network (LAN), or someone at the other side of the globe, who has surreptitiously acquired control of a computer in the LAN. When he makes a credit card transaction over a "secure SSL connection," he assumes that the transaction is secure. He does not pause to consider where and how the credit card information is stored, or who has the ability to access this information, if and what mechanisms are enforced to prevent abuse. When he performs a mundane task like accessing the weather report on a smart phone, he simply accepts the report as valid—even while he has no knowledge of the numerous possible reasons that may result in misrepresentation of the report.

10.1.1 Complexity and Ignorance

This gap between *implicit assumptions* and *reality*—resulting from an inevitable lack of transparency regarding the operation of complex systems—is not limited merely to uninformed users of systems. Just as end users are required to unquestioningly

M. Ramkumar, *Symmetric Cryptographic Protocols,* DOI 10.1007/978-3-319-07584-6_10,

trust a system (about which they may know next to nothing), entities involved in the operation/design of a system are often required to unquestioningly trust various lower-level building blocks/subsystems of the system.

Increasingly, a subsystem is often an application running on a computer or a network of computers. There are two broad reasons as to why assumptions regarding the integrity of subsystems are ill-advised: *complexity* and *ignorance* (lack of domain knowledge).

The complexity of most subsystems render it impractical to rule out the presence of hidden undesired functionality within the subsystem. Hidden functionality may be in the form of deliberate Trojan horses or accidental bugs in any hardware/software component of the subsystem, and/or malice/incompetence in personnel who may be involved in the production, testing, maintenance, or operation of the application.

Users of a system X, who may have very little idea of the inner workings of the system X, nevertheless tend to make some implicit (and often incorrect) assumptions regarding specific properties of system X. Some such users of a system X may include designers of a more complex system Y that employs X as a subsystem. In an increasingly interconnected world, very soon (if not already), we may even lose track of many such implicit (and possibly incorrect) assumptions.

For example, in various security protocols designed to assure the operation of a complex system like the Internet, we are required to trust organizations like registries and registrars (responsible for delegation of domain names, IP addresses, and autonomous system (AS) numbers), certificate authorities (CA) of the public key infrastructure (PKI), etc. We do not ponder if and what mechanisms exist to prevent the domain name registry from issuing the same domain name to multiple entities, or who has legitimate access to the databases maintained by registries, and how such privileges can be abused. We do not question the integrity of personnel who may be able to modify the registry database; if such personnel use strong passwords; if they protect their passwords well; or if a password is accessible to anyone who cares to look under their desk for a Post-it note.

We do not question the integrity of hardware and software running on the platforms used by such organizations or the suppliers of such components. We do not ponder if such platforms can be illegitimately controlled by unknown entities by exploiting a hidden functionality in some component of the platform.

We do not question the security of registration mechanisms used by CAs before they certify the public key for a domain name or the integrity of the process employed by PKI registrars to verify the claimed credentials of the owner of the key before their public key is attested by the CA. We do not question how CAs protect their private keys, or who is responsible for ensuring that they do so. We do not question how the private key of an organization is protected, or who exactly in the organization guards the key, and what safeguards are in place when the private key is actually used (to sign a document or decrypt a secret).

The Internet, notwithstanding its questionable assurances, serves as a foundation for several complex systems. With every passing day, the complexity of systems/applications built on top of the Internet increases, along with our reliance on such systems. We build clouds; we build platforms for running complex web services; we build systems that assimilate data from disparate sources; and "intelligently" fuse and digest data to provide valuable information to end users or to other complex systems that depend on that information. A mundane application like an automatic vacation planner may fuse data from several systems like weather reports, bus/train/airplane schedules, hotel/attraction reviews, etc.

The issue of "trust without adequate transparency" is at the core of almost every security issue we face today, and will only get worse with increasing sophistication and complexity of systems.

10.1.2 System Security Model

Securing any system is a process consisting of three broad elements:

1. Specification of *desired assurances* $\mathcal{D}$
2. Identification of reasonable *assumptions* $\mathcal{A}$ and
3. Design of a *security protocol* $\mathcal{S} : \mathcal{A} \rightarrow \mathcal{D}$ to translate the assumptions $\mathcal{A}$ to the desired system-specific assurances $\mathcal{D}$

The assumptions $\mathcal{A}$ can be tangible or intangible. Examples of tangible (quantifiable) assumptions include assertions regarding the strengths of well-tested cryptographic primitives like AES, SHA-1, RSA, etc. Intangible assumptions include broad assertions like "the integrity of CA of the PKI," and various system-specific assumptions regarding the integrity of specific components of a system.

Ultimately, security violations—or failures to meet desired assurances—rarely[1] result from improper design of the security protocol, or incorrect assumptions regarding the strengths of cryptographic primitives. Often, the unsurprising reason is that some *intangible assumptions turn out to be incorrect*.

Computers with complex hardware and software components, with a high likelihood of hidden malicious functionality, are an increasingly integral part of almost every facet of our day to day lives. Notwithstanding the fact that it is infeasible to provide meaningful assurances regarding the integrity of tasks performed by such computers, they are nevertheless entrusted with various crucial tasks like managing and performing financial transactions, controlling the operation of various critical infrastructures: military systems and civilian systems like nuclear plants, water supply systems, traffic control systems, communication networks, etc.

Ideally, we would like to eliminate all such intangible assumptions. One common approach to minimize the extent of intangible assumptions regarding the integrity of complex entities (for example, computers, or networks of computers, or personnel, or organizations) is to replace the trust in complex entities with the trust in a hardware module. More specifically, the assumption of integrity of a set of functions $\mathcal{F}$

[1] Notable exceptions to this rule are the infamous WEP [104] protocol, and the Xbox hack [105] that exploits a weakness in the Tiny Encryption Algorithm (TEA) cipher [106].

executed inside a trustworthy boundary like a tamper-responsive hardware module can replace some or all intangible assumptions that may be necessary for a security protocol. Two well-known approaches that rely on a trusted hardware module specification to bootstrap security assurances include:

1. The Trusted Computing Group (TCG) trusted platform module (TPM) [107] approach, and
2. Trinc [108].

10.2 Trusted Platform Modules

Currently, the predominant approach to provide some extent of assurance regarding the tasks performed by a general purpose computer is the trusted computing group (TCG) approach to realize a trusted platform.

The TCG approach is an attempt to transform a platform constituted by untrustworthy general-purpose components into a trusted platform by ensuring that "only preverified and authorized software can take control of the platform." To accomplish this goal, the TCG model relies on three roots of trust:

1. Root of trust for measurement (RTM)
2. Root of trust for storage (RTS) and
3. Root of trust reporting (RTR)

Two of three roots of trust are *inside* the trusted boundary of a TPM chip housed in the platform. Specifically, the RTS takes the form of a set of platform configuration registers (PCR) maintained by the TPM; the RTR leverages the private key of a TPM to sign the PCR values. Trust in RTS stems from the assumption that the registers inside the TPM cannot be modified by external entities except by using TPM commands executed by the TPM. Trust in the RTR stems from the assumption that the private key of a TPM cannot be exposed, and is privy only to the TPM, and that the entity/authority that had authenticated the corresponding public key[2] is trustworthy.

The third root, RTM, is constituted by components *outside* the TPM. The implicit assumption behind the trust in the RTM is that "some essential hardware" for running software (like CPU, RAM, BIOS, etc.) are trustworthy.

10.2.1 Realizing a TCG Trusted Platform

Starting from the time a computer is booted up (from the time the CPU receives the first instruction stored at a fixed address), every piece of code is measured before

[2] Typically the manufacturer of the TPM chip.

control is passed to the code. The unit of code is a file and the "measurement" of a unit is a cryptographic hash of the file. The first layer of code—the BIOS—has additional code for:

1. Measuring itself (by hashing BIOS)
2. Measuring the next layer (the boot loader)
3. Reporting measurements to a trusted module (TPM chip) and
4. Passing control to the second layer

The second layer measures and loads the third layer (typically the operating system (OS) kernel), and provides the measurement to the TPM before control is passed to the third layer, and so on.

The TPM merely requires the ability to securely store the measurements in its PCRs, and report the measurements to any entity on request. Entities interacting with the platform can choose to abandon the interaction if the reported measurements differ from expected values (which can happen if any code with uncertified measure was loaded).

In the event that the hash of the BIOS reported by the TPM is the same as the expected value, as long as it is reasonable to assume that the BIOS cannot be modified and that some essential hardware is trustworthy, it can be concluded that only the verified BIOS took control of the platform and loaded the second layer and reported the correct measurement of the second layer to the TPM. Similarly, as long as the hash of the second layer is an expected value, we can expect the second layer to report the correct measurement of the third layer, and so on.

10.2.2 Pitfalls of the TCG Approach

Almost every security issue [109–110] in the TCG–TPM approach stems from our inability to trust the measurement infrastructure. Specifically, in trusting that the measurements reported by a TPM is a correct indication of the actual state of the platform (actual software bits that have taken control of the platform since the last reboot of the platform), we are required to trust:

1. A "trusted" infrastructure to actually *verify the integrity* of complex software.
2. That software cannot be illegally modified *after* it is loaded.
3. The *binding* between the platform and the TPM.

The ability to verify the integrity of complex software implies the ability to rule out the possibility of hidden malicious functionality in software, for example, functionality which can load some illegal software component while reporting the hash of a good/permitted component to the TPM. In such a scenario, the integrity of any software that is loaded after the component with hidden malicious functionality is loaded is questionable.

The well-known time-of-use-time-of-check (TOCTOU) [111] problem in the TCG model is a result of the fact that there are a variety of ways in which a code, which has been measured and loaded, could be modified before it is actually executed. Consequently, the state reported by the TPM may not correspond to the actual state of the platform.

Finally, in trusting that a report from a TPM, X corresponds to a measurement from a specific platform that entails a verifiable binding between the RTM of the platform and the RTS/RTR of the TPM. In practice, during every reboot, the platform establishes a private channel to the TPM chip housed in the platform. From the perspective of the TPM, any external entity that invokes the first TPM command after reboot is assumed to be the platform. In practice, any entity can invoke this command and report expected values to the TPM while actually loading and executing entirely unrelated software components.

Several approaches have been proposed to address different issues in the TCG approach. To address the problem of inability to verify large code bases, approaches like Next-Generation Secure Computing Base (NGSCB) [112] and Terra [113] attempt to remove large chunks of code from the chain of trust. More specifically, the OS is removed from the chain of trust by employing virtual machines (VMs) [114, 115] to provide isolation from the core OS. The VMs may include a stripped-down version of a generic security-enhanced OS to cater for the needs of the specific application.

Approaches that further reduce the size of the trusted code base employ late-launch features [116, 117] made possible by additional instructions supported by some Intel and AMD processors along with the dynamic root of trust measurement (DRTM) [118] feature in the TPM 1.2 specification. In such approaches, small pieces of code could be "late-launched" and run unmolested irrespective of the state of the platform preceding the late launch. Some attacks against this feature have been discovered [119].

10.3 Trinc

A trinket [108] is a hardware module following the *Trinc* specification, which attests the value of monotonic counters stored inside the module. Trinc is not affected by the three issues above that plague TPMs, as no assumptions are made regarding the integrity of software, and no hardware binding is assumed between a subsystem/computer and the associated trinket.

Every trinket has a unique identity, and a key pair bound to the identity. A primary counter of a trinket is leveraged to create a plurality of secondary counters—whenever a new counter is created, it is identified by the current value of the primary counter (which is incremented on creation of the new counter). The functions exposed by a trinket can be used to:

1. Request a trinket to create a new counter with identity n, or
2. In a scenario, where the secondary counter n is currently associated with counter value c_n, an arbitrary value x can be bound to the counter n along with a new counter value $c'_n \geq c_n$.

3. Attest the value of a counter (along with a value bound to the counter); typically this is achieved by providing a signed certificate binding values n, c_n, the value x associated with counter n with the value of the primary counter.
4. Attest the value of a counter using a group secret instead of a digital signature.
5. Verify the integrity of a counter certificate authenticated using a shared secret.

As an example, we shall consider once again a simple remote file system. A typical client–server model of file storage system includes clients (users) who create files or data blobs and a server with access to plentiful storage. Users upload files to the centralized server for later retrieval from any location. Files can be edited and reuploaded any number of times. As a user may employ different client machines at different locations (work, home, on the road), and as the files in different client machines may not be synchronized (except through the untrusted server), users have to trust the server to provide the latest version of the file.

Assume that users upload files along with a signature to attest the file (for example, the signature computed over the file hash). Thus, any person receiving the file can convince themselves of the integrity of the file. If the creator of the file changes the file at a later time, the new file along with a new signature may be uploaded to the file storage system. Unfortunately, there is nothing that prevents the file server from continuing to serve the old file. After all, the old file still has a valid signature from the creator of the file.

This is one scenario where a trinket can help. Assume that a trinket with identity G is expected to assure the freshness of files stored in the remote file storage system. The signature for the file is now computed over a file hash, the trinket identity G, and a counter identity n of trinket G. The owner requests the trinket G to bind a value x and counter value c_n to the counter number n. Anyone receiving the file can request trinket G to attest the values x, n, c_n against the current primary counter.

Whenever the file is updated, the owner of the file ensures that that a fresh x' is bound to counter n with a $c'_n > c_n$. From this moment onwards, the trinket will not attest the old x and c_n values. Thus, the file server cannot replay older versions of updated files, as the current counter value c'_n is no longer consistent with the old x.

The Trinc specification limits the number of counters that can be "remembered" by a Trinc to a small queue length. To reduce the overhead for digital signatures, shared secrets between trinkets can be used to sign and verify trinket attestations. In this case, an additional system-specific trusted third party is required to set up shared secrets bound to specific counters of different trinkets.

Trinc by itself does not offer an explicit mechanism for binding a Trinc identity to a specific subsystem or binding a specific piece of data[3] associated with the subsystem to a specific counter in a specific trinket. For example, in the case of the remote file storage system, users had to sign their files. Implicit in this assumption is that some additional infrastructure like PKI is required for this purpose. Security solutions that leverage Trinc will, therefore, need to rely on other system-specific trusted

[3] For example, a file hash, or signature, or current balance, or a DNS record, etc.

parties to provide various trusted services like distributing symmetric keys, binding subsystems and Trinc identities, and binding specific counters of specific trinkets to system-specific data records, etc.

10.3.1 Virtual Counters

In [120], the authors suggest a strategy for enhancing the ability of current TPM chips, by including new TPM functionality to maintain a Merkle hash tree. Such additional functionality can then be leveraged for securing a wide variety of applications.

Once again, let us consider a remote file storage system. In the virtual counters approach, a trusted module **T** (enhanced TPM) associated with the untrusted file server **U** is intended to eliminate the need to blindly trust the server.

A Merkle tree is used to virtually store a large number of counters $c_0 \ldots c_N$. Each counter is cryptographically bound to a file—say uniquely identified by values $id \parallel l$, where id is the unique identity of a user and l is a label assigned to the file by the user. Every time a file is changed the corresponding counter is incremented and the root is updated.

Even the root of the tree is not stored inside the module. Instead, only a nonresettable monotonic counter is stored inside the module, and the root is cryptographically bound to the counter. The reason for this choice is that the latest TPM specification already includes such a nonresettable monotonic counter.

If the root is r when the primary counter is c, the module issues a certificate binding values r and c. Similarly, a file $id \parallel l$ with file-hash $h_{id,l}$ is bound to the ith counter c_i though the certificate issued by the module. Every time a file is updated the counter bound to the file hash is incremented causing the root r to change. Every time the root r changes the counter c is incremented. New certificates are issued by **T** to (1) bind the updated root and the counter; and (2) bind the incremented c_i to the new file hash.

To send the current version of the file to a user, the server **U** requests the module to authenticate the hash of the current file by submitting the following values:

1. $i \parallel c_i \parallel id \parallel l \parallel h_{id,l}$
2. The certificates binding the file hash to a counter c_i and certificate binding r and primary c, and
3. Verification object (VO) to map c_i to the root.

The module can now attest the values $id \parallel l \parallel h_{id,l}$ to the user.

A counter c_i bound to file $id \parallel l$ will be updated by the module only if an authenticated hash $h'_{id,l}$ (authenticated by the owner id) is provided to the module. Apart from the authenticated hash, the module will be provided inputs ($i \parallel c_i \parallel id \parallel l \parallel h_{id,l}$; two certificates; and the VO required to verify the current leaf against the root. To complete this update the module:

1. Increments c_i to $c'_i = c_i + 1$
2. Updates the root of the tree using the VO

3. Increments the counter c to $c' = c + 1$
4. Issues a certificate binding values r' and c'
5. Issues a certificate binding c_i' with the updated hash $h_{id,l}'$ and
6. Outputs an certificate to the effect that the "update has been carried out"

It would seem reasonable at first sight to assume that when a user has received a confirmation from the module **T** that the "update has been carried out," the server can no longer replay older versions of the file—as the module will not authenticate the old hash. Specifically, after the update to the root, the module will not recognize the hash of older version as authentic. Unfortunately, this is not true.

The security loophole in the virtual counter approach is that there is no way to prevent the server from binding the same file $id \parallel l$ to multiple (say two) counters c_u and c_v. Now, following the first update, the server instructs the module to update counter c_u and issue a confirmation to the user, and deliberately leaves counter c_v intact. The old file corresponding to counter c_v can be replayed by the server as c_v is still a part of the tree. More generally, the server may associate a file with any number of counters (leaves) and maintain different older updates in such leaves.

A second security pitfall of the virtual counter approach is the inability to provide authenticated denial. Consider a scenario where a user submits a request for a nonexistent file $id \parallel l$. As no verifiable leaf bound to $id \parallel l$ exists and the module can only make reliable statements about the leaves of the tree, the module cannot authenticate a denial which conveys the nonexistence of $id \parallel l$. In such a scenario, the untrusted server is implicitly trusted to convey nonexistence. The unfortunate side effect is that the untrusted server can abuse this privilege by "conveying nonexistence" of files that *do* exist.

At the core of *both* problems is the inability of the module to verify nonexistence of leaves. If the server can easily verify that *no* counter has been bound to file $id \parallel l$ then the module will permit binding of $id \parallel l$ to a counter c_i, only if the module can verify that $id \parallel l$ does not currently belong to the tree. In such a scenario, the server cannot force the module to bind $id \parallel l$ to multiple leaves. Similarly, when queried for a nonexistent file, a module which can easily verify the nonexistence can provide authenticated denial, thus eliminating the need to trust the server. Obviously, using an object modeling technique (OMT) instead of a Merkle tree can address both pitfalls of the virtual counter approach.

10.4 Credential Management Modules

Strategies to secure any practical system using Trinc or TPM will almost always (unfortunately) require components *other*[4] than the Trinc/TPM module to be trusted.

[4] Intangible assumptions regarding the integrity of such "other" components can turn out to be incorrect.

Security solutions that leverage TPMs implicitly trust an entity/organization responsible for verifying the integrity of every bit of code that can gain control of the platform. The motherboard assemblers have to be trusted to ensure that the TPM chip cannot be easily removed (without destroying the chip). The BIOS is assumed to be trusted as modifications to the BIOS can result in the BIOS reporting expected values to the TPM while ceding control to unauthorized software.

Security solutions that leverage Trinc will need to rely on other system-specific trusted parties to provide various trusted services like distributing symmetric keys, binding subsystems and Trinc identities, and binding specific counters of specific trinkets to system-specific data records, etc.

The need for additional system-specific trusted parties for a system (say) X poses two types of security risks. First, the rationale for the trust and the precise consequences of misplaced trust in such entities, may not be well understood by designers of other systems that rely on system X (or on some information provided by system X). Second, a system-specific trusted entity is far less likely to be vetted thoroughly (compared to global entities responsible for certifying TPM/Trinc).

Both TPM and Trinc can be seen as *nonconsummate* approaches, as they merely attempt to replace *some* intangible assumptions with the assumption of integrity of trusted module functionality ($\mathcal{F}_{\text{tpm}}/\mathcal{F}_{\text{trinc}}$).

In the rest of this chapter, we outline broad features for an alternate trusted module specification intended to eliminate all intangible assumptions. For reasons that will be clear soon, such trusted modules are called credential management modules (CMM).

10.4.1 Credential Transaction Model

Consider a broad model for systems where any system is seen as "an interconnection of subsystems." Each subsystem has a well-defined role and is associated with a set of dynamic credentials.

For example, subsystems in a financial system have roles like buyer, seller, financial institution, regulator, etc.; subsystems in the domain name system (DNS) [18], [19] have roles like domain name registry, zone owners, DNS servers/resolvers, etc.; roles of subsystems in the Internet's border gateway patrol (BGP) inter-domain routing system [122] include BGP speakers (in BGP routers), autonomous system (AS) owners, AS registry, IP registry and registrars, etc.

To execute role-specific tasks, every subsystem possesses a set of credentials and is permitted to perform specific transactions. In general, a transaction between two entities results in a change in the credentials of both entities. While credentials are dynamic, the rules that govern credential transactions are static. Both credentials and rule are, however, system specific.

As some examples to illustrate the application-dependent nature of credentials and rules for transacting credentials, we shall consider the DNS, BGP, and a geographic information system (GIS).

10.4.1.1 DNS Credentials

In DNS, a credential associated with a zone owner A is the name x of the owned zone. The owner is permitted to:

1. Create new subzones like $w.x$ (names of subzones of x end with x), and
2. Assume ownership or delegate the newly created subzone.

DNS resource records can also be seen as credentials created by a zone owner, where for any owned name (say, $w.x$ or $u.w.x$) the owner A has the freedom to specify two quantities—a record type and a value.

Such credentials are then issued to DNS servers for the zone, as a way of authorizing them to readvertise such credentials to other DNS servers/resolvers. Once a subzone $w.x$ has been delegated by the owner of x (A) to (say) B, the credential of B is modified, as it gains ownership of $w.x$. The credential of A is also modified, as it can no longer create resource records for names ending with $w.x$. The root zone can be seen as the original owner of the entire name space that delegates names to owners of top-level domains (TLD).

10.4.1.2 BGP Credentials

The Internet is an interconnection of autonomous systems (AS) [122]. Each AS owns one or more chunks of the IP address space, where the number of addresses in each chunk is a power of 2. IP chunks are represented using the classless interdomain routing (CIDR) IP prefix notation. For example, the IP prefix 132.5.6.0/25 represents 2^{32-25} IP addresses for which the first 25 bits are the same as the address 132.5.6.0, viz., addresses 132.5.6.0 to 132.5.6.127. An AS registry assigns AS numbers to AS owners. AS owners may acquire ownership of IP prefixes from an IP registry (through IP registrars or ISPs).

While each AS may follow any protocol for routing IP packets within their AS, all ASes need to follow a uniform protocol for inter-AS routing. The current inter-AS protocol is the BGP, where AS owners employ one or more BGP speakers to advertise reachability information for IP prefixes owned by the AS. Specifically, every BGP speaker recognizes a set of neighboring BGP speakers. Neighbors may belong to the same AS or a different AS. The main responsibility of BGP speakers are as follows:

1. Originate BGP update messages for prefixes owned by the AS, and convey such originated messages to neighbors of other ASes
2. Relay BGP update messages received from neighbors to other neighbors
3. Aggregate prefixes for reducing the size of routing tables

BGP is a path vector protocol. BGP update messages communicated between BGP speakers indicate an AS path vector for an IP prefix and a BGP weight for the path. The weight for a path is influenced by the length of the AS path and various parameters like "local preference" and "multiple exit descriptor" specified by the AS owner for each neighboring speaker (from which a BGP speaker can receive BGP update messages or to whom update messages can be sent).

The IP registry can be seen as the original owner of the entire space of IPv4 addresses. The registry delegates[5] chunks of consecutive IP addresses to AS owners. AS owners in turn can delegate "subchunks" of addresses to a BGP speaker for the AS. A speaker that has been delegated a prefix is allowed to initiate path vectors for the prefix, by creating a BGP update message (which is then sent to all neighboring speakers). A BGP speaker receiving a BGP update message (from a neighbor) for a prefix X can insert its AS number in the path and readvertise the path for X to all its neighbors. However, only the best path for prefix X can be relayed.

Credential transactions between registries and AS owners include AS numbers and IP prefixes. Credentials provided by an AS owner to a BGP speaker include part or the entire owned prefix, list of neighbors, and BGP weights associated with each neighbor. The credentials provided to BGP speaker influence the process of creation and relay of BGP update messages. BGP update messages can themselves be seen as a credential created as a function of several credentials.

10.4.1.3 GIS Credentials

In any GIS-based application, the entire surface of the earth may be represented as tessellations bounded by specific lat–long coordinates. Tessellations may be delegated to various owners (subsystems), who in turn may split a tessellation into smaller tessellations for purposes of delegation to other subsystems.

The credentials associated with a subsystem can include lat–long enclosures of tessellations owned and the types of services the subsystem is authorized to advertise. For each type of service (say, service G = "gas station"), a special subsystem may assign such a credential (of participant in a service G or "gas station owner"). Subsystems will be allowed to advertise the exact location of different services (for example, a gas station, a restaurant) only as long as:

1. The advertised location of the service falls within an owned tessellation.
2. They possess the credential to advertise the service (for example, "gas station owner").

The information from such systems may serve as inputs to other complex systems that may utilize such information for providing a wide range of useful services.

In general, across different systems, credentials can take substantially different forms that may include simple scalar values like name, address, current balance, time of expiry, file hash, a secret, etc., or more complex functions of a plurality of scalar values like the cryptographic hash of multiple scalar values, a range of latitude and longitude coordinates, a range of addresses, an access control list, a revocation

[5] In practice the delegation may occur in a hierarchical manner, through multiple levels of registrars and ISPs.

list, a path vector, a look-up table, etc. The versatility of OMTs and the integrity of OMTs can be assured even by resource-limited modules like CMMs, which make them well suited as a strategy for representing dynamic credentials.

10.4.2 Consequential Transactions

In a credential transaction model for any system

1. Interactions between subsystems are *credential transactions*. A transaction results in the modification of the credentials of both subsystems, in accordance with a system-specific *rule*.
2. A transaction is *consequential* if failure to adhere to the credential transaction rule can lead to the violation of the desired assurances.

Under this model, securing any system simply boils down to *ensuring the integrity of all consequential credential transactions*.

This process begins with the explicit enumeration of various properties of the system, like different subsystem roles, types of credentials, and permitted transactions $T_1 \ldots T_n$. Any subsystem that is required to perform a consequential transaction is associated with a trusted CMM that vouches for the integrity of credential transactions performed by the subsystem.

10.4.3 Virtual Networks

Alongside the network of subsystems (say, $A, B, \ldots$) that form a system X, the CMMs associated with each subsystem (say, $A', B', \ldots$) form a parallel *virtual network* (VN) X'. Credential transactions between two (untrusted) subsystems A and B are mirrored by exchange of *VN messages* between the trusted CMMs A' and B' in the VN X', to authorize the transaction.

The identity X' of a VN created to secure a system X is simply a cryptographic commitment to all explicitly enumerated properties $\mathbf{T}_X$ (roles, credentials, permitted credential transactions, system-specific constants, etc.) of system X. The explicitly enumerated properties $\mathbf{T}_X$ can also be seen as a specification of a system-specific security protocol $\mathcal{S}_X : \mathcal{F} \rightarrow \mathcal{D}_X$ that translates the universal assumption (of integrity of) $\mathcal{F}$ to the desired system-specific assurances $\mathcal{D}_X$.

From the perspective of CMMs, the enumerated properties $\mathbf{T}_X$ (or the security protocol $\mathcal{S}_X$) serve as unambiguous instructions for the verification of integrity of system-X transactions. Subsystems are expected to maintain an OMT to represent system-specific credentials. The corresponding OMT root is tracked by a CMM associated with the subsystem.

The tasks performed by CMMs can be broadly classified into:

1. Tasks for managing VN
2. Tasks for operating in a VN

Managing VNs includes tasks like creating a VN, inducting other CMMs as members of the VN, joining a VN, etc.

A CMM can create any number of VNs and take part in any number of VNs. For example, a CMM can be the creator of 5 VNs and a member of 20 other VNs. Members of a VN may have different roles. For example, a VN with a million members may have, as its members, only four members with the role of a server and all other members with the role of a client. VNs can be loaded, unloaded, and deleted. When operating in a VN X, the CMM assumes the personality of a VN member in a VN currently loaded on to the CMM.

Creating a VN will involve constructing a static OMT (say $\mathbf{T}_X$, with root X) whose leaves specify the security protocol for the system. The root of the static OMT is the VN identity. The VN rules also explicitly specify the identity of one CMM, which is regarded as the creator of the CMM. The designer of the VN will typically designate a CMM under his/her control (or physical possession) as the creator of the VN. The CMM itself does not care *how* the rules are constructed, and the purpose of such rules. All that a CMM "knows" is when it is operating in a VN X, it will abide by any rule (elucidated in an OMT leaf) in an OMT with root X.

The VN designer can request the creator (CMM) to induct other CMMs into the VN and assign possibly different roles within the VN. The CMMs that are inducted and the member identities/roles to be assigned to the inducted VN members are entirely up to the discretion of the possessor of the VN creator. For purposes of inducting CMMs into VNs (and for joining VNs), the interfaces exposed by CMMs will facilitate establishment of shared keys between VN members (CMMs that have joined the VN).

Such shared keys are used for authentication of VN messages exchanged between VN members while operating in the VN. During regular operation in a VN, CMM interfaces will enable the subsystem to insert/delete OMT leaves, swap OMT leaves, create VN messages, and update a VN leaf (modify the third field) subject to VN-specific rules (consistent with VN identity X).

10.4.4 VN State Changes

From the perspective of the CMM, irrespective of the specifics of the system or the nature of the credentials represented by an OMT, the subsystem is allowed to insert leaves or swap leaves in any OMT maintained by the subsystem. Corresponding to such changes, the CMM will readily modify the OMT root. More specifically, the rules that dictate *how* the CMM should modify its root to account for insertion, swapping, or modification of a leaf, or how the CMM should verify the integrity of a leaf against the root, are *not* application specific.

However, for creating a VN message or updating an OMT leaf, the CMM will expect as input, additional justification in the form of a rule consistent with the VN identity X'. Specifically, from the perspective of a CMM operating in a VN X', leaves of an OMT with root X' describe rules of the form

$$f(a, a_n, \omega_a) \rightarrow v \text{ or } g(a, a_n, \omega_a, v) = \{\omega'_a, v'\}, \tag{10.1}$$

where:

1. A transaction rule $f()$ specifies how contents (a, a_n, ω_a) of an OMT leaf can be used to generate a *VN message* $v = f(a, a_n, \omega_a)$, and
2. A transaction rule $g()$ specifies how a received VN message v can be used to update an OMT leaf (modify $(a, a_n, \omega_a) \rightarrow (a, a_n, \omega'_a)$), and (optionally) generate a VN message v'.

In other words, a transaction is an atomic operation which results in:

1. The creation of a message of a specific *type* to convey a value v; or
2. (On receipt of a message v) an update $(a, a_n, \omega_a) \rightarrow (a, a_n, \omega'_a)$ to an OMT leaf, or creation of a response VN message v', or both.

In general, the rules for updating a leaf may be different for different roles. All VN members operating in the same VN X', with the same role, follow the same set of rules.

10.4.5 CMM State and VN State

CMMs will possess protected nonvolatile storage for storing a long-lived secret χ, and a monotonic counter C_M. The two values stored in protected nonvolatile memory are used to seal and restore the integrity of the dynamic CMM *state* across power off–on cycles of the CMM.

The state of the CMM is captured by a dynamic value ξ and a CMM identity M_{id}. Before a CMM is powered off, the CMM outputs a self-certificate (MAC)

$$\mu = h(\xi \parallel M_{id} \parallel C_M \parallel \chi) \tag{10.2}$$

along with the CMM's current state ξ.

When a CMM is powered on the values ξ, M_{id}, and the certificate μ are provided as input to restore the CMM state. The monotonic counter C_M is incremented (the certificate μ for the previous state cannot be reused as the counter C_M has been incremented).

The CMM state ξ is the root of an IOMT. The leaves of the IOMT correspond to any number of VNs in which the CMM plays a role (as a creator or a member).

To load a VN, any IOMT leaf (V, V', v) consistent with ξ can be provided. The first field V is the VN identity, and the third field v is the state of the VN (Fig. 10.1).

CMMs can be seen as being in three basic states:

1. Off state, in which the values χ and C_M in nonvolatile memory are protected.
2. On state with no VN loaded. During this state, the CMM state ξ can be modified for purposes of inserting/deleting place holders (each corresponding to a different

Fig. 10.1 CMM state and VN state

$\boxed{\chi, C_\mathrm{M}}$	Non Volatile
$\boxed{\xi, M_\mathrm{id}}$	CMM State
$\boxed{(V, V', \nu)}$	State of VN V
$\boxed{(r_1, r_2, r_3, r_4, c_1, c_2, m_{id})}$	Components of VN State

VN) in the tree with root ξ (by providing a EQ certificate indicating a root equivalent to ξ).

3. On state with VN loaded. During this state, a VN leaf (V, V', ν) consistent with ξ is loaded.

The VN-state ν is a function of several parameters. Specifically, $\nu = 0$ implies that the VN V has not been initialized. For an initialized VN,

$$\nu = h(r_1 \parallel r_2 \parallel r_3 \parallel r_4 \parallel c_1 \parallel c_2 \parallel m_{id}). \tag{10.3}$$

Values $r_1 \ldots r_4$ are OMT roots. c_1 and c_2 are counters used within the VN. m_{id} is the identity of the VN member (whose VN state is ν).

Only when a initialized VN V is loaded, the CMM can assume the personality of the VN member m_{id} to operate in VN V. During this state, the VN member is permitted to:

1. Perform equivalence transformations on roots $r_1 \ldots r_4$
2. Create and process various types of VN specific messages subject to rules specified in any leaf of a static OMT with root V (VN identity)
3. Update leaves of OMTs $r_1 \ldots r_4$ and VN counters c_2, c_2 (subject to VN V rules).

Before a VN is unloaded:

1. The update value of ν is corresponding to changes to $r_1 \ldots r_4$, c_2, c_2, etc., that may have occurred when the VN was loaded
2. The CMM state ξ is updated to record the updated state of VN V

10.4.6 Changing VN State

Every subsystem (for example, A) in system S' is associated with a dynamic set of credentials ($\mathbf{C}_A$). Subsystems maintain their dynamic credentials as leaves of up to four OMTs (some or all of which may be nested). The dynamic roots of the OMTs are tracked by VN-member (CMM) m_A^S.

To engage in a transaction with each other, subsystems A and B request their respective CMMs (more specifically, the VN members m_A^S and m_B^S, respectively) to create/accept VN messages. VN messages are exchanged over VN links (Fig. 10.2).

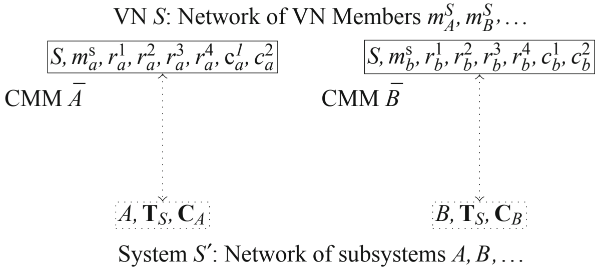

VN S: Network of VN Members $m_A^S, m_B^S, \ldots$
$S, m_a^s, r_a^1, r_a^2, r_a^3, r_a^4, c_a^1, c_a^2$
$S, m_b^s, r_b^1, r_b^2, r_b^3, r_b^4, c_b^1, c_b^2$
CMM $\overline{A}$
CMM $\overline{B}$
$A, \mathbf{T}_S, \mathbf{C}_A$
$B, \mathbf{T}_S, \mathbf{C}_B$
System S': Network of subsystems $A, B, \ldots$

Fig. 10.2 System S' and associated VN S

The broad steps that will need to be performed by two subsystems A and B (belonging to the same system, and thus subject to the same transaction rules $\mathbf{T}_S$) to perform a transaction are as follows:

1. Subsystem A requests its CMM A' to create a message $\mathbf{M}$ consistent with a rule $f()$ in the currently loaded VN.
2. CMM A' authenticates the message $\mathbf{M}$ (by providing a token μ) for verification by B'.
3. Subsystem A sends message $\mathbf{M}$ along with token μ to B.
4. B provides the message $\mathbf{M}$ and token μ to it's CMM B', along with a VN rule $g()$.
5. Depending on the contents of message $\mathbf{M}$, and rule $g()$ B' modifies one or more OMT leaves, and/or creates a response message.

As subsystems are untrusted, that a message was created by CMM A' is no guarantee that the message will be delivered by the subsystem A to B or by B to CMM B'. In other words, in the VN-link between any two CMMs, A' and B' are two untrusted middlemen A and B (the subsystems). In scenarios where it is essential to ensure that the middlemen do not drop messages, the rules for creation of messages can include "locks." For example, the rules may dictate that A' should set a lock for B', which will be removed only if an authenticated response from B' is submitted to A' within some duration τ. The specific implications of the existence of a lock may also be system dependent. As one example, a lock may result in the termination of the link, due to which A and B may not be able to transact credentials (possibly for a specific duration) in the future.

10.4.7 CMMs as ADS Constructors and Verifiers

In a system, secured using CMMs, each subsystem can be seen as an untrusted prover for an ADS. The verifier for the ADS is a trusted CMM associated with the subsystem. Specifically, the CMM is both the constructor and the verifier of the ADS.

In general, there may exist a plurality of asynchronous data sources for the ADS. The ADS is constructed by the CMM in a piecewise manner by updating the ADS when new data is supplied. The data sources are CMMs associated with other subsystems (or other CMMs belonging to the same VN). More specifically a credential transaction is a process where data supplied by a CMM are accepted by another CMM to update the ADS.

In general, the rules for initializing the ADSs corresponding to each member of the VN, and the rules for creating data (supplied to other CMMs) are dictated by VN rules. Specifically, rules of the form $f()$ dictate how data, in the form of a specific type of VN message can be created for delivery to another CMM. The rule $g()$ dictates how the received data, in the form of a VN message from another CMM, should be used for updating the ADS.

Through this ability to act as trusted sources and verifiers of application-dependent data. the CMMs forming the VN guarantee that subsystems need not be trusted. Untrusted subsystems:

1. Actually store the database and
2. Identify VOs for purposes of verification and construction of the ADS by the CMM.

The choice of OMT as the ADS is due to its versatility. To simplify the application-specific representation of rules, the CMM model provides the untrusted subsystems with the ability to make *inconsequential* updates to the OMT. Specifically, such inconsequential updates may be performed for the purposes of swapping OMT leaves, or inserting/deleting placeholders. Simplified application-specific rules for updating OMT leaves have been investigated for a wide range of practical systems like remote file storage systems [123], mobile ad hoc networks [124, 125], supervisory control and data acquisition (SCADA) systems that control critical infrastructures, generic data dissemination [129] and encrypted content dissemination [128] systems, etc.

10.5 CMM System Architecture

Similar to existing trusted module specifications like TPM, Trinc, etc., the CMM specification can be open one—or CMMs can be manufactured by any entity. Like any trusted module specification, CMMs will include explicit functionality for achieving tamper-responsiveness, true random sequence generators (for example, using physical unclonable functions), self-check functions, circuitry for protecting the integrity of clock frequencies, etc.

Similar to TPMs/Trincs, CMMs will generate an asymmetric key pair; CMMs will have a few tens of reserved registers, modest (for example, a few kB) scratch pad memory, and I/O registers. CMMs will be equipped with a hardware pseudo-random function block (for example, a standard cryptographic compression function $h()$ like SHA-1, or the newer standard SHA-3) as almost every task performed by the CMM will involve repeated use of $h()$. CMMs will possess logic circuits necessary to realize functions $\mathcal{F}_{\text{cmm}}$; the contents of nonvolatile memory are assumed to be protected by tamper-responsive circuitry even when the CMM is powered off; the integrity of volatile memory regions and the CMM clock are assured only when the CMM is powered on.

Bootstrapping a CMM is a process where a CMM is inducted to operate in a specific "CMM universe" by a trusted authority for the universe by attesting the public key of the CMM. CMMs in the same universe can form VNs. Specifically, corresponding to a network of consequential subsystems of a system to be secured, a parallel VN is formed by the CMMs associated with each subsystem. A CMM can belong only to one universe. Within the universe it can create or join any number of VNs.

10.5.1 CMM Universe

Associated with a universe $\mathcal{U}$ is a trusted infrastructure $\mathcal{T}_{\mathcal{U}}$. $\mathcal{T}_{\mathcal{U}}$ constituted by n independent entities deemed responsible for verification and certification of CMMs inducted into universe $\mathcal{U}$, and m entities that issue symmetric keys to each certified CMM. Such symmetric keys render the CMM eligible to take part in the universe $\mathcal{U}$.

Each of the $m+n$ entities is associated with a CMM: say $V_1 \dots V_n$ and $Y_1 \dots Y_m$, respectively. To induct a CMM F with public key U_F, at least $n' \leq n$ CMMs $V_1 \dots V_n$ should convey the attested public key U_F and an assigned identity F to each KDC CMM $Y_1 \dots Y_m$.

From the perspective of a KDC (CMM Y_i), a CMM assigned identity F and associated with public key U_F, which has been attested by at least $n' \leq n$ verification CMMs, is eligible to receive a secret generated by the KDC (CMM Y_i).

Unlike TPM and Trinc, the asymmetric key pair of CMM F is used sparingly—only for securely receiving a secret from each KDC for the universe. Each CMM inducted into the universe receives one secret from each KDC. The identities $V_1 \dots V_n$ and $Y_1 \dots Y_m$ of CMMs that induct CMMs into the universe $\mathcal{U}$ are assumed to be of public knowledge. Their identities are one-way functions of their respective public keys. Also n (the number of verification agencies for the universe) and m (the number of KDCs for the universe) are values of public knowledge . For example, one such universe may choose $n = 6, n' = 4, m = 4$, etc.

10.5.2 Creation of Virtual Networks

CMMs in the same universe can form VNs. Consider a system constituted by untrusted subsystems $(A, B, \dots)$. Let O be stakeholder/owner of the system; let O' be the identity of a CMM associated with the stakeholder. The broad steps involved in deploying CMM-based security for the system are as follows:

1. The stakeholder O enumerates all credential transaction rules for the system S as leaves of a static OMT $\mathbf{T}_S$ (with any number of leaves) with root S. S becomes

the identity of the VN to be created to mirror system. The contents of static OMT $\mathbf{T}_S$ will include values like:

(a) The identity O' of the CMM creating the VN
(b) Different roles
(c) Types of messages
(d) Types of credentials
(e) Parameters of $f()$ for creating messages and
(f) Parameters of $g()$ for modifying OMT leaves based on a received message, etc.

2. Stakeholders can now request the CMM to issue any number of NV certificates to bind any leaf of the static OMT $\mathbf{T}_S$ to its static root S.
3. Stakeholder inserts a place holder index S in the IOMT whose root ξ captures the CMM state.
4. Stakeholder invokes a function to initialize VN by loading the place holder for S. The CMM can be provided a NV certificate to demonstrate that O' is the creator of VN S (as a leaf in the tree with root S will specify the VN creator identity).
5. One leaf of the rules OMT will specify the starting roots $r_1 \cdots r_4$ and counter c_1, c_2 and a fixed m_{id} (say, $m_{id} = 0$) for the VN creator. The CMM can thus be requested to set the values accordingly. After this point, whenever the VN is unloaded the new values will be sealed against the CMM state ξ.
6. After the CMM O' has been initialized as the VN creator, CMM O' can now induct other CMMs into the VN. When a CMM X is inducted into a VN S, the stakeholder in possession of O decides the role of the member and the member identity. It is convenient to have a few bits of the member identity reserved to indicate one of several possible roles within the VN. Let m_x be the member identity to be assigned to CMM X.

For the creator of the VN S, the IOMT with root r_1 has leaves which indicate the member identity as index and the CMM identity as the third field. The stakeholder can request the CMM to add place holder for m_x. The root r_2 corresponds to a tree in which the leaves have CMM identity as index and member identity as the third field. CMM O is requested to insert a place holder for X in the tree with root r_2.

One leaf in the static rules tree will indicate the initial values $r_1 \ldots r_4$ and counters c_1, c_2 for members. Specifically, one such leaf will exist for different membership roles within the VN. Such a leaf can be provided to request O' to create an authenticated message to CMM X to induct X into the VN.

Before creating such a message, the CMM will update the place holders (for m_x in the first IOMT and X in the second IOMT). This ensures that:

1. The member identity m_x cannot be assigned to any other member
2. CMM X cannot receive two memberships in the same VN

The induction message will be authenticated by using inter-CMM pairwise secret $K_{O'X}$ facilitated by the KDCs of the CMM universe to which both O' and X belong. The authenticated message will convey all initial parameters for m_x.

The entity in possession of CMM X requests X to insert a placeholder for VN S, and load the uninitialized VN. The induction message is submitted to the CMM to modify its VN state. When the VN S is unloaded, the CMM state will be updated to reflect the modifications to the VN state.

10.5.3 Intra-VN Key Distribution

Pairwise keys between CMMs are used only for induction of CMMs into VNs. When operating in a VN, messages between VN members are authenticated using intra-VN secrets. The VN rules specify the identities of CMMs that will serve as KDCs for the VN.

CMMs will expose functions which can be used to request the creator of a VN to prepare message instructing a KDC CMM (say Y) for the VN to issue secrets to a CMM (say X) corresponding to a VN member identity m_x. The VN rules will also specify the type of key distribution scheme (MLS/SKIT/PBK, etc.) and parameters like m, l associated with the key distribution scheme.

Such a certificate can be submitted to KDC Y which will then prepare a message to convey secret(s) to CMM X (to be used when CMM X assumes the personality of VN member m_x).

For small VNs, the creator CMM itself may serve as the KDC.

10.5.4 VN Links

Armed with intra-VN secrets, a VN member can now exchange VN-specific messages with other VN members. Any number of message types may be specified by the VN rules, along with the rules $f()$ indicating how a message of a specific type can be created.

A message created by a VN member m_j and sent to m_j is authenticated using a MAC μ computed using intra-VN secrets shared between m_i and m_j. All messages however have the same format, viz.,

$$[y \parallel m_{id} \parallel v \parallel t \parallel t'] \tag{10.4}$$

where:

1. y is the message type.
2. v is a value conveyed by the message.
3. t is the clock tick value at the time the message was created.
4. t' is the clock tick time indicated in a previous message from m_j (for which this message is an acknowledgement).

In general, VN messages will be sent over established over VN links. A VN link is created by exchanging $HELLO$ messages between VN members. Messages of type

$HELLO$ are not specific to any VN, and the creation is not subject to VN specific rules. In a $HELLO$, no VN-specific value is conveyed (or $v = 0$). Only the clock tick counts are used. This can be useful in scenarios where it is necessary for some amount of time synchronization between VN members.

For example, an exchange:

$$\begin{array}{ll} m_i \rightarrow m_j & t_i, 0 \\ m_j \rightarrow m_i & t_j, t_i \end{array} \tag{10.5}$$

where an immediate $HELLO$ response is sent by m_j to a $HELLO$ message from m_i permits m_i to detect the offset of the clocks between the two CMMs.

Assume that the response is submitted to m_i when the clock tick count is t_i', and that the round trip time $\delta = t_i' - t_i$ is sufficiently small. The best estimate of the time (according to clock of m_i) t_j is $(t_i + t_i')/2$. Thus, the offset between the clocks of m_i and m_j is $t_j - (t_i + t_i')/2$.

The estimation of offsets may be necessary for application scene, it is necessary to impose validity durations on credentials. Depending on the extent of accuracy required the VN rules may specify the maximum round trip time δ permitted (as the maximum error in the estimate is δ) to compute the offset.

In general, for most VNs, it may be useful to use one IOMT to maintain a VN link table. Each leaf in the IOMT is indexed by the identify of the VN member at the other end of the link. The third value corresponding to a member m_j may be a function of several parameters like:

1. o: the clock offset
2. t_h: the time at which m_j was last heard from
3. K': the (encrypted) pairwise secret shared with m_j, etc.
4. t_l: the time at which an as yet unacknowledged message was sent to the m_j.

The field t_l may be reset to zero when the expected acknowledgement is received. The VN rules may specify the maximum duration allowed for a response, and the action to be taken if t_l is not reset. For example, the VN rules may specify that no VN messages can be sent to member m_j until time $t_l + \Delta$, after which t_l may be set to zero.

10.6 Credential Transaction Model of Representative Systems

The broad purpose of the credential transaction model is to specify rules for credential transactions in a manner that can be readily understood by resource-limited CMMs. From the perspective of CMMs (or VN members), leaves of a static OMT $\mathbf{T}_S$ contain simple instructions for generating messages of a specific type, or for updating IOMT/DOMT leaves.

The leaves of $\mathbf{T}_S$ will explicitly specify various system parameters like subsystem roles; types of credentials, how they should be represented (IOMT or a DOMT); message types; the relationship between contents of a specific message type and the contents of one or more OMT leaves (specified as parameters of a function $f()$); types of messages that can be exchanged between specific VN members (depending on their respective roles); if locking is necessary for assured delivery of messages; influence of a received message on one or more OMT leaves (parameters of a function $g()$); any number of system-specific constants; CMM identity of the creator of the VN; identities of CMMs that will play the role of KDCs for the VN, etc.

10.6.1 Credential Transaction Model for DNS

The domain name system [18] is a hierarchical name space. At the top is the root zone which is allowed to create TLDs—both generic TLDs like `com, edu,` etc., and country code TLDs like `in, fr, ca, tv,` etc. The root zone delegates TLD names to TLD owners. A TLD like `com` can create names like `abc.com`. Names created by TLDs are typically delegated to a zone owner specified by a DNS registrar. A zone owner of (say) `abc.com` can create any number of names that end with `abc.com`. Zone owner either assumes ownership of such names or delegates some such names to other zone owners. For instance, the owner of `abc.com` may assign ownership of names like `x.abc.com` and `x.y.abc.com` to itself, and delegate a name like `go.abc.com` to another entity (which then becomes the owner of the delegated zone).

The owner of a zone can create any *type* of DNS record for a name that it owns. DNS records for a zone are grouped into resource record sets (RRSETs) of DNS records for the same name and type. All RRSETs for a zone are hosted by DNS servers that are deemed authoritative for the zone (by creating name server or NS type records). For example, the owner of `abc.com` creates NS records indicating the authoritative name servers for the zone `abc.com`; the NS records are, however, hosted by the name server authoritative for the parent zone `com`. DNS servers may be queried either directly or indirectly through other DNS servers (like local DNS servers) for DNS records by specifying the name and type.

Some of the desired assurances regarding DNS are as follows:

1. A zone cannot be delegated to multiple owners.
2. A zone is *not* allowed to create DNS records for the names that it has delegated.
3. DNS servers should not modify the DNS records.
4. DNS servers should not be able to incorrectly deny the existence of an RRSET (queried by specifying name and type) [23].
5. In the process of demonstrating nonexistence of a queried name and type, no unsolicited information should be provided (to avoid attacks like DNS-walk [24], [25]).

Assurances (1, 2, and 5) are *not* provided by the current standard (DNSSEC) for securing DNS. While TCB DNS provides assurances 3, 4, and 5, assurances 1 and 2 are not provided. In the rest of this section, we outline a credential transaction model that caters for all five assurances.

10.6.1.1 Roles

We can readily identify five different subsystem roles for DNS participants:

1. Owner of root zone
2. DNS registrars
3. Owner of TLDs
4. Owner of zones (other than TLD or root zone) and
5. DNS servers

Let R be a VN member with the role of the root zone owner; G a VN member with the role of a DNS registrar; Z, a zone owner (of any zone); and D, with the role of a DNS server. The identity assigned to a VN member will have reserved bits (for example, the least significant byte of the VN identity) that explicitly specifies the role of the VN member.

10.6.1.2 Credentials

Credentials in this system include names, record types, and a value (that represents an RRSET) associated with a name and type. More specifically, the credentials are:

1. Hashes of DNS names
2. Hashes of name and type and
3. Hash of an RRSET for a name and type

A constant Φ is defined as the credential corresponding to the root zone; a constant δ corresponds to record type NS.

The credential y corresponding to a TLD name n_1 is obtained by hash-extending Φ as $y = h(\Phi \parallel h(n_1))$.

A child $n_2.n_1$ of n_1 corresponds to $w = h(y \parallel h(n_2))$, and so on.

Corresponding to a specific name credential ν and and type credential p the name-and-type credential is $h(\nu \parallel p)$.

10.6.1.3 OMT Types

All DNS credentials are represented using an IOMT. The index of the leaves are names. The value (third field ω) is the owner of the name. The IOMT maintained by a root owner R is initialized with a single leaf

$$(\Phi, \Phi, R). \tag{10.6}$$

In the IOMT maintained by a TLD owner T, the leaf indexes can be:

1. TLD names owned by itself following delegation by the root zone (for such leaves $\omega = T$).
2. Names derived by hash extending a TLD name. The zone owners to whom the derived name is delegated is the value field.

3. Name and type values where the name is a TLD name and the type corresponds to name-server (NS) records (the value is the hash of the RRSET for NS records to be provided to NSs for the root domain).

Two differences between TLDs and other zone owners are:

1. TLDs cannot create records of type other than NS, while zone owners can create any type of record.
2. TLDs require an authorization from a registrar to delegate a name. Zone owners do not.

In the IOMT maintained by a zone owner Z, the leaf indexes can be:

1. Any name delegated by a TLD owner (or another zone owner)
2. Names derived by hash extending an owned name
3. Name and type derived by hash-extending any owned name

For names received as delegations, the value is set to itself (Z). For names derived by hash extending an owned name, the value can be set to itself (if the name is not going to be delegated) or to another zone owner (if the name is to be delegated). Corresponding to name-and-type indexes the value is a the hash of an RRSET.

In the IOMT maintained by DNS servers, the index of any leaf corresponds to a name-and-type hash. The value corresponds to:

1. A RRSET hash v (if the value was provided directly by a zone authority) or
2. $h(v \parallel t)$ where t is an instant of time, if the value was received from another DNS server.

10.6.1.4 Message Types

Two types of messages are defined. A message of type DG (for "delegation") conveys a value $h(n \parallel v)$ that is the function of a delegated name n and the identity $v = Y$ of the VN member (to which the name has been delegated). Messages of type RR convey a value $h(y \parallel v)$ where y is name-and-type credential, and v is a value associated with the name and type.

10.6.2 DNS Transactions

A transaction can be broadly seen as an unambiguous relationship between a precondition and a postcondition (on completion of the transaction). The preconditions can be the value v of an OMT leaf and/or receipt of a message of a specific type. The postconditions can be the creation of a message of a specific type and/or modification of value v in an OMT leaf. In general,

1. There may be any number of (different) transaction rules for each role (as each rule will be a leaf in OMT $\mathcal{S}$ which can have any number of leaves).

2. A message may be intended for another VN member or to itself (this feature enables complex transaction rules to be split into multiple simple steps).

The transactions for the five different roles are listed below by specifying the purpose of the transaction; preconditions (within squared brackets); and postconditions.

Registrar G transactions

G1 Assign x as owner of name a; [x has role "zone owner"]; $(a, a_n, 0) \rightarrow (a, a_n, x)$.
G2 Inform ownership of v_1 to a TLD; [(v_1, v_2, x) exists]; DG message $h(a, x)$.

Root Zone R Transactions

R1 Assume ownership of root zone; [$a = \Phi$]; $(a, a_n, 0) \rightarrow (a, a_n, R)$.
R2 Reserve a TLD name a for x; [inputs x, $e | v_1 = h(\Phi, e)$]; $(a, a_n, 0) \rightarrow (a, a_n, x)$.
R3 Assign TLD a to x; [(a, a_n, x) exists; x is a TLD owner]; DG message $h(a, x)$.

TLD Owner Transactions

T1 Assume ownership of name a; [DG message $h(a, T)$ from R]; $(a, a_n, 0) \rightarrow (a, a_n, T)$.
T2 Reserve name a for x; [(a', a'_n, T) exists; DG message $h(a, x)$ from G; input $b \mid a = h(a', b)$]; $(a, a_n, 0) \rightarrow (a, a_n, x)$.
T3 Delegate name a to x; [(a, a_n, x) exists; x is a zone owner]; DG message $h(a, x)$.
T4 Create NS record for name a; [(a', a'_n, T) exists; x is *not* zone owner, $a = h(a', \delta)$]; $a, a_n, 0) \rightarrow (a, a_n, x)$.
T5 RR Message to convey NS record to parent (root zone); [(a, a_n, x) exists; x is *not* a zone owner]; RR message $h(a, Y)$.

Zone Owner Z Transactions

Z1 Accept delegation of name a from a TLD/zone owner; [DG message $h(a, Z)$]; $(a, a_n, 0) \rightarrow (a, a_n, Z)$.
Z2 To assume ownership of a new name a, or delegate a name a, or create a name-and-type for name a; [(a', a'_n, Z) exists; inputs x, $b | a = h(a', b)$]; $(a, a_n, 0) \rightarrow (a, a_n, x)$.
Z3 Delegate name a to x; [(a, a_n, x) exists; x is a zone owner;] DG message $h(a, x)$.
Z4 Create RR message for name-and-type a;[(a, a_n, x) exists; x is *not* a zone owner]; RR message $h(a, x)$.

DNS server D Transactions

D1 Accept a DNS RRSET for name-and-type a; [RR message $h(a, x)$]; $(a, a_n, 0) \rightarrow (a, a_n, x)$.
D2 Convey RRSET for a at time t; [$(a, a_n, x \neq 0)$ exists]; RR message $h(a, h(x, t))$.
D3 Deny RRSET for name-and-type a; [$(a, a_n, 0)$ exists]; RR message $h(a, 0)$; note that the receiver of the RR message does not need to know enclosers of a (corresponding to names/types of records that do exist) to be assured of nonexistence of queried name and type.

As the transaction model enumerated above is only for illustrative purposes, not all transactions are included. For example, transactions for surrendering ownership of a name have not been accounted for. For this purpose, some additional subsystems may need to be added to the system. For example, the registrar subsystem may be seen as composed of multiple subsystems with a transaction model that defines how names are assigned/leased to potential zone owners, and how ownership of names can be revoked. Similarly, the transaction model does not specify how ownership of various TLD are assigned by the owner of the root zone. In practice, the root zone may also be broken down into several subsystems.

In general, complex systems can be broken down into simpler systems, each with a different VN. In such a scenario, in a CMM belonging to VN V_1 and V_2, credentials from one VN (say, V_1) may be *imported* into the other VN V_2 (if the rules in V_2 permit). For example, a VN S_1 may correspond to a system for instructing registrars to assign/revoke domain name and zone-owner bindings. A VN S_2 may be the DNS system. Registrars may be members of both VNs S_1 and S_2. Specifically, a CMM X may have the role of registrar R_1 in VN S_1 and registrar R_2 in VN S_2. R_2 may be explicitly permitted (by the rules in VN S_2) to import credentials of R_1.

10.6.3 Transaction Models for Other Systems

In general, the value in any OMT leaf may be a function of multiple credentials (instead of just a name or owner identity in the case of DNS). Similarly, while in the transaction model for DNS every VN member was required to maintain only a single IOMT, in general, each VN member may need to maintain multiple IOMTs/DOMTs. We already saw in Sect. 9.3.7 that a module corresponding to a remote file server may have to maintain two OMTs.

Just as we defined two message types RR and DLG for DNS, models for different systems may define any number of message types constructed using a parametric function $f()$. As an example, in BGP, a message can correspond to a BGP update message. Such a message can be seen as a function of several credentials like (1) IP prefix of the destination; (2) AS path; (3) and multiple BGP weights.

In deriving credential transaction model for any system, a designer has complete freedom in deciding how to "slice and dice" a system into several subsystems. Multiple simple subsystems could be combined into one subsystem. On the other hand, a complex subsystem could be split into multiple subsystems for purposes of simplifying the credential transaction rules for each subsystem.

As an example, consider a subsystem like a BGP speaker. Transactions performed by a BGP speaker include:

1. Receiving BGP update messages from neighboring routers
2. Modifying the weight according to AS policies/preferences
3. Storing all update messages in such a way as to easily identify the path with the best weight for any prefix

4. Advertise the best path for any weight, after adjusting some components of BGP weight depending on the next hop
5. Aggregating IP prefixes when the best paths for two adjacent prefixes go through the same next hop, etc.

For simplifying the transaction model, each router subsystem can be seen (for example) as constituted by three subsystems—one for sending/receiving BGP updates and adjusting weights according to AS policies; the second, responsible for storing all received paths and selecting the best path for any prefix; and the third for aggregating prefixes.

Other BGP subsystems may include:

1. A registry for AS numbers which maintains an IOMT where a is an AS number and value ω is the identity of the AS owner, who assigned the AS number
2. A registry for IP addresses which maintains a DOMT. The range $[a, a_n)$ is the range of IP addresses assigned to AS ω
3. AS owner, which maintains a DOMT where each leaf represents a chunk of addresses owned by the AS. The AS owner can easily divide any chunk into multiple chunks (by inserting a placeholder), and assign subchunks to different BGP speakers for the AS. For each BGP speaker in the AS. The AS owner will also have the freedom to specify:
   (a) Permitted neighbors
   (b) Components of BGP weights like local preference (LP), multiexit discriminator (MED), etc., for each neighbor
   (c) Prepath (or CISCO) weights for different prefixes, etc.
4. And finally, BGP speakers, which could each be split into several subsystems to simplify the transaction model

In GIS applications, tessellations can be represented using a nested DOMT. Consider a leaf (a, a_n, ω) where ω is the root of a DOMT with a leaf (a', a'_n, x). The implication is that a tessellation defined by latitude $[a, a_n)$ and longitude $[a', a'_n)$ is owned by x. The interval $[a, a_n)$ can be split into any number of intervals (by inserting a leaf), where all leaves will have the same ω. Similarly leaves of the nested tree (longitude coordinates) can also be split. While operations for splitting the ranges will not be guided by system dependent rules, system dependent rules will dictate how an x can be modified (for example to delegate a tessellation).

In a system for electronic cash, an IOMT indexed by user identities could be used. The value field is the cash reserve for the user. Any transaction involving transfer of an amount x from one user to another will involve reducing the value ω in one leaf by a value x and increasing the value ω in another leaf by the same value x.

Chapter 11
Conclusions

Almost every security issue in our day-to-day lives stems not from weaknesses in cryptographic algorithms but the environment in which cryptographic algorithms are executed. There is a dire need for investment in a (perhaps global) infrastructure for realizing sufficiently trustworthy hardware modules, which can provide a safe environment for execution of cryptographic algorithms. As security protocols guarantee the existence of specific application-specific relationships between application-specific inputs and outputs, in order to ensure application-specific assurances, the trusted environment should have the capability to execute a possibly unlimited number of application-specific algorithms.

One possible approach outlined in this book, based on trusted credential management modules (CMMs), is to:

1. Identify a set of broad *application-independent* high-level protocols executed in the trusted confines of CMMs and
2. Device a strategy to specify *application-dependent* protocols as a set of static parameters

Application-independent protocols for CMMs identified in this book include protocols for:

1. Key generation, distribution, and using secrets for computing pairwise secrets
2. Verifying, inserting, and updating leaves of an ordered Merkle tree
3. Establishment of virtual networks (VN) to mirror a system/application
4. Time synchronization between VN members
5. Authentication and integrity verification of various types of VN messages with a fixed format

The credential transaction model was adopted as the strategy for the latter, viz., specification of application-dependent parameters. Specifically, application-dependent parameters capture rules for:

1. Generation of various VN-specific message types
2. Updating the value field ω in an OMT leaf

M. Ramkumar, *Symmetric Cryptographic Protocols,* DOI 10.1007/978-3-319-07584-6_11,

Application-independent protocols for key predistribution schemes like modified Leighton–Micali scheme (MLS), parallel basic key predistribution scheme (PBK), and subset key and identity tickets (SKIT) can be useful. Specifically, MLS is preferable as long as the limitation in its scalability is not an issue for the specific application. SKIT and PBK are preferable for the scenarios demanding unlimited scalability. More specifically, SKIT will remain preferable until the time we have developed sufficient confidence in our ability to produce sufficiently tamper-responsive modules. As much of PBK's advantages stem from the ability to take advantage of the decrypt-only-when-necessary (DOWN) assurance, and as the DOWN assurance relies on the assumption that a single highly protected master secret in a tamper responsive module can not be revealed, it is only after gaining experience from real-world deployments can we be confident of the assumption behind the DOWN policy. From then on, PBK will be a preferred approach for scenarios requiring unlimited scalability.

Two variants of the ordered Merkle tree (IOMT) and domain ordered Merkle tree (DOMT), together, were shown to be suitable for representing the dynamic credentials corresponding to a participant in any application/system.

From the perspective of the trustworthy modules, protocols for creation of a VN involves specification of a VN identity, recognition of it's own role in the VN, and distribution of VN specific keys to other VN members,which are intended to be used for mutual authentication of members.

The protocol for time synchronization is required in scenarios where the clocks of the modules are not assumed to be synchronized. More specifically, this assumption obviates the need to trust the integrity of the hardware clock of the module when the module is powered off. Time synchronization is achieved through "quick" handshakes (with a maximum error less than the round-trip time for the handshake).

Many high-level application-independent protocols were constructed by combining inputs and outputs of lower-level protocols using *self-certificates*. The lower level protocols were themselves in turn constructed as simple sequences of a small number of symmetric pseudo random function (PRF) operations. In the same manner, to permit possibly complex application-specific rules to be captured, the parameters may specify simple rules for generation of application-specific self-messages that can be combined using simple application-specific rules to finally yield a message of a certain type to be delivered to a VN member, or for creation of a self-message instructing that the value field in a specific OMT leaf should be updated from $\omega \rightarrow \omega'$.

The main motivation for the CMM-based approach to secure systems is to eliminate the need for vague notions of trust (like "trust in an organization" or "trust in a specific person" or trust in "(possibly complex) software executed by a computer") with something that is arguably less vague—the assumption of read-proof and write-proof CMMs (that secrets protected by CMMs cannot be exposed and that the simple functionality of CMMs cannot be modified).

However, the trust in CMMs themselves is bootstrapped from the trust in the infrastructure for realizing and certifying the integrity of CMMs. Nevertheless, this is not a serious limitation due to the following reasons:

1. The infrastructure could be constituted by any number of independent verifying and certifying authorities.
2. It is possible for any entity to raise questions regarding the trust in such entities by demonstrating the feasibility of attacks that violate the assumed integrity of modules.
3. Deliberately limiting the protocols executed inside CMMs (or, functions executed inside CMMs) to simple PRF and logical operations, and deliberately constraining the storage memory requirements inside CMMs make it easier to verify the integrity of assumed trusted functionality.
4. Deliberately lowering the computational overhead inside the trusted boundary reduces concerns of heat dissipation, and thereby permits unconstrained strategies for (active and passive) shielding CMMs from intrusions.
5. That all CMMs will possess identical fixed functionality implies the ability to automate several tasks required for certification of integrity, and realization of CMMs at low cost.
6. Lower cost of CMMs implies that a larger fraction of CMMs can undergo consummate destructive testing.

In the final analysis, key distribution schemes for multicast security associations may not be as useful as schemes for unicast security. This is due to the fact that the ability to perform atomic relays using unicast security can eliminate the need for broadcast authentication using digital signatures. In those very rare circumstances where this is not feasible, one-time signature schemes could be used.

Broadcast encryption was primarily seen as a strategy suitable for stateless devices—most often in the form of set-top boxes. The rapidly lowering costs of general-purpose computing modules have dramatically modified the trend in the construction of such devices. There is perhaps no set-top device that is manufactured currently that does not employ a general-purpose computer running (a possibly highly stripped version of) a general-purpose operating system. Consequently, the very notion of stateless devices may not exist in a few years.

For stateful devices which can receive group secrets after deployment, the ability to atomically relay secrets, and/or the ability to succinctly specify group memberships using access control lists (ACLs) represented as OMTs—which can specify explicitly included devices or explicitly excluded devices—is perhaps a better alternative to distribute secrets to eligible devices.

References

1. D.R. Stinson, *Cryptography, Theory and Practice*, 2nd edn. (Chapman and Hall/CRC, Boca Raton, 2002)
2. A.J. Menendez, P. van Oorschot, C. Paul, C.S.A. Vanstone, Chapter 7: Block ciphers, in *Handbook of Applied Cryptography* (CRC, Boca Raton, 1997)
3. R.L. Rivest, *The MD5 Message-Digest Algorithm*. Internet RFC 1321, Internet Engineering Task Force, April 1992
4. FIPS 180-4, *Secure Hash Standard (SHS)*, March 2012
5. M. Dworkin, *Recommendation for Block Cipher Modes of Operation*. NIST Special Publication 800-38A, 2001
6. I. Damgard, A design principle in hash functions, in *Advances in Cryptology—CRYPTO '89 Proceedings*, ed. by G. Brassard. Lecture notes in computer science, vol. 435 (Springer, Berlin, 1989), pp. 416–427
7. H. Krawczyk, M. Bellare, R. Canetti, *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104 (1997)
8. M. Bellare, R. Canetti, H. Krawczyk, in *Advances in Cryptology CRYPTO 96*, ed. by N. Koblitz. Lecture notes in computer science, vol. 1109 (International Association for Cryptologic Research/Springer, Berlin, 1996), pp. 1–15
9. L. Lamport, Password authentication with insecure communication. Commun. ACM. **24**(11), 770–772 (1981)
10. R.C. Merkle, Protocols for public key cryptosystems, in *Proceedings of the 1980 IEEE Symposium on Security and Privacy*, ISSN 1540-7993, 1980
11. M. Dyer, T. Fenner, A. Frieze, A. Thomason, On key storage in secure networks. J. Cryptol. **8**, 189–200 (1995)
12. A. Perrig, R. Canetti, J.D. Tygar, D.X. Song, Efficent authentication and signing of multicast streams over lossy channels, in *Proceedings of the IEEE Symposium on Security and Privacy*, 2000
13. R.J. Anderson, F. Bergadano, B. Crispo, J.H. Lee, C. Manifavas, R.M. Needham, A new family of authentication protocols. ACM SIGOPS Oper. Syst. Rev. **32**(4), 9–20 (1998)
14. R. Needham, M. Schroeder, Using encryption for authentication in large networks of computers. Commun. ACM. **21**(12), 993–999 (1978)
15. B.C. Neuman, T. Ts'o, Kerberos: An authentication service for computer networks. IEEE Commun. **32**(9), 33–38 (1994)
16. T. Leighton, S. Micali, Secret-key agreement without public-key cryptography, in *Advances in Cryptology—CRYPTO '93*, ed. by D.R. Stinson. Lecture notes in computer science, vol. 773 (Springer, Berlin, 1994), pp. 456–479
17. M. Ramkumar, On the scalability of a "nonscalable" key distribution scheme, in *IEEE WoWMoM SPAWN*, Newport Beach, 2008
18. P.V. Mockapetris, *Domain Names—Concepts and Facilities*. (RFC Editor, 1987)

M. Ramkumar, *Symmetric Cryptographic Protocols,* DOI 10.1007/978-3-319-07584-6,

19. P.V. Mockapetris, *Domain Names—Implementation and Specification*. (RFC Editor, 1987)
20. D. Kaminsky, Catching up with Kaminsky. Netw. Secur. **2008**(9), 4–7 (2008)
21. A. Del Sorbo, Network Security—Sk-DNSSEC: An alternative to the public key scheme. PhD thesis, Department of Computer Science, University of Salerno, Baronissi, Italy
22. http://dnscurve.org/. Accessed July 2014
23. R. Arends, R. Austein, M. Larson, D. Massey, S. Rose, *DNS Security: Introduction and Requirements*, RFC 4033 (2005)
24. S. Weiler, J. Ihren, *Minimally Covering NSEC Records and DNSSEC On-line Signing*, RFC 4470 (2006)
25. B. Laurie, G. Sisson, R. Arends, Nominet, D. Blacka, *DNS Security (DNSSEC) Hashed Authenticated Denial of Existence*, RFC 5155 (2008)
26. DNS Survey, (2006), http://dns.measurement-factory.com/surveys/200608.html. Accessed July 2014
27. T. Moreau, A short note about DNSSEC impact on root server answer sizes. Document number C003924 (2006) http://www.connotech.com/dnssec_root_impact.pdf. Accessed July 2014
28. C. Fetzer, T. Jim, Incentives and disincentives for DNSSEC deployment (2004) http://research.att.com/trevor/papers/dnssec-incentives.pdf. Accessed July 2014
29. B. Lampson, M. Abadi, M. Burrows, E. Wobber, Authentication in distributed systems: Theory and practice. ACM Trans. Comput. Syst. **10**(4), 265–310 (1992)
30. A. Velagapalli, M. Ramkumar, Trustworthy TCB for DNS servers. Int. J. Netw. Secur. **14**(4), 187–205 (2012)
31. D. Kaminsky, DNS 2008 and the new (old) nature of critical infrastructure. BlackHat DC (2009)
32. P. Vixie, O. Gudmundsson, D. Eastlake 3rd, B. Wellington, *Secret Key Transaction Authentication for DNS (TSIG)*, RFC 2845 (May 2000)
33. CommunityDNS, DNSSEC a way forward for TLD registries. White paper (2009), http://www.communitydns.net/DNSSEC.pdf. Accessed July 2014
34. J. Bau, J.C. Mitchell, A security evaluation of DNSSEC with NSEC3. Paper presented at the seventeenth annual network and distributed systems security symposium (NDSS), San Diego, CA, 2010
35. S. Kent, R. Atkinson, *Security Architecture for the Internet Protocol*, RFC 2401 (1998)
36. D. Maughan, M. Schertler, M. Schneider, J. Turner, *Internet Security Association and Key Management Protocol (ISAKMP)*, RFC 2408 (1998)
37. N. Hallqvist, A.D. Keromytis, Implementing internet key exchange (IKE), in *Proceedings of FREENIX Track, 2000 USENIX Annual Technical Conference*, San Diego, CA, June 2000
38. C. Kaufman (ed.), *Internet Key Exchange (IKEv2) Protocol*, RFC 4306 (2005)
39. B. Aboba, W. Dixon, *IPsec-Network Address Translation (NAT) Compatibility Requirements*. RFC 3715 (2004)
40. D. Clarke, J-E. Elien, M. Fredette, A. Marcos, R.L. Rivest, Certificate chain discovery in SPKI/SDSI. J. Comput. Secur. **9**(4), 285–322 (2001)
41. D. Boneh, M. Franklin, Identity-based encryption from the Weil pairing, in *Advances in Cryptology—CRYPTO 2001*, ed. by J. Kilian. Lecture notes in computer science, vol. 2139 (Springer, Berlin, 2001), pp. 213–229
42. R. Dutta, R. Barua, P. Sarkar, Pairing-based cryptography: A survey. Cryptology ePrint Archive, Report 2004/064 (2004)
43. A. Shamir, Identity-based cryptosystems and signature schemes, in *Advances in Cryptology*, ed. by G.R. Blakley, D. Chaum. Lecture notes in computer science, vol. 196 (Springer, Berlin, 1985), pp. 47–53
44. R. Blom, Non-public key distribution, in *Proceedings of CRYPTO 82* (Plenum, New York, 1983), pp. 231–236
45. R. Blom, An optimal class of symmetric key generation systems. in *Advances in Cryptology: Proceedings of Eurocrypt 84*, Lecture notes in computer science, vol. 209 (Springer, Berlin, 1984), pp. 335–338
46. L. Gong, D.J. Wheeler, A matrix key distribution scheme. J. Cryptol. **2**(2), 51–59 (1990)

47. C.J. Mitchell, F.C. Piper, Key storage in secure networks. Discret. Appl. Math. **21**, 215–228 (1995)
48. C. Padro, I. Gracia, S. Martin, P. Morillo, Linear broadcast encryption schemes. Discret. Appl. Math. **128**(1), 223–238 (2003)
49. P. Erdos, P. Frankl, Z. Furedi, Families of finite sets in which no set is covered by the union of *r* others. Israel J. Math. **51**, 79–89 (1985)
50. L. Eschenauer, V.D. Gligor, A key-management scheme for distributed sensor networks, in *Proceedings of the Ninth ACM Conference on Computer and Communications Security*, Washington DC, Nov 2002, pp. 41–47
51. H. Chan, A. Perrig, D. Song, Random key pre-distribution schemes for sensor networks. Paper presented at the IEEE symposium on security and privacy, Berkeley, California, May 2003, PP. 197–205
52. R. Di Pietro, L.V. Mancini, A. Mei, Random key assignment for secure wireless sensor networks. Paper presented at the ACM workshop on security of ad hoc and sensor networks, Oct 2003
53. S. Zhu, S. Xu, S. Setia S. Jajodia, Establishing pair-wise keys for secure communication in ad hoc networks: A probabilistic approach, in *Proceedings of the 11th IEEE International Conference on Network Protocols (ICNP'03)*, Atlanta, Georgia, 4–7 Nov 2003
54. W. Du, J. Deng, Y.S. Han, P.K.Varshney, A pairwise key pre-distribution scheme for wireless sensor networks, in *Proceedings of the 10th ACM Conference on Computer and Communication Security*, 2003, pp. 42–51
55. D. Liu, P. Ning, Establishing pairwise keys in distributed sensor networks, in *Proceedings of the 10th ACM Conference on Computer and Communication Security*, Washington DC, 2003
56. M. Ramkumar, N. Memon, R. Simha, Pre-loaded key based multicast and broadcast authentication in mobile ad-hoc networks. Paper presented at Globecom '03, 1–5 Dec 2003
57. M. Ramkumar, N. Memon, An efficient random key pre-distribution scheme for MANET security. IEEE J. Sel. Areas Commun. (2005)
58. M. Kwiatkowska, V. Sassone, in *Science for Global Ubiquitous Computing. Grand Challenges in Computing (Research)*, ed. by T. Hoare, R. Milner, Bulletin of the EATCS, vol 82, pp. 325–333
59. M. Ramkumar, I-HARPS: An efficient key pre-distribution scheme for mobile computing spplications. Paper presented at IEEE Globecom, San Francisco, CA, Nov 2006
60. M. Ramkumar, Trustworthy computing under resource constraints with the DOWN policy. IEEE Trans. Secur. Depend. Comput. **5**(1), 49–61 (2008)
61. M. Ramkumar, Proxy aided key pre-distribution schemes for sensor networks. IEEE NSP, Austin, TX, Dec 2008
62. M. Ramkumar, The subset keys and identity tickets (SKIT) key distribution scheme. IEEE Trans. Inf. Forensics Secur. **5**(1), 39–51 (2010)
63. C-Y. Chong, S.P. Kumar, Sensor networks: Evolution, opportunities, and challenges. Proc. IEEE **91**(8), 1247–1256 (2003)
64. T. He, C. Huang, B. Blum, J. Stankovic, T. Abdelzaher, Range-free localization schemes in large scale sensor networks. Paper presented at Mobile Computing and Networking (MobiCom), 2003
65. R. Gennaro, A. Lysyanskaya, T. Malkin, S. Micali, T. Rabin, Tamper proof security: Theoretical foundations for security against hardware tampering. Paper presented at the theory of cryptography conference, Cambridge, MA, Feb 2004
66. S.W. Smith, S. Weingart, Building a high-performance programmable secure coprocessor. Comput. Netw. **31**, 831–860 (1999)
67. D. Lie, C.A. Thekkath, M. Horowitz, Implementing an untrusted operating system on trusted hardware, in *Proceedings of the 19th ACM symposium on operating systems principles*, Oct 2003, pp. 178–192 (ACM)
68. P.C. van Oorschot, A. Somayaji, G. Wurster, Hardware-assisted circumvention of self-hashing software tamper resistance. IEEE Trans. Dependable Secur. Comput. **2**(2), 81–92 (2005)
69. S. Weingart, Physical security for the m-ABYSS System, in *Proceedings of the IEEE Symposium on Security and Privacy* (1987), pp. 38–51

70. S. White, S. Weingart, W. Arnold, E. Palmer, Introduction to the citadel architecture: Security in physically exposed environments. Technical report RC16672, IBM Thomas J. Watson Research Center, March 1991
71. J.D. Tygar, B. Yee, Dyad: A system for using physically secure coprocessors, in *Technological Strategies for the Protection of Intellectual Property in the Networked Multimedia Environment* (CNI, Washington DC, 1994), pp. 121–152
72. B. Chen, R. Morris, Certifying program execution with secure processors, in *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, 1821, Lihue, Hawaii, May 2003
73. E. Biham, A. Shamir, Differential fault analysis of secret key cryptosystems, in *Advances in Cryptology—CRYPTO '97*, ed. by B.S. Kaliski Jr. Lecture notes in computer science, vol. 1294 (Springer, Berlin, 1997)
74. M.G. Karpovsky, K. Kulikowski, A. Taubin, Robust protection against fault-injection attacks of smart cards implementing the advanced encryption standard, in *Proceedings of the International Conference on Dependable Systems and Networks (DNS 2004)*, July 2004
75. P. Kocher, Differential power analysis, in *Advances in Cryptology, CRYPTO 1999*, ed. by M. Wiener. Lecture notes in computer science, vol. 1666 (Springer, Berlin, 1999), pp. 388–397
76. C. Aumeller, P. Bier, W. Fischer, P. Hofreiter, J.P. Seifert, Fault attacks on RSA with CRT: Concrete results and practical countermeasures. Cryptology ePrint Archive, http://eprint.iacr.org/2002/073.pdf
77. S. Moore, R. Anderson, P. Cunningham, R. Mullins, G. Taylor, Improving smart card security using self-timed circuits. Paper presented at the eighth international symposium on advanced research in asynchronous circuits and systems, 2002
78. O. Kommerling, M. Kuhn, Design principles for tamper-resistant smart-card processors, in *Proceedings of the Usenix Workshop on Smartcard Technology* (1999), ACMID 1267117, pp. 9–20
79. R. Anderson, M. Bond, J. Clulow, S. Skorobogatov, Cryptographic processors—A survey. University of Cambridge, Computer laboratory technical report, UCAM-CL-TR-641, Aug 2005
80. Y. Ishai, A. Sahai, D. Wagner, Private circuits: Securing hardware against probing attacks, in *Advances in Cryptology—CRYPTO 2003*, Santa Barbara, CA, Aug 2003
81. B. Gassend, D. Clarke, M. van Dijk, S. Devadas, Delay-based circuit authentication and applications, in *Proceedings of the 2003 ACM symposium on applied computing*, Melbourne, Florida (2003) pp. 294–301
82. B. Gassend, D. Clarke, M. van Dijk, S. Devadas, Silicon physical random functions, in *Proceedings of the Computer and Communications Security Conference*, Nov 2002
83. R. Pappu, Physical one-way functions. PhD Thesis, MIT, 2001
84. P. Gutman, Secure deletion of data from magnetic and solid-state memory. Paper presented at the sixth USENIX security symposium, San Jose, California, July 1996
85. R. Anderson, M. Kuhn, Low cost attacks on tamper resistant devices. Paper presented at the IWSP: international workshop on security protocols, Paris, April 1997
86. C. Couvreur, J.-J. Quisquater, Fast decipherment algorithm for RSA public-key cryptosystem. Electron. Lett. **18**(21), 905–907 (1982)
87. J.P. McGregor, R.B. Lee, Protecting cryptographic keys and computations via virtual secure coprocessing. ACM SIGARCH Comput. Archit. News **33**(1), 16–26 (2005)
88. A. Fiat, M. Noar, Broadcast encryption, in *Advances in Cryptology*, ed. by D.R. Stinson. Lecture notes in computer science, vol. 773, (Springer, Berlin, 1994), pp. 480–491
89. J. Lotspiech, S. Nusser, F. Pestoni, Anonymous trust: Digital rights management using broadcast encryption. Proc. IEEE. **92**(6), 898–909 (2004)
90. P.T. Eugster, P.A. Felber, R. Guerraoui, A-M. Kermarrec, The many faces of publish/subscribe. Technical report. citeseer.ist.psu.edu/649723.html
91. R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, B. Pinkas, Multicast security: A taxonomy and some efficient constructions. Paper presented at the eighteenth annual joint

conference of the IEEE computer and communications societies (INFOCOMM '99), New York, USA, 1999

92. D. Noar, M. Noar, J. Lotspiech, Revocation and tracing schemes for stateless receivers, in *Advances in Cryptology—CRYPTO 2001*, ed. by J. Kilian. Lecture notes in computer science, vol. 2139 (Springer, Berlin, 2001)
93. C.K. Wong, M. Gouda, S. Lam, Secure group communications using key graphs, in *Proceedings of SIGCOMM* (1998), pp. 68–79
94. J. Anzai, N. Matsuzaki, T. Matsumoto, A method for masked sharing of group keys (3). IEICE technical report, ISEC99-38, 1999
95. J. Anzai, N. Matsuzaki, T. Matsumoto, A quick group key distribution scheme with "entity revocation", in *Advances in Cryptology—ASIACRYPT'99*, ed. by K-Y. Lam, E. Okamoto, C. Xing. Lecture notes in computer science, vol. 1716 (Springer, Berlin, 1999), pp. 333–347
96. S.M. Matyas, C.H. Meyer, Generation, distribution and installation of cryptographic keys. IBM Syst. J. **2**, 126–137, (1978)
97. P. Devanbu, M. Gertz, C. Martel, S. Stubblebine, Authentic third-party data publication. Paper presented at the fourteenth IFIP 11.3 conference on database security, 2000
98. A. Buldas, P. Laud, and H. Lipmaa, Accountable certificate management using undeniable attestations, in *Proceedings of the ACM Conference on Computer and Communications Security* (2000), pp. 9–17
99. A. Anagnostopoulos, M.T. Goodrich, R. Tamassia, Persistent authenticated dictionaries and their applications, in *Information Security*, ed. by G.I. Davida, Y. Frankel. Lecture notes in computer science, vol. 2200 (Springer, Berlin, 2001), pp. 379–393
100. C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, S. Stubblebine, A general model for authentic data publication. VC davis department of computer science technical report, 2001
101. P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, S. Stubblebine, Flexible authentication of XML documents, in *Proceedings of the ACM Conference on Computer and Communications Security*, 2001
102. M.T. Goodrich, R. Tamassia, A. Schwerin, Implementation of an authenticated dictionary with skip lists and commutative hashing, in *Proceedings of the DARPA Information Survivability Conference and Exposition*, vol. 2 (IEEE, Piscataway, 2001), pp. 68–82
103. M.T. Goodrich, R. Tamassia, N. Triandopoulos, R. Cohen, Authenticated data structures for graph and geometric searching, in *Topics in Cryptology—CT-RSA 2003*. Lecture notes in computer science, vol. 2612 (Springer, Berlin, 2003), pp. 295–313
104. S.R. Fluhrer, I. Mantin, A. Shamir, Weaknesses in the key scheduling algorithm of RC4, in *Selected Areas in Cryptography*, ed. by S. Vaudenay, A.M. Youssef. Lecture notes in computer science, vol. 2259 (Springer, Berlin, 2001), pp. 1–24 (August 16–17, 2001)
105. Andrew Huang, *Hacking the Xbox: An Introduction to Reverse Engineering*. No Starch Press, 2003, ISBN: 1593270291
106. D. Wheeler, N. Roger, TEA, A tiny encryption algorithm, in *Fast Software Encryption*, ed. by B. Preneel. Lecture notes in computer science, vol. 1008 (Springer, Berlin, 1995), pp. 363–366
107. TCG specification: Architecture overview, Specification revision 1.4, 2nd August 2007
108. D. Levin, D. Douceur, J.R. Lorch, T. Moscibroda, Trinc: Small trusted hardware for large distributed systems, in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, Berkeley, CA, USA (USENIX Association, Berkeley, 2009), pp. 1–14
109. A. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, E. Felten, Lest we remember: Cold boot attacks on encryption keys. *Proceeding of the Seventeenth USENIX Symposium*, San Jose, CA, 2008
110. E. Sparks, A security assessment of trusted platform modules. Computer science technical report TR2007-597, Dartmouth College, 2007

111. S. Bratus, E. Sparks, S.W. Smith, Toctou, traps, and trusted computing, in *Trust 08: Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies* (2008), pp. 14–32
112. M. Peinado, Y. Chen, P. Engl, J. Manferdelli, NGSCN: A trusted open system, in *Proceedings of 9th Australasian Conference on Information Security and Privacy, ACISP* (Springer, Berlin, 2004), pp. 86–97
113. T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, D. Boneh, Terra: A virtual machine-based platform for trusted computing, in *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP)* (ACM, New York, 2003), pp. 193–206
114. P. Barham, B. Dragovic, K. Fraser, H. Steven, T. Harris, A. Ho, R. Neugebauer, Pratt, I., A. Warfield, Xen and the art of virtualization, in *Proceedings of the Nineteenth ACM Symposium on Operating System Principles, SOSP '03* (2003), pp. 164–177
115. E. Bugnion, S. Devine, M. Rosenblum, Disco: Running commodity operating systems on scalable multiprocessors. ACM Trans. Comput. Syst. **15**(4), 143–156 (1997)
116. B. Kauer, OSLO: Improving the security of trusted computing, in *Proceedings of the USENIX Security Symposium*, Aug 2007
117. J.M. McCune, B. Parno, A. Perrig, M.K. Reiter, H. Isozaki, Flicker: An execution infrastructure for TCB minimization, in *Proceedings of the ACM European Conference on Computer Systems (EuroSys'08)*, Glasgow, Scotland, March 2008
118. J.M. McCune, Reducing the trusted computing base for applications on commodity systems. PhD Thesis, School of electrical and computer engineering carnegie Mellon University, Pittsburgh, PA, 2009
119. R. Wojtczuk, J. Rutkowska, Attacking intel trusted execution technology, Black Hat DC, Feb 2009
120. L.F.G. Sarmenta, M.V. Dijk, C.W. Odonnell, J. Rhodes, S. Devadas, Virtual monotonic counters and count-limited objects using a TPM without a trusted OS, in *Proceedings of the 1st ACM CCS Workshop on Scalable Trusted Computing (STC06)* (2006), pp. 27–42
121. M. van Dijk, L.F.G. Sarmenta, J. Rhodes, S. Devadas, Virtual monotonic counters and count-limited objects using a TPM without a trusted OS, in *Proceedings of the First ACM Workshop on Scalable Trusted Computing (STC06)* (2006), p. 2741
122. M. van Dijk, L.F.G. Sarmenta, J. Rhodes, S. Devadas, Securing shared untrusted storage by using TPM 1.2 without requiring a trusted OS, MIT CSG Memo 498, May 2007
123. Y. Rekhter, T. Li, *A Border Gateway Protocol 4 (BGP-4)*. (RFC Editor, 1995)
124. S.D. Mohanty, M. Ramkumar, Securing file storage in an untrusted server using a minimal trusted computing case. Paper presented at the first international conference on cloud computing and services science, Noordwijkerhout, The Netherlands, May 2011
125. V. Thotakura, M. Ramkumar, Minimal TCB for MANET nodes. Paper presented at the 6th IEEE international conference on wireless and mobile computing, networking and communications (WiMob 2010), Niagara Falls, ON, Canada, Sep 2010
126. V. Thotakura, M. Ramkumar, Leveraging a minimal trusted computing base for securing on-demand MANET routing protocols. Paper presented at the 9th international information and telecommunication technologies symposium (I2TS'2010), Rio de Janeiro, Brazil, 2010
127. A. Velagapalli, M. Ramkumar, An efficient trusted computing base (TCB) for a SCADA system monitor. Paper presented at the 10th international information and telecommunication technologies symposium (I2TS'2011), Floripa, Brazil, 2011
128. A. Velagapalli, M. Ramkumar, Minimizing the TCB for securing SCADA systems. Paper presented at ACM's 7th annual cyber security and information intelligence research workshop (CSIIRW), Oak Ridge, TN, Oct 2011
129. S. Mohanty, A. Velagapalli, M. Ramkumar, An efficient TCB for a generic content distribution system. Paper presented at the international conference on cyber-enabled distributed computing and knowledge discovery, Oct 2012
130. A. Velagapalli, S. Mohanty, M. Ramkumar, An efficient TCB for a generic data dissemination system. Paper presented at the international conference on communications in China: Communications theory and security (CTS), ICCC12-CTS, 2012

Index

A
Asymmetric Algorithms, 4
Atomic Relay, 41, 45, 133
  Secrets, 135
  Intelligent, 42
Authenticated Denial, 37, 54
  Hashed, 54

B
BE, 146
  Performance Bounds, 151
  Models, 156
  Multisource, 148
  Over-provisioning Keys, 152
  Probabilistic KPS, 148
  Tree Based, 147
BGP, 209
  Transaction Model, 226
Blockcipher, 2
  Modes, 5
  Properties, 3
Blockcipher Modes, 5
Blockcipher Properties, 3
Border Gateway Protocol (BGP), 209
Broadcast Encryption (BE), 146

C
Certificates Based Schemes, 65, 140
CMM, 208
  ADS, 216
  Architecture, 216
  State, 213
  Universe, 217
Credential Management Modules (CMM), 208
Credential Transaction Model, 208, 221
  DNS, 221
Credentials
  BGP, 209
  DNS, 209
Cryptographic Hash, 3
Cryptographic Protocol, 7

D
Decrypt Only When Necessary(DOWN), 112
DNS
  Cache Poisoning, 35
  Query-Response, 34
  RRSET, 33
  Walk, 38
  Authenticated Denial, 37
  Credential Transaction Model, 221
  Credentials, 209
  DNSCurve, 36
  DNSSEC, 37
  NSEC3 Opt-out, 58
  Symmetric Key DNSSEC, 36
  Transactions, 224
  TSIG, 57
Domain Name System, 31
DOWN
  PKPS, 119
  Assurance, 118, 145
  Complexity, 118
  Elliptic Curves, 115
  Exponential Ciphers, 114
  IBE/IBS, 116
  Policy, 112
  RSA, 113

G
Geographic Information Systems, 210
GIS, 210
  Credentials, 210
  Transaction Model, 226

M. Ramkumar, *Symmetric Cryptographic Protocols,* DOI 10.1007/978-3-319-07584-6,

H
Hash Chains, 11
HMAC, 6

I
Identity Based Schemes, 66, 142
IPSec, 59
  Alternative, 61
  Port Mapping, 63
  Issues, 61
  Operation, 60

K
Kerberos, 21
KPS
  Complexity, 80
  Algorithms, 129
  Basic Key Pre-distribution, 22
  Blom's Schemes, 67
  Deterministic vs Probabilistic, 78
  Identity Ticket, 26
  Leighton-Micali Scheme, 21
  Modified Leighton-Micali Scheme, 24
  Multi-path Diversity, 83
  Multiple KDC, 82
  Probabilistic KPS (PKPS), 69
  Scalable, 67
  Security Protocols, 132

M
Merkle Tree, 12, 169
  Protocols, 170
Message Injection, 99

N
Needham Schroeder Protocol, 20

O
OMT
  Domain Ordered Merkle Tree (DOMT), 180
  Index Ordered Merkle Tree (IOMT), 179
  Algorithms, 182
  Core Functions, 184
  Equivalence Certificates, 188
  Insertion of Leaves, 176
  Leaves, 172
  Nested, 190
  Nodes, 173
  Placeholder, 189
  Properties, 181
  Protocols, 174, 176
  Reordering Leaves, 178
  Self-Certificates, 185
Ordered Merkle Tree (OMT), 171

P
PKPS, 69
  Comparison, 91, 122
  Complexity, 119, 122
  Device Model, 120
  HARPS, 72
  PBK, 86
  PLM, 87
  RPS, 70
  Security Model, 73, 95
  SKIT, 89
Properties of PRFs, 9
Publish Subscribe Systems, 161

R
Random Subsets, 15, 69
Remnance, 110

S
Security Protocol, 8
Self-Certificates, 183, 185
Sensor Networks, 100
System Security Model, 201

T
TCB-DNS, 45
TPM, 202
  Roots of Trust, 202
Trinc, 204
Trusted Computing Base, 40
Trusted Platform Modules, 202

V
Virtual Counters, 206
Virtual Networks (VN), 211
VN, 211
  Creation, 218
  Key Distribution, 219
  Links, 219
  State, 213
  State Changes, 212