SECURE BROADCAST COMMUNICATION
in Wired and Wireless Networks

SECURE BROADCAST COMMUNICATION

in Wired and Wireless Networks

ADRIAN PERRIG
Carnegie Mellon University

J. D. TYGAR
UC Berkeley

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

Library of Congress Cataloging-in-Publication Data

Perrig, Adrian.
Secure broadcast communication in wired and wireless networks / Adrian Perrig, J.D. Tygar.
p. cm.
Includes bibliographical references and index.
ISBN 978-1-4613-4976-1 ISBN 978-1-4615-0229-6 (eBook)
DOI 10.1007/978-1-4615-0229-6
1. Telecommunication--Security measures. I. Tygar, J. D. II. Title.

TK5102.85 .P47 2002
621.382'12--dc21

2002034124

Printed on acid-free paper.

Contents

List of Figures

List of Tables

List of Protocols

Preface

Streaming media, sensor networks, satellite communication, and dozens of emerging applications depend on broadcast communication. This broadcast communication may be a true broadcast (for example, a satellite transmitting to millions of receivers) or it may be implemented on IP multicast. But regardless of the underlying technology, security is an essential requisite for most applications.

Security for broadcast involves different considerations than point-to-point communication. Eavesdropping is particularly simple. The potentially large and dynamic set of broadcast subscribers poses difficult key management problems. Receivers are heterogeneous — they often have different computational resources, different bandwidth, and different latency. For receivers with limited computational resources, even simple cryptography can consume significant overhead. Packets are often lost, and retransmission of lost packets poses challenges given if the number of receivers is large.

This book presents a number of protocols for secure broadcast. We discuss both wired networks and wireless networks, and describe in detail protocols for a special type of wireless network, a sensor network containing a large number of nodes with weak computational ability. We present protocols for key distribution, authentication, and non-repudiation. We show how to protect against adversaries who inject packets or eavesdrop. Because we focus on protocols as fundamental building blocks, this book can be viewed as cookbook full of security recipes. These protocols can be combined with each other or with more traditional security protocols to specify how to build secure systems.

The presentation is on two levels: we primarily present security at a conceptual level and illustrate our work with discussions of actual implementations. We have written this book for graduate students, researchers and engineers in

security. We avoid extensive complexity theoretic discussions and favor functional descriptions of protocols. We have written this book to serve for self-study, as a text in an advanced graduate level seminar, or as a supplementary text in a first year graduate security or networking class.

Acknowledgments

Some of the work described in this book is derived from our research with our colleagues. We have used this research in preparing our book, but unlike the original papers, we have often removed more theoretical or academic material. The original work can be found in the following papers: [HJP02, HPJ01, HPJ02, MP02, Per01, PCB+02, PCST01, PCST01, PCTS00, PCTS02, PST01, PSW+01, PSW+02, SP01]. We are deeply indebted to our fellow researchers and co-authors, Bob Briscoe, David Culler, Ran Canetti, Yih-Chun Hu, David Johnson, Michael Mitzenmacher, Dawn Song, Robert Szewczyk, Victor Wen. We thank them for their creative ideas, their criticism, their encouragement, and their friendship.

Special thanks are due to Ross Anderson, Manuel Blum, Nikita Borisov, Eric Brewer, Monica Chew, John Chuang, Yongdae Kim, Hugo Krawczyk, Markus Kuhn, Markus Jakobsson, Michael Luby, Michael Rabin, Mike Reiter, Pam Samuelson, Yuan Kui Shen, Amin Shokrollahi, Gene Tsudik, David Wagner, Avi Wigderson, and Jeannette Wing for many fruitful discussions, comments on the original broadcast security papers, and comments on earlier drafts of this manuscript.

Most of this work was done while the two authors were at the University of California, Berkeley. UC Berkeley's Electrical Engineering and Computer Science Department and the School of Information Management and Systems have been a wonderful place to develop this material. In particular, we would like to thank Randy Katz, Richard Newton, Christos Papadimitriou, Shankar Sastry, and Hal Varian for making Berkeley such a productive home for research. The first author did some of the research reported in this book at IBM's T. J. Watson Research Laboratory and Digital Fountain and thanks them for allowing him to pursue this work.

The National Science Foundation, the Defense Advanced Research Projects Agency, and the US Postal Service have provided us with grants and contracts that have helped support much of the development of this material. We gratefully acknowledge this support.

Special thanks to Alexander Greene, our editor at Kluwer for his wise advice and for shepherding this book.

The first author dedicates this book to Dawn. The second author dedicates this book to his wife, Xiaoniu. Words can not express our immense debt to them for their love and patience.

Any errors that remain in this book are our own. We welcome comments and corrections. We can be reached at `adrian.perrig@cs.cmu.edu` or `tygar@cs.berkeley.edu`.

ADRIAN PERRIG AND J. D. TYGAR

Chapter 1

INTRODUCTION

Digital broadcast technology is hot. A casual glance at a newspaper reveals it: articles describe the impact of streaming media, questions of intellectual property being broadcast, new applications that rely on broadcast, discussions of privacy concerns with broadcast, etc. We are invited to electronically join broadcast meetings over the Internet, we can choose to listen to famous or obscure streaming radio stations from most corners of the globe, and we are promised a future where we will receive high-definition digital TV.

Clearly, broadcasting is an important technology. But can we do it securely? And what does "secure broadcast" mean? To us, secure broadcast revolves around two themes:

- receivers are certain that material they receive came from the appropriate sender; and
- senders have the option of limiting the recipients of particular messages.

In this chapter, we discuss these themes at length. And in this book, we address these concerns and many others, ranging from implementing secure broadcast on hardware with limited computational power to using broadcast mechanisms to execute denial-of-service attacks.

But first, let us begin by examining some basic terms. *Broadcast communication* is an essential mechanism for scalable information distribution. *Point-to-point communication* has been the dominant form of computer network communication since the beginning of networking. Unfortunately, with the explosive growth of information technology and the proliferation of the Internet and its applications, point-to-point communication faces serious scalability

A. Perrig et al., *Secure Broadcast Communication*

challenges. Distributing content, such as a popular movie, to a large audience over individual point-to-point connections is not economical. In contrast, one server can effortlessly reach vast audiences by using broadcast communication such as IP multicast.

A large number of broadcast applications exist today and even more are emerging. We present some broadcast applications here, and we discuss them in greater detail later in this chapter.

- **Content distribution over the Internet**. Digital TV and digital radio distributed over the Internet are becoming increasingly popular. Most likely, we will receive the future Soccer World Cup final or the opening ceremony of the Olympic games over an Internet or satellite broadcast.

- **Software distribution**. Broadcast enables quick and wide-spread distribution of software updates. For example, consider anti-virus software. Viruses utilize the Internet to infect hundreds of thousands of PC's in a few hours. As a countermeasure, an anti-virus center could broadcast virus scanner updates, and the operating system manufacturer could broadcast security patches.

- **Sensor networks**. The miniaturization of networked sensors brings an opportunity to solve many hard problems. Consider the problems of real-time road monitoring, real-time building safety monitoring (e.g., seismic safety), or fine-grained climate control in buildings. Since typical sensor networks communicate over wireless networks, broadcast is a natural communication method.

- **Transportation control**. By nature, transportation information affects multiple parties and is rapidly changing. Consider air traffic control. The majority of the communication patterns are broadcasts: GPS signals, airport landing system beacons, radio communications. Another example is road traffic control. Future cars will be equipped with wireless communication for driving support. Future highways may be equipped with beacons that broadcast location and current road information to cars.

- **Logistics and fleet monitoring**. Many companies need to track their fleet and can benefit from knowing accurate position information about their vehicles. Consider a taxi cab company or an express delivery service. With accurate position information, the coordination center can take informed decisions when it plans cab availability, or delivery times. An effective method for communication in such a setting is through broadcast. Location

tracking is also useful for public services, such as buses and trains. Future buses may broadcast their current location and schedule to help passengers to predict the arrival time.

- **Personal wireless communications**. With the proliferation of cell phones we expect to see a rapid growth of broadcast-based wireless services and applications. For example, traffic monitors built into highways broadcast the current traffic pattern, public services such as hospitals and pharmacies may broadcast health-related information to people wearing health-support equipment. The potential applications are innumerable.

- **Home automation**. If the vision of ubiquitous or pervasive computing becomes a reality, almost every object within a household will be able to compute and communicate with other objects. Wireless communication (i.e., broadcast) may be an efficient and simple method for managing and querying these objects.

- **Financial markets**. Disseminating real-time financial market information efficiently to a large audience is an important challenge. We want to broadcast the information to all receivers simultaneously to attempt to achieve the goal of having receivers get the information at the same time, while protecting the integrity of the information.

- **Military applications**. The military has a wide range of applications that rely on broadcast to achieve robustness and survivability. Furthermore, many military applications rely on wireless communication.

- **Multi-player games**. Since the early days of networking, multi-player games are popular entertainment applications. Today, some games involve thousands of people simultaneously interacting in the virtual game world. Broadcast communication enables scalable and efficient games.

1.1 Challenges of Broadcast Communication

Point-to-point communication protocols are designed for one sender and one receiver. Unfortunately, the majority of point-to-point protocols do not generalize to broadcasting data to multiple receivers. Namely, broadcast communication encounters the following major challenges:

- **Reliability**. In point-to-point communication, a receiver achieves reliability by detecting missing or corrupted data, and requesting the sender to

retransmit. In large-scale broadcasts, such an approach would not scale because a single lost packet can cause a flood of retransmission requests at the sender (this problem is sometimes called NACK implosion).

- **Receiver heterogeneity**. Some receivers may have high-bandwidth network connections and powerful workstations while others may have low-bandwidth connections with minimal computation resources.

- **Congestion control**. If a link in the Internet is congested, all well-behaved flows should back off until the link is not congested any more. Congestion control for IP multicast is particularly challenging.

- **Security**. Traditional security protocols for point-to-point communication suffer from the following problems in a broadcast setting: they may not scale to large audiences, they may not be secure, they may not be efficient and have a high computation or communication overhead, or they may not be robust to packet loss. We discuss these problems in more detail in Section 1.2.

Since reliability for individual packets is difficult to achieve in large-scale broadcasts, many broadcast applications distribute individual packets *unreliably*. Many broadcast applications do not expect reliability guarantees from the communication protocol, and the sender is not responsible for retransmitting lost packets. Thus, broadcast applications deal with packet loss on the application layer, e.g., multimedia applications experience quality degradation, or file transfer applications use forward error correction (FEC) [BLMR98].

In this book, we consider broadcast communication with the following properties: the sender *unreliably* distributes *real-time data*, the receiver wants to *immediately use data* as it arrives, and the broadcast must be secure. This is an especially challenging scenario, although common for many broadcast applications. Other broadcast settings may be simpler, and thus our protocols can easily provide security in those settings as well. To summarize, we consider broadcasts with the following features and requirements:

- Large numbers of receivers

- Receiver heterogeneity in computation resources (processor, memory, disk), and network resources (bandwidth, delay, reliability)

- The sender cannot retransmit lost packets

- Real-time data: the sender does not know the data in advance

- Streamed data: the receiver uses all data it receives
- Fast sending rate
- Security, in particular data authenticity and confidentiality

Applications with these requirements include real-time video streams, and data distribution that uses FEC [BLMR98, DF02, RMTR02]. We view FEC distribution as real-time data, as these systems often encode a fixed file into a long data stream where all packets are different (since these streams have a very long period, the server cannot pre-compute the stream in advance).

1.2 Why is Security for Broadcasts Hard?

In this section we analyze why efficient security protocols are challenging to design for broadcast environments. In contrast, a variety of protocols exist that provide efficient and secure protocols for authentication, signature, or confidentiality in point-to-point communication: SSL [FKK96a] and the standardized successor protocol TLS [DA99], and IPsec [KA98b, KA98a].

We now consider the security requirements of real-world broadcast applications and we investigate why these security requirements are much harder to achieve for broadcast communication with untrusted receivers, than in point-to-point communication.

1.2.1 Broadcast Authentication

The power of broadcast is that one packet can reach millions of receivers. This great property is unfortunately also a great danger: an attacker that sends one malicious packet can reach millions and just a single malicious network packet may be enough to cause a computer to lock up or reboot. An example of such an attack is the Windows Teardrop attack [CER97], which causes computers with the Windows NT operating system to halt.

This illustrates the importance of authentication. Unfortunately, efficient broadcast authentication is a challenging problem.

In the two-party (point-to-point) communication case, we can achieve data authentication through a purely symmetric mechanism[1]: the sender and the receiver share a secret key to compute a message authentication code (MAC) of all communicated data. When a message with a correct MAC arrives, the receiver is assured that the sender generated that message.

[1] We explain *symmetric cryptography* and *message authentication codes* (MAC) in Section 2.3.

Symmetric MAC authentication is not secure in a broadcast setting, where receivers are mutually untrusted. The symmetric MAC is not secure: every receiver knows the MAC key, and could thus impersonate the sender and forge messages to other receivers. Intuitively, we need an asymmetric mechanism to achieve authenticated broadcast, such that every receiver can *verify* the authenticity of messages it receives, without being able to *generate* authentic messages.[2]

The property we seek is asymmetric, so it is natural to consider asymmetric cryptography, for instance, a digital signature. Digital signatures have the required asymmetric property: the sender generates the signature with its private key, and all receivers can verify the signature with the sender's public key. A digital signature provides non-repudiation, which is a much stronger property than authentication. Unfortunately, digital signatures have a high cost: they have a high computation overhead for both the sender and the receiver, as well as a high communication overhead. Since we assume broadcast settings where the sender does not retransmit lost packets, and the receiver still wants to immediately authenticate each packet it receives, we would need to attach a digital signature to each message. Because of the high overhead of asymmetric cryptography (as we show in Section 2.3), this approach would restrict us to low-rate streams and senders and receivers with powerful workstations. To deal with the high overhead of asymmetric cryptography, we can try to amortize one digital signature over multiple messages. We discuss this in Section 1.2.2. However, such an approach is still expensive in contrast to symmetric cryptography, since symmetric cryptography is in general 3 to 5 orders of magnitude more efficient than asymmetric cryptography.

To achieve higher efficiency with asymmetric cryptography, a sender could use short keys during a short time period (too short for an attacker to break the key with high probability), and to keep the sender and receivers time synchronized. Unfortunately, such an approach has many drawbacks. Consider a system that uses the RSA digital signature algorithm with 512-bit long keys, where the sender signs every packet with the current private key [RSA78]. Assume that every public/private key pair is valid for 5 minutes. Using a 512-bit long RSA key is probably about the smallest permissible key size today (even for short-time usage applications), as those keys can be factored today using only 2% of the computation required to break a 56 bit DES key [CWI99].

[2] In fact, Boneh, Durfee, and Franklin show that a general broadcast authentication protocol can be converted into a signature protocol [BDF01].

With today's resources and factoring technology, such a key can be factored in a few hours. The advantage is that the receiver can immediately verify the signature of the packet right after reception (i.e., no authentication delay). This approach also provides robustness to packet loss, as long as the receiver has a reliable mechanism to get the current public verification key. Unfortunately, this approach also has many drawbacks:

- **High computation overhead.** Despite the short key, a 800 MHz Pentium III workstation today can only sign about 640 messages per second, and verify about 6400 messages per second. For most applications, this would tie up the majority of computation resources. The computation overhead is overwhelming on smaller architectures. For example, a Palm Pilot or RIM pager can only generate between 0.17 and 0.4 signatures per second, and verify between 1.6 and 10 signatures per second [BCH+00].

  Despite recent progress in digital signatures, RSA still provides one of the fastest signature verification for digital signatures based on number-theoretic assumptions.

- **High communication overhead.** With a 512 bit RSA key, each signature is 64 bytes long, which is too long for many applications. Recent signature algorithms have shorter signatures, but most of them have a higher computation overhead than RSA [CGP01, LV00]. Furthermore, the sender needs to update the public key every 10 minutes, which has an additional overhead.

- **Long-lived devices.** Some of the devices will be deployed for several years. Unfortunately, 512 bit long RSA keys will not be secure as factoring algorithms advance and processors continue to double in speed every 18 months.

- **Not perfectly robust to packet loss.** If the packets that carry the new public key are lost, the receivers cannot authenticate subsequent packets.

- **Time synchronization requirement.** The receivers need to be loosely time synchronized with the sender; however, the time synchronization error can be on the order of minutes.

For efficient broadcast authentication, we design two protocols that rely on purely symmetric cryptography: the TESLA protocol (Chapter 3) and the BiBa broadcast authentication protocol (Chapter 4). Both protocols require time synchronization between the sender and the receiver. Furthermore, both protocols

use novel techniques to provide the asymmetric property required for broadcast authentication. TESLA achieves the asymmetry through time delayed key disclosure, and BiBa is a new combinatorial signature.

To summarize, a viable broadcast authentication protocol must have the following properties:

- Low computation overhead for generation and verification of authentication information.
- Low communication overhead.
- Limited buffering required for the sender and the receiver (timely authentication for each individual packet).
- Strong robustness to packet loss.
- Scales to a large number of receivers.

1.2.2 Broadcast Signature

Some applications require that each message of the broadcast stream is digitally signed, to provide non-repudiation of origin in a court of law, for example stock market data, on-line auctions. Digital signatures require an asymmetric property, such that the signer can generate a signature, and the receivers can only verify (i.e., not generate) the signature. Compared to symmetric cryptography, asymmetric cryptographic primitives are orders of magnitude slower as we show in Section 2.3. Hence, if the receiver needs to compute an asymmetric cryptographic primitive for each packet it receives, the receiver needs substantial computation power. A more efficient approach is to amortize one asymmetric operation over multiple packets. We present the EMSS and MESS protocols which amortize one digital signature over thousands of packets.

The requirements are similar to the requirements of broadcast authentication. The properties of an ideal signature protocol for broadcast streams are:

- Low computation overhead for generation and verification of authentication information
- Low communication overhead
- No buffering required for the sender and the receiver
- Instant signature verification for each individual message
- Strong robustness to packet loss
- Scales to large number of receivers

1.2.3 Broadcast Data Integrity

Data integrity ensures that the data was not modified or deleted in any unauthorized way. In this work, we achieve integrity through authentication. Authentication ensures that the data originates from the claimed source, and that the data was not modified in transit. To detect unauthorized data deletion, we can use a variety of techniques, for instance adding a counter to data will allow the recipient to detect missing data. However, in most broadcast settings, the sender does not retransmit lost packets (or missing data) as we discuss in Section 1.1, so we do not need a counter. Hence, we achieve data integrity through data authentication.

1.2.4 Confidential Broadcasts and Restricting Access to Legitimate Receivers

Most contemporary broadcast environments allow any receiver to receive broadcast information. For instance, wireless environments allow anybody within the sending range to receive the broadcast. In IP multicast, for example, any receiver can subscribe to a multicast group. However, in many applications the sender wants to *restrict access* and to control which receivers can receive broadcast information, for example in subscription-based information distribution such as stock quote feeds, weather information, news, or sports broadcasts. The general approach for achieving this requirement is to encrypt broadcast information with a secret key $\mathcal{X}$. Only the sender and legitimate receivers know $\mathcal{X}$. Any illegitimate receiver that does not know $\mathcal{X}$ will not be able to decrypt the broadcast information. This approach also provides *confidentiality* of the broadcast information. So we can solve the access control problem for broadcast information if we can solve the key distribution problem.

Unfortunately, distributing a secret key efficiently to a large number of receivers is a challenge. The main problem is to update the key in dynamically changing groups, since a receiver should only be able to decrypt the broadcast stream while it is a member of the group. More concretely, the group key that the receiver gets when it joins the group should not allow it to decrypt content that was sent before it joined the group (we call this the *backward secrecy* property), and similarly, the member should not be able to decrypt any broadcast content after it leaves the group (we call this the *forward secrecy* property). *Group key secrecy* is another important property, which means that no outsider can find any group key. We discuss these properties in more detail in Section 6.1.1.

To achieve forward and backward secrecy, the sender updates the key after each member join or member leave event and sends the new group key to all legitimate receivers. Updating the group key in a *secure, scalable*, and *reliable* manner is challenging. In particular, achieving reliability is crucial in most current key distribution schemes, since a missing group key prevents a receiver from decrypting subsequent messages. Chapter 6 discusses these issues in more detail. To summarize, a viable key distribution protocol provides the following properties:

- Secure key update
  - Group key secrecy
  - Backward secrecy
  - Forward secrecy
- Scalability for large dynamic groups
- Reliability of key update messages
- Low computation overhead
- Low communication overhead (small key update messages)

1.3 Security Requirements for Broadcast Applications

We now discuss current and emerging broadcast applications and analyze their security requirements.

An important broadcast application is **software distribution**. Distribution of software over the Internet is increasing rapidly. For popular and large software packages, broadcast distribution is probably the most effective and economical distribution mechanism, in particular for security critical updates. For example, consider the dissemination of anti-virus software updates and operating system patches. Modern computer viruses and worms use the Internet to propagate, and can spread quickly and infect hundreds of thousands of connected computers within hours. For instance, the recent Code Red worm infected over a quarter million Microsoft Windows NT servers within hours [CER01a, CER01b]. To counteract these worms, we could distribute anti-virus software updates or operating-system patches through broadcasts, with the hope that we can protect the computers before they are attacked. The important security requirement for software distribution is that all receivers must authenticate the origin of the data, such that an adversary could

not distribute rogue code. It may appear that a digital signature on the entire data suffices. In practice, however, the sender usually uses a technique such as forward error correction (FEC) to achieve reliable content distribution [BLMR98, DF02, RMTR02]. These techniques encode the data into many small chunks, the receivers collect the chunks, and reconstruct the data. If an adversary injects a single bad chunk, the reconstruction fails, or produces garbled data and the signature verification would fail. Hence, injecting bogus packets poses a serious risk for a denial-of-service attack. Due to the exceedingly large number of combinations, it is computationally infeasible for the receiver to try to guess good chunks and attempting to reconstruct the data until the digital signature matches. A better approach is to use authentication on each chunk, so the receiver can authenticate the origin of every individual chunk before it further processes it.

An additional requirement may be to restrict access, so only legitimate receivers receive the data, which is necessary in many software distribution settings. We discuss the access control issue further in Section 1.2.4.

Another important application that requires broadcast communication is the distribution of **audio and video data**. Due to server limitations, current network bandwidth limits, and the relatively high bandwidth requirements of high-fidelity video streams, popular content requires broadcast for scalability. For instance popular sports events (e.g., the soccer world cup final), international events (e.g., the opening ceremony of the Olympic games), new video clips by music celebrities, or important political speeches, may attract up to hundreds of millions of viewers. Such events would require a tremendous server infrastructure to distribute the information by point-to-point communication. In most applications each receiver needs to authenticate the data origin of each message it receives to prevent an attacker from sending bogus content and potentially hijacking the entire stream. In settings where the information distributor restricts access to receivers who pay for the content, the sender may use access control techniques, which we discuss further in Section 1.2.4.

Audio and video broadcasts are also important in corporate environments, for instance for employee training and information distribution. Content authentication may not be necessary within a closed corporate network. Restricting access to legitimate receivers, however, may be necessary for sensitive and competition-critical information.

Another important broadcast application is **real-time stock market quote feeds**. Distributing the current market information such that millions of receivers get the data simultaneously is a challenge. Security is also an important

requirement in this setting: data authenticity is paramount for the receivers, and the information distributors want confidentiality, such that only the paying receivers get the information.

Sensor networks are an emerging technology. Sensors may monitor a wide variety of events, such as sound, temperature, light, cars passing by, motion, water level, etc. The number of applications are endless. These sensors usually communicate over a wireless channel, hence broadcast is a natural communication method. Because wireless communication is easy to eavesdrop or to spoof a message, many sensor networks require confidentiality and authentication. We discuss sensor networks and their applications in detail in Chapter 7.

The **air traffic control** system used today relies heavily on broadcast: GPS signals, airport landing system signals, and radio communications. Unfortunately, the infrastructure deployed today does not use cryptographic methods to protect against eavesdropping or fake signals. To guarantee maximum security for the flights, air traffic control should at the least use strong authentication on all broadcast signals, and if possible, restrict access to legitimate receivers.

1.4 Novel Contributions

This book introduces the following novel contributions:

- We propose to use time to achieve the asymmetry necessary in broadcast authentication (receivers can only verify, but not generate any valid authentication information). The TESLA broadcast authentication protocol achieves asymmetry through delayed key disclosure. Through the use of one-way key chains or Merkle hash trees, we achieve perfect robustness to packet loss. The TIK extension exploits precise time synchronization and enables receivers to instantly authenticate a packet. In case of loosely synchronized clocks, we introduce TESLA-IA (TESLA with instant authentication), where we shift from receiver-side buffering to sender-side buffering.

- We introduce use of multi-way collisions for a signature. This is a novel approach for constructing signatures based on one-way functions without trapdoors. The resulting signature provides low verification overhead. We design the BiBa broadcast authentication protocol based on the BiBa signature, which achieves a previously unachieved set of properties (instant authentication, scalability, robustness to packet loss, efficient authentication, reasonable communication overhead, loose time synchronization).

- The EMSS/MESS stream signature protocols use multiple hash links to achieve robustness to packet loss. The HTSS stream authentication protocol features a previously unachieved low communication overhead with instant authentication. This efficiency stems from observing the sending order of packets, and nodes of the Merkle hash tree; and by minimizing the redundancy of sending the Merkle hash tree nodes.

- The ELK large-group key distribution protocol introduces several novel ideas. The ELK protocol reduces the broadcast communication overhead for group key updates and introduces several mechanisms for achieving reliability. ELK introduces a protocol that does not require any broadcast message after a member joins the group at any time, or after a member leaves the group at a predicted time. A novel key update protocol enables us to reduce the key size, without introducing a vulnerability to dictionary attacks. The key update protocol also enables us to compress key update messages, by exploiting receiver computation. Finally, we reduce the size of redundant key update packets (for achieving robustness to packet loss) by selecting only the key update components which are useful for the majority of group members.

- As a case study, we consider sensor networks with highly resource-starved devices. Despite the hardware constraints, we design secure protocols for point-to-point communication, and adapt the TESLA broadcast authentication protocol to this environment.

1.5 Scope of this Book

This book presents a number of new protocols to secure broadcast communication. The emphasis of the text is on the new mechanisms that these protocols introduce. We describe our mechanisms at a fairly high level, so that the description is general enough to apply to a wide variety of settings. We omit detailed security proofs of our mechanisms, but we give general security arguments and refer to publications containing more detailed security arguments.

1.6 Book Overview

This book presents new methods and protocols for secure broadcast communication. Rather than present protocols for specific settings, we present a flexible framework and a set of building blocks appropriate to a wide variety of settings. To ensure generality of our methods, we intentionally leave out spe-

cific implementation details. However, we do consider specific applications for each protocol, we do walk through a specific solution, and we do discuss performance and tradeoffs. The tools in this book support secure protocols for any broadcast network, including IP multicast, wireless networks, and satellite distribution networks. Here is an outline of this book.

- Chapter 1 is this introduction. We present applications for broadcast networks and discuss their security requirements. We identify broadcast authentication, broadcast signature, and key distribution as the essential and fundamental security requirements for broadcast networks.

- Chapter 2 introduces the basic cryptographic primitives that we use.

- Chapter 3 presents the TESLA protocol, a highly efficient broadcast authentication protocol. TESLA uses time (delayed key disclosure) to achieve the asymmetry property required for secure broadcast authentication. The main features of TESLA are: low sender and receiver computation overhead (around one MAC function computation per packet), low communication overhead, and perfect robustness to packet loss. TESLA requires loosely synchronized clocks between the sender and the receiver, and the authentication is slightly delayed. However, we describe a variant of TESLA with instant authentication, which requires sender buffering instead of receiver buffering, but the receiver can authenticate the message as soon as the packet arrives.

- Chapter 4 presents the BiBa signature algorithm, a new approach to design signature algorithms based on one-way functions without a trapdoor (these signatures are sometimes called one-time signatures). BiBa is an acronym for bins and balls signature. We use bins and balls as an analogy to describe BiBa: to sign a message, the signer uses the message to seed a random process which throws a set of balls into bins. When enough balls fall into the same bin, the combination of those balls constitute a signature. To achieve an asymmetry between the signer and the forger, the BiBa signature exploits the property that the signer who has a large number of balls finds a signature with high probability, but a forger who only has a small number of balls has a negligible probability to find a signature. To the best of our knowledge, the BiBa signature is one of the fastest signatures today for verification. Signature generation is more expensive, but only requires two sequential hash function computations if the signer has many parallel processors. BiBa is about twice as fast than most previous one-time signa-

ture algorithms, and the signature size is less than half as large than most previous signatures based on one-way functions without trapdoors.

Using the BiBa signature as a basis, we design the BiBa broadcast authentication protocol. BiBa requires loose time synchronization between the sender and receivers, but in contrast to TESLA, the authentication is instant and neither the sender nor the receiver need to buffer messages. BiBa is also perfectly robust to packet loss, and scales well. The BiBa broadcast authentication protocol thus achieves a unique set of properties.

TESLA and BiBa feature new approaches to achieve the asymmetric property required by the broadcast authentication: TESLA uses time delay, and BiBa is a new combinatorial signature.

- Chapter 5 introduces a set of broadcast signature protocols: EMSS, MESS, and HTSS. EMSS and MESS use sequences of hashes to amortize one expensive digital signature operation over many messages. In a nutshell, the sender computes the hash of a packet and adds it to later packets. Periodically, the sender signs data packets with a digital signature. Due to the sequence of hashes, the signature extends transitively over the previous packets, i.e., the receiver can follow the sequence of hashes to verify previous packets. Because the hash of a packet appears redundantly in later packets, we achieve tolerance to packet loss. Given a loss probability, we analyze in how much redundancy we need to achieve a high probability that we can verify the signature of a packet. For the special case where the sender knows the entire stream content in advance, we design the broadcast signature protocol HTSS. HTSS constructs a hash tree over all the messages and the sender only needs to sign the root of the tree. Naively, such an approach would seem to require high communication overhead, but we present an efficient method of encoding to make this approach viable.

  Table 1.1 shows an overview of the features of our broadcast authentication and signature protocols presented in Chapters 3 through 5. To select an appropriate protocol, we can choose from three broadcast signature protocols if we need non-repudiation: EMSS, MESS, HTSS. However, many broadcast applications require authentication, so TESLA is ideally suited if loose time synchronization is possible and a short authentication delay is tolerable. If time synchronization is less accurate, or if we want the lowest possible end-to-end authentication delay, the BiBa broadcast authentication protocol is appropriate. If the data may be cached for extended time peri-

	TESLA	TESLA-IA	TIK	BiBa	EMSS / MESS	HTSS
Authentication (A) or signature (S)	A	A	A	A	S	S
Time synchronization	Y	Y	Y	Y	N	N
Authentication delay	Y	N	N	N	Y	N
Sender buffering	N	Y	N	N	N	Y
Receiver buffering	Y	N	N	N	Y	N
Real-time streams	Y	Y	Y	Y	Y	N
Robustness to packet loss	Y	N	Y	Y	N	N
Communication overhead (bytes)	24	32	24	128	40	32
Generation overhead	1	1	1	2048	1	2
Verification overhead	2	2	30	100	1	10

Table 1.1. This table compares the authentication and signature schemes we propose. TESLA is the basic protocol we describe in Chapter 3. TESLA-IA is the instant authentication extension we present in Section 3.5.1. BiBa is the stream authentication protocol we present in Chapter 4. Chapter 5 presents the EMSS, MESS, and HTSS stream signature schemes. The communication overhead row lists the approximate per-packet size of the authentication or signature information. The unit for the generation and verification overheads is the approximate number of hash function computations.

ods and time synchronization is not possible, we suggest use of one of the signature protocols to provide authentication.

- Chapter 6 discusses the problem of how to distribute a key securely and reliably to large numbers of receivers, in a scalable manner. We identify reliability as a crucial factor to achieve scalability, and design multiple protocols that greatly enhance reliability. For instance, we design a new protocol for adding a new receiver to the group that does not require any broadcast message. We also design a new protocol that updates the group key after a member leaves the group at a predicted time, without sending a broadcast message. These two protocols greatly enhance scalability, because fewer messages need to be sent reliably. We furthermore present a new key update protocol that features compressed key update messages. Our protocol enables such short key update messages that the sender can send them along with regular data messages. We present the ELK protocol, which uses all

these protocols to achieve a highly efficient and scalable key distribution protocol.

- Chapter 7 examines a case study in an extremely resource-constrained sensor network. We analyze the security requirements of sensor networks. We present the Sensor Network Encryption Protocol (SNEP). SNEP secures point-to-point communication links and provides confidentiality, authentication, and freshness. Broadcast is a natural method of communication in wireless networks, and authentication of broadcast messages is important for many applications. By designing the μTESLA protocol, we demonstrate that the TESLA protocol can scale down to minimal environments and that broadcast authentication is practical even in sensor network environments.

- Chapter 8 reviews related work.

- Chapter 9 concludes this book. We end with a discussion of open research problems in the area of secure broadcast.

Chapter 2

CRYPTOGRAPHIC FUNDAMENTALS

This chapter presents terminology, notation, requirements, and a variety of cryptographic primitives. These cryptographic primitives serve as building blocks for our protocols.

2.1 Broadcast Network Requirements

We consider security protocols for a wide range of networks and systems: IP multicast, satellite distribution, and wireless communication. Therefore, we work with a broad, generic network model. We need two communication primitives in our model: *broadcast communication* (one to many) and *point-to-point communication* (one to one, also called *unicast*). In the unicast primitive:

- the sender sends a packet to a single receiver,
- with non-zero probability, the receiver receives the packet within a bounded time period.

In the broadcast primitive:

- the sender sends a packet to a set of receivers,
- for each receiver, with non-zero probability, the receiver receives the packet within a bounded time period.

Useful real-world broadcast networks all provide these minimal properties.

A sender distributes *data* (or *information*), splits it up into *messages*, and sends these messages in a series of *network packets* across the network. We often use loose terminology and treat the terms *message* and *packet* as if they

A. Perrig et al., *Secure Broadcast Communication*

were interchangeable. The term *message* emphasizes the information that the sender sends, and the term *packet* emphasizes the data that traverses the network.
When sender S sends a message M to receiver B, we write

$$S \rightarrow B : M$$

When sender S broadcasts message M, we write

$$S \rightarrow * : M$$

To concatenate messages M_1 and M_2, we write

$$M_1 \mid\mid M_2$$

2.2 Cryptographic Primitives

This section introduces cryptographic primitives. Our following definitions are informal. Luby [Lub96], or Goldwasser and Bellare [GB99] give more formal definitions of the terms below. Our explanation of the terms is brief, and we refer the reader to a standard cryptography text, such as Menezes, van Oorschot, and Vanstone [MvOV97], for a full discussion.

2.2.1 Symmetric and Asymmetric Cryptography

Most cryptographic primitives belong to one of two broad categories: *symmetric* or *asymmetric* cryptography. In *asymmetric cryptography* we normally use pairs of keys — one key is public and one key is private (or secret). In contrast, in *symmetric cryptography* the sender and receiver usually share the same (secret) key.

RSA is an example of asymmetric cryptography [RSA78]. In RSA digital signatures, the sender uses the private key to digitally sign a message, and the receiver can verify the signature with the public key.

2.2.2 One-Way Functions and Hash Functions

We define *one-way functions* in the usual way. Informally, a *one-way function* (OWF) is a function that is easy to compute but computationally infeasible to invert. Suppose x is a random string of length k bits. The function $F : \{0,1\}^k \rightarrow \{0,1\}^\ell$ is a *strong one-way function* if F can be computed in time polynomial in k and ℓ, and if given $y = F(x)$ (but not x) it is almost always computationally infeasible to find x' such that $F(x') = y$.

A *hash function* H maps a binary string of arbitrary length to a binary string of fixed length: $H : \{0,1\}^* \rightarrow \{0,1\}^\ell$ (in this case H maps to ℓ-bit long strings).

We use hash functions with one or more of the following three properties in our constructions:

- *Strong one-way.* For a randomly chosen x, given $y = H(x)$ it is almost always computationally infeasible to find any x', such that $H(x') = y = H(x)$. (Other researchers sometimes call this *preimage resistance.*)
- *Weak collision resistance.* Given a random x, it is almost always computationally infeasible to find $x' \neq x$, such that $H(x) = H(x')$. (Other researchers sometimes call this *second preimage resistance.*)
- *Strong collision resistance.* It is computationally infeasible to find two distinct x and x', such that $H(x) = H(x')$. (Other researchers sometimes call this *collision resistance.*)

Candidates for hash functions include MD5 [Riv92], SHA-1 [Lab95], and RIPEMD-160 [DBP96].[1]

A *trapdoor one-way function* (OWFT) is a one-way function $f : X \rightarrow Y$ with the additional property that given $f(x)$ and some additional information T (called *trapdoor information*) it becomes computationally feasible to find $x' \in X$ such that $f(x) = f(x')$.

Most OWFTs rely on number-theoretic properties. OWFTs are predominantly used to design digital signatures and public-key encryption algorithms. Bellare and Goldwasser review examples for one-way functions with trapdoor [GB99], e.g., the RSA function, or the squaring trapdoor function.

---

[1] We can also construct one-way functions from a pseudo-random permutation, or block cipher [MvOV97]. In the following discussion, $y = E_k(x)$ is a block cipher using a key k to map input block x to output block y.

Matyas, Meyer, and Oseas propose the construction $H_0 = IV$ (where IV is an initialization vector), $H_i = E_{g(H_{i-1})}(x_i) \oplus x_i$ (the function g simply maps to a key for the block cipher E) [MMO85]. On input $x_1 \ldots x_t$, the output is H_t. They also describe the Davies-Meyer construction: $H_0 = IV$, $H_i = E_{x_i}(H_{i-1}) \oplus H_{i-1}$. Winternitz improves this construction by applying a universal hash to the key of E in each round [Win84].

Miyaguchi et al. describe another construction [MKOM90]:
$H_0 = IV$, $H_i = E_{g(H_{i-1})}(x_i) \oplus x_i \oplus H_{i-1}$.
The function g simply produces keys of suitable length for the encryption function E. Independently, Preneel also published this hash [Pre93], which is now known as Miyaguchi-Preneel.

Preneel, Govaerts, and Vandewalle analyze these constructions [PGV97].

2.2.3 Pseudo-Random Generator (PRG)

A *pseudo-random generator* (PRG) G takes a k-bit input (a *seed*) and outputs a pseudo-random sequence of bits. G is *pseudo-random* if it is computationally infeasible to distinguish its output from a truly random source.

A *stream cipher* is often constructed with a PRG that produces a pseudo-random stream from a secret key serving as the seed. To encrypt the data, we use the pseudo-random stream as a *one-time pad*: we compute the bitwise exclusive-or (XOR) of the plaintext and pseudo-random stream (the n-th bit of the plaintext is XOR-ed with the n-th bit of the pseudo-random stream) to yield the ciphertext. Since both parties share the same key (seed), stream ciphers are also symmetric cryptographic algorithms.

Impagliazzo, Levin, and Luby show how to construct a PRG from a one-way function [ILL88, HILL99].

2.2.4 Message Authentication Code (MAC)

A *message authentication code* (MAC) $G : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^\ell$ takes as input a k-bit key and a message, and outputs a ℓ-bit *authentication tag*. A receiver who wants to ensure that messages originate from the claimed sender, can verify message authenticity by: 1) sharing a secret key with the sender; 2) the sender adds an authentication tag (or MAC) computed with the shared key to every message it sends; 3) the receiver computes the MAC function using the shared key to verify that the authentication tag is correct. Because the same key is shared between the sender and the receiver, this is also a type of symmetric cryptography. A secure MAC function prevents an attacker (without knowledge of the secret key) from computing the correct MAC for a new message. A MAC achieves authenticity for point-to-point communication, because a receiver knows that a message with the correct MAC must have been generated either by itself or by the sender. So when the receiver gets a fresh message with a correct MAC that it has not generated itself, the message must originate from the sender.

2.2.5 Pseudo-Random Function (PRF)

Goldreich, Goldwasser, and Micali proposed the idea of a *pseudo-random function* (PRF) family [GGM86]. A function family $\{f_k\}_{k \in \{0,1\}^\ell}$ (where ℓ is the key length, a security parameter) is a pseudorandom function family if it is computationally infeasible for any adversary who does not know the secret key k to distinguish between a function f_k (where k is chosen randomly and kept secret) and a random function.

We can construct a PRF in a variety of ways [GB99]. An efficient approach is to use a *pseudo-random permutation* (PRP) (i.e., a block cipher) for a limited output length to construct the PRF.

2.3 Efficiency of Cryptographic Primitives

Asymmetric cryptography can provide all the cryptographic primitives we need, but we prefer symmetric primitives because of their efficiency. Symmetric primitives are often 3 to 5 orders of magnitude faster than their asymmetric counterparts. We illustrate this with some concrete experiments. We measured the speed of widely-used cryptographic primitives on a 800 MHz Pentium III Linux workstation. We used the optimized implementations of the OpenSSL libraries [Ope01]. Table 2.1 shows the results of our experiments.

Algorithm	time per operation	operations per second
RSA 512 sign	1.6ms	641
RSA 512 verify	0.16ms	6404
RSA 1024 sign	8.7ms	115
RSA 1024 verify	0.48ms	2094
RSA 2048 sign	53.9ms	19
RSA 2048 verify	1.7ms	605
DES	0.5μs	$1.9 \cdot 10^6$
RC5-32-12	0.2μs	$4.9 \cdot 10^6$
Rijndael 128	0.7μs	$1.5 \cdot 10^6$
MD5	1.0μs	$1.0 \cdot 10^6$
SHA-1	1.5μs	$0.66 \cdot 10^6$
HMAC-MD5	2.5μs	$0.40 \cdot 10^6$
UMAC 2/8	0.5μs	$1.8 \cdot 10^6$

Table 2.1. The efficiency of widely-used cryptographic primitives. All experiments were performed on a 8-byte input size, using the OpenSSL libraries [Ope01] on a 800 MHz Pentium III Linux workstation.

We use RSA as the reference for asymmetric cryptography. We consider three versions of RSA: 512-bit RSA, which offers low-security (512-bit RSA is currently about 50 times easier to break than 56-bit keys of the data encryption standard (DES) [CWI99]); 1024-bit RSA (a currently popular choice); and 2048-bit RSA, which some believe will become necessary in year 2020 [LV01].

All the versions use 65537 as the public exponent.[2] For the symmetric-key encryption systems, we list the Data Encryption Standard DES [Nat77] (for low-security applications with the 56-bit key), the fast RC5 scheme [Riv94] (64-bit block size), and the Advanced Encryption Standard (AES) Rijndael [DR99] (128-bit block size). All of these encryption schemes are block ciphers, and they map a fixed-size input block to a fixed-size output block based on the secret key.

The table also compares the cryptographic hash functions MD5 [Riv92] and SHA-1 [Lab95]. In addition, it compares MAC functions. A widely used MAC is HMAC-MD5 [BCK96, BCK97]. UMAC is a high-speed MAC by Black et al. [BHK+99] that is particularly efficient for long messages. (The version of UMAC that we list in Table 2.1 produces 64-bit authentication tags; it takes about twice as long to generate 128-bit authentication tags.)

2.4 Commitment Protocols

Commitment functions lock a secret s without revealing s. For example, suppose Alice wants to seal a prediction so she can later reveal it. She can choose a secret value s, compute $c = F(s)$, and publish c. Later she can reveal that she chose s by publishing s (also called *open* the commitment). Anybody can verify that she chose s by computing $F(s)$ and seeing if it equals the commitment value c. To work, this requires that F must be one-way and strong collision resistant (see Section 2.2.2).

If an adversary could predict s or try all possibilities for s, the adversary would know which s the commitment c corresponds to. Using standard *bit-commitment schemes* prevent this attack [Bra88, BCC88, Nao90, Sch96]. In this book, we only use commitment functions in contexts where s is unpredictable and sufficiently long, so the pre-computation attack is computationally infeasible. For our purposes, a one-way function is sufficient as a commitment function.

In later chapters, we use a pseudo-random function G as our commitment function. To commit to s, we use s as a key, and apply G to a constant value: $G_s(0) = c$. If we consider $F(s) = G_s(0)$, then F is a strong one-way function. We assume that G provides *strong collision resistance*.

[2]Using 3 as the public exponent would speed up public-key operations by about 8 times.

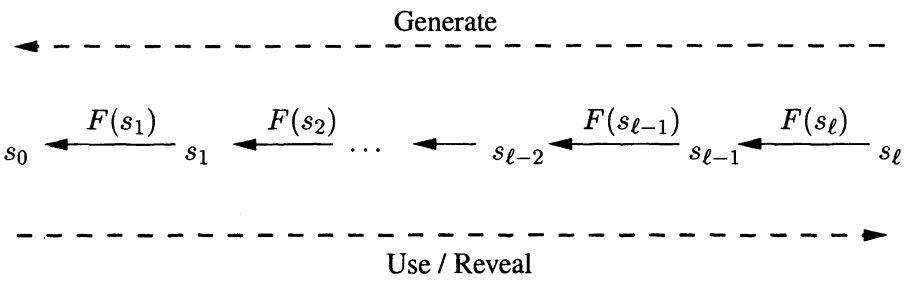


Figure 2.1. One-way chain example. The sender generates this chain by randomly selecting s_ℓ and repeatedly applying the one-way function F. The sender then reveals the values in the opposite order.

2.4.1 One-Way Chain

In many of our protocols, we need to commit to a sequence of random values. For this purpose, we repeatedly use a commitment function to generate a *one-way chain.* One-way chains are a widely-used cryptographic primitive.[3] Figure 2.1 shows this construction. To generate a chain of length ℓ we randomly pick the last element of the chain s_ℓ. We generate the chain by repeatedly applying a one-way function F. Finally, s_0 is a commitment to the entire one-way chain, and we can verify any element of the chain through s_0, e.g., to verify s_i, we check that $F^i(s_i) = s_0$. More generally, s_i commits to s_j if $i < j$ (to verify check that $F^{j-i}(s_j) = s_i$). Usually, we use the chain in this order $s_0, s_1, \ldots, s_{\ell-1}, s_\ell$. How can we store this chain? We can either create it all at once and store each element of the chain, or we can just store s_ℓ and compute any other element on demand. In practice, a hybrid approach helps to reduce storage with a small recomputation penalty.

2.4.2 Merkle Hash Tree

We normally disclose elements in the one-way chain sequentially, since any element discloses all previous elements as well. In some cases, however,

[3]One of the first uses of one-way chains was for one-time passwords by Lamport [Lam81]. Haller later used the same approach for the S/KEY one-time password system [Hal94]. One-way chains are also used in efficient one-time signature algorithms [EGM90, Mer88, Mer90, Roh99]. One-way chains are also widely used in (micro-)payment mechanisms [AMS97, HSW96, Ped97, RS97], server-supported nonrepudiation [ATW97], and for conditional anonymity [HT96]. One-way chains were also used for efficient certificate revocation [Mic96], to authenticate link-state routing updates [HPT97, Zha98, Che97], and for broadcast authentication [PCTS00, Per01]. Jakobsson [Jak02], and Coppersmith and Jakobsson [CJ02] propose a storage efficient mechanism for one-way chains: a one-way chain with N elements only requires $O(\log(N))$ storage and $O(\log(N))$ computation to access an element.

we need to open the commitments in any order, or we only open few of the commitments. We use the Merkle hash tree construction to achieve this property [Mer80].

To commit to the secret values s_i ($0 \leq i < 2^n$) we place the values s_i at the leaves of a binary tree of depth $n + 1$. We compute the interior nodes of the tree as follows. Assume node m has child nodes m_{left} and m_{right}. We compute the value of m with a one-way function F on the concatenation of the two child nodes: $m = F(m_{left} \mid\mid m_{right})$. The root value of the tree commits to all leaves. To open the commitment to a value s_i, we disclose s_i along with all the *verification values*, i.e., all the nodes necessary to verify the path from s_i up to the root. The verification values include only one node from each level of the tree; thus the data to open a commitment has size $\log(N)$ where N is the number of leaves. Figure 2.2 shows an example of a Merkle hash tree, which commits to four values, with the commitment extension of the leaf nodes. The root node m_{03} commits to all s_i (s_0, s_1, s_2, s_3). For example, to open the commitment to s_2, we publish s_2, m_3, and m_{01}, and verify that m_{03} is equal to $F(m_{01}||F(F(s_2)||m_3))$. We first compute a basic commitment (as described in Section 2.4) on the leaves s_i. This allows us to verify the root and reveal a leaf without revealing its neighbor.

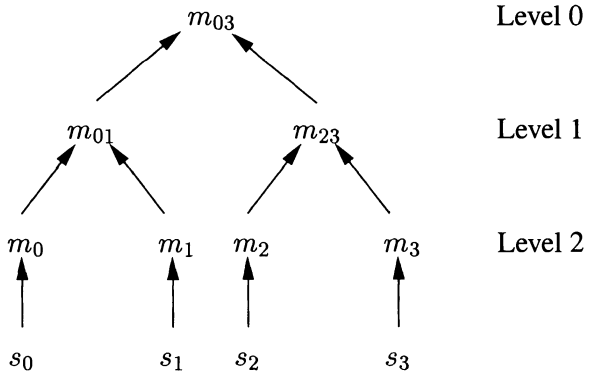


Figure 2.2. Merkle hash tree of depth 3. The root node m_{03} commits to all leaf nodes $s_0 \ldots s_3$. The initial blinding step $m_i = F(s_i)$ hides the sibling node when disclosing the verification values (the values necessary to recompute the path to the root).

2.4.3 Self-Authenticating Values

In our protocols we need short values that a receiver can efficiently authenticate. These values are about the same length as a message authentication code (MAC), so around 80 bits suffice for most current applications [LV99].

Rather than using a separate message authentication code (MAC) for each value (this would double the length of our value), we use values that a receiver can authenticate without any additional information. We call these values *self-authenticating values*.

We use commitment protocols for this purpose. If the receiver obtains a commitment c to a secret s over an authenticated channel, the secret s is a self-authenticating value, because the receiver can instantly authenticate it by verifying $F(s)$ is equal to c. We will usually use the one-way chain construction to generate a sequence of self-authenticating values.

Chapter 3

TESLA BROADCAST AUTHENTICATION

How can we authenticate broadcast messages? This chapter begins to answer our central question. We introduce TESLA, short for Timed Efficient Stream Loss-tolerant Authentication, a protocol for broadcast authentication. TESLA has a number of features, so for clarity of exposition, we develop TESLA in stages. We begin with a basic protocol, and we add additional features in subsequent stages. (In later chapters, we examine alternative approaches to broadcast authentication.)

3.1 Requirements for Broadcast Authentication

In Chapter 1, we argue that broadcast authentication is central to many applications. We now review our list of requirements for broadcast authentication from Section 1.2.1:

- Low computation overhead for generation and verification of authentication information
- Low communication overhead
- Limited buffering required for the sender and the receiver, hence timely authentication for each individual packet
- Robustness to packet loss
- Scales to a large number of receivers

These requirements immediately rule out several approaches. For example, signing each packet would introduce high overhead. Signing a long sequence

A. Perrig et al., *Secure Broadcast Communication*

of packets would make authentication vulnerable to dropped packets. Using message authentication codes (MACs) would require shared keys among all receivers, presenting a security problem.

These problems challenged the distributed systems community. It was not until 1997 that work began to appear proposing protocols for broadcast authentication.Section 8.2 reviews this work in detail — there is a great deal of creativity and important insights in previous work, but all previous solutions fall short in one way or another.

This chapter presents TESLA. The TESLA protocol meets all of the above requirements with low cost — but it has the following special requirements:

- The sender and the receivers must be at least loosely time-synchronized.
- Either the receiver or the sender must buffer some messages.

Despite the buffering, TESLA has a very low authentication delay. In typical configurations, the authentication delay is on the order of one round-trip delay between the sender and receiver.

In the next section, we present the basic TESLA protocol, and later sections present further enhancements. (Chapter 4 presents an alternative authentication protocol, BiBA, that does not require buffering.)

In this chapter, we first describe the basic TESLA protocol, and then several extensions and variations.

3.2 The Basic TESLA Protocol

3.2.1 Sketch of protocol

We first outline the main ideas behind TESLA.

As we argue in Section 1.2.1, broadcast authentication requires a source of asymmetry. TESLA uses time for asymmetry. We assume that receivers are all loosely time synchronized with the sender — up to some time synchronization error Δ, all parties agree on the current time. Here is a sketch of the basic approach:

- The sender splits up the time into time intervals of uniform duration. Next, the sender forms a one-way chain (see Section 2.4.1) of self-authenticating values (see Section 2.4.3), and assigns the values sequentially to the time intervals (one key per time interval). The one-way chain is used in the reverse order of generation, so any value of a time interval can be used to derive values of previous time intervals. The sender defines a disclosure

time for one-way chain values, usually on the order of a few time intervals. The sender publishes the value after the disclosure time.

- The sender attaches a MAC to each packet. The MAC is computed over the contents of the packet. For each packet, the sender determines the time interval and uses the corresponding value from the one-way chain as a cryptographic key to compute the MAC. Along with the packet, the sender also sends the most recent one-way chain value that it can disclose.

- Each receiver that receives the packet performs the following operation. It knows the schedule for disclosing keys and, since the clocks are loosely synchronized, can check that the key used to compute the MAC is still secret by determining that the sender could not have yet reached the time interval for disclosing it. If the MAC key is still secret, then the receiver buffers the packet.

- Each receiver also checks that the disclosed key is correct (using self-authentication and previously released keys) and then checks the correctness of the MAC of buffered packets that were sent in the time interval of the disclosed key. If the MAC is correct, the receiver accepts the packet.

One-way chains have the property that if intermediate values of the one-way chain are lost, they can be recomputed using later values. So, even if some disclosed keys are lost, a receiver can recover the key chain and check the correctness of packets.

The sender distributes a stream of messages $\{M_i\}$, and the sender sends each message M_i in a network packet P_i along with authentication information. The broadcast channel may be lossy, but the sender does not retransmit lost packets. Despite packet loss, each receiver needs to authenticate all the messages it receives.

We now describe the stages of the basic TESLA protocol in this order: sender setup, receiver bootstrap, sender transmission of authenticated broadcast messages, and receiver authentication of broadcast messages.

3.2.2 Sender Setup

TESLA uses self-authenticating one-way chains. The sender divides the time into uniform intervals of duration T_{int}. Time interval 0 will start at time T_0, time interval 1 at time $T_1 = T_0 + T_{int}$, etc. The sender assigns one key from the one-way chain to each time interval in sequence. The one-way chain

is used in the reverse order of generation, so any value of a time interval can be used to derive values of previous time intervals.

The sender determines the length N of the one-way chain $K_0, K_1, \ldots, K_N$, and this length limits the maximum transmission duration before a new one-way chain must be created. (Section 3.5.3 discusses how to handle broadcast streams of unbounded duration by switching in new chains.) The sender picks a random value for K_N. Using a pseudo-random function f, the sender constructs the one-way function F: $F(k) = f_k(0)$. The remainder of the chain is computed recursively using $K_i = F(K_{i+1})$. Note that this gives us $K_i = F^{N-i}(K_N)$, so we can compute any value in the key chain from K_N even if we do not have intermediate values. Each key K_i will be active in time interval i.

3.2.3 Bootstrapping Receivers

Before a receiver can authenticate messages with TESLA, it needs to be loosely time synchronized with the sender, know the disclosure schedule of keys, and receive an authenticated key of the one-way key chain.

Various approaches exist for time synchronization [Mil92, SLWL90, Mil94, LMS85]. TESLA, however, only requires loose time synchronization between the sender and the receivers, so a simple algorithm is sufficient. The time synchronization property that TESLA requires is that each receiver can place an upper bound of the sender's local time. TESLA offers *direct*, *indirect*, and *delayed synchronization* as three default options. We describe these variants in Section 3.4.

The sender sends the key disclosure schedule by transmitting the following information to the receivers over an authenticated channel (either via a digitally signed broadcast message[1], or over unicast with each receiver):

- Time interval schedule: interval duration T_{int}, start time T_i and index of interval i, length of one-way key chain.
- Key disclosure delay d (number of intervals).
- A key commitment to the key chain K_i ($i < j - d$ where j is the current interval index).

[1]Since this is only done rarely, it does not create inefficiencies or vulnerabilities to denial-of-service attacks. Section 3.6.1 describes this in detail.

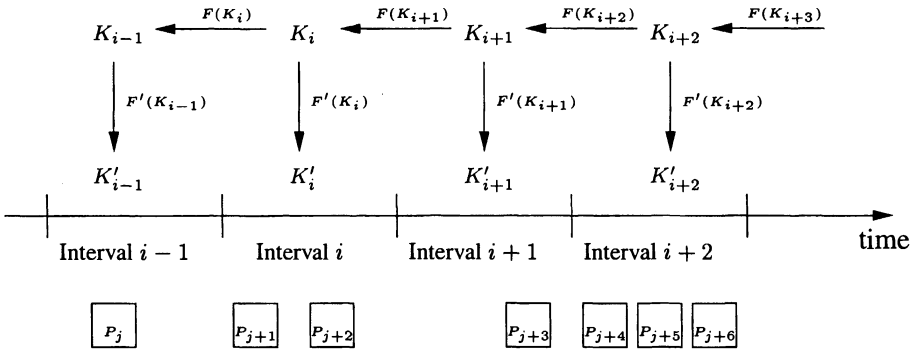


Figure 3.1. At the top of the figure, we see the one-way TESLA key chain (using the one-way function F), and the derived MAC keys (using the one-way function F'). Time advances left-to-right, and the time is split into time intervals of uniform duration. At the bottom of the figure, we can see the packets that the sender sends in each time interval. For each packet, the sender uses the key that corresponds to the time interval to compute the MAC of the packet. For example for packet P_{j+3}, the sender computes a MAC of the data using key K'_{i+1}. Assuming a key disclosure delay of two time intervals ($d = 2$), packet P_{j+3} would also carry key K_{i-1}.

3.2.4 Broadcasting Authenticated Messages

Each key in the one-way key chain corresponds to a time interval. Every time a sender broadcasts a message, it appends a MAC to the message, using the key corresponding to the current time interval. The key remains secret for the next $d - 1$ intervals, so messages sent in interval j effectively disclose key K_{j-d}. We call d the *key disclosure delay.*

As a general rule, using the same key multiple times in different cryptographic operations is ill-advised — it may lead to cryptographic weaknesses. So we do not want to use key K_j both to derive key K_{j-1} and to compute MACs. Using a pseudo-random function family f', we construct the one-way function F': $F'(k) = f'_k(1)$. We use F' to derive the key to compute the MAC of messages: $K'_i = F'(K_i)$. Figure 3.1 depicts the one-way key chain construction and MAC key derivation. To broadcast message M_j in interval i the sender constructs packet $P_j = \{M_j \mid\mid \text{MAC}(K'_i, M_j) \mid\mid K_{i-d}\}$.

Figure 3.1 depicts the one-way key chain derivation, the MAC key derivation, the time intervals, and some sample packets that the sender broadcasts.

3.2.5 Authentication at Receiver

When a sender discloses a key, all parties potentially have access to that key. An adversary can create a bogus message and forge a MAC using the disclosed

key. So as packets arrive, the receiver must verify that their MACs are based on *safe keys*: a *safe key* is one that is only known by the sender, and *safe packets* or *safe messages* have MACs computed with safe keys.

Receivers must discard any packet that is not safe, because they may have been forged.

We now explain TESLA authentication in detail: A sender sends packet P_j in interval i. When the receiver receives packet P_j, the receiver can use the self-authenticating key K_{i-d} disclosed in P_j to determine i. It then checks the latest possible time interval x the sender could currently be in (based on the synchronized clock, see Section 3.4 for details). If $x < i + d$ (recall that d is the key disclosure delay, or number of intervals that the key disclosure is delayed), then the packet is safe. The sender has thus not yet reached the interval where it discloses key K_i, the key that will verify packet P_j.

The receiver cannot yet verify the authenticity of packet P_j sent in interval i. Instead, it adds the triplet $(i, M_j, \text{MAC}(K'_i, M_j))$ to a buffer, and verifies the authenticity after it learns K'_i.

What does a receiver do when it receives the disclosed key K_i? First, it checks whether it already knows K_i or a later key K_j $(j > i)$. If K_i is the latest key received to date, the receiver checks the legitimacy of K_i by verifying, for some earlier key K_v $(v < i)$ that $K_v = F^{i-v}(K_i)$. The receiver then computes $K'_i = F'(K_i)$ and verifies the authenticity of packets of interval i.

Using a disclosed key, we can calculate all previous disclosed keys, so even if packets are lost, we will still be able to verify buffered, safe packets from earlier time intervals.

Note that the security of TESLA does not rely on any assumptions on network propagation delay, since each receiver locally determines if the packet is safe, i.e., whether the sender disclosed the corresponding key. However, if the key disclosure delay is not much longer than the network propagation delay, the receivers will find that the packets are not safe.

3.2.6 TESLA Summary and Security Considerations

Protocol 3.1 summarizes the basic TESLA protocol. The security of TESLA relies on the following assumptions:

- The receiver's clock is time synchronized up to a maximum error of Δ. (We suggest that because of clock drift, the receiver periodically re-synchronizes its clock with the sender.)

- The functions F and F' are secure PRFs, and the function F has weak collision resistance.[2]

1 **Sender setup**. The sender splits up the time into uniform time intervals of duration T_{int}. Time interval 0 starts at time T_0, time interval 1 at time $T_0 + T_{int}$, etc. The sender assigns a key to each time interval using a one-way chain: the sender randomly selects a key K_N for time interval N, and derives previous keys with a PRF F (that has the target-collision resistance property stated in Section 2.4): $K_i = F(K_{i+1})$. The sender picks a key disclosure delay $d \geq 2$, where the unit is in time intervals.

2 **Receiver setup**. The receiver engages in a protocol to receive the authentic parameters of the sender's key chain: T_0, T_{int}, d, and an authentic disclosed key K_i. Furthermore, the receiver synchronizes its clock with the sender and knows the maximum clock synchronization error Δ.

3 **Sending authenticated packets**. To send a message M in time interval j, the sender sends the following message (where F' is another PRF):

$$S \rightarrow * : M, \mathsf{MAC}(F'(K_j), M), K_{j-d}$$

When the receiver receives this message at local time t_r, it first infers the sending time interval from the disclosed key K_{j-d}, by authenticating K_{j-d} using the last authentic key it has stored, i.e., K_{j-d-2}. The receiver checks that $F^2(K_{j-d}) = K_{j-d-2}$. If the key is authentic, the receiver stores the new key K_{j-d}, it can use the keys K_{j-d-1} and K_{j-d} to authenticate stored packets, and it knows that the current packet was sent in time interval j. To check whether message M is safe, the receiver verifies that $t_r + \Delta < T_{j+d}$. If the message is safe, the receiver buffers the message and MAC, otherwise it discards the packet.

Protocol 3.1: Summary of the basic TESLA broadcast authentication protocol.

3.3 TIK: TESLA with Instant Key Disclosure

In ordinary TESLA, either the sender or the receiver must buffer messages. For the special case of wireless communication and precisely synchronized

[2] See our earlier paper for a formal security proof [PCTS00].

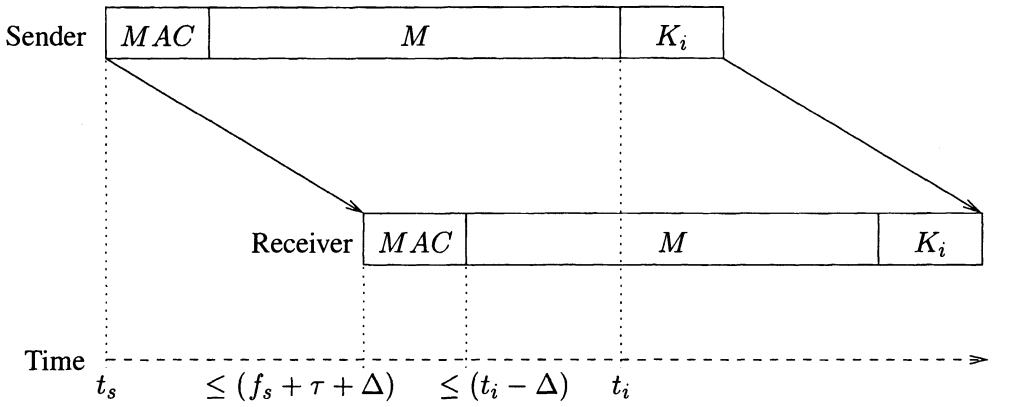


Figure 3.2. Timing of a TIK message.

clocks, we can achieve instant authentication without sender or receiver buffering. We call the new protocol TIK, short for TESLA with Instant Key disclosure.

In TIK, a receiver can instantly authenticate a message immediately after reception. TIK requires accurate time synchronization between nodes and is based on the observation that authentication delay can overlap with the message transmission. Figure 3.2 shows the timing of a packet as it is sent by the sender and received by the receiver. The time t_s here is the time at which the sender begins transmission of the packet, and time t_i is the disclosure time for the corresponding key.

The basic TIK packet contains three parts: a MAC, a message M, and the key K_i used to generate the MAC:

$$S \rightarrow * : \mathsf{MAC}(K_i, M), M, K_i$$

To comply with TESLA's safety condition, the MAC must arrive at each recipient before the recipient's clock reaches time $t_i - \Delta$ (the value Δ is the maximum clock synchronization error between the sender and the receiver), where t_i is the time at which the key is scheduled to be released. Suppose that the transmission is by radio wave and that the sending and receiving delays are negligible. If c is the propagation speed of the radio wave and r is the range of the transmitter, then the maximum transmission delay is $\tau = \frac{r}{c}$. In this case, the key cannot be disclosed until time t_i, and the MAC must arrive at each recipient by time $t_i - \Delta$ according to the recipients clock, so it must be sent by time $t_i - \frac{r}{c} - 2 \cdot \Delta$.

When basic TIK is implemented in a CSMA (Carrier-Sensing Medium Access) control protocol, e.g., IEEE 802.11 [IEE97], it may not be possible to control when a packet is sent relative to TESLA time intervals. In Figure 3.2, the key is released exactly at time t_i, but if a node cannot choose a sending time precisely, the best sending time interval could occur up to one time interval (T_{int}) before the key is released. If we calculate the requirements for this case, we find that the data must begin $T_{int} + \tau + 2 \cdot \Delta$ before the key disclosure, which results in somewhat lower efficiency.

The time interval size and minimum packet size need to be chosen so that a time interval boundary is guaranteed to exist somewhere inside the frame. For example, suppose the physical layer is capable of a peak data rate of 11 Mbps at a range of 300 meters, the minimum frame size is chosen to be the minimum size of an IP header (20 bytes), and the maximum time synchronization error is 1 μs. Then the TESLA time interval must be chosen to be at most $T_{int} \leq 16\,\mu s = 3 \cdot 10^2/(3 \cdot 10^8)s + 20/(11 \cdot 2^{20}/8)s + 1\,\mu s$. The amortized cost of authenticating keys from any node in this case is about 62500 PRF operations per second. As a result, a node capable of performing 1 million PRF operations per second would be able to authenticate broadcasts from only 16 nodes. Though increasing the minimum frame size would reduce the computation overhead, this would also reduce network efficiency and increase jitter. So we can see that if we use the basic TIK mechanism described so far, the protocol will not be practical.

We need to lower the receiver overhead of recomputing the one-way key chain to authenticate keys. We propose two approaches. The first approach uses coarse-grained time intervals as in regular TESLA, which restricts the sender to send only on the boundary of each time interval. Assume a sender uses a $T_{int} = 100ms$, so it has 10 time intervals per second. With TIK, the sender could thus only send 10 packets per second, each packet briefly before the key disclosure.

The second approach uses a Merkle hash tree to authenticate the keys. The rest of this section discusses this approach. Section 2.4.2 describes hash trees in detail. This extension to TIK has a higher communication overhead, a slightly higher setup cost, and a significantly lower verification overhead. The hash tree approach also provides perfect robustness to packet loss, as each packet carries all nodes necessary to verify the leaf key using the authenticated root.

Within the hash tree initialization, the sender generates a sequence of keys. We use a pseudo-random function (PRF) to generate the keys [GGM86]. This

reduces storage overhead for the keys, as the sender only needs to store the seed of the PRF. To establish the keys, the sender randomly chooses a secret master key $\mathcal{X}$ of the PRF $\mathcal{F}$. The sender can then easily compute each key K_i as $K_i = \mathcal{F}_{\mathcal{X}}(i)$. Without the secret master key $\mathcal{X}$, it is computationally infeasible for an attacker to derive a key K_j that the sender has not yet disclosed. This approach allows the sender to efficiently access the keys in any order. The PRF function $\mathcal{F}$ could be a block cipher [GB99] or a MAC, e.g., HMAC [BCK96].

The root value of the hash tree is used to authenticate all leaves. To authenticate a key k_i, the sender discloses i, k_i, and all the nodes necessary to verify the path up to the root, as Section 2.4.2 describes.

In TIK, the depth of the hash tree can be quite large. For example, if the time interval is 16 μs, and nodes are rekeyed once per day, the tree is of depth 34. If each value in the tree is 80 bits, storing the entire tree would require about 160 gigabytes. Computation can be traded for storage, by storing only the upper layers of the tree and re-computing the lower layers on demand. To reconstruct a tree of depth d requires 2^{d-1} applications of the PRF and $2^d - 1$ applications of the hash function, but saves a factor of 2^{d-1} in storage. For example, if a sender with a tree of depth 34 recomputes the lower six layers of the tree, it performs 32 PRF operations and 63 hash operations, but reduces the storage requirement from 160 gigabytes to 5 gigabytes. This technique can be further improved by amortizing this calculation: a node keeps two trees of depth d: one that is fully computed and currently being used, and one that is being filled in. Since a total of $2^{d-1} + 2^d - 1$ operations are required to compute the tree, and the full tree will be used for 2^{d-1} time intervals, the node needs to perform only 3 operations per time interval, independent of the size of the tree.

We can now compute the true calculation and storage cost for the Merkle hash tree that we use in our improved version of TIK. Let D be the depth of the entire tree, and let d be the depth of the part that is recomputed on demand. The initial computation of the tree requires 2^{D-1} evaluations of the PRF, and $2^D - 1$ evaluations of the hash function. This initial computation can be done offline, and is not time critical. To choose d, we consider the value that minimizes total storage. Total storage is given by $2^{D-d+1} - 1 + 2 \cdot (2^d - 1)$, and by differentiating, we can see that storage is minimized when $d = \frac{D}{2}$, and the total storage requirement is $2^{\lceil \frac{D}{2} \rceil + 1} + 2^{\lfloor \frac{D}{2} \rfloor + 1} - 3$. For example, a tree of depth 34 requires almost 26 billion computations to create (which takes about seven hours on a workstation that can compute one million hash functions or PRFs per second), but only 1.25 megabytes to store.

A similar approach can be taken for the generation of future hash trees: once a single hash tree is generated, future hash trees can be generated while the original one is used for a cost of 3 hash functions per time interval plus space of $2^{\lceil \frac{D}{2} \rceil + 1} + 2^{\lfloor \frac{D}{2} \rfloor + 1} - 2$. Only the root of each new tree needs to be sent authenticated, and we can use a similar approach as for switching key chains, which we describe in Section 3.5.3.

3.3.1 TIK Discussion

We assume that hash trees are used to realize TIK. A high-end wireless LAN such as the Proxim Harmony 802.11a [Pro02] has range as far as 250 meters and data rate as high as 100 Mbps, which gives us a maximum propagation delay of $T_{prop} = 0.83\,\mu s$. With time synchronization provided by the Trimble Thunderbolt GPS-Disciplined Clock [Tri02], the synchronization error can be as low as 183 ns with probability $1 - 10^{-10}$; thus we set $\Delta = 0.2\,\mu s$. Assuming authentic keys are re-established every day, with a 20 byte minimum data size and a 80-bit MAC, we compute the time interval duration and the tree depth. The time interval duration depends on the maximum time synchronization error, the maximum propagation delay, and the sending duration of the minimum packet: we set $T_{int} = \Delta + T_{prop} + T_{send}$. The minimum packet size is determined by the minimum data size plus the $D - 1$ hash tree nodes disclosed in each packet. Assuming 80 bit nodes, 20 bytes minimum data size, and 100 Mbit/s transmission speed, we compute $T_{send} = (10 \cdot (D - 1) + 20)/(100 \cdot 2^{20}/8)$. The depth of the hash tree that is valid for one day (86400 seconds) depends recursively on the interval duration: $D = \lceil \log_2(86400/T_{int}) \rceil$. Solving for D yields $D = 33$, and the time interval T_{int} is 20.1 μs. Assuming that the node generates new trees for redistribution while it is using its old trees, it requires 8 megabytes of storage and needs to perform 230000 operations per second to maintain and generate trees. To authenticate a received packet, a node computes 33 hash functions.

Current commodity wireless LAN products such as commonly used 802.11b cards provide 11 Mbps at 250 meters. Given the same time synchronization, rekeying interval, minimum packet size, and MAC length, the tree has depth 30 and the time interval is about 227 μs. Assuming that the node generates new trees for redistribution while it is using its old trees, it requires just 2.6 megabytes of storage and needs to perform 26500 hash function computations per second. To authenticate a received packet, a node computes 30 hash functions. A modern workstation computes 1 million MD5 hash functions per second (see Section 2.3); it would use about 2.6% of their CPU to keep its key

trees up-to-date and use 30 μs of additional CPU time for each packet it receives.

3.3.2 TIK Summary and Security Considerations

Protocol 3.2 summarizes the basic TIK protocol. The security of TIK relies on the following assumptions:

- The receiver's clock is time synchronized up to a maximum error of Δ. In case of clock drift, it is the receiver's responsibility to periodically resynchronize clocks.
- The master key $\mathcal{X}$ is random and thus not predictable by an adversary.
- The PRF $\mathcal{F}$ used to generate the keys $K_0, \ldots, K_N$ is secure, so no adversary can find $\mathcal{X}$ even if it knows the sequence of keys $K_i, \ldots, K_j$. Furthermore, the adversary cannot predict the next key in the sequence from the keys known to date.
- The hash tree satisfies the security properties outlined in Section 2.4.2.

3.4 Time Synchronization

TESLA requires loose time synchronization. TESLA does not need the strong time synchronization properties that sophisticated time synchronization protocols provide [Mil92, SLWL90, Mil94, LMS85]. TESLA can avoid the complexity and high overhead associated with these protocols.

We outline a simple and secure time synchronization protocol that suits TESLA. TESLA requires that the receiver knows an upper bound of the difference between the sender's local time and the receiver's local time, Δ. For simplicity, we assume the clock drift of both sender and receiver are negligible (otherwise the sender and receiver can periodically resynchronize the time). We denote the real difference between the sender and the receiver's time with $\pm\delta$. In loose time synchronization, the receiver does not need to know δ but only some $\Delta \geq \delta$. We now describe three different time synchronization settings: direct, indirect, and delayed time synchronization.

3.4.1 Direct Time Synchronization

In direct time synchronization, the receiver performs an explicit time synchronization with the sender. This approach does not require any extra infrastructure to perform time synchronization. We present a simple two-round time

1 **Sender setup**. The sender splits up the time into uniform time intervals of duration T_{int}. Time interval 0 starts at time T_0, time interval 1 at time $T_0 + T_{int}$, etc. The sender assigns a key to each time interval using a PRF $\mathcal{F}$: the sender randomly selects a master key $\mathcal{X}$, and derives the key for time interval i: $K_i = \mathcal{F}_{\mathcal{X}}(i)$. The sender constructs a hash tree over all the keys (as Section 2.4.2 describes) and derives the root m_r. The sender discloses the key K_i right after the end of the time interval i.

2 **Receiver setup**. The receiver engages in a protocol to receive the authentic parameters of the sender's key chain: T_0, T_{int}, and the authentic root of the hash tree m_r. Furthermore, the receiver synchronizes its clock with the sender and knows the maximum clock synchronization error Δ.

3 **Sending authenticated packets**. To send a message M in time interval j, the sender sends the following message (where m'_j is the neighbor node of key K_j, and $m'_j, \ldots, m''_j$ denote the nodes necessary to verify the root node m_r):

$$S \rightarrow * : \mathsf{MAC}(K_j, M), M, m'_j, \ldots, m''_j, K_j$$

The sender times sending the packet such that it starts sending key K_j right at the end of time interval j. Thus, if the size of the data $\mathsf{MAC}(K_j, M), M, m'_j, \ldots, m''_j$ is s bits, and the bandwidth is b bits per second, the sender starts sending the packet at time $T_0 + (j+1) \cdot T_{int} - \frac{s}{b}$.

When the receiver completely received the MAC $\mathsf{MAC}(K_j, M)$, it stores its local time t_r. After receiving the entire packet, it infers the sending time interval from the disclosed key K_j, by authenticating K_j using the root m_r and the nodes $m'_j, \ldots, m''_j$. The receiver now knows that the packet was sent in time interval j, and it can check that the key K_j was still secret when it received the MAC: $t_r + \Delta < T_0 + (j+1) \cdot T_{int}$. If the condition holds, the packet was safe and the receiver can verify the MAC, otherwise, the receiver has to drop the packet.

Protocol 3.2: Summary of the TIK protocol.

synchronization protocol that satisfies the requirement for TESLA, which is that the receiver knows an upper bound on the sender's clock. Reiter previously described this protocol [Rei93, RBvR94].

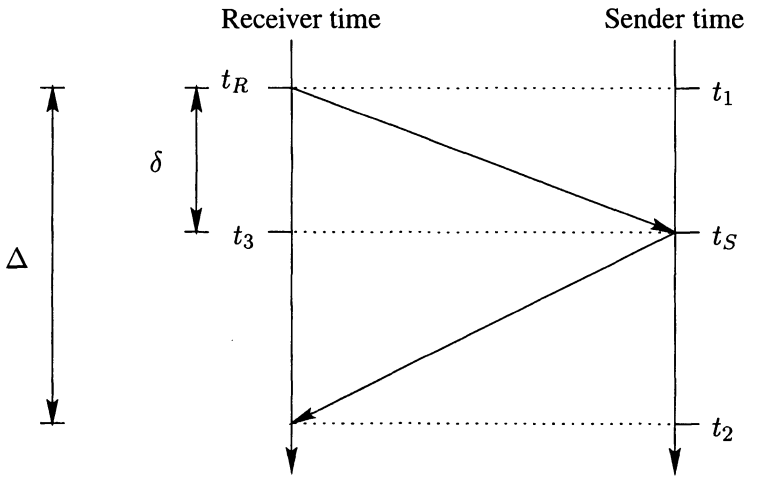


Figure 3.3. Direct time synchronization between the sender and the receiver. The receiver issues a time synchronization request at time t_R, at which time the sender's clock is at time t_1. The sender responds to the request at its local time t_S. In TESLA, the receiver is only interested in an upper bound on the sender's time. When the receiver has its current time t_r, it computes the upper bound on the current sender's time as $t_s \leq t_r - t_R + t_S$. The real synchronization error after this protocol is δ. The receiver, however, does not know the propagation delay of the time synchronization request packet, so it must assume that the time synchronization error is Δ (or the full round-trip time written as RTT).

Figure 3.3 shows a sample time synchronization between the receiver and the sender. In the protocol, the receiver first records its local time t_R and sends a time synchronization request containing a nonce to the sender.[3] Upon receiving the time synchronization request, the sender records its local time t_S and replies with a signed response packet containing t_S and the nonce. The box labeled Protocol 3.3 lists the time synchronization protocol.

Upon receiving the signed response, the receiver checks the validity of the signature and verifies that the nonce in the response packet equals the nonce in the request packet. If all verifications are successful, the receiver uses t_R and t_S to compute the upper bound of the sender's time: when the receiver has the current time t_r, it computes the upper bound on the current sender's time as $t_s \leq t_r - t_R + t_S$. The real synchronization error after this protocol is δ, as Figure 3.3 shows. The receiver, however, does not know the propagation

[3]Note that the security of this time synchronization protocol relies on the unpredictability of the nonce — if an attacker could predict the receiver's nonce, it could send a time synchronization request to the sender with that nonce, and replay the response later to the receiver.

1 **Setup**. The sender S has a digital signature key pair, with the private key K_S^{-1} and the public key K_S. The receiver R knows the the authentic key K_S. The receiver chooses a random and unpredictable nonce.

2 **Protocol steps**. Before sending the first message, the receiver records its local time t_R.

$$R \rightarrow S : \mathsf{Nonce}$$
$$S \rightarrow R : \{\mathsf{Sender\ time}\ t_S, \mathsf{Nonce}\}_{K_S^{-1}}$$

To verify the return message, the receiver verifies the digital signature and checks that the nonce in the packet equals the nonce it randomly generated. If the message is authentic, the receiver stores t_R and t_S. To compute the upper bound on the sender's clock at local time t, the receiver computes $t - t_R + t_S$.

Protocol 3.3: Simple time synchronization protocol.

delay of the time synchronization request packet, so it must assume that the time synchronization error is Δ (or the full round-trip time (RTT)).

The processing and propagation delay of the response message does not change δ, since the receiver is only interested in an upper bound on the sender's clock. (If the receiver were interested in the lower bound on the sender's clock, the return delay would matter.) The processing delay at the sender does not matter, as long as the sender immediately records the arrival time of the synchronization request packet. Thus, delay in computing the digital signature does not affect the real synchronization error δ.

A digital signature operation is computationally expensive, and we need to be careful about the case when the sender is overwhelmed with time synchronization requests from receivers. We address this issue in Section 3.6.1.

3.4.2 Indirect Time Synchronization

In indirect time synchronization, both the sender and receiver independently synchronize their time with a time reference. Let Δ_{SC} denote the maximum synchronization error between the sender and the time reference. The receiver synchronizes its clock with the time reference (by using the same protocol as in direct time synchronization), and obtains the values t_R, t_C, and Δ_{RC} as before. When the receiver has the current time t_r, it computes the upper bound on the current sender's time as $t_s \leq t_r - t_R + t_C + \Delta_{SC}$. This determines

an upper bound on the time reference's clock, and then we simply add the maximum synchronization error of the sender to get an upper bound on the sender's clock.

In indirect time synchronization, the receiver does not need to send any information to the sender. The sender can periodically broadcast digitally signed packets containing its time synchronization error Δ_{SC} with the time reference, the time interval, and the key chain information outlined in Section 3.2.3. A new receiver can start authenticating the data stream immediately after it receives one of the signed advertisements. This is particularly useful in the case of satellite broadcast.

3.4.3 Delayed Time Synchronization

Sometimes, a receiver accumulates and stores data without processing it. At a later time, the receiver wishes to authenticate and process the data. The sender will have already disclosed all the keys in the meantime, so the stored packets are not safe. But if the sender's and receiver's clock drifts are negligible, the receiver can still authenticate the packets by simply storing the arrival time of each packet. This is useful for many applications. For example, a number of researchers have proposed collecting IP traceback information to identify the origins of denial-of-service attacks [Bel00, SWKA00, SP01]. A receiver can use TESLA to authenticate IP traceback information. Ideally, receivers collect IP traceback data, but processes it only if under attack. TESLA with delayed time synchronization is particularly useful to authenticate itrace messages [Bel00], or to authenticate the tags in packet marking techniques [SWKA00, SP01].

3.4.4 Determining the Key Disclosure Delay

An important TESLA parameter is the key disclosure delay d. Although the choice of the disclosure delay does not affect the security of the system, it is an important performance factor. A short disclosure delay will cause packets to lose their safety property, so receivers will discard them; but a long disclosure delay leads to a long authentication delay for receivers. We recommend choosing the disclosure delay as follows: in direct time synchronization let the RTT be a reasonable upper bound on the round trip time between the sender

and the receiver; then choose $d = \lceil RTT/T_{int} \rceil + 1$. Note that $d \geq 2$, and that $d = 1$ does not work.[4]

In indirect time synchronization, we choose $d = \lceil (D_{SR} + \Delta_{SC} + 2 \cdot \Delta_{RC})/T_{int} \rceil + 1$, where D_{SR} is a reasonable upper bound on the network delay of a packet traveling from the sender to the receiver.

3.5 Variations

In this section, we describe some alternative versions of TESLA.

3.5.1 Instant Authentication

In the basic TESLA protocol, the receiver must buffer packets for the duration of one disclosure delay before it can authenticate them. This may cause problems for receivers with limited buffer space: if the sender sends a burst of packets, the receiver may not have enough space to buffer them until they can be authenticated. Moreover, Section 3.6.2 discusses how receiver buffering introduces vulnerability to denial-of-service attacks. Thus, we propose an extension to TESLA that provides *instant authentication*: the receiver can authenticate packets as soon as they arrive. We achieve this property by replacing receiver buffering with sender buffering. If the sender buffers packets during one disclosure delay, it could put the hash value of the message of a later packet into an earlier packet. So as soon as the receiver authenticates the earlier packet (that includes the hash of the message of the later packet), the receiver also knows that the message in the later packet is authentic.

For instant authentication, the sender buffers packets for the duration of the disclosure delay. For simplicity, assume the sender sends out a constant number v of packets in each time interval. To construct the packet for the message M_j in time interval i, the sender appends the hash value of the message M_{j+vd} (one disclosure delay away) to M_j and then computes the MAC value also over $H(M_{j+vd})$ (using the key K_i). Figure 3.4 illustrates how the sender constructs packet P_j and appends $H(M_{j+vd})$, $\text{MAC}(K_i, M_j \mid\mid H(M_{j+vd}))$, and the key disclosure K_{i-d}.

When the receiver receives packet P_{j+vd}, the receiver also gets key K_i. If K_i is authentic, the receiver can authenticate packet P_j. P_j carries the hash of

[4] A disclosure delay of one time interval ($d = 1$) does not work. Consider packets sent close to the boundary of the time interval: after the network propagation delay and the receiver time synchronization error, a receiver will need to discard the packet, because the sender will already be in the next time interval, when it discloses the corresponding key.

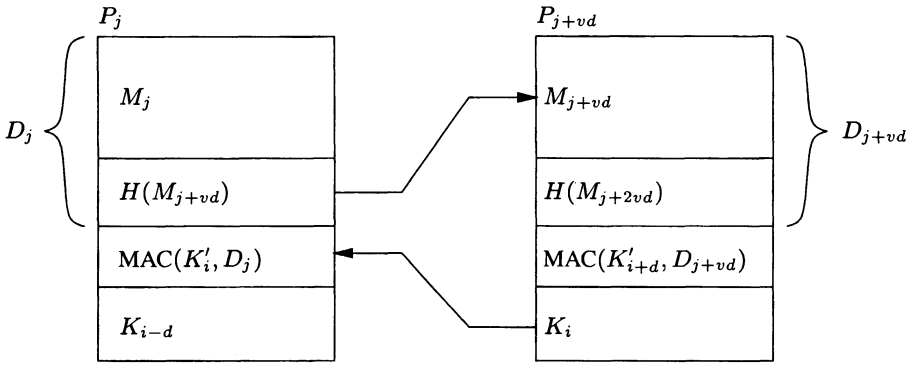


Figure 3.4. Instant authentication example. The figure shows two packets P_j and P_{j+vd}. Packet P_j contains the data $D_j = M_j \mid\mid H(M_{j+vd})$, authenticated with the MAC computed with key K_i. Packet P_{j+vd} discloses the key K_i, so the receiver can authenticate data D_j in packet P_j. If D_j is authentic, the receiver can use the hash of message $H(M_{j+vd})$ to authenticate message M_{j+vd}.

the message M_{j+vd}, so if P_j is authentic, the hash $H(M_{j+vd})$ is also authentic. The receiver buffers the hash $H(M_{j+vd})$, and can instantly authenticate message M_{j+vd} when it arrives. If P_j is lost or not safe, the receiver cannot authenticate message M_{j+vd} instantly, but it can fall back to the basic TESLA protocol and authenticate M_{j+vd} after it learns the key K_{i+d}.

If each packet only carries the hash of one other packet, it is clear that the sending rate needs to remain constant. Also it is clear that if a packet is lost, the corresponding packet cannot be instantly authenticated. To achieve flexibility for dynamic sending rate and robustness to packet loss, the sender can add the hash values of multiple future packets to a packet. This is similar to the EMSS protocol, which we present in Chapter 5.

3.5.2 Concurrent TESLA Instances

In this section, we extend and optimize the basic TESLA protocol for multiple concurrent TESLA instances to accommodate settings with heterogenous receivers.

Choosing the disclosure delay involves a tradeoff. Short key disclosure delays yield short authentication delays, but also mean more unsafe packets for receivers with a long network delay. A long key disclosure delay increases the probability that packets arrive safely, but increases the authentication delay for receivers with a low network delay.

Our solution is to simultaneously use multiple instances of TESLA with different disclosure delays. Each receiver can verify the safety of each TESLA instance when the packet arrives, and choose the instance for that packet that yields the shortest authentication delay.

A simple approach for concurrent TESLA instances is to treat each TESLA instance independently, with one key chain per instance. However, multiple independent one-way key chains can use up more space in packets. We now present an optimization that reduces the space overhead for concurrent TESLA instances.

Instead of using one independent key chain per TESLA instance, we use one key chain but each instance uses a different key schedule. The basic scheme works as follows. All TESLA instances share the same time interval duration and the same key chain. Each key K_i in the key chain is associated with the corresponding time interval i, and the sender discloses K_i in time interval i.[5] Assume that the sender uses w instances of TESLA, denoted as $\tau_1 \ldots \tau_w$. Each TESLA instance τ_u has a different disclosure delay d_u.

Note that only a single key is disclosed, so no additional space is required over a single instance of TESLA. Of course, since we have multiple MACs with multiple instances of TESLA, the MACs will take more space than a single instance of TESLA. From a single key disclosure K_i, we can get multiple MAC verifications. We simply modify the key derivation formula in Section 3.2.2. With $K_i^{(u)}$, we denote the MAC key of TESLA instance u in time interval i. We derive $K_i^{(u)} = f'_{K_i}(u)$, where f' is the same PRF we use to derive the MAC key in Section 3.2.2. Note that the keys of the first instance are the same as in the basic TESLA protocol with one instance.

Figure 3.5 shows an example with two TESLA instances: one instance has a key disclosure delay of two time intervals, and the other instance has four time intervals. The lowest line of keys shows the key disclosure schedule, i.e., which key is disclosed in which time interval. The middle and top line of keys shows the key schedule of the first and second instance respectively, i.e., which key is used to compute the MAC for the packets in the given time interval for the given instance.

[5]Note that this key disclosure schedule is different from the disclosure schedule in Section 3.2.2, where key K_i was used to compute the MAC in interval i and was disclosed in interval $i + d$.

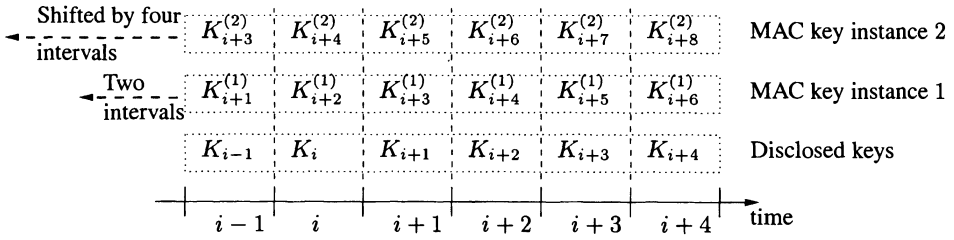


Figure 3.5. This figure shows the key chain when the sender uses multiple TESLA instances. The bottom row shows the key that the sender discloses in each time interval. The top two rows show the keys that the sender uses to compute the MAC of the packet in a certain time interval. In this figure, the sender concurrently uses two TESLA instances: instance 1 has a disclosure delay of $d_1 = 2$ time intervals, and instance 2 has a disclosure delay of $d_2 = 4$ time intervals.

3.5.3 Switching Key Chains

In long-lived broadcasts, the key chain may be used up and the sender must switch to a new key chain. A simple approach is for the sender to stop using the old chain, send a new authenticated packet that commits to the new chain and to start using the new chain, but if a receiver does not receive the authenticated packet, the receiver could not authenticate subsequent packets. Another approach is to use two key chains that overlap during a short transition period, but the authentication field size would be larger during the transition period.

A better approach is to commit to the new key chain with the current key chain. So we add the commitment to the new key chain to a packet that is authenticated with the current chain; for robustness to packet loss, we add the new key chain commitment to multiple packets. If we add the commitment to the packet instead of disclosing a key, the packet size does not increase. Some packets will contain the commitment to the new key chain (authenticated with the MAC), and the other packets disclose a key.

The sender needs to send the new key chain commitment value early, so receivers receive and authenticate the commitment to the new key chain before the previous chain expires. This ensures that the receivers authenticate the new key chain commitment before the previous chain is used up.

Figure 3.6 shows an example. Assume the disclosure delay is two time intervals ($d = 2$). The old key chain stops at Key K_n and the new chain starts with key K_{n+1}. To commit to the new key chain, the sender includes the commitment $F(K_{n+1})$ to the new one-way key chain to packets authenticated under the old chain. To guard against packet loss, the sender repeats the commitment for i time intervals. In Figure 3.6, the sender repeats the commitment $F(K_{n+1})$

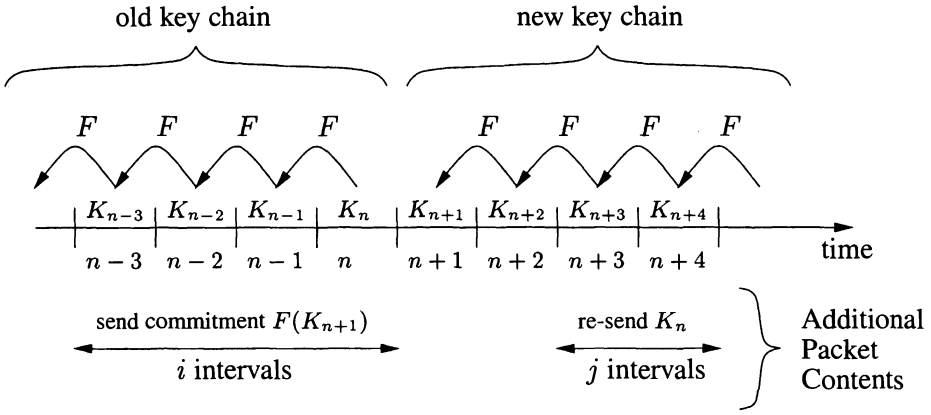


Figure 3.6. This figure shows how to switch key chains reliably. The old key chain ends at key K_n, and the new key chain starts with key K_{n+1}. The disclosure delay in this figure is $d = 2$ time intervals. $F(K_{n+1})$ is a commitment to the new key chain, so the sender adds $F(K_{n+1})$ to packets for i time intervals of the old key chain. The receiver needs to authenticate the new commitment $F(K_{n+1})$ prior to time interval $n + 3$, because that is when the receiver first uses $F(K_{n+1})$ to authenticate K_{n+1}. To prevent the case where the receiver loses many packets and can not authenticate the packets of the old chain and can hence not authenticate the new commitment $F(K_{n+1})$, the sender continues to re-send K_n. Note that this is necessary, since the receiver cannot recover K_n from the new key chain as $F(K_{n+1}) \neq K_n$.

during the four time intervals $n - 3, \ldots, n$, and embeds it in outgoing packets. The sender discloses key K_n in interval $n + 2$, so when the receiver obtains K_n it can authenticate the commitment $F(K_{n+1})$. However, if the receiver misses all packets in interval $n + 2$, it may be unable to recover K_n. To prevent this case, the sender continues to send K_n for j more time intervals. In Figure 3.6, the sender continues to send K_n for 2 more time intervals.

3.5.4 Further Extensions

We present some miscellanious TESLA extensions here. The first extension explores TESLA in environments without time synchronization. The second extension describes how TESLA can use time-release cryptography to achieve implicit key disclosure, which removes the need to disclose keys.

We can use TESLA without explicit time synchronization to provide weak authentication. In environments that do not allow an adversary to slow down (or capture and re-send) broadcast messages, time synchronization may not be necessary. Consider, for example, satellite broadcast. To verify packet safety, the sender and receiver still need to be time synchronized. So the sender

periodically sends out a digitally signed beacon message, which contains the sender's current time, a key to commit to the one-way key chain, and the interval timing parameters that communicate the key disclosure schedule. If we assume that this message eventually arrives, and if receivers use the policy that they only advance their clocks and ignore packets with older time stamps, all the receivers are eventually synchronized. The maximum synchronization error here is the duration it takes to sign the beacon message, and the maximum propagation delay. For normal message authentication, the sender and receiver engage in the regular TESLA protocol.

A drawback of TESLA is that receivers cannot authenticate the last message, because they do not know the verification key. A simple solution is to continue sending empty packets, until all keys are disclosed and the receivers reach a high probability for receiving the final key. An academically interesting solution is to use timed-release cryptography [RSW96]. The time-controlled disclosure (through a time-lock puzzle) of the keys in timed-release cryptography matches the delayed key disclosure in TESLA. The sender adds the MAC key to each packet, encrypted under a timed-release key. After the receiver decodes the time-lock puzzle, it can decrypt the MAC key, verify if the MAC key is authentic (using the one-way key chain as in standard TESLA), and finally verify that the message data is authentic.

3.6 Denial-of-Service Protection

In many broadcast environments, denial-of-service (DoS) attacks are a considerable threat, because an adversary can broadcast bogus messages to the group (e.g., in IP multicast).[6] A variety of DoS attacks are possible, for example congesting the network, sending a malicious packet that exploits a vulnerability in the Operating System or application, or flooding the receivers with bogus traffic.

In particular, we discuss packet flooding DoS attacks and show how to strengthen TESLA to defend against them. The sender is safe against DoS attacks if the receivers perform indirect time synchronization (as we explain below). For direct time synchronization, we show how to defend against DoS attacks on the sender. We also consider DoS attacks on the receiver. We show that TESLA does not create additional vulnerability to DoS attacks if the re-

[6]Source-Specific Multicast (SSM) is a new multicast protocol, and an IETF working group was formed in August 2000 [Mul02]. SSM addresses this problem by enforcing that only one sender can send to the multicast group.

ceiver has sufficient buffer space, and we describe techniques that reduce the receiver's exposure to DoS attacks if buffer space is limited.

3.6.1 DoS Attack on the Sender

A DoS attack on the sender is not possible if TESLA is used with indirect time synchronization, because the sender only sends packets to receivers; the sender simply drops all packets it receives. In direct time synchronization, a DoS attack is possible, since the sender digitally signs the response packet for each time synchronization request. An attacker may perform a DoS attack by flooding the sender with time synchronization requests.

The signer includes the receiver's nonce in the response packet, and digitally signs the response packet (with any digital signature scheme, such as RSA [RSA78], or DSA [Nat91]) so the receiver can authenticate the response packet. Since public-key signature algorithms are computationally expensive (see Section 2.3), signing the response packet can become a performance bottleneck for the sender. A simple technique can alleviate this situation. The sender can aggregate multiple requests, compute and sign a Merkle hash tree generated over all nonces and arrival time stamps (see Section 2.4.2). At each leaf of the hash tree, the sender places $H(N_i \mid\mid t_i)$, where N_i is the receiver's nonce and t_i is the time the request packet arrived at the sender. Adding the arrival timestamp to the hash tree gives each receiver the same time synchronization accuracy as the single-client time synchronization protocol that Section 3.4.1 describes.

Instead of individually signing each nonce, the sender only signs the root of the hash tree. With each response packet, the sender includes all the nodes of the hash tree that a receiver needs to reconstruct the root of the tree.

To verify the digital signature of the response packet, each receiver needs to reconstruct the root node of the hash tree, by using additional nodes the sender includes in the response packet. Note the number of nodes returned in the response packet is logarithmic in the number of nonces in the tree.

Assuming a 50 ms interval time (so the sender computes at most 20 signatures per second) and assuming 100000 receivers want to synchronize time in one interval, the return packet would contain only 17 hash nodes or 170 bytes (if we use 80 bit long hashes). Any cryptographically secure hash function can be used for H (see Section 2.2.2).

3.6.2 DoS Attack against the Receiver

This section discusses two DoS attacks on the client: the packet buffer DoS attack, and the key chain DoS attack. We assume that the attacker has full control over the network, so the attacker can always delay or drop packets.

Delaying packets may also result in a dropped packet. When the receiver receives a delayed packet, it is likely that the packet is not safe, so the receiver drops the packet. Speeding up packets can only benefit the receiver. Replay can easily be addressed: first, the receiver only accepts a duplicated packet during a short time period (a late packet would not be safe); and second, we can prevent a replay attack by adding a sequence number to each packet, and by protecting the sequence number with the MAC. The receiver can easily filter out duplicates when it authenticates packets within a time interval.

The remainder of the section discusses more complicated DoS attacks and shows how to mitigate or prevent these attacks. First we discuss a flooding attack which fills up the receiver buffers. Second we discuss an attack that tries to waste the receiver's computation resources by unnecessarily re-computing the key chain.

3.6.2.1 Packet Buffer DoS Attack

In a flooding attack, the attacker floods the multicast group with bogus traffic. This attack is serious because current multicast protocols do not enforce sending access control. However, the instant authentication protocol (see Section 3.5.1) provides partial protection against a flooding attack.

If the receiver has a large buffer, flooding cannot do much harm. TESLA only requires that the receiver buffers packets during one disclosure delay, so the buffer size only needs to be as large as the product of the network bandwidth and the disclosure delay time. Assuming that the receiver has a 10 Mbps network connection and a 500ms disclosure delay, the required buffer size needs to be at most 640 KBytes. If network packets are 1024 bytes large, the computation overhead to authenticate the packets is about 1280 PRF operations per second. OpenSSL HMAC-MD5 [BCK96, BCK97, Ope01] processes about 90000 1024-byte blocks per second on a 800 MHz Pentium III Linux workstation, so the estimated processor overhead for TESLA authentication is at most 1.5% of the total CPU time.

If the receiver's buffer size is not as large as computed above, flooding could result in a DoS attack because the receiver would drop packets due to a lack of buffer space. We propose to use TESLA with instant authentication in this case (see Section 3.5.1). In instant authentication, the receiver can immedi-

ately authenticate incoming packets, and simply drop packets that it cannot authenticate since they are likely to be spoofed anyway.

If the receiver has too little memory to buffer all incoming traffic during the disclosure delay, it needs to decide on a drop or replacement policy when the buffer fills up. Dropping all packets of a particular interval once the buffer is full is a poor policy, because an attacker might insert the spoofed traffic early in each time interval, causing the receivers to buffer mostly spoofed packets. Ideally, the receiver uses a random replacement policy once the buffer overflows. Using probabilistic buffer replacement, any incoming packet in one time interval has equal probability of being in the buffer, preventing an adversary from displacing legitimate packets [Kuh00].

3.6.2.2 DoS on the Key Chain

Another DoS attack on the receiver is specific to how the TESLA receiver verifies the authenticity of a key of the key chain. If an attacker could fool a receiver into believing that a packet was sent out far in the future, then the receiver would try to verify the key disclosed in the packet by applying the pseudo-random function until the last committed key chain value. By verifying feasible time intervals the sender can be in, the receiver can prevent this attack. For an incoming packet sent in interval j, the receiver can verify interval j is not in the far future. If the time at the receiver is t, and the packet has an interval j corresponding to a time greater than $t+\Delta$, it must be bogus (where Δ is the maximum time synchronization error between the sender and receiver).

Chapter 4

BIBA BROADCAST AUTHENTICATION

To design a broadcast authentication protocol that provides real-time authentication, robustness to packet loss, efficient verification, and scales to large audiences, we invent a new signature algorithm. We call our new algorithm the BiBa signature, short for Bins and Balls signature. With a new construction, we use the BiBa signature to design the BiBa broadcast authentication protocol with the following properties:

- Fast generation of authentication information (in particular, if the sender uses parallel computers, generating the BiBa authentication information only requires two sequential hash function computations).
- Fast verification by the receiver.
- Instant authentication. Neither the sender nor the receiver need to buffer any packets, so BiBa is ideal for authenticating real-time data.
- Robustness to packet loss.
- Perfect scalability. The authentication information is independent of the number of receivers.
- Reasonable communication overhead, on the order of 100 bytes per packet.

BiBa thus contrasts with previously proposed broadcast authentication protocols that cannot simultaneously support real-time authentication, robustness to packet loss, robustness to denial-of-service attack, efficient verification, and scalability for large audiences [CGI+99, GR97, PCST01, PCTS00, PCTS02, Roh99, WL98]. The TESLA protocol that we describe in Chapter 3 came

A. Perrig et al., *Secure Broadcast Communication*

close, but it has delayed authentication. As we describe in Section 1.2, a digital signature on each packet satisfies real-time authentication, robustness to packet loss, and scalability for large audiences, but current signature algorithms have a high verification overhead, as Section 1.2.1 discusses.

The drawback of the BiBa broadcast authentication protocol is that it requires loose time synchronization between the sender and receivers.

In the remainder of this chapter, we will first present our new BiBa signature algorithm. Next we use BiBa to design an efficient broadcast authentication protocol and study the performance in real-world settings. Finally, we analyze the security of BiBa.

4.1 The BiBa Signature Algorithm

For the past 25 years researchers have created and refined digital signature algorithms using one-way functions without trapdoors [BM94, BM96b, EGM90, Lam75, Lam79, Mer88, Mer90, Rab78, Roh99]. These signature algorithms are efficient for signature generation and verification, but the signatures are too large for many applications. We propose a new construction for a signature algorithm based on one-way functions without trapdoors. We call our signature algorithm BiBa, short for Bins and Balls signature. BiBa is a new combinatorial signature, and provides the asymmetric property between the signer and an adversary required for secure broadcast authentication. The size of a BiBa signature is less than half the size of previous schemes based on one-way functions; and the verification is at least twice as fast. However, our public keys are larger than most previous systems, and the time to generate signatures is also higher. We review related work in more detail in Section 8.2.

In this section we review self-authenticating values (which we introduce in Section 2.4.3), give a simple example to motivate the main idea of BiBa, and present the BiBa signature algorithm in detail.

We use the following notation in the remainder of this chapter. F and F' represent two pseudo-random functions (see Section 2.2.5 for a summary of PRFs):

$F : \{0,1\}^{m_2} \times \{0,1\}^{m_1} \rightarrow \{0,1\}^{m_2}$

$F' : \{0,1\}^{m_1} \times \{0,1\}^{m_1} \rightarrow \{0,1\}^{m_1}$.

H is a hash function in the random oracle model[BR97]. G represents a hash function family in the random oracle model and $G_h : \{0,1\}^{m_2} \rightarrow [0, n-1]$ is an instance in the hash function family G selected with an indicator h. Many different methods exist to pick G, we propose to use the following approach. We use $G_h(x) = \mathsf{MAC}(h, x) \mod n$. If the MAC function out-

puts τ bits, and n does not divide 2^τ, some bins will occur more frequently than others. However, since the range of the MAC function is much larger than n, this bias is insignificant. To remove this bias, we could set $G_h(x) = \mathsf{MAC}(h, x \mid\mid c) \mod n$, for $c = 0$ initially, and we keep incrementing c until $\mathsf{MAC}(h, x \mid\mid c) < 2^\tau - (2^\tau \mod n)$. A good choice for the MAC function is HMAC-MD5 [BCK96, BCK97].

4.1.1 The Self-Authenticating Values

We review self-authenticating values from Section 2.4.3. The signer precomputes values that it subsequently uses to generate BiBa signatures. These values are random numbers generated in a way that the receivers can instantly authenticate them with the *public key* (which is sometimes referred to as *public validation information* in this context). We call these precomputed values *self-authenticating values*. Self-authenticating values have the following properties: the verifier can efficiently authenticate the self-authenticating value based on the public validation information; and it is computationally infeasible for an adversary to find the self-authenticating value given the public validation information. The simplest approach is to use the PRF F as a commitment scheme. Given a self-authenticating value s, the public validation information is $f_s = F_s(0)$. If the verifier learns f_s in an authentic fashion, it can easily authenticate s by verifying $F_s(0) = f_s$. In BiBa, the signer needs multiple self-authenticating values, so a public key could consist of the concatenation of multiple validation informations.

A Merkle hash tree is an alternative for constructing self-authenticating values (see Section 2.4.2). The self-authenticating values would be the leaf nodes of the tree and the public validation information is the root node of the tree.

We describe efficient methods to generate and check self-authenticating values in the context of broadcast authentication in Section 4.2. For now we consider the case where the signer has t precomputed values $s_1, \ldots, s_t$, and receivers know the authentic validation information $F_{s_i}(0)$ of each value.

4.1.2 Intuition for the BiBa Signature

Every signature algorithm relies on an asymmetric property, such that it is easy for the signer to sign a message, easy for a verifier to verify the signature of a message, but hard for an adversary to forge a signature of a new message.

BiBa stands for *Bins and Balls* signature. In BiBa, the signer randomly throws many balls into a set of bins. Since the signer throws many balls, it has a high probability of observing a collision of balls in a bin. The balls that

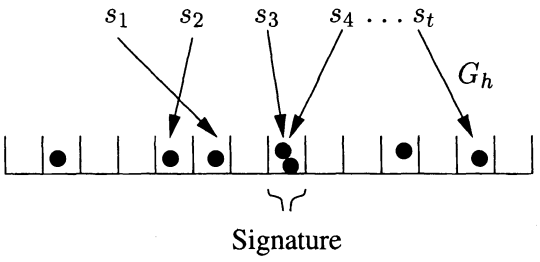


Figure 4.1. Simplified BiBa signature. The function G_h maps the balls $s_1 \ldots s_t$ into the bins. A two-way collision of two balls in a bin form the signature.

participate in the collision are the signature. An adversary that wants to forge a signature only learns the balls that the signer discloses in signatures, a small number compared to the number of balls of the signer. Since the adversary can only throw relatively few balls into the bins, it has a very low probability of observing a collision of balls in a bin (and hence the adversary has a very low probability to forge a signature). We now illustrate the BiBa signature with a simplified example.

4.1.3 Signature Generation

To sign a message m, the signer first computes a hash $h = H(m)$. The signer uses the hash h of the message to pick the hash function G_h from a family of random hash functions, as we describe at the beginning of this section. In the analogy of the bins and balls, the function G_h maps the balls into bins. Each ball is a self-authenticating value, and the bins correspond to the range of the random function G_h. To randomly throw the balls into the bins, the signer maps each ball s_i to a bin $G_h(s_i)$. The signer looks for a two-way collision of two balls: $G_h(s_i) = G_h(s_j)$, with $s_i \neq s_j$. The pair $\langle s_i, s_j \rangle$ forms the signature. Figure 4.1 shows an example.

We exploit the asymmetric property that the signer has more balls than the adversary, and hence the signer can easily generate the BiBa signature with high probability. On the other hand, an adversary only knows the few disclosed balls and hence has a low probability to find a valid BiBa signature.

4.1.4 Signature Verification

The verifier receives message M and the BiBa signature $\langle s_i, s_j \rangle$. We assume for now that the verifier has an efficient method to authenticate the balls s_i and

s_j. To verify the BiBa signature the verifier computes $h = H(m)$, checks that $s_i \neq s_j$, and $G_h(s_i) = G_h(s_j)$.

Note that the verification is very light-weight: Without considering the ball authentication, the verification only requires one hash function computation to hash the message, and two hash function computations to map the balls to the bins.

4.1.5 Security of BiBa

Assume the signer has t balls and the range of the hash function G_h is $[0, n-1]$. Given a message m, the probability of finding a BiBa signature is equal to the probability of finding at least one two-way collision, i.e., at least two balls end up in the same bin, when throwing t balls uniformly randomly into n bins. The probability of at least one collision P_C is:

$$P_C = 1 - \prod_{i=1}^{t-1} \frac{n-i}{n} \approx 1 - e^{\frac{t(t+1)}{2n}}$$

Figure 4.2 shows the probability of at least one collision when throwing t balls into 10000 bins.

The security of the BiBa signature comes from the fact that the adversary has few balls and hence has a small probability to find a collision. For example, if the signer has 200 balls, as marked with the letter A in Figure 4.2, it has a 87% probability of finding a signature after one try (one two-way collision after throwing 200 balls into 10000 bins) for a given a message. If we assume that an adversary only knows 10 balls (which it learned from 5 BiBa signatures), it has a $0.4\% = 2^{-8.0}$ probability to find a collision (forge a signature) after one try. Figure 4.2 marks the adversary's probability with the letter B.

4.1.6 BiBa Extensions

One attempt to increase the security is to increase the number of balls and bins, but that approach would increase the size of the public key.

A better approach for increasing the security is to use multiple two-way collisions to generate a signature. For example, a signature on message m, with $h = H(m)$, would consist of z two-way collisions. The signature is composed of z pairs of balls $\langle S_{a_1}, S_{b_1} \rangle, \ldots, \langle S_{a_z}, S_{b_z} \rangle$, with all balls distinct and $G_h(S_{a_i}) = G_h(S_{b_i})$ $(1 \leq i \leq z)$. Figure 4.3 shows the probability of finding six two-way collisions with $n = 236650$ bins, for a varying number of balls. We select the number of bins such that the signer with 1024 balls has a 50% probability of finding a signature after one try.

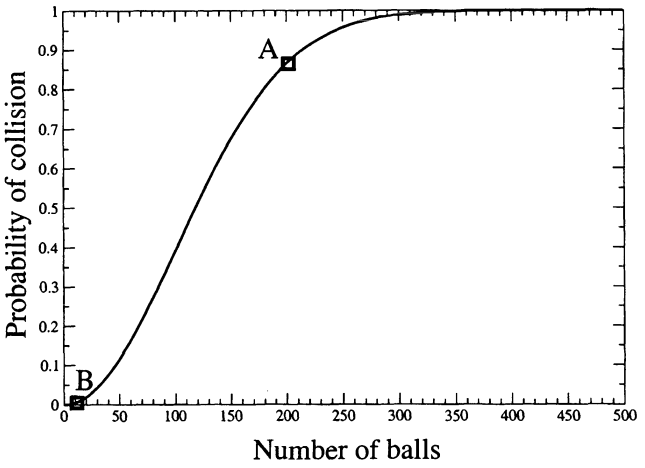


Figure 4.2. Probability of finding a two-way collision when throwing x balls into 10000 bins. The letters A and B indicate the number of balls the signer and adversary have, respectively. This figure illustrates the asymmetric nature of the BiBa signature: the signer has a high probability of finding a collision, but the adversary has a low probability of finding a collision.

The third way to increase security is to require multi-way collisions, instead of two-way collisions. If the BiBa signature requires a k-way collision, the BiBa signature of message m (with $h = H(m)$) is $\langle S_{x_1}, \ldots, S_{x_k} \rangle$, where all balls $S_{x_1}, \ldots, S_{x_k}$ are distinct and collide under G: $G_h(S_{x_1}) = \ldots = G_h(S_{x_k})$. Figure 4.3 shows the probability of finding a six-way collision with $n = 222$ bins, for a varying number of balls. We chose $n = 222$ such that the signer with 1024 balls has a 50% probability of finding a signature after one try. The figure shows clearly that this approach is better than using six two-way collisions, because the probability drops off faster for an adversary that has fewer balls.

The fourth way we attempted to improve the security is to use a multi-round scheme, where only the bins that have a k_1-way collision in the first round proceed as balls into the next round. Intuitively, multi-round schemes may seem to improve the security. However, in Section 4.6 we describe multi-round schemes in more detail and show that a one-round scheme is as secure as a corresponding multi-round scheme, assuming that an adversary knows the balls disclosed in one signature.

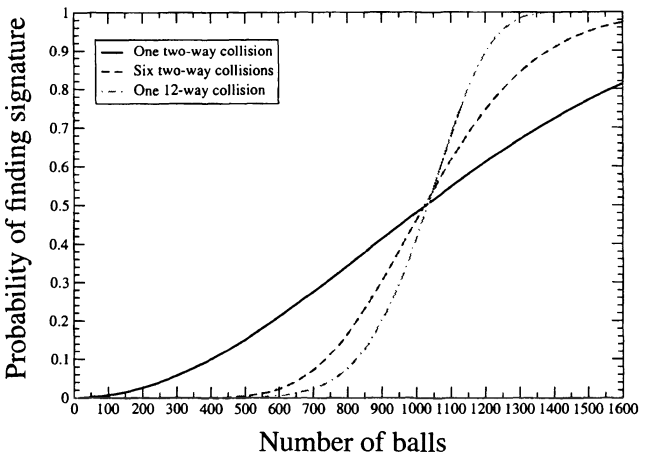


Figure 4.3. Probability of finding a signature for three cases. The solid line shows the probability for finding a two-way collision when throwing x balls into 762460 bins. The dashed line shows the probability of finding six two-way collisions when throwing x balls into 236650 bins. The dot-dashed line shows the probability of finding a six-way collision when throwing x balls into 222 bins. In this example we select the number of bins for each scheme such that the signer with 1024 balls has a 50% probability of finding a signature after one try.

4.1.7 The BiBa Signature Scheme

In this section, we describe the BiBa signature scheme in more detail. To sign message m, the signer computes the hash $h = H(m \mid\mid c)$, which is of length m_3 bits and where c is a counter value that the signer increments if it cannot find a signature. The signer has t balls (each is m_2 bits long), and maps them with the hash function G_h into n bins. Any k-way collision of balls forms the signature. Figure 4.4 shows an example where the signature consists of a 4-way collision.

Verification is straightforward. We assume that the verifier knows the BiBa parameters k and n, the hash function H, and the hash function family G.

Assume the verifier receives the message m and also the BiBa signature $\langle S_{x_1}, \ldots, S_{x_k}, c \rangle$. First, the verifier verifies that all k balls are distinct and authentic. Next, the verifier computes $h = H(m \mid\mid c)$, and accepts the signature if all $G_h(S_{x_i})$ (for $1 \leq i \leq k$) are equal. The box labeled Protocol 4.1 summarizes the BiBa protocol.

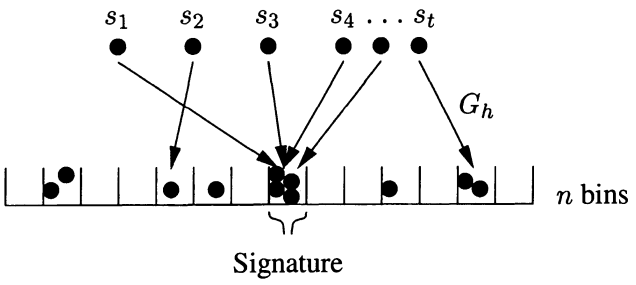


Figure 4.4. Basic BiBa signature. The function G_h maps the balls $s_1 \ldots s_t$ into the n bins. A four-way collision of balls in a bin form the signature.

1 **Setup**. The signer first generates the private key, by selecting t random balls $s_1, \ldots, s_t$. Next, the signer generates the public key, by committing to each ball in the private key using a commitment function F as Section 2.4 describes. The public key is thus $F(s_1), \ldots, F(s_t)$. We assume that the verifier learns the public key in an authenticated manner. Other public parameters are the number of bins n, and the number of balls that need to collide k.

2 **Signature generation**. To sign a message m, the signer first resets the counter $c = 0$, and computes a hash $h = H(m \mid\mid c)$. The signer uses the hash h of the message to pick the hash function G_h from a family of random hash functions. The signer uses G_h to map each ball s_i of the private key to a bin $G_h(s_i)$. Next, the signer looks for a k-way collision. If no k-way collision exists, the signer increments c and restarts. If a k-way collision exists, the signer collects the k balls and publishes them along with the counter: $\langle s'_1, \ldots, s'_k, c \rangle$.

3 **Signature verification**. The verifier receives message m and the signature $\langle s'_1, \ldots, s'_k, c \rangle$. First, the verifier verifies that all k balls are distinct. Next, the verifier checks that all balls are authentic, by checking that $F(s'_1)$ through $F(s'_k)$ are all elements of the public key. Finally, the verifier computes $h = H(m \mid\mid c)$, and accepts the signature if all $G_h(s'_i)$ (for $1 \leq i \leq k$) are equal.

Protocol 4.1: Summary of the BiBa signature algorithm.

4.1.8 Security Considerations

Given a number of disclosed balls, we can derive the probability that an adversary can find a valid BiBa signature using standard combinatorial tech-

k	n	P_f	k	n	P_f
2	762460	$2^{-19.5403}$	13	192	$2^{-91.0196}$
3	15616	$2^{-27.8615}$	14	168	$2^{-96.1001}$
4	3742	$2^{-35.6088}$	15	151	$2^{-101.3377}$
5	1690	$2^{-42.8912}$	16	136	$2^{-106.3119}$
6	994	$2^{-49.7855}$	17	123	$2^{-111.0802}$
7	672	$2^{-56.3539}$	18	112	$2^{-115.7250}$
8	494	$2^{-62.6386}$	19	104	$2^{-120.6079}$
9	384	$2^{-68.6797}$	20	96	$2^{-125.1143}$
10	310	$2^{-74.4851}$	21	89	$2^{-129.5147}$
11	260	$2^{-80.2237}$	22	83	$2^{-133.8758}$
12	222	$2^{-85.7386}$	23	78	$2^{-138.2788}$

Table 4.1. The security of some BiBa instances. The signer knows $t = 1024$ balls and the adversary has $r = k$ balls. The table shows the probability of forgery P_f to find a k-way collision when throwing k balls into n bins.

niques. In Section 4.6 we derive a tight upper bound of the probability P_f that the adversary can successfully forge a signature after one trial (r is the number of balls that the adversary knows):

$$P_f = \frac{\binom{r}{k}(n-1)^{r-k}}{n^{r-1}}$$

We assume the signer has $t = 1024$ balls, and the attacker has $r = k$ balls which is the number of balls disclosed after the signer signs one message. Let P_S denote the probability that the signer can find a signature after one trial with t balls, and P_f denotes the probability that an adversary can forge a signature after one trial knowing r balls. A high value for P_S means the signer is likely to produce a signature on the message and hence signing is efficient. For the remainder of this paper we set $P_S = 0.5$, such that the signer can find a signature after 2 tries on average. A low value for P_f means that it is unlikely for an attacker to produce a valid signature, and indicates the security of the scheme. Table 4.1 shows the P_f value for different instances with varying parameters n and k. In Section 4.4 we discuss methods on how to choose the BiBa parameters, and we discuss the resulting communication and computation costs. BiBa achieves a high level of security using only a few collisions. Considering that 2^{80} PRF operations is considered as secure

until year 2010 [LV99], a BiBa signature with an 11-way collision provides sufficient security.

An adversary has two main ways to collect balls to forge signatures. First, the adversary can simply collect balls disclosed in signatures generated by the signer. Second, the adversary can try to find balls by brute-force computation to invert the PRF F used to authenticate the balls. In our analysis, however, we assume that the latter attack is impractical, and the adversary only knows the balls that the signer discloses with a signature.

The basic BiBa signature as described so far cannot be used to commit to a secret. If the signer can select the private key after it knows the secrets it needs to sign, it can create a specific private key that will produce the same signature for each secret it wants to commit to. Consider the following setting. The signer claims that it can always predict whether the sun is shining or whether it is raining on the next day. The signer creates two messages: $M_1 = \langle \text{sunny} \rangle$, $M_2 = \langle \text{rainy} \rangle$. The signer computes $h_1 = H(M_1 \mid\mid 0)$, $h_2 = H(M_2 \mid\mid 0)$, and selects a random bin x, where the balls of the two signatures will collide in. When selecting the balls for the private key, the signer selects k balls $s_1, \ldots, s_k$ with the property that $G_{h_1}(s_i) = G_{h_2}(s_i) = x$, for $1 \leq i \leq k$. Given a random ball, the probability that it satisfies this requirement is n^{-2}. The signer would then pass the public key to the verifier, along with the signature $\langle s_1, \ldots, s_k, 0 \rangle$. On the next day, depending on the weather, the signer then discloses either M_1 or M_2, either of which will be signed by the signature.

Fortunately, we can guard against this attack by adding the hash of the public key when computing the hash of the message. Let h' be the hash of the public key. We now compute the hash of the message M as follows: $h = H(M \mid\mid h' \mid\mid c)$. This prevents the signer from picking balls as described above, because any new ball will change the public key, which changes h', which again changes the bin a ball falls into.

Finding two messages with the same signature is improbable: $\frac{1}{\binom{t}{k}}$. (This analysis assumes that finding a collision of the hash function H is less likely.) In the case of $t = 1024$ and $k = 11$, the probability that two messages have the same signature is $2^{-84.7}$. However, the signer can take advantage of the birthday paradox to find two messages with the same signature. With the same parameters, the signer can expect to find such a message after calculating $\approx 2^{43}$ signatures. If we want a signature that prevents this case, we need to increase k and t. Other one-time signatures that only sign an 80 bit hash value suffer from the same problem [BM94, BM96b, BM96a, EGM90, Lam79, Mer88, Mer90,

Rab78, Roh99, Zha98], the signer can expect to find two messages that have the same signature after trying $\approx 2^{40}$ random messages.

4.2 The BiBa Broadcast Authentication Protocol

In this section we describe how we use the BiBa signature to design the BiBa broadcast authentication protocol.

A broadcast authentication protocol requires that each receiver can verify that data originates from the sender. An obvious approach is for the sender to compute a BiBa signature on each message it broadcasts. Since the sender can only disclose a small number of balls, it could only sign a small number of messages (given a public key which commits to a fixed number of balls). For a viable broadcast authentication protocol, however, the sender needs to authenticate a potentially infinite stream of messages. So we construct a protocol that replenishes the balls disclosed with each signature. In a straightforward approach, the sender adds a new commitment (for each ball that it discloses) to the packet, and includes all the new commitments in the signature. This approach doubles the size of the signature and is not robust to packet loss. We now present a better approach for constructing the balls. We will not review signature generation and verification in this section, since it is the same as in the BiBa signature scheme that we describe in Section 4.1.7.

4.2.1 One-way Ball Chains

For our authentication protocol, we need a method that allows the receiver to instantly authenticate the balls when it receives them, and to automatically replenish balls. We use *one-way chains* to achieve the self-authenticating property of balls and for replenishment. One-way chains are used in many schemes, for example by Lamport in a one-time password system [Lam81], and the S/Key one-time password system [Hal94].

We use the PRF F to generate the one-way ball chains, and the PRF F' to generate a one-way salt chain in the following construction. The sender first generates the *one-way salt chain* of length l, $\{K_i\}_{1 \le i \le l}$, using the PRF F' as follows: the sender randomly selects K_l (of length m_1 bits): $K_l \xleftarrow{R} \{0,1\}^{m_1}$, and then recursively computes all other salts: $K_i = F'_{K_{i+1}}(0)$ $(1 \le i < l)$.

The sender then generates a set of *one-way ball chains*, $\{S_{\langle i,j \rangle}\}_{1 \le i \le t, 1 \le j \le l}$, where $S_{\langle i,\_ \rangle}$ forms a one-way chain as Figure 4.5 shows. The ball chains are constructed as follows. The sender first randomly selects all the seed ball values $S_{\langle \_,l \rangle}$ of length m_2 bits: $S_{\langle i,l \rangle} \xleftarrow{R} \{0,1\}^{m_2}$ $(1 \le i \le t)$. The sender then computes all other ball values recursively: $S_{\langle i,j \rangle} = F_{S_{\langle i,j+1 \rangle}}(K_{j+1})$ $(1 \le j <$

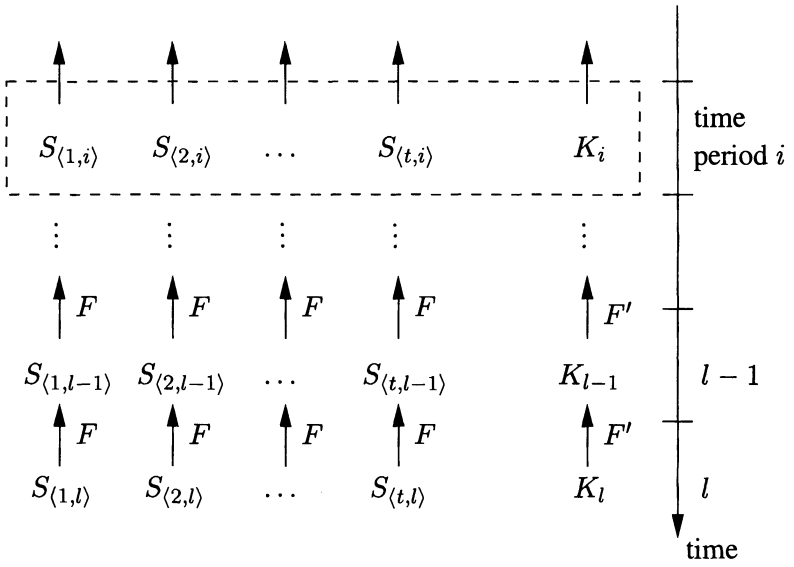


Figure 4.5. Using one-way chains to construct balls. Each vertical chain is for one ball, the arrows show how balls are derived: $S_{\langle 1,i-1\rangle} = F_{S_{\langle 1,i\rangle}}(K_i)$. The salt chain is derived as follows: $K_{i-1} = F'_{K_i}(0)$. The dashed box highlights the balls and salt that are active in time period i. Time advances from top to bottom.

l). Note the way we use the salts of the one-way salt chain to derive the ball values, so an attacker first would need to find a pre-image of the salt of the one-way salt chain before it could try to find pre-images for the ball chains. We chose this specific construction to allow for relatively compact balls, while the longer salts mitigate attacks to find other pre-images for the balls by pre-computation. However, if the balls are long enough to prevent such attacks, the one-way salt chain may not be necessary.

The sender divides the time up into time periods of equal duration T_d. In each time period i, the balls $S_{\langle \_,i\rangle}$ and the salt K_i are *active*. Figure 4.5 shows the time periods and the corresponding *active balls* and *active salt*. As time advances an entire row of balls expires and a new row becomes active. The sender publishes each salt at the beginning of the time period when it becomes active, but the sender only discloses the active balls of a time period that are part of a BiBa signature.

To bootstrap a new receiver we assume for now that the sender sends it all the balls and the salt of a previous time period over an authenticated channel. We present extensions that provide more efficient receiver bootstrapping in Section 4.4. It is clear that a receiver who knows all the authenticated balls

and salt of a time period can authenticate balls and salts of later time periods. For example, assume the receiver knows the authentic salt K_i of time period i. When the receiver receives K_{i+1} of the following time period the receiver authenticates it by verifying that K_i equals $F'_{K_{i+1}}(0)$. The receiver authenticates balls by following the one-way ball chain back to a ball that it knows is authentic.

4.2.2 Security Condition

To prevent an adversary from forging a signature, we need to ensure that an adversary knows few active balls. Hence, when the receiver receives a packet, the receiver has to be certain that an adversary could only know a small number of balls. The receiver can verify such a condition if it is time synchronized with the sender and knows the sending schedule of packets. We refer to TESLA for more details on time synchronization [PCST01]. Assuming a maximum time synchronization error of δ between the sender and the receivers, the sender is limited to signing $\lfloor r/k \rfloor$ messages within time δ, where r is the maximum number of active balls that the adversary can know, and k is the number of balls revealed in a signature. When the receiver gets a packet, it needs to verify that the sender did not yet disclose more than r active balls. Because of the one-way ball chains, balls of one time period also disclose balls from previous time periods. Hence we require that the sender does not use a BiBa instance for time δ after it disclosed k balls of that instance. To send continuously, the sender needs to use multiple BiBa broadcast authentication instances in a round-robin fashion.

4.3 BiBa Broadcast Protocol Extensions

We briefly present two extensions to the BiBa broadcast protocol. An optimal protocol would satisfy the following three properties: low receiver computation overhead, low communication overhead (only the disclosed balls are in the packet), and perfect robustness to packet loss. Finding an optimal protocol is an open problem. However, we can achieve two out of the three properties. The standard BiBa broadcast authentication protocol we describe in the previous section has low communication overhead, perfect robustness to packet loss, but requires more receiver computation overhead than the standard BiBa signature protocol (to verify the authenticity of the balls). In this section, we propose the extension A and B. Extension A does not tolerate packet loss, and extension B has a higher communication overhead. Hybrid schemes exist, but

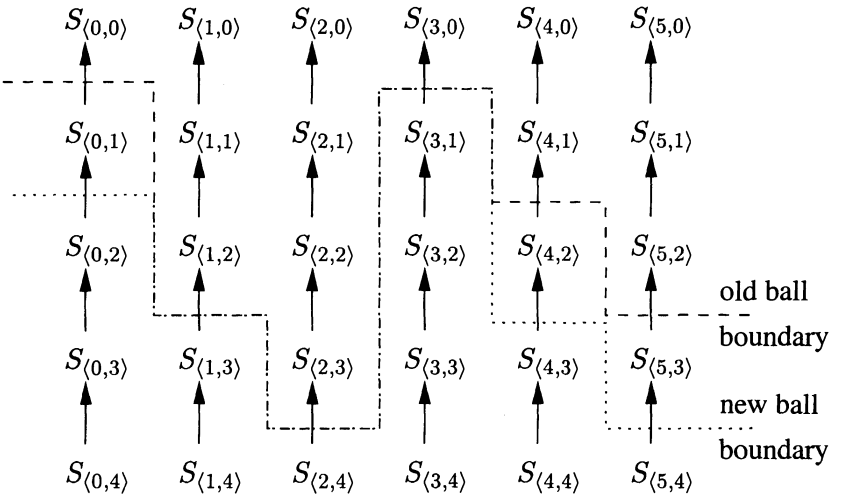


Figure 4.6. The ball boundary. In contrast to the example in Figure 4.5, this example does not use the salt chain, so the balls are derived as follows: $S_{\langle i,j\rangle} = F_{S_{\langle i,j+1\rangle}}(0)$. The signer disclosed all the balls above the ball boundary. The signer uses the balls adjacent and below the ball boundary to sign future messages. For instance, if the signer uses balls $S_{\langle 0,1\rangle}$, $S_{\langle 4,2\rangle}$, and $S_{\langle 5,3\rangle}$ in a signature, the boundary will move down such that these balls are above the boundary, shown as new ball boundary in the figure.

we do not describe them here. The difference among the three protocols is how they manage the balls.

4.3.1 Extension A

Extension A provides low receiver computation overhead and low communication overhead, but it does not tolerate packet loss. The basic BiBa broadcast authentication protocol has a high receiver computation overhead because the majority of the balls are never used in a signature, so to authenticate a ball the receiver needs to recompute many balls in a one-way ball chain until it reaches a previously stored ball. We solve this problem in extensions A and B by using every ball of each one-way ball chain in a BiBa signature.

In extension A we use the concept of *ball boundary*, which Figure 4.6 depicts. The balls above the boundary are disclosed, and they serve as commitment to the balls on the other side of the boundary. The sender and receiver always know the current ball boundary. The sender only uses balls that are directly adjacent to (below) the boundary. After each BiBa signature the sender and receiver extend the ball boundary past the newly disclosed balls.

This scheme would not be secure if an adversary could slow down the data traffic to the receiver, and collect enough packets such that it knows a large number of balls on the lower side of the perceived ball boundary of the receiver. This large number of balls would enable the adversary to spoof subsequent data traffic, because the adversary continuously receives fresh balls that the sender discloses. This illustrates that the sender and receiver need to be time synchronized, such that the receiver knows the sending schedule of the packets.

As an additional security measure, the sender can also sign all the balls directly above the current ball boundary with the message signature. This mechanism would ensure the receiver that it knows the correct ball boundary.

4.3.2 Extension B

Extension B is similar to extension A, but it tolerates packet loss. Extension A does not tolerate packet loss because the receiver needs to know which balls the sender discloses so it can update the ball boundary. To improve the tolerance to packet loss, we add ball boundary information to packets, which increases the communication overhead. This approach allows us to trade off robustness to packet loss with communication overhead.

Two main methods exist to encode the ball boundary in packets: *absolute encoding* or *relative encoding*. The absolute encoding sends the index of each ball of the ball boundary in the packet. For instance the ball boundary in Figure 4.6 is $\langle 0, 2, 3, 0, 1, 2 \rangle$. A relative encoding only communicates the change of the ball boundary with respect to a previous boundary.

As in extension A, the receiver needs to know the exact sending rate of messages, to verify that the ball boundary always grows by the number of balls that the sender discloses. To prevent that an attacker collects more than r balls, the receiver needs to receive at least one packet every ν packets (where $\nu = \lfloor r/k \rfloor$). Hence this scheme does not tolerate more than $\nu - 1$ consecutively lost packets. Otherwise, an attacker could collect balls during a long period of packet loss, and forge subsequent packets by claiming a bogus ball boundary.

4.4 Practical Considerations

In this section we first discuss how to derive the BiBa parameters from a given set of system requirements, and we analyze the impact of BiBa parameters on the overhead and security. Next, we look at the overhead of signature generation and verification. We will then discuss a concrete example on how to construct a viable broadcast authentication scheme and discuss the performance.

4.4.1 Selection of BiBa Parameters

We assume that the sender has $t = 1024$ balls. Let P_f denote the probability that an attacker can find a signature with one trial of one message knowing at most r balls. The security parameter is generally expressed as the expected number of hash function operations that an adversary has to perform to forge a signature [NES99]. For BiBa, the minimum number of hash function operations to forge a signature is $2/P_f$; for simplicity we use $1/P_f$.

Let P_S denote the probability that the sender can find a signature in one trial. The expected number of tries that the sender performs to find a signature is $1/P_S$. Without loss of generality, we set $P_S = 0.5$.

To achieve good security, the sender can disclose approximately up to $\gamma = 10\%$ of the balls. Each signature reveals k balls. The sender knows t balls, so it can produce $\nu = \lfloor \gamma \cdot t/k \rfloor$ signatures in a time period. As we discuss in Section 4.2, the sender needs to wait for time δ until it can disclose the balls of the next time period. Hence it needs multiple BiBa instances if it wants to send more than ν messages per time period δ. Given the packet sending rate β, the number of BiBa instances needed is $\lceil \delta\beta/\nu \rceil$.

We now discuss how we choose n and k. The choice of k directly determines the signature size. We can derive the number of bins n from k and the probability P_S that the sender finds a signature after one trial. Figure 4.7 shows how P_S decreases as we increase n.

Once we fix n and k we can derive the number of balls that the sender can disclose such that the adversary has at most a probability of P_f to forge a signature. Figure 4.8 depicts the probability distribution to find a signature given a certain number of balls. As we can see in Figure 4.8(a), P_f quickly decreases as the sender decreases the number of balls it discloses. If P_f is too high (insufficient security) for a k-way collision, we need to increase k.

4.4.2 BiBa Overhead

Table 4.2 lists the computation, memory, and communication overhead of the sender and the verifier during the precomputation, signature generation, and signature verification phases, where T_F, T_G, and T_H denote the time to compute the functions F, G, H, respectively.

4.4.3 Example: Real-time stock quotes

Consider a real-time stock quote broadcasting system. The main requirement is a low authentication delay for the real-time data, hence buffering on either the sender or receiver side is not an option. Another requirement is the

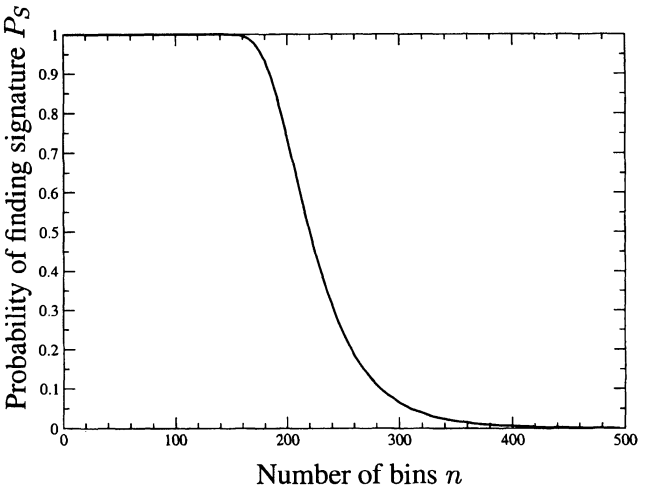
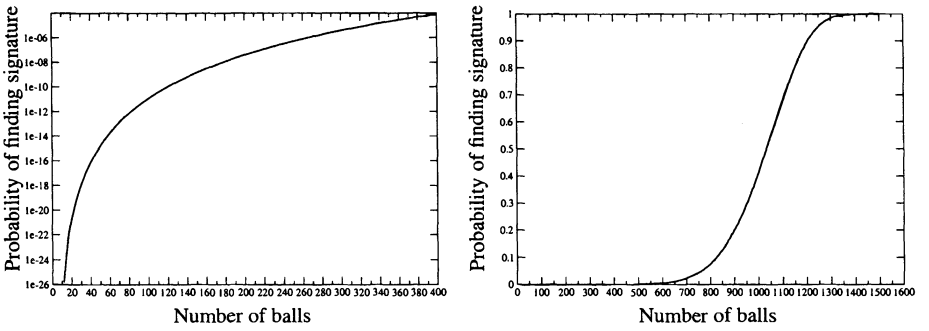


Figure 4.7. This figure shows the probability of finding a twelve-way collision when throwing 1024 balls into n bins. As we increase the number of bins, the probability for the signer to find a signature P_S decreases. We choose n such that $P_S = 0.5$, which in this case is for $n = 222$ as Table 4.1 shows.



(a) $12 \leq x \leq 350$ (b) $12 \leq x \leq 1600$

Figure 4.8. Probability of finding a signature given x balls for the BiBa instance with $k = 12$ (12-way collisions) and $n = 222$ bins. Figure a demonstrates that the probability of finding a signature is exceedingly small for small numbers of balls.

robustness to packet loss, so receivers can instantly and efficiently authenticate

	Computation	Memory
Precomputation	$l(t+1)T_F$	$l(m_1 + t \cdot m_2)$
Signature Generation	$(t \cdot T_G + T_H)/P_S$	$l(m_1 + t \cdot m_2)$
Signature Verification	$2 \cdot k \cdot T_G + T_H$	$m_1 + (k+n) \cdot m_2$

Table 4.2. BiBa overhead. The salts are m_1 bits long, and the balls are m_2 bits long. The communication overhead (signature size) is $k \cdot m_2$ ($+m_1$ if we also send the salt).

each message they receive, despite previously lost packets. To the best of our knowledge, no previous protocol satisfactorily addresses these requirements.

We assume the following system requirements. Receivers are time synchronized with the sender, with a maximum time synchronization error of ten seconds ($\delta = 10$s). The sending rate is approximately 10 packets per second ($\beta = 10$ p/s). The sender has $t = 1024$ ball chains. We set the security parameter $1/P_f = 2^{58}$ (the attacker needs to perform 2^{58} hash function computations within time δ to forge a signature). We set $\gamma = 1/16$, so the adversary can know up to $r = t\gamma = 64$ active balls.

For this example, we set $m_1 = 128, m_2 = 64, k = 16$. We then compute n to get P_S close to 50% and compute the corresponding P_f assuming that the adversary knows 64 balls. From Table 4.1 we pick $n = 136$, and the resulting forging probability is $P_f = 2^{-58.03}$. The signature size is about 128 bytes.

Each signature discloses 16 balls, hence after 4 packets an attacker knows at most 64 balls. Each row of the ball chains is active for 400ms (10 packets/s and 4 packets sent per row). Because $\delta = 10s$ is the maximum time synchronization error, we cannot disclose any balls of the next row for 10 seconds. Otherwise we would disclose more balls of the previous rows which the adversary could use to forge a signature. This requires that we use 25 instances of the BiBa scheme in parallel in a round-robin fashion.

The sender overhead to generate a signature is approximately $(1024 \cdot T_G + T_H)/P_S$. On a 800 MHz Pentium III the sender can compute $\approx 10^6$ MD5 hash function evaluations per second (uniprocessor, software-based implementation of MD5). On this architecture, generating one BiBa signature takes approximately 2 ms, about 5 times faster than to generate a 1024-bit RSA signature using the OpenSSL library. BiBa enjoys a linear speedup for multiple processors, which a hardware implementation can easily exploit. With maximum parallelization, generating a BiBa signature only requires two sequential hash function computations.

Signature verification (excluding the verification of the authenticity of the balls) only requires 17 hash function evaluations. However, the ball verification is about 256 PRF evaluations on average. This is because the 1024 active balls of one time period are amortized only by 4 packets. On a 800 MHz Pentium III Linux workstation the sender can compute $\approx 5 \cdot 10^6$ RC5 function evaluations per second (uniprocessor, software-based implementation of RC5). On this architecture, verification takes on the order of 50μs, which is about 20 times faster than verifying a 1024-bit RSA signature (Section 2.3 lists the RSA performance).

Decreasing the security requirement and increasing the number of disclosed balls reduces the ball verification overhead (e.g., if the adversary knows 128 balls, which would result in $P_f = 2^{-41.2}$, then the verifier only needs to perform 128 PRF evaluations on average to authenticate the balls). Using either extension A or B would reduce the verification overhead to 17 hash function or 16 PRF computations, respectively, however with the tradeoffs we describe in Section 4.3.

4.4.4 Efficient Public-Key Distribution

Sending the public key to all receivers is a potential bottleneck. In the schemes we discuss in this paper, the public key size is on the order of 10 KBytes for each BiBa instance. We now present a trick that makes public key distribution efficient for the sender, but requires a longer time to bootstrap receivers. The intuition is that receivers can collect balls while they receive signed messages, and reconstruct the one-way ball chains and the one-way salt chain. Periodically, the sender broadcasts a message containing the hash of all balls and the salt of one time period, signed with a traditional digital signature scheme. Once the receiver collects all ball chains, it can authenticate them with the digital signature and authenticate subsequent traffic. This assumes that the receiver is already time synchronized with a maximum time synchronization error δ. The well-known coupon collector problem predicts how long the receiver needs to wait: After collecting $t \cdot \log(t)$ random balls, it has one ball of each one-way chain with high probability, where t is the number of ball chains. In the schemes we consider in this paper $t = 1024$, hence the receiver needs to collect about $1024 \cdot \log(1024) = 7098$ balls. In our first example, the sender discloses 64 balls in each time period, so the receiver needs to collect balls during 110 time periods.

4.5 Variations and Extensions

Here we discuss some variations and extensions on BiBa.

4.5.1 Randomized Verification to Prevent DoS

Many applications that rely on digital signatures are susceptible to a denial-of-service (DoS) attack: an attacker floods the victim with a large number of bogus signatures. Because signature verification is generally a slow and expensive operation (a 1024-bit RSA signature takes on the order of 0.5 millisecond to verify on a 800 MHz Pentium III processor), the victim is computationally overwhelmed just checking all signatures. BiBa has a nice property: even if a forger can find a signature where $k-1$ balls land in the correct bin, a verifier that checks the balls of the signature in random order discovers the bad ball after checking $k/2$ balls with a probability of $1/2$. In practice, the forger can find even fewer matching balls, so the verifier can detect an invalid signature after a few hash function computations. The BiBa signature is thus appropriate to defend against these DoS attacks.

4.5.2 Multi-BiBa

This section presents multi-BiBa, a variation of the BiBa signature. Multi-BiBa provides multiple signatures from a single public key, and fast signature verification. The BiBa signature scheme already provides fast signature verification, hence the computation impact of bogus signatures is negligible (on the order of a few hash function computations). We now present a multi-BiBa mechanism which allows the sender to sign m messages, with the drawback that the signature size increases by a factor $\log_2(m)$, compared to the standard BiBa signature.

We achieve this property by using a Merkle hash tree for each ball [Mer80]. Figure 4.9 shows the Merkle hash tree for ball S_i. The scheme depicted in the figure allows us to sign 4 messages. The sender uses each leaf node in one signature computation, so to sign message j, the sender considers all balls at leaf nodes $S_{\langle \_,j \rangle}$. From our description of Merkle hash trees in Section 2.4.2 it is clear that the signer has to send $\log_2(m)$ values for each ball, hence the signature expansion.

In this scheme, the receiver can first verify the authenticity of each ball, which requires $\log_2(m)$ hash function computations per ball. After authenticating each ball, the receiver can verify that that ball really collides with the previous balls, which requires one more hash function evaluation. If we assume that the forger can find a $k/2$-way collision, using a randomized order to

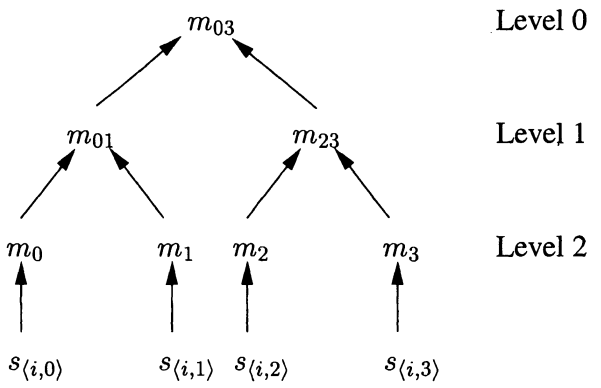


Figure 4.9. Merkle hash tree for ball i (same as Figure 2.2 on Page 26).

verify the BiBa signature will require approximately $2 \cdot (\log_2(m) + 1)$ hash function computations, which is negligible compared with traditional signature verification overhead (for reasonable values of m).

4.5.3 The Powerball Extension

This section introduces the Powerball signature, an improvement over the basic BiBa signature. Powerball is based on the following observation. The original BiBa scheme has a fixed number of known signature patterns, i.e., a collision of k balls in one bin is a valid signature pattern. In BiBa, these patterns are implicit; all the participants agree on them. In Powerball, the signature patterns are explicit. In the same way the signer commits to t balls in the public key, the signer also commits to t' patterns P_i ($1 \leq i \leq t'$). Each pattern specifies k bins, so $P_i = \langle b_1, \ldots, b_k \rangle$.

As in BiBa, to sign message M, the signer computes the hash of the message $h = H(M \mid\mid c)$ (c is a counter that the sender increments if it didn't find a signature) and uses h to select a one-way function g_h from a family of hash functions G (in the random oracle model [BR93]). The hash function g_h maps each ball to one of the n bins. To find a valid signature, the signer searches for a *complete pattern* P_i, where every bin in the pattern contains a ball. (If a bin appears β times in the pattern, the corresponding bin contains at least β balls.) If the signer finds a complete pattern P_i, it creates the signature $\langle B_{\alpha_1}, \ldots, B_{\alpha_k}, P_i, c \rangle$ (where α_j are the indices of the balls that landed in the bins of pattern P_i).

To verify the signature $\langle B_{\alpha_1}, \ldots, B_{\alpha_k}, P_i, c \rangle$ on message M, the verifier performs the steps: (1) check that all balls of the signature are distinct ($B_{\alpha_i} \neq$

B_{α_j} for $i \neq j$); (2) verify the authenticity of the balls using the public key (check that the commitment $F(B_{\alpha_i})$ is in the public key); (3) verify the authenticity of the pattern P_i using the public key (check that the commitment $F(P_i)$ is in the public key); (4) compute $h = H(M \mid\mid c)$ and select g_h from the one-way function family; (5) verify that the k balls cover all k bins of pattern $P_i = \langle b_1, \ldots, b_k \rangle$, so $g_h(B_{\alpha_1}) = b_1, \ldots, g_h(B_{\alpha_k}) = b_k$.

Let us consider the ratio of success between the sender and the forger in this model. The forger knows k balls and a pattern. Recall that in the standard scheme with $k+1$ balls sent, we found an upper bound on this ratio $\frac{t^{k+1}}{(k+1)!}$. The probability of success for the forger in our new scheme is $P_f = \frac{k!}{n^k}$.

For the signer, we approximate the expected number of matched patterns, which in turn approximates P_s. For simplicity we assume that the signer has $t' = t$ possible patterns; we further assume that the system is arranged so that these patterns are distinct. As before, the probability that each bin is covered is upper bounded by $(t/n)^k$; this is a good approximation if n is much larger than t. Hence our approximation for P_s is t^{k+1}/n^k, and hence the ratio between the sender and forger is $\frac{t^{k+1}}{k!}$.

Adding $t' = t$ commitments of the patterns to the public key would double its size, a rather severe additional cost. We introduce a method to add the patterns to the public key without increasing its size. Imagine that the ball is the commitment of the pattern, so a commitment in the public key commits to both the ball and the pattern. We call this structure a *Powerball*. We create a Powerball by specifying a bit string that represents a pattern P_i. (For now we assume a simple mapping from bit strings to patterns.) The ball B_i is derived from the pattern P_i using the one-way function F: $B_i = F(P_i)$. The commitment C_i is then a further application of F on B_i: $C_i = F(B_i) = F(F(P_i))$. This requires the additional assumption that F is one-way and pseudo-random, so that we may assume the balls are distributed independently and uniformly at random. Some hash functions claim this property [BR93].

Note that the forger can obtain a $(k+1)$st ball from P_i by computing $B_i = F(P_i)$. We solve this problem by requiring that the ball B_i does not occur as a ball in the signature. If the forger does not have another pattern, it cannot use B_i because it has to use the only pattern it knows.

Table 4.5.3 shows results from simulations of the Powerball scheme. Comparing with Table 4.1, we see that the Powerball scheme does improve performance. A Powerball is worth almost another ball; that is, using $k + 1 = 11$ Powerballs is almost as good as requiring 12 balls to fall into a bin using the original BiBa scheme.

k	n	P_f
9	1734	$2^{-78.37}$
10	1548	$2^{-84.17}$
11	1407	$2^{-89.79}$
12	1295	$2^{-95.23}$
13	1204	$2^{-100.50}$

Table 4.3. Results with the Powerball scheme when a signature pattern uses k bins, and therefore $k+1$ Powerball are used.

We can slightly enhance the advantage for the signer by further changing the meaning of a Powerball. For example, suppose we require that two Powerballs must be combined in some order to represent a pattern. For example, we may take the exclusive-or of bits in the P_i in order to obtain a pattern. In this case we use $k+2$ Powerballs to represent a signature; k correspond to balls, and two correspond to a pattern. In this case we still have $P_f = \frac{k!}{n^k}$. On the other hand, for the signer we have $E[X] \approx \frac{t^{k+2}}{2n^k}$. Note the introduction of the factor of two in the denominator, since there are $\binom{t}{2}$ possible patterns for the signer. Hence the upper bound on the ratio P_s/P_f is about $t^{k+2}/2k!$. This is a factor $\binom{k}{2}$ better than the scheme without Powerballs. Again, there are tradeoffs to using such mechanisms, including the difficulty for the signer to find a matched pattern, so these Powerball variations may be of theoretical interest only. However, this demonstrates how small changes in the model can lead to different analyzes.

A similar idea can be used to reduce the size of the public key, which is very large in the standard BiBa scheme. Suppose we require that two Powerballs be combined, say via an exclusive-or, in order to construct a ball. In this case, a sender with t Powerballs has roughly $\binom{t}{2}$ balls to play with; this number is not exact because we restrict each pair of Powerballs to be disjoint. Now a forger with k non-pattern Powerballs has $(k-1)\cdot(k-3)\cdot\ldots 3\cdot 1$ ways of pairing up the k Powerballs into $k/2$ actual balls. Hence at the cost of increasing the power of the forger somewhat (by giving the forger more than one set of balls to use), we can dramatically reduce the size of the public key. Whether this tradeoff is useful may depend on the desired system parameters.

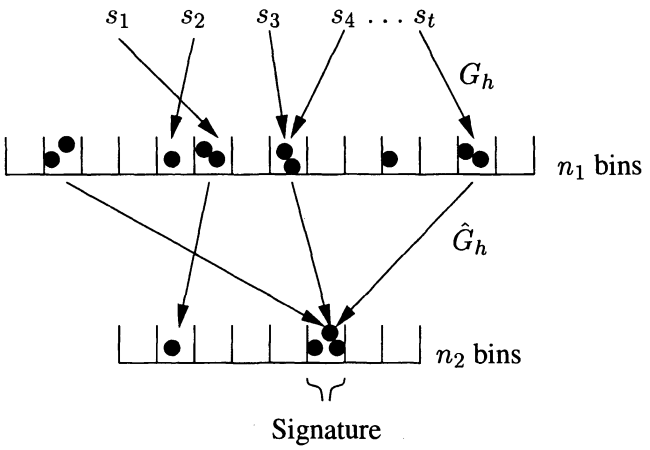


Figure 4.10. Signature scheme with two rounds. In this example, all the bins that have at least two balls in the first round, become balls in the second round. Finally, a three-way collision in the second round constitutes the signature.

4.6 One-Round BiBa is as secure as Multi-Round BiBa

We illustrate a multi-round scheme with a concrete example. Figure 4.10 illustrates this approach for two rounds. In the first round, we look for two-way collisions under a hash function G_h. The indices of the bins that have at least a two-way collision in the first round are used as balls in the second round, and a three-way collision is required under $\hat{G}_h$. The indices that participate in the three-way collision in the second round and the balls necessary to verify that each index corresponds to a bin with the two-way collision in the first round, form the BiBa signature: $\langle S_{x_1}, \ldots, S_{x_6} \rangle$.

To verify the BiBa signature $\langle S_{x_1}, S_{x_2}, S_{x_3}, S_{x_4}, S_{x_5}, S_{x_6} \rangle$ of message m (with $h = H(m)$), the verifier verifies the following conditions:

1 All six balls S_i are distinct and authentic.

2 Throwing all six balls into n_1 bins results in three two-way collisions in bins with indices b_1, b_2, b_3. These indices form a three-way collision under $\hat{G}$: $\hat{G}_h(b_1) = \hat{G}_h(b_2) = \hat{G}_h(b_3)$.

We now sketch a proof that a one-round BiBa signature scheme is as good as a multi-round signature scheme.

Lemma 4.6.1 *A one-round BiBa signature is asymptotically as secure as a two-round BiBa signature.*

Proof:[sketch] We start out by showing that a one-round scheme offers the same security as the corresponding two-round scheme with the same signature size. We then use an inductive argument to show that an $n+1$-round scheme is no more secure than an n-round scheme.

Some of the following formulas are due to von Mises's seminal article on the occupancy probabilities in bins and balls models [vM39]. The expected number of bins that contain exactly k balls (k-way collisions) after throwing t balls into n bins is:

$$E[k] = \frac{\binom{t}{k}(n-1)^{t-k}}{n^{t-1}} \tag{4.1}$$

We use the following inequalities:

$$E[k] = \frac{\binom{t}{k}(n-1)^{t-k}}{n^{t-1}} < \frac{\binom{t}{k}(n)^{t-k}}{n^{t-1}} = \frac{\binom{t}{k}}{n^{k-1}} \tag{4.2}$$

$$\frac{\binom{t}{k}}{n^{k-1}} = \frac{t!}{(t-k)!k!n^{k-1}} < \frac{t^k}{k!n^{k-1}} \tag{4.3}$$

We write $P_k(x)$ for the probability that exactly x k-way collisions occur. In the settings that we study, we have $P_k(1) > P_k(2)$, and in particular for the adversary who tries to forge a signature we have $P_k(1) \gg P_k(2) \gg P_k(3)$. Similarly, we have $P_k(1) \gg P_{k+1}(1)$ since the signer can barely find a k-way collision, and for the adversary we have $P_{k+1}(1) = 0$ because it only knows k balls that were disclosed in a signature. From the first equality of equation 4.1 we approximate the probability to find a valid signature

$$\begin{aligned} P_f &\approx P_k(1) + P_k(2) + \cdots \\ &= E[k] - P_k(2) - 2P_k(3) - \cdots \\ &\approx E[k] \\ P_f &\approx \frac{\binom{t}{k}(n-1)^{t-k}}{n^{t-1}} \end{aligned}$$

We use equations 4.2 and 4.3 to derive the number of bins such that the signer has the probability of $P_S \approx 50\%$ to find a signature, given that the signer has t balls and the signature is composed of a k-way collision:

$$\begin{aligned} P_S \approx \frac{t^k}{k!n^{k-1}} &= \frac{1}{2} \\ n^{k-1} &= \frac{2t^k}{k!} \end{aligned} \tag{4.4}$$

We can now compute the probability that the adversary can forge a signature, given that it has seen a single signature (hence it has k balls), and using equation 4.4, we get:

$$P_f = \left(\frac{1}{n}\right)^{k-1} = \frac{k!}{2t^k} \tag{4.5}$$

For the two-round scheme, we assume that the signer has t balls, and that it looks for a k_1-way collision in the first round, and a k_2-way collision in the second round. It is clear that the signature comprises $k_1 \cdot k_2$ balls, so since we require that the signature size is the same for the one-round and two-round schemes we have $k = k_1 \cdot k_2$. We assume that the number n_1 of bins in the first round is fixed, and we compute the number of bins in the second round such that the probability of the signer P_S to find a signature is approximately 50%. The expected number of k_1-way collisions e_1 in the first round is

$$e_1 = \frac{t^{k_1}}{k_1! n_1^{k_1-1}}$$

We can now compute the number of bins n_2 of the second round

$$P_S \approx \frac{e_1^{k_2}}{k_2! n_2^{k_2-1}} = \frac{1}{2}$$

$$n_2^{k_2-1} = \frac{2e_1^{k_2}}{k_2!} = \frac{2t^{k_1 k_2}}{k_2!(k_1! n_1^{k_1-1})^{k_2}} \tag{4.6}$$

The probability that the attacker can find another signature with $k_1 \cdot k_2$ balls is

$$P_f = P_c \cdot \left(\frac{1}{n_2}\right)^{k_2-1} \tag{4.7}$$

where P_c is the probability to find k_2 k_1-way collisions after throwing $k_1 \cdot k_2$ balls into n_1 bins:

$$P_c = \frac{\binom{n_1}{k_2}(k_1 \cdot k_2)!}{(k_1!)^{k_2} n_1^{k_1 k_2}} \tag{4.8}$$

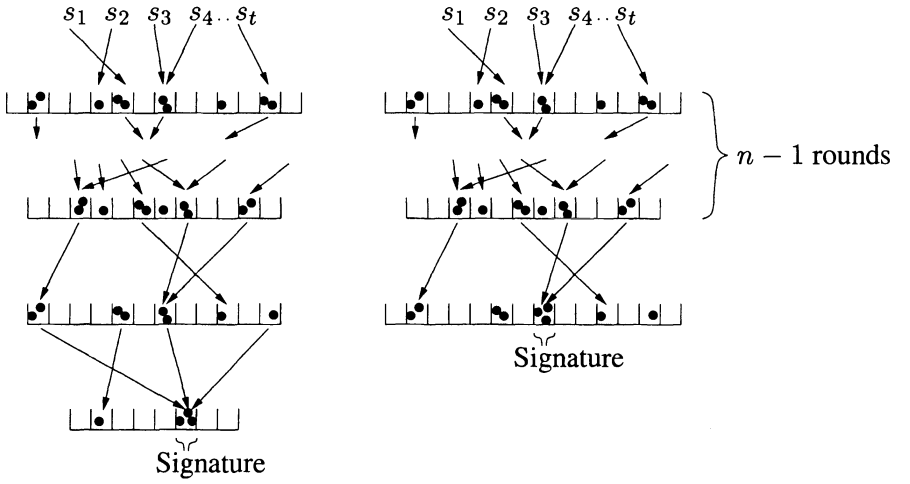


Figure 4.11. Companion figure to the proof sketch that a N-round BiBa scheme is as secure as a $N+1$-round scheme.

Combining equations 4.6, 4.7, and 4.8 we get

$$\begin{aligned} P_f &= \frac{\binom{n_1}{k_2}(k_1 \cdot k_2)!}{(k_1!)^{k_2} n_1^{k_1 k_2}} \frac{k_2!(k_1!)^{k_2} n_1^{(k_1-1)k_2}}{2t^{k_1 k_2}} \\ &= \frac{\binom{n_1}{k_2}(k_1 \cdot k_2)! k_2!}{(n_1)^{k_2} 2t^{k_1 k_2}} \\ &\approx \frac{(k_1 \cdot k_2)!}{2t^{k_1 k_2}} \end{aligned}$$

When we set $k_1 \cdot k_2 = k$ in Equation 4.8 we can see that the probability to forge a signature P_f is the same as in Equation 4.3, which shows that the two-round scheme offers no better security than a one-round scheme. ■

Repeated application of Lemma 4.6.1 immediately yields:

Theorem 4.6.2 *A one-round Biba signature is asymptotically as secure as a n-round Biba signature.*

4.7 Merkle Hash Trees for Ball Authentication

We can use a Merkle hash tree for the ball authentication [Mer80]. The signer chooses the balls randomly, places them at the leaves of a binary tree and computes each internal node as the hash of the concatenation of the two

child values.[1] The root node of the hash tree becomes the public key, hence the public key is small.

The signer computes the signature as before, but it needs to add additional verification nodes of the hash tree to the signature, such that the verifier can reconstruct all the paths from all balls to the root of the hash tree. Unfortunately, the overhead of these additional verification nodes can be high, as we now compute.

We assume that the disclosed balls are distributed randomly in the hash tree. To compute the overhead, we need to compute the probability that all the leaves below a given tree node are all empty. We set function $\pi(a, r, n)$ as the probability that a leaves are empty after randomly choosing r leaves among n leaves (note that $\pi(a, r, n) = 0$ if $n - r < a$):

$$\pi(a, r, n) = \prod_{i=0}^{r-1} \frac{n - a - i}{n - i}$$

The expected number of nodes of the Merkle hash tree that need to be sent to authenticate b leaf nodes depends on the depth d of the hash tree. The number of expected nodes is:

$$\sum_{i=1}^{d} 2^i \pi(2^{d-i}, b, 2^d)(1 - \pi(2^{d-i}, r, 2^d - 2^{d-i}))$$

The intuition behind this formula is that we need to send a node of the tree if exactly one child has all empty leaf nodes, and the other child node has at least one chosen leaf node.

When we evaluate this probability for the real-time stock quote example in Section 4.4.3, we would need to add 83.3 hash tree nodes on average to authenticate the 16 ball values for the scheme in the first example, and 67.5 nodes to authenticate the 12 balls in the second example. Clearly, this would increase the signature size by a factor of 6, and increase the verification overhead.

To generalize this idea, we can construct many small hash trees of height d that contain 2^d balls. The public key would then contain all the root nodes of all small hash trees, and hence we reduce the size by a factor of 2^d. To authenticate each ball, the signer adds the d verification nodes to each ball.

[1] A minor point is that the signer needs to compute a one-way function on the ball before placing the ball at the leaf node. Otherwise the signer would disclose neighboring balls when it discloses additional nodes for verification.

Hence, the public key size is reduced by a factor of 2^d and the signature size is expanded by a factor of d.

Chapter 5

EMSS, MESS, & HTSS: SIGNATURES FOR BROADCAST

The previous two chapters present two protocols to authenticate broadcast streams. These protocol allow a receiver to authenticate the sender of a message, but they do not provide a signature, which would allow the receiver to convince a third party that the sender really did send that message. So how can we sign (or provide non-repudiation) for each broadcast message?

Broadcast authentication, provided by TESLA or BiBa, can not provide a signature, because any receiver can forge receiver data after the sender discloses sufficiently many keys. Some applications, however, require a digital signature for a broadcast stream (e.g., to provide non-repudiation). So this chapter presents protocols to sign individual messages of a broadcast stream.

Time synchronization is difficult in some settings. In other settings, data is cached and distributed later, so timely packet delivery is not provided. In these settings TESLA and BiBa do not work. We propose to use a broadcast signature scheme for broadcast authentication, without the need for time synchronization.

A stream signature scheme should provide the following properties:

- Low computation overhead for generation and verification of authentication information
- Low communication overhead
- No buffering required for the sender and the receiver
- Instant signature verification for each individual message
- Strong robustness to packet loss

A. Perrig et al., *Secure Broadcast Communication*

- Scales to large number of receivers

Signing broadcast messages by adding a digital signature to each message satisfies almost all desired properties, but this is expensive in terms of communication, and computation for both sender and receiver. To reduce the overhead of digital signatures, we amortize the cost of one signature across multiple messages. But how can we support amortized signatures in the face of packet loss? Many researchers have worked on this problem [GR97, WL98, Roh99, GM01, MS01], and we review their work in Chapter 8.

We take the following approach to amortize one digital signature over multiple messages[1]: the sender adds the hash of a packet to the following packet. Periodically, the signer digitally signs a packet, we call such a packet a *signature packet*. Obviously, the receiver can verify the signature of the signature packet, so the message in the signature packet is signed. The signature packet contains the hash of the preceding packets, so the signature in the signature packet also signs the hash of the preceding packet which is equivalent with signing the preceding packet as well. Similarly, the signature extends over all preceding packets.

To verify the signature on a packet, the receiver needs to follow the path of hashes in packets until it reaches a signature packet. Unfortunately, packet loss might sever the path to the signature packet preventing the receiver from authenticating packets. To tolerate packet loss, we use redundant hashes, adding the hash of a packet to multiple following packets.

A natural way to discuss this signature scheme is in a graph-theoretic setting [MS01]. Signature packets are special nodes, other packets correspond to regular nodes, and embedding the hash of a packet into another packet corresponds to a directed edge (which originates from the packet that adds its hash, and ends in the packet that caries the hash). In this setting, a packet is signed if it is a special node (i.e., a signature packet), or if there exists a directed path from the packet to a special node.

Note that cryptographic hash functions restrict us from using cyclic graphs. To illustrate this, consider a short cycle over two nodes. The signer would have to find h_1 and h_2 given M_1 and M_2, such that $h_1 = H(M_1 \mid\mid h_2)$ and $h_2 = H(M_2 \mid\mid h_1)$. If H is a cryptographically secure hash function, we do not know how to find h_1 and h_2, so the signer would have to find h_1 and h_2

[1] We describe our use of the terms *messages* and *packets* in Section 2.1.

through brute-force computation, which is computationally infeasible. We thus only consider acyclic graphs.[2]

In the following discussion we use the term *hash link* to describe adding a hash to a later packet. In this chapter we first present our Efficient Multicast Stream Signature (EMSS), which uses constant distances for the hash links. We then present MESS,[3] which uses a randomized method to pick hash links. If the entire stream content is known in advance, we present a Hash Tree Stream Signature (HTSS) in Section 5.4.

5.1 Efficient Multicast Stream Signature (EMSS)

EMSS uses constant hash link distances (Section 5.2 considers a more general setting). The hash links of a packet form a *pattern*, and we call the pattern in EMSS a *static pattern*. In EMSS, the number of outgoing hash links is constant, hence the number of incoming links (i.e., the number of hashes embedded in each packet) is constant as well.

We now discuss the direction of the hash links. As long as the graph is acyclic, the sender can put the hash of a packet in either an earlier or later packet. If the sender puts the hash of a later packet into the current packet, it could not send the current packet until it knows the content of the later packet. Sender-side buffering would solve this, but delays sending. If all packets were linked (the first packet depends on the last packet), the sender would need to know the entire stream content before sending the first packet. (One scheme by Gennaro and Rohatgi uses such an approach, so the sender must know all content before it sends the first packet [GR97].) Because we target EMSS to real-time streams, we assume in the rest of this section that the sender embeds the hash links of the packet into later packets.[4]

---

[2] In a followup work to our early work on EMSS, Miner and Staddon do consider cyclic graphs — they construct forward hash links (in the same way we construct them) by embedding the hash in the corresponding packet, and they use the TESLA construction for backwards links [MS01]. We do not consider this approach here, since we target our stream signature protocol also for settings where time synchronization is not available.

[3] The name MESS does not stand for anything — since it uses randomized links it is a messy version of EMSS. We are indebted to Mike Luby for suggesting this name.

[4] If the sender knows the entire stream content in advance, it can still use our scheme and reverse all the hash links — our analysis also applies for that case. An advantage of using backwards links is that the receiver can instantly authenticate messages that it receives. This is similar to the instant authentication protocol of TESLA: depending on the direction of the links, we can choose between sender-side or receiver-side buffering. Reversing the direction of the links has the disadvantage that a receiver can only verify the signature of packets after it receives a signature packet. Indeed, the receiver would need to discard any packets it receives before the first signature packet. To remedy this drawback, we design the Hash Tree

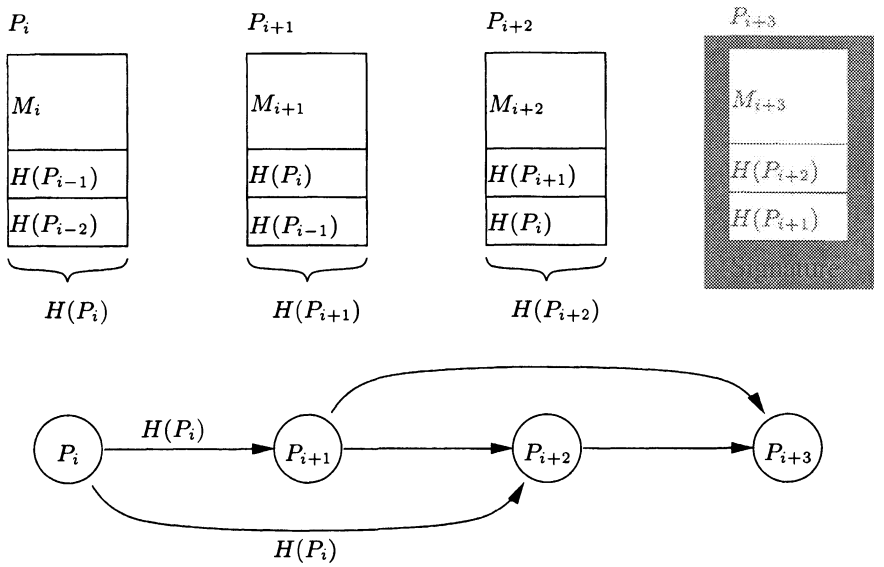


Figure 5.1. This figure shows four EMSS packets, for the static pattern $\langle 1 - 2 \rangle$ (the text in Section 5.1 describes this notation). Time advances from left to right. On top of the figure, we can see the composition of the packets, and on the bottom we can see the hash links between the same packets (each circle corresponds to a packet). Packet P_{i+3} is digitally signed, so P_{i+3} is the signature packet. P_{i+3} contains the hash of packets P_{i+1} and P_{i+2}, so when the receiver receives P_{i+3}, it can also authenticate packets P_{i+1} and P_{i+2}. Similarly, since P_{i+1} contains the hash of packet P_i, the receiver can authenticate P_i if it knows that P_{i+1} is authentic. Reasoning in a hash-link graph, we can see that a receiver can authenticate a packet, if the receiver has all authentic packets on a path to a signature node.

We now describe EMSS in more detail. The sender embeds the hash of the current packet P_i into k following data packets where the link distance is a constant: $\{d_1, \ldots, d_k\}$. So the packets $P_{i+d_1}, \ldots, P_{i+d_k}$ carry the hash of packet P_i. We use the abbreviation $\langle d_1 - \ldots - d_k \rangle$ to describe a static pattern with k hash links with distance $d_1, \ldots, d_k$. Because EMSS uses static patterns, each packet carries k hashes. Figure 5.1 shows an example of the static pattern $\langle 1 - 2 \rangle$. The signer signs one in every N packets, so the distance between signature packets is N.

We define the receiver's probability $\mathcal{P}_v$ of packet verification as the average probability to verify a packet in an infinite stream (π_i is the probability that

Stream Signature (HTSS) protocol for the case where the sender knows the entire stream content in advance (see Section 5.4). HTSS features a low overhead, and has no authentication delay.

packet i is verifiable, regardless of whether packet P_i arrives or not):

$$\mathcal{P}_v = \frac{1}{N} \sum_{i=1}^{N} \pi_i$$

We also call $\mathcal{P}_v$ the *probability of verifiability*.

We use the probability of verifiability $\mathcal{P}_v$ as a measure for evaluating EMSS. We consider two models of packet loss: independent and correlated packet loss. Both patterns occur in real networks. Independent loss could occur if a router drops packets with a randomized drop policy (i.e., RED [FJ93]) in case of congestion. Researchers found that packet loss is correlated and that the length of losses exhibit infinite variance [Pax99]. Yajnik et al. show that a k-state Markov chain is a good model for correlated Internet packet loss [YMKT99].

We first consider the case of independent packet loss. We can sometimes use analytic methods to compute the probability of verifiability: this analytic computation only depends on the probability of packet loss, the distance of the packet from the signature packet, and the static pattern. For example, for the $\langle 1 - 2 - 4 \rangle$ static pattern, we obtain the following recursive formula for the probability that packet P_i is verifiable ($\mathcal{P}_v[i]$ denotes the probability that packet P_i is verifiable, short for $Pr[V_i]$, where V_i is the random variable that packet P_i is verifiable):

$$\begin{aligned} \mathcal{P}_v[i] \;=\; & q \cdot \mathcal{P}_v[i+1] \\ & + q \cdot (1-q) \cdot \mathcal{P}_v[i+2] \\ & + q \cdot (1+q) \cdot (1-q)^2 \cdot \mathcal{P}_v[i+4] \\ & - q^3 \cdot (1-q)^2 \cdot \mathcal{P}_v[i+5] \end{aligned} \tag{5.1}$$

Equation 5.1 is derived as follows.

$$\begin{aligned} Pr[V_i] \;=\; q \cdot (\; & Pr[V_{i+1}] \\ & + Pr[V_{i+2} \wedge \neg V_{i+1}] \\ & + Pr[V_{i+4} \wedge \neg V_{i+2} \wedge \neg V_{i+1}]) \end{aligned}$$

$$\begin{aligned} Pr[V_{i+2} \wedge \neg V_{i+1}] &= Pr[V_{i+2}] \cdot Pr[\neg V_{i+1} \mid V_{i+2}] \\ &= Pr[V_{i+2}] \cdot (1-q) \end{aligned}$$

$$Pr[V_{i+4} \wedge \neg V_{i+2} \wedge \neg V_{i+1}] = Pr[V_{i+4}] \cdot Pr[\neg V_{i+2} \wedge \neg V_{i+1} \mid V_{i+4}]$$

$$Pr[\neg V_{i+2} \wedge \neg V_{i+1} \mid V_{i+4}] = Pr[\neg V_{i+2} \mid V_{i+4}] \cdot Pr[\neg V_{i+1} \mid \neg V_{i+2} \wedge V_{i+4}]$$

$$Pr[\neg V_{i+2} \mid V_{i+4}] = 1 - q$$

$$Pr[\neg V_{i+1} \mid \neg V_{i+2} \wedge V_{i+4}] = Pr[\neg V_{i+3} \wedge \neg V_{i+5} \mid \neg V_{i+2} \wedge V_{i+4}] \cdot q + (1 - q)$$

$$Pr[\neg V_{i+3} \mid \neg V_{i+2} \wedge V_{i+4}] = 1 - q$$

$$Pr[\neg V_{i+5} \mid \neg V_{i+2} \wedge \neg V_{i+3} \wedge V_{i+4}] = Pr[\neg V_{i+5} \mid V_{i+4}]$$

$$\begin{aligned} Pr[\neg V_{i+5} \mid V_{i+4}] &= 1 - Pr[V_{i+5} \mid V_{i+4}] \\ &= 1 - \frac{Pr[V_{i+5} \wedge V_{i+4}]}{Pr[V_{i+4}]} \end{aligned}$$

$$Pr[V_{i+5} \wedge V_{i+4}] = q \cdot Pr[V_{i+5}]$$

$$\begin{aligned} Pr[V_i] &= q \cdot Pr[V_{i+1}] \\ &\quad + q \cdot (1 - q) \cdot Pr[V_{i+2}] \\ &\quad + Pr[V_{i+4}] \cdot (1 - q) \cdot q \cdot \left((1 - q) + q \cdot (1 - q) \cdot \left(1 - \frac{q \cdot Pr[V_{i+5}]}{Pr[V_{i+4}]} \right) \right) \\ &= q \cdot Pr[V_{i+1}] \\ &\quad + q \cdot (1 - q) \cdot Pr[V_{i+2}] \\ &\quad + q \cdot (1 - q)^2 \cdot (1 + q) \cdot Pr[V_{i+4}] \\ &\quad - q^3 \cdot (1 - q)^2 \cdot Pr[V_{i+5}] \end{aligned}$$

where $\mathcal{P}_v(i)$ is the probability of verifiability for packet i (i.e., there exists a path of hash links from packet P_i up to the signature packet P_N), and q is the probability that a packet arrives.

The probability of verifiability $\mathcal{P}_v$ for an arbitrary static pattern is challenging even in the case of independent packet loss. To get accurate numbers, we perform simulations. We can verify the accuracy of the simulation using the cases for which we know the exact solution, such as the static patterns $\langle 1 - 2 - 4 \rangle$ and $\langle 1 - 2 - 3 - 4 \rangle$. Our simulation shows that the verification probability diverged at most 2% from the analytical result (in both cases). We computed the probability of verifiability for the 1000 packets preceding the signature packet, averaged over 2500 simulations.

We ran extensive simulations to find a good distribution of edges withstanding high packet loss. We searched through all combinations of six edges per node, where the maximum length of any edge was 51, and the probability of

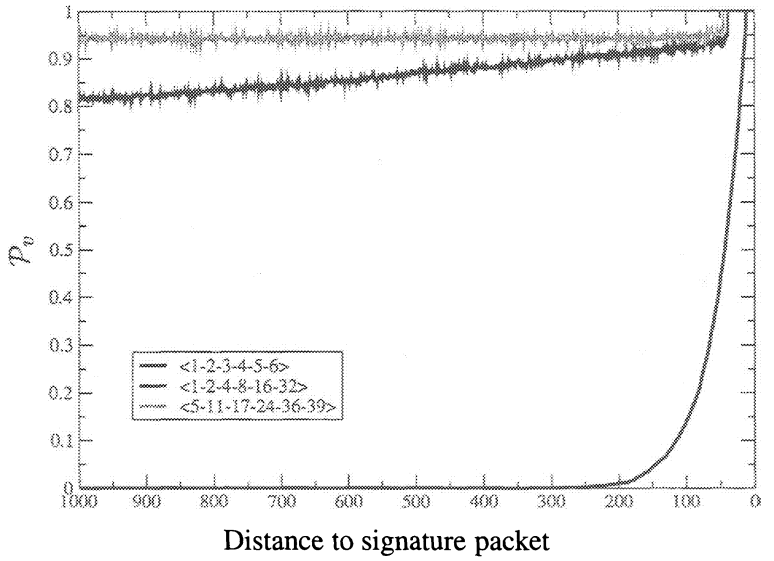


Figure 5.2. EMSS simulation for the three different static patterns $\langle 1-2-3-4-5-6\rangle$, $\langle 1-2-4-8-16-32\rangle$, and $\langle 5-11-17-24-36-39\rangle$. Time advances from left to right. The graph shows 1000 packets, followed by seven final signature packet (that we assume all arrive at the receiver). The y axis shows the probability of verifiability $\mathcal{P}_v$ for each packet. The figure nicely shows that $\mathcal{P}_v$ increases, as the distance between a packet and the signature packets decreases. The text discusses the simulation results in more detail.

dropping a node was 60%. In our simulation we assumed that the seven final packets all arrived and that they were all signed.

Figure 5.2 shows the results of our simulation. We found that the static pattern $\langle 1-2-3-4-5-6\rangle$ performs poorly, i.e., it has a very low probability that the packet is verifiable unless the distance to a signature packet is small. The pattern $\langle 1-2-4-8-16-32\rangle$ performs much better. However, the pattern $\langle 5-11-17-24-36-39\rangle$ emerged as particularly robust to packet loss. We could not find a method that accurately predicts the resilience to packet loss given a particular pattern, but we do have some observations. First, a pattern is more resilient to loss if it has far-reaching hash chains, because far-reaching hash chains often allow a short path to a signature packet (with few intermediate links). It is obvious that the probability of all packets of a path with few intermediate links arriving is larger than the probability that all packets of a path with many intermediate links all arrive. (Far-reaching links increase the robustness to packet loss, however, they generally also increase

the delay of packet verification.) This helps explain the poor performance of the $\langle 1-2-3-4-5-6 \rangle$ pattern in Figure 5.2. Second, the lengths of hash links should be co-prime. A pattern where the lengths of hash links are not co-prime is weak, because many different paths to the signature packet depend on the same set of packets. If one of these packets is lost, many hash link paths are destroyed. This helps explain the poor performance of the $\langle 1-2-4-8-16-32 \rangle$ pattern in Figure 5.2.

5.1.1 EMSS Summary and Security Argument

The box labeled Protocol 5.1 summarizes the EMSS protocol. For the EMSS protocol to be secure, we need the following requirements:

- The receivers need to receive an authentic public key K_S of the sender.
- The digital signature algorithm must be secure against existential forgery under a chosen message attack.
- The hash function H needs to be second pre-image collision resistant.

5.2 MESS

In MESS, the sender adds the hash of a packet to k randomly selected packets, uniformly distributed over the interval $[1, D]$. While other distributions may also be promising, the graph in Figure 5.3 indicates that the majority of static link patterns is highly robust to packet loss. We thus consider the uniform distribution, which also simplifies the analysis. Section 5.3 discusses other distributions.

The number of hash links that each packet carries follows a Poisson distribution (with $\lambda = k$) [vM39]. Table 5.1 shows the probability distribution of the number of incoming links in a scheme for $k = 3$.[5]

As in EMSS, the sender signs each Nth packet, and the probability that each packet arrives is q. We consider the case of independent and correlated packet loss separately. We first discuss independent packet loss.

[5] In practice, the varying size of the MESS information per packet is inconvenient since many protocols have a fixed header size, such as the MESP header [CRC00]. If we allocate a header sufficiently large to store hash links for 99% of all packets (e.g., space for 8 links in the example of Table 5.1), the empty entries would waste precious packet payload in most packets. Ideally, each packet carries the same number of hash links as in EMSS. We discuss schemes with this property in Section 5.3, but for now we assume that we simply embed the hash links of a packet in k randomly chosen following packets.

1 **Setup**. The sender selects a static pattern $\langle x_1 - \ldots - x_k \rangle$. We assume that all x_i are distinct and that x_k is the largest offset. The sender allocates x_k vectors of k dimensions: $\vec{v}_1, \ldots, \vec{v}_{x_k}$. The k elements of each vector contain the hash values of the previous packets. The vectors are all initialized to the null vector $\vec{0}$.

The sender S has a digital signature key pair, with the private key K_S^{-1} and the public key K_S. We assume a mechanism that allows a receiver R to learn the authenticated public key K_S, as well as the static pattern $\langle x_1 - \ldots - x_k \rangle$.

The signer signs every Nth packet with the private key K_S^{-1}.

2 **Sending packets**. To send message M_i, the sender attaches the vector $\vec{v}_1$ to the message, and computes the hash: $h = H(M_i \mid\mid \vec{v}_1)$. If this packet is a signature packet (i.e., $i \bmod N = 0$), the sender sends a signed packet: $S \rightarrow * : \{M_i, \vec{v}_1\}_{K_S^{-1}}$, otherwise, the sender sends an unsigned packet: $S \rightarrow * : M_i, \vec{v}_1$. Next, the sender assigns the vectors to the preceding vector: $\vec{v}_i \leftarrow \vec{v}_{i+1}$ for $1 \leq i \leq x_k - 1$, and $\vec{v}_{x_k} \leftarrow \vec{0}$. The sender then stores the hash of the current packet into the vectors (where $\vec{v}[i] \leftarrow y$ means that we store y at the ith element of the vector): $\vec{v}_{x_i}[i] \leftarrow h$, for $1 \leq i \leq k$.

3 **Verifying packets**. We describe a simple, but probably not the most efficient mechanism to authenticate packets. The receiver stores all packets in a buffer, resetting the *authenticated* flag. After it receives a signed packet P_i, it first verifies the digital signature. If the digital signature is authentic, the receiver trusts the current packet and starts with the following procedure, otherwise it discards the packet.

```
for j = i downto 0 do
  if packet P_j authentic then
    split up packet P_j into components: P_j = M_j, v_j
    for jj = 1 to k do
      split up packet P_{j-x_jj} into components: P_{j-x_jj} = M_{j-x_jj}, v_{j-x_jj}
      compute h = H(M_{j-x_jj}, v_{j-x_jj})
      if h equals v_j[jj] then
        mark packet P_{j-x_jj} as authentic
      endif
    endfor
  endif
endfor
```

Protocol 5.1: EMSS protocol summary.

# links j	0	1	2	3	4	5	6	7	8
Probability	0.050	0.149	0.224	0.224	0.168	0.101	0.050	0.022	0.008

Table 5.1. This table shows the probability distribution that a packet has j incoming hash links (i.e., the packet carries j hashes), for MESS with $k = 3$ outgoing hash links per packet.

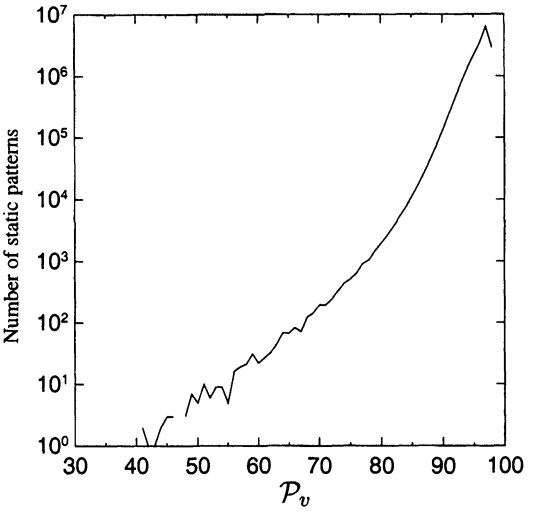


Figure 5.3. This figure shows the number of static link patterns that achieve an average $\mathcal{P}_v$. Number of combinations of six hashes that resulted in a given average verification probability. The probability of packet loss is 60%.

5.2.1 Analysis for Independent Packet Loss

We take a pragmatic approach in this section: we approximate $\mathcal{P}_v$ for the case of independent packet loss, so we can easily compute the number of hash links we need given a certain rate of packet loss. In our analysis, we assume that the maximum length of hash links D is a multiple of the distance between signature packets N, so all packets have an equal probability that their hash is included in a signature packet, independent of the distance to the next signature packet. We analyze MESS for streams of infinite length. If the packet is a signature packet, the probability of verifiability is 1. To simplify the analysis, we assume that if a packet is not a signature packet, then the probability of verifiability is a constant $\mathcal{P}_v = c_v$.[6]

We introduce the random variable L_i: for each packet, if $L_i = 1$, the receiver can authenticate the packet by following hash link i, otherwise $L_i = 0$. Since a packet can have at most k hash links $1 \leq i \leq k$. We compute the

[6]We do not know if this assumption is true or not, although experimental results support it. We can attempt to justify this assumption through a symmetry argument. Since each packet has an equal probability to add its hash to a signature packet, and since each packet adds its hash to the same number of other packets, all packets (besides the signature packet) are indistinguishable. Since no packet has an advantage over another packet, all packets have the same probability to be verifiable.

probability for L_i as follows:

$$Pr[L_i = 1] = \frac{1}{N} q + \frac{N-1}{N} q \mathcal{P}_v \tag{5.2}$$

(Recall that q is the probability that a packet arrives.) Equation 5.2 is straightforward: the link is either verifiable if the packet that embeds the link is a signature packet (probability $\frac{1}{N}$) and that it arrives (probability q), or the packet is not a signature packet (probability $\frac{N-1}{N}$) and it arrives and it is verifiable (probability $\mathcal{P}_v$).

To approximate $\mathcal{P}_v$ we assume that all L_i are independent. In reality, however, it is easy to see that they are not independent. Since we never add the same hash link twice to the same packet, it is clear that if the first link ends in a signature packet, the probability that any one of the other links are embedded in a signature packet is zero (assuming $N = D$).

Nevertheless, our goal is to approximate $\mathcal{P}_v$ such that we can easily find out how many hash links we need to use to achieve a certain probability of verifiability. By combining our two assumptions (that $\mathcal{P}_v$ is equal for all packets and that all L_i are independent) we obtain the following equation:

$$\begin{aligned} \mathcal{P}_v &\approx 1 - \prod_{i=1}^{k} Pr[L_i = 0] \\ &= 1 - (Pr[L_i = 0])^k \\ &= 1 - (1 - Pr[L_i = 1])^k \end{aligned} \tag{5.3}$$

By combining Equation 5.2 with 5.3 we get:

$$\begin{aligned} \mathcal{P}_v &\approx 1 - \left(1 - \frac{1}{N} q - \frac{N-1}{N} q \mathcal{P}_v\right)^k \\ &= 1 - \left(1 - \frac{q}{N}\left(N \mathcal{P}_v + (1 - \mathcal{P}_v)\right)\right)^k \end{aligned} \tag{5.4}$$

We are only interested in schemes where the probability of verifiability is close to one, so if we assume that $\mathcal{P}_v$ is close to 1 and for large N we have $N \cdot \mathcal{P}_v \gg (1 - \mathcal{P}_v)$ and we can simplify Equation 5.3:

$$\mathcal{P}_v \approx 1 - (1 - q \mathcal{P}_v)^k \tag{5.5}$$

Unfortunately, we cannot solve Equation 5.5 for general k. For $k = 2$ we get Equation 5.6, and $k = 3$ we get Equation 5.7.

$$\mathcal{P}_v \approx \frac{-1+2q}{q^2} \qquad (k=2,\ q \geq \frac{1}{2}) \tag{5.6}$$

$$\mathcal{P}_v \approx \frac{3q - \sqrt{-3q^2+4q}}{2q^2} \qquad (k=3,\ q \geq \frac{1}{3}) \tag{5.7}$$

Assuming a general $k > 3$, we define the following function f such that $\mathcal{P}_v$ is the root of this function ($0 < \mathcal{P}_v \leq 1$):

$$f(\mathcal{P}_v) \approx 1 - \mathcal{P}_v - (1 - q\mathcal{P}_v)^k \tag{5.8}$$

Since a viable MESS scheme has a $\mathcal{P}_v$ close to one, we look for an approximation to solve Equation 5.8 that is accurate if $\mathcal{P}_v$ is close to one. We can use one step of the iterative Newton method to numerically find the root of a function, and use 1 as the starting point. Newton's method works by drawing a tangent to the starting point and using the point of intersection with the X-axis as a better approximation for the root. For most functions, this method rapidly converges to the root value.

Since we are particularly interested in solutions that are close to 1, we use the point $(x_1 = 1, y_1 = f(1))$ as the starting point. The slope of the tangent is $f'(1)$, and the intersection point with the X-axis is $(x_2, y_2 = 0)$, where $x_2 = x_1 - y_1/f'(x_1) = 1 - f(1)/f'(1)$. We then use x_2 as an approximation to $\mathcal{P}_v$. Solving for $\mathcal{P}_v$ yields Equation 5.10:

$$f'(\mathcal{P}_v) \approx -1 + \frac{(1-q\mathcal{P}_v)^k kq}{1-q\mathcal{P}_v} \tag{5.9}$$

$$\mathcal{P}_v \approx 1 - \frac{f(1)}{f'(1)} = 1 - \frac{(1-q)^k}{1-(1-q)^{k-1}kq} \tag{5.10}$$

Figure 5.4 illustrates Equation 5.10 when $\mathcal{P}_v$ is close to 1. In this example $k = 7$ and $q = 0.4$. From Equation 5.10 we find that $\mathcal{P}_v = 0.9678$. The exact result is $\mathcal{P}_v = 0.9675$. The accuracy of this approximation is clearly sufficient for our purposes.

We verified the accuracy of our analysis with simulations. Figure 5.5 shows the result of simulating a MESS scheme where the sender adds the hash of each packet to seven following packets ($k = 7$) that are uniformly selected from the 1000 following packets. The sender signs every 1000th packet ($N = 1000$). Packet loss is independent at 60%. For the purpose of our simulation, we assume that the hashes of the final 20 packets of the stream are all contained

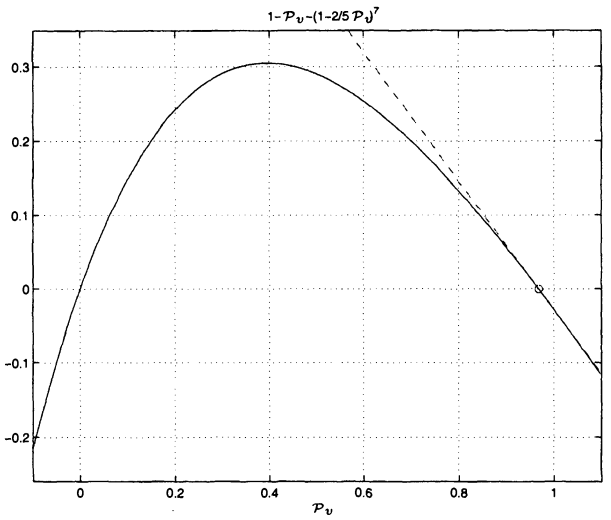


Figure 5.4. This figure illustrates how Equation 5.10 approximates $\mathcal{P}_v$ with one iteration of Newton's method ($k = 7, q = 0.4$). The solid line shows the function f (Equation 5.8), and the dashed line is the tangent on f at $x = 1$. After one iteration of Newton's method, the new $\mathcal{P}_v$ is where the tangent intersects the x-axis, and the figure shows this intersection with the circle on the dashed line. As the real $\mathcal{P}_v$ is close to 1, we can intuitively see that the approximation is quite accurate.

in a special signature packet, and that the signature packet arrives. To compute $\mathcal{P}_v$, we perform 1000 simulations and average the probability that a packet is verifiable over the packets that are between 10000 and 1000 packets away from the final signature packet. The resulting probability for $\mathcal{P}_v$ is 0.9674, which is very close to the approximated value. Note that an upper bound to the verification probability is the probability that not all packets which carry a hash link are lost is: $\mathcal{P}_v \leq 1 - (1 - q)^k$. If the packet loss probability is 60% and we use 7 links, then the verification probability is: $\mathcal{P}_v \leq 1 - (1 - 0.4)^7 \approx 0.9720$. The verification probability provided by MESS is very close to the upper bound.

Figure 5.6 plots the Equation 5.10 for $2 \leq k \leq 11$ and $0 < q \leq 1$. We can see that increasing either the number of hash links k, or decreasing the amount of packet loss (increasing q) greatly increases the probability that a packet is verifiable.

The number of hash links the sender should use depends on the packet loss probability and the required probability that a packet is verifiable $\mathcal{P}_v$. Assuming that using a $\mathcal{P}_v \geq 95\%$ is a viable choice, Figure 5.7 shows the minimum

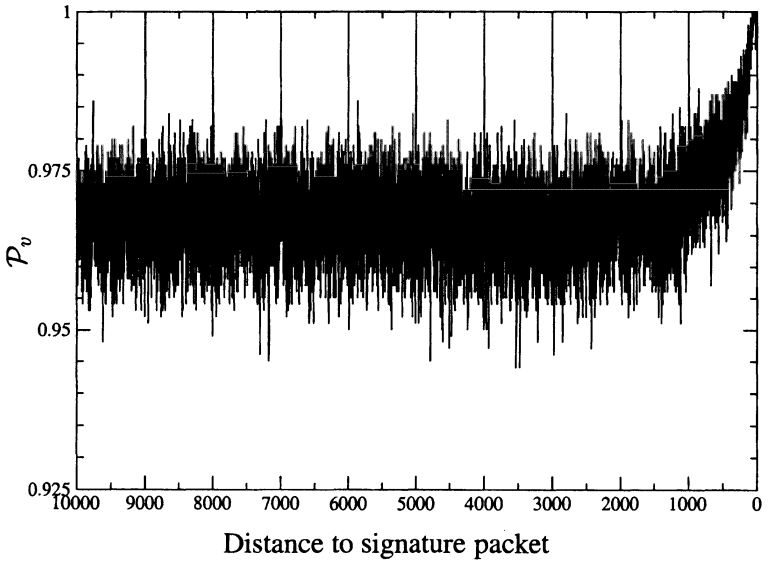


Figure 5.5. MESS simulation for independent packet loss, with $k = 7$ and $q = 0.4$. Time advances from left to right. The graph shows 10000 packets, followed by seven final signature packet (that we assume all arrive at the receiver). The y axis shows the probability of verifiability $\mathcal{P}_v$ for each packet. The sender sends a signature packet every 1000 packets. The simulation assumes that all the final signature packets arrive, so $\mathcal{P}_v$ quickly raises at the simulation boundary. The text discusses the simulation results in more detail.

number of hash links the sender needs to use to achieve a $\mathcal{P}_v$ that is at least 95%. Note that 3 hash links suffice to achieve $\mathcal{P}_v \geq 95\%$ to withstand 34% of packet loss.

5.2.2 Correlated Packet Loss

Modeling packet loss as independent loss facilitates the analysis; however, network researchers point out that packet loss in the Internet is correlated [YMKT99], i.e., the probability that a packet is lost increases if the previous packet is also lost. In this section, we analyze the MESS scheme for correlated packet loss.

We simulate correlated packet loss with a simple two-state Markov chain. It is a common practice to use Markov chains to model correlated packet loss [BSUB98, YMKT99].

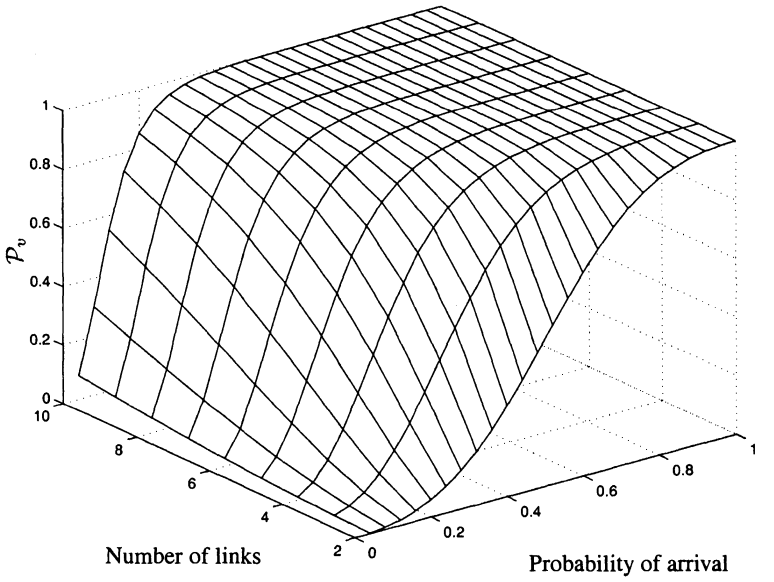


Figure 5.6. This figure plots the average $\mathcal{P}_v$ (computed with Equation 5.10) for $2 \leq k \leq 11$ and $0 < q \leq 1$. The figure nicely shows that we can compensate packet loss with more hash links per packet (larger values of k).

Figure 5.8 shows our two-state Markov chain with the transition probabilities. The Markov chain has a loss state L and a no-loss state N, with the corresponding transition probabilities. We fix two parameters of correlated packet loss: the average loss probability (or probability of arrival q), and the average length l of the sequence of lost packets. We call the a sequence of lost packets a *loss run*. From the average probability of arrival q and the average loss length l we can determine the transition probabilities $P_{NN}, P_{NL}, P_{LL}, P_{LN}$ of the Markov chain, where $P_{NN} = 1 - P_{NL}$ and $P_{LL} = 1 - P_{LN}$. We first determine P_{LL} and P_{LN} from l. From the nature of the Markov chain, the length of the loss burst follows a geometric distribution, of which the PDF is $f(x) = P_{LN}(1-P_{LN})^x = P_{LN}(P_{LL})^x$. The average loss length that we want is the mean of the geometric distribution, so we have $l = (1 - P_{LN})/P_{LN}$. Hence we can easily derive two transition probabilities: $P_{LN} = 1/(1+l)$, and $P_{LL} = 1 - P_{LN}$. Note that independent loss is a special case of our two-state Markov model for correlated loss: independent loss with a probability of arrival of q can be modeled with a two-state Markov chain with the transition probabilities $P_{NN} = P_{LN} = q, P_{LL} = P_{NL} = 1 - q$. The average loss run length is $l = \frac{1-q}{q}$.

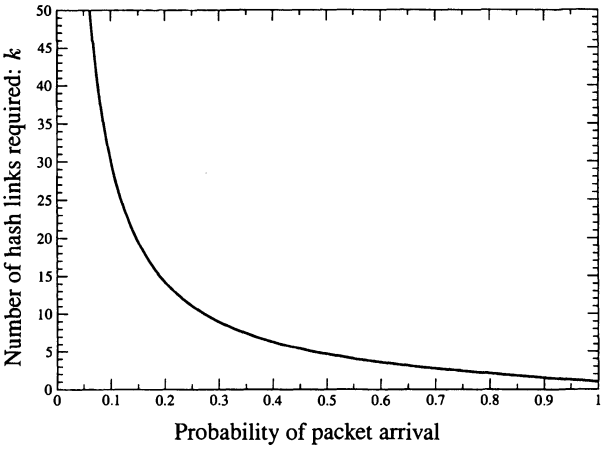


Figure 5.7. This figure shows how many hash links we need to use, such that $\mathcal{P}_v \geq 95\%$. The x-axis shows the packet arrival probability, and the y-axis shows the number of hash links we should use. Note that if the packet loss is less than 19%, two hash links suffice to achieve a verification probability of over 95%.

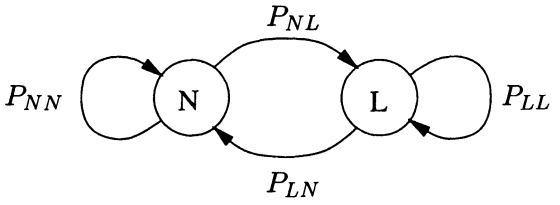


Figure 5.8. We use a simple two-state Markov chain model to simulate correlated packet loss. Our model has a no-loss (denoted with the letter N in the figure) and a loss state (denoted with the letter L in the figure). The arrows are labeled with the transition probabilities.

Next, we derive P_{NN} and P_{NL} from P_{LN} and q. Assume P_N is the probability that we are in the no-loss state, and P_L is the probability that we are in the loss state. Clearly, $P_N = q$ and $P_L = 1 - P_N$. Assuming a stationary distribution, we can write $P_N = P_N \cdot P_{NN} + P_L \cdot P_{LN}$. Solving for P_{NN} we get $P_{NN} = P_{LN} \cdot P_L / P_N = P_{LN} \cdot (1 - q)/q$.

Using our two-state Markov model with the above parameters we simulated MESS to analyze the robustness to correlated packet loss. Figure 5.9 shows the average probability (computed over 1000 simulations) that a packet is verifiable, with the parameters $k = 7$, $q = 0.4$, and $l = 50$. The average probability that a packet is verifiable is $\mathcal{P}_v = 0.943$. Clearly, this is lower than the av-

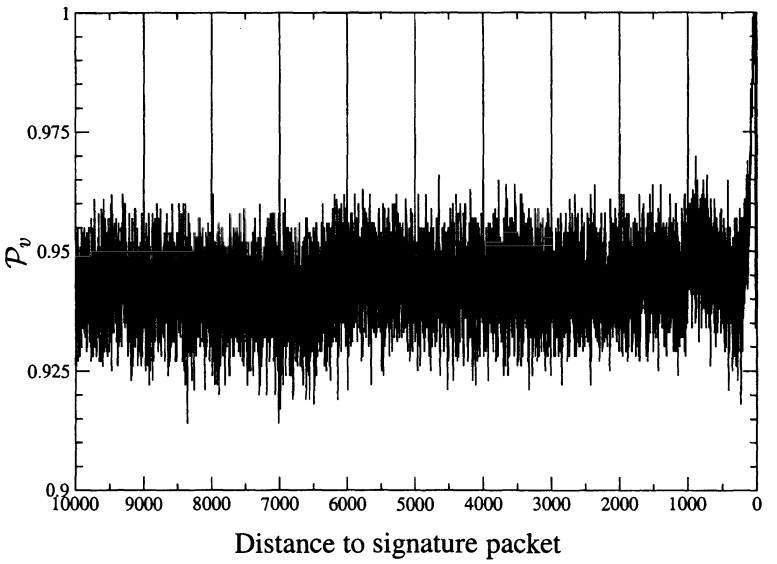


Figure 5.9. MESS simulation for correlated packet loss, with $k = 7$, $q = 0.4$, and the average burst length is 50. Time advances from left to right. The graph shows 10000 packets, followed by seven final signature packet (that we assume all arrive at the receiver). The y axis shows the probability of verifiability $\mathcal{P}_v$ for each packet. The sender sends a signature packet every 1000 packets. The simulation assumes that all the final signature packets arrive, so $\mathcal{P}_v$ quickly raises at the simulation boundary. The text discusses the simulation results in more detail.

erage verification probability of independent packet loss $\mathcal{P}_v = 0.967$. This is maybe surprising, since the average packet loss probability is the same in both simulations.

To find out why the verification probability is much lower in the case of correlated loss, we plot the relationship between average loss probability and average verification probability. Figure 5.10 plots the relationship for 1000 simulations that have independent or correlated loss. The plot clearly indicates one explanation for the lower verification probability of correlated loss: first, the variance of the loss probability (or arrival probability) is larger in the correlated loss simulations, and second, the verification probability drops off quickly for higher packet loss. Hence, when we compute the average verification probability across all simulations, it is clear that the average $\mathcal{P}_v$ is lower for correlated loss. However, even if we consider only simulations with simi-

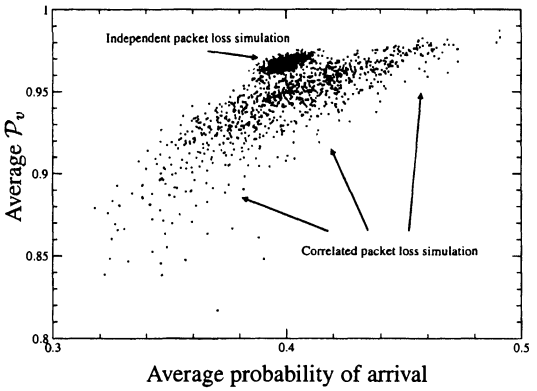


Figure 5.10. This figure compares the distribution of total packet loss to the verification probability; for independent and correlated packet loss. For each case of packet loss, we simulate 1000 independent runs with $k = 7$, $q = 0.4$. The average burst length is 50 packets. After each simulation, we compute the average packet loss (x coordinate); and the average $\mathcal{P}_v$ (y coordinate). The small, compact cloud at the top of the figure represents the 1000 samples from the independent packet loss simulation. The scattered points represent the 1000 samples from the correlated packet loss. The text describes more detail, and Figure 5.11 enlarges the area around the independent loss simulation.

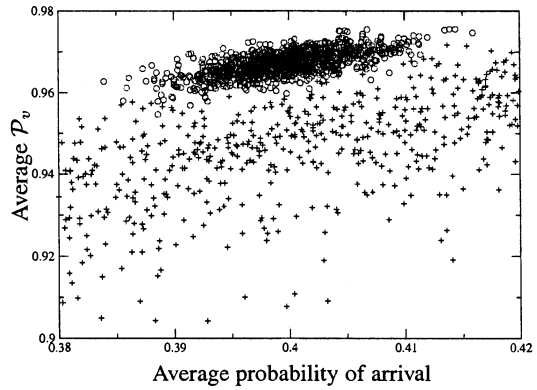
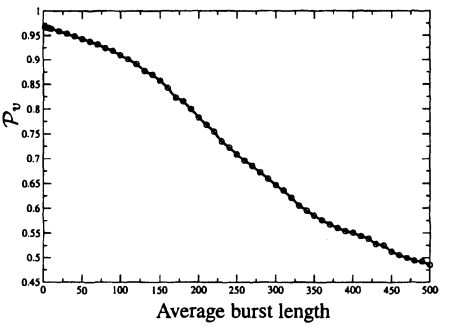
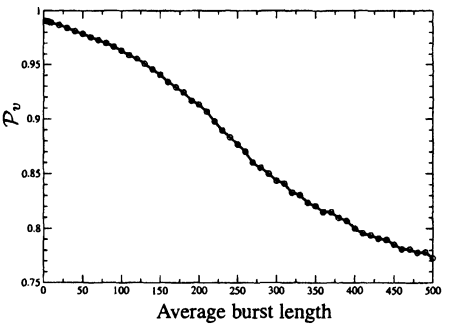


Figure 5.11. Enlarged area of Figure 5.10. Each circle symbol represents one simulation with independent packet loss, and each plus sign represents one simulation with correlated packet loss. Note that even for the same average packet loss rate, the correlated loss results in a lower verification probability.



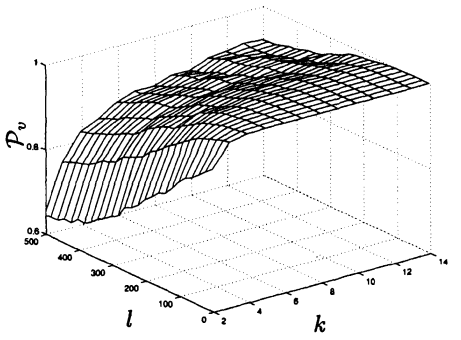
(a) $p = 0.4, k = 7$



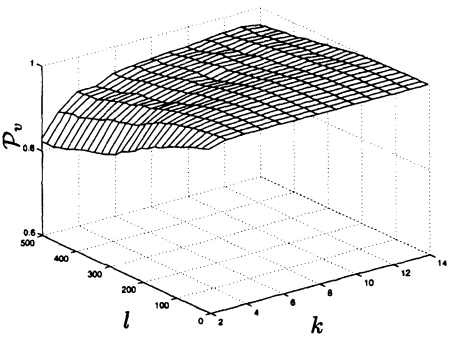
(b) $p = 0.8, k = 3$

Figure 5.12. This figure shows the change of the average $\mathcal{P}_v$ when the average burst loss length increases. For each point in the graph, we compute the average $\mathcal{P}_v$ over 10000 packets over 1000 simulations.

lar packet loss probabilities, the resulting verification probability for correlated loss is worse than for independent loss, as Figure 5.11 shows.



(a) $q = 0.8$



(b) $q = 0.9$

Figure 5.13. This figure shows how $\mathcal{P}_v$ depends on the average burst loss length ($5 \leq l \leq 505$) and the number of hash links ($2 \leq k \leq 14$).

Computing a closed form for $\mathcal{P}_v$ for correlated packet loss is an open problem. However, we ran extensive simulations to help choose the MESS param-

eters. Figure 5.12 shows the relationship between the average burst length and $\mathcal{P}_v$. Since independent loss corresponds to correlated loss with a small average burst length (around 1), we know how to compute the beginning point of each line in Figure 5.12. If we could compute another point, we could get an indication of $\mathcal{P}_v$ through a linear approximation. Figure 5.13 shows the $\mathcal{P}_v$ based on q, k, and l.

5.3 Variations

Splitting-up Hash Links

In earlier work, we suggest a variant of EMSS/MESS which we call "the extended scheme" [PCTS00]. The core idea is to split the hash into k chunks, where a quorum of any k' chunks is sufficient to allow the receiver to validate the information. One approach is to use Rabin's Information Dispersal Algorithm [Rab90], which has precisely this property. Another approach is to use a family of universal hash functions [CW79] produce independent outputs, but only look at a subset of those outputs.

The main advantage of this scheme is that *any* k' out of the k packets need to arrive, which is more robust in some circumstances than receiving 1 packet out of d in the basic scheme. For example, if we use the basic scheme with 80-bit hashes and six hashes per packet, the communication overhead is at least 60 bytes, and the probability that at least one out of six packets arrives is $P_1 = 1 - p^6$, where p is the loss probability. In contrast, if we use the extended scheme with a hash of 480 bits, chunks of 16 bits, $k = 30$, and $k' = 5$, the probability that the receiver gets more than four packets is

$$P_2 = 1 - \sum_{i=0}^{i<5} \binom{30}{i} \cdot p^{30-i} \cdot (1-p)^i$$

However, requiring only one out of k hashes has a higher verification probability than the new approach if packet loss is high. For our example above, $P_1 > P_2$ when $p > 80\%$. Although both probabilities only provide an upper bound on the verification probability, splitting up hash links still provides higher robustness to packet loss when packet loss is low.

We need to select the hash sizes with care; for example, 1-bit hash sizes would lead to a scheme that is not secure, because an attacker could produce a fake message and drop the packets when the 1-bit hash does not match.[7]

[7] We are indebted to Ross Anderson for this observation.

We now compute the probability of an adversary forging a packet in the extended scheme. The signer generates k independent pieces of c bits each, and the receiver needs to receive at least k' pieces to authenticate the packet. When the attacker chooses a random message, the probability that it can generate a valid signature by selecting a set of k' packets and discarding the other packets is:

$$Pr[\text{match}] = 1 - \sum_{i=0}^{k'-1} \binom{k}{i} \left(\frac{k}{2^c}\right)^i \left(1 - \frac{k}{2^c}\right)^{k-i}$$

Hash Link Distributions

As we discuss in the beginning of Section 5.2 our simple randomized method for choosing hash link distances results in an uneven number of incoming hash links. In fact, we show that the number of incoming hash links follows a Poisson distribution. Hence, the size of the MESS data varies widely. Ideally, the size of the MESS data would remain constant to prevent data wastage in protocols that require fixed-size fields [Atk95]. In addition, many streaming applications prefer fixed-size packets, to ideally exploit the maximum transfer unit (MTU) of a given network.

Various mechanisms achieve fixed-size data for MESS:

- If the packet P_r in which we want to add hash link already has k links, we can use various policies to select another packet. This problem is similar to keeping the entries in hash tables balanced [Knu98]. We propose the following approaches:
  - Randomly select a new packet and add the hash to that packet if it contains less than k links, otherwise iterate up to a finite number of times.
  - Search a packet close to P_r with less than k links. This approach has the caveat that the resulting links might not be uniformly distributed.
  - Increase the hash link length and pick a new random packet. The drawback of this approach is that the average link length continuously increases as the stream progresses. To counteract this tendency, we can allocate $k + 1$ spaces for hash links in each packet. The advantage of such a scheme is that it gives us long-ranging hash links which increase the verification probability (but this technique would also increase the verification delay).

- Instead of picking k later packets which include the current packet's hash, we pick k previous packets of which we include the hash into the current packet. This method ensures that each packet has exactly k links, however, the number of outgoing hash links now varies (Poisson distribution with $\lambda = k$).

Super-Powered Signature Packets

So far we only considered signature packets that contain k hash links. To boost the verification probability, the sender creates special signature packets that contain a large number of hash links. If the sender uses 80-bit long hashes and signs the packets with a RSA signature (assuming a 1024-bit long modulus) that is 128 bytes long, a 512 byte long signature packet can carry 38 hash links, and a 1024 byte long signature packet can carry up to 89 hash links. Various approaches exist for choosing hash links in signature packets, for example an exponential distribution with a few long-ranging hash links may be particularly robust to long loss runs.

Denial-of-Service Considerations

The EMSS/MESS schemes rely on a traditional signature scheme, for example RSA [RSA78]. As we discuss in Section 2.3, verification of a traditional signature scheme is expensive. Even RSA digital signatures, well known for having relatively fast verification times, require 0.5 ms for verification on a 800 MHz Pentium III workstation (assuming a 1024-bit modulus). So a simple denial-of-service (DoS) attack is to flood receivers with bogus signature packets to exhaust the receiver's computation resources (see Section 2.3). To counteract this attack, we propose to use the multi-BiBa signature we propose in Section 4.5.2. The multi-BiBa signature scheme provides fast signature verification; the computation impact of bogus signatures is thus small (on the order of a few hash function computations).

5.4 HTSS

The Hash Tree Stream Signature (HTSS) is an efficient stream signature for the special case where the sender knows the entire stream content in advance. HTSS has an advantage over EMSS/MESS because it provides instant message authentication, and only requires a single signature. HTSS uses the Merkle hash tree construction [Mer80], which we discuss in Section 2.4.2.

For a stream with N messages, the sender creates a binary hash tree that has $d + 1$ levels ($L_0, \ldots, L_d$, with $d = \lceil \log_2(N) \rceil$) and that uses all N of

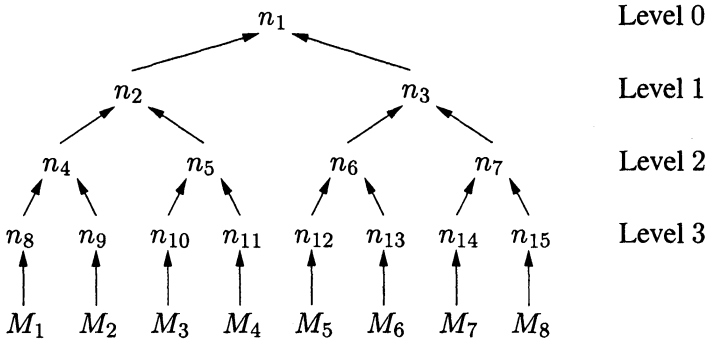


Figure 5.14. This figure shows a hash tree built over a stream of eight messages.

the messages $M_1, \ldots, M_N$. Figure 5.14 shows an example of a hash tree over eight messages. Each leaf node n_i ($2^d \leq i < 2^d + N$) corresponds to the hash of a message $n_i = H(M_{i-2^d+1})$. We derive each interior node n_i ($1 \leq i < 2^d$) from its two child nodes: $n_i = H(n_{2i} \mid\mid n_{2i+1})$. The signer signs the root node of the hash tree. One advantage to the hash tree is that the signature on the root also signs all the leaf nodes.

A receiver can verify the signature of a leaf node from the signed root node, and the nodes necessary to reconstruct the path from the leaf node up to the root. For a stream with N messages, each message contains $d = \lceil \log_2(N) \rceil$ additional hash tree nodes. For example, to verify the message M_3 in Figure 5.14, the receiver needs nodes n_{11}, n_4, and n_3 to reconstruct the path to the root:

$$n_1 = H(H(n_4 \mid\mid H(H(M_3) \mid\mid n_{11})) \mid\mid n_3)$$

For sufficient security, we need hashes that are at least 80 bits long. Assume we divide a 16 Mbyte long stream into 2^{15} messages of 512 bytes. Each packet would need to carry 15 nodes of 80-bit hashes of the hash tree, but those 150 bytes are a high communication overhead. In HTSS, we use a probabilistic approach to significantly reduce the number of supporting nodes sent with each message.

In HTSS, we take advantage of the following two observations:

- A node close to the root is sent more frequently than a node further away from the root. In fact, a node at level i of the tree is necessary to verify 2^{N-i} leaf nodes, assuming the basic hash tree approach. To reduce overhead, we stop sending a node after all receivers know it with high probability.

- Since the sender sends the messages sequentially, the receiver must already know a node on the path of a previous packet. For example, consider the stream $M_1, \ldots, M_8$ in Figure 5.14. After a receiving messages $M_1, \ldots, M_4$, the receiver already knows node n_2 (because it reconstructed the paths that all included n_2 to verify the root node). Thus, the sender would not need to send n_2 with messages $M_5, \ldots, M_8$.

  We now discuss this optimization more generally. Assume the tree is laid out as in Figure 5.14. Consider the subset of messages that have the internal node n_i on their path to the root. For the messages below the left child node n_{2i}, the sender adds the right child node n_{2i+1} necessary to reconstruct n_i. In the process of reconstructing the path to the root node, a receiver recomputes node n_{2i}. So for the messages below the right child node n_{2i+1}, the sender does not need to send the left child node n_{2i} any more, since the receivers still know it from before.

For the following discussion of HTSS, we make the following simplifying assumptions. First, we assume that all the receivers start receiving the stream from the beginning. If a receiver joins late, we assume the sender sends it all the necessary tree nodes by unicast. Second, we assume packet loss is independent, and we call q the probability that a packet arrives, and $p = 1 - q$ is the probability of packet loss.[8] Third, we assume flexible header size.[9] Finally, we assume that a receiver has the signature of the hash tree's root node. The sender can re-broadcast that signature packet until all receivers know the signature with overwhelming probability.

As in EMSS/MESS, we call $\mathcal{P}_v$ the probability that the receiver can verify the signature on a packet. A receiver needs to know d nodes to verify the root of the hash tree, so the probability that a receiver knows a specific node of the hash tree needs to be at least ϕ:

$$\begin{aligned} \mathcal{P}_v &\leq \phi^d \\ \phi &\geq \mathcal{P}_v{}^{\frac{1}{d}} \end{aligned}$$

[8] We do not discuss correlated packet loss in this section.

[9] For applications with a fixed header size, we could derive an amortized scheme that balances the load over multiple headers of equal size.

p	0.1	0.2	0.3	0.4	0.5	0.6	0.7
t	3	4	5	7	9	12	16
$\rho+1$	3	3	4	4	5	5	5

Table 5.2. This table shows the average number of nodes contained in a packet. We assume that $\mathcal{P}_v = 0.95$, $N = 2^{15}$ so $d = 15$, $\phi = {\mathcal{P}_v}^{(1/d)} = 0.997$.

To achieve a probability of at least ϕ that each receiver gets a node, we need to send the node at least t times:

$$\begin{aligned} 1 - p^t &\geq \phi \\ t \cdot \log(p) &\leq \log(1-\phi) \\ t &\geq \frac{\log(1-\phi)}{\log(p)} \\ t &= \left\lceil \frac{\log(1-\phi)}{\log(p)} \right\rceil \end{aligned}$$

Note that a node of level i needs to be sent at most 2^{N-i} times (it has 2^{N-i} leaf nodes), and for the levels far away from the root we may have $2^{N-i} < t$. This condition is true when $i > N - \log_2(t)$, so the nodes of the lowest $\rho = \lceil \log_2(t) \rceil$ levels need to be carried in each packet.

The nodes in the $N - \rho$ levels closest to the root only need to be sent t times. We now compute how many nodes we need to add to a packet on average. The topmost $N - \rho$ levels contain $2^{N-\rho+1} - 2$ nodes (excluding the root node). The sender only needs to send the right sibling nodes (see second observation above) while the left sibling node is on the path from the message leaf node to the root. The receiver will compute the left neighbor node of level i, $q \cdot \mathcal{P}_v \cdot 2^{N-i}$ times, so we assume that the receiver has received it and we do not need to re-send it (assuming $\mathcal{P}_v$ close to one). The total number of nodes we need to send for the topmost $N - \rho$ levels is $t \cdot (2^{N-\rho} - 1)$ nodes, and for the remaining ρ levels we need to send $\rho \cdot 2^N$ nodes. The average number of nodes we send per packet is $(\rho \cdot 2^N + t \cdot 2^{N-\rho})/2^N = \rho + t \cdot 2^{-\rho} = \rho + t \cdot 2^{\lceil \log_2(t) \rceil} \approx \rho + 1$. Table 5.2 shows the average number of nodes per packet for various packet loss probabilities.

5.4.1 HTSS Summary and Security Argument

1 **Setup.** The sender splits the data into N messages, $M_1, \ldots, M_N$, constructs a hash tree over the messages, and computes the root node n_1 of the tree. The depth of the hash tree is $d + 1$, where $d = \lceil \log_2(N) \rceil$. For each node n_i in the tree, the sender assigns a number η_i of how many times it needs to send it (to achieve a lower bound on the receiver's probability to verify a packet, as described in the text).

The sender S has a digital signature key pair, with the private key K_S^{-1} and the public key K_S. We assume a mechanism that allows a receiver R to learn the authenticated public key K_S, the signed root of the hash tree $\{n_1\}_{K_S^{-1}}$, and the depth d of the key tree.

2 **Sending packets.** To send message M_i, the sender creates a list of nodes it needs to add to the packet such that the receiver can verify the root n_1. For each node n_j, the sender adds that node to the packet if $\eta_j > 0$, otherwise it does not add the node. If the sender sends the node n_j, it appends (j, n_j) to the node list, and decrements η_j. The sender also decrements η for every node on the path from the leaf node of M_i up to the root n_1. The sender sends the following packet:

$$S \rightarrow * : M_i, (j, n_j), \ldots, (j', n_{j'})$$

3 **Verifying packets.** We assume that the receiver knows the authentic root node n_1. The receiver keeps a partial list of tree nodes to authenticate subsequent packets. When it receives an incoming packet $P_i = \langle M_i, (j, n_j), \ldots, (j', n_{j'}) \rangle$, it computes the path to the root, computes n'_r, and verifies that the recomputed root matches the stored root $n_r = n'_r$. Finally, the receiver stores the new tree nodes it received or recomputed, as it may need them to authenticate subsequent packets.

Protocol 5.2: HTSS protocol summary.

The box labeled Protocol 5.2 summarizes the HTSS protocol. For the HTSS protocol to be secure, we need the following requirements:

- The receivers need to receive an authentic public key K_S of the sender.
- The digital signature algorithm must be secure against existential forgery under a chosen message attack.
- The hash tree is secure against insertion of a bogus leaf node, or change of a current leaf node.

Chapter 6

ELK KEY DISTRIBUTION

Broadcast streams require authentication and confidentiality. Previous chapters present protocols for efficient authentication and signature protocols for broadcast streams. This chapter discusses confidentiality and access control.

Access control is a challenge for broadcasting. Most broadcast networks send data indiscriminately to all receivers, so senders need a mechanism for restricting access to legitimate receivers. A common solution is to encrypt the broadcast data and disclose the group key to legitimate receivers. How can the sender securely and efficiently establish a shared secret group key among legitimate receivers? How can the sender securely and efficiently retransmit the updated group key securely and reliably when receivers join or leave the group?

This chapter presents the Efficient Large-group Key-distribution protocol (ELK). ELK leverages the power of four independent protocols to achieve reliability and improve scalability:

- **Evolving tree (ET).** We propose that the key server periodically evolves the entire tree (by computing a one-way function on all keys in the key tree). This technique has the advantage that a member join event does not require any broadcast message.
- **Time-structured tree (TST).** If the key server places new members in specific parts of the tree, a member leave event that occurs at a predicted time does not require any broadcast message. Thus, only unpredicted leave events require a broadcast key update message.

A. Perrig et al., *Secure Broadcast Communication*

- **Entropy injection key update (EIKU).** We propose a new key update mechanism that has a number of advantages. First, it enables to trade off security with key update size, and second, it allows for small key updates that we call hints. These techniques substantially decrease the size of key updates.

- **Very important bits (VIB).** In protocols that use hierarchical key trees, some keys in the key update message are needed by most members to update their group key. We identify those keys and propose key update messages that only contain the VIBs, which can greatly reduce the communication overhead.

6.1 Introduction

Key change in a scalable, reliable, and efficient manner is particularly challenging [MSE02, HT00, HH99, WHA99, WGL98]. A solution must deal with arbitrary packet loss, including lost key update messages. The general approach is to use a *central key server* or *group controller* for key management. Key management is complicated by dynamic groups, where members may *join* and *leave* at any time, because members should only be able to decrypt the data while they are members of the group, and so the key server needs to update the group key after members join or leave the group.

We consider the following group operations or membership events. When a member wants to join the group it contacts the key server. We call this a *join event*. The new member may inform the key server for how long it wants to stay in the group. Based on the member information and membership statistics, the key server may predict the time a member wants to leave the group, which we call a *leave event*. If the member leaves at the predicted time, we call this a *predicted leave event*. If the member leaves the group earlier than the predicted leave time, or if the key server evicts the member from the group, we call this an *unpredicted leave event*. In ELK, we are not concerned about late leave events (i.e., members who stay longer in the group than predicted), because the key server can efficiently join these members again to the group.

Recent research makes substantial progress to address the challenge of scalable group key update. Current protocols feature efficient key update mechanisms to update the group key after members join or leave (we review related work in Section 8.6). However, most protocols require that the key server broadcasts a key update message after a member joins or leaves the group (although some protocols aggregate join and leave events). Briscoe's MARKS protocol is an exception, which gives receivers keys that are valid during a pre-

determined time period, so no key updates are necessary [Bri99]. MARKS, however, only allows for predicted leave events, and cannot update the group key after an unpredicted leave event (no member eviction, and members cannot leave the group earlier than their predicted leave time). Other protocols either broadcast a key update or send a unicast message to all receivers after a member join or leave event. The key server needs to send these key update messages reliably to all receivers; otherwise, if a receiver misses a key update message it will not be able to decrypt subsequent messages. Current key distribution protocols are efficient if the group key update messages are reliable. In practice, however, achieving reliability in broadcasts is a difficult problem, as we discuss in Section 1.1.

We believe that scalability and reliability of key update messages are closely related. Missing a key update is catastrophic for a receiver, since the receiver cannot decode subsequent broadcast information. Most key distribution proposals propose that a member who misses a key update, contacts the key server by unicast to request the missing key update. Such an approach does not scale to large dynamic groups because a lost key update message will result in a flood of members requesting a key update by unicast. Indeed, unreliable key updates inhibit the scalability of key distribution in large dynamic groups. As we discuss in Section 6.4, achieving reliability for key update messages is a challenge. ELK improves the reliability of key updates to improve scalability.

6.1.1 Requirements for Group Key Distribution

We consider dynamic groups where users can join or leave the group at any time. Below are the main security properties that are commonly used in secure group communication. Steiner, Tsudik, and Waidner give more formal definitions in their paper [STW00]. Here are intuitive definitions of these properties:

- *Group Key Secrecy* guarantees that it is computationally infeasible for a passive adversary (who eavesdrops all group communication) to discover any group key.
- *Forward Secrecy* guarantees that an adversary who knows a contiguous subset of old group keys cannot discover subsequent group keys (not to be confused with Perfect Forward Secrecy or PFS in key establishment protocols [DvOW92, BM92]). This property ensures that a member cannot learn future group keys after it leaves the group.
- *Backward Secrecy* guarantees that an adversary who knows a subset of group keys cannot discover previous group keys. This property ensures

that a new member who joins the group cannot learn any previous group key.

- *Key Independence* guarantees that an adversary who knows a proper subset of group keys $\hat{K} \subset \mathcal{K}$ cannot discover any other group key $\bar{K} \in (\mathcal{K} - \hat{K})$. Note that key independence implies forward and backward secrecy. In the most extreme case, key independence implies that an adversary who knows all but one group key, cannot derive the group key it misses.

To satisfy these properties, the key server has to update the group key after every individual join and leave event. To gain efficiency, we consider relaxing these stringent requirements. Related research finds that aggregating join or leave events can reduce join and leave rekey overhead [CEK+99, SKJ00, YLZL01, LYGL01]. When aggregating joins and leaves, the members are consequently not immediately joined to or removed from the group.

We first consider join events. To aggregate joining members, the key server evenly splits up the time into *join intervals*. The key server collects all joining members that arrive during one join interval and joins them all the end of the interval. Since members are not immediately joined to the group, this approach results in a *join latency*. Backward secrecy, however, is preserved. To keep the maximum join latency low, the join interval must be relatively small. The key server can further reduce the join overhead by increasing the join interval duration, and keeps the join latency low by relaxing backward secrecy and issuing a new member a key that was valid before it joined the group. We call this approach *join bonus*. Figure 6.1 depicts these concepts. The figure shows how the sender splits up the time into join intervals of uniform duration and changes the group key at each transition. The key server chooses the join bonus and join latency times, which the figure shows with a dotted line. For instance, member A joins within the join bonus time of key K_2, so A receives K_2 and can potentially decode group messages sent before it joined the group. Member B joins later, falls into the period of key K_3, and experiences a join latency since it cannot decrypt the broadcast until K_3 becomes active.

Similar to joins, aggregating leave events reduces the overhead of re-keying. In the simplest case, the key server aggregates leave events during a *leave interval* and processes them at the end of each interval. The member experiences a *leave bonus* since it can decode group data after its leave request. This approach relaxes forward secrecy. In case of predicted leave events, the key server may schedule a member leave earlier, which we call *leave bias*. Figure 6.2 depicts these concepts. In this example, the key server aggregates leave

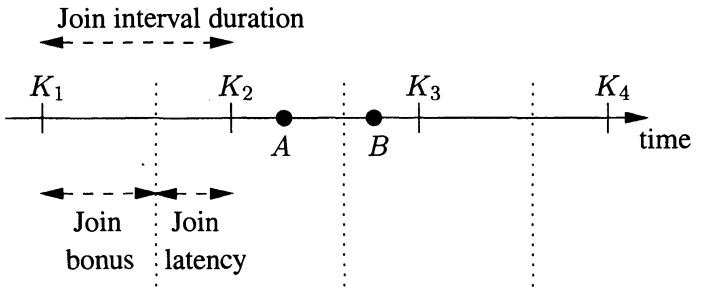


Figure 6.1. This figure depicts the relationship between join intervals, join bonus, and join latency. The dotted line shows the threshold between join bonus and latency. Member A gets a join bonus and receives K_2, member B experiences a join latency and gets K_3.

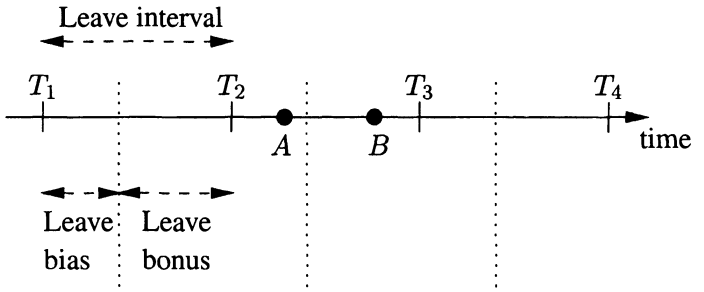


Figure 6.2. This figure depicts the relationship between leave intervals, leave bonus, and leave bias. The dotted line shows the partition between leave bonus and bias. Member A has a leave bias, and member B experiences a leave bonus.

events during fixed-size leave intervals, and sends a key update at the end of the leave interval at time T_i. The dashed line shows the threshold between the leave bias and leave bonus. Member A's predicted leave time falls into a leave bias interval, so the key server evicts A at time T_2. Member B leaves later and gets a leave bonus, since it remains in the group until time T_3.

We define the *granularity of membership events* as the duration of join or leave intervals. Short join or leave intervals result in a *fine-grained membership processing*, and we say that long intervals give us a *coarse granularity of processing membership events.*

6.2 Review of the LKH Key Distribution Protocol

We first discuss the general key distribution setting. We then briefly describe the *Logical Key Hierarchy (LKH)* protocol [HH99, WHA99, WGL97, WGL98]. Finally, we discuss two extensions to make LKH more efficient: LKH+ and LKH++.

We consider broadcast key distribution in the following setting. We assume a central key distribution server that can authenticate and authorize individual receivers. The model is that a new receiver contacts the key server by unicast and requests the group key. The key server authenticates the receiver with a standard authentication protocol and sets up a secure channel (offering confidentiality, integrity, authenticity). If the receiver is authorized, the key server sends group keying information to the receiver.

The sender encrypts the broadcast information with the group key to achieve confidentiality and to restrict access to legitimate receivers. To ensure forward and backward secrecy after receivers join or leave events, the key server updates the group key, which sometimes involves broadcasting encrypted key update messages.

In LKH, a *central key server* maintains a key tree to efficiently update the group key after join or leave events. Figure 6.3 shows a sample key tree. Each node in the tree represents a cryptographic symmetric key. The key distribution center associates each group member with one leaf node of the tree and the following invariant will always hold: each group member knows all the keys from its leaf node up to the root node, but no other key in the key tree. We call the set of keys that a member knows the *key path*. Since all members know the key at the root node, that key is used as the group key, which we denote with K_G (or K_1 in Figure 6.3). For illustration, the key path of member M_2 in Figure 6.3 consists of the keys K_5, K_2, and K_1. When a member joins the group, the key server sends it all the keys on the key path over a secure channel. When a member leaves the group, the key server needs to update all the keys that the member knows (i.e., the keys on the key path). The main reason for using such a key tree is to efficiently update the group key when a member joins or leaves the group.

When a member joins the group, the key server authenticates the member and assigns it to a leaf node of the key tree. The key server sends all the keys on the key path to the member. To preserve backward secrecy all the keys that the new member receives need to be computationally independent from any previous keys (the new member should not be able to decrypt traffic that was

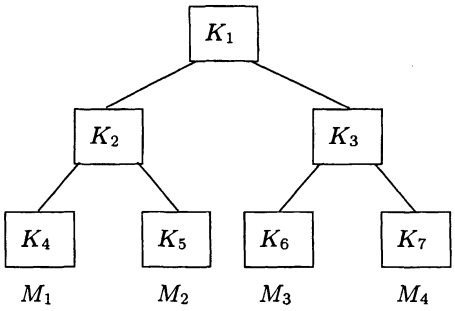


Figure 6.3. Sample hierarchical key tree. Each node of the tree corresponds to a symmetric cryptographic key. The members are placed at the leaf nodes of the tree. Each member knows all the keys on the path from its leaf node up to the root of the tree. We call this path the *key path*. Since all members know the key K_1 associated with the root of the tree, K_1 is the group key.

sent before it joined the group). The key server thus replaces all keys on the new member's key path with fresh, random keys and sends each of these new keys to the group on a "need to know" basis. We illustrate this protocol with an example. Assume the setting depicted in Figure 6.3 and assume that a new member M_4 joins the group. Assuming that the last leaf node (corresponding to the key K_7) of the tree is empty, the key server places the new member M_4 at that leaf, chooses new the keys on M_4's key path and sends K_7', K_3', and K_1' to M_4 over a secure channel. To update the key paths of the previous members, the server broadcasts the following key update message to the group:

$$\{K_3'\}_{K_6}, \{K_1'\}_{K_3'}, \{K_1'\}_{K_2}$$

Member M_3 needs to update keys K_3 and K_1 on its key path. Since M_3 knows K_6, it can decrypt the first part of the key update message and recover K_3', and as soon as it knows K_3' it can decrypt the new group key K_1'. Members M_1 and M_2 both know K_2, so they can decrypt the new group key K_1' from the final part of the key update message. Below we show how to improve on this join protocol so no broadcast message is necessary.

The member leave protocol is similar. To preserve forward secrecy the key server updates all the keys on the key path of a leaving member. Assume that member M_2 leaves the group that Figure 6.3 shows. The key server picks new, random keys for all keys on M_2's key path: K_5', K_2', and K_1'. The key server

broadcasts the following key update message:

$$\{K_2'\}_{K_4}, \{K_2'\}_{K_5'}, \{K_1'\}_{K_2'}, \{K_1'\}_{K_3}$$

Member M_1 needs the new keys K_2' and K_1'. Since it knows K_4, it can decrypt K_2' from the key update message. Since M_2 now knows K_2', it can decrypt the new group key K_1'. Members M_3 and M_4 only need the new group key; since they both know K_3 they can readily decrypt it from the key update message.

Note that the key update after a member leave event only requires a message of length $2 \cdot k \cdot \log(N)$ bits (where k is the length of a key, and N is the number of members). The main contribution of LKH is that it provides secure key update that only requires a broadcast message size that is logarithmic in the number of group members.

Note that an efficient and secure (providing backward secrecy) key update after a member join event is easy to achieve. For example, a single group key K_G achieves backward secrecy and efficient join: after a new member joins, the key server picks a new group key K_G', and broadcasts the new group key encrypted with the old group key to the group $\{K_G'\}_{K_G}$. Unfortunately, this approach does not provide for an efficient mechanism for member leave events. This approach is adopted by the GKMP protocol [HM97a, HM97b], which distributes group key updates through unicast messages to all members. GKMP is a fine protocol for small groups with static membership, but it does not scale to large groups.

6.2.1 Extension I: Efficient Join (LKH+)

A simple optimization can reduce the size of the key update message after a join event to $O(\log(N))$ bits (assuming a balanced tree). The join operation needs to satisfy backward secrecy, so, a joining member cannot learn any previous group keys. In standard LKH, the key server chooses fresh, random keys on the new member's key path. If each key is k bits long, and the key path contains d keys, the key update message is $2 \cdot k \cdot (d-1)$ bits long.

In LKH+ the key server computes a one-way function on each key on the new member's key path. The key update message only needs to encode the new member's position within the key tree, which only requires d bits. From the new member's position, all other members can infer the new member's key path, and can independently update (by computing the one-way function) the keys on their own key path that are common with the with new member's key path. This approach achieves backward secrecy because the new member cannot invert the one-way function to derive previous group keys. This approach

is attributed to Radia Perlman, who suggested it at a LKH presentation during an IETF working group meeting. Independently, the Versakey framework also proposes this approach [CWS+99].

6.2.2 Extension II: Efficient Leave (LKH++)

Canetti et al. describe a method to reduce the size of a key update message (after a member leave event) by a factor of two [CGI+99]. We describe a simplified version of their approach, more details are in their paper [CGI+99]. They propose to use a pseudo-random generator G (PRG, see Section 2.2.3) as a one-way function to derive new keys on the key path. The idea is that the key path resembles a one-way chain starting at the leaf node and ending at the root. (We describe one-way chains in detail in Section 2.4.1.) The parent key K_p is thus directly derived from the child key K_c on the key path: $K_p = G(K_c)$. Given any key on the key chain, anybody can derive the remaining keys up to the root key. We call this approach LKH++, and it results in key update messages after a leave event that are half the size of standard LKH. (We assume that LKH++ uses the efficient member join mechanism of LKH+.)

We illustrate this approach with an example. Consider the example that Figure 6.3 shows. Let us assume that the key server expels member M_2, so it needs to update keys K_5, K_2, and K_1. The key server picks a new key for the leaf node, say K_5', and the new parent key becomes $K_2' = G(K_5')$. Consequently, the new group key is $K_1' = G(K_2)$. With standard LKH, the key update would be $\{K_2'\}_{K_4}, \{K_2'\}_{K_5'}, \{K_1'\}_{K_2'}, \{K_1'\}_{K_3}$. In contrast, the LKH++ key update is $\{K_2'\}_{K_4}, \{K_1'\}_{K_3}$. So the component $\{K_1'\}_{K_2'}$ of the LKH key update is not necessary any more, since $K_1' = G(K_2')$.

6.3 Review of the OFT Key Distribution Protocol

McGrew and Sherman present the One-way Function Tree (OFT) protocol [MS98]. OFT uses a binary key tree as LKH. The new idea in OFT is that the parent node in the tree is directly derived from the two child nodes. Consider a parent node K_{parent} and the two child nodes K_{left} and K_{right}. The following relation holds for all nodes in the key tree (except the leaf nodes) $K_{parent} = f(g(K_{left}), g(K_{right}))$. They propose to use a one-way function (see Section 2.2.2) for the function g to *blind* the node key, and the exclusive-or operation (denoted as $\oplus$) for the combining function f. The members in both subtrees below K_{left} and K_{right} receive the blinded key of the other side, which allows them to compute the parent key. For instance, the members in the

left hand subtree know K_{left} and they can compute $g(K_{left})$, but they need the blinded neighbor key $g(K_{right})$ to compute $K_{parent} = g(K_{left}) \oplus g(K_{right})$.

Consider the key tree in Figure 6.3. An OFT key tree is the same, but the invariant must hold for each node in the tree that is not a leaf node: $K_{parent} = g(K_{left}) \oplus g(K_{right})$. The root key depends on all the leaf keys:

$$K_1 = g(g(K_4) \oplus g(K_5)) \oplus g(g(K_6) \oplus g(K_7))$$

As in LKH, each group member knows all the keys from its leaf node up to the root, but in addition, each member knows all the blinded keys of the child nodes of each key of the key path. To join a new member to the group, the key server assigns a new member to a leaf node, chooses a new key for that leaf node, updates the key tree, sends out the leaf key and blinded keys to the new member, and broadcasts the new blinded keys in encrypted form to the group (such that each previous member can update their key path and group key). We give a practical example to illustrate the join protocol. Consider the case of member M_2 joining the group in Figure 6.3. The key server chooses a new key K_5' for M_2's leaf node. Consequently, the keys K_2 and K_1 change, as well as the blinded versions of these keys $g(K_5)$ and $g(K_2)$ (the blinded group key $g(K_1)$ is not needed, since the group key has no parent). Member M_1 needs the blinded key $g(K_5')$ to compute the new K_2. Members M_3 and M_4 need the blinded key $g(K_2)$ to compute the new group key K_1'. The key server broadcasts the following key update message allowing all previous members to update their key paths: $\{g(K_2')\}_{K_3}, \{g(K_5')\}_{K_4}$. To derive $K_2' = g(K_4) \oplus g(K_5')$, member M_1 needs the blinded key $g(K_5')$, which it can decrypt from the key update message since it knows K_4. Member M_1 already knows the blinded key $g(K_3)$, so it can directly derive the new group key $K_1' = g(K_2') \oplus g(K_3)$. Members M_3 and M_4 only need to update the group key, so they need the new blinded key $g(K_2')$ which they extract from the key update message since they know K_3. They can both compute the new group key $K_1' = g(K_2') \oplus g(K_3)$.

The key update for a member leave event is similar: the key server can simply pick a new key for the leaf node of the leaving member, recompute the key tree, and distribute the key update message.[1]

[1]Note that it would not be secure to simply delete leaf and parent keys (of the leaving member) from the key tree, and to promote the neighbor node (of the leaving member) to the parent node. We leave the attack as a simple exercise. To make this approach secure, the neighbor node (of the leaving member) has to change to a new value, unpredictable to the leaving member.

The key update message of OFT is about half as long as it is in LKH. In standard LKH, the key update message contains each changing key encrypted separately with the two child keys. In OFT, the key update message only contains the blinded keys that changed from the previous key tree, encrypted with the neighboring key. In comparison to LKH++, the key update message after a member leave of OFT has the same size, but the key update after a member join of OFT is k times longer (where k is the length of a key).

6.4 Reliability for Key Update Messages

When members join or leave a group, the key server updates the group key and broadcasts a key update message to the group. If a group member does not receive the key update message, it will not be able to decrypt the subsequent messages encrypted with the new group key. Reliable key update is thus a necessary component of any key distribution protocol. In fact, key distribution requires a strong notion of reliability that we call *timely reliability*. Since the key server continuously broadcasts a stream of messages encrypted under the current group key, reliability is only useful in this context if the receiver instantly receives the key update messages (otherwise it cannot decrypt the stream and it will send a key update request by unicast to the key server).

With the exception of the recently proposed Keystone protocol by Wong and Lam [WL00, YLZL01, LYGL01], previous systems addressed the problem of lost key updates only marginally. Previous protocols assume to recover from lost key updates through the following mechanisms:

1 Allowing members to request a key update by unicast from the key server. Clearly the naive unicast recovery mechanism does not scale, although it can be used in conjunction with other techniques as a fallback recovery mechanism. In fact, both ELK and the Keystone protocol use unicast as a fallback mechanism. We sketch such a recovery protocol in Section 6.7.6.

2 Replicating key update packets (multi-send). Although replication is a powerful method to achieve reliability, it is well known that packet loss in the Internet is correlated [Pax99]. This implies that key update packets sent in close succession risk loss if the first one is lost. A strategy that separates redundant packets would cause the client to wait for replicated key update messages when it receives data that it cannot decrypt.

3 Using reliable multicast protocols, such as the SRM protocol [FJM+95] or the STORM protocol [XMZY97], may achieve reliable delivery of key updates. These protocols add substantial complexity and might not scale

to large audiences. Furthermore, these systems are not designed to be robust in an adversarial environment, and hence opportunities for denial-of-service attacks exist. Similarly, a reliable group communication toolkit, such as those used in small-group key agreement protocols, such as TOTEM [MMSA+96], or HORUS [vRBM96] are expensive and do not scale to large groups.

4 Using Forward-Error Correction (FEC) or error-correcting codes, as proposed by Lam et al. [WL00, YLZL01, LYGL01], may achieve reliability for key update packets. More specifically, they suggest using algorithms, such as Rabin's IDA [Rab90], Reed-Solomon [RS60] codes, or Digital Fountain codes [BLMR98], to split up a key update packet into n packets (with some additional redundancy) — and the receiver can reconstruct the key update from any m packets. However, since packet loss is correlated, packets that are sent in close succession may all be dropped [Pax99]. Lam et al. assume statistically independent loss, which does not cover correlated packet loss (e.g., due to temporary congestion) [WL00]. Despite the FEC, correlated loss can wipe out a key update message, and we still need a recovery mechanism for that case as well.

In this work, we use a combination of new mechanisms to achieve reliable key updates. Our work is motivated by the following observations:

1 Member joins are reliable if no broadcast is necessary. Our **evolving tree (ET)** protocol for member join events does not require any broadcast messages. However, if a large number of members join concurrently, the key server may need to broadcast a message of length $O(\log(N))$ bits to encode the location of the joining node, where N is the number of group members (assuming a balanced tree).

2 If a member leaves at its predicted time, our **time-structured tree (TST)** protocol will not require any key update message. However, if a member leaves at an unpredicted time or if the key server evicts it, the key server broadcasts a rekey message.

3 Our new **entropy injection key update (EIKU)** protocol features small key update messages. In addition, it allows for *hints* which further reduce the size of key updates, but require a higher receiver computation overhead. Most key management protocols separate key update messages and encrypted data packets. The receiver must receive both of them to read en-

crypted data. Since the hints are very small, we propose that the sender add the key update directly to data packets.

4 Not all keys in a key update message are equally important. We identify the **very important bits (VIB)**, which are the keys that help the majority group members to recompute the group key. To achieve reliability, the sender may repeatedly send the same key update message. We propose that the sender re-distributes only the VIBs in redundant key update messages, such that most members can recover from a lost key update message, and only a fraction of receivers need to fall back on requesting a key update by unicast.

6.5 Four Basic Techniques

This section presents four orthogonal techniques to achieve higher reliability and scalability for broadcast key distribution. Section 6.6 further discusses the advantages and tradeoffs of each technique in more detail.

6.5.1 Evolving Tree (ET) Protocol

This section presents the Evolving Tree (ET) protocol. In ET, new members can join the group securely (preserving backward secrecy) without the key server broadcasting a key update message.

In a member join event, the key server assigns the new member to a node in the key tree and the new member receives all the keys on the key path (see Section 6.2). To preserve backward secrecy, all the keys that the new member receives must be computationally independent of previous group keys. LKH+ uses a one-way function to update all keys on the key path of the new member; the key server only needs to broadcast the joining location of the new member and all previous members can update their key paths independently. This key update message is only $O(\log(N))$ bits long (assuming a balanced key tree), but that message might still get lost. To prevent sending a broadcast message, we compute a one-way function on all keys of the key tree in each join interval (see Section 6.1.1 for a discussion of join intervals, join latency, and join bonus). We harness the (typically abundant) computation resources of the key server to trade off computation for a lower communication overhead. The new member receives the updated keys on its key path, and because of the one-way function it cannot derive any previous group keys (thus this approach achieves backward secrecy).

The box labeled Protocol 6.1 describes a summary of the ET protocol. The details are in Appendix 6.8.2.

The key server updates the entire key tree in each join interval with a function F. To achieve backward secrecy, F needs to be a one-way function, so the new member cannot find past group keys. Since each node in the key tree is a key for a cryptographic operation, these keys also need to be pseudo-random (see Section 2.2.5). So we choose F as a PRF with the properties described in Section 2.2.5.
Consider a new member M joining the group. The key server decides to insert M at an empty leaf node and sends all the keys on the key path to M.
The protocol looks as follows (assuming that M and key server S already share a secret key K_{MS}:

$$M \rightarrow S : \text{Join request}$$
$$S \rightarrow M : \{\text{Key path, join interval, leave interval}\}_{K_{MS}}$$

Protocol 6.1: Summary of the ET protocol.

Multiple member join events work equally well. Several possibilities exist to deal with them. Here is one way. If the members are placed into empty leaf nodes no overhead is generated besides registering new members with the key server. If no empty leaf nodes exist, the key server can first generate a smaller key tree with the new members, and join that tree to one node of the current group key tree. In this case the key server only needs to communicate the location of a single node to the members that live below the joining node.

Since the computation of the PRF is efficient, the computational overhead of the receiver is negligible. The computation overhead of the server depends on the join interval and the size of the key tree. However, a number of approaches can reduce the computation overhead. First, the server can pre-compute the future keys of the tree in a low-priority background process. In case the server needs a key that the background process did not update yet, it can quickly compute the keys it needs (recall from Section 2.3 that a workstation today can perform around half a million PRF computations per second with HMAC-MD5). Next, the server does not need to update the leaf keys of the key tree — since only one member knows the leaf key (no joining member ever receives it). Finally, the key server does not need to evolve the entire key tree after each

join interval. An alternative policy would be to evolve only part of the key tree, and to join new members only to that part of the tree.

6.5.2 The Time-Structured Tree (TST) Protocol

This section presents the Time-Structured Tree (TST) protocol. The novelty of TST is that no broadcast message is necessary on a predicted member leave event. Despite the missing broadcast message, TST still provides forward security.

Recall that a predicted member leave event happens when a member leaves the group when it predicted to leave, (or when the key server correctly predicted the leave time). An unpredicted member leave event happens when a member leaves early or is evicted from the group. A member who leaves later than predicted is not a problem, since it can join the group again after its previous membership expires. Since we only need to send a group key update message to the group after an unpredicted member leave event, TST can substantially enhance scalability. We achieve this property through a special arrangement of members in the key tree.

TST works as follows. Similar to the ET protocol in the previous section, TST defines leave intervals (time intervals with uniform length). Section 6.1.1 describes leave intervals, leave bias, and leave bonus.

We give a special meaning to some subtrees of the key tree. As Figure 6.4 shows, the sender splits the time into leave intervals t_i. When a member expects to leave in time period t_i, the key server issues it a key from subtree T_i.

The key server updates the key tree and group key as follows. We consider again the example that Figure 6.4 shows. In time period t_i, the key K_i is the group key. When time period t_{i+1} starts, the key server crops off the entire subtree T_i and the key K_i. The key K_{i+1} becomes the new group key. No member in subtree T_i knows K_{i+1}, so all these members simultaneously leave the group. This technique does not require any broadcast message to the group and is therefore perfectly robust to packet loss. We describe the case of an unpredicted member leave event or member evict in Section 6.8.3.3.

We review related work by Poovendran and Baras [PB99], and Selcuk, McCubbin, and Sidhu [SMS00] in Section 8.6.

6.5.3 Entropy Injection Key Update (EIKU)

This section introduces our Entropy Injection Key Update (EIKU) mechanism. EIKU assumes a key tree, similar to LKH and OFT (see Sections 6.2 and

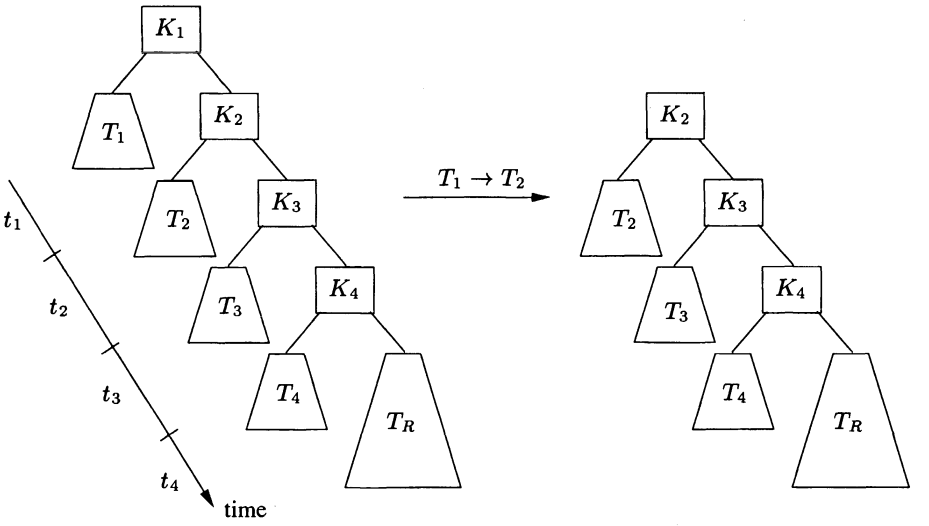


Figure 6.4. This figure shows the evolution of the time-structured tree (TST) protocol. Each subtree T_1, T_2, T_3, T_4 and T_R stands for a regular key tree as Figure 6.3 shows. Each subtree T_i corresponds to a time interval t_i. As time progresses, the key server removes the subtree T_i and current root node K_i of after time period t_i is over. With this mechanism, all the members in subtree T_i automatically leave the group at the end of time period t_i. The figure shows the change of the key tree as time passes from period t_1 to period t_2.

6.3). We use EIKU to update the key tree after an unexpected leave event. The key update is similar in nature to the OFT protocol (see Section 6.3), since both child keys contribute (i.e., inject entropy) to the parent key. In contrast to OFT, the child keys only add entropy to the parent key, and no global invariant needs to hold in the key tree. This enables EIKU and ET to interoperate (OFT can not use the ET protocol, because ET would violate the node invariant). This section also highlights other advantages and unique features of EIKU: short key updates, and compressed key updates (hints). Although the key server could use EIKU to update the key tree after a member join event, we assume that the key server uses the ET protocol to handle join and so we will not discuss member join events here. We now overview the EIKU protocol and introduce the hint mechanism. Appendix 6.8.3 describes EIKU in detail.

To update a key tree after a member leave event, the key server needs to change all the keys that the leaving member knows, i.e., all the keys on the member's key path.

We need a new key update protocol for two reasons. First, OFT does not work with the ET protocol. Second, we would like to get a protocol that allows us to shorten the key update size without exposing the protocol to dictionary attacks. Large groups with thousands of receivers can afford lower security, so ideally we could shorten the key length to get smaller key update messages. To the best of our knowledge, previous protocols become vulnerable to pre-computation or dictionary attacks when we shorten the key length.

We now discuss the EIKU key update protocol. (Note that we describe a simplified version of EIKU; Section 6.8.3 contains the detailed description.) In contrast to OFT, the parent key is not directly derived from the two child keys, but in a key update, both child keys contribute entropy to the parent key. Consider the parent key K_P with the left and right child keys K_L and K_R. For a key update, both child keys add n bits of entropy to K_P, which we call *contributions*. We use the group key K_G and the PRF F to extract the contributions from the child keys: $C_L = F_{K_L}(K_G)$ is the left contribution, and $C_R = F_{K_R}(K_G)$ is the right contribution. We use the combined contributions $C_{LR} = C_L \mid\mid C_R$ to derive the new parent key: $K'_P = F'_{C_{LR}}(K_P)$, where F' is another PRF with appropriate key size, input and output lengths. Note that this procedure adds $2n$ bits of entropy to the parent key.

The key update is similar to OFT. The members under the left subtree know K_G, K_P, K_L, and C_L, but they need C_R. Similarly, the members under the right hand subtree need C_L. So the key update message is $2n$ bits long (assuming an encryption function that does not expand the input size) contains $\{K_L\}_{C_R}, \{K_R\}_{C_L}$.

Note how this approach allows us to adapt the key update size to the desired security margin. If we want that an attacker performs at least $2^{n'}$ operations to break a key, we choose $n = \lceil n' \rceil$. The keys in the key tree can be much larger without increasing the size of the key update message. Intuitively, this mechanism prevents dictionary and pre-computation attacks. A more detailed study of the security of EIKU is in [PST01].

The EIKU key update enables another mechanism to reduce the size of key updates even further. Suppose that a receiver can perform 2^n operations, but an attacker cannot perform 2^{2n} operations.[2] We present the *hint* protocol, which halves the size of the key update and lets both receivers brute-force compute

[2]We generalize our scheme in our detailed discussion in Section 6.8.3 to an attacker that is arbitrarily powerful. The general result is that when a receiver can perform 2^n operations, we can make the key updates n bits smaller.

the new group key. Here is the intuition for hints. A receiver in a subtree knows half the contribution, but not the remaining n bits. So it could brute-force compute the 2^n candidates for the parent key and it knows that the correct new parent key is one of them. The hint is a checksum on the new parent key that allows the receiver to find the correct candidate key. We use another PRF F'' for the key checksum, which outputs n bits. So the hint H for the new key K'_P is $H = F''_{K'_P}(0)$.

We illustrate the key reconstruction from the hint H with an example. Consider a member under the left subtree of K_P. It knows C_L, but needs to try the 2^n variants of C_R. For each guess C'_R it computes the candidate key $\tilde{K}'_P$: $C'_{LR} = C_L \mid\mid C'_R$, $\tilde{K}'_P = F'_{C'_{LR}}(K_G)$. For each candidate key it verifies if the hint is equal to the checksum $H = F''_{\tilde{K}'_P}(0)$. If they match, it accepts the candidate key.[3] Note that the hint key update is only half the size of the original key update, but requires receiver computation.

The hint discloses information about the secret key, which may considerably reduce the group key security in some cases. We thus discourage using hints for high-security applications. However, in most large-scale key distribution applications, thousands of users share the group key on (typically) untrusted hardware, so the benefits of the hint mechanism outweigh the reduced security.

In the case of an unpredicted leave event, the key server can either erase the leaf node or select a new leaf key. The change in the key tree causes all the keys that the leaving member knows to change. Consider the example in Figure 6.5. Member M_6 unexpectedly leaves the group. The key server chooses a new key K_{13} and performs a key update. Consequently, the keys K_6, K_3, and K_1 are updated. The key update message is

$$\{C_{K_2}\}_{K'_3}, \{C_{K'_3}\}_{K_2}, \{C_{K'_6}\}_{K_7}, \{C_{K_7}\}_{K'_6}, \{C_{K_{12}}\}_{K'_{13}}, \{C_{K'_{13}}\}_{K_{12}}$$

A shorter hint message has the following components:

$$F''_{K'_1}(0), F''_{K'_3}(0), F''_{K'_6}(0)$$

6.5.4 Very-Important Bits (VIB)

To achieve reliability, the sender may repeatedly send the same key update message. Can we find a strategy for key update messages that lowers the communication overhead and improves reliability?

[3] To prevent false positive keys, the hint needs to be larger than n bits. We address these concerns in more detail in Section 6.8.3.2.

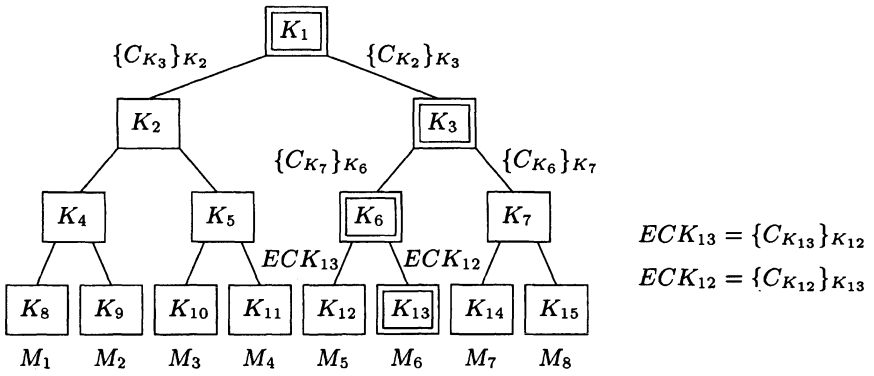


Figure 6.5. This figure depicts the key update message after member M_6 leaves the group early. The keys in the double-outlined boxes show the updated keys (member M_6's key path). We denote the entropy contribution of a key K_i with the shortcut C_{K_i}; the full entropy contribution also depends on the parent K_p, so C_{K_i} is a shortcut for $C_{K_i} = FK_iK_p$. The full key update message consists of all child entropy contributions, encrypted with the neighbor key (e.g., $\{C_{K_2}\}_{K_3}$). The annotated edges show the members of a subtree that need a part of the key update message.

We observe that different keys of the key update message help a different number of members to update their key paths. We illustrate this with an example. Consider the key tree that Figure 6.5 shows. After member M_6 leaves the group, the key server updates all the keys on M_6's key path. The key update message is:

$$\{C_{K'_3}\}_{K_2}, \{C_{K_2}\}_{K'_3}, \{C_{K_7}\}_{K'_6}, \{C_{K'_6}\}_{K_7}, \{C_{K'_{13}}\}_{K_{12}}, \{C_{K_{12}}\}_{K'_{13}}$$

Note that different parts of the key update message are used by a different number of receivers. For example, four members use the part $\{C_{K'_3}\}_{K_2}$ to update the root key, three members use $\{C_{K_2}\}_{K'_3}$ to update the root key, two members use $\{C_{K'_6}\}_{K_7}$ to update K'_3, etc. Thus, each key is useful to a certain number of members. This motivates our definition of the *bit value* of a part of the key update message. For a given part of the key update message, say $\{C_{K_i}\}_{K_j}$, $\#\mathsf{Members}(\{C_{K_i}\}_{K_j})$ defines the number of members that use that part of the key update message, and $|\{C_{K_i}\}_{K_j}|$ gives the size in number of bits. So the bit value of a part of the key update message $\{C_{K_i}\}_{K_j}$ is

$$\mathsf{BV}(\{C_{K_i}\}_{K_j}) = \frac{\#\mathsf{Members}(\{C_{K_i}\}_{K_j})}{|\{C_{K_i}\}_{K_j}|}$$

The key server computes the bit value of each part of the key update message. So to reduce the size for redundant key updates, the key server only adds the parts with the highest bit value — the *Very Important Bits (VIB)*.

6.6 ELK: Efficient Large-Group Key Distribution

ELK is an acronym for the Efficient Large-group Key distribution protocol. ELK uses all four mechanisms that Section 6.5 describes. ELK has many tuneable parameters and enables many tradeoffs. We now present the components of ELK and discuss the tradeoffs and parameters in more detail. Our main objective for ELK is maximum scalability. For a given bandwidth overhead for the key distribution protocol (unicast and broadcast overheads), we aim for a protocol that achieves high reliability.

We consider the following resources in ELK (the total number of receivers is N).

- **Receiver storage**. Each receiver stores a number of member-specific keys and one group key. Since the number of keys to store is usually small ($O(\log(N))$), receiver storage is not an issue.

- **Key server storage**. The key server stores all member-specific keys. In current schemes the number of member-specific keys is about $2N$. For groups with millions of receivers, the key server storage can be on the order of multiple megabytes. Researchers have investigated how to reduce the server storage overhead [CMN99, LPB01, Poo99]. In this work we assume that the key server has sufficient storage.

- **Receiver and key server computation**. In most previous key distribution schemes, key server and receiver computation are not a limiting factor (an exception are the small group key agreement schemes that rely on asymmetric cryptographic primitives, see Section 8.5). The steady increase in computation power opens opportunities to harness abundant computation resources to minimize communication. Indeed, our ET protocol leverages server computation such that join events do not require a broadcast message, and the hint mechanism (in conjunction with EIKU) utilizes receiver computation resources to minimize the size of key update messages.

- **Bandwidth** is the most constrained resource. In particular, key update information broadcast to all receivers must be minimal. Since we assume abundant storage and computation resources, our goal is to trade off com-

putation or storage to lower communication cost. We put a premium on broadcasts, however we also try to limit unicast messages.

- **Trusted hardware**. We are interested in moderate security. We consider the case when many receivers use commodity hardware, e.g., PCs without tamper-resistant hardware. Note that the absence of tamper-resistant hardware limits the ultimate security of keys [YT95, Yee94].

ELK is based on hierarchical key trees, similar to the OFT or I KH protocols (see Sections 6.2 and 6.3). In dynamic groups, members join and leave all the time. It would be a serious scalability problem to broadcast a key update message after each membership change. ELK uses the Evolving Tree (ET) protocol (see Section 6.5.1), to trade off increased key server computation for a lower broadcast communication overhead.

We believe that the key server computation overhead is small, compared to the benefits ET provides. The pseudo-random function we use to update the key tree is very efficient, e.g., if we use HMAC-MD5 for this purpose, a 800 MHz Pentium III Linux workstation can update 400000 keys per second (see Section 2.3). In addition, the key server does not need to update all keys simultaneously, but can use a lazy evaluation mechanism and only update the keys on a key path when necessary. Another approach is to use a low-priority background process to update the key tree. Another ET optimization is to update only part of the key tree, so the key server would join new members only to that part of the key tree. ET requires a modest amount of computation resources, but provides substantial reliability benefits because fewer receivers contact the key server for key updates.

The key server can evolve the tree either periodically, or after some join events. In ELK we propose to update the key tree periodically, so the key server does not need to broadcast any information to indicate that the key tree evolved (even though this would only require a small number of bits).

To reduce the overhead even further, the key server may give the members a limited join bonus time. So when a receiver joins, the key server issues it a key path in the current tree, which might also allow it to decode messages during a brief time period before it joined the group, thereby violating backward secrecy to achieve increased receiver convenience.

In many applications, the key server can predict the leaving time of members. For example in pay-per view settings, the receiver pays to receive content during a specific time period. By using the time-structured tree (TST) protocol (see Section 6.5.2), the key server does not need to broadcast any key update

message when members leave at the predicted time. The tradeoff, however, is that TST has an unbalanced key tree, which results in larger join and key update messages.

Similar to ET, TST can remove an expired subtree either periodically or at varying times. For simplicity we choose the periodic approach.

The time between tree evolutions and subtree expirations are other parameters for ET and TST. The tradeoff is between controlling the granularity of membership and overhead. For instance, if the key server chooses to use a one second granularity for TST, it would need to send a joining receiver over 3600 keys if that receiver wants to stay in the group for one hour.

Join events and predicted leave events do not require a broadcast message, only early leaves or evictions require a broadcast message. To prevent the high penalty of a broadcast key update, the key server can predict an early leave time, and if the member stays in the group, the key server re-joins the member by sending it a unicast message. (This approach also has the advantage of a shorter unicast join message, and a shorter key update, in case the member decides to leave even earlier.) However, members joining for short time intervals will result in a high unicast overhead, as in the GKMP protocol which we review in Section 8.6.

Based on the probability of packet loss, the key server can estimate the overhead of broadcasting a key update (which would include the overhead of replying to unicast key update requests). With the current member dynamics, the server can find the optimal strategy for placing joining members in earlier trees.

To manage the key tree, the key server uses the entropy injection key update (EIKU) protocol (see Section 6.5.3). EIKU enables the tradeoff between security and key size. Since the group key is shared by potentially hundreds of thousands of members, the secrecy of the group key is weakened. Consequently, the key server picks a small key size to account for the lower security margin. A reasonable key size for large groups may be 40 bits. However, previous key distribution schemes with small keys are vulnerable to a dictionary or pre-computation attack. In contrast, EIKU can scale down to a lower security margin and get the benefit of a small key update message, without being vulnerable to dictionary or pre-computation attacks as we show in [PST01].

In addition to other properties, EIKU also achieves the smallest size for key updates, as low as the best variants of LKH or OFT. See Section 6.7.5 for a comparison of the schemes. EIKU furthermore has hints (even more compressed version of key updates) by trading off key update size with receiver

computation. Assuming current workstations, the hint mechanism reduces the size of each key in the key update by about 16 bits. This approach can result in substantial savings, especially in low-security settings.

To achieve robustness to packet loss, ELK uses redundant re-sending of key update messages. This is a direct tradeoff between communication overhead and robustness to packet loss. To reduce the overhead of sending many redundant packets, the key server uses the VIB technique (see Section 6.5.4).

Since encrypted data without a corresponding decryption key is useless, combining the two is natural. The small size of hint key update messages and the VIB technique in ELK allows the sender to add a few bytes of key update message to each data packet. The majority of receivers can thus recover a missed group from the information contained in a data packet, faster than contacting the key server.

Even though group key distribution protocols allow users to join or leave at any time, aggregating join and leave events can bring substantial savings [CEK+99, SKJ00]. The ET and TST techniques both naturally aggregate events. However, aggregating unpredicted leave events also results in a smaller key update message. The duration during which the key server aggregates events is application specific, picking this parameter trades off overhead with membership granularity.

6.7 Applications and Practical Issues

This section discusses the choice for the parameters for ELK. The parameters we discuss are: n, n_1, n_2, and n_3, (the number of bits of the key, the left contribution, the right contribution, and the size of the key verification, respectively), and the number of levels of keys that are added to the hint. The choice of these parameters is driven by the tradeoff between efficiency (communication and computation) and security.

6.7.1 Security Model

Our attacker model assumes a "reasonable" attacker who breaks a system by breaking the weakest link. The main application of this work is a broadcast environment where the receivers do not have tamper-resistant security devices. This implies that a user has access to the decryption keys, because they are stored in memory. Hence, an attacker can always obtain the current group key by subscribing to the service. From another perspective, how secret can a group key be if it is shared by 10^5 members? We judge our key distribution

protocol as secure if it is considerably more difficult and expensive to get the key by breaking the key distribution protocol than by other means.

6.7.2 System Requirements

Besides the security requirements, we also have system requirements. We want to have have a key update protocol that has low computation and communication overhead. For the hint messages we require that at least 98% of all receivers can recover the key from the hint message. This implies that the hint message includes the keys from at least 6 levels. The hint to a key of level i may help a fraction of 2^{-i} members, in the case of a single member leave event (assuming that the key tree is balanced). Since $\sum_{i=1}^{6} 2^{-i} = 0.984$, 6 levels are sufficient to reach 98% of the members.

Furthermore we require that the key reconstruction be faster than requesting a key update by unicast from the key server. We assume that such a request message may take around 200 ms. Hence the requirement is that a fast workstation could reconstruct 6 keys in less than 200 ms.

As we discuss in the implementation section, our test workstation computes 5 million PRF functions per second.

6.7.3 Parameters

The first parameter we choose is the group key size n. In this instance we assume that ELK is used for a medium-security environment, and we choose the key size of 64 bits.

Next, we want to achieve that the key reconstruction from a hint takes at most 200 ms, which allows us to compute n_1. In the worst case, a member needs to reconstruct six keys. Assuming it can compute 5 million PRFs per second, this leaves 166666 PRF computations per key. For each key guess, two PRF computations are necessary, one to compute the key, and the other to verify the key with the hint. Therefore, we chose to use 16 bits for n_1, which translates into $2 \cdot 2^{16} = 131072$ computations for each key. As we discussed previously, if the hint message is also 16 bits long, the member expects to get one additional false positive key per level. This implies that the member who computes six keys will end up with seven candidate keys for the group key, which requires six times more work to compute the group key. For this reason, we make the key hint $n_3 = 17$ bits long, which reduces the false positives.

We now compute the number of bits for n_2 based on a sample scenario.

Protecting Perishable Information Goods

Perishable information goods loose their value with time. An example is a live "pay-per-view" media transmissions such as a sports event video feed. Consider an example of perishable data that we want to protect for 10 minutes. The security requirements dictate that an attacker needs at least 1000 computers to break the key in less than 10 minutes.

We assume that the attacker has fast machines that compute 10^7 PRFs per second. The 1000 computers can hence compute $3.6 \cdot 10^{13}$ PRFs in 10 minutes. Since we know that the attacker needs at least $2 \cdot 2^{n_1+n_2}$ operations to break the key, we can derive $n_1 + n_2 \approx 44$, and $n_2 = 28$ bits. The key update per key is $n_1 + n_2 = 44$ bits long, and the total size per key of a hint is $n_2 - n_1 + n_3 = 29$ bits. Hence the savings of the hint message are 34%. For each key hint we also encode the position of the hint in the key tree with an additional bit (left or right). Hence the cost for 6 key hints is $6 * 30 = 180$ bits, which translates into 23 bytes.

6.7.4 Advantages

As computers get faster, the savings of ELK improve. We achieve the largest savings for the hint messages if $n_1 = n_2$, in which case the hint is only half the size of the key update. In this case the legitimate members perform 2^{n_1} operations, but the attacker cannot perform $2^{n_1+n_2} = 2^{2n_1}$ operations. Merkle used a similar argument to construct a public/private key encryption algorithm [Mer78]. In his *Merkle Puzzle* system, he assumes that legitimate users can perform 2^n operations, but an attacker would need to perform 2^{n+m} operations to break the encrypted message. Brute-force search for a parameter was also used by Dwork and Naor to fight spam mail [DN93]. Similarly, Manber [Man96], and Abadi, Lomas, and Needham [ALN97] used brute-force search to improve the security of the UNIX one-way password function.

The above examples demonstrate that ELK can trade off security and efficiency. It allows the content distributor to choose the desired level of security on a fine-grained scale and lower security directly translates into smaller key update messages. Note that previous protocols do not offer this feature. If one desires a lower security margin it is not safe to shorten the key length, because new attacks that exploit short key lengths become possible. For instance, an attacker may perform pre-computation and use memory lookup tables to break the short group key. ELK does not suffer from these attacks, as we show in [PST01].

	LKH++	OFT	MARKS	ELK		
				Full	Hint (1/2)	Hint $(1-2^{-i})$
Single member join						
B'cast size	d	dn	0	0	0	0
Multiple member join (j members)						
B'cast size	dj	$a_j n$	0	0	0	0
Predicted single/multiple member leave						
B'cast size	$(d-1)n$	dn	0	0	0	0
Unpredicted single member leave						
B'cast size	$(d-1)n$	dn	N/A	$(d-1)(n_1+n_2)$	n_2	in_2
Unpredicted multiple member leave (j members)						
B'cast size	$(a_j-j)n$	$(a_j-1)n$	N/A	$(a_j-j)(n_1+n_2)$	$b_j n_2$	$c_j n_2$

Table 6.1. Comparison between current key distribution schemes. All quantities are in number of bits, and we do not account for the tree location information that needs to be passed along with each key. The parameters a_j, b_j, and c_j are explained in the text.

6.7.5 Comparison with Related Work

In this work we focus on broadcast overhead, since we consider it to be the most important quantity. Unicast cost, memory overhead at the key server, and computation overhead are of lesser concern.[4] By reducing the size of the key updates, one can have a higher level of redundancy (assuming that a constant fraction of the bandwidth is dedicated for key updates) which results in higher reliability.

Table 6.1 shows a comparison of the standard key distribution protocols. N is the number of users in the group and d is the height of the key tree. Assuming that the tree is balanced, we have $d = \lceil \log_2(N) \rceil$. The location information of a node in the tree also takes d bits. The quantity a_j determines the number of keys that change in the tree when j members join or leave.

The Keystone protocol [WL00, YLZL01, LYGL01] is based on key trees, but since it does not incorporate recent developments to reduce the size of the key update messages it is more expensive than LKH++, OFT, or ELK. We can clearly see that ELK is the most efficient protocol, even in the case where ELK has the same security parameter as the other protocols ($n_1 + n_2 = n$). Our new join protocol does not require any broadcast message, since the entire tree changes in each interval. In particular for the case of j leaving members, ELK provides savings with a factor of $a_j - j$ factor instead of the OFT's $a_j - 1$ factor.

[4] If a broadcasting company has 10^6 paying customers, it can also afford a server with sufficient memory to store the $2 \cdot 10^6$ keys.

To display the overhead of the hint message, we list two cases: One where the hint helps half the members to reconstruct the group key, and the other that helps a fraction of $1 - 2^{-i}$ members. To simplify the table, we marked the size of the hint message as n_2[5] In the case when j members leave the group, the size of the hint message is more difficult to give. The expected number of keys that allows half the members to recover the group key is $b_j = \sum_{w=0}^{v-1} 2^w(1 - (1 - 2^{-w})^j)$ with $v = \lceil -\frac{\log(1-2^{-1/j})}{\log(2)} \rceil$
The formula for the case where a fraction of $1 - 2^{-i}$ member wish to recover the group key from the hint is
$c_j = \sum_{w=0}^{v-1} 2^w(1 - (1 - 2^{-w})^j)$ with $v = \lceil -\frac{\log(1-2^{-i/j})}{\log(2)} \rceil$

So far, Briscoe's MARKS protocol was the only protocol that did not require any broadcast message for member join and predicted leave events [Bri99]. However, the MARKS protocol cannot support early leave or evict operations (i.e., unpredicted member leave events). The ELK protocol is the first protocol that provides unpredicted member leave events, while providing join and predicted leave without broadcasting a key update message.

6.7.6 Unicast Key Recovery Protocol

In case the group member misses a key update message, it needs to request the keys from the key server. This request protocol is straightforward to design. The most important requirement is that the protocol is efficient and scales well. In contrast to previous work by Wong and Lam, we find that TCP is not the appropriate protocol for this key request protocol [WL00]. The reason is that TCP requires one round-trip message just to set up the connection (in most implementations, no data is sent in the initial SYN packet). This slows down the request unnecessarily. An advantage of TCP might be that the receiver could keep the connection open for future key requests. This approach, however, does not scale to large number of receivers because each TCP connection requires a considerable amount of state at the server. Therefore, we propose to use a light-weight key update protocol based on UDP, where the receiver achieves reliability through timeout and request retransmissions.[6]

[5] If the hint size is $n_2 - n_1 + n_3$, and $n_3 = n_1 + 1$, the hint size would actually be $n_2 + 1$.

[6] The only advantage we see for using TCP is to get through firewalls, as a firewall might filter out UDP packets.

6.8 Appendix

In this Appendix, we present details of the Evolving Tree (ET) and the Entropy Injection Key Update (EIKU) protocols.

6.8.1 Additional Cryptographic Primitives

In addition to the notation in Chapter 2, we use the following notation:

- In the following description, we use many different pseudo-random functions (PRFs) with varying input and output lengths. We describe PRFs in Section 2.2.5. Instead of defining one PRF for each purpose, we define a family of PRFs that use the k-bit long key K on input M of length m bits, and output n bits:
  $\mathsf{PRF}^{\langle m \to n \rangle} : \{0,1\}^k \times \{0,1\}^m \to \{0,1\}^n$. We write $\mathsf{PRF}_K^{\langle m \to n \rangle}(M)$.
- The function $\mathsf{LSB}^{\langle n \rangle}(M)$ returns the n least significant bits of M (assuming that M is longer than n bits).
- We use a number of key derivation functions to ensure that the keys and arguments in various places are independent. The following keys are derived from K_i as follows:
  $K_i^{\alpha} = \mathsf{PRF}_{K_i}^{\langle n \to n \rangle}(1)$, $K_i^{\beta} = \mathsf{PRF}_{K_i}^{\langle n \to n \rangle}(2)$,
  $K_i^{\gamma} = \mathsf{PRF}_{K_i}^{\langle n \to n \rangle}(3)$, and $K_i^{\delta} = \mathsf{PRF}_{K_i}^{\langle n \to n \rangle}(4)$
  (Note that $1, 2, 3$ and 4 are arguments to the PRFs. The purpose of this key derivation is to ensure key independence. However, to simplify understanding of the protocols we advise to ignore the Greek superscripts on the first reading.)

6.8.2 ET Detailed Description

At the beginning of every join interval, the key server updates the K_i as follows: $K_i' = \mathsf{PRF}_{K_i^{\delta}}^{\langle n \to n \rangle}(K_G)$, where $K_i^{\delta} = \mathsf{PRF}_{K_i}^{\langle n \to n \rangle}(4)$ and K_G is the current group key. To update the group key we use the derivation $K_G' = \mathsf{PRF}_{K_G^{\delta}}^{\langle n \to n \rangle}(0)$. Each member can update its key path independently in each time interval. The box labeled Protocol 6.2 shows the detailed join protocol.

To illustrate the join protocol, Figure 6.6 shows a sample join event where member M_4 joins the group and the key server decides to insert M_4 at the leaf node of M_3 right away, thus giving the member a join bonus. Step 2 of Protocol 6.2 does not apply since no empty leaf nodes are available. In Step 3 the key server generates the new leaf node and assigns it a new random key K_7. In Step 4, the key server merges M_4 to the node of K_5', and

1 Based on the group policy and arrival time of the joining member, the key server decides whether to give the new member a join bonus (current keys, violating backward secrecy), or whether to pass the keys of the next join interval (so new member would experience a join latency).

2 If an empty leaf node is available, the server assigns a new random key to the leaf node and sends it over a secure channel to the new member along with the keys on the key path. Thus the join event is done, and no more exchanges or broadcasts are necessary.

3 If no empty leaf node is available, the key server assigns the new member M to a new leaf node N_M of the key tree and assigns it a new random key K_M.

4 The key server picks a node N_j of the key tree to insert the new member. Assume that the key at that node is K_j. The server demotes the node N_j and generates a new parent node N_P for the leaf N_M and node N_j. The node N_j becomes the left child, and the node N_M the right child of N_P. The key value of the parent becomes $K_P = \mathsf{PRF}_{K_j^\delta}^{\langle n \to n \rangle}(1)$ (such that every member in subtree N_j can derive the key of the new parent node K_P).

5 The key server sends the key path to the new member over a secure channel.

6 The key server sends the joining location to the members that are below node N_j, which allows them to independently update their own key paths. In the general case, N_j is a leaf node and the key server sends a unicast message to that member. If the key server needs to inform a large number of members, a broadcast message may be more efficient: the key server broadcasts the join location N_j in a key update message, which takes at most $O(\log(N))$ bits.

Protocol 6.2: Single member join protocol

generates the parent node K_6 of K_5' and K_7. The key server computes the new key: $K_6 = \mathsf{PRF}_{K_5^\delta}^{\langle n \to n \rangle}(1)$. In Step 5 the key server sends M_4 the message $\{K_1', K_3', K_6\}_{K_7}$. In Step 6, the key server sends the joining location of the new member to M_3 by unicast, so M_3 updates its key tree and computes $K_6 = \mathsf{PRF}_{K_5^\delta}^{\langle n \to n \rangle}(1)$.

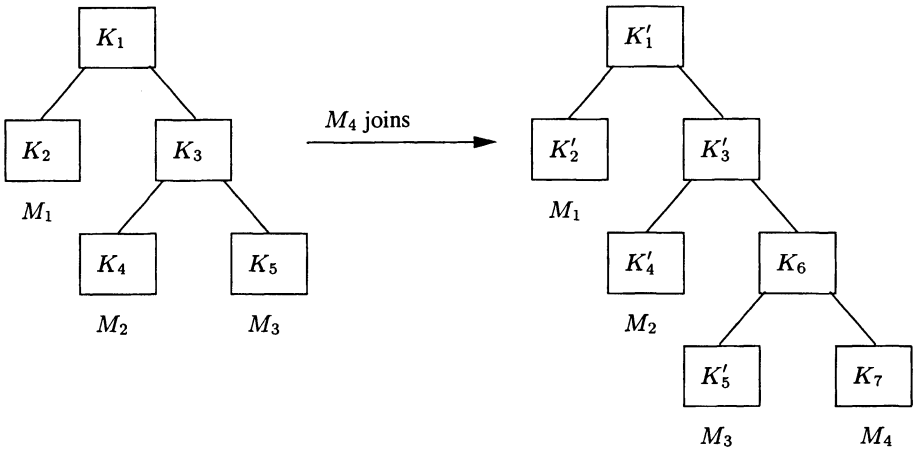


Figure 6.6. Member join event. This figure shows the evolution of the tree when member M_4 joins the group. In this case, the key server decides to insert member M_4 at the node that corresponds to key K_5. The key server demotes K_5 and inserts a new node K_6 and K_7. In the ET protocol, all members update their keys, so neither M_1 nor M_2 need to know about the new joining member. The key server sends a unicast message to inform M_3 of the new member, and M_3 can itself compute $K_6 = \mathsf{PRF}_{K_5^\delta}^{\langle n \to n \rangle}(1)$. The key server sends the new member M_4 the keys K'_1, K'_3, K_6, and K_7 by unicast.

6.8.3 EIKU Detailed Description

Now we present details of the Entropy Injection Key Update protocol.

6.8.3.1 Entropy Contribution for Parent Key Update

To preserve forward secrecy, the key server needs to update all the key paths of all leaving members (see Section 6.8.1).

We now describe how to update a key K that has two child keys K_L and K_R. We assume that the key length is n bits. The new key K' is derived from K and entropy contributions from both children. The left child key K_L contributes n_1 bits of entropy to the new key. We call the contribution from the left child C_L. To hide K_L, we derive C_L with the one-way construction using pseudo-random functions (PRF, as we explain in Section 2.2.5): $C_L = \mathsf{PRF}_{K_L^\alpha}^{\langle n \to n_1 \rangle}(K)$. The entropy contribution from the right child key is $C_R = \mathsf{PRF}_{K_R^\alpha}^{\langle n \to n_2 \rangle}(K)$. Note that C_R is n_2 bits long. For now we may assume that $n_1 = n_2$ — for the hint mechanism we present in the next section, however, we generally have $n_1 \neq n_2$. We then combine the left and right entropy contributions by concatenating

them $C_{LR} = C_L \mid\mid C_R$. We derive the new ($n$-bit) key as follows: $K' = \mathsf{PRF}_{C_{LR}}^{\langle n \to n \rangle}(K)$.

In Sections 6.8.3.5 we show that the security of this approach is $n_1 + n_2$ bits for a leaving member (because we assume that the leaving member knows the n-bit long key K), and n bits for an outsider (because the outsider does not know any group keys).

The key update message is $\{C_R\}_{K_L}, \{C_L\}_{K_R}$. If we use a non-expanding encryption mode (e.g., a stream cipher) the size of the key update is $n_1 + n_2$ bits. The members in the left-hand subtree know K_L, so they can decrypt C_R and derive the new parent key K' from the old parent key K and their own contribution:

$$C_{LR} = \mathsf{PRF}_{K_L^a}^{\langle n \to n_1 \rangle}(K) \mid\mid C_R$$
$$K' = \mathsf{PRF}_{C_{LR}}^{\langle n \to n \rangle}(K)$$

The right hand members can decrypt the left contribution C_L, and derive the new key in the same way.

The box labeled Protocol 6.3 lists the details of this key update. The EIKU construction has many nice properties. First, the key update message size is as small as the best related approaches LKH++ and OFT. In the next section, we describe the hint mechanism, which compresses the key update message size. As we mentioned above, EIKU allows us to use ET to join new members (in contrast to OFT).

Finally, EIKU is the first key update method that allows the key server to shrink the key update size without opening the doors to pre-computation and memory-lookup attacks. For example, consider a broadcast setting that does not require strong security, so management decided that 40-bit long keys are long enough. Unfortunately, a 40-bit long key in LKH++ or OFT does not mean that an attacker needs to perform 2^{40} operations to break the key, because a clever attacker can, for example, pre-compute a table with some known plaintext encrypted under every possible key. When the attacker sees an encrypted block, it can simply look up the corresponding key. Thus, previous key distribution require much longer keys, even for low-security applications. Recall that EIKU uses large internal keys (of size n bits, for example $n = 128$ bits), but the size of the key update message is still $n_1 + n_2$ bits. This construction foils these pre-computation attacks, as we show in [PST01]. Thus, EIKU allows us to safely scale down the security and to take advantage of the lower key update message size.

- The left-hand entropy contribution is $C_L = \mathsf{PRF}^{\langle n \to n_1 \rangle}_{K_L^\alpha}(K)$
- The right-hand entropy contribution is $C_R = \mathsf{PRF}^{\langle n \to n_2 \rangle}_{K_R^\alpha}(K)$
- The combined entropy contribution is $C_{LR} = C_L \mid\mid C_R$
- The new key is $K' = \mathsf{PRF}^{\langle n \to n \rangle}_{C_{LR}}(K)$
- Recall from Section 6.8.1 that $K_L^\alpha = \mathsf{PRF}^{\langle n \to n \rangle}_{K_L}(1)$ and also recall that $K_L^\beta = \mathsf{PRF}^{\langle n \to n \rangle}_{K_L}(2)$. The purpose of this key derivation is to make K_L^α (used to derive the entropy contribution) and K_L^β (used as key for encryption) computationally independent.
- To update the key, the server broadcasts $\{C_R\}_{K_L^\beta}, \{C_L\}_{K_R^\beta}$.
- This key update message has a length of $n_1 + n_2$ bits. The members who know K_L can derive K_L^β, decrypt C_R from the key update, and compute K'. The same applies to the members who know K_R.

Protocol 6.3: EIKU key update protocol

6.8.3.2 Key Recovery from Hints

A unique property of EIKU is that it allows us to use receiver computation to make key update messages smaller. This section presents this mechanism.

The current EIKU key update message (to update key K to K', with child keys K_L and K_R) is n_1+n_2 bits long. If we assume that a member can perform 2^{n_1} computations, a member who knows K and either K_L or K_R can recover the new key K' from a *hint* that is n_1 bits smaller than the key update message. Without loss of generality, we assume that $n_1 \leq n_2$.

Consider the members who are in the right-hand subtree of K — they know K and K_R. They can derive the right contribution C_R that is n_2 bits long $C_R = \mathsf{PRF}^{\langle n \to n_2 \rangle}_{K_R^\alpha}(K)$. Intuitively, the hint is a checksum of the new key K'. The hint message contains the *key verification* $V_{K'}$, which is derived from the new key $V_{K'} = \mathsf{PRF}_{K'}(0)$ and has a length of n_3 bits (for now, assume that $n_3 = n_1$). The members can compute the left side contribution C_L by brute force and validate each guess of C_L with the key verification in the hint. More concretely, for each one of the 2^{n_1} possibilities of C'_L, they derive $C'_{LR} =$

$C'_L \mid\mid C_R$ and compute the candidate key $\tilde{K} = \mathsf{PRF}_{C'_{LR}}(K)$. The members validate the candidate key by verifying $\mathsf{PRF}_{\tilde{K}}(0) = V_{K'}$.

If $n_1 = n_2$, the members who are in left-hand subtree of K compute the key in the same way as the right-hand members. In the general case, however, $n_1 < n_2$; and the left-hand members need to obtain $n_2 - n_1$ bits of the right entropy contribution, so that they still only need to brute-force n_1 bits. So the hint message also contains the least significant $n_2 - n_1$ bits of C_R, encrypted with K_L. We split up C_R into the n_1 bit long chunk C_{R1}, and into the $n_2 - n_1$ bit long chunk C_{R2}, so $C_R = C_{R1} \mid\mid C_{R2}$. So the members in the left subtree only need to brute force the n_1 bit long chunk C_{R1} (so the amount of work is the same for the members in both subtrees), and the hint message contains $\{C_{R2}\}_{K_L}$. With this help, the left-hand members validate the 2^{n_1} possibilities for C'_{R1}. They derive $C'_{LR} = C_L \mid\mid C'_{R1} \mid\mid C_{R2}$ and compute the candidate key $\tilde{K} = \mathsf{PRF}_{C'_{LR}}(K)$. The members validate the candidate key by verifying $\mathsf{PRF}_{\tilde{K}}(0) = V_{K'}$.

This description was simplified, see the box labeled Protocol 6.4 for the full details.

This procedure outputs the correct key. A problem is that the procedure might output more than one candidate key, in which case some of them are false positives. If the key verification is also n_1 bits long ($n_3 = n_1$), we expect to receive one false positive key next to the correct key. If a member recomputes multiple keys, it will receive an additional key on each level, since a false positive key produces on average just one false positive key, but the correct key will result in the correct key for the next level, most likely along with one false positive key. To prevent this, we set $n_3 > n_1$. Generally setting $n_3 = n_1 + 1$ results in "half a false positive key" on average on each level, which prevents the steady increase of false positives over multiple levels.

The advantage of the hint is that it is $2n_1 - n_3$ bits shorter than the key update. (In general, $n_3 = n_1 + 1$ and hence the hint is $n_1 - 1$ bits shorter than the full key update.) In the ideal case $n_1 = n_2$ and then the hint is about half the size of the key update.

Here is how we might choose n, n_1, n_2, and n_3. The length of the keys n needs to be large enough to prevent brute force attacks, since n does not influence the size of the key update message, we may choose n sufficiently large (e.g., $n = 80$ bits or $n = 128$ bits). Each key hint basically discloses a checksum of the key, so we still need to make n large enough to compensate for that. Since the checksum is n_3 bits long, we advise to increase the size of the group key by at least n_3 bits. However, n does influence the size of key

- The left-hand entropy contribution is n_1 bits long: $C_L = \mathsf{PRF}_{K_L^\alpha}^{\langle n \to n_1 \rangle}(K)$
- The right-hand entropy contribution is n_2 bits long: $C_R = \mathsf{PRF}_{K_R^\alpha}^{\langle n \to n_2 \rangle}(K)$
- We split up the right-hand entropy contribution into two chunks: C_{R1} is n_1 bits, and C_{R2} is $n_2 - n_1$ bits long, such that $C_R = C_{R1} \mid\mid C_{R2}$
- The combined entropy contribution is $C_{LR} = C_L \mid\mid C_R$
- The updated key is $K' = \mathsf{PRF}_{C_{LR}}^{\langle n \to n \rangle}(K)$
- The hint message is composed of the key verification and the partial right contribution: $V_{K'} = \mathsf{PRF}_{K'^\gamma}^{\langle n \to n_3 \rangle}(0),\ \{C_{R2}\}_{K_L^\beta}$.
- The key reconstruction is slightly different for the members in the right and left subtree.
  - The members on the right know K_R and they can compute n_2 bits of C_{LR}, so they only need to exhaustively try 2^{n_1} combinations for C'_L. For each guess of C'_L, they set $C'_{LR} = C'_L \mid\mid C_R$, compute $\tilde{K} = \mathsf{PRF}_{C'_{LR}}^{\langle n_1 + n_2 \to n \rangle}(K)$, and verify $\mathsf{PRF}_{\tilde{K}^\gamma}^{\langle n \to n_3 \rangle}(0) \stackrel{?}{=} V_{K'}$. (Recall that $K'^\gamma = \mathsf{PRF}_{K'}^{\langle n \to n \rangle}(3)$.) If they are equal, the key is a candidate.
  - The members on the left know K_L, derive K_L^β, and can hence decrypt part of the right key's contribution C_{R2}. Now they know all but n_1 bits of C_{LR}. So for each guess of C'_{R1}, they set $C'_{LR} = C_L \mid\mid C'_{R1} \mid\mid C_{R2}$, compute $\tilde{K} = \mathsf{PRF}_{C'_{LR}}^{\langle n_1 + n_2 \to n \rangle}(K)$, and verify $\mathsf{PRF}_{\tilde{K}^\gamma}^{\langle n \to n_3 \rangle}(0) \stackrel{?}{=} V_{K'}$. If they are equal, the key is a candidate.

Protocol 6.4: Key recovery from hint

messages that the server sends by unicast to individual members (these packets usually contain the explicit keys of the key paths, and not just the entropy contribution). Based on the expected receiver's computation resources and the time delay we expect a member to re-compute a key from a hint, we compute n_1 (since the receiver needs to perform 2^{n_1} computations to recompute one key). Next, we choose the security parameter for the group: how many PRF

computations does an adversary expect to perform to break the group key? In EIKU, the adversary has to compute $2^{n_1+n_2}$ PRF operations to break the group key, so we can derive n_2. Finally, we pick $n_3 = n_1 + 1$, such that the expected number of false positive root keys is less than one. We discuss these choices with an example in Section 6.7.

6.8.3.3 Unpredicted Member Leave Event

To achieve forward secrecy, the key server needs to update all keys on all the key paths of all leaving members. After deleting the leaf nodes (or replacing the key with a fresh key) of leaving members, the key server uses EIKU to update the key tree. The server broadcasts a key update message containing the updated keys and also attach hint messages to data packets to enable key recovery in case the key update message is lost.

1 The server deletes the leaf node corresponding to the leaving member, as well as the parent node of the leaf node, and promotes the sibling node.

2 All remaining nodes on the key path of the leaving member need to be updated. For each of these keys, the update procedure for key K_i is as follows. The new key is $K_i' = \mathsf{PRF}_{C_{LR}}^{\langle n \to n \rangle}(K_i)$, with $C_{LR} = \mathsf{PRF}_{K_{il}^{\alpha}}^{\langle n \to n_1 \rangle}(K_i) \mid\mid \mathsf{PRF}_{K_{ir}^{\alpha}}^{\langle n \to n_2 \rangle}(K_i)$, where K_{il} and K_{ir} are the left and right child keys, respectively.

3 The server broadcasts the key update message of all keys that were updated in the previous step. Hence, for each key K_i' the update message contains $\{\mathsf{PRF}_{K_{il}^{\alpha}}^{\langle n \to n_1 \rangle}(K_i)\}_{K_{ir}^{\beta}}, \{\mathsf{PRF}_{K_{ir}^{\alpha}}^{\langle n \to n_2 \rangle}(K_i)\}_{K_{il}^{\beta}}$.

4 The server attaches the following hint message to each data packet. For each new key K_i' the server may send $\mathsf{PRF}_{K_i'^{\gamma}}^{\langle n \to n_3 \rangle}(0), \{\mathsf{LSB}^{\langle n_2 - n_1 \rangle}(\mathsf{PRF}_{K_{ir}^{\alpha}}^{\langle n \to n_2 \rangle}(K_i))\}_{K_{il}^{\beta}}$.[7]

Protocol 6.5: Detailed member leave protocol

This is clearer in an example. See Figure 6.7 and the box labeled Example 6.1.

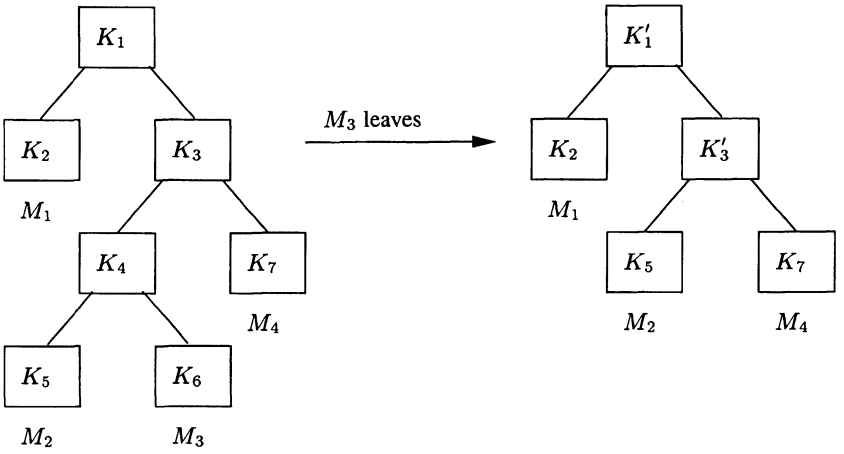


Figure 6.7. This figure shows the change of the key tree as member M_3 leaves the group. We describe in the text how the key server updates the key tree.

6.8.3.4 Multiple Member Leave Events

Above, we write that there are several good ways to realize multiple member join events. Multiple member leave events have some subtlety though, so we sketch the approach in slightly greater detail.

In case multiple members leave in the same interval, the key server aggregates all the leaving members and creates a joint key update message. EIKU can aggregate the concurrent member leave events particularly well and provides (in addition to current savings) up to 50% savings over OFT for keys where both children change. In OFT, if both child keys change, the key update contains two blinded keys (one for each child). In contrast, the EIKU key update already contains the entropy contribution from both child keys, regardless of whether the child keys changed or not. The size of the key update message in EIKU is $(a - j) \cdot (n_1 + n_2)$, where a is the number of updated keys and j is the number of leaving members. Since the number of updated keys is always less than the sum of all keys in each key path, it is always advantageous to aggregate multiple member leave events. In contrast, the size of the OFT key update message is $(a - 1) \cdot n$ bits.

We illustrate multiple member leave events with an example. Assume the setting of Figure 6.8. For this example we assume that the leaving member nodes are not collapsed, because the key server replaces the leaving member nodes with new members. If members M_3 and M_4 both leave, and new mem-

Consider the example in Figure 6.7. Member M_3 leaves the group. In step 1, the key server deletes the nodes that correspond to the keys K_4 and K_6. The server promotes the node of K_5. In step 2, the server updates the keys on the key path of the leaving member K_3 and K_1.

- The server computes the new key K'_3 as follows: $K'_3 = \mathsf{PRF}^{\langle n \to n \rangle}_{C_{LR3}}(K_3)$, with
  $C_{LR3} = \mathsf{PRF}^{\langle n \to n_1 \rangle}_{K_5^\alpha}(K_3) \mid\mid \mathsf{PRF}^{\langle n \to n_2 \rangle}_{K_7^\alpha}(K_3)$.
- To update key K_1 the key server computes $K'_1 = \mathsf{PRF}^{\langle n \to n \rangle}_{C_{LR1}}(K_1)$ and
  $C_{LR1} = \mathsf{PRF}^{\langle n \to n_1 \rangle}_{K_2^\alpha}(K_1) \mid\mid \mathsf{PRF}^{\langle n \to n_2 \rangle}_{K_3'^\alpha}(K_1)$.
- In step 3 the server broadcasts the key update, which contains the following:
  $\{\mathsf{PRF}^{\langle n \to n_1 \rangle}_{K_5^\alpha}(K_3)\}_{K_7^\beta}, \{\mathsf{PRF}^{\langle n \to n_2 \rangle}_{K_7^\alpha}(K_3)\}_{K_5^\beta}$ for K'_3, and
  $\{\mathsf{PRF}^{\langle n \to n_1 \rangle}_{K_2^\alpha}(K_1)\}_{K_3'^\beta}, \{\mathsf{PRF}^{\langle n \to n_2 \rangle}_{K_3'^\alpha}(K_1)\}_{K_2^\beta}$ for K'_1.
- Finally, subsequent data packets contain the following hint (step 4):
  $\mathsf{PRF}^{\langle n \to n_3 \rangle}_{K_1'^\gamma}(0), \{\mathsf{LSB}^{\langle n_2 - n_1 \rangle}(\mathsf{PRF}^{\langle n \to n_2 \rangle}_{K_3'^\alpha}(K_1))\}_{K_2'^\beta}$,
  $\mathsf{PRF}^{\langle n \to n_3 \rangle}_{K_3'^\gamma}(0), \{\mathsf{LSB}^{\langle n_2 - n_1 \rangle}(\mathsf{PRF}^{\langle n \to n_2 \rangle}_{K_7^\alpha}(K_3))\}_{K_5^\beta}$

Example 6.1: Detailed member leave protocol

bers M_5 and M_6 take their spot, the keys K_4, K_3, and K_1 need to be updated. If the member leave events are processed sequentially, the update message for M_3 leaving is $3(n_1 + n_2)$ bits long, and the message for M_4 is $2(n_1 + n_2)$ bits long. If the server aggregates leaves, the message is only $3(n_1 + n_2)$ bits long.

6.8.3.5 Security Analysis

A sketch of the security analysis appears in [PST01]. The following observations hold:

- With overwhelming probability, a passive adversary must perform $\Omega(2^n)$ operations to brute-force a EIKU group key.
- With overwhelming probability, an active adversary must perform $\Omega(2^n)$ operations to brute-force any EIKU group key preceeding the time it joins the group.

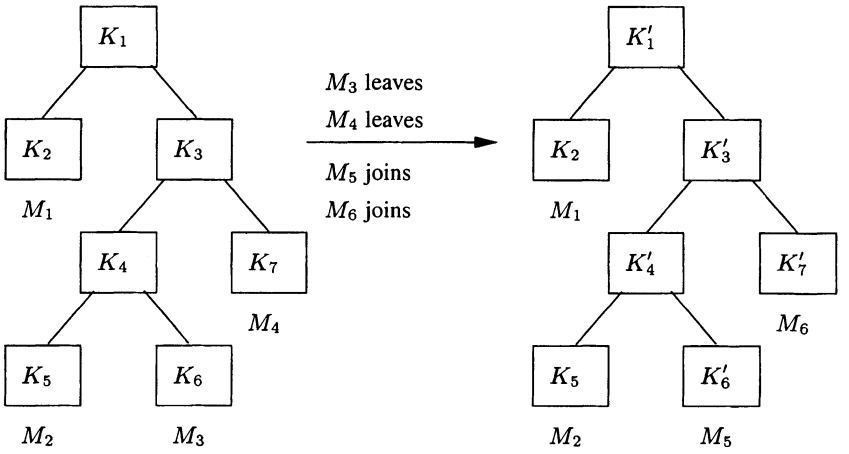


Figure 6.8. This figure shows the change of the key tree on a multiple member join and leave event, as members M_3 and M_4 leave the group and members M_5 and M_6 join the group. We describe in the text how the key server updates the key tree.

- After the active adversary leaves the group, with overwhelming probability it must perform $\Omega(2^{n_1+n_2})$ operations to derive the new EIKU group key.

[PST01] shows that pre-computation does not reduce the effort to brute-force a later EIKU group key.

Chapter 7

SENSOR NETWORK SECURITY

A central goal of this book is to propose secure mechanisms for broadcast communication applicable in a wide range of settings. To demonstrate the flexibility and generality of our techniques, we consider a highly resource-constrained broadcast environment and show that our methods can scale down to this environment, and achieve a secure system.

Wireless sensor networks will be widely deployed in the near future. While much research has focused on making these networks feasible and useful, security has received little attention.

We expand on our suite of security building blocks by considering protocols especially tailored for resource-constrained environments and wireless communication. We design two secure building blocks: SNEP and μTESLA. SNEP provides the following important baseline security primitives:

- data confidentiality,
- two-party data authentication (point-to-point),
- and evidence of data freshness.

A particularly important but challenging problem is to provide efficient broadcast authentication in this limited environment. μTESLA is a new protocol which provides authenticated broadcast for severely resource-constrained environments. We implemented the above protocols, and show that they are practical even on minimal hardware: the performance of the protocol suite easily matches the data rate of our network. Additionally, we demonstrate that the suite can be used for building higher level protocols.

A. Perrig et al., *Secure Broadcast Communication*

We envision a future where thousands to millions of small sensors form self-organizing wireless networks. How can we provide security for these sensor networks? Security is not easy; compared with conventional desktop computers, severe challenges exist — these sensors will have limited processing power, storage, bandwidth, and energy.

We need to surmount these challenges, because security is so important. Sensor networks will expand to fill all aspects of our lives. Here are some typical applications:

- *Emergency response information*: sensor networks will collect information about the status of buildings, people, and transportation pathways. Sensor information must be collected and passed on in meaningful, secure ways to emergency response personnel.

- *Energy management*: in 2001 power blackouts plagued California. Energy distribution will be better managed when we begin to use remote sensors. For example, the power load that can be carried on an electrical line depends on ambient temperature and the immediate temperature on the wire. If these were monitored by remote sensors and the remote sensors received information about desired load and current load, it would be possible to distribute load better. This would avoid circumstances where Californians cannot receive electricity while surplus electricity exists in other parts of the country.

- *Medical monitoring*: we envision a future where individuals with some types of medical conditions receive constant monitoring through sensors that monitor health conditions. For some types of medical conditions, remote sensors may apply remedies (such as instant release of emergency medication to the bloodstream).

- *Logistics and inventory management*: commerce in America is based on moving goods, including commodities from locations where surpluses exist to locations where needs exist. Using remote sensors can substantially improve these mechanisms. These mechanisms will vary in scale — ranging from worldwide distribution of goods through transportation and pipeline networks to inventory management within a single retail store.

- *Battlefield management*: remote sensors can help eliminate some of the confusion associated with combat. They can allow accurate collection of information about current battlefield conditions as well as giving appropriate information to soldiers, weapons, and vehicles in the battlefield.

At UC Berkeley, we think these systems are important, and we are starting a major initiative to explore the use of wireless sensor networks. More information on this new initiative, CITRIS, can be found at `www.citris.berkeley.edu`. Serious security and privacy questions arise if third parties can read or tamper with sensor data. We envision wireless sensor networks being widely used — including for emergency and life-critical systems — and here the questions of security are foremost.

We present *SPINS*, a set of *Security Protocols for Sensor Networks*. This chapter explores the challenges for security in sensor networks and identifies authenticated broadcast as a necessary but difficult security property. We adapt the TESLA protocol from Chapter 3 and design μTESLA (the "micro" version of TESLA), to provide authenticated streaming broadcast. To provide data confidentiality, two-party data authentication, and data freshness with low overhead we design and develop our *Sensor Network Encryption Protocol* (SNEP).

7.1 Background

Here we outline the limited environment of sensor networks.

7.1.1 Sensor Hardware

At UC Berkeley, we are building prototype networks of small sensor devices under the SmartDust program [PKB99, KKP99, WLLP01], one of the components of CITRIS. We have deployed these in one of our EECS buildings, Cory Hall. We are currently using these for a very simple application — heating and air-conditioning control in the building. However, the same mechanisms that we describe in this paper can be modified to support sensors that handle emergency systems such as fire, earthquake, and hazardous material response.

By design, these sensors are inexpensive, low-power devices. As a result, they have limited computational and communication resources. The sensors form a self-organizing wireless network and form a multihop routing topology. Typical applications may periodically transmit sensor readings for processing.

Our current prototype consists of *nodes*, small battery powered devices, that communicate with a more powerful *base station*, which in turn is connected to an outside network. Table 7.1 summarizes the performance characteristics of these devices. At 4 MHz, they are slow and underpowered (the CPU has good support for bit and byte level I/O operations, but lacks support for many arithmetic and some logic operations). They are only 8-bit processors (note that according to [Ten00], 80% of all microprocessors shipped in 2000 were 4 bit or 8 bit devices). Communication is slow at 10 kilobits per second.

CPU	8-bit, 4 MHz
Storage	8K bytes instruction flash
	512 bytes RAM
	512 bytes EEPROM
Communication	916 MHz radio
Bandwidth	10 Kbps
Operating System	TinyOS
OS code space	3500 bytes
Available code space	4500 bytes

Table 7.1. Characteristics of prototype SmartDust nodes

The operating system is particularly interesting for these devices. We use TinyOS [HSW+00]. This small, event-driven operating system consumes almost half of 8K bytes of instruction flash memory, leaving just 4500 bytes for security and the application.

Significantly more powerful devices would also consume significantly more power. The energy source on our devices is a small battery, so we are stuck with relatively limited computational devices. Wireless communication is the most energy-consuming function performed by these devices, so we need to minimize communications overhead. The limited energy supplies create tensions for security: on the one hand, security needs to limit its consumption of processor power; on the other hand, limited power supply limits the lifetime of keys (battery replacement is designed to reinitialize devices and zero out keys.)[1]

7.1.2 Is Security on Sensors Possible?

These constraints make it impractical to use most current secure algorithms, since they were designed for powerful processors. For example, the working memory of a sensor node is not sufficient to even hold the variables for asymmetric cryptographic algorithms (e.g., RSA [RSA78] with 1024 bits), let alone perform operations with them.

A particular challenge is broadcasting authenticated data to the entire sensor network. Current proposals for authenticated broadcast are impractical for

[1]Base stations differ from nodes in having longer-lived energy supplies and additional communications connections to outside networks.

sensor networks. Most proposals rely on asymmetric digital signatures for the authentication, which are impractical for multiple reasons (e.g., long signatures with high communication overhead of $50 - 1000$ bytes per packet, and high overhead to create and verify the signature). Brown et al. analyze the computation time of various digital signature algorithms on various platforms [BCH+00]: Elliptic Curve Cryptography (ECC) signature algorithms require 1.0–2.2 seconds for one signature generation, and 1.8–5.3 seconds for verification on a Palm Pilot or RIM pager. On the same architecture, a 512-bit RSA signature (which is 64 bytes long) requires 2.4–5.7 seconds for generation, and 0.1–0.6 seconds for verification (depending on the public exponent). More recent signature algorithms provide short signatures, but their verification is always slower than RSA [Sch91, PS98, PS99b, CGP01, HPS01].

Previously proposed purely symmetric solutions for broadcast authentication are impractical for sensor networks: Gennaro and Rohatgi's initial work requires over 1K byte of authentication information per packet [GR97]. Rohatgi's improved k-time signature requires over 300 bytes per packet [Roh99]. We use the TESLA broadcast authentication protocol that Chapter 3 presents. TESLA works well on regular desktop workstations, but uses too much communication and memory on our resource-starved sensor nodes. This chapter extends and adapts TESLA to make it practical for broadcast authentication for sensor networks. We call our new protocol μTESLA.

We have implemented all of these primitives. Our measurements show that adding security to a highly resource-constrained sensor network is feasible.

Given the severe hardware and energy constraints, we must be careful in the choice of cryptographic primitives and the security protocols in the sensor networks.

7.2 System Assumptions

Before we outline the security requirements and present our security infrastructure, we need to define the system architecture and the trust requirements. The goal of this work is to propose a general security infrastructure that is applicable to a variety of sensor networks.

7.2.1 Communication Architecture

Generally, the sensor nodes communicate over a wireless network, so broadcast is the fundamental communication primitive. The baseline protocols account for this property: on one hand they affect the trust assumptions, and on the other they minimize energy usage.

A typical SmartDust sensor network forms around one or more *base stations*, which interface the sensor network to the outside network. The sensor nodes establish a routing forest, with a base station at the root of every tree. Periodic transmission of beacons allows nodes to create a routing topology. Each node can forward a message towards a base station, recognize packets addressed to it, and handle message broadcasts. The base station accesses individual nodes using source routing. We assume that the base station has capabilities similar to the network nodes, except that it has sufficient battery power to surpass the lifetime of all sensor nodes, sufficient memory to store cryptographic keys, and means for communicating with outside networks.

We do have an advantage with sensor networks, because most communication involves the base station and is not between two local nodes. The communication patterns within our network fall into three categories:

- Node to base station communication, e.g., sensor readings.
- Base station to node communication, e.g., specific requests.
- Base station to all nodes, e.g., routing beacons, queries or reprogramming of the entire network.

Our security goal is to address these communication patterns, though we also show how to adapt our baseline protocols to other communication patterns, i.e., node to node or node broadcast.

7.2.2 Trust Requirements

Generally, the sensor networks may be deployed in untrusted locations. While it may be possible to guarantee the integrity of each node through dedicated secure microcontrollers (e.g., [Atm02] or [Dal01]), we feel that such an architecture is too restrictive and does not generalize to the majority of sensor networks. Instead, we assume that individual sensors are untrusted. Our goal is to design the key setup such that a compromise of a node does not spread to other nodes.

Basic wireless communication is not secure. Because it is broadcast, any adversary can eavesdrop on traffic, inject new messages, and replay old messages. Hence, our protocols do not place any trust assumptions on the communication infrastructure, except that messages are delivered to the destination with non-zero probability.

Since the base station is the gateway for the nodes to communicate with the outside world, compromising the base station can render the entire sensor

network useless. Thus the base stations are a necessary part of our trusted computing base. Our trust setup reflects this and so all sensor nodes intimately trust the base station: at creation time, each node gets a *master secret key* $\mathcal{X}$ which it shares with the base station. All other keys are derived from this key, as we show in Section 7.6.

Finally, each node trusts itself. This assumption seems necessary to make any forward progress. In particular, we trust the local clock to be accurate, i.e., to have small drift. This is necessary for the authenticated broadcast protocol we describe in Section 7.5.2.

7.2.3 Design Guidelines

With the limited computation resources available on our platform, we cannot afford to use asymmetric cryptography and so we use symmetric cryptographic primitives to construct our protocols. Due to the limited program store, we construct all cryptographic primitives (i.e., encryption, message authentication code (MAC), hash, random number generator) out of a single block cipher for code reuse. To reduce communication overhead we exploit common state between the communicating parties.

7.3 Requirements for Sensor Network Security

This section formalizes the security properties required by sensor networks, and shows how they are directly applicable in a typical sensor network.

7.3.1 Data Confidentiality

A sensor network should not leak sensor readings to neighboring networks. In many applications (e.g., key distribution) nodes communicate highly sensitive data. The standard approach for keeping sensitive data secret is to encrypt the data with a secret key that only intended receivers possess, hence achieving confidentiality. Given the observed communication patterns, we set up secure channels between nodes and base stations and later bootstrap other secure channels as necessary.

7.3.2 Data Authentication

Message authentication is important for many applications in sensor networks (including administrative tasks such as network reprogramming or controlling sensor node duty cycle). Since an adversary can easily inject messages, the receiver needs to ensure that data used in any decision-making process orig-

inates from a trusted source. Informally, *data authentication* allows a receiver to verify that the data was really sent by the claimed sender.

In the two-party communication case, data authentication can be achieved through a purely symmetric mechanism: The sender and the receiver share a secret key to compute a message authentication code (MAC) of all communicated data. When a message with a correct MAC arrives, the receiver knows that it must have been sent by the sender.

This style of authentication cannot be applied to a broadcast setting, without placing much stronger trust assumptions on the network nodes. If one sender wants to send authentic data to mutually untrusted receivers, using a symmetric MAC is insecure: Any one of the receivers knows the MAC key, and hence could impersonate the sender and forge messages to other receivers. Hence, we need an asymmetric mechanism to achieve authenticated broadcast. One of our contributions is to construct authenticated broadcast from symmetric primitives only, and introduce asymmetry with delayed key disclosure and one-way function key chains.

7.3.3 Data Freshness

Sensor networks send measurements over time, so it is not enough to guarantee confidentiality and authentication; we also must ensure each message is *fresh*. Informally, data freshness implies that the data is recent, and it ensures that no adversary replayed old messages. We identify two types of freshness: weak freshness, which provides partial message ordering, but carries no delay information; and strong freshness, which provides a total order on a request-response pair, and allows the receiver to put an upper bound on the delay. Weak freshness is useful for sensor measurements, while strong freshness is needed for time synchronization within the network.

7.4 Additional Notation

In addition to the notation we introduce in Chapter 2, we use the following notation to describe security protocols and cryptographic operations.

A, B are principals, such as communicating nodes.

N_A is a nonce generated by A (a nonce is an unpredictable bit string, usually used to achieve freshness).

$\mathcal{X}_{AB}$ denotes the master secret (symmetric) key which is shared between A and B. No direction information is stored in this key, so we have $\mathcal{X}_{AB} = \mathcal{X}_{BA}$.

K_{AB} and K_{BA} denote the secret encryption keys shared between A and B. A and B derive the encryption key from the master secret key $\mathcal{X}_{AB}$ based on the direction of the communication: $K_{AB} = F_{\mathcal{X}_{AB}}(1)$ and $K_{BA} = F_{\mathcal{X}_{AB}}(3)$, where F is a pseudo-random function, as we defined in Section 2.2.5.[2] Section 7.6 further describes the details of key derivation.

K'_{AB} and K'_{BA} denote the secret MAC keys shared between A and B. A and B derive the encryption key from the master secret key $\mathcal{X}_{AB}$ based on the direction of the communication: $K'_{AB} = F_{\mathcal{X}_{AB}}(2)$ and $K'_{BA} = F_{\mathcal{X}_{AB}}(4)$, where F is a pseudo-random function.

$\{M\}_{K_{AB}}$ is the encryption of message M with the encryption key K_{AB}.

$\{M\}_{\langle K_{AB}, IV\rangle}$ denotes the encryption of message M, with key K_{AB}, and the initialization vector IV which is used in encryption modes such as cipher-block chaining (CBC), output feedback mode (OFB), or counter mode (CTR) [DH79, LRW00, MvOV97].

$\mathrm{MAC}(K'_{AB}, M)$ denotes the computation of the message authentication code (MAC) of message M, with MAC key K'_{AB}.

By a *secure channel*, we mean a channel that offers confidentiality, data authentication, integrity, and freshness.

7.5 SNEP and μTESLA

To achieve our security requirements, we present two security protocols: SNEP and μTESLA. SNEP provides data confidentiality, two-party data authentication, integrity, and freshness. μTESLA provides authentication for data broadcast. We bootstrap the security for both mechanisms with a shared secret key between each node and the base station (see Section 7.2). We demonstrate in Section 7.8 how we can extend the trust to node-to-node interactions from the node-to-base-station trust.

7.5.1 SNEP: Data Confidentiality, Authentication, and Freshness

SNEP provides a number of unique advantages. First, it has low communication overhead; it only adds 8 bytes per message. Second, like many cryptographic protocols it uses a counter, but we avoid transmitting the counter value

[2] To uniquely define K_{AB} and K_{BA}, the identifiers A and B of $\mathcal{X}_{AB}$ are lexicographically sorted.

by keeping state at both end points. Third, SNEP achieves semantic security, a strong security property which prevents eavesdroppers from inferring the message content from the encrypted message (see discussion below). Finally, the same simple and efficient protocol also gives us data authentication, replay protection, and weak message freshness.

Data confidentiality is one of the most basic security primitives and it is used in almost every security protocol. A simple form of confidentiality can be achieved through encryption, but pure encryption is not sufficient. Another important security property is *semantic security*, which ensures that an eavesdropper has no information about the plaintext, even if it sees multiple encryptions of the same plaintext [GM84]. For example, even if an attacker has an encryption of a 0 bit and an encryption of a 1 bit, it will not help it distinguish whether a new encryption is an encryption of 0 or 1. A basic technique to achieve this is randomization: Before encrypting the message with a chaining encryption function (i.e., DES-CBC), the sender precedes the message with a random bit string. This prevents the attacker from inferring the plaintext of encrypted messages if it knows plaintext-ciphertext pairs encrypted with the same key.

Sending the randomized data over a wireless channel, however, requires more energy. So we construct another cryptographic mechanism that achieves semantic security with no additional transmission overhead. We use two counters shared by the parties (one for each direction of communication) for the block cipher in counter mode (CTR) (as we discuss in Section 7.6). A traditional approach to manage the counters is to send the counter along with each message. But since we are using sensors and the communicating parties share the counter and increment it after each block, the sender can save energy by sending the message without the counter. At the end of this section we describe a counter exchange protocol, which the communicating parties use to synchronize (or re-synchronize) their counter values. To achieve two-party authentication and data integrity, we use a message authentication code (MAC).

A good security design practice is not to re-use the same cryptographic key for different cryptographic primitives; this prevents any potential interaction between the primitives that might introduce a weakness. Therefore we derive independent keys for our encryption and MAC operations. The two communicating parties A and B share a master secret key $\mathcal{X}_{AB}$, and they derive independent keys using the pseudo-random function F: encryption keys $K_{AB} = F_{\mathcal{X}}(1)$ and $K_{BA} = F_{\mathcal{X}}(3)$ for each direction of communication, and

MAC keys $K'_{AB} = F_{\mathcal{X}}(2)$ and $K'_{BA} = F_{\mathcal{X}}(4)$ for each direction of communication. Figure 7.4 on Page 169 shows how these keys are derived.

The combination of these mechanisms form our Sensor Network Encryption Protocol SNEP. The encrypted data has the following format: $E = \{D\}_{\langle K,C\rangle}$, where D is the data, the encryption key is K, and the counter is C. The MAC is $M = \mathsf{MAC}(K', C \mid\mid E)$. The complete message that A sends to B is:

$$A \rightarrow B: \quad \{D\}_{\langle K_{AB},C_A\rangle},\ \mathsf{MAC}(K'_{AB}, C_A \mid\mid \{D\}_{\langle K_{AB},C_A\rangle}) \quad (7.1)$$

SNEP offers the following properties:

- *Semantic security*: Since the counter value is incremented after each message, the same message is encrypted differently [BDJR97, LRW00]. The counter value is sufficiently long enough to never repeat within the lifetime of the node.
- *Data authentication*: If the MAC verifies correctly, a receiver knows that the message originated from the claimed sender.
- *Replay protection*: The counter value in the MAC prevents replay of old messages. Note that if the counter were not present in the MAC, an adversary could easily replay messages.
- *Weak freshness*: If the message verifies correctly, a receiver knows that the message must have been sent after the previous message it received correctly (that had a lower counter value). This enforces a message ordering and yields weak freshness.
- *Low communication overhead*: The counter state is kept at each end point and does not need to be sent in each message.[3]

Plain SNEP provides weak data freshness only, because it only enforces a sending order on the messages within node B, but no absolute assurance to node A that a message was created by B in response to an event in node A.

Node A achieves strong data freshness for a response from node B through a nonce N_A (which is a random number so long that exhaustive search of all possible nonces is not feasible). Node A generates N_A randomly and sends it along with a request message R_A to node B. The simplest way to achieve

[3] If the MAC does not match, the receiver can try a fixed, small number of counter increments to recover from message loss. If this still fails, the two parties engage in the counter exchange protocol we describe below.

strong freshness is for B to return the nonce with the response message R_B in an authenticated protocol. However, instead of returning the nonce to the sender, we can optimize the process by using the nonce implicitly in the MAC computation. The entire SNEP protocol providing strong freshness for B's response is:

$$\begin{aligned} A \rightarrow B: &\quad N_A, R_A \\ B \rightarrow A: &\quad \{R_B\}_{\langle K_{BA}, C_B \rangle}, \mathsf{MAC}(K'_{BA}, N_A \,||\, C_B \,||\, \{R_B\}_{\langle K_{BA}, C_B \rangle}) \end{aligned}$$

If the MAC verifies correctly, node A knows that node B generated the response after it sent the request. The first message can also use plain SNEP (as described in Protocol 7.1) if confidentiality and data authentication are needed.

7.5.1.1 Counter Exchange Protocol

To achieve small SNEP messages, we assume that the communicating parties A and B know each other's counter values C_A and C_B and thus the counter does not need to be added to each encrypted message. In practice, however, messages might get lost and the shared counter state can become inconsistent. We now present protocols to synchronize the counter state. To bootstrap the counter values initially, we use the following protocol:

$$\begin{aligned} A \rightarrow B: &\quad C_A \\ B \rightarrow A: &\quad C_B, \mathsf{MAC}(K'_{BA}, C_A \,||\, C_B) \\ A \rightarrow B: &\quad \mathsf{MAC}(K'_{AB}, C_A \,||\, C_B) \end{aligned}$$

Note that the counter values are not secret, so we do not need encryption. However, this protocol needs strong freshness, so both parties use their counters as a nonce (assuming that the protocol never runs twice with the same counter values, hence incrementing the counters if necessary). Also note that the MAC does not need to include the names of A or B, since the MAC keys K'_{AB} and K'_{BA} implicitly bind the message to the parties, and ensure the direction of the message.

If party A realizes that the counter C_B of party B is not synchronized any more, A can request the current counter of B using a nonce N_A to ensure strong freshness of the reply:

$$A \rightarrow B: \quad N_A$$
$$B \rightarrow A: \quad C_B, \ \mathsf{MAC}(K'_{BA}, N_A \,||\, C_B)$$

To prevent a potential denial-of-service (DoS) attack, where an attacker keeps sending bogus messages to lure the nodes into performing counter synchronization, the nodes can switch to sending the counter with each encrypted message they send. Another approach to detect such a DoS attack is to attach another short MAC to the message that does not depend on the counter.

7.5.2 μTESLA: Authenticated Broadcast

The TESLA protocol we present in Chapter 3 provides efficient authenticated broadcast. However, TESLA is not designed for such limited computing environments as we encounter in sensor networks for three reasons.

TESLA authenticates the initial packet with a digital signature. Clearly, digital signatures are too expensive to compute on our sensor nodes, since even fitting the code into the memory is a major challenge. For the same reason as we mention above, one-time signatures are a challenge to use on our nodes.

Standard TESLA has an overhead of approximately 24 bytes per packet. For networks connecting workstations this is usually not significant. Sensor nodes, however, send very small messages that are around 30 bytes long. It is simply impractical to disclose the TESLA key for the previous intervals with every packet: with 64 bit keys and MACs, the TESLA-related part of the packet would constitute over 50% of the packet.

Finally, the one-way key chain does not fit into the memory of our sensor node. So pure TESLA is not practical for a node to broadcast authenticated data.

We design μTESLA to solve the following inadequacies of TESLA in sensor networks:

- TESLA authenticates the initial packet with a digital signature, which is too expensive for our sensor nodes. μTESLA uses only symmetric mechanisms.
- Disclosing a key in each packet requires too much energy for sending and receiving. μTESLA discloses the key once per time interval.
- It is expensive to store a one-way key chain in a sensor node. μTESLA restricts the number of authenticated senders.

7.5.2.1 μTESLA Overview

We give a brief overview of μTESLA, followed by a detailed description.

As we discussed in Section 7.3, authenticated broadcast requires an asymmetric mechanism, otherwise any compromised receiver could forge messages from the sender. Unfortunately, asymmetric cryptographic mechanisms have high computation, communication, and storage overhead, making their usage on resource-constrained devices impractical. μTESLA overcomes this problem by introducing asymmetry through a delayed disclosure of symmetric keys, which results in an efficient broadcast authentication scheme.

We first explain μTESLA for the case where the base station broadcasts authenticated messages to the nodes. Later we discuss the case where the nodes broadcast authenticated messages.

μTESLA requires that the base station and nodes be loosely time synchronized, and each node knows an upper bound on the maximum synchronization error. To send an authenticated packet, the base station computes a MAC on the packet with a key that is secret at that point in time. When a node gets a packet, it can verify that the corresponding MAC key was not yet disclosed by the base station (based on its loosely synchronized clock, its maximum synchronization error, and the time schedule at which keys are disclosed). Since a receiving node is assured that the MAC key is known only by the base station, the receiving node is assured that no adversary could have altered the packet in transit. The node stores the packet in a buffer. At the time of key disclosure, the base station broadcasts the verification key to all receivers. When a node receives the disclosed key, it can verify the correctness of the key (which we explain below). If the key is correct, the node can now use it to authenticate the packet stored in its buffer.

Each MAC key is a key of a key chain, generated by a public one-way function F. To generate the one-way key chain, the sender chooses the last key K_n of the chain randomly, and repeatedly applies F to compute all other keys: $K_i = F(K_{i+1})$. Each node can easily perform time synchronization and retrieve an authenticated key of the key chain for the commitment in a secure and authenticated manner, using the SNEP building block. (We explain more details below.)

Example Figure 7.1 shows the μTESLA one-way key chain derivation, the time intervals, and some sample packets that the sender broadcasts. Each key of the key chain corresponds to a time interval and all packets sent within one time interval are authenticated with the same key. In this example, the sender discloses keys two time intervals after it uses them to compute MACs. We

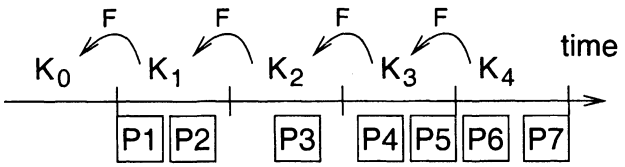


Figure 7.1. This figure shows the μTESLA one-way key chain. The sender generates the one-way key chain right-to-left by repeatedly applying the one-way function F. The sender associates each key of the one-way key chain with a time interval. Time runs left-to-right, so the sender uses the keys of the key chain in reverse order, and computes the MAC of the packets of a time interval with the key of that time interval. In contrast to the key chain that Figure 3.1 shows, the μTESLA key chain uses the key chain keys directly to compute the MAC of the packet (to reduce the overhead for the nodes).

assume that the receiver node is loosely time synchronized and knows K_0 (a commitment to the key chain). Packets P_1 and P_2 sent in interval 1 contain a MAC with key K_1. Packet P_3 has a MAC using key K_2. So far, the receiver cannot authenticate any packets yet. Assume that packets P_4, P_5, and P_6 are all lost, as well as the packet that discloses key K_1, so the receiver can still not authenticate P_1, P_2, or P_3. In interval 4 the base station broadcasts key K_2, which the node authenticates by verifying $K_0 = F(F(K_2))$. The node derives $K_1 = F(K_2)$, so it can authenticate packets P_1, P_2 with K_1, and P_3 with K_2. ■

Key disclosure is independent from the packets broadcast, and is tied to time intervals. In μTESLA, the sender broadcasts the current key periodically in a special packet.

7.5.2.2 μTESLA Detailed Description

μTESLA has multiple phases: Sender setup, sending authenticated packets, bootstrapping new receivers, and authenticating packets. We first explain how μTESLA allows the base station to broadcast authenticated information to the nodes, and we then explain how TESLA allows nodes to broadcast authenticated messages.

Sender setup The sender first generates a sequence of secret keys (a one-way key chain). To generate a one-way key chain of length n, the sender chooses the last key K_n randomly, and generates the remaining values by successively applying a one-way function F: $K_j = F(K_{j+1})$.

Broadcasting authenticated packets Time is divided into uniform time intervals, and the sender associates each key of the one-way key chain with one

time interval. In time interval i, the sender uses the key of the current interval, K_i, to compute the message authentication code (MAC) of packets in that interval. In time interval $(i+d)$, the sender reveals key K_i. The key disclosure time delay is on the order of a few time intervals, as long as it is greater than any reasonable round trip time between the sender and the receivers.

Bootstrapping a new receiver In a one-way key chain, keys are self-authenticating. The receiver can easily and efficiently authenticate subsequent keys of the one-way key chain using one authenticated key. For example, if a receiver has an authenticated value K_i of the key chain, it can easily authenticate K_{i+1}, by verifying $K_i = F(K_{i+1})$. To bootstrap μTESLA, each receiver needs to have one authentic key of the one-way key chain as a commitment to the entire chain. Other requirements are that the sender and receiver be loosely time synchronized, and that the receiver knows the key disclosure schedule of the keys of the one-way key chain. Both the loose time synchronization and the authenticated key chain commitment can be established with a mechanism providing strong freshness and point-to-point authentication. A receiver R sends a nonce N_R in the request message to the sender S. The sender S replies with a message containing its current time T_S, a key K_i of the one-way key chain used in a past interval i (the commitment to the key chain), the starting time T_i of interval i, the duration T_{int} of a time interval, and the disclosure delay d (the last three values describe the key disclosure schedule).

$$
\begin{aligned}
M \rightarrow S: &\quad N_M \\
S \rightarrow M: &\quad T_S \,||\, K_i \,||\, T_i \,||\, T_{\text{int}} \,||\, d \\
&\quad \mathsf{MAC}(K_{MS}, N_M \,||\, T_S \,||\, K_i \,||\, T_i \,||\, T_{\text{int}} \,||\, d)
\end{aligned}
$$

Since we do not need confidentiality, the sender does not need to encrypt the data. The MAC uses the secret key shared by the node and base station to authenticate the data, the nonce N_M allows the node to verify freshness. Instead of using a digital signature scheme as in TESLA, we use the node-to-base-station authenticated channel to bootstrap the authenticated broadcast.

Authenticating broadcast packets When a receiver receives the packets with the MAC, it needs to ensure that the packet is not a spoof from an adversary. The adversary already knows the disclosed key of a time interval, so it could forge the packet since it knows the key used to compute the MAC. We say that the receiver needs to be sure that the packet is *safe* — i.e., that the sender did not yet disclose the key that was used to compute the MAC of an incoming packet. As stated above, the sender and receivers need to be loosely

time synchronized and the receivers need to know the key disclosure schedule. If the incoming packet is safe, the receiver stores the packet (it can verify it only once the corresponding key is disclosed). If the incoming packet is not safe (the packet had an unusually long delay), the receiver needs to drop the packet, since an adversary might have altered it.

As soon as the node receives a new key K_i, it authenticates the key by checking that it matches the last authentic key it knows K_v, using a small number of applications of the one-way function F: $K_v = F^{i-v}(K_i)$. If the check is successful, the new key K_i is authentic and the receiver can authenticate all packets that were sent within the time intervals v to i. The receiver also replaces the stored K_v with K_i.

Nodes broadcasting authenticated data New challenges arise if a node broadcasts authenticated data. Since the node is memory limited, it cannot store the keys of a one-way key chain. Moreover, re-computing each key from the initial generating key K_n is computationally expensive. Also, the node might not share a key with each receiver, so sending out the authenticated commitment to the key chain would involve an expensive node-to-node key agreement. Finally, broadcasting the disclosed keys to all receivers is expensive for the node and drains precious battery energy.

Here are two solutions to the problem:

- The node broadcasts the data through the base station. It uses SNEP to send the data in an authenticated way to the base station, which subsequently broadcasts it.

- The node broadcasts the data. However, the base station keeps the one-way key chain and sends keys to the broadcasting node as needed. To conserve energy for the broadcasting node, the base station can also broadcast the disclosed keys, and/or perform the initial bootstrapping procedure for new receivers.

7.6 Implementation

Because of stringent resource constraints on the sensor nodes, implementation of the cryptographic primitives is a major challenge. We can sacrifice some security to achieve feasibility and efficiency, but we still need a core of strong cryptography. Below we discuss how we provide strong cryptography despite restricted resources.

Memory size is a constraint: our sensor nodes have 8K bytes of read-only program memory, and 512 bytes of RAM. The program memory is used for

TinyOS, our security infrastructure, and the actual sensor net application. To save program memory we implement all cryptographic primitives from one single block cipher [MvOV97, Sch96].

Block cipher We evaluated several algorithms for use as a block cipher. An initial choice was the AES algorithm Rijndael [DR99]; however, after further inspection, we sought alternatives with smaller code size and higher speed. The baseline version of Rijndael uses over 800 bytes of lookup tables which is too large for our memory-deprived nodes. An optimized version of that algorithm (about a 100 times faster) uses over 10K bytes of lookup tables. Similarly, we rejected the DES block cipher which requires a 512-entry SBox table and a 256-entry table for various permutations [Nat77]. A small encryption algorithm such as TEA [WN94] is a possibility, but is has not yet been subject to cryptanalytic scrutiny.[4] We use RC5 [Riv94] because of its small code size and high efficiency. RC5 does not rely on multiplication and does not require large tables. However, RC5 does use 32-bit data-dependent rotates, which are expensive on our Atmel processor (it only supports an 8-bit single bit rotate operation).

Even though the RC5 algorithm can be expressed succinctly, the common RC5 libraries are too large to fit on our platform. With a judicious selection of functionality, we use a subset of RC5 from OpenSSL, and after further tuning of the code we achieve an additional 40% reduction in code size.

Encryption function To save code space, we use the same function for both encryption and decryption. The counter (CTR) mode of block ciphers (Figure 7.2) has this property. CTR mode is a stream cipher. Therefore the size of the ciphertext is exactly the size of the plaintext and not a multiple of the block size.[5] This property is particularly desirable in our environment. Message sending and receiving consume a lot of energy. Also, longer messages have a higher probability of data corruption. Therefore, block cipher message expansion is undesirable. CTR mode requires a counter for proper operation. Reusing a counter value severely degrades security. In addition, CTR-mode offers semantic security: the same plaintext sent at different times is encrypted into different ciphertext since the encryption pads are generated from different counters. To an adversary who does not know the key, these messages will

[4]TREYFER [Yuv97] by Yuval is a small and efficient cipher, but Biryukov and Wagner describe an attack on it [BW99].

[5]The same property can be achieved with a block cipher and the *ciphertext-stealing* method described by Schneier [Sch96]. The downside is that Schneier's approach requires both encryption and decryption functions.

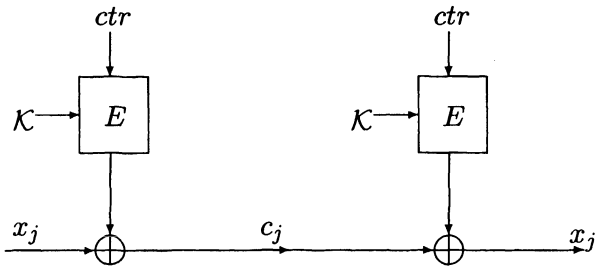


Figure 7.2. Counter mode encryption and decryption. The encryption function is applied to a monotonically increasing counter to generate a one time pad. This pad is then XORed with the plaintext. The decryption operation is identical.

appear as two unrelated random strings. Since the sender and the receiver share the counter, we do not need to include it in the message. If the two nodes lose the synchronization of the counter, they can simply transmit the counter explicitly to resynchronize using SNEP with strong freshness.

Freshness Weak freshness is automatically provided by the CTR encryption. Since the sender increments the counter after each message, the receiver verifies weak freshness by verifying that received messages have a monotonically increasing counter. For applications requiring strong freshness, the sender creates a random nonce N_M (an unpredictable 64-bit value) and includes it in the request message to the receiver. The receiver generates the response message and includes the nonce in the MAC computation (see Section 7.5.1). If the MAC of the response verifies successfully, the node knows that the response was generated after it sent the request message and hence achieves strong freshness.

Random-number generation The node has its own sensors, wireless receiver, and scheduling process, from which we could derive random digits. But to minimize power requirements, we use a MAC function as our pseudo-random number generator (PRG), with the secret pseudo-random number generator key $\mathcal{X}_{rand}$. We also keep a counter C that we increment after each pseudo-random block we generate. We compute the C-th pseudo-random output block as $\mathsf{MAC}(\mathcal{X}_{rand}, C)$. If C wraps around (which should never happen because the node will run out of energy first), we can derive a new PRG key from the master secret key and the current PRG key using our MAC as a pseudo-random function (PRF): $\mathcal{X}_{rand} = \mathsf{MAC}(\mathcal{X}, \mathcal{X}_{rand})$.

Message authentication We also need a secure message authentication code. Because we intend to re-use our block cipher, we use the well-known

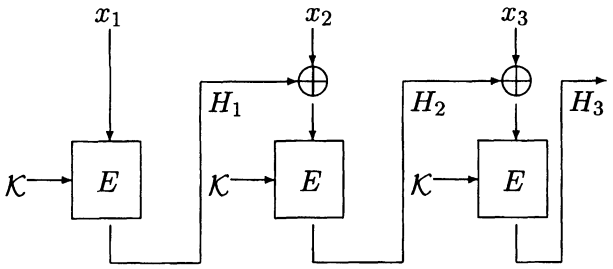


Figure 7.3. CBC MAC. The output of the last stage serves as the authentication code.

CBC-MAC [Nat80]. A block diagram for computing CBC MAC is shown in Figure 7.3.

To achieve authentication and message integrity we use the following standard approach. Assuming a message M, an encryption key $\mathcal{K}$, and a MAC key $\mathcal{K}'$, we use the following construction: $\{M\}_{\mathcal{K}}, \mathsf{MAC}(\mathcal{K}', \{M\}_{\mathcal{K}})$. This construction prevents the nodes from decrypting erroneous ciphertext, which is a potential security risk.

In our implementation, we decided to compute a MAC per packet. This approach fits well with the lossy nature of communications within this environment. Furthermore, at this granularity, the MAC is used to check both authentication and integrity of messages, eliminating the need for mechanisms such as CRC.

Key setup Recall that our key setup depends on a secret master key, initially shared by the base station and the node. We call that shared key $\mathcal{X}_{AS}$ for node A and base station S. All other keys are bootstrapped from the initial master secret key. Figure 7.4 shows our key derivation procedure. We use the pseudo-random function (PRF) F to derive the keys, which we implement as $F_K(x) = \mathsf{MAC}(K, x)$. Again, this allows for more code reuse. Because of cryptographic properties of the MAC, it must also be a good pseudo-random function. All keys derived in this manner are computationally indistinguishable. Even if the attacker could break one of the keys, the knowledge of that key would not help it find the master secret or any other key. Additionally, if we detect that a key has been compromised, both parties can derive a new key without transmitting any confidential information.

7.7 Evaluation

We evaluate the implementation of our protocols by code size, RAM size, and processor and communication overhead.

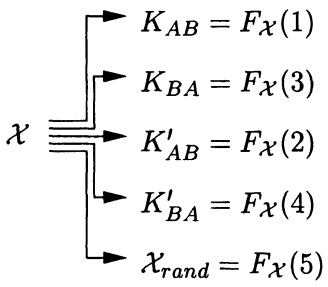


Figure 7.4. This figure shows how node A derives internal keys from the master secret to communicate with node B.

Version	Total Size	MAC	Encrypt	Key Setup
Smallest	1580	580	402	598
Fastest	1844	728	518	598
Original	2674	1210	802	686

Table 7.2. Code size breakdown (in bytes) for the security modules.

Code size Table 7.2 shows the code size of three implementations of crypto routines in TinyOS. The smallest version of the crypto routines occupies about 20% of the available code space. The difference between the fastest and the smallest implementation stems from two different implementations of the variable rotate function. The μTESLA protocol uses another 574 bytes. Together, the crypto library and the protocol implementation consume about 2K bytes of program memory, which is acceptable in most applications.

It is important to identify reusable routines to minimize call setup costs. For example, OpenSSL implements RC5 encryption as a function. On our sensor hardware, the code size of call setup and return outweigh the code size of the body of the RC5 function. We implement RC5 as a macro and only expose interfaces to the MAC and CTR-ENCRYPT functions.

Performance The performance of the cryptographic primitives is adequate for the bandwidth supported by the current generation of network sensors. Key setup is relatively expensive ($\approx$ 4 ms). In contrast, the fast version of the code uses 1.28 ms to encrypt a 16 byte message and to compute the MAC (the smaller but slower version takes 1.69 ms). Let us compare these time figures against the speed of our network. Our radio operates at 10 kilobits per second at the physical layer. If we assume that we communicate at this rate, we can

Operation	Time in milliseconds fast implementation	Time in milliseconds small implementation
Encrypt (16 bytes)	1.10	1.69
MAC (16 bytes)	1.28	1.63
Key Setup	3.92	3.92

Table 7.3. Performance of security primitives in TinyOS.

perform a key setup, an encryption, and a MAC for every message we send out.[6] Table 7.3 lists the performance number in more detail.

In our implementation, μTESLA discloses the key after two intervals ($d = 2$). The stringent buffering requirements also dictate that we cannot drop more than one key disclosure beacon. We require a maximum of two key setup operations and two CTR encryptions to check the validity of a disclosed TESLA key. Additionally, we perform up to two key setup operations, two CTR encryptions, and up to four MAC operation to check the integrity of a TESLA message.[7] That gives an upper bound of 17.8 milliseconds for checking the buffered messages. This amount of work is easily performed on our processor. In fact, the limiting factor on the bandwidth of authenticated broadcast traffic is the amount of buffering we can dedicate on individual sensor nodes. Table 7.4 shows the memory size required by the security modules. We configure the μTESLA protocol with four messages: the disclosure interval dictates a buffer space of three messages just for key disclosure, and we need an additional buffer to use this primitive in a more flexible way. Despite allocating minimal amounts of memory to μTESLA, the protocols we implement consume half of the available memory, and we cannot afford any more memory.

Energy costs We examine the energy costs of security mechanisms. Most energy costs will come from extra transmissions required by the protocols.

Table 7.5 lists the energy costs of computation and communication for the SNEP protocol. The energy costs are computed for 30 byte packets. The energy overhead for the transmission dominates energy overhead for computation. Since we use a stream cipher for encryption, the size of encrypted message is the same as the size of the plaintext. The MAC adds 8 bytes to a

[6]The data rate available to the application is significantly smaller, due to physical layer encoding, forward error correction, media access protocols, and packet format overheads.

[7]Key setup operations are dependent on the minimal and maximal disclosure interval, but the number of MAC operations depends on the number of buffered messages.

Module	RAM size (bytes)
RC5	80
TESLA	120
Encrypt/MAC	20

Table 7.4. RAM requirements for security modules.

71%	Data transmission
20%	MAC transmission
7%	Nonce transmission (for freshness)
2%	MAC and encryption computation

Table 7.5. Energy costs of adding security protocols to the sensor network. Most of the overhead arises from the transmission of extra data rather than from any computational costs.

message. But, because the MAC gives us integrity guarantees, we do not need an extra 2 bytes of CRC, so the net overhead is only 6 bytes. The transmission of these 6 bytes requires 20% of the total energy for a 30 byte packet, as Table 7.5 shows.

Messages broadcast using μTESLA have the same costs of authentication per message. Additionally, μTESLA requires a periodic key disclosure, but these messages are combined with routing updates. We can take two views regarding the costs of these messages. If we accept that the routing beacons are necessary, then μTESLA key disclosure is nearly free, because energy of transmitting or receiving dominate the computational costs of our protocols. On the other hand, one might claim that the routing beacons are not necessary and that it is possible to construct an *ad hoc* multihop network implicitly. In that case the overhead of key disclosure would be one message per time interval, regardless of the traffic pattern within the network. We believe that the benefits of authenticated routing justify the costs of explicit beacons.

Remaining security issues Although this protocol suite addresses many security related problems, there remain many additional issues. First, we do not address the problem of information leakage through covert channels. Second, we do not deal completely with compromised sensors, we merely ensure that compromising a single sensor does not reveal the keys of all the sensors in the network. Third, we do not deal with denial-of-service (DoS) attacks in this work. Since we operate on a wireless network, an adversary can always

perform a DoS attack by jamming the wireless channel with a strong signal. Finally, due to our hardware limitations, we cannot provide Diffie-Hellman style key agreement or use digital signatures to achieve non-repudiation. For the majority of sensor network applications, authentication is sufficient.

7.8 Application of SNEP: Node-to-Node Key Agreement

In this section we demonstrate how we can design a two-party key agreement protocol using SNEP.

A convenient technology for bootstrapping secure connections is to use public key cryptography protocols for symmetric key setup [BM93, HC98]. Unfortunately, our resource constrained sensor nodes prevent us from using computationally expensive public key cryptography. We need to construct our protocols solely from symmetric key algorithms. We design a symmetric protocol that uses the base station as a trusted agent for key setup.

Assume that the node A wants to establish a shared secret session key SK_{AB} with node B. Since A and B do not share any secrets, they need to use a trusted third party S, which is the base station in our case. In our trust setup, both A and B share a master secret key with the base station, $\mathcal{X}_{AS}$ and $\mathcal{X}_{BS}$, respectively. The following protocol achieves secure key agreement as well as strong key freshness:

$$A \rightarrow B : N_A, A$$
$$B \rightarrow S : N_A, N_B, A, B, \mathsf{MAC}(K'_{BS}, N_A \,||\, N_B \,||\, A \,||\, B)$$
$$S \rightarrow A : \{SK_{AB}\}_{K_{SA}}, \mathsf{MAC}(K'_{SA}, N_A \,||\, B \,||\, \{SK_{AB}\}_{K_{SA}})$$
$$S \rightarrow B : \{SK_{AB}\}_{K_{SB}}, \mathsf{MAC}(K'_{SB}, N_B \,||\, A \,||\, \{SK_{AB}\}_{K_{SB}})$$

The protocol uses our SNEP protocol with strong freshness. The nonces N_A and N_B ensure strong key freshness to both A and B. The SNEP protocol ensures confidentiality (through encryption with the keys K_{AS} and K_{BS}) of the established session key SK_{AB}, as well as message authentication (through the MAC using keys K'_{AS} and K'_{BS}), so we are sure that the key was really generated by the base station. Note that the MAC in the second protocol message helps defend the base station from denial-of-service attacks, and the base station only sends two messages to A and B if it received a legitimate request from one of the nodes.

A nice feature of the above protocol is that the base station performs most of the transmission work. Many other protocols involve a ticket that the server

sends to one of the parties which forwards it to the other node, which requires more energy for the nodes to forward the message.

The Kerberos key agreement protocol achieves similar properties, but it does not provide strong key freshness [KN93, MNSS88]. If Kerberos used SNEP with strong freshness, then Kerberos would have greater security.

Chapter 8

RELATED WORK

We first review related work that addresses broadcast security in general. We then consider related work on the following more specialized topics: broadcast authentication protocols, broadcast signatures, one-time signatures, and key distribution protocols for large dynamic groups.

8.1 General Broadcast Security

Canetti et al. present a taxonomy of multicast security [CGI+99]. They discuss the different group characteristics such as group size and member characteristics (computation power, memory, etc.), membership dynamics (static, dynamic) and life cycle (join and leave distribution), group life time, sender profile and number of senders, traffic profile (bandwidth, delay between messages, packet size, allowed latency, burstiness), centralized or decentralized groups, and the routing algorithm that is used. For the security requirements they consider secrecy (ephemeral or long-term), authenticity (group or source), anonymity, access control, service availability. The paper also proposes a new mechanism for source authentication and an extension to LKH for efficient group key distribution, which we review below.

Moyer, Rao, and Rohatgi discuss security issues in Multicast communication [MRR99]. They overview group key management, source authentication, and stream signing techniques.

Hardjono and Tsudik discuss several multicast security issues [HT00]. In the core problem area they discuss data confidentiality and integrity, source authentication, group key management, and security policies. In the infrastructure problem area they discuss the security of multicast routing protocols,

A. Perrig et al., *Secure Broadcast Communication*

and the security of reliable multicast protocols. In the complex applications problem area they consider distributed group key generation, group and member certification, robustness against attacks, among others.

The Group Security Research Group (GSEC), formerly known as Secure Multicast Research Group (SMuG), in the Internet Research Task Force (IRTF) is actively researching security in group communication [GSE02]. The current focus of GSEC is group policy, decentralized groups, investigating security for reliable multicast, etc. The Multicast Security (MSEC) working group in the IETF develops Internet standards for multicast security protocols [MSE02].

Security policies for multicast groups is a challenging problem because policy negotiation is hard when multiple parties are present. We refer to the work of McDaniel et al. for further details on group security policies [McD01, MPH99, HCM01]. The policy approach we take in our work is that a central group controller defines the group security policy and new group members need to accept these policies when they join the group. We refer to the work of McDaniel et al. on how to deal with the problem of policy definition.

8.2 Broadcast Authentication

Researchers proposed information-theoretically secure broadcast authentication mechanisms [Sim90, DY91, DF92, DFY92, FKK96b, KO97, SNW98, SNW99]. These protocols have a high overhead in large groups with many receivers. The following protocols in the complexity-theoretic model can scale to large groups.

Gennaro and Rohatgi motivate and present techniques for signing digital streams [GR97]. We review this work in the following section on broadcast signatures.

Cheung [Che97] proposes a scheme akin to the basic TESLA protocol to authenticate link-state routing updates between routers. He assumes that all the routers in a network are time synchronized up to $\pm\epsilon$, and does not consider the case of heterogeneous receivers.

Briscoe proposes the FLAMeS protocol, which is similar to the Cheung's protocol [Che97] and also to the basic TESLA protocol. Bergadano, Cavalino, and Crispo present an authentication protocol for multicast [BCC00b]. Their protocol is also similar to Cheung's protocol [Che97] and to the basic TESLA protocol.

In seminal work, Anderson et al. present the Guy Fawkes protocol which provides message authentication between two parties [ABC+98]. Their protocol does not tolerate packet loss. They propose two methods to guarantee

that the keys are not revealed too soon. The first method is that the sender and receiver are in lockstep, i.e., the receiver acknowledges every packet before the sender can send the next packet. This limits the sending rate and does not scale to a large number of receivers. The second method to secure their scheme is to time-stamp each packet at a time-stamping service.

Canetti et al. construct a multicast source authentication scheme [CGI+99]. Their solution is to use k different keys to authenticate every message with k different MAC's. Every receiver knows m keys and can hence verify m MAC's. The keys are distributed in such a way that no coalition of w receivers can forge a packet for a specific receiver. The security of their scheme depends on the assumption that at most a bounded number (which is on the order of k) of receivers collude.

Bergadano, Cavagnino, and Crispo, propose a protocol similar to the Guy Fawkes protocol to individually authenticate data streams sent within a group [BCC00a]. Their scheme requires that the sender receives an acknowledgment packet from each receiver before it can send the next packet. Such an approach prevents scalability to a large group. The advantage is that their protocol does not rely on time synchronization.

Unfortunately, their protocol is vulnerable to a man-in-the-middle attack. To illustrate the attack, we briefly review the protocol for one sender and one receiver:

$$
\begin{aligned}
&B \rightarrow A : KB_0, SN, \{KB_0, SN\}_{K_B^{-1}} \\
&A \rightarrow B : A_1, MAC(KA_1, A_1), KA_0, SN, \{KA_0, SN\}_{K_A^{-1}} \\
&B \rightarrow A : KB_1 \\
&A \rightarrow B : A_2, MAC(KA_2, A_2), KA_1
\end{aligned}
$$

In their scheme, both A (the sender) and B (the receiver) pre-compute a key chain, KA_i and KB_i, respectively. In the following attack, B intends to authenticate data from A, but we will show that the attacker I can forge all data. The attacker I captures all messages from B and it can pretend to B that all the messages come from A. To A, the attacker I just pretends to be itself.

$$
\begin{aligned}
B \rightarrow I(A) &: KB_0, SN, \{KB_0, SN\}_{K_B^{-1}} \\
I \rightarrow A &: KI_0, SN, \{KI_0, SN\}_{K_I^{-1}} \\
A \rightarrow I &: A_1, MAC(KA_1, A_1), KA_0, SN, \{KA_0, SN\}_{K_A^{-1}} \\
I \rightarrow A &: KI_1 \\
A \rightarrow I &: A_2, MAC(KA_2, A_2), KA_1 \\
I(A) \rightarrow B &: A_1', MAC(KA_1, A_1'), KA_0, SN, \{KA_0, SN\}_{K_A^{-1}}
\end{aligned}
$$

Note that the attacker I can forge the content of the message A_1 sent to B, because it knows the key KI_0. The attacker I can forge the entire subsequent message stream, without B noticing.

Another attack is that an eavesdropper that records a message exchange between A (sender) and B (receiver) can impersonate either A or B as a receiver to another sender C. This attack can be serious if the sender performs access control based on the initial signature packet and the revealed key chain. The attack is simple, the eavesdropper only needs to replay the initial signatures and all the disclosed keys collected.

Boneh, Durfee, and Franklin show that one cannot build a short collusion resistant multicast MAC without relying on digital signatures [BDF01] or on time synchronization. They prove that any secure multicast MAC with length slightly less than the number of receivers can be converted into a signature scheme.

8.3 Broadcast Signature

Gennaro and Rohatgi are arguably the fathers of the stream signature field. Their paper was the first to introduce a technique for signing digital streams [GR97]. They present two different schemes, one for the off-line case (the entire stream content is known in advance) and the other for the on-line case (real-time streams). For the off-line case, they suggest signing the first packet and embedding in each packet P_i the hash of the next packet P_{i+1} (including the hash stored in P_{i+1}). While this method is elegant and provides for a stream signature, it does not tolerate packet loss. The on-line scheme solves this problem through a regular signature of the initial packet and embedding the public key of the Lamport one-time signature scheme in each packet, which is used to sign the subsequent packet. The limitations are that this scheme is not robust against packet loss, and the high communication overhead.

Wong and Lam address the problem of data authenticity and integrity for delay-sensitive and lossy multicast streams [WL98]. They propose to use Merkle's signature trees to sign streams. Their idea to make asymmetric digital signatures more efficient is to amortize one signature generation and verification over multiple messages. Merkle describes how to construct a hash tree over all messages where the signer only digitally signs the root [Mer90, Mer80] (see Section 2.4.2). However, to make this scheme robust against packet loss, every packet needs to contain the signature along with all the nodes necessary to compute the root. In practice, this scheme adds around 200 bytes to each packet (assuming a 1024 bit RSA signature and a signature tree over 16 packets). This approach also requires sender-side buffering and hence introduces a time delay.

Rohatgi improves the one-time signature approach proposed by Winternitz and Merkle [Mer88, Mer90], refined by Even, Goldreich, and Micali [EGM90], and proposes the k-times signature algorithm [Roh99]. One public allows k signatures, which increases the robustness to packet loss over [GR97] (the public key for the following k packets is included and signed in each packet). The per-packet overhead is about 300 bytes. The signer computes approximately 350 off-line one-way functions per packet, and the verifier computes 184 one-way functions on average per packet.

Golle and Modadugu independently developed a protocol [GM01] similar to EMSS. Miner and Staddon propose to combine TESLA and EMSS/MESS, and analyze the robustness of their protocols in a graph-theoretical setting [MS01]. Their work advanced the analysis of EMSS-like protocols and stimulated our research.

8.4 Digital Signatures Based on One-way Functions without Trapdoors

Rabin proposed one of the first one-time signature algorithms [Rab78]. The scheme requires interaction between the signer and the verifier. The validation parameters and the signature are on the order of 1 Kbyte.

Lamport shows how to construct a digital signature out of a one-way function [Lam79]. His approach does not require interaction between the signer and verifier, however, the size of the validation parameters and signature are still on the order of 1 Kbyte. Improvements are by Winternitz (described by Merkle [Mer88, Mer90]), Even, Goldreich, and Micali [EGM90]. Rohatgi further refined Winternitz's approach and proposes the k-times signature scheme [Roh99]. We give more details above.

Bleichenbacher and Maurer analyzed signature algorithms with a minimal number of nodes in the graph [BM94, BM96b, BM96a].

Zhang also proposes to use the elements of multiple one-way chains to construct multiple one-time signature instances [Zha98]. His approach allows to efficiently authenticate link state routing updates.

Similar to the MicroMint payment scheme by Rivest and Shamir [RS97], the security of the BiBa signature comes from the difficulty of finding k-way collisions for a one-way function. The main difference, however, is the security assumption: MicroMint assumes that the bank has more computational resources than an adversary, but BiBa enjoys exponentially increasing security such that it is secure even if the signer only has modest computation resources.[1]

Table 8.1 compares the various one-time signature algorithms. We consider the computation and communication overhead as a basis for comparison. We choose the signature parameters such that a forger has a probability of 2^{-80} to find a valid signature after one try. For the computation overhead, we consider the number of one-way function computations the signer needs to perform to compute the public key (off-line), and the expected number of one-way function computations the signer performs to actually generate the signature (on-line). For the verification overhead we list the expected number of one-way function computations the verifier performs to check the signature. For the computation overhead, we consider the size of the public key, and the size of a signature. We express the signature and public key size in number of nodes. In practice, each node may be on the order of 96–128 bits long.[2]

8.5 Small-Group Key Agreement

Many researchers worked on the problem of providing secure group communication among a small number of participants [AG00, AST00, AST98, BW98, Ber91, BD93, BD95, BD97, JV96, Gon97, KPT00, Per99, RBH+98, RBD00a, RBD00b, STW96, STW98, SSDW90, STW00, TT00, TT01]. For the majority of this work, the participants use asymmetric cryptographic primitives to compute a common group key. Most of these protocols do not re-

[1] MicroMint gains an additional computational advantage because the bank can pre-compute coins, and an adversary has a small, limited time to forge coins. Hence MicroMint requires very loose time synchronization.

[2] Recently, we were made aware of an exciting new improvement based on BiBa, [RR02]. This paper by Reyzin and Reyzin is based on our BiBa work and is called the HORS signature. If added to the table above, HORS would do even better than BiBa, with only a generation cost of only one hash function, a verification cost of only 11 hash function computations, and a signature size of only 10 hashes. The values are otherwise the same as BiBa.

	Signature Generation		Verification	Signature size	Public key size
	Off-line	On-line	(expected)		
Lamport	160	1	80	80	160
Merkle-Winternitz	355	1	169	23	1
Bleichenbacher-Maurer	182	1	72	45	1
BiBa	1024	2048	23	11	1024

Table 8.1. Comparison of one-time signature algorithms. The table considers a signature of an 80-bit hash. For the Merkle-Winternitz signature, we use the parameters that Rohatgi proposes to sign 80 bits [Roh99].

Units are measured in number of hash function computations for signature generation and verification, and in the number of hash values for signature size and public key size.

quire a central server, and the group model usually assumes that each member is a sender and a receiver. In this book, we consider the case where one sender broadcasts information to a large number of receivers, and the following section discusses the related work on the key distribution protocols needed to achieve confidentiality in that setting.

8.6 Large-Group Key Distribution

Harney and Muckenhirn introduced GKMP [HM97a, HM97b], a centralized server approach that distributes group keys through unicast. Mittra's Iolus aims for scalability through distributed hierarchical key servers [Mit97]. Molva and Pannetrat involve routers on the multicast distribution path into the security [MP99, MP00].

The logical key tree hierarchy was independently discovered by Wallner et al. [WHA99], and by Wong, Gouda, and Lam [WGL97, WGL98]. This approach is usually referred to as the Logical Key Hierarchy (LKH), which Section 6.2 reviews.

An optimization that halves the size of the key update message is described by Canetti et al. [CGI+99]. We call this variant LKH++ and we review it in Section 6.2.2.

To reduce the overhead of join, the current approach is to simply compute a one-way function on each key that the new member obtains, which was suggested by Radia Perlman at a presentation in the IETF, and later also by the Versakey framework [CWS+99].

Another method to halve the size of the key update message is the "One-way Function Tree" protocol (OFT) by Balenson, McGrew and Sherman [BMS99].

In OFT, the key of the node is derived from the two sibling keys. We review OFT in Section 6.3.

Even though the LKH++ protocol greatly diminishes the overhead of a group key change to $O(\log(N))$ (where N is the number of group members), the constant key change with the resulting key update messages can still result in an unscalable protocol in large dynamic groups. If members join and leave frequently, the resulting key update traffic can overwhelm the group. Chang et al. [CEK+99], and Setia, Koussih, and Jajodia [SKJ00] propose to batch re-keying messages, which results in an aggregating members that join or leave during a short time interval. They do not address the issue of reliability of key updates, however. Unfortunately, the protocol proposed by Chang et al. [CEK+99] is vulnerable to a collusion attack.

Briscoe designed the MARKS protocol [Bri99]. MARKS is scalable and does not require any key update messages, but the protocol only works if the leaving time of the member is fixed when the member joins the group, so members cannot be expelled.

Trappe et al. [TSPL01] also observed that key updates should be distributed with data. In fact, they propose that the key updates be embedded within the data, for example using techniques similar to watermarking. The focus of their work, however, is different. While the ELK key updates encode the same key as was used to encrypt the data, [TSPL01] proposed embedding future keys in current data.

Wong and Lam designed Keystone [WL00, YLZL01, LYGL01], which addresses the reliable delivery for key update messages. Since most reliable multicast transport protocols do not scale well to large groups, they propose that the key server uses forward error correction (FEC) to encode the key update message. As long as the member receives a sufficient fraction of the key update packets, it can reconstruct the information. If too many packets are lost, the member uses a unicast connection to the key server to recover the missing keys. Since the authors assume independent packet loss, this scheme is quite effective. In practice, however, the packet loss in the Internet is correlated, which means that the probability of loss for a packet increases drastically if the previous packet was lost [YMKT99, BSUB98]. This means that even though the key update packets are replicated, sending them in close succession introduces considerable vulnerability against correlated packet loss.

Analyses of the lower bound of the size of key updates and key server storage are by Poovendran et al. [Poo99, PB99, LPB01], by Canetti, Malkin, and Naor [CMN99], and by Snoeyink, Suri, and Varghese [SSV01].

Poovendran and Baras [PB99], and Selcuk, McCubbin, and Sidhu [SMS00] also consider an optimized placement of members within the key tree to minimize the key update overhead after a member leave event. However, their methods minimize the size of the key update message, but TST eliminates the key update altogether.

Broadcast encryption by Fiat and Naor [FN94] is another model for distributing a shared key to a group of receivers. Subsequent papers further develop this approach [BC95, BMS96, LS98, GSW00].

Chapter 9

CONCLUSION

Chapters 1 through 8 present a variety of powerful techniques for making broadcast communication secure. They examine requirements for broadcast security and give real protocols that achieve these goals.

- Chapter 3 presents TESLA, the most efficient protocol for broadcast authentication. If the receivers are loosely time synchronized with the sender, the per-packet overhead is around 20 bytes, the computation overhead is about one MAC function computation for both the sender and receiver, the protocol tolerates packet loss, and the receivers can authenticate each packet after a brief time delay. Chapter 7 discusses the viability of Tesla even for resource-starved sensor network nodes. These properties make TESLA a viable authentication protocol for the majority of applications. One drawback of TESLA is that authentication has slightly delayed in the general case, although if highly accurate time synchronization is available, TIK ("TESLA with Instant Key disclosure") suffices.

- Chapter 4 presents the BiBa broadcast authentication protocol. For real-time applications intolerant to authentication delay on platforms without accurate time synchronization, BiBa provides instant authentication, so neither the sender nor the receiver need to buffer data. The BiBa broadcast authentication protocol still requires loose time synchronization, and has a higher computation and communication overhead than TESLA.

- Chapter 5 presents HTSS, a signature technology, for the case where time synchronization is impossible (or if data is cached), or if we want a signature on every packet of the stream. The HTSS protocol is particularly

A. Perrig et al., *Secure Broadcast Communication*

efficient if the sender knows the entire content in advance. In HTSS, the receivers only need to verify one digital signature, and they can subsequently verify each packet instantly after a few hash function computations. To sign real-time streams, we design the EMSS and MESS protocols. These protocols have a slightly higher overhead than HTSS, but the main drawback is that the signature verification is delayed; the receiver has to receive a signature packet before it can verify the previous packets.

- Chapter 6 presents access control — the sender distributes a group key to all legitimate receivers and encrypts all messages with the group key. This reduces the problem of confidentiality and access control to the problem of key distribution. We find that missing reliability in key distribution prevents scalability. To achieve a highly scalable protocol, we improve scalability by trading off sender computation, receiver computation, and increased unicast bandwidth (which however is directly amortized because fewer receivers send key update requests by unicast). The combination of all these mechanisms constitutes the ELK protocol.

Together, these chapters present a cookbook of security recipes for broadcast authentication, signature, and key distribution. But, they also suggest many interesting research problems. Below, we briefly conclude our book with a list of a few of the open problems generated by this research.

9.1 Open Problems

- All methods we propose for broadcast authentication require time synchronization. Can we build an secure, efficient, real-time, and scalable broadcast authentication mechanism that is robust to packet loss and that does not require time synchronization?

- One approach to the previous open question would be to design an efficient digital signature algorithm with small signature size. Current algorithms have high overhead. Is there some inherent reason why low-overhead digital signature algorithms seem not to exist?

- How far can the TESLA protocol be extended beyond broadcast authentication? For example, can TESLA can be used as a central component for securing (ad hoc) routing protocols?

- We can imagine using TESLA in a network with n mutually communicating nodes. In such a system TESLA could be used to authenticate point-to-point communication, with the advantage that only n keys would need to be

set up instead of $n \cdot (n-1)/2$ pair-wise keys. What optimizations are possible in such a system? How can we build a high-reliability fault-tolerant system, where TESLA replaces digital signatures to ensure authenticity of broadcast messages?

- It is often desirable to make sure that messages are all received at precisely the same time. Consider, for example, a global trading system where even a few milliseconds discrepancy in receiving a buying opportunity could unfairly advantage one party. Can we modify TESLA to address this concern?

- Many of our protocols, in particular TESLA, rely on loose time synchronization. TIK depends on accurate time synchronization. Can the time synchronization function be integrated with TESLA is such a way that the resulting system is more robust or more efficient than implementing them separately?

- BiBa signatures introduce a new approach for designing an asymmetric cryptographic primitive: the collision-based approach. Can we extend this to develop other cryptographic primitives? Can we develop hybrid approaches which combine collision analysis with number theoretic or algebraic geometric primitives (such as elliptic curve techniques)?

- Can we make BiBa signatures more efficient? Can we reduce the size of the public key? What is the theoretical limit of the BiBa signature system?

- If a verifier already has a set of committed values, and the signing party knows the pre-images of those values, can we exploit this fact to improve BiBa?

- Our analysis for the MESS and HTSS stream signatures only considers independent packet loss. Can we extend this to correlated packet loss?

- Many streams have packets with mutual dependencies, such as MPEG-4 streams. Can we build special MESS link structures where the packets have mutual dependencies?

- The ELK key distribution protocol suggest many novel approaches to reduce the key update overhead and increase reliability of large-group key distribution protocols. Engineering and testing these mechanisms remains opens, as does adapting them to particular individual settings. What techniques exist for specialized use of ELK in access control?

- ELK greatly reduces the broadcast communication overhead, but is still be expensive in highly dynamic groups where many members unexpectedly leave. Can we build an efficient large scale key distribution problem for this case?

- What about receivers that temporarily leave the group for a short time period? Consider, for example, receivers that do not receive any packets during one minute due to a network outage. In highly dynamic groups, these receivers would all have to contact the key server. Can we modify ELK so receivers can re-synchronize their key tree without contacting the key server?

- Sensor networks present many challenges, both in the area of applications and in underlying technology. One problem that we are especially concerned with is building data-level privacy mechanisms. Can we build privacy mechanisms that can support applications such as emergency response without subjecting users to "Big Brother" style monitoring?

We hope that we have inspired you, dear reader, to think about the opportunities and challenges in making broadcast networks secure. We hope that you might try your hand at some of the open problems listed above, or the many other open problems suggested by this text. And when it comes time to implement applications, may all your broadcasts be efficient and secure.

Chapter 10

GLOSSARY

Active adversary: An active adversary not only eavesdrops on all communication (as does a passive adversary), but also sends messages that are either freshly generated or derived or replayed from previous messages.

Asymmetric cryptography: Most cryptographic primitives belong to one of two broad categories: *symmetric* and *asymmetric* cryptography. In *asymmetric cryptography* we normally use pairs of keys — one key is public and one key is private (or secret). RSA is an example of asymmetric cryptography [RSA78]. In RSA digital signatures, the sender uses the private key to digitally sign a message, and the receiver can verify the signature with the public key. Also see **symmetric cryptography**.

Backward Secrecy: Backward secrecy guarantees that a passive adversary who knows a subset of group keys cannot discover previous group keys. This property ensures that a new member who joins the group cannot learn any previous group key.

BiBa: The BiBa (bins and balls) signature is a new combinatorial signature. The BiBa broadcast authentication protocol is based upon the BiBa signature. Chapter 4 describes BiBa in detail.

Broadcast communication: In broadcast communication, a sender sends information to a set of receivers.

A. Perrig et al., *Secure Broadcast Communication*

Commitment function: A commitment function allows us to lock-in a secret s without revealing s. For more details consult Section 2.4.

Early leave event: See **join event**.

EIKU: Entropy injection key update, part of the ELK key distribution protocol. EIKU has many features. First, it enables to trade off security with key update size, and second, it allows for small key updates that we call hints. These techniques substantially decrease the size of key updates.

EMSS: The efficient multicast stream signature (EMSS) signs a sequence of packets using chained hash values and periodically signed packets. EMSS uses fixed hash links, and MESS uses randomized hash links. See Sections 5.1 and 5.2 for details.

ET: Evolving tree protocol, part of the ELK key distribution protocol. We propose that the key server periodically evolves the entire tree (by computing a one-way function on all keys in the key tree). This technique has the advantage that a member join event does not require any broadcast message.

Forward Secrecy: Forward secrecy guarantees that a passive adversary who knows a contiguous subset of old group keys cannot discover subsequent group keys. This property ensures that a member cannot learn future group keys after it leaves the group.

Group Key Secrecy: Group key secrecy guarantees that it is computationally infeasible for a passive adversary to discover any group key.

HTSS: The hash tree stream signature (HTSS) signs a stream of messages. The main advantages of HTSS are the low overhead (computation and communication), robustness to packet loss, and instant verification. However, HTSS requires that the sender knows the entire stream in advance. Section 5.4 describes HTSS.

Join event: In a group key distribution setting, members may join or leave the group of receivers at any time. If a member leaves the group at the time it predicted to leave (or at the time the server predicted), we call the leave event a predicted leave event. If the member leaves the group before the predicted

time, we call this an early leave event. Similarly, a late leave event is when a member leaves after the predicted leave time.

Key disclosure delay: In the TESLA protocol family, the key disclosure delay is the maximum time between the first usage of a key and the time at which that key becomes public. The key disclosure delay is usually expressed as a number of time intervals. For more details, see Section 3.2.4 on Page 33.

Key independence: Key independence guarantees that a passive adversary who knows all but one group keys cannot discover the group key it misses.

Leave event: See **join event**.

MESS: See **EMSS**.

Message authentication code: A Message Authentication Code (MAC) is used for data authentication and integrity. A MAC is a cryptographic checksum, which anybody with the secret key can generate or verify. Section 2.2.4 discusses MACs in more detail.

Non-repudiation of origin: Non-repudiation of origin for a message means that the producer of a message cannot deny that it generated that message. This property is often achieved with a secure digital signature (see also Section 2.2.1).

One-way chain: To commit to a sequence of random values, we repeatedly use a commitment scheme to generate a one-way chain.

Passive adversary: A passive adversary can eavesdrop on all communication. See also **active adversary**.

Point-to-point communication: In point-to-point communication, one sender sends information to a single receiver.

Predicted leave event: See **join event**.

Symmetric cryptography: In *symmetric cryptography* the sender and receiver share the same (secret) key. Symmetric cryptography is based on shared keys.

Symmetric cryptographic functions are in general 3–4 orders of magnitude faster than asymmetric cryptographic functions (see Section 2.3, and also see **asymmetric cryptography**).

TST: Time-structured tree protocol, part of the ELK key distribution protocol. If the key server places new members in specific parts of the tree, a member leave event that occurs at a predicted time does not require any broadcast message. Hence, only unpredicted leave events require a broadcast key update message.

VIB: Very important bits, part of the ELK key distribution protocol. In protocols that use hierarchical key trees, some keys in the key update message are needed by most members to update their group key. We identify those keys and propose key update messages that only contain the VIBs, which can greatly reduce the communication overhead.

References

[ABC+98] R. Anderson, F. Bergadano, B. Crispo, J. Lee, C. Manifavas, and R. Needham. A new family of authentication protocols. *ACM Operating Systems Review*, 32(4):9–20, October 1998.

[AG00] N. Asokan and Philip Ginzboorg. Key-agreement in ad-hoc networks. *Computer Communications*, 23(17):1627–1637, November 2000.

[ALN97] M. Abadi, T. Lomas, and R. Needham. Strengthening passwords. SRC Technical Note 1997 - 033, December, Systems Research Center, December 1997.

[AMS97] R. Anderson, C. Manifavas, and C. Sutherland. NetCard – a practical electronic cash system. In *Security Protocols—International Workshop*, volume 1189 of *Lecture Notes in Computer Science*, pages 49–57. Springer-Verlag, Berlin Germany, April 1997.

[AST98] G. Ateniese, M. Steiner, and G. Tsudik. Authenticated group key agreement and friends. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 17–26. ACM Press, November 1998. A journal version of this paper appeared later in JSAC [AST00].

[AST00] G. Ateniese, M. Steiner, and G. Tsudik. New multiparty authentication services and key agreement protocols. *IEEE Journal on Selected Areas in Communications*, 18(4):628–639, April 2000.

[Atk95] R. Atkinson (editor). IP encapsulating security payload (ESP). Internet Request for Comment RFC 1827, Internet Engineering Task Force, August 1995. obsoleted by [KA98a].

[Atm02] Secure Microcontrollers for SmartCards. `http://www.atmel.com/atmel/acrobat/1065s.pdf`, 2002.

[ATW97] N. Asokan, G. Tsudik, and M. Waidner. Server-supported signatures. *Journal of Computer Security*, 5(1):91–108, 1997.

[BC95] C. Blundo and A. Cresti. Space requirements for broadcast encryption. In *Advances in Cryptology – EUROCRYPT '94*, volume 950 of *Lecture Notes in Computer Science*, pages 287–298. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1995.

[BCC88] G. Brassard, D. Chaum, and C. Crépeau. Minimum disclosure proofs of knowledge. *Journal of Computer and System Sciences*, 37(2):156–189, October 1988.

[BCC00a] F. Bergadano, D. Cavagnino, and B. Crispo. Chained stream authentication. In *Selected Areas in Cryptography, 7th Annual International Workshop, SAC 2000*, volume 2012 of *Lecture Notes in Computer Science*, pages 144–157. Springer-Verlag, Berlin Germany, August 2000.

[BCC00b] F. Bergadano, D. Cavalino, and B. Crispo. Individual single source authentication on the mbone. In *2000 IEEE International Conference on Multimedia and Expo, ICME 2000*, pages 541–544, August 2000. A talk containing this work was given at IBM T. J. Watson Research Laboratory, August 1998.

[BCH+00] M. Brown, D. Cheung, D. Hankerson, J. Hernandez, M. Kirkup, and A. Menezes. PGP in constrained wireless devices. In *Proceedings of the 9th USENIX Security Symposium*, pages 247–261. USENIX, August 2000.

[BCK96] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology – CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1996.

[BCK97] M. Bellare, R. Canetti, and H. Krawczyk. HMAC: Keyed-hashing for message authentication. Internet Request for Comment RFC 2104, Internet Engineering Task Force, February 1997.

[BD93] M. Burmester and Y. Desmedt. Towards practical "proven secure" authenticated key distribution. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 228–231. ACM Press, November 1993.

[BD95] M. Burmester and Y. Desmedt. A secure and efficient conference key distribution system. In *Advances in Cryptology – EUROCRYPT '94*, volume 950 of *Lecture Notes in Computer Science*, pages 275–286. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1995.

[BD97] M. Burmester and Y. Desmedt. Efficient and secure conference key distribution. In *Security Protocols—International Workshop*, volume 1189 of *Lecture Notes in Computer Science*, pages 119–129. Springer-Verlag, Berlin Germany, April 1997.

[BDF01] D. Boneh, G. Durfee, and M. Franklin. Lower bounds for multicast message authentication. In *Advances in Cryptology – EUROCRYPT '2001*, volume

2045 of *Lecture Notes in Computer Science*, pages 434–450. Springer-Verlag, Berlin Germany, 2001.

[BDJR97] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption: Analysis of the DES modes of operation. In *Proceedings of the 38th Symposium on Foundations of Computer Science (FOCS)*, pages 394–403. IEEE Computer Society Press, 1997.

[Bel00] S. Bellovin. The ICMP traceback message. `http://www.research.att.com/~smb`, 2000.

[Ber91] S. Berkovits. How to broadcast a secret. In *Advances in Cryptology – EUROCRYPT '91*, volume 547 of *Lecture Notes in Computer Science*, pages 535–541. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1991.

[BFH+00] J. Byers, M. Frumin, G. Horn, M. Luby, M. Mitzenmacher, A. Roetter, and W. Shaver. FLID-DL: Congestion control for layered multicast. In *Second International Workshop on Networked Group Communication (NGC 2000)*, Palo Alto, California, November 2000.

[BHK+99] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 216–233. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1999.

[BLMR98] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A Digital Fountain approach to reliable distribution of bulk data. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 56–67, 1998.

[BM92] S. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 72–84. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press, May 1992.

[BM93] S. Bellovin and M. Merritt. Augmented encrypted key exchange: A password-based protocol secure against dictionary atttacks and password file compromise. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 244–250. ACM Press, November 1993.

[BM94] D. Bleichenbacher and U. Maurer. Directed acyclic graphs, one-way functions and digital signatures. In *Advances in Cryptology – CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 75–82. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1994.

[BM96a] D. Bleichenbacher and U. Maurer. On the efficiency of one-time digital signatures. In *Advances in Cryptology – ASIACRYPT '96*, volume 1163 of *Lecture Notes in Computer Science*, pages 196–209. Springer-Verlag, Berlin Germany, 1996.

[BM96b] D. Bleichenbacher and U. Maurer. Optimal tree-based one-time digital signature schemes. In *13th Symposium on Theoretical Aspects of Computer Science (STACS'96)*, volume 1046 of *Lecture Notes in Computer Science*, pages 363–374. Springer-Verlag, Berlin Germany, 1996.

[BMS96] C. Blundo, L. Mattos, and D. Stinson. Trade-offs between communication and storage in unconditionally secure schemes for broadcast encryption and interactive key distribution. In *Advances in Cryptology – CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 387–400. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1996.

[BMS99] D. Balenson, D. McGrew, and A. Sherman. Key management for large dynamic groups: One-way function trees and amortized initialization. Internet Draft, Internet Engineering Task Force, March 1999.

[BR93] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 62–73. ACM Press, November 1993.

[BR97] M. Bellare and P. Rogaway. Collision-resistant hashing: Towards making UOWHFs practical. In *Advances in Cryptology – CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 470–484. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1997.

[Bra88] G. Brassard. *Modern Cryptology*, volume 325 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin Germany, 1988.

[Bri99] B. Briscoe. MARKS: Zero side-effect multicast key management using arbitrarily revealed key sequences. In *First International Workshop on Networked Group Communication*, pages 301–320, November 1999.

[BSUB98] M. Borella, D. Swider, S. Uludag, and G. Brewster. Internet packet loss: Measurement and implications for end-to-end QoS. In *1998 ICPP Workshop on Architectural and OS Support for Multimedia Applications Flexible Communication Systems*, pages 3–12. IEEE, August 1998.

[BW98] K. Becker and U. Wille. Communication complexity of group key distribution. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 1–6. ACM Press, November 1998.

[BW99] A. Biryukov and D. Wagner. Slide attacks. In *Proceedings of the 6th International Workshop on Fast Software Encryption*, volume 1636 of *Lecture*

Notes in Computer Science, pages 245–259. Springer-Verlag, Berlin Germany, March 1999.

[CEK+99] I. Chang, R. Engel, D. Kandlur, D. Pendarakis, and D. Saha. Key management for secure internet multicast using boolean function minimization techniques. In *Proceedings IEEE Infocomm'99*, volume 2, pages 689–698, March 1999.

[CER97] CERT Coordination Center. CERT advisory CA-1997-28 IP denial-of-service attacks. Technical Report CA-1997-28, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, December 1997.

[CER01a] CERT Coordination Center. CERT advisory CA-2001-19 "Code Red" worm exploiting buffer overflow in IIS indexing service DLL. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, July 2001.

[CER01b] CERT Coordination Center. CERT advisory CA-2001-23 continued threat of the "Code Red" worm. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, July 2001.

[CGI+99] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast security: A taxonomy and some efficient constructions. In *INFOCOMM'99*, pages 708–716, March 1999.

[CGP01] N. Courtois, L. Goubin, and J. Patarin. Flash, a fast multivariate signature algorithm. In *Progress in Cryptology - CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 298–307. Springer-Verlag, Berlin Germany, April 2001.

[Che97] S. Cheung. An efficient message authentication scheme for link state routing. In *13th Annual Computer Security Applications Conference*, pages 90–98, 1997.

[CJ02] D. Coppersmith and M. Jakobsson. Almost optimal hash sequence traversal. In *Proceedings of the Fourth Conference on Financial Cryptography (FC '02)*, Lecture Notes in Computer Science. International Financial Cryptography Association (IFCA), Springer-Verlag, Berlin Germany, 2002.

[CMN99] R. Canetti, T. Malkin, and K. Nissim. Efficient communication-storage tradeoffs for multicast encryption. In *Advances in Cryptology – EUROCRYPT '99*, volume 1599 of *Lecture Notes in Computer Science*, pages 459–474. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1999.

[CRC00] R. Canetti, P. Rohatgi, and P. Cheng. Multicast data security transformations: Requirements, considerations, and prominent choices. Internet draft, Internet Engineering Task Force, 2000.

[CW79] L. Carter and M. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[CWI99] Security of e-commerce threatened by 512-bit number factorization. `http://www.cwi.nl/~kik/persb-UK.html`, August 1999. CWI press release.

[CWS+99] G. Caronni, M. Waldvogel, D. Sun, N. Weiler, and B. Plattner. The VersaKey framework: Versatile group key management. *IEEE Journal on Selected Areas in Communications*, 17(9):1614–1631, September 1999.

[DA99] T. Dierks and C. Allen. The TLS protocol version 1.0. Internet Request for Comment RFC 2246, Internet Engineering Task Force, January 1999. Proposed Standard.

[Dal01] iButton: A Java-Powered Cryptographic iButton. `http://www.ibutton.com/ibuttons/java.html`, 2001.

[DBP96] H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: A strengthened version of RIPEMD. In *Proceedings of the 3rd International Workshop on Fast Software Encryption*, volume 1039 of *Lecture Notes in Computer Science*, pages 71–82. Springer-Verlag, Berlin Germany, 1996.

[DF92] Y. Desmedt and Y. Frankel. Shared generation of authenticators and signatures. In *Advances in Cryptology – CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 457–469. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1992.

[DF02] Digital fountain corporation. `http://www.digitalfountain.com`, 2002.

[DFY92] Y. Desmedt, Y. Frankel, and M. Yung. Multi-receiver / multi-sender network security: Efficient authenticated multicast / feedback. In *Proceedings IEEE Infocom '92*, pages 2045–2054, 1992.

[DH79] W. Diffie and M. Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE*, 67(3):397–427, March 1979.

[DN93] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology – CRYPTO '92*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1993.

[DP00] R. Dhamija and A. Perrig. Déjà Vu: A user study. using images for authentication. In *Proceedings of the 9th USENIX Security Symposium*, pages 45–58. USENIX, August 2000.

[DR99] J. Daemen and V. Rijmen. AES proposal: Rijndael, March 1999.

[DvOW92] W. Diffie, P. van Oorschot, and M. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, pages 107–125, 1992.

[DY91] Y. Desmedt and M. Yung. Arbitrated unconditionally secure authentication can be unconditionally protected against arbiter's attacks. In *Advances in Cryptology – CRYPTO '90*, volume 537 of *Lecture Notes in Computer Science*, pages 177–188. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1991.

[EGM90] S. Even, O. Goldreich, and S. Micali. On-line/off-line digital signatures. In *Advances in Cryptology – CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 263–277. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1990.

[FJ93] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397—413, August 1993.

[FJM+95] S. Floyd, V. Jacobson, S. McCanne, C. Liu, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. In *Proceedings of the ACM SIGCOMM 95*, pages 342–356, Boston, MA, August 1995.

[FKK96a] A. Freier, P. Kariton, and P. Kocher. The SSL protocol: Version 3.0. Internet draft, Netscape Communications, 1996.

[FKK96b] F. Fujii, W. Kachen, and K. Kurosawa. Combinatorial bounds and design of broadcast authentication. *IEICE Transactions*, E79-A(4):502–506, 1996.

[FN94] A. Fiat and M. Naor. Broadcast encryption. In *Advances in Cryptology – CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 480–491. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1994.

[GB99] S. Goldwasser and M. Bellare. Lecture notes on cryptography. Summer Course "Cryptography and Computer Security" at MIT, 1996–1999, August 1999. Available at `http://www-cse.ucsd.edu/users/mihir/papers/gb.pdf`.

[GGM86] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986.

[GM84] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer Security*, 28:270–299, 1984.

[GM01] P. Golle and N. Modadugu. Authenticating streamed data in the presence of random packet loss. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS 2001)*, pages 13–22. Internet Society, February 2001.

[Gon97] L. Gong. Enclaves: Enabling secure collaboration over the Internet. *IEEE Journal on Selected Areas in Communications*, pages 567–575, 1997.

[GR97] R. Gennaro and P. Rohatgi. How to sign digital streams. In *Advances in Cryptology – CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 180–197. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1997.

[GSE02] Group security research group (GSEC). `http://www.irtf.org/charters/gsec.html` and `http://www.securemulticast.org/smug-index.htm`, 2002. Research group in the Internet Engineering Task Force (IETF).

[GSW00] J. Garay, J. Staddon, and A. Wool. Long-lived broadcast encryption. In *Advances in Cryptology – CRYPTO '2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 333–352. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 2000.

[Hal94] N. Haller. The S/Key one-time password system. In *Proceedings of the Symposium on Network and Distributed Systems Security*, pages 151–157. Internet Society, February 1994.

[HC98] D. Harkins and D. Carrel. The Internet key exchange (IKE). Internet Request for Comment RFC 2409, Internet Engineering Task Force, November 1998.

[HCM01] H. Harney, A. Colegrove, and P. McDaniel. Principles of policy in secure groups. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS 2001)*, pages 125–135. Internet Society, February 2001.

[HH99] H. Harney and E. Harder. Logical key hierarchy protocol. Internet Draft, Internet Engineering Task Force, April 1999.

[HILL99] J. Håstad, R. Impagliazzo, L. Levin, and M. Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999. A preliminary version appeared in 21st STOC, 1989.

[HJP02] Y.-C. Hu, D. B. Johnson, and A. Perrig. Secure efficient distance vector routing in mobile wireless ad hoc networks. In *Fourth IEEE Workshop on Mobile Computing Systems and Applications* (WMCSA '02), June 2002.

[HM97a] H. Harney and C. Muckenhirn. Group key management protocol (GKMP) architecture. Internet Request for Comment RFC 2094, Internet Engineering Task Force, July 1997.

[HM97b] H. Harney and C. Muckenhirn. Group key management protocol (GKMP) specification. Internet Request for Comment RFC 2093, Internet Engineering Task Force, July 1997.

[HPJ01] Y.-C. Hu, A. Perrig, and D. B. Johnson. Wormhole detection in wireless ad hoc networks. Technical Report TR01-384, Department of Computer Science, Rice University, December 2001.

[HPJ02] Y.-C. Hu, A. Perrig, and D. B. Johnson. Ariadne: A secure on-demand routing protocol for ad hoc networks. In *Proceedings of the Eighth ACM International Conference on Mobile Computing and Networking (Mobicom 2002)*, September 2002.

[HPS01] J. Hoffstein, J. Pipher, and J. Silverman. NSS: An NTRU lattice-based signature scheme. In *Advances in Cryptology – EUROCRYPT '2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 211–228. Springer-Verlag, Berlin Germany, 2001.

[HPT97] R. Hauser, A. Przygienda, and G. Tsudik. Reducing the cost of security in link state routing. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS '97)*, pages 93–99. Internet Society, February 1997.

[HSW96] R. Hauser, M. Steiner, and M. Waidner. Micro-payments based on iKP. Research Report 2791, IBM Research, February 1996.

[HSW+00] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, November 2000.

[HT96] R. Hauser and G. Tsudik. On shopping *incognito*. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*, pages 251–257. USENIX, November 1996.

[HT00] T. Hardjono and G. Tsudik. IP multicast security: Issues and directions. *Annales de Telecom*, 2000.

[IEE97] IEEE Computer Society LAN MAN Standards Committee. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, IEEE Std 802.11-1997. The Institute of Electrical and Electronics Engineers, 1997.

[ILL88] R. Impagliazzo, L. Levin, and M. Luby. Pseudo-random generation from one-way functions. In *Proceedings of the 21th Annual Symposium on Theory of Computing (STOC)*, pages 12–24. ACM Press, 1988.

[Jak02] M. Jakobsson. Fractal hash sequence representation and traversal. Cryptology ePrint Archive, `http://eprint.iacr.org/2002/001/`, January 2002.

[JV96] M. Just and S. Vaudenay. Authenticated multi-party key agreement. In *Advances in Cryptology – ASIACRYPT '96*, volume 1163 of *Lecture Notes in Computer Science*, pages 36–49. Springer-Verlag, Berlin Germany, 1996.

[KA98a] S. Kent and R. Atkinson. IP encapsulating security payload (ESP). Internet Request for Comment RFC 2406, Internet Engineering Task Force, November 1998.

[KA98b] S. Kent and R. Atkinson. Security architecture for the Internet Protocol. Internet Request for Comment RFC 2401, Internet Engineering Task Force, November 1998.

[KKP99] J. Kahn, R. Katz, and K. Pister. Next century challenges: mobile networking for smart dust. In *International Conference on Mobile Computing and Networking (MOBICOM '99)*, pages 271–278, August 1999.

[KN93] J. Kohl and C. Neuman. The Kerberos network authentication service (V5). Internet Request for Comment RFC 1510, Internet Engineering Task Force, 1993.

[Knu98] D. Knuth. *Sorting and Searching, second edition*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1998.

[KO97] K. Kurosawa and S. Obana. Characterization of (k,n) multi-receiver authentication. In *Proceedings of the 2nd Australasian Conference on Information Security and Privacy (ACISP '97)*, volume 1270 of *Lecture Notes in Computer Science*, pages 205–215. Springer-Verlag, Berlin Germany, 1997.

[KPT00] Y. Kim, A. Perrig, and G. Tsudik. Simple and fault-tolerant key agreement for dynamic collaborative groups. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 235–244. ACM Press, November 2000.

[Kuh00] M. Kuhn. Probabilistic counting of large digital signature collections. In *Proceedings of the 9th USENIX Security Symposium*, pages 73–83. USENIX, August 2000.

[Lab95] National Institute of Standards and Technology (NIST)(Computer Systems Laboratory). Secure hash standard. Federal Information Processing Standards Publication FIPS PUB 180-1, April 1995.

[Lam75] L. Lamport. Discussion with Whitfield Diffie. `http://research.compaq.com/SRC/personal/lamport/pubs/pubs.html#dig-sig`, 1975.

[Lam79] L. Lamport. Constructing digital signatures from a one-way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, October 1979.

[Lam81] L. Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–772, November 1981.

[LMS85] L. Lamport and P. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, 1985.

[LPB01] M. Li, R. Poovendran, and C. Berenstein. Optimization of key storage for secure multicast. In *35th Annual Conference on Information Sciences and Systems (CISS)*, March 2001.

[LRW00] H. Lipmaa, P. Rogaway, and D. Wagner. Counter mode encryption. `http://csrc.nist.gov/encryption/modes/`, 2000.

[LS98] M. Luby and J. Staddon. Combinatorial bounds for broadcast encryption. In *Advances in Cryptology – EUROCRYPT '98*, volume 1403 of *Lecture Notes in Computer Science*, pages 512–526. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1998.

[Lub96] M. Luby. *Pseudorandomness and Cryptographic Applications*. Princeton Computer Society Notes, 1996.

[LV99] A. Lenstra and E. Verheul. Selecting cryptographic key sizes. `http://www.cryptosavvy.com`, November 1999. A shorter version of the report appeared in the proceedings of the Public Key Cryptography Conference (PKC2000) and in the Autumn '99 PricewaterhouseCoopers CCE newsletter. A revised version appeared later in the Journal of Cryptology.

[LV00] A. Lenstra and E. Verheul. Key improvements to XTR. In *Advances in Cryptology – ASIACRYPT '2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 220–233. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 2000.

[LV01] A. Lenstra and E. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.

[LYGL01] X. Li, Y. Yang, M. Gouda, and S. Lam. Batch rekeying for secure group communications. In *Proceedings of the tenth international World Wide Web conference on World Wide Web*, pages 525–534, October 2001.

[Man96] U. Manber. A simple scheme to make passwords based on one-way functions much harder to crack. *Computers and Security*, 15(2):171–176, 1996.

[McD01] P. McDaniel. *Policy Management in Secure Group Communication*. PhD thesis, Univerity of Michigan, 2001.

[Mer78] R. Merkle. Secure communication over insecure channels. *Communications of the ACM*, 21(4):294–299, April 1978.

[Mer80] R. Merkle. Protocols for public key cryptosystems. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 122–134. IEEE Computer Society Press, April 1980.

[Mer88] R. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology – CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1988.

[Mer90] R. Merkle. A certified digital signature. In *Advances in Cryptology – CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1990.

[Mic96] S. Micali. Efficient certificate revocation. Technical Report MIT/LCS/TM-542b, Massachusetts Institute of Technology, Laboratory for Computer Science, March 1996.

[Mil92] D. Mills. Network Time Protocol (version 3) specification, implementation and analysis. Internet Request for Comment RFC 1305, Internet Engineering Task Force, March 1992.

[Mil94] D. Mills. Improved algorithms for synchronizing computer network clocks. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications, SIGCOMM 94*, pages 317–327, London, England, 1994.

[Mit97] S. Mittra. Iolus: A framework for scalable secure multicasting. In *ACM SIGCOMM'97*, pages 277–288, September 1997.

[MKOM90] S. Miyaguchi, S. Kurihara, K. Ohta, and H. Morita. 128-bit hash function (N-hash). *NTT Review*, 2(6):128–132, 1990.

[MMO85] S. Matyas, C. Meyer, and J. Oseas. Generating strong one-way functions with cryptographic algorithm. *IBM Technical Disclosure Bulletin*, 27:5658–5659, 1985.

[MMSA+96] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[MNSS88] S. Miller, B. Neuman, J. Schiller, and J. Saltzer. Kerberos authentication and authorization system. Technical report, MIT, October 1988. Project Athena Technical Plan.

[MP99] R. Molva and A. Pannetrat. Scalable multicast security in dynamic groups. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, pages 101–112. ACM Press, November 1999.

[MP00] R. Molva and A. Pannetrat. Scalable multicast security with dynamic recipient groups. *ACM Transactions on Information and System Security*, 3(3):136–160, August 2000.

[MP02] M. Mitzenmacher and A. Perrig. Bounds and improvements for biba signature schemes. Technical Report TR-02-02, Harvard Computer Science Technical Report, 2002.

[MPH99] P. McDaniel, A. Prakash, and P. Honeyman. Antigone: A flexible framework for secure group communication. In *Proceedings of the 8th USENIX Security Symposium*, pages 99–114. USENIX, August 1999.

[MRR99] M. Moyer, J. Rao, and P. Rohatgi. A survey of security issues in multicast communications. *IEEE Network*, 13(6):12–23, November/December 1999.

[MS98] D. McGrew and A. Sherman. Key establishment in large dynamic groups using one-way function trees. Manuscript, May 1998.

[MS01] S. Miner and J. Staddon. Graph-based authentication of digital streams. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 232–246. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press, May 2001.

[MSE02] Multicast security (msec). `http://www.ietf.org/html.charters/msec-charter.html` and `http://www.securemulticast.org/msec-index.htm`, 2002. Working group within the Internet Engineering Task Force (IETF).

[Mul02] Source-Specific Multicast. `http://www.ietf.org/html.charters/ssm-charter.html`, 2002.

[MvOV97] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[Nao90] M. Naor. Bit commitment using pseudo-randomness (extended abstract). In *Advances in Cryptology – CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 128–137. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1990.

[Nat77] National Bureau of Standards (NBS). Specification for the Data Encryption Standard. Federal Information Processing Standards Publication 46 (FIPS PUB 46), January 1977.

[Nat80] National Institute of Standards and Technology (NIST). DES model of operation. Federal Information Processing Standards Publication 81 (FIPS PUB 81), December 1980.

[Nat91] National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS), Federal Register 56. Draft Tech. Rep. FIPS PUB 186, August 1991.

[NES99] NESSIE: New European Schemes for Signatures, Integrity, and Encryption. `http://www.cryptonessie.org`, 1999.

[Ope01] OpenSSL. The OpenSSL project. `http://www.openssl.org/`, 2001.

[Pax99] V. Paxson. End-to-end Internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, June 1999.

[PB99] R. Poovendran and J. Baras. An information theoretic analysis of rooted-tree based secure multicast key distribution schemes. In *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 624–638. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1999.

[PCB+02] Adrian Perrig, Ran Canetti, Bob Briscoe, J.D. Tygar, and Dawn Song. TESLA: Multicast source authentication transform introduction. Internet Draft, Internet Engineering Task Force, February 2002. Work in progress.

[PCST01] A. Perrig, R. Canetti, D. Song, and J. D. Tygar. Efficient and secure source authentication for multicast. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS 2001)*, pages 35–46. Internet Society, February 2001.

[PCTS00] A. Perrig, R. Canetti, J. D. Tygar, and D. Song. Efficient authentication and signature of multicast streams over lossy channels. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 56–73. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press, May 2000.

[PCTS02] A. Perrig, R. Canetti, J.D. Tygar, and D. Song. The tesla broadcast authentication protocol. *RSA CryptoBytes*, 5(Summer), 2002.

[Ped97] T. Pedersen. Electronic payments of small amounts. In *Security Protocols—International Workshop*, volume 1189 of *Lecture Notes in Computer Science*, pages 59–68. Springer-Verlag, Berlin Germany, April 1997.

[Per99] A. Perrig. Efficient collaborative key management protocols for secure autonomous group communication. In *International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC '99)*, July 1999.

[Per01] A. Perrig. The BiBa one-time signature and broadcast authentication protocol. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, pages 28–37. ACM Press, November 2001.

[PGV97] B. Preneel, R. Govaerts, and J. Vandewalle. Hash functions based on block ciphers: A synthetic approach. In *Advances in Cryptology – CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1997.

[PKB99] K. Pister, J. Kahn, and B. Boser. Smart dust: Wireless networks of millimeter-scale sensor nodes, 1999. In 1999 UC Berkeley Electronics Research Laboratory Research Summary.

[Poo99] R. Poovendran. *Key Management for Secure Multicast Communications*. PhD thesis, Department of Electrical and Computer Engineering, University of Maryland, College Park, 1999.

[Pre93] B. Preneel. *Analysis and design of cryptographic hash functions*. PhD thesis, Katholieke Universiteit Leuven (Belgium), 1993.

[Pro02] Proxim, Inc. Data sheet for Proxim Harmony 802.11a CardBus Card. Sunnyvale, CA. Available at: `http://www.proxim.com/products/all/harmony/docs/ds/harmony_11a_cardbus.%pdf`, 2002.

[PS98] G. Poupard and J. Stern. Security analysis of a practical "on the fly" authentication and signature generation. In *Advances in Cryptology – EUROCRYPT '98*, volume 1403 of *Lecture Notes in Computer Science*, pages 422–436. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1998.

[PS99a] A. Perrig and D. Song. Hash visualization: A new technique to improve real-world security. In *International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC '99)*, pages 131–138, July 1999.

[PS99b] G. Poupard and J. Stern. On the fly signatures based on factoring. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, pages 37–45. ACM Press, November 1999.

[PS00a] A. Perrig and D. Song. A first step towards the automatic generation of security protocols. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS '00)*, pages 73–83. Internet Society, February 2000.

[PS00b] A. Perrig and D. Song. Looking for diamonds in the desert — extending automatic protocol generation to three-party authentication and key agreement protocols. In *13th IEEE Computer Security Foundations Workshop*, pages 64–76. IEEE Computer Society Press, July 2000.

[PST01] A. Perrig, D. Song, and J. D. Tygar. ELK, a new protocol for efficient large-group key distribution. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 247–262. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press, May 2001.

[PSW+01] Adrian Perrig, Robert Szewczyk, Victor Wen, David Culler, and J. D. Tygar. SPINS: Security protocols for sensor networks. In *Seventh Annual International Conference on Mobile Computing and Networks (MobiCOM 2001)*, pages 189–199, 2001.

[PSW+02] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. D. Tygar. SPINS: Security protocols for sensor networks. *Wireless Networks*, 8(5):521–534, September 2002.

[Rab78] M. Rabin. Digitalized signatures. In *Foundations of Secure Computation*, pages 155–168. Academic Press, 1978.

[Rab90] M. Rabin. The information dispersal algorithm and its applications. In *Sequences: Combinatorics, Compression, Security and Transmission*, pages 406–419. Springer-Verlag, Berlin Germany, 1990.

[RBD00a] O. Rodeh, K. Birman, and D. Dolev. Optimized rekey for group communication systems. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS '00)*, pages 37–48. Internet Society, February 2000.

[RBD00b] O. Rodeh, K. Birman, and D. Dolev. A study of group rekeying. Technical Report TR2000-1791, Computer Science Department, Cornell University, March 2000.

[RBH+98] O. Rodeh, K. Birman, M. Hayden, Z. Xiao, and D. Dolev. The architecture and performance of security protocols in the ensemble group communication system. Technical Report TR98-1703, Computer Science Department, Cornell University, September 1998.

[RBvR94] M. Reiter, K. Birman, and R. van Renesse. A security architecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 12(4):340–371, November 1994.

[Rei93] M. Reiter. *A Security Architecture for Fault-Tolerant Systems*. PhD thesis, Department of Computer Science, Cornell University, August 1993.

[Riv92] R. Rivest. The MD5 message-digest algorithm. Internet Request for Comment RFC 1321, Internet Engineering Task Force, April 1992.

[Riv94] R. Rivest. The RC5 encryption algorithm. In *Proceedings of the 1st International Workshop on Fast Software Encryption*, volume 809 of *Lecture Notes in Computer Science*, pages 86–96. Springer-Verlag, Berlin Germany, 1994.

[Riz00] L. Rizzo. PGMCC: a TCP-friendly single-rate multicast congestion control scheme. In *SIGCOMM 2000*, pages 17–28, August 2000.

[RMTR02] Reliable Multicast Transport (RMT). `http://www.ietf.org/html.charters/rmt-charter.html`, 2002.

[Roh99] P. Rohatgi. A compact and fast hybrid signature scheme for multicast packet. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, pages 93–100. ACM Press, November 1999.

[RR02] Leonid Reyzin and Natan Reyzin. Better than biba: Short one-time signatures with fast signing and verifying. In *Seventh Australasian Conference on Information Security and Privacy (ACISP 2002)*, July 2002.

[RS60] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8:300–304, 1960.

[RS97] R. Rivest and A. Shamir. PayWord and MicroMint: Two simple micropayment schemes. In *Security Protocols—International Workshop*, volume 1189 of *Lecture Notes in Computer Science*, pages 69 – 88. Springer-Verlag, Berlin Germany, April 1997.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[RSW96] R. Rivest, A. Shamir, and D. Wagner. Time-lock puzzles and timed-release crypto, March 1996. Published at `http://theory.lcs.mit.edu/~rivest/RivestShamirWagner-timelock.ps`.

[Sch91] C. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[Sch96] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, second edition, 1996.

[Sim90] G. Simmons. A Cartesian product construction for unconditionally secure authentication codes that permit arbitration. *Journal of Cryptology*, 2(2):77–104, 1990.

[SKJ00] S. Setia, S. Koussih, and S. Jajodia. Kronos: A scalable group re-keying approach for secure multicast. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 215–228. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press, May 2000.

[SLWL90] B. Simons, J. Lundelius-Welch, and N. Lynch. An overview of clock synchronization. In B. Simons and A. Spector, editors, *Fault-Tolerant Distributed Computing*, number 448 in LNCS, pages 84–96. Springer-Verlag, Berlin Germany, 1990.

[SMS00] A. Selcuk, C. McCubbin, and D. Sidhu. Probabilistic optimization of LKH-based multicast key distribution schemes. Internet Draft, Internet Engineering Task Force, January 2000.

[SNW98] R. Safavi-Naini and H. Wang. New results on multireceiver authentication codes. In *Advances in Cryptology – EUROCRYPT '98*, volume 1403 of *Lecture Notes in Computer Science*, pages 527–541. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1998.

[SNW99] R. Safavi-Naini and H. Wang. Multireceiver authentication codes: Models, bounds, constructions and extensions. *Information and Computation*, 151(1/2):148–172, 1999.

[SP01] D. Song and A. Perrig. Advanced and authenticated marking schemes for IP traceback. In *Proceedings IEEE Infocomm 2001*, pages 878–886, April 2001.

[SSDW90] D. Steer, L. Strawczynski, W. Diffie, and M. Wiener. A secure audio teleconference system. In *Advances in Cryptology – CRYPTO '88*, volume 403 of *Lecture Notes in Computer Science*, pages 520–528. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1990.

[SSV01] J. Snoeyink, S. Suri, and G. Varghese. A lower bound for multicast key distribution. In *Proceedings IEEE Infocomm 2001*, pages 422–431, April 2001.

[STW96] M. Steiner, G. Tsudik, and M. Waidner. Diffie-Hellman key distribution extended to groups. In *Proceedings of the 3rd ACM Conference on Computer and Communications Security*, pages 31–37. ACM Press, March 1996. Appeared as revised and extended journal version as [STW00].

[STW98] M. Steiner, G. Tsudik, and M. Waidner. CLIQUES: A new approach to group key agreement. In *18th International Conference on Distributed Computing Systems (ICDCS'98)*, pages 380–387. IEEE Computer Society Press, May 1998. Appeared as heavily revised and extended journal version in [STW00].

[STW00] M. Steiner, G. Tsudik, and M. Waidner. Key agreement in dynamic peer groups. *IEEE Transactions on Parallel and Distributed Systems*, 11(8):769–780, August 2000.

[SWKA00] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. In *Proceedings of the 2000 ACM SIGCOMM Conference*, August 2000. An early version of the paper appeared as techreport UW-CSE-00-02-01 available at: `http://www.cs.washington.edu/homes/savage/traceback.html`.

[SWP00] D. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 44–55. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press, May 2000.

[Ten00] D. Tennenhouse. Embedding the Internet: Proactive computing. *Communications of the ACM*, 43(5):43–50, 2000.

[Tri02] Trimble Navigation Limited. Data sheet and specifications for Trimble Thunderbolt GPS Disciplined Clock. Sunnyvale, California. Available at `http://www.trimble.com/thunderbolt.html`,2002.

[TSPL01] W. Trappe, J. Song, R. Poovendran, and K. Liu. Key distribution for secure multimedia multicast via data embedding. In *IEEE ICASSP 2001*, May 2001.

[TT00] W. Tzeng and Z. Tzeng. Round-efficient conference-key agreement protocols with provable security. In *Advances in Cryptology – ASIACRYPT '2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 614–628. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 2000.

[TT01] W. Tzeng and Z. Tzeng. Robust key-evolving public key encryption schemes. Report 2001/009, Cryptology ePrint Archive, 2001.

[vM39] R. von Mises. Über Aufteilungs- und Besetzungswahrscheinlichkeiten. *Revue de la Faculté des Sciences de l'Université d'Istanbul*, 4:145—163, 1939.

[vRBM96] R. van Renesse, K. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[WGL97] C. Wong, M. Gouda, and S. Lam. Secure group communications using key graphs. Technical Report TR-97-23, University of Texas at Austin, Department of Computer Sciences, August 1997.

[WGL98] C. Wong, M. Gouda, and S. Lam. Secure group communications using key graphs. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 68–79, 1998. Appeared in ACM SIGCOMM Computer Communication Review, Vol. 28, No. 4 (Oct. 1998).

[WHA99] D. Wallner, E. Harder, and R. Agee. Key management for multicast: Issues and architectures. Internet Request for Comment RFC 2627, Internet Engineering Task Force, June 1999.

[Win84] R. Winternitz. A secure one-way hash function built from DES. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 88–90. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press, April 1984.

[WL98] C. Wong and S. Lam. Digital signatures for flows and multicasts. In *IEEE ICNP '98*, 1998.

[WL00] C. Wong and S. Lam. Keystone: A group key management service. In *International Conference on Telecommunications, ICT 2000*, 2000.

[WLLP01] B. Warneke, M. Last, B. Liebowitz, and K. Pister. Smart dust: Communicating with a cubic-millimeter computer. *IEEE Computer*, pages 44–51, January 2001.

[WN94] D. Wheeler and R. Needham. TEA, a tiny encryption algorithm. `http://www.ftp.cl.cam.ac.uk/ftp/papers/djw-rmn/djw-rmn-tea.html`, November 1994.

[XMZY97] X. Xu, A. Myers, H. Zhang, and R. Yavatkar. Resilient multicast support for continuous-media applications. In *IEEE 7th International Workshop on Network and Operating Systems Support for Digital Audio and Video, NOSSDAV 97*, pages 183–194, 1997.

[Yee94] B. Yee. *Using Secure Coprocessors*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1994. CMU-CS-94-149.

[YLZL01] Y. Yang, X. Li, X. Zhang, and S. Lam. Reliable group rekeying: A performance analysis. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM 2001*, pages 27–38, 2001.

[YMKT99] M. Yajnik, S. Moon, J. Kurose, and D. Towsley. Measurement and modelling of the temporal dependence in packet loss. In *Proceedings IEEE Infocom '99*, March 1999.

[YT95] B. Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. In *Proceedings of the First USENIX Workshop on Electronic Commerce*. USENIX, July 1995.

[Yuv97] G. Yuval. Reinventing the Travois: Encryption/MAC in 30 ROM bytes. In *Proceedings of the 4th International Workshop on Fast Software Encryption*, volume 1267 of *Lecture Notes in Computer Science*, pages 205–209. Springer-Verlag, Berlin Germany, 1997.

[Zha98] K. Zhang. Efficient protocols for signing routing messages. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS '98)*. Internet Society, March 1998.