SYNGRESS®

Cryptography for Developers

The Only Cryptography Guide Written for Software Developers

- Complete Coverage of Symmetric Ciphers, One-Way Hashes, Message Authentication Codes, Combined Authentication and Encryption Modes, Public Key Cryptography, and More
- Special Examples of Cryptographic Goals: Privacy, Integrity, Authentication, and Nonrepudiation
- Full Discussion of Large Integer Arithmetic, Public Key Algorithms, and Advanced Encryption

Tom St Denis, Elliptic Semiconductor Inc.
and Author of the LibTom Projects

Simon Johnson

SYNGRESS®

Cryptography for Developers

Tom St Denis, Elliptic Semiconductor Inc.
and Author of the LibTom Project

Simon Johnson

KEY	SERIAL NUMBER
001	HJIRTCV764
002	PO9873D5FG
003	829KM8NJH2
004	GPPQQW722M
005	CVPLQ6WQ23
006	VBP965T5T5
007	HJJJ863WD3E
008	2987GVTWMK
009	629MP5SDJT
010	IMWQ295T6T

PUBLISHED BY
Syngress Publishing, Inc.
800 Hingham Street
Rockland, MA 02370

Cryptography for Developers

1 2 3 4 5 6 7 8 9 0
ISBN-10: 1-59749-104-7
ISBN-13: 978-1-59749-104-4

Publisher: Andrew Williams
Acquisitions Editor: Erin Heffernan
Technical Editor: Simon Johnson
Cover Designer: Michael Kavish

Page Layout and Art: Patricia Lupien
Copy Editor: Beth Roberts
Indexer: J. Edmund Rush

Distributed by O'Reilly Media, Inc. in the United States and Canada.
For information on rights, translations, and bulk sales, contact Matt Pedersen, Director of Sales and Rights, at Syngress Publishing; email matt@syngress.com or fax to 781-681-3585.

Acknowledgments

Syngress would like to acknowledge the following people for their kindness and support in making this book possible.

Syngress books are now distributed in the United States and Canada by O'Reilly Media, Inc.The enthusiasm and work ethic at O'Reilly are incredible, and we would like to thank everyone there for their time and efforts to bring Syngress books to market:Tim O'Reilly, Laura Baldwin, Mark Brokering, Mike Leonard, Donna Selenko, Bonnie Sheehan, Cindy Davis, Grant Kikkert, Opol Matsutaro, Steve Hazelwood, Mark Wilson, Rick Brown,Tim Hinton, Kyle Hart, Sara Winge, Peter Pardo, Leslie Crandell, Regina Aggio Wilkinson, Pascal Honscher, Preston Paull, Susan Thompson, Bruce Stewart, Laura Schmier, Sue Willing, Mark Jacobsen, Betsy Waliszewski, Kathryn Barrett,John Chodacki, Rob Bullington, Kerry Beck, Karen Montgomery, and Patrick Dirden.

The incredibly hardworking team at Elsevier Science, including Jonathan Bunkell, Ian Seager, Duncan Enright, David Burton, Rosanna Ramacciotti, Robert Fairbrother, Miguel Sanchez, Klaus Beran, Emma Wyatt, Krista Leppiko, Marcel Koppes,Judy Chappell, Radek Janousek, Rosie Moss, David Lockley, Nicola Haden, Bill Kennedy, Martina Morris, Kai Wuerfl-Davidek, Christiane Leipersberger,Yvonne Grueneklee, Nadia Balavoine, and Chris Reinders for making certain that our vision remains worldwide in scope.

David Buckland, Marie Chieng, Lucy Chong, Leslie Lim,Audrey Gan, Pang Ai Hua,Joseph Chan,June Lim, and Siti Zuraidah Ahmad of Pansing Distributors for the enthusiasm with which they receive our books.

David Scott, Tricia Wilden, Marilla Burgess, Annette Scott, Andrew Swaffer, Stephen O'Donoghue, Bec Lowe, Mark Langley, and Anyo Geddes of Woodslane for distributing our books throughout Australia, New Zealand, Papua New Guinea, Fiji,Tonga, Solomon Islands, and the Cook Islands.

Lead Author

Tom St Denis is a software developer known best for his LibTom series of public domain cryptographic libraries. He has spent the last five years distributing, developing, and supporting the cause of open source cryptography, and has championed its safe deployment. Tom currently is employed for Elliptic Semiconductor Inc. where he designs and develops software libraries for embedded systems. He works closely with a team of diverse hardware engineers to create a best of breed hardware and software combination.

Tom is also the author (with Greg Rose) of *BigNum Math: Implementing Cryptographic Multiple Precision Arithmetic* (Syngress Publishing, ISBN: 1-59749-112-8), which discusses the deployment of crypytographic integer mathematics.

Technical Editor and Coauthor

Simon Johnson is a security engineer for a technology outfit based in the United Kingdom. Simon became interested in cryptography during his teenage years, studying all aspects of conventional software cryptography. He has been an active contributor to the cryptographic usenet group Sci.Crypt since the age of 17, attends various security conferences around the world, and continues to openly promote safe computing practices.

Contents

Preface

Here we are, in the preface of my 2nd text. I do not know exactly what to tell you, the reader, other than this one is more dramatic and engaging than the last. I do not want to leak too many details, but let's just say that RSA has an affair with SHA behinds MD5's back. In all seriousness, let's get down to business now.

As I write this, nearly on the eve of the print date, I anticipate the final product and hope that I have hit my target thesis for the text. This text is the product of a year's worth of effort, spanning from early 2006 to nearly November of 2006. I spent many evenings writing after work; my only hope is that this text reaches the target audience effectively. It certainly was an entertaining process, albeit at times laborious, and like my first text, well worth it.

First, I should explain who the authors are before I go into too much depth about this text. This text was written mostly by me, Tom St Denis, with the help of my co-author, Simon Johnson, as a technical reviewer. I am a computer scientist from Ontario, Canada with a passion for all things cryptography related. In particular, I am a fan of working with specialty hardware and embedded systems.

My claim to fame and probably how you came to know about this text is through the LibTom series of projects. These are a series of cryptographic and mathematic libraries written to solve various problems that real-life developers have. They were also written to be educational for the readers. My first project, *LibTomCrypt*, is the product of nearly five years of work. It supports quite a few useful cryptographic primitives, and is actually a very good resource for this text. Continuing the line of cryptographic projects, I started *LibTomMath* in 2002. It is a portable math library to manipulate large integers. It has found a home with *LibTomCrypt* as one of the default math providers, and is also integral to other projects such as *Tcl* and *Dropbear*. To improve upon *LibTomMath*, I wrote *TomsFastMath*, which is an insanely fast and easy to port math library for cryptographic operations.

I wrote all of these projects to be free, not only in the sense that people can acquire them free of charge, but also in the sense that there are no strings attached. They are, in fact, all public domain. For me, at least, it was not enough just to provide code. I also provide documentation that explains how to use the projects. Even that was not enough. I also document and clean the source code; the code itself is of educational value. The first project to be used in this manner was the *LibTomMath* project. In 2003, I wrote a text, *BigNum Math: Implementing Cryptographic Multiple Precision Arithmetic* (ISBN:1597491128), which Syngress Publishing published in 2006. The project literally inserts code from the project into the text. Coupled with pseudo-code, the text teaches how to manipulate large integers quite effortlessly.

The LibTom projects are themselves guided by a simple motto that I've developed over the years.

"Open Source. Open Academia. Open Minds"

What this means is that, by providing source code along with useful documentation and supporting material, we can educate others and open their minds to new ideas and techniques. It extends the typical open source philosophy in an educational capacity. For instance, it is nice that the GNU Compiler Collection (GCC) is open source, but it is hardly an educational project. Enough of this though; this line of thinking is the subject of my next text (due sometime in 2009).

I continue to work on my LibTom projects and am constantly vigilant so as to promote them whenever possible. I regularly attend conferences such as Toorcon to spread the word of the LibTom philosophy in hopes of recruiting new open-source developers to the educational path.

So, who is Simon? Simon Johnson is a computer programmer from England. He spends his days reading about computer security and cryptographic techniques. Professionally, he is a security engineer working with C# applications and the like. Simon and I met through the Usenet wasteland that is sci.crypt, and have collaborated on various projects. Throughout this text, Simon played the role of technical reviewer. His schedule did not quite afford him as much time to help on this project as he would have liked, but his help was still crucial. It is safe to say we can expect a text or two from Simon in the years to come.

So what is this book about? *Cryptography for Developers*. Sounds authorative and independent: Right and wrong. This text is an essential *guide* for developers who are not cryptographers. It is not, however, meant to be the only text on the subject. We often refer to other texts as solid references. Definitely, you will want a copy of "BigNum Math." It is an essential text on implementing the large integer arithmetic required by public key algorithms. Another essential is *The Guide to Elliptic Curve Cryptography* (ISBN 038795273X), which covers, at a nice introductory level, all that a developer requires to know about elliptic curve algorithms. It is our stance that we do you, the reader, more good by referring to well-read texts on the subject instead of trying to duplicate their effort. There are also the standards you may want to pick up. For instance, if you are to implement RSA cryptography, you really need a copy of *PKCS #1* (which is free). While this text covers PKCS #1 operations, having the standard handy is always nice. Finally, I strongly encourage the reader to acquire copies of the LibTom projects to get first-hand experience working with cryptographic software.

Who is this book for? I wrote this book for the sort of people who send me support e-mail for my projects. That is not to say this text is *about* the projects, merely about the problems users seem to have when using them. Often, developers tasked with security problems are not cryptographers. They are bright people, who, with careful guidance, can implement secure cryptosystems. This text aims to guide developers in their journey towards solving various cryptographic problems. If you have ever sat down and asked yourself, "Just how do I setup AES anyways?" then this text is for you.

This text is **not** for people looking at a solid academic track in cryptography. This is not the Handbook of Applied Cryptography, nor is it the Foundations of Cryptography. Simply put, if you are not tasked with implementing cryptography, this book may not be for you. This is part of the thinking that went into the design and writing of this text. We strived to include enough technical and academic details as to make the discussions accurate and useful. However, we omitted quite a few cryptographic discussions when they did not fit well in the thesis of the text.

I would like to thank various people for helping throughout this project. Greg Rose helped review a chapter. He also provided some inspiration and insightful comments. I would like to thank Simon for joining the project and contributing to the quality of the text. I would like to thank Microsoft Word for giving me a hard time. I would like to thank Andrew, Erin, and the others at Syngress for putting this book together. I should also thank the LibTom project users who were the inspiration for this book. Without their queries and sharing of their experiences, I would never have had a thesis to write about in the first place.

Finally, I would like to thank the pre-order readers who put up with the slipped print date. My bad.

—Tom St Denis
Ottawa, Ontario, Canada
October 2006

Chapter 1

Introduction

Solutions in this chapter:

- Threat Models
- What Is Cryptography?
- Asset Management
- Common Wisdom
- Developer Tools

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

Computer security is an important field of study for most day-to-day transactions. It arises when we turn on our cellular phones, check our voice mail and e-mail, use debit or credit cards, order a pay-per view movie, use a transponder through EZ-Pass, sign on to online video games, and even during visits to the doctor. It is also often used to establish virtual private networks (VPNs) and Secure Shell connections (SSH), which allows employees to telecommute and access computers remotely.

The use, and often misuse, of cryptography to solve security problems are driven by one cause: the need for security. Simply *needing* security does not make it so, a lesson all too often learned after the fact, or more importantly, after the exploits.

Notes from the Underground…

Known Exploit—*Dark Age of Camelot*

URL: http://capnbry.net/daoc/advisory20040323/daoc-advisory2.html

In March 2004, an exploit for the video game *Dark Age of Camelot* (Mythic Entertainment) made use of the weak server authentication the game used to perform secure billing transactions. It allowed attackers to intercept communication between a real server and client and read all the private billing data.

Even though the developers used a known and tested cryptographic library to provide core algorithms, they had used the algorithms incorrectly. As a result, the attackers did not have to break hard cryptographic algorithms such as RSA or RC4, just the weak construction in which they were used.

The Mythic exploit is a classic example of not knowing how to use tools properly. It is hard to fault the developer team. They are, after all, video game developers, not fulltime cryptographers. They do not have the resources to bring a cryptographer on team, let alone contract out to independent firms to provide the same services.

The circumstances Mythic was in at the time are very common for most software development companies throughout the world. As more and more small businesses are created, the fewer resources they have to pool on security staff. Security is not the goal of the end-user product, but merely a requirement for the product to be useful.

For example, banking hardly requires cryptography to function; you can easily hand someone $10 without first performing an RSA key exchange. Similarly, cell phones do not require cryptography to function. The concept of digitizing speech, compressing it, encoding the bits for transmission over a radio and the reverse process are done all the time without one thought toward cryptography.

Because security is not a core product value, it is either neglected or relegated to a secondary "desired" goal list. This is rather unfortunate, since cryptography and the deployment of is often a highly manageable task that does not require an advanced degree in cryptography or mathematics to accomplish. In fact, simply knowing *how* to use existing cryptographic toolkits is all a given security task will need.

Threat Models

A threat model explicitly addresses venues of attack adversaries will exploit in their attempts to circumvent your system. If you are a bank, they want credentials; if you are an e-mail service, they want private messages, and so on. Simply put, a threat model goes beyond the normal use of a system to examine what happens on the corner cases. If you expect a response in the set *X*, what happens when they send you a response *Y* that does not belong to that set?

The simplest example of this modeling is the *atoi*() function in C, which is often used in programs without regard to error detection. The program expects an ASCII encoded integer, but what happens when the input is not an integer? While this is hardly a security flaw, it is the exact sort of corner cases attackers will exploit in your system.

A threat model begins at the levels at which anyone, including insiders, can interact with the system. Often, the insiders are treated as special users; with virtually unlimited access to data they are able to commit rather obtuse mistakes such as leaving confidential data of thousands of customers in a laptop inside a car (see, for instance, http://business.timesonline.co.uk/article/0,,13129-2100897,00.html).

The model essentially represents the *Use Cases* of the system in terms of what they potentially allow if broken or circumvented. For example, if the user must first provide a password, attackers will see if the password is handled improperly. They will see if the system prevents users from selecting easily guessable passwords, and so on.

The major contributing factor to development of an accurate threat model is to not think of the use cases in terms of what a proper user will do. For example, if your program submits data to a database, attackers may try an *injection attack* by sending SQL queries inside the submitted data. A normal user probably would never do that. It is nearly impossible to document the entire threat model design process, as the threat model is as complicated if not more so than the system design itself.

This book does not pretend to offer a treatment of secure coding practices sufficient to solve this problem. However, even with that said, there are simple rules of threat model design developers should follow when designing their system:

Simple Rules of Threat Model Design

1. How many ways can someone transition into this use case?
   - i. Think outside of the intended transitions.
   - ii. Are invalid contexts handled?
2. What components does the input interact with?
   - i. What would "invalid inputs" be?
3. Is this use case effective?
   - i. Is it explicitly weak? What are the assumptions?
   - ii. Does it accomplish the intended goal?

What Is Cryptography?

Cryptography is the automated (or algorithmic) method in which security goals are accomplished. Typically, when we say "crypto algorithm" we are discussing an algorithm meant to be executed on a computer. These algorithms operate on messages in the form of groups of bits.

More specifically, people often think of cryptography as the study of ciphers; that is, algorithms that conceal the meaning of a message. Privacy, the actual name of this said goal, is all but one of an entire set of problems cryptography is meant to address. It is perhaps most popular, as it is the oldest cryptography related security goal and feeds into our natural desire to have secrets. Secrets in the form of desires, wants, faults, and fears are natural emotions and thoughts all people have. It of course helps that modern Hollywood plays into this with movies such as *Swordfish* and *Mercury Rising*.

Cryptographic Goals

However, there are other natural cryptographic problems to be solved and they can be equally if not more important depending on who is attacking you and what you are trying to secure against attackers. The cryptographic goals covered in this text (in order of appearance) are privacy, integrity, authentication, and nonrepudiation.

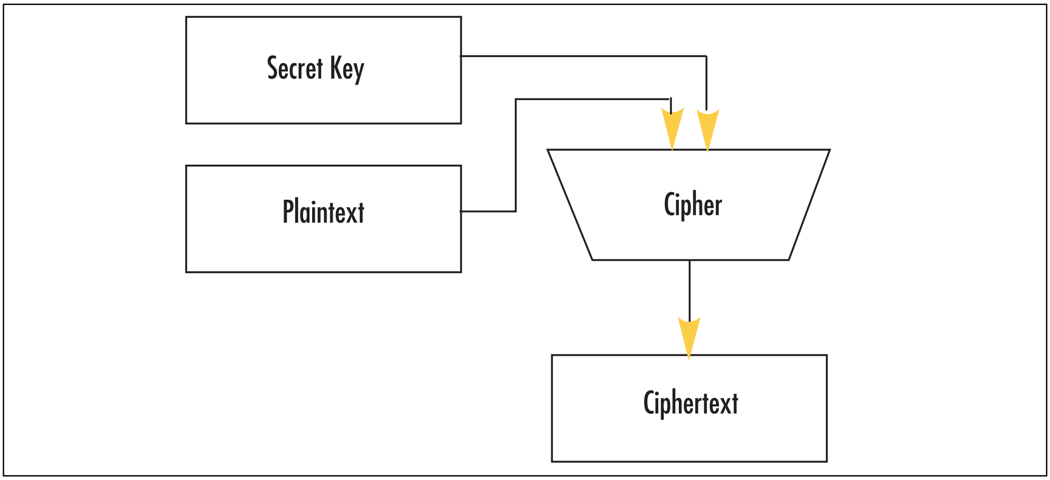
Privacy

Privacy is the property of concealing the meaning or intent of a message. In particular, it is to conceal it from undesired parties to an information transmission medium such as the Internet, wireless network link, cellular phone network, and the like.

Privacy is typically achieved using *symmetric key ciphers*. These algorithms accept a secret key and then proceed to encrypt the original message, known as *plaintext*, and turn it into piece of information known as *ciphertext*. From a standpoint of information theory, ciphertext contains the same amount of entropy (uncertainty, or simply put, information) as the plaintext. This means that a receiver (or recipient) merely requires the same secret key and ciphertext to reconstruct the original plaintext.

Ciphers are instantiated in one of two forms, both having their own strengths and weaknesses. This book only covers *block ciphers* in depth, and in particular, the National Institute for Standards and Technologies (NIST) Advanced Encryption Standard (AES) block cipher. The AES cipher is particularly popular, as it is reasonably efficient in large and small processors and in hardware implementations using low-cost design techniques. Block ciphers are also more popular than their stream cipher cousins, as they are universal. As we will see, AES can be used to create various privacy algorithms (including one mode that resembles a stream cipher) and integrity and authentication algorithms. AES is free, from an intellectual property (IP) point of view, well documented and based on sound cryptographic theory (Figure 1.1).

Figure 1.1 Block Diagram of a Block Cipher



NOTE

URL: http://csrc.nist.gov/CryptoToolkit/aes/rijndael/

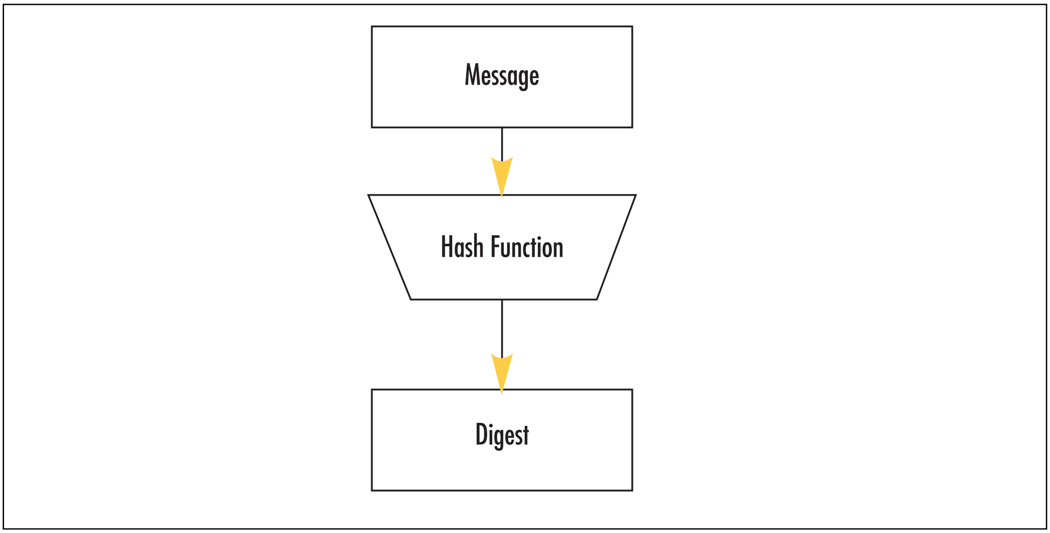
The Advanced Encryption Standard is the official NIST recommended block cipher. Its adoption is highly widespread, as it is the direct replacement for the aging and much slower Data Encryption Standard (DES) block cipher.

Integrity

Integrity is the property of ensuring correctness in the absence of an actively participating adversary. That sounds more complicated than it really is. What this means in a nutshell is ensuring that a message can be delivered from point A to point B without having the meaning (or content) of the original message change in the process. Integrity is limited to the instances where adversaries are not actively trying to subvert the correctness of the delivery.

Integrity is usually accomplished using cryptographic *one-way hash functions*. These functions accept as an input an arbitrary length message and produce a fixed size *message digest*. The message digest, or *digest* for short, usually ranging in sizes from 160 to 512 bits, is meant to be a representative of the message. That is, given a message and a matching digest, one could presume that outside the possibility of an active attacker the message has been delivered intact. Hash algorithms are designed to have various other interesting properties such as being one-way and collision resistance (Figure 1.2).

Figure 1.2 Block Diagram of a One-Way Hash Function



Hashes are designed to be one-way primarily because they are used as methods of achieving password-based authenticators. This implies that given a message digest, you cannot compute the input that created that digest in a feasible (less than exponential) amount of time. Being one-way is also required for various algorithms such as the Hash Message Authentication Code (see Chapter 5, "Hash Functions") algorithm to be secure.

Hashes are also required to be collision resistant in two significant manners. First, a hash must be a pre-image resistant against a fixed target (Figure 1.3). That is, given some value *y* it is hard to find a message *M* such that *hash(M)* = *y*. The second form of resistance, often cited as 2nd pre-image resistance (Figure 1.4) is the inability to find two messages *M1* (given) and *M2* (chosen at random) such that *hash (M1) = hash(M2)*. Together, these imply collision resistance.

Figure 1.3 Pre-Image Collision Resistance

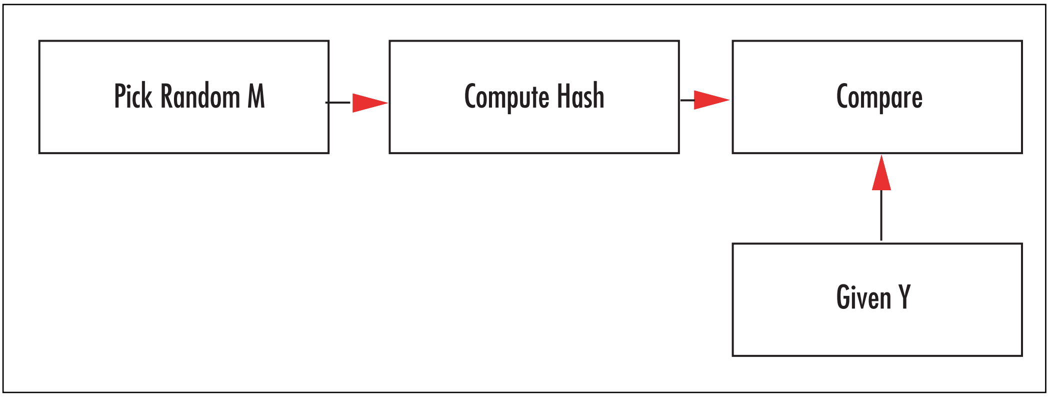
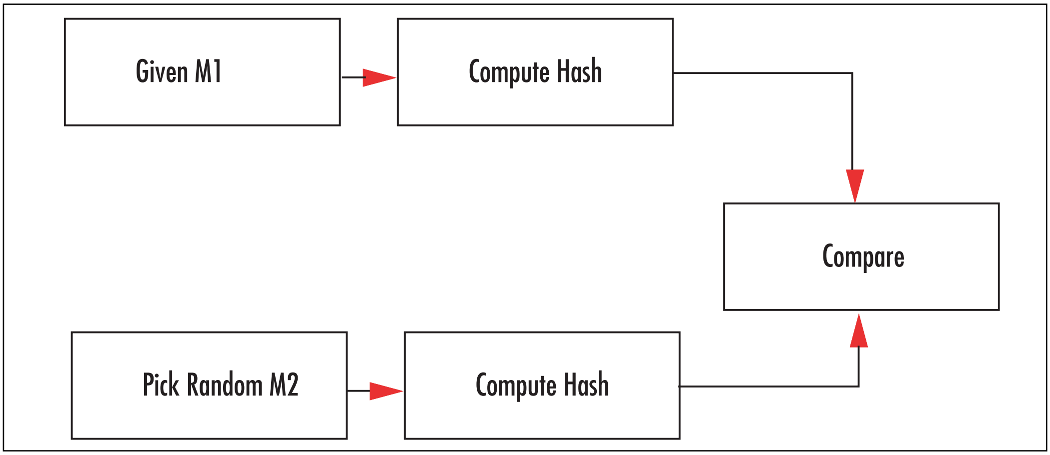


Figure 1.4 2nd Pre-Image Collision Resistance



Hashes are not keyed algorithms, which means there is no secret information to which attackers would not be privy in the process of the algorithms workflow. If you can compute the message digest of a public message, so can they. For this reason, in the presence of an attacker the integrity of a message cannot be determined.

Even in light of this pitfall, they are still used widely in computing. For example, most online Linux and BSD distributions provide a digest from programs such as md5sum as part of their file manifests. Normally, as part of an update process the user will download both the file (tarball, RPM, .DEB, etc.) and the digest of the file. This is used under the assumption that the threat model does not include active adversaries but merely storage and distribution errors such as truncated or overwritten files.

This book discusses the popular Secure Hash Standard (SHS) SHA-1 and SHA-2 family of hash functions, which are part of the NIST portfolio of cryptographic functions. The SHA-2 family in particular is fairly attractive, as it introduces a range of hashes that produce digests from 224 to 512 bits in size. They are fairly efficient algorithms given that they require no tables or complicated instructions and are fairly easy to reproduce from specification.

WARNING!

The MD5 hash algorithm has long been considered fairly weak. Dobbertin found flaws in key components of the algorithm, and in 2005 researchers found full collisions on the function. New papers appearing in early 2006 are discussing faster and faster methods of finding collisions.

These researchers are mostly looking at 2nd pre-image collisions, but there are already methods of using these collisions against IDS and distribution systems.

It is highly recommended that developers avoid the MD5 hash function. To a certain extent, even the SHA-1 hash function should be avoided in favor of the new SHA-2 hash functions. For those following the European standards, there is also the Whirlpool hash function to choose from.

Authentication

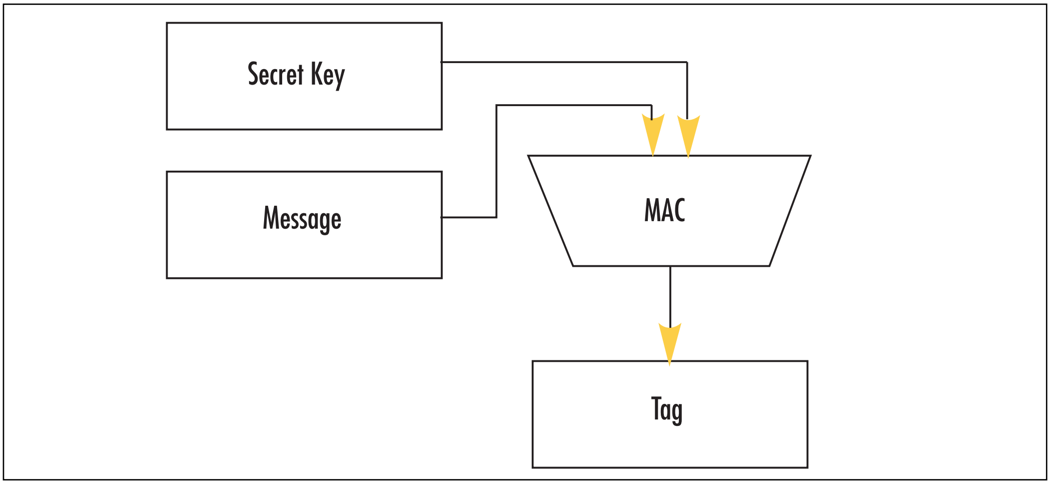
Authentication is the property of attributing an identity or representative of the integrity of a message. A classic example would be the wax seal applied to letters. The mark would typically be hard to forge at the time they were being used, and the presence of the unbroken mark would imply the documents were authentic.

Another common form of authentication would be entering a personal identification number (PIN) or password to authorize a transaction. This is not to be confused with nonrepudiation, the inability to refute agreement, or with authentication protocols as far as key agreement or establishment protocols are concerned. When we say we are authenticating a message, it means we are performing additional steps such that the recipient can verify the integrity of a message in the presence of an active adversary.

The process of key negotiation and the related subject of authenticity are a subject of public key protocols. They use much the same primitives but have different constraints and goals. An authentication algorithm is typically meant to be symmetric such that all parties can produce verifiable data. Authenticity with the quality of nonrepudiation is usually left to a single producer with many verifiers.

In the cryptographic world, these authentication algorithms are often called Message Authentication Codes (MAC), and like hash functions produce a fixed sized output called a message *tag*. The tag would be the information a verifier could use to validate a document. Unlike hash functions, the set of MAC functions requires a secret key to prevent anyone from forging tags (Figure 1.5).

Figure 1.5 Block Diagram for a MAC Function



The two most common forms of MAC algorithms are the CBC-MAC (now implemented per the OMAC1 algorithm and called CMAC in the NIST world) and the HMAC functions. The CBC-MAC (or CMAC) functions use a block cipher, while the HMAC functions use a hash function. This book covers both NIST endorsed CMAC and HMAC message authentication code algorithms.

Another method of achieving authentication is using public key algorithms such as RSA using the PKCS #1 standard or the Elliptic Curve DSA (EC-DSA or ANSI X9.62) standard. Unlike CMAC or HMAC, a public key based authenticator does not require both parties to share private information prior to communicating. Public key algorithms are therefore not limited to online transactions when dealing with random unknown parties. That is, you can sign a document, and anyone with access to your public key can verify without first communicating with you.

Public key algorithms are used in different manners than MAC algorithms and as such are discussed later in the book as a different subject (Table 1.1).

Table 1.1 Comparing CMAC, HMAC and Public Key Algorithms

Characteristic	CMAC	HMAC	RSA PKCS #1	EC-DSA
Speed	Fast	Fastest	Slowest	Slow
Complexity	Simple	Simplest	Hard	Hardest
Needs secret keys	Yes	Yes	No	No
Ease of deployment	Harder	Harder	Simple	Simple

The manner in which the authenticators are used depends on their construction. Public key based authenticators are typically used to negotiate an initial greeting on a newly opened communication medium. For example, when you first connect to that SSL enabled Web site, you are verifying that the signature on the Web site's certificate was really signed by a root signing authority. By contrast, algorithms such as CMAC and HMAC are typically used after the communication channel has been secured. They are used instead to ensure that communication traffic was delivered without tampering. Since CMAC and HMAC are much faster at processing message data, they are far more valuable for high traffic mediums.

Nonrepudiation

Nonrepudiation is the property of agreeing to adhere to an obligation. More specifically, it is the inability to refute responsibility. For example, if you take a pen and sign a (legal) contract your signature is a nonrepudiation device. You cannot later disagree to the terms of the contract or refute ever taking party to the agreement.

Nonrepudiation is much like the property of authentication in that their implementations often share much of the same primitives. For example, a public key signature can be a nonrepudiation device if only one specific party has the ability to produce signatures. For this reason, other MAC algorithms such as CMAC and HMAC cannot be nonrepudiation devices.

Nonrepudiation is a very important property of billing and accounting that is more often than not improperly addressed. For example, pen signatures on credit card receipts are rarely verified, and even when the clerk glances at the back of the card, he is probably not a handwriting expert and could not tell a trivial forgery from the real thing. Cell phones also typically use MAC algorithms as resource usage authenticators, and therefore do not have nonrepudiation qualities.

Goals in a Nutshell

Table 1.2 compares the four primary cryptographic goals.

Table 1.2 Common Cryptographic Goals

Goal	Properties	
Privacy	1.	Concerned with concealing the meaning of a message from unintended participants to a communication medium.
	2.	Solved with symmetric key block ciphers.
	3.	Recipient does not know if the message is intact.
	4.	Output of a cipher is ciphertext.
Integrity	1.	Concerned with the correctness of a message in transit.
	2.	Assumes there is no active adversary.
	3.	Solved with one-way hash functions.
	4.	Output of a hash is a message digest.
Authentication	1.	Concerned with the correctness of a message in transit
	2.	Assumes there are active adversaries.
	3.	Solved with Message Authentication Functions (MAC).
	4.	Output of a MAC is a message tag.
Nonrepudiation	1.	Concerned with binding transaction.
	2.	Goal is to prevent a party from refuting taking party to a transaction.
	3.	Solved with public key digital signatures.
	4.	Output of a signature algorithm is a signature.

Asset Management

A difficult challenge when implementing cryptography is the ability to manage user assets and credentials securely and efficiently. Assets could be anything from messages and files to things such as medical information and contact lists; things the user possesses that do not specifically identify the user, or more so, are not used traditionally to identify users. On the other hand, credentials do just that. Typically, credentials are things such as usernames, passwords, PINs, two-factor authentication tokens, and RFID badges. They can also include information such as private RSA and ECC keys used to perform signatures.

Assets are by and large not managed in any particularly secure fashion. They are routinely assumed authentic and rarely privacy protected. Few programs offer integrated security features for assets, and instead assume the user will use additional tools such as encrypted file stores or manual tools such as GnuPG. Assets can also be mistaken for credentials in very real manners.

For instance, in 2005 it was possible to fly across Canada with nothing more than a credit card. Automated check-in terminals allowed the retrieval of e-tickets, and boarding agents did not check photo identification while traveling inside Canada. The system assumed possession of the credit card meant the same person who bought the ticket was also standing at the check-in gate.

Privacy and Authentication

Two important questions concerning assets are whether the asset is private and whether it has to be intact. For example, many disk encryption users apply the tool to their entire system drive. Many system files are universally accessible as part of the install media. They are in no way private assets, and applying cryptography to them is a waste of resources. Worse, most systems rarely apply authentication tools to the files on disk (EncFS is a rare exception to the lack of authentication rule. http://encfs.sourceforge.net/), meaning that many files, including applications, that are user accessible can be modified. Usually, the worse they can accomplish is a denial of service (DoS) attack on the system, but it is entirely possible to modify a working program in a manner such that the alterations introduce flaws to the system. With authentication, the file is either readable or it is not. Alterations are simply not accepted.

Usually, if information is not meant to be public it is probably a good idea to authenticate it. This provides users with two forms of protection around their assets. Seeing this as an emerging trend, NIST and IEEE have both gone as far as to recommend combined modes (CCM and GCM, respectively) that actually perform both encryption and authentication. The modes are also of theoretical interest, as they formally reduce to the security of their inherited primitives (usually the AES block cipher). From a performance point of view, these modes are less efficient than just encryption or authentication alone. However, that is offset by the stability they offer the user.

Life of Data

Throughout the life of a particular asset or credential, it may be necessary to update, add to, or even remove parts of or the entire asset. Unlike simply creating an asset, the security implications of allowing further modifications are not trivial. Certain modes of operation are not secure if their parameters are left static. For instance, the CTR chaining mode (discussed in Chapter 4, "Advanced Encryption Standard") requires a fresh initial value (IV) whenever encrypting data. Simply re-using an existing IV, say to allow an inline modification, is entirely insecure against privacy threats. Similarly, other modes such as CCM (see Chapter 7, "Encrypt and Authenticade Modes") require a fresh Nonce value per message to assure both the privacy and authenticity of the output.

Certain data, such as medical data, must also possess a *lifespan*, which in cryptographic terms implies access control restrictions. Usually, this has been implemented with public key digital signatures that specify an expiry date. Strictly speaking, access control requires a trusted distribution party (e.g., a server) that complies with the rules set forth on the data being distributed (e.g., strictly voluntary).

The concept of in-order data flow stems from the requirement to have stateful communication. For example, if you were sending a banking request to a server, you'd like the request to be issued and accepted only once—specially if it is a bill payment! A *replay attack* arises when an attacker can re-issue events in a communication session that are accepted by the receiving party. The most trivial (initially) solution to the problem is to introduce timestamps or incremental counters into the messages that are authenticated.

The concept of timestamps within cryptography and computer science in general is also a thorny issue. For starters, what time is it? My computer likely has a different time than yours (when measured in even seconds). Things get even more complicated in online protocols where in real-time we must communicate even with skewed clocks that are out of sync and may be changing at different paces. Combined with the out-of-order nature of UDP networks, many online protocols can quickly become beastly. For these reasons, counters are more desirable than timers. However, as nice as counters sound, they are not always applicable. For example, offline protocols have no concept of a counter since the message they see is always the first message they see. There is no "second" message.

It is a good rule of thumb to include a counter inside the authentication portion of data channels and, more importantly, to check the counter. Sometimes, silent rejection of out of order messages is a better solution to just blindly allowing all traffic through regardless of order. The GCM and CCM modes have IVs (or nonces depending), which are trivial to use as both as an IV and a counter.

TIP

A simple trick with IVs or nonces in protocols such as GCM and CCM is to use a few bytes as part of a packet counter. For example, CCM accepts 13 byte nonces when the packet has fewer than 65,536 plaintext bytes. If you know you will have fewer than 4 billion packets, you can use the first four bytes as a packet counter.

For more information, see Chapter 7.

Common Wisdom

There is a common wisdom among cryptographers that security is best left to cryptographers; that the discussion of cryptographic algorithms will lead people to immediately use them incorrectly. In particular, Bruce Schneier wrote in the abstract of his *Secret and Lies* text:

> I have written this book partly to correct a mistake.

> Seven years ago, I wrote another book: *Applied Cryptography*. In it, I described a mathematical utopia: algorithms that would keep your deepest secrets safe for millennia, protocols that could perform the most fantastical electronic interactions—unregulated gambling, undetectable authentication, anonymous cash—safely and securely. In my vision, cryptography was the great technological equalizer; anyone with a cheap (and getting cheaper every year) computer could have the same security as the largest government. In the second edition of the same book, written two years later, I went so far as to write, "It is insufficient to protect ourselves with laws; we need to protect ourselves with mathematics."
>
> *Abstract. Secret and Lies*, Bruce Schneier

In this quote, he's talking abstractly about the concept that cryptography as a whole, on its own, cannot make us "safe." This is in practice perhaps at least a little true. There are many instances where perfectly valid cryptography has been circumvented by user error or physical attacks (card skimming, for example). However, the notion that the distribution of cryptographic knowledge to the layperson is somehow something to be ashamed of or in anyway avoid is just not valid.

While relying solely on security experts is likely a surefire method of obtaining secure systems, it also is entirely impractical and inefficient. One of the major themes and goals of this book is to dispel the notion that cryptography is harder (or needs to be harder) than it really is. Often, a clear understanding of your threats can lead to efficient and easily designed cryptosystems that address the security needs. There are already many seemingly secure products available in which the developers performed just enough research to find the right tools for the job—software libraries, algorithms, or implementations—and a method of using them that is secure for their task at hand.

Keep in mind what this book is meant to address. We are using standard algorithms that already have been designed by cryptographers and finding out how to use them properly. These are two very different tasks. First, a very strong background in cryptography and mathematics is required. Second, all we have to understand is what they are meant to solve, why they are the way they are, and how not to use them.

Upon a recent visit to a video game networking development team, we inspected their cryptosystem that lies behind the scenes away from the end user. It was trivial to spot several things we would personally have designed differently. After a few hours of staring at their design, however, we could not really find anything blatantly wrong with it. They clearly had a threat model in mind, and designed something to work within their computing power limitations that also addressed the model. In short, they designed a working system that allows their product to be effective in the wild. Not one developer on the team studied cryptography or pursued it as a hobby.

It is true that simply knowing you need security and knowing that algorithms exist can lead to fatally inadequate results. This is, in part, why this book exists: to show what algorithms exist, how to implement them in a variety of manners, and the perils of their deployment and usage. A text can both address the ingredients and cooking instructions.

Developer Tools

Throughout this book, we make use of readily access tools such as the GNU Compiler Collection C compiler (GCC) for the X86 32-bit and 64-bit platforms, and ARMv4 processors. They were chosen as they are highly professional, freely accessible, and provide intrinsic value for the reader.

The various algorithms presented in this book are implemented in portable C for the most part to allow the listings to have the greatest and widest audience possible. The listings are complete C routines the reader can take away immediately, and will also be available on the companion Web site. Where appropriate, assembler listings as generated by the C compiler are analyzed. It is most ideal for the reader to be familiar with AT&T style X86 assemblers and ARM style ARMv4 assemblers.

For most algorithms or problems, there are multiple implementation techniques possible. Multiplication, for instance, has three practical underlying algorithms and various actual implementation techniques each. Where appropriate, the various configurations are compared for size and efficiency on the listed platforms. The reader is certainly encouraged to benchmark and compare the configurations on other platforms not studied in this book.

Some algorithms also have security trade-offs. AES, for instance, is fastest when using lookup tables. In such a configuration, it is also possible to leak *Side Channel* information (discussed in Chapter 4). This book explores variations that seek to minimize such leaks. The analysis of these implementations is tied tightly to the design of modern processors, in particular those with data caches. The reader is encouraged to become familiar with how, at the very least, X86 processors such as the Intel Pentium 4 and AMD Athlon64 operate a block level.

Occasionally, the book makes reference to existing works of source code such as TomsFastMath and LibTomCrypt. These are public domain open source libraries written in C and used throughout industry in platforms as small as network sensors to as large as enterprise servers. While the code listings in this book are independent of the libraries, the reader is encouraged to seek out the libraries, study their design, and even use them en lieu of re-inventing the wheel. That is not to say you shouldn't try to implement the algorithms yourself; instead, where the circumstances permit the use of released source can speed product development and shorten testing cycles. Where the users are encouraged to "roll their own" implementations would be when such libraries do not fit within project constraints. The projects are all available on the Internet at the www.libtomcrypt.com Web site.

Timing data on the X86 processors is gathered with the RDTSC instruction, which provides cycle accurate timing data. The reader is encouraged to become familiar with this instruction and its use.

Summary

The development of professional cryptosystems as we shall learn is not a highly complicated or exclusive practice. By clearly defining a threat model that encompasses the systems' points of exposures, one begins to understand how cryptography plays a role in the security of the product. Cryptography is only the most difficult to approach when one does not understand how to seek out vulnerabilities.

Construction of a threat model from all use of the use cases can be considered at least partially completed when the questions "Does this address privacy?" and "Does this address authenticity?" are answered with either a "yes" or "does not apply."

Knowing where the faults lay is only one of the first steps to a secure cryptosystem. Next, one must determine what cryptography has to offer toward a solution to the problem—be it block ciphers, hashes, random number generators, public key cryptography, message authentication codes or challenge response systems. If you design a system that requires an RSA public key and the user does not have one, it will obviously not work. Credential and asset management are integral portions of the cryptosystem design. They determine what cryptographic algorithms are appropriate, and what you must protect.

The runtime constraints of a product determine available space (memory) in which program code and data can reside, and the processing power available for a given task. These constraints more often than not play a pivotal role in the selection of algorithms and the subsequent design of the protocols used in the cryptosystem. If you are CPU bound, for instance, ECC may be more suitable over RSA, and in some instances, no public key cryptography is practical.

All of these design parameters—from the nature of the program to the assets and credentials the users have access to and finally to the runtime environment—must be constantly juggled when designing a cryptosystem. It is the role of the security engineer to perform this juggle and design the appropriate solution. This book addresses their needs by showing the what, how, and why of cryptographic algorithms.

Organization

The book is organized to group problems categories by chapter. In this manner, each chapter is a complete treatment of its respective areas of cryptography as far as developers are concerned. This chapter serves as a quick introduction to the subject of cryptography in general. Readers are encouraged to read other texts, such as *Applied Cryptography*, *Practical Cryptography*, or *The Handbook of Applied Cryptography* if they want a more in-depth treatment of cryptography particulars.

Chapter 2, "ASN.1 Encoding," delivers a treatment of the Abstract Syntax Notation One (ASN.1) encoding rules for data elements such as strings, binary strings, integers, dates and times, and sets and sequences. ASN.1 is used throughout public key standards as a standard method of transporting multityped data structures in multiplatform environments. ASN.1 is also useful for generic data structures, as it is a standard and well understood set of encoding rules. For example, you could encode file headers in ASN.1 format and give your users the ability to use third-party tools to debug and modify the headers on their own. There is a significant "value add" to any project by using ASN.1 encoding over self-established standards. This chapter examines a subset of ASN.1 rules specific to common cryptographic tasks.

Chapter 3, "Random Number Generation," discusses the design and construction of standard random number generators (RNGs) such as those specified by NIST. RNGs and pseudo (or deterministic) random number generators (PRNGs or DRNGs) are vital portions of any cryptosystem, as most algorithms are randomized and require material (such as initial vectors or nonces) that is unpredictable and nonrepeating. Since PRNGs form a part of essentially all cryptosystems, it is logical to start the cryptographic discussions with them. This chapter explores various PRNG constructions, how to initialize them, maintain them, and various hazards to avoid. Readers are encouraged to take the same philosophy to their designs in the field. Always addressing where their "random bits" will come from first is important.

Chapter 4, "Advanced Encryption Standard," discusses the AES block cipher design, implementation tradeoffs, side channel hazards, and modes of use. The chapter provides only a cursory glance at the AES design, concentrating more on the key design elements important to implementers and how to exploit them in various tradeoff conditions. The data cache side channel attack of Bernstein is covered here as a design hazard. The chapter concludes with the treatment of CBC and CTR modes of use for the AES cipher, specifically concentrating on what problems the modes are useful for, how to initialize them, and their respective use hazards.

Chapter 5, "Hash Functions," discusses the NIST SHA-1 and SHA-2 series of one-way hash functions. The designs are covered first, followed by implementation tradeoffs. The chapter discusses collision resistance, provides examples of exploits, and concludes with known incorrect usage patterns.

Chapter 6, "Message Authentication Code Algorithms," discusses the HMAC and CMAC Message Authentication Code (MAC) algorithms, which are constructed from hash and cipher functions, respectively. Each mode is presented in turn, covering the design, implementation tradeoffs, goals, and usage hazards. Particular attention is paid to replay prevention using both counters and timers to address both online and offline scenarios.

Chapter 7, "Encrypt and Authenticate Modes," discusses the IEEE and NIST encrypt and authenticate modes GCM and CCM, respectively. Both modes introduce new concepts to cryptographic functions, which is where the chapter begins. In particular, it introduces the concept of "additional authentication data," which is message data to be authenticated but not encrypted. This adds a new dimension to the use of cryptographic algorithms. The designs of both GCM and CCM are broken down in turn. GCM, in particular, due to its raw mathematical elements possesses efficient table-driven implementations that are explored. Like the MAC chapter, focus is given to the concept of replay attacks, and initialization techniques are explored in depth. The GCM and LRW modes are related in that share a particular multiplication. The reader is encouraged to first read the treatment of the LRW mode in Chapter 4 before reading this chapter.

Chapter 8, "Large Integer Arithmatic," discusses the techniques behind manipulating large integers such as those used in public key algorithms. It focuses on primarily the bottleneck algorithms developers will face in their routine tasks. The reader is encouraged to read the supplementary "Multi-Precision Math" text available on the Web site to obtain a more in-depth treatment. The chapter focuses mainly on fast multiplication, squaring, reduction and exponentiation, and the various practical tradeoffs. Code size and performance comparisons highlight the chapter, providing the readers with an effective guide to code design for their runtime constraints. This chapter lightly touches on various topics in number theory sufficient to successfully navigate the ninth chapter.

Chapter 9, "Public Key Algorithms," introduces public key cryptography. First, the RSA algorithm and its related PKCS #1 padding schemes are presented. The reader is introduced to various timing attacks and their respective counter-measures. The ECC public key algorithms EC-DH and EC-DSA are subsequently discussed. They'll make use of the NIST elliptic curves while exploring the ANSI X9.62 and X9.63 standards. The chapter introduces new math in the form of various elliptic curve point multipliers, each with unique performance tradeoffs. The reader is encouraged to read the text "Guide to Elliptic Curve Cryptography" to obtain a deeper understanding of the mathematics behind elliptic curve math.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: When should the development of a threat model begin?

A: Usually alongside the project design itself. However, concerns such as runtime constraints may not be known in advance, limiting the accuracy of the threat model solutions.

Q: Is there any benefit to rolling our own cryptographic algorithms in place of using algorithms such as AES or SHA-2?

A: Usually not, as far as security is concerned. It is entirely possible to make a more efficient algorithm for a given platform. Such design decisions should be avoided, as they remove the product from the realm of standards compliance and into the skeptic category. Loosely speaking, you can also limit your liability by using best practices, which include the use of standard algorithms.

Q: What is certified cryptography? FIPS certification?

A: FIPS certification (see http://csrc.nist.gov/cryptval/) is a process in which a binary or physical implementation of specific algorithms is placed through FIPS licensed validation centers, the result of which is either a pass (and certificate) or failure for specific instance of the implementation. You cannot certify a design or source code. There are various levels of certification depending on how resistant to tampering the product desires to be: level one being a known answer battery, and level four being a physical audit for side channels.

Q: Where can I find the LibTomCrypt and TomsFastMath projects? What platforms do they work with? What is the license?

A: They are hosted at www.libtomcrypt.com. They build with MSVC and the GNU CC compilers, and are supported on all 32- and 64-bit platforms. Both projects are released as public domain and are free for all purposes.

Chapter 2

ASN.1 Encoding

Solutions in this chapter:

- **Overview of ASN.1**
- **ASN.1 Syntax**
- **ASN.1 Data Types**
- **Implementation**
- **Putting It All Together**

☑ **Summary**

☑ **Solutions Fast Track**

☑ **Frequently Asked Questions**

Overview of ASN.1

Abstract Syntax Notation One (ASN.1) is an ITU-T set of standards for encoding and representing common data types such as printable strings, octet strings, bit strings, integers, and composite sequences of other types as byte arrays in a portable fashion. Simply put, ASN.1 specifies how to encode nontrivial data types in a fashion such that any other platform or third-party tool can interpret the content.

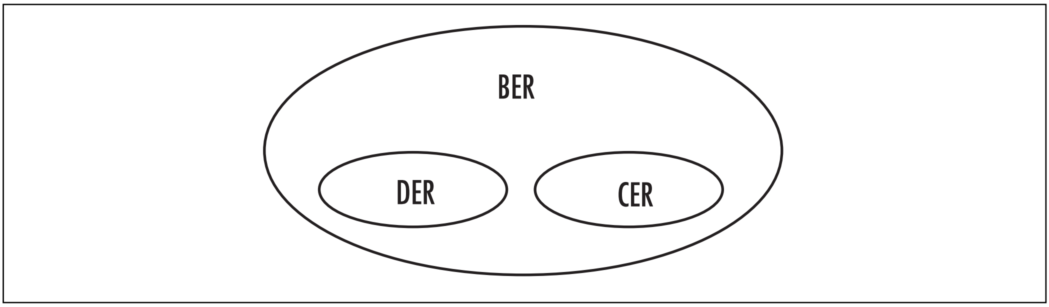
For example, on certain platforms the literal character "a" would be encoded in ASCII (or IA5) as the decimal value 97, whereas on other non-ASCII platforms it may have another encoding. ASN.1 specifies an encoding such that all platforms can decode the string uniformly. Similarly, large integers, bit strings, and dates are also platform sensitive data types that benefit from standardization.

This is beneficial for the developer and for the client (or customer) alike, as it allows the emitted data to be well modeled and interpreted. For example, the developer can tell the client that the data the program they are paying for is in a format another developer down the road can work with—avoiding vendor lock-in problems and building trust.

Formally, the ASN.1 specification we are concerned with is ITU-T X.680, which documents the common data types we will encounter in cryptographic applications. ASN.1 is used in cryptography as it formally specifies the encodings down to the byte level of various data types that are not always inherently portable and, as we will see shortly, encodes them in a deterministic fashion. Being deterministic is particularly important for signature protocols that require a single encoding for any given message to avoid ambiguity.

ASN.1 supports Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and the Distinguished Encoding Rules (DER) (Figure 2.1). These three modes specify how to encode and decode the same ASN.1 types. Where they differ is in the choice of encoding rules. Specifically, as we will see, various parameters such as field length, Boolean representation, and various other parameters can have multiple valid encodings. The DER and CER rules are similar in that they specify fully deterministic encoding rules. That is, where multiple encoding rules are possible only one is actually valid.

Figure 2.1 The Set of ASN.1 Encoding Rules



Basic encoding rules are the most liberal set of encoding rules, allowing a variety of encodings for the same data. Effectively, any ASN.1 encoding that is either CER or DER can be BER decoded, but not the opposite. All of the data types ASN.1 allows are first described in terms of BER rules, and then CER or DER rules can be applied to them. The actual complete ASN.1 specification is far more complex than that required by the average cryptographic tasking. We will only be looking at the DER rules, as they are what most cryptographic standards require. It should also be noted that we will not be supporting the "constructed" encodings even though we will discuss them initially.

ASN.1 was originally standardized in 1994 and again in 1997 and 2002. The current standard is ITU-T X.680, also filed as ISO/IEC 8824-1:2002, and is freely available on the Internet (ASN.1 Reference: http://asn1.elibel.tm.fr/standards/). This chapter discusses the implementation of ASN.1 encodings more than the theory behind the ASN.1 design. The reader is encouraged to read the ITU-T documents for a deeper treatment. The standards we are aiming to support by learning ASN.1 are the PKCS #1/#7 and ANSI X9.62/X9.63 public key cryptography standards. As it turns out, to support these standards your ASN.1 routines have to handle quite a bit of the ASN.1 types in the DER rule-set, resulting in encoding routines usable for a broad range of other tasks.

While formally ASN.1 was defined for more than just cryptographic tasks, it has largely been ignored by cryptographic projects. Often, custom headers are encoded in proprietary formats that are not universally decodable (such as some of the ANSI X9.63 data), making their adoption slower as getting compliant and interoperable software is much harder than it needs be. As we explore at the end of this chapter, ASN.1 encoding can be quite easy to use in practical software and very advantageous for the both the developer and the end users.

ASN.1 Syntax

ASN.1 grammar follows a rather traditional Backus-Naur Form (BNF) style grammar, which is fairly loosely interpreted throughout the cryptographic industry. The elements of the grammar of importance to us are the *name*, *type*, *modifiers*, *allowable values*, and *containers*. As mentioned earlier, we are only lightly covering the semantics of the ASN.1 grammar. The goal of this section is to enable the reader to understand ASN.1 definitions sufficient to implement encoders or decoders to handle them. The most basic expression (also referred to as a production in ITU-T documentation) would be

```
Name ::= type
```

which literally states that some element named "Name" is an instance of a given ASN.1 type called "type." For example,

```
MyName ::= IA5String
```

which would mean that we want an element or variable named "MyName" that is of the ASN.1 type IA5String (like an ASCII string).

ASN.1 Explicit Values

Occasionally, we want to specify an ASN.1 type where subsets of the elements have predetermined values. This is accomplished with the following grammar.

```
Name ::= type (Explicit Value)
```

The explicit value has to be an allowable value for the ASN.1 type chosen and is the only value allowed for the element. For example, using our previous example we can specify a default name.

```
MyName ::= IA5String (Tom)
```

This means that "MyName" is the IA5String encoding of the string "Tom". To give the language more flexibility the grammar allows other interpretations of the explicit values. One common exception is the composition vertical bar | . Expanding on the previous example again,

```
MyName ::= IA5String (Tom|Joe)
```

This expression means the string can have either value "Tom" or "Joe." The use of such grammar is to expand on deterministic decoders. For example,

```
PublicKey ::= SEQUENCE {
   KeyType     BOOLEAN(0),
   Modulus     INTEGER,
   PubExponent INTEGER
}

PrivateKey ::= SEQUENCE {
   KeyType         BOOLEAN(1),
   Modulus         INTEGER,
   PubExponent     INTEGER,
   PrivateExponent INTEGER
}
```

Do not dwell on the structure used just yet. The point of this example is to show two similar structures can be easily differentiated by a Boolean value that is explicitly set. This means that as a decoder parses the structure, it will first encounter the "KeyType" element and be able to decide how to parse the rest of the encoded data.

ASN.1 Containers

A container data type (such as a SEQUENCE or Set type) is one that contains other elements of the same or various other types. The purpose is to group a complex set of data elements in one logical element that can be encoded, decoded, or even included in an even larger container.

The ASN.1 specification defines four container types: SEQUENCE, SEQUENCE OF, SET, and SET OF. While their meanings are different, their grammar are the same, and are expressed as

```
Name ::= Container { Name Type [Name Type …] }
```

The text in square brackets is optional, as are the number of elements in the container. Certain containers as we shall see have rules as to the assortment of types allowed within the container. To simplify the expression and make it more legible, the elements are often specified one per line and comma separation.

```
Name ::= Container {
    Name Type,
   [Name Type, …]
}
```

This specifies the same element as the previous example. Nested containers are specified in the exact same manner.

```
Name ::= Container {
   Name Container {
      Name Type,
      [Name Type, …]
   },
   [Name Type, …]
}
```

A complete example of which could be

```
UserRecord ::= SEQUENCE {
   Name SEQUENCE {
       First  IA5String,
       Last   IA5String
   },
   DoB  UTCTIME
}
```

This last example roughly translates into the following C structure in terms of data it represents.

```
struct UserRecord {
   struct Name {
      char *First;
      char *Last;
   };
   time_t DoB;
}
```

As we will see, practical decoding routines for containers are not quite as direct as the last example. However, they are quite approachable and in the end highly flexible.

It is, of course, possible to mix container types within an expression.

ASN.1 Modifiers

ASN.1 specifies various modifiers such as OPTIONAL, DEFAULT, and CHOICE that can alter the interpretation of an expression. They are typically applied where a type requires flexibility in the encoding but without getting overly verbose in the description.

OPTIONAL

OPTIONAL, as the name implies, modified an element such that its presence in the encoding is optional. A valid encoder can omit the element and the decoder cannot assume it will be present. This can present problems to decoders when two adjacent elements are of the same type, which means a look-ahead is required to properly parse the data. The basic OPTIONAL modifier looks like

```
Name ::= Type OPTIONAL
```

This can be problematic in containers, such as

```
Float ::= SEQUENCE {
   Exponent    INTEGER OPTIONAL,
   Mantissa    INTEGER,
   Sign        BOOLEAN
}
```

When the decoder reads the structure, the first INTEGER it sees could be the "Exponent" member and at worse would be the "Mantissa." The decoder must look-ahead by one element to determine the proper decoding of the container.

Generally, it is inadvisable to generate structures this way. Often, there are simpler ways of expressing a container with redundancy such that the decoding is determinable before decoding has gone fully underway. This leads to coder that is easier to audit and review. However, as we shall see, generic decoding of arbitrary ASN.1 encoded data is possible with a flexible linked list decoding scheme.

DEFAULT

The DEFAULT modifier allows containers to have a value that is implied if absent. The standard specifies that "The encoding of a set value or sequence value shall not include an encoding for any component value that is equal to its default value" (Section 11.5 of ITU-T Recommendation X.690 International Standards 8825-1). This means quite simply that if the data to be encoded matches the default value, it will be omitted from the data stream emitted. For example, consider the following container.

```
Command ::= SEQUENCE {
   Token  IA5STRING(NOP) DEFAULT,
   Parameter INTEGER
}
```

If the encoder sees that "Token" is representing the string "NOP," the SEQUENCE will be encoded as if it was specified as

```
Command ::= SEQUENCE {
    Parameter INTEGER
}
```

It is the responsibility of the decoder to perform the look-ahead and substitute the default value if the element has been determined to be missing. Clearly, the default value must be deterministic or the decoder would not know what to replace it with.

CHOICE

The CHOICE modifier allows an element to have more than one possible type in a given instance. Essentially, the decoder tries all the expected decoding algorithms until one of the types matches. The CHOICE modifier is useful when a complex container contains other containers. For instance,

```
UserKey ::= SEQUENCE {
   Name       IA5STRING,
   StartDate  UTCTIME,
   Expire     UTCTIME,
   KeyData    CHOICE {
      ECCKey ECCKeyType,
      RSAKey RSAKeyType
   }
}
```

This last example is a simple public key certificate that presumably allows both ECC and RSA key types. The encoding of this data type is the same as if one of the choices were made; that is, one of the two following containers.

```
ECCUserKey ::= SEQUENCE {
   Name       IA5STRING,
   StartDate  UTCTIME,
   Expire     UTCTIME,
   ECCKey     ECCKeyType,
}

RSAUserKey ::= SEQUENCE {
   Name       IA5STRING,
   StartDate  UTCTIME,
   Expire     UTCTIME,
   RSAKey     RSAKeyType
}
```

The decoder must accept the original sequence "UserKey" and be able to detect which choice was made during encoding, even if the choices involve complicated container structures of their own.

ASN.1 Data Types

Now that we have a basic grasp of ASN.1 syntax, we can examine the data types and their encodings that make ASN.1 so useful. ASN.1 specifies many data types for a wide range of applications—most of which have no bearing whatsoever on cryptography and are omitted from our discussions. Readers are encouraged to read the X.680 and X.690 series of specifications if they want to master all that ASN.1 has to offer.

Any ASN.1 encoding begins with two common bytes (or octets, groupings of eight bits) that are universally applied regardless of the type. The first byte is the type indicator, which also includes some modification bits we shall briefly touch upon. The second byte is the length header. Lengths are a bit complex at first to decode, but in practice are fairly easy to implement.

The data types we shall be examining consist of the following types.

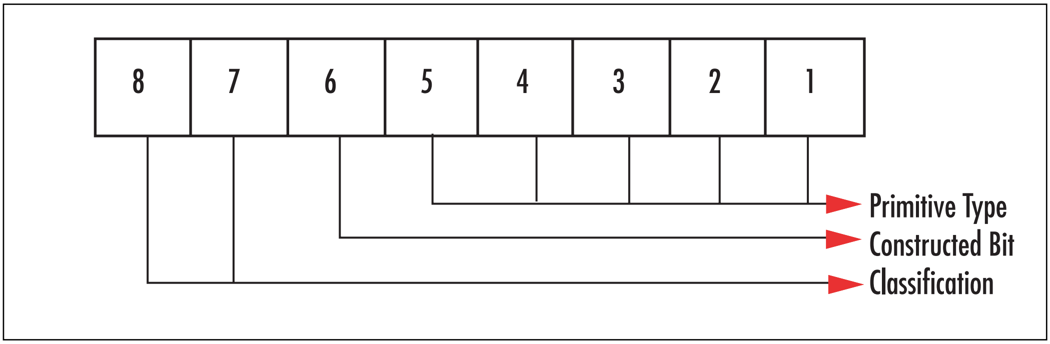
- Boolean
- OCTET String
- BIT String
- IA5String
- PrintableString
- INTEGER
- OBJECT Identifier (OID)
- UTCTIME
- NULL
- SEQUENCE, SEQUENCE OF
- SET
- SET OF

This is enough of the types from the ASN.1 specifications to implement PKCS #1 and ANSI X9.62 standards, yet not too much to overwhelm the developer.

ASN.1 Header Byte

The header byte is always placed at the start of any ASN.1 encoding and is divides into three parts: the classification, the constructed bit, and the primitive type. The header byte is broken as shown in Figure 2.2.

Figure 2.2 The ASN.1 Header Byte



In the ASN.1 world, they label the bits from one to eight, from least significant bit to most significant bit. Setting bit eight would be equivalent in C to OR'ing the value 0x80 to the byte; similarly, a primitive type of value 15 would be encoded as {0, 1, 1, 1, 1} from bit five to one, respectively.

Classification Bits

The classification bits form a two-bit value that does not modify the encoding but describes the context in which the data is to be interpreted. Table 2.1 lists the bit configurations for classifications.

Table 2.1 The ASN.1 Classifications

Bit 8	Bit 7	Class
0	0	Universal
0	1	Application
1	0	Context Specific
1	1	Private

Of all the types, the universal classification is most common. Picking one class over another is mostly a cosmetic or side-band piece of information. A valid DER decoder should be able to parse ASN.1 types regardless of the class. It is up to the protocol using the decoder to determine what to do with the parsed data based on the classification.

Constructed Bit

The constructed bit indicates whether a given encoding is the construction of multiple sub-encodings of the same type. This is useful in general when an application wants to encode what is logically one element but does not have all the components at once. Constructed elements are also essential for the container types, as they are logically just a gathering of other elements.

Constructed elements have their own header byte and length byte(s) followed by the individual encodings of the constituent components of the element. That is, on their own, the constituent component is an individually decodable ASN.1 data type.

Strictly speaking, with the Distinguished Encoding Rules (DER) the only constructed data types allowed are the container class. This is simply because for any other data type and given contents only one encoding is allowable. We will assume the constructed bit is zero for all data types except the containers.

Primitive Types

The lower five bits of the ASN.1 header byte specify one of 32 possible ASN.1 primitives (Table 2.2).

Table 2.2 The ASN.1 Primitives

Code	ASN.1 Type	Use Of
1	Boolean Type	Store Booleans
2	INTEGER	Store large integers
3	BIT STRING	Store an array of bits
4	OCTET STRING	Store an array of bytes
5	NULL	Place holder (e.g., in a CHOICE)
6	OBJECT IDENTIFIER	Identify algorithms or protocols
16	SEQUENCE and SEQUENCE OF	Container of unsorted elements
17	SET and SET OF	Container of sorted elements
19	PrintableString	ASCII Encoding (omitting several non-printable chars)
22	IA5STRING	ASCII Encoding
23	UTCTIME	Time in a universal format

At first glance through the specifications, it may seem odd there is no CHOICE primitive. However, as mentioned earlier, CHOICE is a modifier and not a type; as such, the element chosen would be encoded instead. Each of these types is explained in depth later in this chapter; for now, we will just assume they exist.

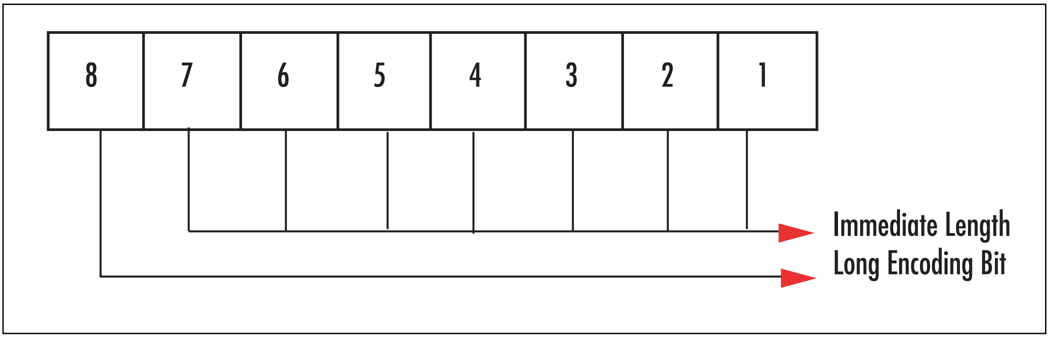
ASN.1 Length Encodings

ASN.1 specifies two methods of encoding lengths depending on the actual length of the element. They can be encoded in either definite or indefinite forms and further split into short or long encodings depending on the length and circumstances.

In this case, we are only concerned with the definite encodings and must support both short and long encodings of all types. In the Basic Encoding Rules, the encoder is free to choose either the short or long encodings, provided it can fully represent the length of the element. The Distinguished Encoding Rules specify we must choose the shortest encoding that fully represents the length of the element. The encoded lengths *do not* include the ASN.1 header or length bytes, simply the payload.

The first byte of the encoding determines whether short or long encoding was used (Figure 2.3).

Figure 2.3 Length Encoding Byte



The most significant bit determines whether the encoding is short or long, while the lower seven bits form an immediate length.

Short Encodings

In short encodings, the length of the payload must be less than 128 bytes. The immediate length field is used to represent the length of the payload, which is where the restriction on size comes from. This is the mandatory encoding method for all lengths less than 128 bytes.

For example, to encode the length 65 (0x41) we simply use the byte 0x41. Since the value does not set the most significant bit, a decoder can determine it is a short encoding and know the length is 65 bytes.

Long Encodings

In long encodings we have an additional level of abstraction on the encoding of the length—it is meant for all payloads of length 128 bytes or more. In this mode, the immediate length field indicates the number of bytes in the length of the payload. To clarify, it specifies how many bytes are required to encode the length of the payload. The length must be encoded in big endian format.

Let us consider an example to show how this works. To encode the length 47,310 (0xB8CE), we first realize that it is greater than 127 so we must use the long encoding format. The actual length requires two octets to represent the value so we use two immediate length bytes. If you look at Figure 2.3, you will see the eighth bit is used to signify long encoding mode and we need two bytes to store the length. Therefore, the first byte is 0x80 | 0x02 or 0x82. Next, we store the value in big endian format. Therefore, the complete encoding is 0x82 B8 CE.

As you can see, this is very efficient because with a single byte we can represent lengths of up to 127 bytes, which would allow the encoding of objects up to 2^{1016} bits in length. This is a truly huge amount of storage and will not be exceeded sometime in the next century.

That said, according to the DER rules the length of the payload length value must be minimal. As a result, the all 1s byte (that is, a long encoding with immediate length of 127) is not valid. Generally, for long encodings it is safe to assume that an immediate length larger than four bytes is not valid, as few cryptographic protocols involve exchanging more than four gigabytes of data in one packet.

TIP

Generally, it is a good idea to refuse ASN.1 long encoded lengths of more than four bytes, as that would imply the payload is more than four gigabytes in size. While this is a good initial check for safety, it is not sufficient. Readers are encouraged to check the fully decoded payload length against their output buffer size to avoid buffer overflow attacks.

Traditionally, such checks should be performed before payload decoding has commenced. This avoids wasting time on erroneous encodings and is much simpler to code audit and review.

ASN.1 Boolean Type

The Boolean data type was provided to encode simple Boolean values without significant overhead in the encoder or decoder.

The payload of a Boolean encoding is either all zero or all one bits in a single octet. The header byte begins with 0x01, the length byte is always short encoded as 0x01, and the contents are either 0x00 or 0xFF depending on the value of the Boolean (Table 2.3).

Table 2.3 ASN.1 Boolean Encodings

Value of Boolean	Encoding
False	0x01 01 00
True	0x01 01 FF

According to BER rules, the true encoding may be any nonzero value; however, DER requires that the true encoding be the 0xFF byte.

ASN.1 Integer Type

The integer type represents a signed arbitrary precision scalar with a portable encoding that is platform agnostic. The encoding is fairly simple for positive numbers.

The actual number that will be stored (which is different for negative numbers as we shall see) is broken up into byte-sized digits and stored in big endian format. For example, if you are encoding the variable *x* such that $x = 256^k * x_k + 256^{k-1} * x_{k-1} + \ldots + 256^0 * x_0$, then the octets $\{x_k, x_{k-1}, \ldots, x_0\}$ are stored in descending order from x_k to x_0. The encoding process stipulates that for positive numbers the most significant bit of the first byte must be zero.

As a result, suppose the first byte was larger than 127 (say, the value 49,468 (0xC13C and 0xC1 > 0x7F)), the obvious encoding would be **0x02 02 C1 3C**; however, it has the most significant bit set and would be considered negative. The simplest solution (and the correct one) is to pad with a leading zero byte. That is, the value 49,468 would be encoded as **0x02 03 00 C1 3C**, which is valid since $256^2 * 0x00 + 256^1 * 0xC1 + 256^0 * 0x3C$ is equal to 49,468.

Encoding negative numbers is less straightforward. The process involves finding the next power of 256 greater than the absolute value you want to encode. For example, if you want to encode –1555, the next power of 256 greater than 1555 would be $256^2 = 65536$. Next, you add the two values to get the two's compliment representation—63,981 in this case. The actual integer encoded is this sum.

So, in this case the value of –1555 would be ASN.1 encoded as **0x02 02 F9 ED**. Two additional rules are then applied to the encoding of integers to minimize the size of the output.

The bits of the first octet and bit 8 of the second octet must

- Not all be ones
- Not all be zero

Decoding integers is fairly straightforward. If the first most significant bit is zero, the value encoded is positive and the payload is the absolute scalar value. If the most significant bit is one, the value is negative and you must subtract the next largest power of 256 from the encoded value (Table 2.4).

Table 2.4 Example INTEGER Encodings

Value	Encoding
0	0x02 01 00
1	0x02 01 01
2	0x02 01 02
127	0x02 01 7F
128	0x02 02 00 80
–1	0x02 01 FF
–128	0x02 01 80
–32768	0x02 02 80 00
1234567890	0x02 04 49 96 02 D2

Note in particular the difference between 128 and –128 in this encoding. They both evaluate to 0x80, but the positive value requires the 0x00 prefix to differentiate it.

ASN.1 BIT STRING Type

The BIT STRING type is used to represent an array of bits in a portable fashion. It has an additional header beyond the ASN.1 headers that indicates padding as we'll shortly see.

The bits are encoded by placing the first bit in the most significant bit of the first payload byte. The next bit will be stored in bit seven of the first payload byte, and so on. For example, to encode the bit string {1, 0, 0, 0, 1, 1, 1, 0}, we would set bit eight, four, three, and two, respectively. That is, {1, 0, 0, 0, 1, 1, 1, 0} encodes as the byte 0x8E.

The first byte of the encoding specifies the number of padding bits required to complete a full byte. Where bits are missing, we place zeroes. For example, the string {1, 0, 0, 1} would turn into {1, 0, 0, 1, **0**, **0**, **0**, **0**} and the padding count would be four. When zero bits are in the string, the padding count is zero. Valid padding lengths are between zero and seven inclusively.

The length of the payload includes the padding count byte and the bits encoded. Table 2.5 demonstrates the encoding of the previous bit string.

Table 2.5 Example BIT STRING Encoding

Code	ASN.1 Type	Use Of
1	Boolean Type	Store Booleans
2	INTEGER	Store large integers
3	BIT STRING	Store an array of bits
4	OCTET STRING	Store an array of bytes
5	NULL	Place holder (e.g. in a CHOICE)
6	OBJECT IDENTIFIER	Identify algorithms or protocols
16	SEQUENCE and SEQUENCE OF	Container of unsorted elements
17	SET and SET OF	Container of sorted elements
19	PrintableString	ASCII Encoding (omitting several non-printable chars)
22	IA5STRING	ASCII Encoding
23	UTCTIME	Time in a universal format

In Figure 2.4, we see the encoding of the BIT STRING {1, 0, 0, 1} as 0x03 02 04 90. Note that the payload length is 0x02 and not 0x01, since we include the padding byte as part of the payload.

The decoder knows the amount of bits to store as output by computing 8*payload_length – padding_count.

ASN.1 OCTET STRING Type

The OCTET STRING type is like the BIT STRING type except to hold an array of bytes (octets). Encoding this type is fairly simple. Encode the ASN.1 header as with any other type, and then copy the octets over.

For example, to encode the octet string {FE, ED, 6A, B4}, you would store the type 0x04 followed by the length 0x04 and the bytes themselves 0xFE ED 6A B4. It could not be simpler.

ASN.1 NULL Type

The NULL type is the de facto "placeholder" especially made for CHOICE modifiers where you may want to have a blank option. For example, consider the following SEQUENCE.

```
MyAccount ::= SEQUENCE {
   Name      IA5String,
   Group     IA5String,
   Credentials CHOICE {
               rsaKey      RSAPublicKey,
               passwdHash  OCTET STRING,
               none        NULL
   },
   LastLogin UTCTIME,
   …
}
```

In this structure, the account could have an RSA key, a password hash, or nothing as credentials. If the NULL type did not exist, the encoder would have to pick one of the two and have some other context specific "null" type.

The NULL type is encoded as 0x05 00. There is no payload in the DER encoding, whereas technically with BER encoding you could have payload that is to be ignored.

ASN.1 OBJECT IDENTIFIER Type

The OBJECT IDENTIFIER (OID) type is used to represent standard specifications in a hierarchical fashion. The identifier tree is specified by a dotted decimal notation starting with the organization, sub-part, then type of standard and its respective sub-identifiers.

As an example, the MD5 hash algorithm has the OID 1.2.840.113549.2.5, which may look long and complicated but can actually be traced through the OID tree to "*iso(1) member-body(2) US(840) rsadsi(113549) digestAlgorithm(2) md5(5).*" Whenever this OID is found, the decoding application (but not the decoder itself) can realize that this is the MD5 hash algorithm.

For this reason, OIDs are popular in public key standards to specify what hash algorithm was bound to the certificate. OIDs are not limited to hashes, though. There are OID entries for public key algorithms and ciphers and modes of operation. They are an efficient and portable fashion of denoting algorithm choices in data packets without forcing the user (or third-party user) to figure out the "magic decoding" of algorithm types.

The dotted decimal format is straightforward except for two rules:

- The first part must be in the range 0 <= x <= 3.
- If the first part is less than two, the second part must be less than 40.

Other than that, the rest of the parts can hold any positive unsigned value. Generally, they are less than 32 bits in size but that is not guaranteed.

The encoding of parts is a little nontrivial but manageable just the same. The first two parts if specified as x.y are merged into one word 40x + y, and the rest of the parts are encoded as words individually.

Each word is encoded by first splitting it into the fewest number of seven-bit digits without leading zero digits. The digits are organized in big endian format and packed one by one into bytes. The most significant bit (bit eight) of every byte is one for all but the last byte of the encoding per word. For example, the number 30,331 splits into the seven-bit digits {1, 108, 123}, and with the most significant bit set as per the rules turn into {129, 236, 123}. If the word has only one seven-bit digit, the most significant bit will be zero.

Applying this to the MD5 OID, we first transform the dotted decimal form into the array of words. Thus, 1.2.840.113549.2.5 becomes {42, 840, 113549, 2, 5}, and then further split into seven-bit digits with the proper most significant bits as {{0x2A}, {0x86, 0x48}, {0x86, 0xF7, 0x0D}, {0x02}, {0x05}}. Therefore, the full encoding for MD5 is 0x06 08 2A 86 48 86 F7 0D 02 05.

Decoding is rather straightforward except the first word must be split into two parts by first finding the value of the first word modulo 40. The remainder will be the second part. Subtracting that from the word and dividing by 40 will yield the first part.

ASN.1 SEQUENCE and SET Types

The SEQUENCE and SEQUENCE OF and corresponding SET and SET OF types are known as "constructed" types or simply containers. They were provided as a simple method of gathering related data elements into one individually decodable element.

As per the X.690 specification, a SEQUENCE has been defined as having the following properties.

- The encoding shall be constructed.
- The contents of the encoding shall consist of the complete encoding of one data value from each type listed in the ASN.1 definition of the sequence type, in order of appearance, unless the type was referenced with the OPTIONAL or DEFAULT keyword modifiers.

The fact that it is constructed means bit 6 must be set, which turns the SEQUENCE header byte from 0x10 to 0x30. The constructed encoding is simply a nested encoding. For example, consider the following SEQUENCE.

```
User ::= SEQUENCE {
   ID     INTEGER,
   Active BOOLEAN
}
```

When encoding the values {32, TRUE}, we first emit the 0x30 byte to signal this is a constructed SEQUENCE. Next, we emit the length of the payload; that is, the length of the INTEGER and BOOLEAN encodings, which is six bytes so 0x06. Now the constructed part begins. We emit the INTEGER as 0x02 01 20 and the BOOLEAN as 0x01 01 FF. The entire encoding is therefore 0x30 06 02 01 20 01 01 FF. In ASN.1 documentation, they use white space to illustrate the nature of the encoding.

```
0x30 06
        02 01 20
        01 01 FF
```

This notation is particularly useful if you have nested structures such as the following:

```
Account ::= SEQUENCE {
   User SEQUENCE {
      Name    PrintableString,
      Group   PrintableString,
      Credential SEQUENCE {
         PasswdHash   OCTET STRING,
         RSAKey       RSAPublicKey  OPTIONAL
      }
   },
   LastOn   UTCTIME,
   Valid    BOOLEAN
}
```

which, when given the sequence {{"tom","users", {{0x01 02 03 04 05 06 07 08}}, "060416180000Z",TRUE} would encode as

```
Account    0x30 2C
User               30 18
Name                     13 03 74 6F 6D
Group                    13 05 75 73 65 74 75
Credential               30 0A
PasswdHash                     04 08 01 02 03 04 05 06 07 08
LastOn             17 0D 30 36 30 34 31 36 31 38 30 30 30 30 30 5A
Valid              01 01 FF
```

In this example, we clearly see the nesting of the SEQUENCEs in the encoding. In particular, we see the omitted optional RSA key as part of the user credentials. Note that the payload length is the length of all the parts that make up the SEQUENCE.

TIP

The *openssl* command that is installed with the OpenSSL library allows an easy way to convert DER encoded files into human-readable indented print. This is useful for debugging your ASN.1 routines against a known working third-party tool. The following command will read a file and display the decodable elements

openssl asn1parse –inform der –in $INFILE –i

where $INFILE is the file you wish to read. You can omit "-in $INFILE" if you want to read from a pipe.

```
tom@bigbox ~ $ openssl asn1parse -inform DER -in test.der -i
    0:d=0  hl=3 l= 159 cons: SEQUENCE
    3:d=1  hl=2 l=  13 cons:  SEQUENCE
    5:d=2  hl=2 l=   9 prim:   OBJECT            :rsaEncryption
   16:d=2  hl=2 l=   0 prim:   NULL
   18:d=1  hl=3 l= 141 prim:  BIT STRING
```

The first column specifies the offset in the file, "d" specifies the nesting depth, "hl" specifies the header length, and "l" the payload length. The words "cons" and "prim" specify whether it's a constructed (container) or primitive type (bit 6 of the header byte), and the final word specifies the primitive type.

From the indentation, we see that the SEQUENCE that starts at offset 3 is an element within the first SEQUENCE. Similarly, the OBJECT and NULL elements are elements within the second SEQUENCE. Here we also see that OpenSSL recognized the OBJECT as an "rsaEncryption" blob, in this case it is a public key.

SEQUENCE OF

A SEQUENCE OF is related to a SEQUENCE with the exception that it is a container of one type. This is the ASN.1 equivalent of an array. The encoding of a SEQUENCE OF contains zero or more encodings of the listed ASN.1 type in the order specified by the encoder (as presented). SEQUENCE OF uses the same 0x30 header byte to signify its part of the SEQUENCE family. This implies the decoder has to be able to read both SEQUENCE and SEQUENCE OF types.

SET

A SET is a constructed type like a SEQUENCE with the exceptions that the header byte is 0x31 instead of 0x30 and the order of the encodings of the constituent members is *not* the order specified by the SET definition. Strictly speaking, for BER encoding the order is decidable by the sender. This means that the SET cannot contain two identical types without first transmitting the SET order to the receiver. With DER encoding rules, the order is dictated by order of the type values in ascending order. If two elements have the same type, their original order in the submitted SET determines the tiebreaker. That is, the first occurrence of a repeated type is the winner.

Consider the previous SEQUENCE but instead encoded as a SET.

```
User ::= SET {
   ID     INTEGER,
   Active BOOLEAN

}
```

When encoding the values {32, TRUE}, we first emit the 0x31 byte to signal this is a constructed SET. We know the length is six bytes from before; this will not change for a set. So, we now emit the 0x06 byte. Now, according to DER rules we sort the elements based on their types first. Since BOOLEAN has a type of 0x01 and INTEGER the type 0x02, the BOOLEAN comes first. Therefore, the complete encoding is 0x31 06 01 01 FF 02 01 20.

The SET listed here contains a collision.

```
User ::= SET {
   ID       INTEGER,
   Active   BOOLEAN,
   LogCount INTEGER
}
```

In this case, both ID and LogCount have the same type of INTEGER. The encoding of the instance {32, TRUE, 1023} would start with the 0x31 header byte, followed by the length byte; in this case, 0x0A. Next, we encode the BOOLEAN since its type is numerically lower to 0x01 01 FF. Both ID and LogCount have the same type, but ID occurred first so it is stored next as 0x02 01 20. Finally, LogCount is stored as 0x02 02 3F FF. Therefore, the complete encoding is 0x31 0A 01 01 FF 02 01 20 02 02 3F FF.

SET OF

The SET OF type is the SET analogous to a SEQUENCE OF. According to BER, the order of the elements does not matter. In the case for DER rules, instead of sorting on the type, we sort based on the ASN.1 DER encoding of the constituent elements in ascending order. Consider the array INTEGERS {1, 10007, 0, 20, –300}; they individually encoded as shown in Table 2.6.

Table 2.6 Array of INTEGERs

Number	Encoding
1	0x02 01 01
10007	0x02 02 27 17
0	0x02 00
20	0x02 01 14
–300	0x02 02 FE D4

When we sort the encodings from Table 2.6 in ascending order, we find the listing shown in Table 2.7 as the order in which they are to be encoded.

Table 2.7 Sorted SET OF INTEGERs

Number	Encoding
0	0x02 00
1	0x02 01 01
20	0x02 01 14
10007	0x02 02 27 17
–300	0x02 02 FE D4

The first thing to note about the sorted data is that for a given SET OF, the encodings are always first sorted based on their length and then by their payload. This is by virtue of the length encoding ASN.1 uses. Next, we note that the sorting does not always make logical sense for the given data we are sorting. The value of –300 appears last even after the value of zero, even though as an INTEGER it represents a lower number.

The purpose of the sorting with DER is simply to ensure that the encoding is deterministic (or distinguished as per ASN.1 specifications) regardless of the order of the inputs as presented to the encoder.

The encoding of this array as a SET OF would therefore be 0x31 10 02 00 02 01 01 02 01 14 02 02 27 17 02 02 FE D4.

ASN.1 PrintableString and IA5STRING Types

The PrintableString and IA5STRING types define portable methods of encoding ASCII strings readable on any platform regardless of the local code page and character set definitions.

PrintableString encodes a limited subset of the ASCII set including the values 32 (space), 39 (single quote), 40–41, 43–58, 61, 63, and 65–122. Anything outside those ranges is invalid and should signal an error. PrintableString is meant for characters that can be printed on most terminals without changing the flow of the text being displayed. For this reason, it omits the values below 32.

IA5STRING encodes most of the ASCII set, including NUL, BEL, TAB, NL, LF, CR, and the ASCII values from 32 through to 126 inclusive. Generally, IA5 is not safe to display with a TTY without filtering, as it allows the encoded value to do things like blank the screen, replace characters, and the like, depending on the terminal type used.

The encoding of both is similar to that of the OCTET STRING except for the restrictions and the different header byte. PrintableString uses 0x13 and IA5STRING uses 0x16 for the header byte. For example, the string "Hello World" would encode as 0x13 0B 48 65 6D 6D 6F 20 57 6F 72 6D 64.

Note that it is the responsibility of both the encoder and the decoder to verify the values are within range for the specified data type.

ASN.1 UTCTIME Type

The UTCTIME defines a standard encoding of time (and date) relative to GMT. Earlier drafts of ASN.1 allowed time offsets (zones) and collapsible encodings (such as omitting seconds). This meant for DER at least, that there were six possible ways to encode a date provided the seconds were zero. As of the 2002 draft of X.690, all UTC encodings shall follow the format "YYMMDDHHMMSSZ," which is year, month (1–12), day (0–31), hour (0–23), minute (0–59), and second (0–59).

The "Z" is legacy from the original UTCTIME where the absence of the "Z" would allow two additional groups the "[+/-]hh'mm'," which were the hours (hh') and minutes (mm') offset from GMT (either positive or negative). The presence of "Z" means the time was represented in Zulu or GMT time.

The encoding of the string follows the IA5STRING rules for character to byte conversion (that is, using the ASCII character set), except the ASN.1 header byte is 0x17 instead of

0x16. For example, the encoding of July 4, 2003 at 11:33 and 28 seconds would be “030704113328Z” and be encoded as 0x17 0D 30 33 30 37 30 34 31 31 33 33 32 38 5A.

Implementation

Now we will consider how to implement ASN.1 encoders and decoders. Fortunately, most ASN.1 types are primitive and fairly simple to process. The constructed types are slightly harder to develop if the intent is to have a user-friendly API. In our case, we're going to strive for maximum effort while writing the ASN.1 routines such that the resulting code has the maximal amount of use.

All of the ASN.1 routines are found in the “ch2” directory of the source code repository. There is a collection of C source files, a single H header file to gather up the prototypes, and a GNU Makefile that will build the collection into an archive using GCC.

The first routines we examine deal with getting, reading, and encoding the length of ASN.1 encodings. The logic is shared by all other ASN.1 types, and as such, makes sense to re-use the code where possible.

ASN.1 Length Routines

The first routine simply returns the length of an encoding, including the header, length bytes, and payload.

```
der_length.c:
#include “asn1.h"
unsigned long der_length(unsigned long payload)
{
   unsigned long x;

   if (payload > 127) {
      x = payload;
      while (x) {
        x >>= 8;
        ++payload;
      }
   }

   return payload + 2;
}
```

The function accepts as input the payload length and returns the size of the eventual encoding. This function is suitable for all types where the payload length is known in advance; that is, primitive (nonconstructed) types. Note that for the BIT STRING type the calling function will have to add in the padding counter byte to the payload length for this to work.

This function is useful for encoders, as it allows the caller to know the length of the eventual output and report an error when a buffer overflow occurs. This simple check before encoding begins can save a lot of hassle down the road, provided the check has been consistently applied.

Encoding an ASN.1 header is not hard, but we shall put a twist on the encoding. Since the function will be used from encoders, it makes sense to make it something an encoder can call with contextual data it will maintain during the encoding process.

```
der_put_header_length.c:
void der_put_header_length(unsigned char **out,
                           unsigned primitive_type,
                           unsigned long payload_length,
                           unsigned long *outlen)
{
   unsigned char *ptr;
   unsigned long x, y, pl;

   ptr = *out;

   /* store header */
   *ptr++ = primitive_type;

   /* encode payload length */
   if (payload_length < 128) {
      *ptr++ = payload_length;
   } else {
      /* determine length of length */
      x = payload_length;
      y = 0;
      while (x) {
        ++y;
        x >>= 8;
      }

      /* store length of length  */
      *ptr++ = 0x80 | y;

      /* align length on 32-bit boundary, we assume y < 5 */
      x  = y;
      pl = payload_length;
      while (x < 4) {
         pl <<= 8;
         ++x;
      }

      /* store it */
      while (y--) {
         *ptr++ = (pl >> 24) & 0xFF;
      }
   }

  /* get stored size */
  *outlen = (ptr - *out) + payload_length;

```

This function will store the ASN.1 header and the length in a buffer specified by the caller. The pointer to the buffer is actually passed as a pointer to a pointer. This allows this function to update the output pointer and have the caller be able to resume encoding after the header.

This function assumes the payload is less than four gigabytes, which is fairly practical. The encoding of the length when it is larger than 127 bytes is performed by shifting the nonzero most significant bytes out of the payload length (line 30). This makes extracting the length in big endian format (line 36), as required by ASN.1 a simple matter of extracting the most significant byte and then shifting the length up by eight bits.

Note that this function does not check the output buffer length and it is up to the caller to ensure the output is large enough before calling this. The function maintains an internal copy of the output pointer in *ptr* and copies it out before exiting. This was performed not strictly for performance reasons, but to make the code simpler to read.

Now that we can encode headers, we will want to be able to decode them as well. The decoder function should in theory have a similar prototype, as it is merely the opposite direction as the encoder. In this function, we introduce our first function, which can have a fail condition.

```
der_get_header_length.c:
001   unsigned long der_get_header_length(unsigned char **in,
002                             unsigned long   inlen,
003                             unsigned       *primitive_type,
004                             unsigned long  *payload_length)
005   {
006      unsigned long x, y;
007      unsigned char *ptr;
008
009      ptr = *in;
010
011      /* ensure inlen is at least two */
012      if (inlen < 2) {
013         return -1;
014      }
015
016      /* get the type and first length byte */
017      *primitive_type = *ptr++;
018      y = *ptr++;
019
020      if (y < 128) {
021         *payload_length = y;
022      } else {
023         y &= 0x7F;  /* strip off bit 8 */
024
025         /* simple safety check */
026         if (y > 4 || (2 + y) > inlen) {
027            return -1;
028         }
029
030         /* read in the length */
031         x = 0;
```

```
      while (y--) {
         x = (x << 8) | *ptr++;
      }
      *payload_length = x;
   }

   *in = ptr;
   return 0;
}
   }

  /* get stored size */
  *outlen = (ptr - *out) + payload_length;

   /* update the output pointer */
  *out = ptr;

}
```

This function decodes the ASN.1 header, and if successful will store the primitive type and the payload length in the pointers passed to by the caller. This function introduces our error signaling mechanism, which is to return nonzero when an error occurs.

This function also requires the caller to pass the length of the input as a parameter. This is used to prevent buffer overruns, which are used by attackers to try to read data off the stack (or heap depending on where this was read from). Besides checking the input length for read errors, the function performs the sanity check and requires all payload lengths to be less than four gigabytes.

We also note this function will store the final output length. This is useful as it ensures that any encoder that uses this function will automatically have the output length set for the caller (to the encoder).

Note that for the BIT STRING type, the payload length returned will include the padding count byte.

ASN.1 Primitive Encoders

Now that we can parse the ASN.1 header, we can begin processing ASN.1 types. Each primitive type will provide three routines: a length determination function, an encoder, and a decoder.

The length determination functions are called der_XYZ_length() and accept as input (where appropriate) the input to be encoded and return the entire encoding length. These functions are useful for determining if an encoding can take place in the buffer provided. It is also useful in processing constructed types.

The encoder and decoder are relatively self-explanatory; they are der_XYZ_encode() and der_XYZ_decode(), respectively. Both the encoder and decoder can fail, so their return values should be inspected by the caller. The encoder can fail if the input is invalid or the output buffer too small. The decoder can fail for the same reasons as failing sanity checks.

BOOLEAN Encoding

We start with the first ASN.1 types in the order of their type values. Fortunately for us, that means BOOLEAN is the first type, which is also the simplest to deal with.

The length determination function is fairly trivial.

```
der_boolean_length.c:
unsigned long der_boolean_length(void)
{
   return 3;
}
```

As we can see, this function has only four lines. All BOOLEAN encodings are three bytes long regardless of whether they are true or false. It is important to note that the length determination functions can fail. In such cases, we will use zero length as our error indicator. In the case of BOOLEAN, all inputs are valid.

```
der_boolean_encode.c:
001   #include “asn1.h"
002   int der_boolean_encode(int            bool,
003                          unsigned char *out,
004                          unsigned long *outlen)
005   {
006      /* check output size */
007      if (der_boolean_length() > *outlen) {
008         return -1;
009      }
010
011      /* store header and length  */
012      der_put_header_length(&out, ASN1_DER_BOOLEAN, 1);
013
014      /* store payload */
015      *out = (bool == 0) ? 0x00 : 0xFF;
016
017      /* finished ok */
018      return 0;
019   }
```

This function will encode a BOOLEAN in the ASN.1 DER format. It accepts *bool* as the boolean to encode. The value can be zero to indicate false, or any nonzero to indicate true.

The function makes use of the der_boolean_length() function (line 7) to determine if the output buffer is large enough to hold the encoding. Even though we know the encoding is always three bytes, we still use the function, as it will be the pattern all other ASN.1 encoders will use. It is a habit worth getting into, especially considering this function will not be a performance bottleneck.

After we check the length of the output buffer, we store the ASN.1 header using the der_put_header_length() function (line 12). This line also uses the ASN1_DER_BOOLEAN symbol, which was defined in the “asn1.h” header file. This function call stores the ASN.1 header and length byte for us.

When we return from the function call to der_put_header_length(), the output pointer will have been updated to point at the byte just past the end of the ASN.1 header. This allows us to begin encoding the payload without having to know the length of the ASN.1 header. The payload in this case is simply 0x00 or 0xFF depending on whether the input is false or true.

Now, to decode ASN.1 DER BOOLEANs, we proceed with a similarly “stock” function.

```
der_boolean_decode.c:
001   #include “asn1.h"
002   int der_boolean_decode(unsigned char *in,
003                          unsigned long  inlen,
004                          int           *bool)
005   {
006      unsigned type;
007      unsigned long payload_length;
008      int      ret;
009
010      /* decode header */
011      ret = der_get_header_length(&in, inlen,
012                                  &type, &payload_length);
013      if (ret < 0) {
014         return ret;
015      }
016
017      /* payload must be 1 byte and 0x00 or 0xFF */
018      if (type           != ASN1_DER_BOOLEAN ||
019          payload_length != 1 ||
020          (in[0] != 0x00 && in[0] != 0xFF)) {
021          return -2;
022      }
023
024      /* decode payload */
025      *bool = (in[0] == 0xFF) ? 1 : 0;
026
027      return 0;
028   }
```

The first thing we attempt is to decode the ASN.1 header and make sure we can read a payload length. The function call der_get_header_length() (line 11) does this all for us in one function call. The return value is less than zero if the function failed, so we return any error code directly to the caller. As in the encoder case, our input pointer was updated to point to the first byte of the payload.

After we have parsed the ASN.1 header, we verify that the type, length, and payload are all valid (lines 18 through 20). We will return –2 to signal a decoding error when the contents are invalid. Once we have verified the packet, we store the boolean back in the pointer passed by the caller.

INTEGER Encoding

Encoding an integer is fairly straightforward when the numbers are positive. In the case of positive numbers, the encoding is simply the byte encoding of the integer stored in big endian format. For negative numbers, we need to be able to find a power of 256 larger than the absolute value and encode the sum of the two.

The ASN.1 specification does not put a small limit on the size of the integer to be encoded. So, long as the length can be encoded (less than 128 bytes of length), the integer is within range. However, at this point we do not have any way of dealing with large integers. Therefore, we are going to *artificially* limit ourselves to using the C *long* data type. The reader is encouraged to take note of the limited functionality required to properly support the INTEGER type.

Before we get into encoding and decoding, we must develop a few helper functions for dealing with integers. These functions are commonplace in the typical large number library and trivial to adapt.

```
int_help.c:
#include “asn1.h"

int count_bits(long num)
{
   int x;
   x = 0;
   while (num) { ++x; num >>= 1; }
   return x;
}
```

This function returns the number of bits in the integer. It assumes the integer is positive (or at least not sign extended) and simply shifts the integer right until it is zero.

```
int count_lsbs(long num)
{
   int x;
   x = 0;
   if (!num) return 0;
   while (!(num&1)) { ++x; num >>= 1; }
   return x;
}
```

This function counts the number of consecutive zero least significant bits in the integer. It also assumes the integer has not been sign extended. Note that this function is one-based, not zero-based. For example, if the number is 10_2, the returned value is one, 100_2 returns 2, and so on.

```
void store_unsigned(unsigned char *dst, long num)
{
   int x, y;
   unsigned char t;

   x = y = 0;
   while (num) {
```

```
027         dst[x++] = num & 255;
028         num >>= 8;
029     }
030
031     /* reverse */
032     --x;
033     while (y < x) {
034        t = dst[x]; dst[x] = dst[y]; dst[y] = t;
035        --x; ++y;
036     }
037  }
```

This function stores a positive integer in big endian byte format. The first loop (line 26) extracts the bytes from the number. At this point, the *dst* array holds the representation of the integer in little endian format. This is because we were storing the least significant byte in each of the iterations of the loop.

The next loop (line 33) swaps the bytes around so they are in big endian format. This is accomplished by starting at both ends, swapping the bytes, and moving inward.

```
039  long read_unsigned(unsigned char *dst, unsigned long len)
040  {
041     long tmp;
042
043     tmp = 0;
044     while (len--) {
045        tmp = (tmp << 8) | *dst++;
046     }
047     return tmp;
048  }
049
```

This function reads bytes and stores them in an integer. Since it shifts upward, it will always interpret the input in big endian format. There is no need to swap in this function.

These four functions are all we have to provide at this stage to properly handle ASN.1 INTEGER types inside the encoder and decoder. We shall begin with the length function. In this case, we introduce a new function, "paylen," which determines only the payload length of the encoding and not the complete encoding length.

The paylen function will come in handy to judge the size of the final output and tell the header encoder what payload length to use.

```
#include “asn1.h"
unsigned long der_integer_paylen(long num)
{
   unsigned long x, y, pad, paylen;

   if (num >= 0) {
      /* it's positive */
      /* count # of bits  */
      x = count_bits(num);

      /* if the 8th bit is set we pad */
012         if ((x & 7) == 0) {
013            pad = 1;
014         } else {
015            pad = 0;
016         }
017
018         /* round count up to the next byte */
019         x = x + ((8 - x) & 7);
020         paylen = (x >> 3) + pad;
021      } else {
022         /* it's negative */
023         x = count_bits(-num);
024         y = count_lsbs(-num);
025
026         /* round count up to the next byte */
027         paylen = x + ((7 - x) & 7);
028         paylen = (paylen >> 3) + 1;
029
030         /* if lsbs+1==bits and bits mod 8 == 0 reduce by 1 */
031         if ((y+1)==x && !(x & 7)) {
032            --paylen;
033         }
034      }
035      return paylen;
036   }
```

This function computes the payload length of the INTEGER. First, we determine if the number is positive (line 6), and handle positive and negative numbers differently. In the case of positive numbers, we simply count the number of bits in the representation, add padding if required (line 12), and round up to the next byte (line 19).The padding required is for when the most significant bit of the first byte is set. In this case, we must add a leading zero byte to ensure the encoding will be interpreted as positive.

For negative numbers, we need both the count of bits and leading zero least significant bits. First, we round upward to the next byte and add an additional byte.This additional byte is required, as we are computing the addition of the next power of 256, which is one byte too large.

This opens an exception for numbers of the form $-(256^k)/2$, which would be redundantly encoded with leading 0xFF 80 bytes, which is forbidden by the ASN.1 DER specification.

```
039   unsigned long der_integer_length(long num)
040   {
041      /* get payload length */
042      return der_length(der_integer_paylen(num));
043   }
```

Now we simply return the entire DER encoding length by using the der_length() function on the payload length.

```
der_integer_encode.c:
001   #include “asn1.h"
```

```
002   int der_integer_encode(         long  num,
003                          unsigned char *out,
004                          unsigned long *outlen)
005   {
006      /* check output size */
007      if (der_integer_length(num) > *outlen) {
008         return -1;
009      }
010
011      /* encode header */
012      der_put_header_length(&out, ASN1_DER_INTEGER,
013                            der_integer_paylen(num), outlen);
014
015      /* store number */
016      if (num >= 0) {
017         /* leading msb? */
018         if ((count_bits(num) & 7) == 0) {
019            *out++ = 0x00;
020         }
021         store_unsigned(out, num);
022      } else {
023         /* find power of 256 greater than it */
024         int x, y, z;
025         long tmp;
026
027         x = count_bits(-num);
028         y = count_lsbs(-num);
029         z = ((x + ((7 - x) & 7)) >> 3) + 1;
030
031         /* handle special case */
032         if ((y+1)==x && !(x & 7)) {
033             --z;
034         }
035
036         /* get our constant */
037         tmp = 1L << (z << 3);
038
039         /* encoding */
040         store_unsigned(out, tmp + num);
041      }
042
043      return 0;
044   }
```

As in the case of the BOOLEAN encoder, we first check if we have enough space using the der_integer_length() function and next encode the ASN.1 header. At this point, we split the encoder into two paths based on whether the INTEGER to encode is negative or positive.

When the INTEGER is positive, we simply output any leading zero bytes required (depending on the MSB of the first byte) and then store the INTEGER in byte format (line 21).

When the INTEGER is negative, we have to find the next power of 256 larger than the absolute value of the INTEGER. This is accomplished much as in the case of the der_integer_length() function. To actually compute the power, we use a simple left shift (line 37). Finally, we store the sum by encoding it in twos complement format.

```
der_integer_decode.c:
001   #include “asn1.h"
002
003   int der_integer_decode(unsigned char *in,
004                          unsigned long  inlen,
005                          long *num)
006   {
007      unsigned type;
008      unsigned long payload_length;
009      long     tmp;
010      int      ret;
011
012      /* decode header */
013      ret = der_get_header_length(&in, inlen,
014                                  &type, &payload_length);
015      if (ret < 0) {
016         return ret;
017      }
018
019      if (type != ASN1_DER_INTEGER) {
020         return -2;
021      }
022
023      /* read in the value */
024      tmp = read_unsigned(in, payload_length);
025
026      /* if the leading byte has 0x80 set it's negative */
027      if (in[0] & 0x80) {
028         /* it's negative */
029         *num = tmp - (1L << (payload_length << 3));
030      } else {
031         *num = tmp;
032      }
033
034      return 0;
035   }
```

This decodes the INTEGER type by first decoding the header, reading in the INTEGER, and then parsing it based on the most significant bit of the first byte of payload. If the bit is (line 27), the number is negative and we have to add the next leading power of 256 to the number.

BIT STRING Encoding

BIT STRINGs are arrays of bits encoded eight per byte from most significant bit to least significant bit. There are up to seven padding bits to ensure the payload is a proper multiple of eight bits in length. As mentioned earlier, the payload length encoded in the ASN.1 header includes the single byte required to indicate the padding length.

der_bitstring_length.c:

```
#include “asn1.h"
unsigned long der_bitstring_length(unsigned long nbits)
{
   unsigned long bytes;

   /* get # of payload bytes */
   bytes = 1 + (nbits >> 3) + ((nbits & 7) ? 1 : 0);

   return der_length(bytes);
}
```

This function computes the length of the BIT STRING by rounding up the bit count and adds the padding counte byte.

der_bitstring_encode.c:

```
#include “asn1.h"
int der_bitstring_encode(unsigned char *bits,
                         unsigned long  nbits,
                         unsigned char *out,
                         unsigned long *outlen)
{
   unsigned long bytes, bitbuf, bitcnt, tcnt;

   /* check output size */
   if (der_bitstring_length(nbits) > *outlen) {
      return -1;
   }

   /* store header and length  */
   bytes = 1 + (nbits >> 3) + ((nbits & 7) ? 1 : 0);
   der_put_header_length(&out, ASN1_DER_BITSTRING,
                          bytes, outlen);

   /* store padding count */
   *out++ = (8 - nbits) & 7;

   /* accumulate bits */
   tcnt   = nbits;
   bitbuf = bitcnt = 0;
   while (tcnt--) {
      bitbuf = (bitbuf << 1) | (*bits++ & 1);
      if (++bitcnt == 8) {
         *out++ = bitbuf;
         bitbuf = bitcnt = 0;
      }
   }

   /* pad any remaining bytes */
   if (nbits & 7) {
      bitbuf <<= ((8 - nbits) & 7);
      *out++ = bitbuf;
037      }
038
039      return 0;
040   }
```

After the initial formalities, the encoding proceeds by reading through all of the supplied bits and packing them eight at a time into a byte (line 26). When a byte is full (line 27), it will be stored to the output and the buffer reset (lines 28–29).

Any remaining bits are appropriately shifted upward (line 35), emulating the insertion of padding bits and stored to the output.

```
der_bitstring_decode.c:
001   #include “asn1.h"
002   int der_bitstring_decode(unsigned char *in,
003                            unsigned long  inlen,
004                            unsigned char *out,
005                            unsigned long *outlen)
006   {
007      unsigned type;
008      unsigned long payload_length, nbits, bitbuf, bitcnt;
009      int      ret;
010
011      /* decode header */
012      ret = der_get_header_length(&in, inlen,
013                                  &type, &payload_length);
014      if (ret < 0) {
015         return ret;
016      }
017
018      if (type != ASN1_DER_BITSTRING || payload_length < 1) {
019         return -2;
020      }
021
022      /* get # of bits */
023      nbits = ((payload_length - 1) << 3) - in[0];
024      ++in;
025
026      /* too many? */
027      if (nbits > *outlen) {
028         return -1;
029      }
030      *outlen = nbits;
031
032      /* start decoding */
033      bitbuf = *in++;
034      bitcnt = 8;
035
036      while (nbits--) {
037         *out++   = (bitbuf & 0x80) >> 7;
038         bitbuf <<= 1;
039         if (--bitcnt == 0) {
040             bitbuf = *in++;
041             bitcnt = 8;
042         }
043      }
044      return 0;
045   }
```

As per usual, we decode the ASN.1 header to retrieve both the type and length. Next, we verify the type is correct and the payload is at least one byte (line 18). The payload must be at least one byte to accommodate the padding count byte.

Next, we compare the output length and store the number of bits in the caller supplied output length (lines 27–30). We compute the number of output bits by multiplying the payload by eight (shift left by three) and subtracting the padding count (in[0]).

Just like the encoder, we set up a bit buffer to process the input. The bitbuf variable holds the current byte being processed, and bitcnt holds the number of bits left. The bits are read from the most significant bit downward and stored into the output array (lines 36–42).

OCTET STRING Encodings

OCTET STRING encoding is by far the simplest to process. There are no invalid inputs to the encoder and the payload length is equal to the length of the input.

```
der_octetstring_length.c:
#include “asn1.h"
unsigned long der_octetstring_length(unsigned long noctets)
{
   return der_length(noctets);
}
```

This rather simple looking function computes the DER encoded length of an OCTET STRING with a given number of octets.

```
der_octetstring_encode.c:
#include “asn1.h"
int der_octetstring_encode(unsigned char *octets,
                           unsigned long  noctets,
                           unsigned char *out,
                           unsigned long *outlen)
{
   /* check output size */
   if (der_octetstring_length(noctets) > *outlen) {
      return -1;
   }

   /* store header and length  */
   der_put_header_length(&out, ASN1_DER_OCTETSTRING,
                          noctets, outlen);

   /* store bytes */
   memcpy(out, octets, noctets);

   return 0;
}
```

By far, this is the simplest of the nontrivial encoders. It checks the output length (line 8), emits a header (line 13), and then simply copies the input to the output (line 17).

TIP

Many innocent looking C functions such as memcpy, memcmp, malloc, and free (among others) can be hazardous to embedded development platforms. In many cross-compiler development environments, it is possible to not have a complete standard C library, or a library at all.

The heap is another thorny issue. With many platforms, free memory space is tightly regulated, which often means it will be managed by the application.

A relatively simple solution to this problem adopted by the LibTom projects is to use C pre-processing macros for common functions. A definition such as XMALLOC can be programmed to default to malloc with the following code.

```
#ifndef XMALLOC
#define XMALLOC malloc
#endif
```

In this case, if XMALLOC was defined before this is sent to the pre-processor, it is possible to redirect XMALLOC “calls” to other functions. For example,

```
CFLAGS="-DXMALLOC=mymalloc" gcc myprog.c –lmylib –o myprog
```

Now within the program instead of calling malloc() directly, you would simply call XMALLOC. For example,

```
unsigned char *buffer = XMALLOC(buffer_size);
```

This technique can be applied to all other standard C functions within the scope of the application.

For completeness, here is the OCTET STRING decoder.

```
der_octetstring_decode.c:
#include “asn1.h"
int der_octetstring_decode(unsigned char *in,
                           unsigned long  inlen,
                           unsigned char *out,
                           unsigned long *outlen)
{
   unsigned type;
   unsigned long payload_length;
   int      ret;

   /* decode header */
   ret = der_get_header_length(&in, inlen,
                               &type, &payload_length);
   if (ret < 0) {
      return ret;
   }

   if (type != ASN1_DER_OCTETSTRING) {
      return -2;
   }

   /* check output size */
   if (payload_length > *outlen) {
      return -1;
   }

   /* copy out */
   *outlen = payload_length;
   memcpy(out, in, payload_length);

   return 0;
}
```

NULL Encoding

The NULL type is the simplest of all encodings. It can only be represented in one method and a fixed length. For completeness, here are the NULL routines.

```
der_null_length.c:
unsigned long der_null_length(void)
{
  return 2;
}
```

```
der_null_encode.c:
#include “asn1.h"
int der_null_encode(unsigned char *out,
                    unsigned long *outlen)
{
   if (*outlen < 2) return -1;

   out[0]  = ASN1_DER_NULL;
   out[1]  = 0x00;
   *outlen = 2;

   return 0;
}
```

```
der_null_decode.c:
#include “asn1.h"
int der_null_decode(unsigned char *in,
                    unsigned long inlen)
{
   if (inlen != 2 || in[0] != ASN1_DER_NULL || in[1] != 0x00) {
      return -2;
   }
   return 0;
}
```

In both the encoder and decoder, we simply store or read the data directly without using the helper functions. Normally, that is a bad coding practice and it certainly is a bad habit to get into. In this case, though, calling the functions would actually take more lines of code, so we make the exception. The reader should note that even though we know the type of NULL is 0x05, we still use the symbol ASN1_DER_NULL. This is certainly a worthwhile practice to adhere to.

OBJECT IDENTIFIER Encodings

OBJECT IDENTIFIER (OID) types are encoded somewhat like positive INTEGERS, except instead of using eight-bit digits we use seven-bit digits. OIDs are the collection of several positive unsigned numbers (called words), so as far as encodings are concerned they are back to back. The most significant bit of each byte represents whether this is the last seven-bit digit of a given word from the OID.

The first two words of an OID specify which standards body the OID refers to. They are special in that the first word must be in the range 0 to 3, and the second word must be in the range 0 to 39. When we encode these, the first two words are encoded as one single unsigned value by multiplying the first word by 40 and adding the second word. The rest of the words are encoded individually.

```
der_oid_length.c:
#include “asn1.h"

unsigned long der_oid_paylen(unsigned long *in,
                             unsigned long inlen)
{
   unsigned long wordbuf, y, z;

   /* form first word */
   wordbuf = in[0] * 40 + in[1];
   z       = 0;
   for (y = 1; y < inlen; y++) {
      if (wordbuf == 0) {
         ++z;
      } else {
         /* count the # of 7 bit digits in wordbuf */
         while (wordbuf) {
            ++z;
            wordbuf >>= 7;
         }
      }
      if (y < inlen - 1)
         wordbuf = in[y + 1];
   }
   return z;
025   }
```

Here we see the introduction of the payload length function; this is because finding the payload length is nontrivial and is best left to a separate function. The first word encoded is actually the first two inputs joined together as previously mentioned. To keep a consistent flow of logic we use a buffer *wordbuf* to hold the current word that would be encoded to determine the number of seven-bit digits it contains.

We make an exception (line 12) for words that are zero, as they have no nonzero seven-bit digits but still require at least one byte of output to be properly represented. After the exception, we begin to extract seven-bit digits from the *wordbuf* variable. If this is not the last loop of the algorithm (line 21), we fetch the next word and iterate.

SECURITY!

On line 21 of der_oid_paylen(), we check to see if we are at the end of the loop before reading in the next word on line 22. On most platforms, the read on line 22 would not cause a problem even if we were at the last iteration. This can be masked by being within a stack frame or a heap that is sufficiently larger than the array being read.

There are occasions when this could cause problems, from a stability point of view and security. In certain x86 models, a segment can be limited to very small sizes, or reading past the end of the array could cross a page boundary (causing a page fault). Therefore, it is entirely possible to crash a program from a simple read operation.

There are also security problems if we performed the read. It is possible that wordbuf is placed on the stack, which means whatever is passed the end of the array (perhaps a secret key?) is also placed on the stack. While this function alone would not directly leak a secret, another function that has not overwritten that part of the stack could easily leak the stack contents.

This security concept revolves around asset management, and while innocent looking enough is very important for the developer to be constantly aware of.

```
027   unsigned long der_oid_length(unsigned long *in,
028                                unsigned long inlen)
029   {
030      if (in[0] > 3 || in[1] > 39) {
031         return 0;
032      }
033      return der_length(der_oid_paylen(in, inlen));
034   }
```

This function is highly dependent on the payload length function. It initially checks the first two words to make sure they are within range and then computes the DER encoded length.

```
der_oid_encode.c:
001   #include “asn1.h"
002   int der_oid_encode(unsigned long *in,
003                      unsigned long  inlen,
004                      unsigned char *out,
005                      unsigned long *outlen)
006   {
007      unsigned long x, y, z, t, wordbuf, mask;
008      unsigned char tmp;
009      /* check output size */
010      x = der_oid_length(in, inlen);
011      if (x == 0 || x > *outlen) {
012         return -1;
013      }
014
015      /* store header and length  */
016      der_put_header_length(&out, ASN1_DER_OID,
017                             der_oid_paylen(in, inlen),
018                             outlen);
019
```

So far, this is the standard encoding practice. Since the OID inputs can be invalid, we check for an overflow and invalid input (line 11), which would be indicated by der_oid_length() returning a zero. Note the usage of der_oid_paylen() here as well. Re-using the functionality has saved quite a bit of trouble, as we do not have to re-invent the wheel.

```
020      /* encode words */
021      wordbuf = in[0] * 40 + in[1];
022      for (y = 1; y < inlen; y++) {
```

Here, the first word to be encoded (line 21) is the first two words joined into one word. Also note our loop starts at one instead of zero as expected.

```
023          if (wordbuf) {
024             /* mark current spot and clear mask */
025             x    = 0;
026             mask = 0x00;
027             while (wordbuf) {
028                out[x++]   = (wordbuf & 0x7F) | mask;
029                wordbuf >>= 7;
030                mask     |= 0x80;
031             }
```

At this point, if *wordbuf* was not originally zero, the array *out[0..x-1]* will contain the encoding of the word in little endian. The use of *mask* allows us to set the most significant bit of all but the first digit stored.

```
033             /* now swap x-1 bytes */
034             z = 0;
```

```
035           t = x--;
036           while (z < x) {
037               tmp = out[z]; out[z] = out[x]; out[x] = tmp;
038               ++z; --x;
039           }
```

As with our INTEGER routines, we have swapped the endianess of the output to big endian. The variable *t* holds the length of this word. We post decrement it since we are swapping from 0 to *x*–1, not 0 to *x*.

```
041           /* move pointer */
042           out += t;
043        } else {
044           *out++ = 0x00;
045        }
046        if (y < inlen - 1) {
047           wordbuf = in[y+1];
048        }
049     }
050     return 0;
051  }
```

We also handle the zero case (line 44) by storing a single 0x00 byte. Since the most significant bit is not set, the decoder will interpret it as a single byte for the word. After the word has been encoded, we fetch the next word if we are not finished; otherwise, we return.

```
der_oid_decode.c:
#include “asn1.h"
int der_oid_decode(unsigned char *in,
                   unsigned long  inlen,
                   unsigned long *out,
                   unsigned long *outlen)
{
   unsigned type;
   unsigned long payload_length, wordbuf, y;
   int      ret;

   /* decode header */
   ret = der_get_header_length(&in, inlen,
                               &type, &payload_length);
   if (ret < 0) {
      return ret;
   }

   /* check type and enforce a minimum output size of 2 */
   if (type != ASN1_DER_OID || *outlen < 2) {
      return -2;
   }
```

So far, this is the fairly standard decoder logic. We also perform the initial size check for the output. Since all OIDs must have at least the first two words, an output length cannot be smaller than two.

Note that we do not count the number of words prior to decoding. This is because we are not allocating resources for every decoded word. In the event we have to allocate resources (say heap), it is simpler to first count and allocate resources before decoding. This helps prevents leaks, as the resource usage becomes all or nothing.

```
023      wordbuf = 0;
024      y       = 0;
025      while (payload_length--) {
026         wordbuf = (wordbuf << 7) | (*in & 0x7F);
```

Like the INTEGER case, we read in bytes and shift them upward. In this case, we only use the lower seven bits of every digit read. The while loop runs over all of the payload bytes, and in theory, when this while loop terminates all of the OID words will have been read in.

```
027         if (!(*in++ & 0x80)) {
028            /* last 7 bit digit */
029            if (y == 0) {
030               /* first words */
031               out[0] = wordbuf / 40;
032               out[1] = wordbuf % 40;
033               y      = 2;
034            } else {
035               if (y < *outlen) {
036                  out[y++] = wordbuf;
037               } else {
038                  return -2;
039               }
040            }
041            wordbuf = 0;
042         }
043      }
044
045      /* store size */
046      *outlen = y;
047
048      return 0;
049   }
```

In the event the most significant bit is not set (see line 27), the byte read is the last byte of the given word. In this implementation, the y variable is the count of words stored so far. If it is zero, we know we have just decoded what will become the first two words of output. We extract the two words (lines 31 and 32) and update the count. Otherwise, we check the output length and store if possible.

While the ASN.1 specification does not say so, we have limited the words to be unsigned long types. This means they cannot be larger than $2^{32} - 1$ in size. In practice, this makes the code simpler and does not conflict with any known cryptographic standard.

PRINTABLE and IA5 STRING Encodings

PRINTABLE and IA5 STRING encodings work much like the OCTET STRING encodings, except their inputs are limited in range and must be converted through a portable mechanism before being stored. This is because of the way various platforms handle code pages. For example, where

```
char c = 'a';
```

may be equivalent to

```
char c = 97;
```

on most ASCII platforms (e.g., most English Linux and Windows installs), it is not so on all other platforms. To facilitate this process, we need a table that can be interpreted by the C compiler in the native code page and then converted to an integer type for encoding and decoding.

For the sake of brevity, we shall demonstrate the PRINTABLE encoding from which the IA5 encoding can be derived (IA5 routines are available on the book's Web site).

```
der_printablestring_length.c:
#include “asn1.h"

static const struct {
   int code, value;
} printable_table[] = {
{ ' ', 32 }, { '\'', 39 }, { '(', 40 }, { ')', 41 },
{ '+', 43 }, { ',', 44 }, { '-', 45 }, { '.', 46 },
{ '/', 47 }, { '0', 48 }, { '1', 49 }, { '2', 50 },
{ '3', 51 }, { '4', 52 }, { '5', 53 }, { '6', 54 },
{ '7', 55 }, { '8', 56 }, { '9', 57 }, { ':', 58 },
{ '=', 61 }, { '?', 63 }, { 'A', 65 }, { 'B', 66 },
{ 'C', 67 }, { 'D', 68 }, { 'E', 69 }, { 'F', 70 },
{ 'G', 71 }, { 'H', 72 }, { 'I', 73 }, { 'J', 74 },
{ 'K', 75 }, { 'L', 76 }, { 'M', 77 }, { 'N', 78 },
{ 'O', 79 }, { 'P', 80 }, { 'Q', 81 }, { 'R', 82 },
{ 'S', 83 }, { 'T', 84 }, { 'U', 85 }, { 'V', 86 },
{ 'W', 87 }, { 'X', 88 }, { 'Y', 89 }, { 'Z', 90 },
{ 'a', 97 }, { 'b', 98 }, { 'c', 99 }, { 'd', 100 },
{ 'e', 101 }, { 'f', 102 }, { 'g', 103 }, { 'h', 104 },
{ 'i', 105 }, { 'j', 106 }, { 'k', 107 }, { 'l', 108 },
{ 'm', 109 }, { 'n', 110 }, { 'o', 111 }, { 'p', 112 },
{ 'q', 113 }, { 'r', 114 }, { 's', 115 }, { 't', 116 },
{ 'u', 117 }, { 'v', 118 }, { 'w', 119 }, { 'x', 120 },
{ 'y', 121 }, { 'z', 122 }, };
```

This is the mapping table that will convert characters from their native page to their (in this case) ASCII numerical representation. The table has been marked as both static and const to avoid having the table pollute the symbol namespace and sit in code space.

TIP

The use of the *const* and *static* keywords is particularly helpful in embedded development to manage symbol names and memory usage.

The *const* keyword will tell the compiler (in most cases) to keep the data in the code section of the application. The default is to keep a copy in the code section but to copy it from there to the memory (bss section with GNU tools) at runtime. This means the table would occupy both RAM and ROM.

The *static* keyword tells the compiler to not make the symbol global. This is useful if you have common functions across similar algorithms that may collide. It also helps keep pollution to a minimum, which is important when working with third-party libraries.

```
int der_printable_char_encode(int c)
{
   int x;
   for (x = 0; x <
        (int)(sizeof(printable_table) /
              sizeof(printable_table[0])); x++) {
       if (printable_table[x].code == c) {
          return printable_table[x].value;
       }
   }
   return -1;
}

int der_printable_value_decode(int v)
{
   int x;
   for (x = 0; x <
        (int)(sizeof(printable_table) /
              sizeof(printable_table[0])); x++) {
       if (printable_table[x].value == v) {
          return printable_table[x].code;
       }
   }
   return -1;
}
```

These two functions convert to numerical and from numerical (respectively). If the character or value is found, they return the encoding (or decoding), and in the case of a failure they return –1.

```
unsigned long der_printablestring_length(unsigned char *in,
                                         unsigned long inlen)
{
   unsigned long x;

   for (x = 0; x < inlen; x++) {
     if (der_printable_char_encode(in[x]) == -1) {
        return 0;
     }
   }

   return der_length(inlen);
}
```

This function is much like the der_octetstring_length() counterpart, except that it must first ensure all of the input values are valid.

```
der_printablestring_encode.c:
001   #include “asn1.h"
002   int der_printablestring_encode(unsigned char *in,
003                                  unsigned long  inlen,
004                                  unsigned char *out,
005                                  unsigned long *outlen)
006   {
007      unsigned long x;
008
009      /* check output size */
010      x = der_printablestring_length(in, inlen);
011      if (x == 0 || x > *outlen) {
012         return -1;
013      }
014
015      /* store header and length  */
016      der_put_header_length(&out, ASN1_DER_PRINTABLESTRING,
017                            inlen, outlen);
018
019      for (x = 0; x < inlen; x++) {
020         *out++ = der_printable_char_encode(*in++);
021      }
022
023      return 0;
024   }
```

We check the length and validity of the input (line 10) before encoding. It is important that the length function checks the validity, as the rest of the code assumes the input is valid. The encoding is performed by the simple translation function call (line 20).

```
der_printablestring_decode.c:
#include “asn1.h"
int der_printablestring_decode(unsigned char *in,
                               unsigned long  inlen,
                                 unsigned char *out,
                               unsigned long *outlen)
{
   unsigned type;
   unsigned long payload_length, x;
   int      ret;

   /* decode header */
012      ret = der_get_header_length(&in, inlen,
013                                  &type, &payload_length);
014      if (ret < 0) {
015         return ret;
016      }
017
018      if (type != ASN1_DER_PRINTABLESTRING) {
019         return -2;
020      }
021
022      if (payload_length > *outlen) {
023         return -1;
024      }
025      *outlen = payload_length;
026
027      for (x = 0; x < payload_length; x++) {
028          ret = der_printable_value_decode(*in++);
029          if (ret == -1) {
030              return -2;
031          }
032          *out++ = ret;
033      }
034      return 0;
035   }
```

Unlike the encoder, the decoder does not know if the input is valid before proceeding. As it decodes each byte (line 28), it checks the return value. If it is –1, the input byte is invalid and we must return an error code.

Now that we have seen PRINTABLE encoding, the IA5 encodings are essentially the same. The only difference is we use a different table.

```
der_ia5string_length.c:
static const struct {
   int code, value;
} ia5_table[] = {
{ '\0', 0 },{ '\a', 7 }, { '\b', 8 }, { '\t', 9 },
{ '\n', 10 }, { '\f', 12 }, { '\r', 13 }, { ' ', 32 },
{ '!', 33 }, { '"', 34 }, { '#', 35 }, { '$', 36 },
{ '%', 37 }, { '&', 38 }, { '\'', 39 }, { '(', 40 },
{ ')', 41 }, { '*', 42 }, { '+', 43 }, { ',', 44 },
{ '-', 45 }, { '.', 46 }, { '/', 47 }, { '0', 48 },
{ '1', 49 }, { '2', 50 }, { '3', 51 }, { '4', 52 },
{ '5', 53 }, { '6', 54 }, { '7', 55 }, { '8', 56 },
{ '9', 57 }, { ':', 58 }, { ';', 59 }, { '<', 60 },
{ '=', 61 }, { '>', 62 }, { '?', 63 }, { '@', 64 },
{ 'A', 65 }, { 'B', 66 }, { 'C', 67 }, { 'D', 68 },
{ 'E', 69 }, { 'F', 70 }, { 'G', 71 }, { 'H', 72 },
{ 'I', 73 }, { 'J', 74 }, { 'K', 75 }, { 'L', 76 },
{ 'M', 77 }, { 'N', 78 }, { 'O', 79 }, { 'P', 80 },
{ 'Q', 81 }, { 'R', 82 }, { 'S', 83 }, { 'T', 84 },
{ 'U', 85 }, { 'V', 86 }, { 'W', 87 }, { 'X', 88 },
{ 'Y', 89 }, { 'Z', 90 }, { '[', 91 }, { '\\', 92 },
{ ']', 93 }, { '^', 94 }, { '_', 95 }, { '`', 96 },
{ 'a', 97 }, { 'b', 98 }, { 'c', 99 }, { 'd', 100 },
{ 'e', 101 }, { 'f', 102 }, { 'g', 103 }, { 'h', 104 },
{ 'i', 105 }, { 'j', 106 }, { 'k', 107 }, { 'l', 108 },
{ 'm', 109 }, { 'n', 110 }, { 'o', 111 }, { 'p', 112 },
{ 'q', 113 }, { 'r', 114 }, { 's', 115 }, { 't', 116 },
{ 'u', 117 }, { 'v', 118 }, { 'w', 119 }, { 'x', 120 },
{ 'y', 121 }, { 'z', 122 }, { '{', 123 }, { '|', 124 },
{ '}', 125 }, { '~', 126 } };
```

For the interested reader it is possible to save space both in the table and in the code space. If a third parameter were added to the table to say which code type the symbol belonged to (e.g., IA5 or PRINTABLE), a single encode/decoder could be written with an additional input argument specifying which target code we are trying to use.

For the sake of simplicity, the routines demonstrated were implemented individually. They are fairly small and in the grand scheme of things are not significant code size contributors.

UTCTIME Encodings

UTCTIME encoding has been simplified in the 2002 specifications to only include one format. The time is encoded as a string in the format "YYMMDDHHMMSSZ" using two digits per component.

The year is encoded by examining the last two digits. Nothing beyond the year 2069 can be encoded with this format as it will be interpreted as 1970 (of course, by that time we can just reprogram the software to treat it as 2070). Despite the Y2K debacle, they still used two digits for the date.

The actual byte encoding of the string is using the ASCII code, and fortunately, we have to look no further than our PRINTABLE STRING routines for help.

To efficiently handle dates we require some structure to hold the parameters. We could just pass all six of them as arguments to the function. However, a more useful way (as we shall see when dealing with SEQUENCE types) is to have a structure. This structure is stored in our ASN.1 header file.

```
asn1.h:
/* UTCTIME */
typedef struct {
   int year,
       month,
       day,
       hour,
       min,
       sec;
} UTCTIME;
```

We have made this a type so that we can simply pass UTCTIME instead of "struct UTCTIME."

```
der_utctime_length.c:
#include “asn1.h"
unsigned long der_utctime_length(void)
{
   return 15;
}
```

My thanks go to the revised ASN.1 specifications! ☺

```
der_utctime_encode.c:
#include “asn1.h"

static int putnum(int val, unsigned char **out)
{
   unsigned char *ptr;
   int            h, l;

   if (val < 0) return -1;

   ptr = *out;
   h   = val / 10;
   l   = val % 10;

   if (h > 9 || l > 9) {
      return -1;
   }

   *ptr++ = der_printable_char_encode(“0123456789"[h]);
   *ptr++ = der_printable_char_encode(“0123456789"[l]);
   *out = ptr;

   return 0;
}
```

This function stores an integer in a two-digit ASCII representation. The number must be in the range 0–99 for this function to succeed. It updates the output pointer, which as we shall see in the next function is immediately useful.

We are re-using the der_printable_char_encode() to convert our integers to the ASCII code required.

```
int der_utctime_encode(      UTCTIME *in,
                       unsigned char *out,
                       unsigned long *outlen)
{
   /* check output size */
   if (der_utctime_length() > *outlen) {
      return -1;
   }

   /* store header and length  */
   der_put_header_length(&out, ASN1_DER_UTCTIME, 13, outlen);

   /* store data */
   if (putnum(in->year % 100, &out) ||
       putnum(in->month, &out) ||
       putnum(in->day, &out) ||
       putnum(in->hour, &out) ||
       putnum(in->min, &out) ||
       putnum(in->sec, &out)) {
       return -1;
   }

   *out++ = der_printable_char_encode('Z');

   return 0;
}
```

After the standard issue header encoding we then proceed to store the fields of the date and time in order as expected. We make extensive use of the way the C language handles the double-OR bars (||). Specifically, that is always implemented from left to right and will abort on the first nonzero return value without proceeding to the next case.

This means, for example, that if after encoding the month the routine fails, it will not encode the rest and will proceed inside to the braced statements directly.

der_utctime_decode.c:

```
#include “asn1.h"

static int readnum(unsigned char **in, int *dest)
{
   int x, y, z, num;
   unsigned char *ptr;

   num = 0;
   ptr = *in;
   for (x = 0; x < 2; x++) {
       num *= 10;
       z    = der_printable_value_decode(*ptr++);
       if (z < 0) {
          return -1;
       }
       for (y = 0; y < 10; y++) {
           if (“0123456789"[y] == z) {
               num += y;
               break;
           }
       }
       if (y == 10) {
          return -1;
       }
   }

   *dest = num;
   *in   = ptr;

   return 0;
}
```

This function decodes two bytes into the numerical value they represent. It borrows the PRINTABLE STRING routines again to decode the bytes to characters. It returns the number in the "dest" field and signals errors by the return code. We will use the same trick in the decoder to cascade a series of reads, which is why storing the result to a pointer provided by the caller is important.

```
033   int der_utctime_decode(unsigned char *in,
034                          unsigned long  inlen,
035                                UTCTIME *out)
036   {
037      unsigned type;
038      unsigned long payload_length;
039      int      ret;
040
041      /* decode header */
042      ret = der_get_header_length(&in, inlen,
043                                  &type, &payload_length);
044      if (ret < 0) {
045         return ret;
046      }
047      if (type != ASN1_DER_UTCTIME || payload_length != 13) {
048         return -2;
049      }
050
051      if (readnum(&in, &out->year)  ||
052          readnum(&in, &out->month) ||
053          readnum(&in, &out->day) ||
054          readnum(&in, &out->hour) ||
055          readnum(&in, &out->min) ||
056          readnum(&in, &out->sec)) {
057          return -1;
058      }
059
060      /* must be a Z here */
061      if (der_printable_value_decode(in[0]) != 'Z') {
062         return -2;
063      }
064
065      /* fix up year */
066      if (out->year < 70) {
067         out->year += 2000;
068      } else {
069         out->year += 1900;
070      }
071
072      return 0;
073   }
```

Here we use the cascade trick (lines 51 to 56) to parse the input and store the decoded fields one at a time to the output structure. After we have decoded the fields, we check for the required 'Z' (line 61) and then fix the year field to the appropriate century (lines 66 to 70).

SEQUENCE Encodings

The SEQUENCE type along with the SET and SET OF types are by far the most encompassing types to develop. They require all of the ASN.1 functions including them as well!

We will demonstrate relatively simple SEQUENCE encodings and omit SET types for brevity. The complete listing includes routines to handle SET types and the reader is encouraged to seek them out.

The first thing we need before we can encode a SEQUENCE is some way of representing it in the C programming language. In this case, we are going to use an array of a structure to represent the elements of the SEQUENCE.

```
asn1.h:
typedef struct {
   int               type;
   unsigned long     length;
   void             *data;
} asn1_list;
```

This is the structure for a SEQUENCE element type (it will also be used for SET types). The *type* indicates the ASN.1 type of the element, *length* the length or size of the value, and *data* is a pointer to the native representation of the given data type. An array of this type describes the elements of a SEQUENCE.

The length and data fields have different meanings depending on what the ASN.1 type is. Table 2.8 describes their use.

Table 2.8 Definitions for the ASN.1_List Type

ASN.1 Type	Meaning of "Data"	Meaning of "Length"
BOOLEAN	Pointer to an int	Ignored
INTEGER	Pointer to a long	Ignored
BIT STRING	Pointer to array of unsigned char	Number of bits
OCTET STRING	Pointer to array of unsigned char	Number of octets
NULL	Ignored	Ignored
OBJECT IDENTIFIER	Pointer to array of unsigned long	Number of words in the OID
IA5 STRING	Pointer to array of unsigned char	Number of characters
PRINTABLE STRING	Pointer to array of unsigned char	Number of characters
UTCTIME	Pointer to a UTCTIME structure	Ignored
SEQUENCE	Pointer to an array of asn1_list	Number of elements in the list

The use of the length field takes on a dual role depending on whether we are encoding for decoding. For all but the SEQUENCE type where the length is not ignored, the length specifies the maximum output size when decoding. It will be updated by the decoder to the actual length of the object decoded (in terms of what you would pass to the encoder).

For example, if you specify a length of 16 for a BIT STRING element and the decoder places a 7 in its place, that means that the decoder read a BIT STRING of seven bits.

We will also define a macro that is handy for creating lists at runtime, but first let us proceed through the SEQUENCE functions.

```
der_sequence_length.c:
#include “asn1.h"
unsigned long der_sequence_paylen(asn1_list     *list,
                                  unsigned long  length)
{
   unsigned long i, paylen, x;

   for (i = paylen = 0; i < length; i++) {
       switch (list[i].type) {
```

The crux of the SEQUENCE and SET routines is a huge switch statement for all the types. Here, we are using the ASN1_DER_* values, which fortunately do map to the ASN.1 types. It is best, though, that you use ASN1_DER_* symbols instead of literals.

```
          case ASN1_DER_BOOLEAN:
              paylen += der_boolean_length();
              break;
          case ASN1_DER_INTEGER:
              paylen +=
                der_integer_length(*((long *)list[i].data));
              break;
```

Here we see the use of the .data member of the structure to get access to what will eventually be encoded. We make use of the fact that in C, the void pointer can be cast to and from any other pointer type without violating the standard. What the data pointer actually points to changes based on what we are encoding. In this case, it is a long.

```
          case ASN1_DER_BITSTRING:
              paylen += der_bitstring_length(list[i].length);
              break;
```

Here we see the use of the .length member of the structure. The length means the number of units of a given type. In the case of BIT STRING, it means the number of bits to be encoded, whereas, for example, in the case of OID, length means the number of words in the OID encoding.

```
          case ASN1_DER_OCTETSTRING:
              paylen += der_octetstring_length(list[i].length);
              break;
          case ASN1_DER_NULL:
              paylen += der_null_length();
              break;
          case ASN1_DER_OID:
              x = der_oid_length(list[i].data, list[i].length);
              if (x == 0) return 0;
              paylen += x;
              break;
          case ASN1_DER_PRINTABLESTRING:
              x = der_printablestring_length(list[i].data,
                                             list[i].length);
              if (x == 0) return 0;
              paylen += x;
              break;
          case ASN1_DER_IA5STRING:
              x = der_ia5string_length(list[i].data,
                                       list[i].length);
              if (x == 0) return 0;
              paylen += x;
              break;
          case ASN1_DER_UTCTIME:
              paylen += der_utctime_length();
              break;
          case ASN1_DER_SEQUENCE:
              x = der_sequence_length(list[i].data,
                                      list[i].length);
              if (x == 0) return 0;
              paylen += x;
              break;
          default:
              return 0;
      }
   }
   return paylen;
}

unsigned long der_sequence_length(asn1_list     *list,
                                  unsigned long  length)
{
   return der_length(der_sequence_paylen(list, length));
}
```

```
der_sequence_encode.c:
      #include “asn1.h"
      int der_sequence_encode(asn1_list     *list,
                              unsigned long  length,
                              unsigned char *out,
                              unsigned long *outlen)
      {
         unsigned long i, x;
         int           err;

         /* check output size */
         if (der_sequence_length(list, length) > *outlen) {
            return -1;
         }

         /* store header and length  */
         der_put_header_length(&out, ASN1_DER_SEQUENCE,
                                der_sequence_paylen(list, length),
                                outlen);

         /* now encode each element */
         for (i = 0; i < length; i++) {
             switch (list[i].type) {
                case ASN1_DER_BOOLEAN:
                    x = *outlen;
                    err = der_boolean_encode(*((int *)list[i].data),
                                             out, &x);
                    if (err < 0) return err;
                    out += x;
                    break;
                case ASN1_DER_INTEGER:
                    x = *outlen;
                    err = der_integer_encode(*((long *)list[i].data),
                                             out, &x);
                    if (err < 0) return err;
                    out += x;
                    break;
                case ASN1_DER_BITSTRING:
                    x = *outlen;
                    err = der_bitstring_encode(list[i].data,
                                               list[i].length,
                                               out, &x);
                    if (err < 0) return err;
                    out += x;
                    break;
                case ASN1_DER_OCTETSTRING:
                    x = *outlen;
                    err = der_octetstring_encode(list[i].data,
                                                 list[i].length,
                                                 out, &x);
                    if (err < 0) return err;
                    out += x;
                    break;
                case ASN1_DER_NULL:
                    x = *outlen;
                    err = der_null_encode(out, &x);
                    if (err < 0) return err;
                    out += x;
                    break;
                case ASN1_DER_OID:
                    x = *outlen;
                    err = der_oid_encode(list[i].data,
                                         list[i].length, out, &x);
                    if (err < 0) return err;
                    out += x;
                    break;
                case ASN1_DER_PRINTABLESTRING:
                    x = *outlen;
                    err = der_printablestring_encode(list[i].data,
                                                     list[i].length,
                                                     out, &x);
```

```
              if (err < 0) return err;
              out += x;
              break;
          case ASN1_DER_IA5STRING:
              x = *outlen;
              err = der_ia5string_encode(list[i].data,
                                         list[i].length,
                                         out, &x);
              if (err < 0) return err;
              out += x;
              break;
          case ASN1_DER_UTCTIME:
              x = *outlen;
              err = der_utctime_encode(list[i].data, out, &x);
              if (err < 0) return err;
              out += x;
              break;
          case ASN1_DER_SEQUENCE:
              x = *outlen;
              err = der_sequence_encode(list[i].data,
                                        list[i].length,
                                        out, &x);
              if (err < 0) return err;
              out += x;
              break;
          default:
              return -1;
      }
   }
   return 0;
}
```

der_sequence_decode.c:

```
#include “asn1.h"
int der_sequence_decode(unsigned char *in,
                        unsigned long  inlen,
                        asn1_list     *list,
                        unsigned long  length)
{
   unsigned type;
   unsigned long payload_length, i, z;
   int      ret;

   /* decode header */
   ret = der_get_header_length(&in, inlen,
                               &type, &payload_length);
   if (ret < 0) {
      return ret;
   }

   if (type != ASN1_DER_SEQUENCE) {
      return -2;
   }

```

```
    for (i = 0; i < length; i++) {
        switch (list[i].type) {
           case ASN1_DER_BOOLEAN:
               ret = der_boolean_decode(in, payload_length,
                                        ((int *)list[i].data));
               if (ret < 0) return ret;
               z   = der_boolean_length();
               break;
           case ASN1_DER_INTEGER:
               ret = der_integer_decode(in, payload_length,
                                        ((long *)list[i].data));
               if (ret < 0) return ret;
               z   = der_integer_length(*((long *)list[i].data));
               break;
           case ASN1_DER_BITSTRING:
               ret = der_bitstring_decode(in, payload_length,
                                          list[i].data,
                                          &list[i].length);
               if (ret < 0) return ret;
               z   = list[i].length;
               break;
           case ASN1_DER_OCTETSTRING:
               ret = der_octetstring_decode(in, payload_length,
                                            list[i].data,
                                            &list[i].length);
               if (ret < 0) return ret;
               z   = der_octetstring_length(list[i].length);
               break;
           case ASN1_DER_NULL:
               ret = der_null_decode(in, payload_length);
               if (ret < 0) return ret;
               z   = der_null_length();
               break;
           case ASN1_DER_OID:
               ret =
                 der_oid_decode(in, payload_length,
                               ((unsigned long *)list[i].data),
                                &list[i].length);
               if (ret < 0) return ret;
               z   =
                 der_oid_length(((unsigned long *)list[i].data),
                                list[i].length);
               break;
           case ASN1_DER_PRINTABLESTRING:
               ret = der_printablestring_decode(in,
                                                payload_length,
                                                list[i].data,
                                                &list[i].length);
               if (ret < 0) return ret;
               z   = der_printablestring_length(list[i].data,
                                                list[i].length);
               break;
           case ASN1_DER_IA5STRING:
               ret = der_ia5string_decode(in, payload_length,
                                          list[i].data,
                                          &list[i].length);
               if (ret < 0) return ret;
               z   = der_ia5string_length(list[i].data,
                                          list[i].length);
               break;
           case ASN1_DER_UTCTIME:
               ret = der_utctime_decode(in, payload_length,
                                        list[i].data);
               if (ret < 0) return ret;
               z   = der_utctime_length();
               break;
           case ASN1_DER_SEQUENCE:
               ret = der_sequence_decode(in, payload_length,
                                         list[i].data,
                                         list[i].length);
               if (ret < 0) return ret;
               z   = der_sequence_length(list[i].data,
                                         list[i].length);
               break;
           default:
               return -1;
       }
       payload_length -= z;
       in             += z;
    }
    return 0;
 }
```

From these routines, we can easily spot a few shortcomings of this implementation. First, the routines here do not support modifiers such as OPTIONAL, DEFAULT, or CHOICE. Second, the decoding structure must match exactly or the decoding routine will abort. When we are encoding a structure, the calling application has to understand what it is encoding. The lists being encoded are meant to be generated at runtime instead of at compile time. This allows us a great level of flexibility as to how we interact with the modifiers. One of the key aspects of the implementation is that it allows new ASN.1 types to be supported by adding a minimal amount of code to these three routines.

Despite the ability to generate encoding SEQUENCEs at runtime, decoding is still problematic. The only way to use these functions to decode speculative SEQUENCEs is to continuously update the list using decoding failures as feedback. For example, if an element is listed as DEFAULT, then the de facto decoding list will have the element. If the decoding fails, the next logical step is to attempt decoding without the default item present.

For simple SEQUENCEs this can work, but as they grow in size (such as an X.509 certificate), this approach is completely inappropriate. One solution to this would be to have a more complete decoder that understands the modifiers and can perform the required element lookahead (much like a parser in any programming language who lookahead for tokens) to speculatively decode. However, this is actually much more work than the ideal solution requires.

ASN.1 Flexi Decoder

Our approach to this problem is to employ a Flexible Decoder (A.K.A. The Flexi Decoder), which decodes the ASN.1 data on the fly and generates its own linked list of ASN.1 objects. This allows us to decode essentially any ASN.1 data as long as we recognize the ASN.1 types encoded. The linked list grows in two directions—left to right and parent to child. The left to right links are for elements that are at the same depth level, and the parent to child links are for depth changes (e.g., SEQUENCE).

```
asn1.h
171   /* Flexi Decoding */
172   typedef struct Flexi {
173      int            type;
174      unsigned long  length;
175      void          *data;
176      struct Flexi  *prev, *next,
177                    *child, *parent;
178   } asn1_flexi;
```

This is our Flexi list, which resembles the *asn1_list* except that it has the doubly linked list pointers inside it. From a caller's point of view, they simply pass in an array of bytes and get as output a linked list structure for all the elements decoded.

While this solution allows us to easily decode arbitrary ASN.1 data, it still does not immediately yield a method of parsing it. First, we present the decoding algorithm, and in the next section present how to parse with it.

```
der_flexi_decode.c:
001   #include “asn1.h"
002   #include <stdlib.h>
003   int der_flexi_decode(unsigned char  *in,
004                        unsigned long   inlen,
005                        asn1_flexi    **out)
006   {
007      asn1_flexi    *list, *tlist;
008      unsigned long  len, x, y;
009      int            err;
010
011      list = NULL;
012
013      while (inlen >= 2) {
```

We start with an empty list (line 11) and handle its construction later. We then proceed to parse data as long as there are at least two bytes left. This allows us to soft error out when we hit the end of the perceived ASN.1 data.

```
014         /* make sure list points to a valid node */
015         if (list == NULL) {
016            list = calloc(1, sizeof(asn1_flexi));
017            if (list == NULL) return -3;
018         } else {
019            list->next = calloc(1, sizeof(asn1_flexi));
020            if (list->next == NULL) {
021               der_flexi_free(list);
022               return -3;
023            }
024            list->next->prev = list;
025            list = list->next;
026         }
```

Inside the loop, we always allocate at least one element of the linked list. If this is not the first element, we make an adjacent node (line 19) and doubly link it (line 24) so we can walk the list in any direction.

```
028         /* decode the payload length */
029         if (in[1] < 128) {
030            /* short form */
031            len = in[1];
032         } else {
033            /* get length of length */
034            x   = in[1] & 0x7F;
035
036            /* can we actually read this many bytes? */
037            if (inlen < 2 + x) { return -2; }
038
039            /* load it */
040            len = 0;
041            y   = 2;
042            while (x--) {
043               len = (len << 8) | in[y++];
044            }
045         }
```

We require the payload length for most types. For example, before we can decode an OCTET STRING we need to know the length so we can allocate the required memory. The encoded payload length immediately reveals this. For other types such as BIT STRING and OID, their maximum size can be calculated from the payload length, which is good enough for our purposes.

```
047         switch (in[0]) {
048            case ASN1_DER_BOOLEAN: /* BOOLEAN */
049               list->type   = ASN1_DER_BOOLEAN;
050               list->length = 1;
051               list->data   = calloc(1, sizeof(int));
052               if (list->data == NULL) { goto MEM_ERR; }
053
054               err = der_boolean_decode(in, inlen, list->data);
055               if (err < 0) { goto DEC_ERR; }
056
057               len = der_boolean_length();
058               if (len == 0) { goto LEN_ERR; }
059               break;
060            case ASN1_DER_INTEGER: /* INTEGER */
061               list->type   = ASN1_DER_INTEGER;
062               list->length = 1;
```

```
              list->data   = calloc(1, sizeof(long));
              if (list->data == NULL) { goto MEM_ERR; }

              err = der_integer_decode(in, inlen, list->data);
              if (err < 0) { goto DEC_ERR; }

              len = der_integer_length(*((long *)list->data));
              if (len == 0) { goto LEN_ERR; }
              break;
           case ASN1_DER_BITSTRING: /* BIT STRING */
              list->type   = ASN1_DER_BITSTRING;
              list->length = (len-1)<<3;
              list->data   =
                  calloc(list->length, sizeof(unsigned char));
              if (list->data == NULL) { goto MEM_ERR; }

              err = der_bitstring_decode(in, inlen,
                                         list->data,
                                         &list->length);
              if (err < 0) { goto DEC_ERR; }

              len = der_bitstring_length(list->length);
              if (len == 0) { goto LEN_ERR; }
              break;
```

The length for BIT STRING is estimated as 8*(len–1), which is the maximum size required. We subtract one because the payload includes the padding count byte. At most, we waste seven bytes of memory by allocating only in multiples of eight.

```
           case ASN1_DER_OCTETSTRING: /* OCTET STRING */
              list->type   = ASN1_DER_OCTETSTRING;
              list->length = len;
              list->data   = calloc(list->length,
                                    sizeof(unsigned char));
              if (list->data == NULL) { goto MEM_ERR; }

              err = der_octetstring_decode(in, inlen,
                                           list->data,
                                           &list->length);
              if (err < 0) { goto DEC_ERR; }

              len = der_octetstring_length(list->length);
              if (len == 0) { goto LEN_ERR; }
              break;
           case ASN1_DER_NULL: /* NULL */
              list->type   = ASN1_DER_NULL;
              list->length = 0;
              if (in[1] != 0x00) { goto DEC_ERR; }
              len = 2;
              break;
           case ASN1_DER_OID: /* OID */
              list->type   = ASN1_DER_OID;
              list->length = len;
              list->data   = calloc(list->length,
                                    sizeof(unsigned long));
              if (list->data == NULL) { goto MEM_ERR; }

              err = der_oid_decode(in, inlen,
                                   list->data,
                                   &list->length);
              if (err < 0) { goto DEC_ERR; }

              len = der_oid_length(list->data, list->length);
              if (len == 0) { goto LEN_ERR; }
              break;
```

Similar as for the BIT STRING, we have to estimate the length here. In this case, we use the fact that there cannot be more OID words than there are bytes of payload (the smallest word encoding is one byte). The wasted memory here can be several words, which for most circumstances will require to a trivial amount of storage.

The interested reader may want to throw a realloc() call in there to free up the unused words.

```
      case ASN1_DER_PRINTABLESTRING: /* PRINTABLE STRING */
         list->type   = ASN1_DER_PRINTABLESTRING;
         list->length = len;
         list->data   = calloc(list->length,
                               sizeof(unsigned char));
         if (list->data == NULL) { goto MEM_ERR; }

         err = der_printablestring_decode(in, inlen,
                                          list->data,
                                          &list->length);
         if (err < 0) { goto DEC_ERR; }

         len = der_printablestring_length(list->data,
                                          list->length);
         if (len == 0) { goto LEN_ERR; }
         break;
      case ASN1_DER_IA5STRING: /* IA5 STRING */
         list->type   = ASN1_DER_IA5STRING;
         list->length = len;
         list->data   = calloc(list->length,
                               sizeof(unsigned char));
         if (list->data == NULL) { goto MEM_ERR; }

         err = der_ia5string_decode(in, inlen,
                                    list->data,
                                    &list->length);
         if (err < 0) { goto DEC_ERR; }

         len = der_ia5string_length(list->data,
                                    list->length);
         if (len == 0) { goto LEN_ERR; }
         break;
      case ASN1_DER_UTCTIME: /* UTC TIME */
         list->type   = ASN1_DER_UTCTIME;
```

```
157               list->length = 1;
158               list->data   = calloc(1, sizeof(UTCTIME));
159               if (list->data == NULL) { goto MEM_ERR; }
160
161               err = der_utctime_decode(in, inlen, list->data);
162               if (err < 0) { goto DEC_ERR; }
163
164               len = der_utctime_length();
165               if (len == 0) { goto LEN_ERR; }
166               break;
167            case ASN1_DER_SEQUENCE: /* SEQUENCE */
168               list->type   = ASN1_DER_SEQUENCE;
169               list->length = len;
170
171               /* we know the length of the objects in
172                  the sequence, it's len bytes */
173               err = der_flexi_decode(in+2, len, &list->child);
174               if (err < 0) { goto DEC_ERR; }
175               list->child->parent = list;
176
177               /* len is the payload length, we have
178                  to add the header+length */
179               len += 2;
180               break;
181            default:
182               /* invalid type, soft error */
183               inlen = 0;
184               len   = 0;
185               break;
186         }
187         inlen -= len;
188         in    += len;
```

When we encounter a type we do not recognize (line 181), we do not give a fatal error; we simply reset the lengths, terminate the decoding, and exit. We have to set the payload length to zero (line 184) to avoid having the subtraction of payload (line 187) going below zero.

We use the der_*_length() function to compute the length of the decoded type and store the value back into *len*. By time we get to the end (line 187), we know how much to move the input pointer up by and decrease the remaining input length by.

```
189      }
190
191      /* we may have allocated one more than we need */
192      if (list->type == 0) {
193          tlist      = list->prev;
194          free(list);
195          list       = tlist;
196          list->next = NULL;
197      }
```

At this point, we may have added one too many nodes to the list. This is determined by having a type of zero, which is not a valid ASN.1 type. If this is the case, we get the previous link, free the leaf node, and update the pointer (lines 193 through 196).

```
198
199      /* rewind */
200      while (list->prev) {
201         list = list->prev;
202      }
203
204      *out = list;
205      return 0;
206   MEM_ERR:
207      der_flexi_free(list);
208      return -3;
209   DEC_ERR:
210      der_flexi_free(list);
211      return -2;
212   LEN_ERR:
213      der_flexi_free(list);
214      return -1;
215   }
```

Putting It All Together

At this point, we have all the code we require to implement public key (PK) standards such as PKCS or ANSI X9.62. The first thing we must master is working with SEQUENCEs. No common PK standard will use the ASN.1 types (other than SEQUENCE) directly.

Building Lists

Essentially container ASN.1 type is an array of the asn1_list type. So let us examine how we convert a simple container to code these routines can use.

```
RSAKey ::= SEQUENCE {
   N        INTEGER,
   E        INTEGER
}
```

First, we will begin with defining a list.

```
asn1_list RSAKey[2];
```

Next, we need two integers to hold the values. Keep in mind that these are the C long type and not a BigNum as we actually require for secure RSA. This is for demonstration purposes only.

```
long N, E;
```

Now we must actually assign the elements of the RSAKey.

```
RSAKey[0].type   = ASN1_DER_INTEGER;
```

```
RSAKey[0].length = 1;
RSAKey[0].data   = &N;
```

This triplet of code is fairly awkward and makes ASN.1 coding a serious pain. A simpler solution to this problem is to define a macro to assign the values in a single line of C.

asn1.h:

```
#define asn1_set(list, index, Type, Length, Data) \
do {                                               \
   asn1_list     *ABC_list;                        \
   int            ABC_index;                       \
                                                   \
   ABC_list   = list;                              \
   ABC_index  = index;                             \
                                                   \
   ABC_list[ABC_index].type   = Type;              \
   ABC_list[ABC_index].length = Length;            \
   ABC_list[ABC_index].data   = Data;              \
} while (0);
```

The macro for those unfamiliar with the C preprocessor looks fairly indirect. First, we make a copy of the list and index parameters. This allows them to be temporal. Consider the invocation as follows.

```
asn1_set(mylist, i++, type, length, data);
```

If we did not first make a copy of the index, it would be different in every instance it was used during the macro. Similarly, we could have

```
asn1_set(mylist++, 0, type, length, data);
```

The use of the do-while loop, allows us to use the macro as a single C statement. For example:

```
if (x > 0)
    asn1_set(mylist, x, type, length, data);
```

Now we can reconsider the RSAKey definition.

```
asn1_set(RSAKey, 0, ASN1_DER_INTEGER, 1, &N);
asn1_set(RSAKey, 1, ASN1_DER_INTEGER, 1, &E);
```

This is much more sensible and pleasant to the eyes.

Now suppose we have the RSA key with a modulus N=17*13=221 and E=7 and want to encode this. First, we need a place to store the key.

```
unsigned char output[100];
unsigned char output_length;
```

Now we can encode the key. Let us put the entire example together.

```
asn1_list     RSAKey[2];
unsigned char output[100];
unsigned char output_length;
```

```
int          err;
long         N, E;

/* Set the key  and list */
N = 221;
E = 7;
asn1_set(RSAKey, 0, ASN1_DER_INTEGER, 1, &N);
asn1_set(RSAKey, 1, ASN1_DER_INTEGER, 1, &E);

/* Encode it */
output_length = sizeof(output);
err = der_sequence_encode(&RSAKey, 2, output, &output_length);
if (err < 0) {
   printf("SEQUENCE encoding failed: %d\n", err);
   exit(EXIT_FAILURE);
}

printf("We encoded a SEQUENCE into %lu bytes\n", output_length);
```

At this point the array *output[0..output_length – 1]* contains the DER encoding of the RSAKey SEQUENCE.

Nested Lists

Handling SEQUENCEs within a SEQUENCE is essentially an extension of what we already know. Let us consider the following ASN.1 construction.

```
User := SEQUENCE {
   Name       PRINTABLE STRING,
   Age        INTEGER,
   Credentials SEQUENCE {
      passwdHash  OCTET STRING
   }
}
```

As before, we build two lists. Here is the complete example.

```
asn1_list     User[3], Credentials;
unsigned char output[100];
unsigned char output_length;
int           err;

long          Age;
unsigned char Name[MAXLEN+1], passwdHash[HASHLEN];

/* build the first list */
asn1_set(User, 0, ASN1_DER_PRINTABLESTRING, strlen(Name), Name);
asn1_set(User, 1, ASN1_DER_INTEGER,                    1, &Age);
asn1_set(User, 2, ASN1_DER_SEQUENCE,                   1, &Credentials);

/* build second list */
asn1_set(Credentials, 0, ASN1_DER_OCTETSTRING, HASHLEN, passwdHash);

/* encode it */
```

```
output_length = sizeof(output);
err = der_sequence_encode(User, 3, output, &output_length);
if (err < 0) { printf("Error encoding %d\n", err); exit(EXIT_FAILURE); }
```

When building the first list we pass a pointer to the second list (the third entry in the User array). The corresponding "1" marked in the length field for the third entry is actually the length of the second list. That is, if the Credentials list had two items we would see a two in place of the one.

Note that we encode the entire construction by invoking the encoder once with the User list. The encoder will follow the pointer to the second list and encode it in turn as well.

Decoding Lists

Decoding a container is much the same as encoding except that where applicable the length parameter is the size of the destination. Let us consider the following SEQUENCE.

```
User ::= SEQUENCE {
   Name       PRINTABLE STRING,
   Age        INTEGER,
   Flags      BIT STRING
}
```

For this SEQUENCE, we will require two unsigned char arrays and a long. Lets get those out of the way.

```
long          Age;
unsigned char Name[MAXNAMELEN+1], Flags[MAXFLAGSLEN];
asn1_list     User[3];
```

Now we have to setup the list.

```
asn1_set(User, 0, ASN1_DER_PRINTABLESTRING, sizeof(Name)-1,  Name);
asn1_set(User, 1, ASN1_DER_INTEGER,         1,              &Age);
asn1_set(User, 2, ASN1_DER_BITSTRING,       sizeof(Flags),   Flags);
```

Note that we are using sizeof(Name)-1 and not just the size of the object. This allows us to have a trailing NUL byte so that the C string functions can work on the data upon decoding it.

Let us put this entire example together:

```
long          Age;
unsigned char Name[MAXNAMELEN+1], Flags[MAXFLAGSLEN];
asn1_list     User[3];
int           err;

asn1_set(User, 0, ASN1_DER_PRINTABLESTRING, sizeof(Name)-1,  Name);
asn1_set(User, 1, ASN1_DER_INTEGER,         1,              &Age);
asn1_set(User, 2, ASN1_DER_BITSTRING,       sizeof(Flags),   Flags);

memset(Name, 0, sizeof(Name));
err = der_sequence_decode(input, input_len, &User, 3);
if (err < 0) {
   printf("Error decoding the sequence: %d\n", err);
   exit(EXIT_FAILURE);
}

printf("Decoded the sequence\n");
printf("User Name[%lu] == [%s]\n", User[0].length, Name);
printf("Age == %ld\n", Age);
```

This example assumes that some array of unsigned char called input has been provided and its length is input_len. Upon successful decoding, the program output will display the user name and the age. For instance, the output may resemble the following.

```
Decoded the sequence
User Name[3] == [Tom]
Age == 24
```

As see the *User* array is updated for specific elements such as the STRING types. *User[0].length* will hold the length of the decoded value. Note that in our example we first memset the array to zero. This allows us to use the decoded array as a valid C string regardless of the length.

FlexiLists

As we discussed in the previous section the SEQUENCE decoder is not very amenable to speculative encodings. The flexi decoder allows us to decode ASN.1 data without knowing the order or even the types of the elements that were encoded.

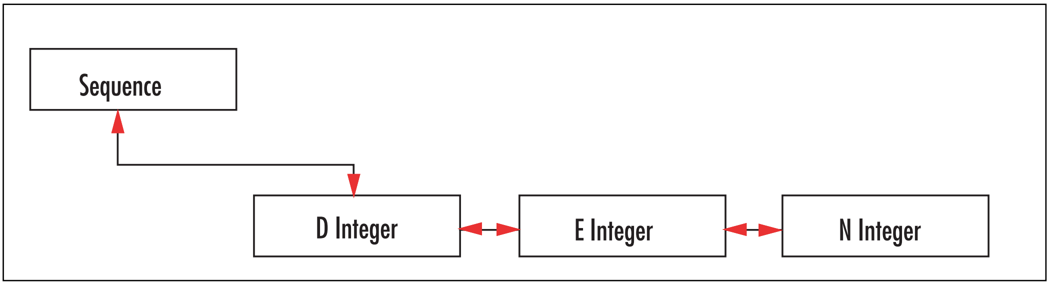
While the flexi decoder allows for speculative decoding, it does not allow for speculative parsing. What we are given by the decoder is simply a multi-way doubly linked list. Each node of the list contains the ASN.1 type, its respective length parameter and a pointer (if appropriate) to the decoding of the data (in a respective format).

The encodings do not tell us which element of the constructed type we are looking at. For example, consider the following SEQUENCE.

```
RSAKey ::= SEQUENCE {
   D       INTEGER OPTIONAL,
   E       INTEGER
   N       INTEGER
}
```

In this structure, we can easily differentiate a public and private RSA key by the presence of the D INTEGER in the encoding. When we use the flexi decoder to process this we will get two depths to the list. The first depth will contain just the SEQUENCE and the child of that node will be a list of up to three items (see Figure 2.4).

Figure 2.4 Organization of the Flexi Decoding of RSAKey



Suppose we store the outcome in MyKey such as the following:

```
asn1_flexi *MyKey;
der_flexi_decode(keyPacket, keyPacketLen, &MyKey);
```

In this instance, MyKey would point to the SEQUENCE node of the list. Its *next* and *prev* pointers will be NULL and it will have a pointer to a *child* node. The child node will be the "D" INTEGER, if it is present, and it will have a pointer to the "E" INTEGER through the *next* pointer.

Given the MyKey pointer, we can move to the child node with the following code:

```
MyKey = MyKey->child;
```

We do not have to worry about losing the parent pointer as we can walk back to the parent with the following code.

```
MyKey = MyKey->parent;
```

Note that only the first entry in the child list will have a parent pointer. If we walked to the "E" INTEGER then we could not move directly to the parent:

```
MyKey = MyKey->child;
MyKey = MyKey->next;    /* now we point to "E" */
MyKey = MyKey->parent;  /* this is invalid */
```

Now how do we know if we have a private or public key? The simplest method in this particular case is to count the number of children:

```
int x = 0;
while (MyKey->next) {
     ++x;
     MyKey = MyKey->next;
}
```

If the *x* variable is two, it is a public key; otherwise, it is a private key. This approach works well for simple OPTIONAL types but only if there is one OPTIONAL element. Similarly, for CHOICE modifiers we cannot simply look at the length but need also the types and their contents. The simple answer is to walk the list and stop once you see a difference or characteristic that allows you to decide what you are looking at.

Fortunately for us, the only real PK based need for the flexi decoder is for X.509 certificates, which have many OPTIONAL components. As we will see later, the PKCS and ANSI public key standards have uniquely decodable SEQUENCEs that do not require the flexi decoder.

Other Providers

The code in this chapter is loosely based on that of the LibTomCrypt project. In fact, the idea for the flexi decoder is taken directly from it where it is used by industry developers to work with X.509 certificates. The reader is encouraged to use it where appropriate as it is a more complete (and tested) ASN.1 implementation which also sports proper support for INTEGER, SET and SET OF types.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: What is ASN.1 and why do I care?

A: ASN.1 defines portable methods of storing and reading various common data types such that various programs can interoperate. It is a benefit for customers and developers as it allows third party tools to interpret data created by a host program. In many cases, it is a highly valuable selling point for customers as it assures them their data is not part of a proprietary encoding scheme. It avoids vendor lock-in problems.

Q: Why isn't a format like XML used?

A: The standards were written nearly a decade before XML was even a twinkle in the Tim Bray's eye. Despite the fact that ASN.1 predates XML, it is still not the only reason that ASN.1 is preferred over XML. XML is huge in comparison to its ASN.1 equivalent. This makes it less well adapted to use in hardware where memory constraints are tight. Since a lot of cryptography is done on devices with very little memory (smart-cards are a good example of this), it makes sense to use a compact format. It's funny that many people are clamoring for a binary version of XML with the view to speeding up parsing and reducing size when a perfectly good standard that meets these requirements has been in place since the 80s.

Q: What standards define ASN.1?

A: The ITU-T X.680 and X.690 series of standards.

Q: Who uses ASN.1?

A: ASN.1 is part of the PKCS (#1 and #7 for example), ANSI X9.62, X9.63 and X.509 series of standards.

Q: What pre-built cryptographic libraries support ASN.1?

A: The OpenSSL and LibTomCrypt projects both support ASN.1 DER encoding and decoding. The code presented in this book has been loosely based on that in the LibTomCrypt library. The reader is encouraged to use it where possible as it is well integrated into the rest of the LibTomCrypt library and also properly supports the INTEGER type (as well as SET and SET OF types).

Chapter 3

Random Number Generation

Solutions in this chapter:

- Concept of Random
- Measuring Entropy
- RNG Design
- PRNG Algorithms
- Putting It All Together

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

In this chapter, we begin to get into actual cryptography by studying one of the most crucial but often hard to describe components of any modern cryptosystem: the random bit generator.

Many algorithms depend on being able to collect bits or, more collectively, numbers that are difficult for an observer to guess or predict for the sake of security. For example, as we shall see, the RSA algorithm requires two random primes to make the public key hard to break. Similarly, protocols based on (say) symmetric ciphers require random symmetric keys an attacker cannot learn. In essence, no cryptography can take place without random bits. This is because our algorithms are public and only specific variables are private. Think of it as solving linear equations: if I want to stop you from solving a system of four equations, I have to make sure you only know at most three independent variables. While modern cryptography is more complex than a simple linear system, the idea is the same.

Throughout this chapter, we refer to both random bit generators and random number generators. For all intents and purposes, they are the same. That is, any bit generator can have its bits concatenated to form a number, and any number can be split into at least one bit if not more.

When we say random bit generator, we are usually talking about a deterministic algorithm such as a Pseudo Random Number Generator (PRNG) also known as a Deterministic Random Bit Generator (DRBG in NIST terms). These algorithms are deterministic because they run as software or hardware (FSM) algorithms that follow a set of rules accurately. On the face of it, this sounds rather contradictory to the concept of a random bit generator to be deterministic. However, as it turns out, PRNGs are highly practical and provide real security guarantees in both theory and practice. On the upside, PRNGs are typically very fast, but this comes at a price. They require some outside source to *seed* them with entropy. That is, some part of the PRNG deterministic *state* must be unpredictable to an outside observer.

This hole is where the random bit generators come into action. Their sole purpose is to kick-start a PRNG so it can *stretch* the fixed size seed into a longer size string of random bits. Often, we refer to this stretching as *bit extraction*.

Concept of Random

We have debated the concept and existence of true randomness for many years. Something, such as an event, that is random cannot be predicted with more probability than a given model will allow. There has been a debate running throughout the ages on randomness. Some have claimed that some events cannot be fully modeled without disturbing the state (such as observing the spin of photons) and therefore can be random, while others claim at some level all influential variables controlling an event can be modeled. The debate has pretty much been resolved. Quantum mechanics has shown us that randomness does in fact exist in the real world, and is a critical part of the rules that govern our universe. (For interest, the Bell Inequality gives us good reason to believe there are no "deeper" systems that would explain the perceived randomness in the universe. Wikipedia is your friend.)

It is not an embellishment to say that the very reason the floor is solid is direct proof that uncertainty exists in the universe. Does it ever strike you as odd that electrons are never found sitting on the surface of the nucleus of the atom, despite the fact opposite charges should attract? The reason for this is that there is a fundamental uncertainty in the universe. Quantum theory says that if I know the position on an electron to a certain degree, I must be uncertain of its momentum to some degree.

When you take this principle to its logical conclusion, you are able to justify the existence of a stable atom in the presence of attractive charges. The fact you are here to read this chapter, as we are here to write it, is made possible by the fundamental uncertainty present in the universe.

Wow, this is heavy stuff and we are only on the second page of this chapter. Do not despair; the fact that there is real uncertainty in the world does not preclude us trying to understand it in a scientific and well-contained way.

Claude Shannon gave us the tools to do this in his groundbreaking paper published in 1949. In it, he said that being uncertain of a result implied there was information to be had in knowing it. This offends common sense, but it is easy to see why it is true.

Consider there's a program that has printed a million "a"s to the screen. How sure are you that the next letter from the program will be "a?" We would say that based on what you've seen, the chances of the next letter being "a" are very high indeed. Of course, you can't be truly certain that the next letter is "a" until you have observed it. So, what would it tell us if there was another "a?" Not a lot really; we already knew there was going to be an "a" with very high probability. The amount of information that "a" contains, therefore, is not very much.

These arguments lead into a concept called entropy or the uncertainty of an event—rather complicated but trivial to explain. Consider a coin toss. The coin has two sides, and given that the coin is symmetric through its center (when lying flat), you would expect it to land on either face with a probability of 50%. The amount of entropy in this event is said to be one bit, which comes from the equation $E = -\log_2(p)$, or in this case $-\log_2(0.5) = 1$. This is a bit simplistic; the amount of entropy in this event can be determined using a formula derived by Bell Labs' Claude Shannon in 1949. The event (the coin toss) can have a number of outcomes (in this case, either heads or tails). The entropy associated with each outcome is $-\log_2(p)$[1], where p is the probability of that particular outcome. The overall entropy is just the weighted average of the entropy of all the possible outcomes. Since there are two possible outcomes, and their entropy is the same, their average is simply $E = -\log_2(0.5) = 1$ bit. Entropy in this case is another way of expressing uncertainty. That is, if you wrote down an infinite stream of coin toss outcomes, the theory would tell us that you could not represent the list of outcomes with anything less than one bit per toss on average. The reader is encouraged to read up on Arithmetic Encoding to see how we can compress binary strings that are biased.

This, however, assumes that there are influences on the coin toss that we cannot predict or model. In this case, it is actually not true. Given an average rotational speed and height, it is possible to predict the coin toss with a probability higher than 50%. This does not mean that the coin toss has zero entropy; in fact, even with a semi-accurate model it is likely still very close to one bit per toss. The key concept here is how to model entropy.

So, what is "random?" People have probably written many philosophy papers on that question. Being scientists, we find the best way to approach this is from the computer science angle: a string of digits is random if there is no program shorter than it that can describe its form.

This is called Kolmogorov complexity, and to cover this in depth here is beyond the scope of this text. The interested reader should consult The *Quest for Omega* by G. J. Chaitin. The e-book is available at www.arxiv.org/abs/math.HO/0404335.

What is in scope is how to create nondeterministic (from the software point of view) random bit generators, which we shall see in the upcoming section. First, we have to determine how we can observe the entropy in a stream of events.

Measuring Entropy

The first thing we need to be able to do before we can construct a random bit generator is have a method or methods that can *estimate* how much entropy we are actually generating with every bit generated. This allows us to use carefully the RNG to seed a PRNG with the assumption that it has some minimum amount of entropy in the state.

Note clearly that we said *estimate*, not *determine*. There are many known useful RNG tests; however, there is no short list of ways of modeling a given subset of random bits. Strictly speaking, for any bit sequence of length L bits, you would have to test all generators of length L–1 bits or less to see if they can generate the sequence. If they can, then the L bits actually have L–1 (or less) bits of entropy.

In fact, many RNG tests do not even count entropy, at least not directly. Instead, many tests are simply simulations known as Monte Carlo simulations. They run a well-studied simulation using the RNG output at critical decision points to change the outcome. In addition to simulations, there are predictors that observe RNG output bits and use that to try to simulate the events that are generating the bits.

It is often much easier to predict the output of a PRNGs than a block cipher or hash doing the same job. The fact that PRNGS typically use fewer operations per byte than a block cipher means there are less operations per byte to actually secure the information. It is simplistic to say that this, in itself, is the cause of weakness, but this is how it can be understood intuitively. Since PRNGS are often used to seed other cryptographic primitives, such as block ciphers or message authentication codes, it is worth pointing out that someone who breaks the PRNG may well be able to take down the rest of the system.

Various programs such as DIEHARD and ENT have surfaced on the Internet over the years (DIEHARD being particularly popular—www.arxiv.org/abs/math.HO/0404335) that run a variety of simulations and predictors. While they are usually good at pointing out flawed designs, they are not good at pointing out good ones. That is, an algorithm that passes one, or even all, of these tests can still be flawed and insecure for cryptographic purposes. That said, we shall not dismiss them outright and explore some basic tests that we can apply.

A few key tests are useful when designing a RNG. These are by no means an exhaustive list of tests, but are sufficient to quickly filter out RNGs.

Bit Count

The "bit count" test simply counts the number of zeroes and ones. Ideally, you would expect an even distribution over a large stream of bits. This test immediately rules out any RNG that is biased toward one bit or another.

Word Count

Like the bit count, this test counts the number of k-bit words. Usually, this should be applied for values of k from two through 16. Over a sufficiently large stream you should expect to see any k-bit word occurring with a probability of $1/2^k$. This test rules out oscillations in the RNG; for example, the stream 01010101… will pass the bit count but fail the word count for k=2. The stream 001110001110… will pass both the bit count and for k=2, but fail at k=3 and so on.

Gap Space Count

This test looks for the size of the gaps between the zero bits (or between the one bits depending how you want to look at it). For example, the string 00 has a gap of zero, 010 has a gap of one, and so on. Over a sufficiently large stream you would expect a gap of k bits to occur with a probability of $1/2^{k+1}$. This test is designed to catch RNGs that latch (even for a short period) to a given value after a clock period has finished.

Autocorrelation Test

This test tries to determine if a subset of bits is related to another subset from the same string. Formally defined for continuous streams, it can be adopted for finite discrete signals as well. The following equation defines the autocorrelation.

$$R(j) = \Sigma_n x_n x_{n-j}$$

Where *x* is the signal being observed and R(j) is the autocorrelation coefficient for the lag *j*. Before we get too far ahead of ourselves, let us examine some terminology we will use in this section. Two items are correlated if they are more similar (related) than unlike. For example, the strings 1111 and 1110 are similar, but by that token, so are 1111 and 0000, as the second is just the exact opposite of the first. Two items are uncorrelated if they have equal amounts of distinction and similarity. For example, the strings 1100 and 1010 are perfectly uncorrelated.

In the discrete world where samples are bits and take on the values {0, 1}, we must map them first to the values {–1, 1} before applying the transform. If we do not, the autocorrelation function will tend toward zero for correlated samples (which are the exact opposites).

There is another more hardware friendly solution, which is to sum the XOR difference of the two. The new autocorrelation function becomes

$$R(j) = \Sigma_n x_n \text{ XOR } x_{n-j}$$

This will tend toward n/2 for uncorrelated streams and toward 0 or n for correlated streams. Applying this to a finite stream now becomes tricky. What are the samples values below the zero index? The typical solution is to start summation from j onward and look to have a summation close to (n–j)/2 instead.

autocorrelate.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
void printauto(int *bits, int size, int maxj)
{
   int x, j, sum;
   for (j = 1; j <= maxj; j++) {
      for (x = j, sum = 0; x < size; x++) {
         sum += (bits[x] ^ bits[x-j]);
      }
      printf("Lag[%4d] = %5d (expected %5d)\n",
              j, sum, (size - j)/2);
   }
}
```

This function prints the autocorrelation coefficients for an array of samples up to a maximum lag. Let's consider a simple biased PRNG.

```
l = 0;
for (x = 0; x < SIZE; x++) {
   if (rand()&1) {
     bits[x] = 1;
   } else {
     l = bits[x] = rand()&1;
   }
}
```

This generator will output the last bit with a probability of 50%, and otherwise will output a random bit. Let's examine the data through the autocorrelation test.

```
Lag[   1] = 262638 (expected 524287)
Lag[   2] = 393784 (expected 524287)
Lag[   3] = 459840 (expected 524286)
Lag[   4] = 492175 (expected 524286)
Lag[   5] = 508232 (expected 524285)
Lag[   6] = 515917 (expected 524285)
```

```
Lag[   7] = 520401 (expected 524284)
Lag[   8] = 521771 (expected 524284)
```

This was over 1048576 samples (2^{20}). We can see that we are far from ideal in the first eight lags listed. Now consider the test on just `rand() & 1`.

```
Lag[   1] = 524999 (expected 524287)
Lag[   2] = 525284 (expected 524287)
Lag[   3] = 524638 (expected 524286)
Lag[   4] = 525564 (expected 524286)
Lag[   5] = 524480 (expected 524285)
Lag[   6] = 523917 (expected 524285)
Lag[   7] = 524692 (expected 524284)
Lag[   8] = 524081 (expected 524284)
```

As we can see, the autocorrelation values are closer to the expected mean than before. The conclusion from this is experiment would be that the former is a bad bit generator and the latter is more appropriate. Keep in mind this test does not mean the latter is ideal for use as a RNG. The rand() function from glibc is certainly not a secure PRNG and has a very small internal state. This is a common mistake; do not make it.

One downside to this trivial implementation is that it requires a huge buffer and cannot be run on the fly. In particular, it would be nice to run the test while the RNG is active to ensure it does not start producing correlated outputs. It turns out a windowed correlation test is fairly simple to implement as well.

wincor.c:

```
/* windowed autocorrelation */
#define MAXLAG 16
int window[MAXLAG], correlation[MAXLAG];

void wincor_add_bit(int bit)
{
   int x;
   /* compute lags */
   for (x = 0; x < MAXLAG; x++) {
      correlation[x] += window[x] ^ bit;
   }

   /* shift */
   for (x = 0; x < MAXLAG-1; x++) {
      window[x] = window[x + 1];
   }
   window[MAXLAG-1] = bit;
}
```

In this case, we defined 16 taps (MAXLAG), and this function will process one bit at a time through the autocorrelation function. It does not output a result, but instead updates the global *correlation* array. This is essentially a shift register with a parallel summation operation. It is actually well suited to both hardware implementation and SIMD software (especially with longer LAG values). This test functions best only on longer streams of bits. The default window will populate with zero bits, which may or may not correlate to the bits being added. In general, avoid applying the autocorrelation test to strings fewer than at least a couple thousand bits.

How Bad Can It Be?

In all the tests mentioned previously, you know the "correct" answer, but you won't often get it. For example, if you toss a perfect coin 20 times, you expect 10 heads. But what if you get 11, or only 9? Does that mean the coin was biased? In fact, of all the 2^{20} (1048576) possible (and equally likely) outcomes of your 20 coin tosses, only 184756, about 18%, actually have exactly 10 heads.

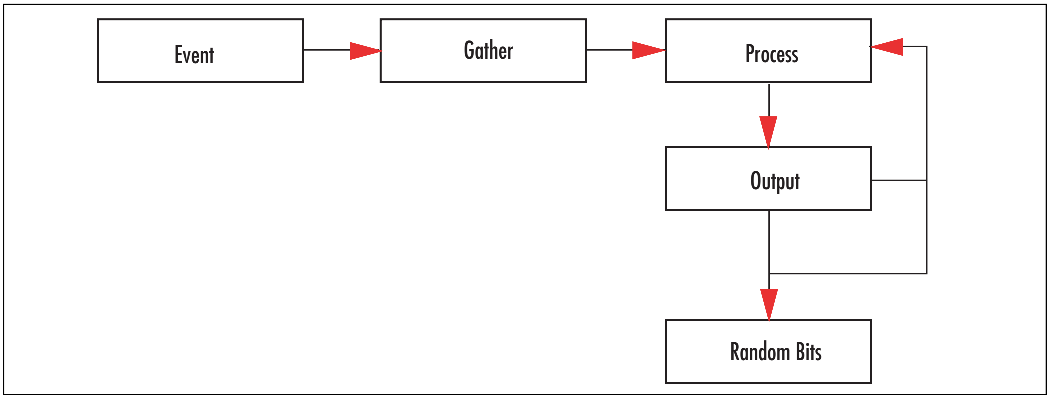
Statistics texts can tell us just how (un)likely a particular set of output bits is, based on the assumption that the generator is good. In practice, though, a little common sense will often be enough; while even 8 heads isn't that unlikely, 2 is definitely unexpected, and 20 pretty much convinces you it was a two-headed coin. The more data you use for the test, the more reliable your intuition should be.

RNG Design

Most RNGs in practice are designed around some event gathering process that follows the pipeline described in Figure 3.1. There are no real standards for RNG design, since the standards bodies tend to focus more on the deterministic side (e.g., PRNGs). Several classic RNGs such as the Yarrow and Fortuna designs (which we will cover later) are flexible and designed by some of the best cryptographers in the field. However, what we truly want is a simple yet effective RNG.

For this task, we will borrow from the Linux kernel RNG. It is a particularly good model, as it is integrated inside the kernel as opposed to an external entity sitting in user space. As such, it has to behave itself with respect to stack, heap, and CPU time resource usage.

Figure 3.1 RNG Flow Diagram



The typical RNG workflow can be broken into the following discrete steps: events, gathering, processing, and output feedback. Not all RNGs follow this overall design concept, but it is fairly common.

The first misconception people tend to have is to assume that RNGs feed off some random source like interrupt timers and return output directly. In reality, most entropy sources are, at least from an information theoretic viewpoint, less than ideal. The goal of the RNG is to extract the entropy from the input events (the seed data) and return only as many random bits to the caller as entropy bits with which it had been seeded. This is actually the only key difference in practice between an RNG and a PRNG.

An RNG also differs from a PRNG in that it is supposed to be fed constantly seeding entropy. A PRNG is meant to be seeded once (or seldom) and then outputs many more bits than the entropy of its internal state. This key difference limits the use of an RNG in situations where seeding events are hard to come by or are of low entropy.

RNG Events

An RNG event is what the RNG will be observing to try to extract entropy from. They are events in systems that do not occur with highly predictable intervals or states. The goal of the RNG is then to capture the event, gather the entropy available, and pass it on to be processed into the RNG internal state.

Events come in all shapes and sizes. What events you use for an RNG depends highly on for what the platform is being developed. The first low-hanging fruit in terms of a desktop or server platform are hardware interrupts, such as those caused by keyboard, mouse, timer (drift), network, and storage devices. Some, if not all, of these are available on most desktop and server platforms in one form of another. Even on embedded platforms with network connections, they are readily available.

Other typical sources are timer skews, analogue-to-digital conversion noise, and diode leakage. Let us first consider the desktop and server platforms.

Hardware Interrupts

In virtually all platforms with hardware interrupts, the process of triggering an interrupt is fairly consistent. In devices capable of asserting an interrupt, they raise a signal (usually a dedicated pin) that a controller (such as the Programmable Interrupt Controller (PIC)) detects, prioritizes, and then signals the processor. The processor detects the interrupt, stops processing the current task, and swaps to a kernel handler, which then acknowledges the interrupt and processes the event. After handling the event the interrupt handler returns control to the interrupted task and processing resumes as normal.

It is inside the handler where we will observe and gather the entropy from the event. This additional step raises a concern, or it should raise a concern for most developers, since the latency of the interrupt handler must be minimal to maintain system performance. If a 1 kHz timer interrupt takes 10,000 cycles to process, it is effectively stealing 10,000,000 cycles every second from the users' tasks (that is, dropping 10 MHz off your processor's speed). Interrupts have to be fast, which implies that what we do to "gather" the entropy must be trivial, perhaps at most a memory copy. This is why there are distinct gather and process steps in the RNG construction. The processing will occur later, usually as a result of a read from the RNG from a user task.

The first piece of entropy we can gather from the interrupt handler is which interrupt this is; that (say) a timer interrupt occurred is not highly uncertain, that it occurred between (say) two different interrupts is, especially in systems under load. Note that timers are tricky to use in RNG construction, which we shall cover shortly.

The next piece of entropy is the data associated with event. For example, for a keyboard interrupt the actual key pressed, for a network interrupt the frame, for a mouse interrupt the co-ordinates or input stream, and so on. (Clearly, copying entire frames on systems under load would be a significant performance bottleneck. Some level of filtering has to take place here.)

The last useful piece of entropy is the capturing of a high-resolution free running counter. This is extremely useful for systems where the load is not high enough for combinations of interrupts to have enough entropy. For example, that a network interrupt occurred may not have a lot of entropy. The exact clock tick at which it occurred can be in doubt and therefore have entropy. This is particularly practical where the clock being read is free running and not tied to the task scheduling or interrupt scheduling process. On x86 processors, the RDTSC instruction can be used for this process as it is essentially a very high-precision free running counter not affected by bus timing, device timing, or even the processor's instruction flow. This last source of entropy is a form of timer skew, which we shall cover in the next section.

WARNING

It is useful for cryptographic purposes to include a high-resolution timer as part of the event to increase the entropy collected. However, on certain platforms capturing the timer can add significant instruction cycle penalties to the process.

On the x86 Intel and AMD platforms, the RDTSC instruction can take from a dozen cycles to a hundred cycles (depending on model and processor state). RDTSC is also a serialization operation, which means the processor must first retire all opcodes before the count is read. It is also very likely atomic on most processors, which means the processor cannot be interrupted.

In the grand scheme of things, a 100 cycle spent is not horrible depending on the frequency of the interrupt. A keyboard interrupt may trigger ~300 times per minute; at 2 GHz, this would mean the user loses 0.00025 milliseconds per second of CPU time to calling the RDTSC instruction.

TIP

On certain platforms, such as the PowerPC found in G3 and G4 Apple computers, the CPU timer is actually just post-scaled off the bus timer. All other devices connected to the FSB and off the Northbridge use the bus timer. In these platforms, tying the CPU timer in along with interrupts is not a bad idea; it simply is not specifically a good thing either. That is, it won't hurt, but it also may not help.

All three pieces of information alone may yield very little entropy, but together will usually have a nonzero amount of entropy sufficient for gathering and further processing.

Timer Skew

Timer skew occurs when two or more circuits are not phase locked; they may oscillate at the same frequency, but the start and stop of a period may shift depending on many factors such as voltage and temperature, to name two. In digital equipment, this usually is combated with the use of reducing the number of distinct clocks, PLL devices, and self-clocking (such as the HyperTransport bus).

In software, it is usually hard to come by timers that are distinct and not locked to one another. For example, the PCI clock should be in sync with every device on the PCI bus. It should also be locked to that of the Southbridge that connects to the PCI controllers, and so on.

In the x86 world, it turns out there is at least one sufficiently useful (albeit slow) source. Yet again, the RDTSC instruction is used. The processor's timer, at least for Intel and AMD, is not directly tied to the PIT or ACPI timers. Entropy therefore can be extracted with a fairly simple loop.

timer_bit.c:

```
#include <signal.h>
#include <stdio.h>

volatile int x, quit, capture;
void sighandle(int signo) { capture = x; quit = 1; }
int main(void)
{
   int y;
   signal(SIGALRM, sighandle);
   for (y = 0; y < 16; y++) {
      quit = 0;
      alarm(1);
      while (!quit) { x ^= 1; }
      printf("%d", capture);
      fflush(stdout);
   }
   printf("\n");
```

```
    return 0;
}
```

This example displays 16 bits and then returns to the shell. In this example, the XOR'ing of the *x* variable with one produces a high frequency clock oscillating between 0 and 1 as fast as the processor can decode, schedule, execute, and retire the operation. The exact frequency of this clock for a given one-second period depends on many factors, including cache contents, other interrupts, other tasks pre-empting this task, and where the processor is in executing the loop when the alarm signals.

On the x86 platform, the while loop would essentially resemble

```
top:
   mov eax, [x]
   xor eax, 1
   mov [x], eax
   mov eax, [quit]
   test eax,eax
   jz top
```

At first, it may seem odd that the compiler would produce three opcodes for the XOR operation when a single XOR with a memory operand would suffice. However, the "`x ^= 1`" operation actually defines three atomic operations since *x* is volatile, a load, an XOR, and a store. From the program's point of view, even with a signal by time we exit the while loop the entire XOR statement has executed. This is why the signal handler will capture the value of *x* before setting the *quit* flag.

The interrupt (or in this case a signal) can stop the program flow before any one of the instructions. Only one of them actually updates the value of *x* in memory where the signal handler can see it. Figure 3.2 shows some sample outputs produced on an Opteron workstation.

Figure 3.2 Sample Output of the timer_bit Routine

```
0010001010111010
0101000110111100
1000101110111111
```

If we continued, we would likely see that there is some form of bias, but at the very least, the entropy is not zero. One problem, however, is that this generator is extremely slow. It takes one second per bit, and the entropy per output is likely nowhere close to one bit per bit. A more conservative estimate would be that there is at most 0.1 bits of entropy per bit output. This means that for a 128-bit sequence, it would take 1280 seconds, or nearly 22 minutes.

One solution would be to decrease the length of the delay on the alarm. However, if we sample for too short a period the likelihood of a frequency shift will be decreased, lowering the entropy of the output. That is, it is more likely that two free running counters remain in synchronization for a shorter period of time than a longer period of time.

Fine tuning the delay is beyond the goal of this text. Every platform, even in the x86 world, and every implementation is different. The power supply, location, and quality of the onboard components will affect how much and how often phase lock between clocks is lost.

In hardware, such situations can be recreated for profit but are typically hard to incorporate. Two clocks, fed from the same power source (or rail), will be exposed to nearly the same voltage provided they are of the same design and have equal length power traces. In effect, you would not expect them to be in phase lock, but would also not expect them to experience a large amount of drift, certainly over a short period of time. It is true that no two clocks will be exactly the same; however, for the purposes of an RNG they will most likely be similar enough that any uncertainty in their states is far too little to be useful. Hardware clocks must be fed from different power rails, such as two distinct batteries. This alone makes them hard to incorporate on a cost basis.

The most practical form of this entropy collection would be with interrupts alongside with a high-resolution free running timer. Even the regularly scheduled system timer interrupt should be gathered with this high precision timer.

Analogue to Digital Errors

Analogue to Digital converts (ADC) capture an input waveform and then quantize and digitize the signal. The result is usually a Pulse Code Modulation stream where groups of bits represent the signal strength at a discrete moment in time. These are typically found in soundcards as microphone or line-in components, on TV tuner decoders, and in wireless radio devices.

As in the case of the timer skews, we are trying to exploit any otherwise perceivable failings in the circuit to extract entropy. The most obvious source is the least significant bit of the samples, which can settle on one value or another depending on when the sample as latched, the comparative voltage going to the ADC, and other environmental noises. In fact, timer skew (signaling when to latch the sample) can add entropy to the retrieved samples.

As an experiment, consider playing a CD and recording the output from a microphone while sitting in a properly isolated sound studio. The chances are very high that the two bit streams will not agree exactly. The cross-correlation between the two will be very strong, especially if the equipment is of high quality. However, even in the best of situations there is still limited entropy to be had.

Another experiment that is easier to attempt is to record on your desktop with (or without) a microphone attached the ambient noise where your machine is located. Even without a microphone, the sound codec on a Tyan 2877 will detect low levels of noise through the ADC. The samples are extremely small, but even in this extreme case there is entropy to be gathered. Ideally, it is best if a recording device such as a microphone is attached so it can capture environmental ambient noise as well.

In practice, ADCs are harder to use in desktop and server setups. They are more likely to be available in custom embedded designs where such components can be used without conflicting with the user. For example, if your OS started capturing audio while you were trying to use the sound card, it may be bothersome. In terms of the data we wish to gather, it is only the least significant bit.This reduces the storage requirement of the gathering process significantly.

RNG Data Gathering

Now that we know a few sources where entropy is to be had, we must design a quick way of collecting it.The goal of the gathering step is to reduce the latency of the event stage such that interrupts are serviced in a reasonably well controlled amount of time.

Effectively, the gathering stage collects entropy and feeds it to the processing stages when not servicing an interrupt. We are pushing when we have to do something meaningful with the entropy not removing the step altogether.

The logical first step is to have a block of memory pre-allocated in which we can dump the data. However, this yields a new problem.The block of memory will be a fixed size, which means that when the block is full, new inputs must either be ignored or the block must be sent to processing. Simply dropping new (or older) events to ensure the buffer is not overflowed is not an option.

An attacker who knows that entropy can be discarded will simply try to trigger a series of low entropy events to ensure the collected data is of low quality.The Linux kernel approaches this problem by using a two-stage processing algorithm. In the first stage, they mix the entropy with an entropy preserving Linear Feedback Shift Register (LFSR). An LFSR is effectively a PRNG device that produces outputs as a linear combination of its internal state. (Why use an LFSR? Suppose we just XORed the bits that came off the devices and there was a bias.The bias would tend to collect on distinct bits in the dumping area.The LFSR, while not perfect, is quick way to make sure the bias is spread across the memory range.) In the Linux kernel, they step the LFSR but instead of just updating the state of the LFSR with a linear combination of itself, they XOR in bits from the events gathered.

The XOR operation is particularly useful for this, as it is what is known as entropy preserving, or simply put, a balanced operation. Obviously, your entropy cannot exceed the size of the state. However, if the entropy of the state is full, it cannot be decreased regardless of the input. For example, consider the One Time Pad (OTP). In an OTP, even if the plaintext is of very low entropy such as English text, the output ciphertext will have exactly one bit of entropy per bit.

LFSR Basics

LFSRs are also particularly useful for this step as they are fast to implement. The basic LFSR is composed of an L-bit register that is shifted once and the lost bit is XOR'ed against select bits of the shifted register. The bits chosen are known as "tap bits". For example,

```
unsigned long clock_lfsr(unsigned long state)
{
   return (state >> 1) ^ ((state & 1) ? 0x800000C5 : 0x00);
}
```

This function produces a 32-bit LFSR with a tap pattern 0x800000C5 (bits 31, 7, 6, 2, and 0). Now, to shift in RNG data we could simply do the following,

```
unsigned long feed_lfsr(unsigned long state, int seed_bit)
{
   state ^= seed_bit;
   return clock_lfsr(state);
}
```

This function would be called for every bit in the gathering block until it is empty. As the LFSR is entropy preserving, at most this construction will yield 32 bits of entropy in the state regardless of how many seed bits you feed—far too little to be of any cryptographic significance and must be added as part of a larger pool.

Table-based LFSRs

Clocking an LFSR one bit at a time is a very slow operation when adding several bytes. It is far too slow for servicing an interrupt. It turns out we do not have to clock LFSRs one bit a time. In fact, we can step them any number of bits, and in particular, with lookup tables the entire operation can be done without branches (the test on the LSB). Strictly speaking, we could use an LFSR over a different field such as extension fields of the form $GF(p^k)^m[x]$. However, these are out of the scope of this book so we shall cautiously avoid them.

The most useful quantity to clock by is by eight bits, which allows us to merge a byte of seed data a time. It also keeps the table size small at one kilobyte.

lfsr32.c:

```
static unsigned long shiftab[256];
unsigned long step_one(unsigned long state)
{
   return (state >> 1) ^ ((state & 1) ? 0x800000C5 : 0x00);
}
```

This is our familiar 32-bit step function which clocks the LFSR once.

```
void make_tab(void)
{
   unsigned long x, y, state;

   /* step through all 8-bit sequences */
012      for (x = 0; x < 256; x++) {
013         state = x;
014         /* clock it 8 times */
015         for (y = 0; y < 8; y++) {
016            state = step_one(state);
017         }
018         /* store it */
019         shiftab[x] = state;
020      }
021   }
```

This function creates the 256 entry table shiftab. We run through all 256 lower eight bits and clock the register eight times. The final product of this (line 19) is what we would have XORed against the register.

```
023   /* clock the LFSR 8 times */
024   unsigned long step_eight(unsigned long state)
025   {
026      return (state >> 8) ^ shiftab[state & 0xFF];
027   }
028
029   /* seed 8 bits of entropy at once */
030   unsigned long feed_eight(unsigned long state,
031                            unsigned char seed)
032   {
033      state ^= seed;
034      return step_eight(state);
035   }
```

The first function (line 24) steps the LFSR eight times with no more than a shift, lookup, and an XOR. In practice, this is actually more than eight times faster, as we are moving the conditional XOR from the code path.

The second function (line 30) seeds the LFSR with eight bits in a single function call. It would be exactly equivalent to feeding one bit (from least significant to most significant) with a single stepping function.

```
037   #include <stdio.h>
038   #include <stdlib.h>
039   #include <time.h>
040   int main(void)
041   {
042      unsigned long x, state, v;
043
044      make_tab();
045      srand(time(NULL));
046      v = rand();
047
048      state = v;
049      for (x = 0; x < 8; x++) state = step_one(state);
050      printf("%08lx stepped eight times: %08lx\n", v, state);
051
052      state = step_eight(v);
053      printf("%08lx stepped eight times: %08lx\n", v, state);
```

```
     return 0;
  }
```

If you don't believe this trick works, consider this demo program over many runs. Now that we have a more efficient way of updating an LFSR, we can proceed to making it larger.

Large LFSR Implementation

Ideally, from an implementation point of view, you want the LFSR to have two qualities: be short and have very few taps. The shorter the LFSR, the quicker the shift is and the smaller the tables have to be. Additionally, if there are few taps, most of the table will contain zero bits and we can compress it.

However, from a security point of view we want the exact opposite. A larger LFSR can contain more entropy, and the higher number of taps the more widespread the entropy will be in the register. The job of a cryptographer is to know how and where to draw the line between implementation efficiency and security properties.

In the Linux kernel, they use a large LFSR to mix entropy before sending it on for cryptographic processing. This yields several pools that occupy space and pollute the processor's data cache. Our approach to this is much simpler and just as effective. Instead of having a large LFSR, we will stick with the small 32-bit LFSR and forward it to the processing stage periodically. The actual step of adding the LFSR seed to the processing pool will be as trivial as possible to keep the latency low.

RNG Processing and Output

The purpose of the RNG processing step is to take the seed data and turn it into something you can emit to a caller without compromising the internal state of the RNG. At this point, we've only linearly mixed all of our input entropy into the processing block. If we simply handed this over to a caller, they could solve the linear equations for the LFSR and know what events are going on inside the kernel. (Strictly speaking, if the entropy of the pool is the size of the pool this is not a problem. However, in practice this will not always be the case.)

The usual trick for the processing stage is to use a cryptographic one-way hash function to take the seed data and "churn" it into an RNG state from which entropy can be derived. In our construction, we will use the SHA-256 hash function, as it is fairly large and fairly efficient.

The first part of the processing stage is mixing the seed entropy from the gathering stage. We note that the output of SHA-256 is effectively eight 32-bit words. Our processing pool is therefore 256 bits as well. We will use a round-robin process of XOR'ing the 32-bit LFSR seed data into one of the eight words of the state. We must collect at least eight words (but possibly more will arrive) from the gathering stage before we can churn the state to produce output.

Actually churning the data is fairly simple. We first XOR in a count of how many times the churn function has been called into the first word of the state. This prevents the RNG from falling into fixed points when used in nonblocking mode (as we will cover briefly).

Next, we append the first 23 bytes of the current RNG pool data and hash the 55-byte string to form the new 256-bit entropy pool from which a caller can read to get random bytes. The hash output is then XOR'ed against the 256-bit gather state to help prevent backtracking attacks.

The first odd thing about this is the use of a churn counter. One mode for the RNG to work in is known as "non-blocking mode." In this mode, we act like a PRNG rather than as an RNG. When the pool is empty, we do not wait for another eight words from the gathering stage; we simply re-churn the existing state, pool, and churn counter. The counter ensures the input to the hash is unique in every churning, preventing fixed points and short cycles. (A fixed point occurs when the output of the hash is equal to the input of the hash, and a short cycle occurs when a collision is found. Both are extremely unlikely to occur, but the cost of avoiding them is so trivial that it is a good safety measure.)

Another oddity is that we only use 23 bytes of the pool instead of all 32. In theory, there is no reason why we cannot use all 32. This choice is a performance concern more than any other. SHA-256 operates (see Chapter 5, "Hash Functions," for more details) on blocks of 64 bytes at once. A message being hashed is always padded by the hash function with a 0x80 byte followed by the 64-bit encoding of the length of the message. This is a technique known as MD-Strengthening. All we care about are the lengths. Had we used the full 64 bytes, the message would be padded with the nine bytes and require two blocks (of 64 bytes) to be hashed to create the output. Instead, by using 32 bytes of state and 23 bytes of the pool, the 9 bytes of padding fits in one block, which doubles the performance.

You may wonder why we include bytes from the previous output cycle at all. After all, they potentially are given to an attacker to observe. The reason is a matter of more practical security than theoretic. Most RNG reads will be to private buffers for things like RSA key generation or symmetric key selection. The output can still contain entropy. The operation of including it is also free since the SHA-256 hash would have just filled the 23 bytes with zeros. In essence, it does not hurt and can in some circumstances help. Clearly, you still need a fresh source of entropy to use this RNG securely. Recycling the output of the hash does not add to the entropy, it only helps prevent degradation.

Let us put this all together now:

rng.c:

```
/* our SHA256 function we need */
void sha256_memory(const unsigned char *in, unsigned long len,
                         unsigned char *out);

/* the LFSR table */
static const unsigned long shiftab[256] = {
   0x00000000, 0x1700001c, 0x2e000038, 0x39000024, 0x5c000070,
   0x4b00006c, 0x72000048, 0x65000054, 0xb80000e0, 0xaf0000fc,
   0x960000d8, 0x810000c4, 0xe4000090, 0xf300008c, 0xca0000a8,
   0xdd0000b4, 0x7000004b, 0x67000057, 0x5e000073, 0x4900006f,
<snip>
   0x9b0000d3, 0xa20000f7, 0xb50000eb, 0x6800005f, 0x7f000043,
   0x46000067, 0x5100007b, 0x3400002f, 0x23000033, 0x1a000017,
   0x0d00000b
};
```

This is the table for the 32-bit LFSR with the taps 0x800000C5. We have trimmed the listing to save space. The full listing is available online.

```
/* portably load and store 32-bit quantities as bytes */
#define STORE32L(x, y)                           \
     { (y)[3] = (unsigned char)(((x)>>24)&255);   \
       (y)[2] = (unsigned char)(((x)>>16)&255);   \
       (y)[1] = (unsigned char)(((x)>>8)&255);    \
       (y)[0] = (unsigned char)((x)&255); }

#define LOAD32L(x, y)                            \
     { x = ((unsigned long)((y)[3] & 255)<<24) | \
           ((unsigned long)((y)[2] & 255)<<16) | \
           ((unsigned long)((y)[1] & 255)<<8)  | \
           ((unsigned long)((y)[0] & 255)); }
```

These two macros are new to us, but will come up more and more in subsequent chapters. These macros store and load little endian 32-bit data (resp.) in a portable fashion. This allows us to avoid compatibility issues between platforms.

```
/* our RNG state */
static unsigned long LFSR,      state[8],    word_count,
                     bit_count, churn_count;
```

This is the RNG internal state. *LFSR* is the current 32-bit word being accumulated. *state* is the array of 32-bit words that forms the gathering pool of entropy. *word_count* counts the number of words that have been added to the *state* from the *LFSR*. *bit_count* counts the number of bits that have been added to the *LFSR*, and *churn_count* counts the number of times the churn function has been called.

The *bit_count* variable is interpreted as a .4 fixed point encoded value. This means that we split the integer into two parts: a 28-bit (on 32-bit platforms, 60 bit on 64-bit platforms) integer and a 4-bit fraction. The value of *bit_count* literally equates to **(bit_count >> 4) + (bit_count/16.0)** using C syntax. This allows us to add fractions of bits of entropy to the pool.

For example, a mouse interrupt may occur, and we add the X,Y buttons and scroll positions to the RNG. We may say all of them have 1 bit of entropy or 0.25 bits per sample fed to the RNG. So, we pass 0.25 * 16 = 4 as the entropy count.

```
/* pool the RNG data comes out of */
static unsigned char pool[32];
static unsigned long pool_len, pool_idx;
```

This is the pool state from the RNG. It contains the data that is to be returned to the caller. The *pool* array holds up to 32 bytes of RNG output, *pool_len* indicates how many bytes are left, and *pool_idx* indicates the next byte to be read from the array.

```
/* add a byte of entropy to the RNG */
void rng_add_byte(unsigned char seed, unsigned entropy)
{
   /* update the LFSR */
086      LFSR ^= seed;
087      LFSR  = (LFSR >> 8) ^ shiftab[LFSR & 0xFF];
088
089      /* credit the bits */
090      bit_count += entropy;
091
092      /* we use a .4 fixed point representation for entropy */
093      if (bit_count >= (32 << 4)) {
094         state[word_count++ & 7] ^= LFSR;
095         bit_count                = 0;
096      }
097   }
```

This function adds a byte of entropy from some event to the RNG state. First, we mix in the entropy (line 86) and update the LFSR (line 87). Next, we credit the entropy (line 90) and then check if we can add this LFSR word to the state (line 93). If the entropy count is equal to or greater than 32 bits, we perform the mixing.

```
099   static void rng_churn_data(void)
100   {
101      unsigned char buf[64];
102      unsigned long x, y;
103
104      /* update churn count and mix in */
105      state[0] ^= churn_count++;
106
107      /* store the state */
108      for (x = 0; x < 8; x++) {
109         STORE32L(state[x], buf + (x << 3));
110      }
111
112      /* copy the output pool as well (only 23 bytes) */
113      for (x = 0; x < 23; x++) {
114         buf[x+32] = pool[x];
115      }
```

At this point, the local array buf contains 55 bytes of data to be hashed. We recall from earlier that we chose 55 bytes to make this routine as efficient as possible.

```
117      /* hash it */
118      sha256_memory(buf, 55, pool);
```

Here we invoke the SHA-256 hash, which hashes the 55 bytes of data in the *buf* array and stores the 32-byte digest in the *pool* array. Do not worry too much about how SHA-256 works at this stage.

```
120      /* mix the output directly into the state */
121      for (x = 0; x < 8; x++) {
122         LOAD32L(y, pool + (x << 3)); state[x] ^= y;
123      }
```

Note that we are XOR'ing the hash output against the state and not replacing it. This helps prevent backtracking attacks. That is, if we simply left the state as is, an attacker who can determine the state from the output can run the PRNG backward or forward. This is less of a problem if the RNG is used in blocking mode.

```

   /* reset states */
   pool_len   = 32;
   pool_idx   = 0;
   word_count = 0;
}

unsigned long rng_read(unsigned char *out,
                       unsigned long  len,
                                 int  block)
{
   unsigned long x, y;

   x = 0;
   while (len) {
     /* can we read? */
     if (pool_len > 0) {
        /* copy upto pool_len bytes */
        for (y = 0; y < pool_len && y < len; y++) {
           *out++ = pool[pool_idx++];
        }
        pool_len -= y;
        len      -= y;
        x        += y;
     } else {
        /* can we churn? (or are non-blocking?)  */
        if (word_count >= 8 || !block) {
           rng_churn_data();
        } else {
           /* we can't so lets return */
           return x;
        }
     }
  }
  return x;
}
```

The read function rng_read() is what a caller would use to return random bytes from the RNG system. It can operate in one of two modes depending on the *block* argument. If it is nonzero, the function acts like a RNG and only reads as many bytes as the pool has to offer. Unlike a true blocking function, it will return partial reads instead of waiting to fulfill the read. The caller would have to loop if they required traditional blocking functionality. This flexibility is usually a source of errors, as callers do not check the return value of the function. Unfortunately, even an error code returned to the caller would not be noticed unless you significantly altered the code flow of the program (e.g., terminate the application).

If the *block* variable is zero, the function behaves like a PRNG and will rechurn the existing state and pool regardless of the amount of additional entropy. Provided the state has enough entropy in it, running it as a PRNG for a modest amount of time should be for all purposes just like running a proper RNG.

WARNING

The RNG presented in this chapter is not thread safe, but is at least real-time compatible. This is particularly important to note as it cannot be directly plugged into a kernel without causing havoc.

The functions rng_add_byte() and rng_read() require locks to prevent more than one caller from being inside the function. The trivial way to solve this is to use a mutex locking device. However, keep in mind in real-time platforms you may have to drop rng_add_byte() calls if the mutex is locked to keep latency at a minimum.

TIP

If you ran a RNG event trapping system based on the rng_add_byte() mechanism alone, the state could have been fed more than 256 bits of entropy and you would never have a way to get at it.

A simple solution is to have a background task that calls rng_read() and buffers the data in a larger buffer from which callers can read from. This allows the system to buffer a useful amount of entropy beyond just 32 bytes. Most cryptographic tasks only require a couple hundred bytes of RNG data at most. A four-kilobyte buffer would be more than enough to keep things moving smoothly.

RNG Estimation

At this point, we know what to gather, how to process it, and how to produce output. What we need now are sources of entropy trapped and a conservative estimate of their entropy. It is important to understand the model for each source to be able to extract entropy from it efficiently. For all of our sources we will feed the interrupt (or device ID) and the least significant byte of a high precision timer to the RNG. After those, we will feed a list of other pieces of data depending on the type of interrupt.

Let us begin with user input devices.

Keyboard and Mouse

The keyboard is fairly easy to trap. We simply want the keyboard scan codes and will feed them to the RNG as a whole. For most platforms, we can assume that scan codes are at most 16 bits long. Adjust accordingly to suit your platform. In the PC world, the keyboard controller sends one byte if the key pressed was one of the typical alphanumeric characters.

When a less frequent key is pressed such as the function keys, arrow keys, or keys on the number pad, the keyboard controller will send two bytes. At the very least, we can assume the least significant byte of the scan code will contain something and upper may not.

In English text, the average character has about 1.3 bits of entropy, taking into account key repeats and other "common" sequences; the lower eight bits of the scan code will likely contain at least 0.5 bits of entropy. The upper eight bits is likely to be zero, so its entropy is estimated at a mere 1/16 bits. Similarly, the interrupt code and high-resolution timer source are given 1/16 bits each as well.

rng_src.c:

```
/* KEYBOARD */
void rng_keyboard(int INT, unsigned scancode, unsigned hrt)
{
   rng_add_byte(INT, 1);                     /* 1/16 bits  */
   rng_add_byte(scancode & 0xFF, 8);         /* 1/2 bits  */
   rng_add_byte((scancode >> 8) & 0xFF, 1); /* 1/16 bits */
   rng_add_byte(hrt, 1);                     /* timer */
}
```

This code is callable from a keyboard, which should pass the interrupt number (or device ID), scancode, and the least significant byte of a high resolution timer. Obviously, where PC AT scan codes are not being used the logic should be changed appropriately. A rough guide is to take the average entropy per character in the host language and divide it by at least two.

For the mouse, we basically use the same principle except instead of a scan code we use the mouse position and status.

rng_src.c:

```
/* MOUSE */
void rng_mouse(int INT, int x, int y, int z,
               int buttons, unsigned hrt)
{
   rng_add_byte(INT, 1);                     /* 1/16 bits  */
   rng_add_byte(x & 255, 2);                 /* 1/8 bits */
   rng_add_byte(y & 255, 2);                 /* 1/8 bits */
   rng_add_byte(z & 255, 1);                 /* 1/16 bits */
   rng_add_byte(buttons & 255, 1);           /* 1/16 bits */
   rng_add_byte(hrt, 1);                     /* timer */
}
```

Here we add the lower eight bits of the mouse x-, y-, and z-coordinates (z being the scroll wheel). We estimate that the x and y will give us 1/8th of a bit of entropy since it is only really the least significant bit in which we are interested. For example, move your mouse in a vertical line (pretend you are going to the File menu); it is unlikely that your x-coordinate will vary by any huge amount. It may move slightly left or right of the upward direction, but for the most part it is straight. The entropy would be on when and where the mouse is, and the "error" in the x-coordinate as you try to move the mouse upward.

We assume in this function that the mouse buttons (up to eight) are packed as booleans in the lower eight bits of the *button* argument. In reality, the button's state contains very little entropy, as most mouse events are the user trying to locate something on which to click.

Timer

The timer interrupt or system clock can be tapped for entropy as well. Here we are looking for the skew between the two. If your system clock and processor clock are locked (say they are based on one another), you ought to ignore this function.

rng_src.c:

```
/* TIMER */
void rng_timer(int INT, unsigned timer, unsigned hrt)
{
   rng_add_byte(INT, 1);                    /* 1/16 bits */
   rng_add_byte(timer^hrt, 1);              /* 1/16 bits */
}
```

We estimate that the entropy is 1/16 bits for the XOR of the two timers.

Generic Devices

This last routine is for generic devices such as storage devices or network devices where trapping all of the user data would be far too costly. Instead, we trap the ID of the device that caused the event and the current high-resolution time.

rng_src.c:

```
/* DEVICE */
void rng_device(     int INT, unsigned minor,
                unsigned major, unsigned hrt)
{
   rng_add_byte(INT, 1);                    /* 1/16 bits */
   rng_add_byte(minor, 1);                  /* 1/16 bits */
   rng_add_byte(major, 1);                  /* 1/16 bits */
   rng_add_byte(hrt, 1);                    /* timer */
}
```

We assume the devices have some form of major:minor identification scheme such as that used by Linux. It could also the USB or PCI device ID if a major/minor is not available, but in that case you would have to update the function to add 16 bits from both major and minor instead of just the lower eight bits.

RNG Setup

A significant problem most platforms face is a lack of entropy when they first boot up. There is not likely to be many (if at all any) events collected by time the first user space program launches.

The Linux solution to this problem is to gather a sizable block of RNG output and write it to a file. On the next bootup, the bits in the file are added to the RNG a rate of one bit per bit. It is important not to use these bits directly as RNG output and to destroy the file as soon as possible.

That approach has security risks since the entire entropy of the RNG can possibly be exposed to attackers if they can read the seed file before it is removed. Obviously, the file would have to be marked as owned by root with permissions 0400.

On platforms on which storage of a seed would be a problem, it may be more appropriate to spend a few seconds reading a device like an ADC. At 8 KHz, a five-second recording of audio should have at least 256 bits of entropy if not more. A simple approach to this would be to collect the five seconds, hash the data with SHA-256, and feed the output to the RNG at a rate of one bit of entropy per bit.

PRNG Algorithms

We now have a good starting point for constructing an RNG if need be. Keep in mind that many platforms such as Windows, the BSDs, and Linux distributions provide kernel level RNG functionality for the user. If possible, use those instead of writing your own.

What we need now, though, are fast sources of entropy. Here, we change entropy from a universal concept to an adversarial concept. From our point of view, it is ok if we can predict the output of the generator, as long as our attackers cannot. If our attackers cannot predict bits, then to them, we are effectively producing random bits.

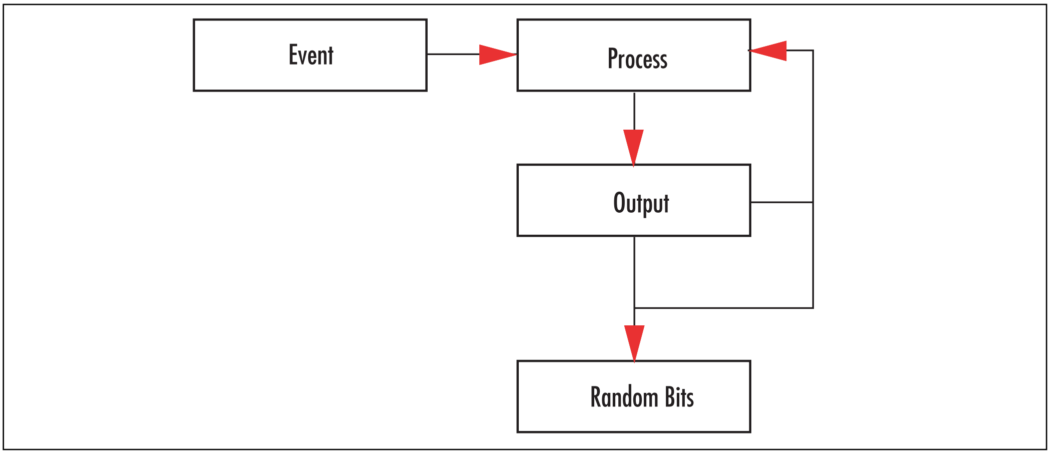
PRNG Design

From a high-level point of view, a PRNG is much like an RNG. In fact, most popular PRNG algorithms such as Fortuna and the NIST suite are capable of being used as RNGs (when event latency is not a concern).

The process diagram (Figure 3.3) for the typical PRNG is much like that of the RNG, except there is no need for a gather stage. Any input entropy is sent directly to the (higher latency) processing stage and made part of the PRNG state immediately.

The goal of most PRNGs differs from that of RNGs. The desire for high entropy output, at least from an outsider's point of view, is still present. When we say "outsider," we mean those who do not know the internal state of the PRNG algorithm. For PRNGs, they are used in systems where there is a ready demand for entropy; in particular, in systems where there is not much processing power available for the output stage. PRNGs must generate high entropy output and must do so efficiently.

Figure 3.3 PRNG Process Diagram



Bit Extractors

Formal cryptography calls PRNG algorithms "bit extractors" or "seed lengtheners." This is because the formal model (Oded Goldreich, Foundations of Cryptography, Basic Tools, Cambridge University Press, 1st Edition) views them as something that literally takes the seed and stretches it to the length desired. Effectively, you are spreading the entropy over the length of the output. The longer the output, the more likely a distinguisher will be able to detect the bits as coming from an algorithm with a specific seed.

Seeding and Lifetime

Just like RNGs, a PRNG must be fed seed data to function as desired. While most PRNGs support re-seeding after they have been initialized, not all do, and it is not a requirement for their security threat model. Some PRNGs such as those based on stream ciphers (RC4, SOBER-128, and SEAL for instance) do not directly support re-seeding and would have to be re-initialized to accept the seed data.

In most applications, the PRNG usefulness can be broken into one of two classifications depending on the application. For many short runtime applications, such as file encryption tools, the PRNG must live for a short period of time and is not sensitive to the length of the output. For longer runtime applications, such as servers and user daemons, the PRNG has a long life and must be properly maintained.

On the short end, we will look at a derivative of the Yarrow PRNG, which is very easy to construct with a block cipher and hash function. It is also relatively quick producing output as fast as the cipher can encrypt blocks. We will also examine the NIST hash based DRBG function, which is more complex but fills the applications where NIST crypto is a must.

On the longer end, we will look at the Fortuna PRNG. It is more complex and difficult to set up, but better suited to running for a length of time. In particular, we will see how the Fortuna design defends against state discovery attacks.

PRNG Attacks

Now that we have what we consider a reasonable looking PRNG, can we pull it apart and break it?

First, we have to understand what breaking it means. The goal of a PRNG is to emit bits (or bytes), which are in all meaningful statistical ways unpredictable. A PRNG would therefore be broken if attackers were able to predict the output more often then they ought to. More precisely, a break has occurred if an algorithm exists that can distinguish the PRNG from random in some feasible amount of time.

A break in a PRNG can occur at one of several spots. We can predict or control the inputs going in as events in an attempt to make sure the entropy in the state is as low as possible. The other way is to backtrack from the output to the internal state and then use low entropy events to trick a user into reading (unblocked) from the PRNG what would be low entropy bytes.

Input Control

First, let us consider controlling the inputs. Any PRNG will fail if its only inputs are from an attacker. Entropy estimation (as attempted by the Linux kernel) is not a sound solution since an attacker may always use a higher order model to generate what the estimator thinks is random data. For example, consider an estimator that only looks at the 0th order statistics; that is, the number of one bits and number of zero bits. An attacker could feed a stream of 01010101… to the PRNG and it would be none the wiser.

Increasing the model order only means the attacker has to be one step ahead. If we use a 1st order model (counting pair of bits), the attacker could feed 0001101100011011… and so on. Effectively, if your attacker controls all of your entropy inputs you are going to successfully be attacked regardless of what PRNG design you choose.

This leaves us only to consider the case where the attacker can control some of the inputs. This is easily proven to be thwarted by the following observation. Suppose you send in one bit of entropy (in one bit) to the PRNG and the attacker can successfully send in the appropriate data to the LFSR such that your bit cancels out. By the very definition of entropy, this cannot happen as your bit had uncertainty. If the attacker could predict it, then the entropy was not one for that bit. Effectively, if there truly is any entropy in your inputs, an attacker will not be able to "cancel" them out regardless of the fact a LFSR is used to mix the data.

Malleability Attacks

These attacks are like chosen plaintext attacks on ciphers except the goal is to guide the PRNG to given internal state based on chosen inputs. If the PRNG uses any data-dependant operations in the process stage, the attacker could use that to control how the algorithm behaves. For example, suppose you only hashed the state if there was not an even balance of zero and one bits. The attacker could take advantage of this and feed inputs that avoided the hash.

Backtracking Attacks

A backtracking attack occurs when your output data leaks information about the internal state of the PRNG, to the point where an attacker can then step the state backward. The goal would be to find previous outputs. For example, if the PRNG is used to make an RSA key, figuring out the previous output gives the attacker the factors to the RSA key.

As an example of the attack, suppose the PRNG was merely an LFSR. The output is a linear combination of the internal state. An attacker could solve for it and then proceed to retrieve any previous or future output of the PRNG.

Even if the PRNG is well designed, learning the current state must not reveal the previous state. For example, consider our RNG construction in rng.c; if we removed the XOR near line 122 the state would not really change between invocations when being used as a PRNG. This means, if the attacker learns the state and we had not placed that XOR there, he could run it forward indefinitely and backward partially.

Yarrow PRNG

The Yarrow design is a PRNG that originally was meant for a long-lived system wide deployment. It achieved some popularity as a PRNG daemon on various UNIX like platforms, but mostly with the advent of Fortuna it has been relegated to a quick and dirty PRNG design.

Essentially, the design hashes entropy along with the existing pool; this pool is then used a symmetric key for a cipher running in CTR mode (see Chapter 4, "Advanced Encryption Standard"). The cipher running in CTR mode and produces the PRNG output fairly efficiently. The actual design for Yarrow specifies how and when reseeding should be issued, how to gather entropy, and so on. For our purposes, we are using it as a PRNG so we do not care where the seed comes from. That is, you should be able to assume the seed has enough entropy to address your threat model.

Readers are encouraged to read the design paper "Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator" by John Kelsey, Bruce Schneier, and Niels Ferguson to get the exact details, as our description here is rather simplistic.

Design

From the block diagram in Figures 3.4 and 3.5, we see that the existing pool and seed are hashed together to form the new pool (or state). The use of the hash avoids backtracking attacks. Suppose the hash of the seed data was simply XOR'ed into the pool; if an attacker knows the pool and can guess the seed data being fed to it, he can backtrack the state.

Figure 3.4 Block Diagram of the Simplified Yarrow PRNG

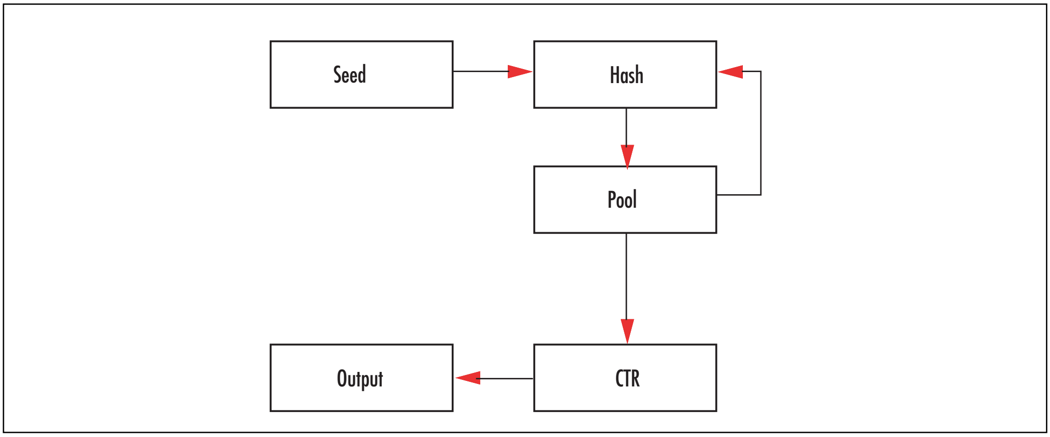


Figure 3.5 Algorithm: Yarrow Reseed

```
Input:
    pool: The current pool
    seed: The seed to add to the pool
Output:
    pool: The newly updated pool

1. $W = pool \,||\, seed$
2. $pool = \mathrm{Hash}(W)$
3. return $pool$
```

The hash of both pieces also helps prevent degradation where the existing entropy pool is used far too long. That is, while hashing the pool does not increase the entropy, it does change the key the CTR block uses. Even though the keys would be related (by the hash), it would be infeasible to exploit such a relationship. The CTR block need not refer to a cipher either; it could be a hash in CTR mode. For performance reasons, it is better to use a block cipher for the CTR block.

In the original Yarrow specification, the design called for the SHA-1 hash function and Blowfish block cipher (or only a hash). While these are not bad choices, today it is better to choose SHA-256 and AES, as they are more modern, more efficient, and part of various standards including the FIPS series. In this algorithm (Figure 3.6) we use the pool as a symmetric key and then proceed to CTR encrypt a zero string to produce output.

Figure 3.6 Algorithm: Yarrow Generate

```
Input:
        pool: The current pool
        IV: The current IV value.
        outlen: The number of bytes to read
Output:
        output: The random bytes
        IV: The new value for the IV

    1. K = Schedule pool as a cipher key (see Chapter 4)
    2. D = CTR($0x00^{outlen}$, K, IV)
    3. Return D, IV.
```

The notation $0x00^{outlen}$ indicates a string of length outlen bytes of all zeroes. In step 2, we invoke the CTR routine for our cipher, which has not been defined yet. The parameters are <*plaintext, key, IV*>, where the *IV* is initially zeroed and then preserved through every call to the reseed and generate functions. Replaying the IV for the same key is dangerous, which is why it is important to keep updating it. The random bytes are in the string *D*, which is the output of encrypting the zero string with the CTR block.

Reseeding

The original Yarrow specification called for system-wide integration with entropy sources to feed the Yarrow PRNG. While not a bad idea, it is not the strong suit of the Yarrow design. As we shall see in the discussion of Fortuna, there are better ways to gather entropy from system events.

For most cryptographic tasks that are short lived, it is safe to seed Yarrow once from a system RNG. A safe practical limit is to use a single seed for at most one hour, or for no more than $2^{(w/4)}$ CTR blocks where w is the bit size of the cipher (e.g., w=128 for AES). For AES, this limit would be 2^{32} blocks, or 64 gigabytes of PRNG output. (On AMD Opteron processors, it is possible to generate 2^{32} outputs in far less than one hour (would take roughly 500 seconds). So, this limitation is actually of practical importance and should be kept in mind.)

Technically, the limit for CTR mode is dictated by the birthday paradox and would be $2^{(w/2)}$; using $2^{(w/4)}$ ensures that no distinguisher on the underlying cipher is likely to be possible. Currently, there are no attacks on the full AES faster than brute force; however, that can change, and if it does, it will probably not be a trivial break. Limiting ourselves to a smaller output run will prevent this from being a problem effectively indefinitely.

These guidelines are merely suggestions. The secure limits depend on your threat model. In some systems, you may want to limit a seed's use to mere minutes or to single outputs. In particular, after generating long-term credentials such as public key certificates, it is best to invalidate the current PRNG state and reseed immediately.

Statefulness

The pool and the current CTR counter value can dictate the entire Yarrowstate (see Chapter 4). In some platforms, such as embedded platforms, we may wish to save the state, or at least preserve the entropy it contains for a future run.

This can be a thorny issue depending on if there are other users on the system who could be able to read system files. The typical solution, as used by most Linux distributions, is to output random data from the system RNG to a file and then read it at startup. This is certainly a valid solution for this problem. Another would be to hash the current pool and store that instead. Hashing the pool itself directly captures any entropy in the pool.

From a security point of view, both techniques are equally valid. The remaining threat comes from an attacker who has read the seed file. Therefore, it is important to always introduce a fresh seed whenever possible upon startup. The usefulness of a seed file is for the occasions when local users either do not exist or cannot read the seed file. This allows the system to start with entropy in the PRNG state even if there are few or no events captured yet.

Pros and Cons

The Yarrow design is highly effective at turning a seed into a lengthy random looking string. It relies heavily on the one-wayness and collision resistance of the hash, and the behavior of the symmetric cipher as a proper pseudo-random permutation. Provided with a seed of sufficient entropy, it is entirely possible to use Yarrow for a lengthy run.

Yarrow is also easy to construct out of basic cryptographic primitives. This makes implementation errors less likely, and cuts down on code space and memory usage.

On the other hand, Yarrow has a very limited state, which always runs the risk of state discovery attacks. While they are not practical, it does run this theoretical risk. As such, it should be avoided for longer term or system-wide deployment.

Yarrow is well suited for many short-lived tasks in which a small amount of entropy is required. Such tasks could be things like command-line tools (e.g., GnuPG), network sensors, and small servers or clients (e.g., DSL router boxes).

Fortuna PRNG

The Fortuna design was proposed by Niels Ferguson and Bruce Schneier[2] as effective an upgrade to the Yarrow design. It still uses the same CTR mechanism to produce output, but instead has more reseeding elements and a more complicated pooling system. (See Niels Ferguson and Bruce Schneier, *Practical Cryptography*, published by Wiley in 2003.)

Fortuna addresses the small PRNG state of Yarrow by having multiple pools and only using a selection of them to create the symmetric key used by the CTR block. It is more suited for long-lived tasks that have periodic re-seeding and need security against malleability and backtracking.

Design

The Fortuna design is characterized mostly by the number of entropy pools you want gathering entropy. The number depends on how many events and how frequently you plan to gather them, how long the application will run, and how much memory you have. A reasonable number of pools is anywhere between 4 and 32; the latter is the default for the Fortuna design.

When the algorithm is started (Figure 3.7), all pools are effectively zeroed out and emptied. A pool counter, *pool_idx*, indicates which pool we are pointing at; *pool0_cnt* indicates the number of bytes added to the zero'th pool; and *reset_cnt* indicates the number of times the PRNG has been reseeded (reset the CTR key).

Figure 3.7 Algorithm: Fortuna Init

Input:

Numpools: Number of Entropy pools to use.

Output:

pool: Array of pools

pool_idx: The pool index

pool0_cnt: The number of bytes in pool zero

reset_cnt: The number of reseedings

IV: The current cipher IV

K: The symmetric key

```
1. For j from 0 to NUMPOOLS – 1 do
   1. pool[j] = null
   2. pool_idx = 0
3. pool0_cnt = 0
4. reset_cnt = 0
5. IV = 0
6. K = null
7. return <pool, pool_idx, pool0_cnt, reset_cnt, IV, K>
```

The pools are not actually buffers for data; in practice, they are implemented as hash states that accept data. We use hash states instead of buffers to allow large amounts of data to be added without wasting memory.

When entropy is added (Figures 3.8 and 3.9), it is prepended with a two-byte header that contains an ID byte and length byte. The ID byte is simply an identifier the developer chooses to separate between entropy sources. The length byte indicates the length in bytes of the entropy being added. The entropy is logically added to the pool_idx'th pool, which as we shall see amounts to adding it the hash of the pool. If we are adding to the zero'th pool, we increment *pool0_cnt* by the number of bytes added. Next, we increment *pool_idx* and wrap back to zero when needed. If during the addition of entropy to the zero'th pool it exceeds a minimal size (say 64 bytes), the reseed algorithm should be called.

Note that in our figures we use a system with four pools. Fortuna is by no means limited to that. We chose four to make the diagrams smaller.

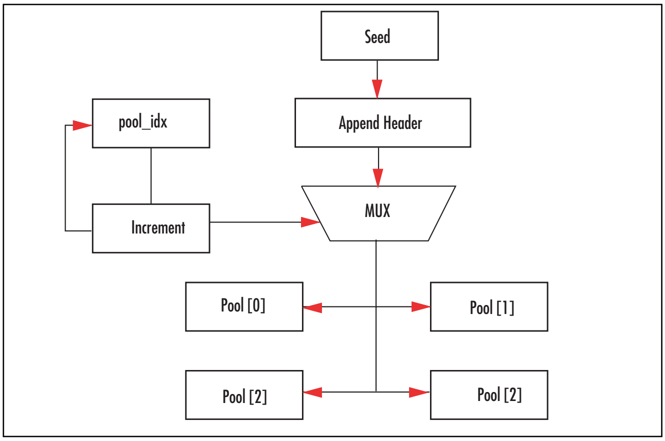
Figure 3.8 Algorithm: Fortuna Add Entropy

```
Input:
    pool: The current pools
    seed: The entropy to add
    seedID: The application (and event) specific seed identifier
    pool0_cnt: The number of bytes in the zero'th pool
    pool_idx: The current pool index
  Output:
    pool: The updated pool
    pool0_cnt: Current number of bytes in the zero'th pool
    pool_idx: New pool index

1. W = seedID || length(seed) || seed
2. Add W to the pool[pool_idx]
```

```
3. If pool_idx = 0 then pool0_cnt = pool0_cnt + length(seed)
4. If pool0_cnt >= 64 call reseed algorithm.
5. pool_idx = (pool_idx + 1) mod NUMPOOLS
6. return <pool, pool0_cnt, pool_idx>
```

Figure 3.9 Fortuna Entropy Addition Diagram



Reseeding will take entropy from select pools and turn it into a symmetric key (K) for the cipher to use to produce output (Figures 3.10 and 3.11). First, the *reset_cnt* counter is incremented. The value of *reset_cnt* is interpreted as a bit mask; that is, if bit x is set, pool x will be used during this algorithm. All selected pools are hashed, all the hashes are concatenated to the existing symmetric key (in order from first to last), and the hash of the string of hashes is used as the symmetric key. All selected pools are emptied and reset to zero. The zero'th pool is always selected.

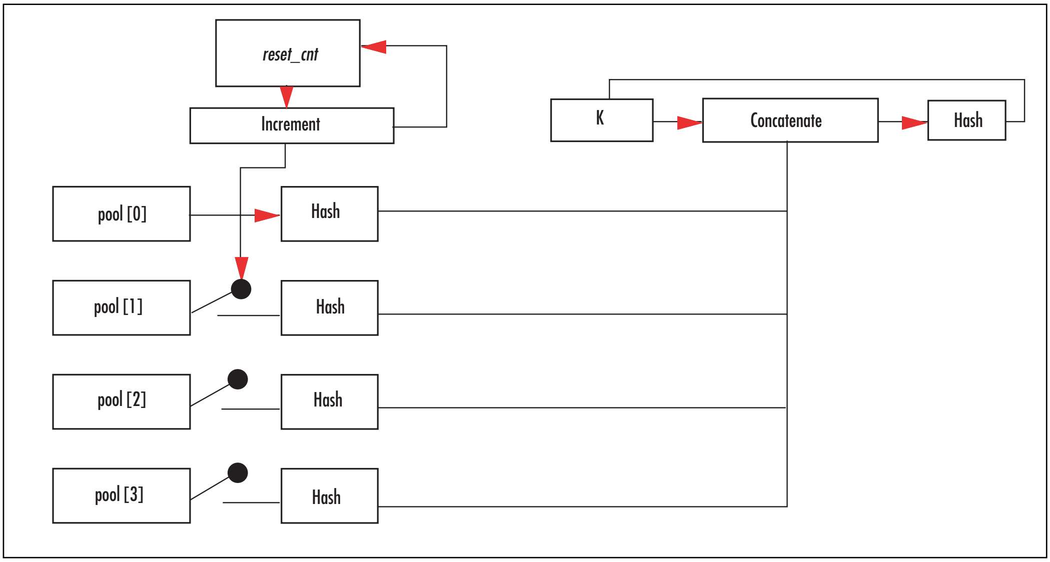
Figure 3.10 Algorithm: Fortuna Reseed

```
Input:
    pool: The current pools
    K: The current symmetric key
    reset_cnt: Current reset counter
Output:
    pool: The updated pool
    K: The new symmetric key
    reset_cnt: updated reset counter
```

Continued

```
1.   $reset\_cnt = reset\_cnt + 1$
2.   $W = K$
3.   for $j$ from 0 to NUMPOOLS – 1
     1.   if ($j$ = 0) OR (((1<<$j$) AND $reset\_cnt$) > 0) then
            i.  $W = W \mid\mid$ hash($pool[j]$)
            ii. $pool[j]$ = null
4.   $K$ = Hash($W$)
5.   return <$pool$, $K$, $reset\_cnt$>
```

Figure 3.11 Fortuna Reseeding Diagram



Extracting entropy from Fortuna is relatively straightforward. First, if a given amount of time has elapsed or a number of reads has occurred, the reseed function is called. Typically, as a system-wide PRNG the time delay should be short; for example, every 10 seconds. If a timer isn't available or this isn't running as a daemon, you could use every 10 calls to the function.

After the reseeding has been handled, the cipher in CTR mode, can generate as many bytes as the caller requests. The IV for the cipher is initially zeroed when the algorithm starts, and then left running throughout the life of the algorithm. The counter is handled in a little-endian fashion incremented from byte 0 to byte 15, respectively.

After all the output has been generated, the cipher is clocked twice to re-key itself for the next usage. In the design of Fortuna they use an AES candidate cipher; we can just use AES for this (Fortuna was described before the AES algorithm was decided upon). Two clocks produce 256 bits, which is the maximally allowed key size. The new key is then scheduled and used for future read operations.

Reseeding

Fortuna does not perform a reseed operation whenever entropy is being added. Instead, the data is simply concatenated to one of the pools (Figure 3.5) in a round-robin fashion. Fortuna is meant to gather entropy from pretty much any source just like our system RNG described in rng.c.

In systems in which interrupt latency is not a significant problem, Fortuna can easily take the place of a system RNG. In applications where there will be limited entropy additions (such as command-line tools), Fortuna breaks down to become Yarrow, minus the re-keying bit). For that reason alone, Fortuna is not meant for applications with short life spans.

A classic example of where Fortuna would be a cromulent choice is within an HTTP server. It gets a lot of events (requests), and the requests can take variable (uncertain) amounts of time to complete. Seeding Fortuna with these pieces of information, and from a system RNG occasionally, and other system resources, can produce a relatively efficient and very secure userspace PRNG.

Statefulness

The entire state of the Fortuna PRNG can be described by the current state of the pools, current symmetric key, and the various counters. In terms of what to save when shutting down it is more complicated than Yarrow.

Simply emitting a handful of bytes will not use the entropy in the later pools that have yet to be used. The simplest solution is to perform a reseeding with *reset_cnt* equal to the all ones pattern subtracted by one; all pools will get affected the symmetric key. Then, emit 32 bytes from the PRNG to the seed file.

If you have the space, storing the hash of each pool into the seed file will help preserve longevity of the PRNG when it is next loaded.

Pros and Cons

Fortuna is a good design for systems that have a long lifetime. It provides for forward secrecy after state discovery attacks by distributing the entropy over long bit extractions. It is based on well-thought-out design concepts, making its analysis a much easier process.

Fortuna is best suited for servers and daemon style applications where it is likely to gather many entropy events. By contrast, it is not well suited for short-lived applications. The design is more complicated than Yarrow, and for short-lived applications, it is totally unnecessary. The design also uses more memory in terms of data and code.

NIST Hash Based DRBG

The NIST Hash Based DRBG (deterministic random bit generator) is one of three newly proposed random functions under NIST SP 800-90. Here we are only talking about the Hash_DRBG algorithm described in section 10.1 of the specification. The PRNG has a set of parameters that define various variables within the algorithm (Table 3.1).

Table 3.1 Parameters for Hash_DRBG

	SHA-1	**SHA-224**	**SHA-256**	**SHA-384**	**SHA-512**
Supported Security Strengths	80	112	128	192	256
highest_supported_security_strength	80	112	128	192	256
Output Block length (**outlen**)	160	224	256	384	512
Required minimum entropy for instantiate and reseed	*security_ strength*				
Minimum entropy input length (**min_length**)					
Maximum entropy input length (**max_length**)	$< 2^{35}$ bits				
Seed Length (**seedlen**) for Hash_DRBG	440	440	440	888	888
max_personalization_string_length	$< 2^{35}$				
max_additional_input_length	$< 2^{35}$				
max_number_of_bits_per_request	$< 2^{19}$				
reseed_interval	$< 2^{18}$				

Design

The internal state of Hash_DRBG consists of:

- A value *V* of *seedlen* bits that is updated during each call to the DRBG.
- A constant *C* of *seedlen* bits that depends on the *seed*.
- A counter *reseed_counter* that indicates the number of requests for pseudorandom bits since new *entropy_input* was obtained during instantiation or reseeding.

The PRNG is initialized through the Hash_DRBG instantiate process (section 10.1.1.2 of the specification).

This algorithm returns a working state the rest of the Hash_DRBG functions can work with (Figure 3.12). It relies on the function hash_df() (Figure 3.13), which we have yet to define. Before that, let's examine the reseed algorithm.

Figure 3.12 Algorithm: Hash_DRBG Instantiate

Input:

entropy_input: The string of bits obtained from an entropy source.

nonce: A second entropy source (or non-repeating source from a PRNG)

Output:

The working state of <*V, C, reseed_counter*>

```
1. seed_material = entropy_input || nonce
2. seed = Hash_df(seed_material, seedlen)
3. V = seed
4. C = Hash_df((0x00 || V), seedlen)
5. reseed_counter = 1
6. Return V, C, reseed_counter
```

Figure 3.13 Algorithm: Hash_DRBG Reseed

Input:

<*V, C, reseed_counter*>: The current working state

entropy_input: String of new bits to add to state

additional_input: String of bits identifying the application (can be null)

Output:

New working state: <*V, C, reseed_counter*>

```
1. seed_material = 0x01 || V || entropy_input || additional_input
2. seed = Hash_df(seed_material, seedlen)
3. V = seed
4. C = Hash_df((0x00 || V), seedlen)
5. reseed_counter = 1
6. Return <V, C, reseed_counter>
```

This algorithm (Figure 3.14) takes new entropy and mixes it into the state of the DRBG. It accepts additional input in the form of application specific bits. It can be null (empty), and generally it is best to leave it that way. This algorithm is much like Yarrow in that it hashes the existing pool (V) in with the new entropy. The seed must be seedlen bits long to be technically following the specification. This alone can make the algorithm rather obtrusive to use.

Figure 3.14 Algorithm: Hash_DRBG Generate

```
Input:
        <V, C, reseed_counter>: The current working state
        requested_number_of_bits: The number of bits to be returned
        additional_input: Application specific input
Output:
        status: status of whether the call was successful
        returned_bits: The number of bits returned
        New working state: <V, C, reseed_counter>

   1.   If reseed_counter > reseed_internal, then return an indication that a reseed is required.
   2.   If (additional_input is not null) then do
        1.   w = Hash(0x02 || V || additional_input)
        2.   $V = (V + w) \bmod 2^{seedlen}$
        3. returned_bits = Hashgen(requested_number_of_bits, V)
        4. H = Hash(0x03 || V)
        5. $V = (V + H + C + reseed\_counter) \bmod 2^{seedlen}$
        6. reseed_counter = reseed_counter + 1
        7.   Return success, returned_bits, and the new values of <V, C, reseed_counter>
```

This function takes the working state and extracts a string of bits. It uses a hash function denoted by Hash() and a new function Hashgen(), which we have yet to present. Much like Fortuna, this algorithm modifies the pool (V) before returning (step 5). This is to prevent backtracking attacks.

This function performs a stretch of the input string to the desired number of bits of output (Figure 3.15). It is similar to algorithm Hashgen (Figure 3.16) and currently it is not obvious why both functions exist when they perform similar functions.

Figure 3.15 Algorithm: Hash_df

```
Input:
    input_string: The string to be hashed.
    no_of_bits_to_return: The number of bits to return.
Output:
    status: Status of whether the call was successful
    requested_bits: The number of bits returned

1. If (no_of_bits_to_return > max_number_of_bits) then return an error.
2. temp = null string
3. len = no_of_bits_to_return / outlen (rounded up)
4. counter = 0x01
5. for j = 1 to len do
       temp = temp || Hash(counter || no_of_bits_to_return || input_string)
       counter = counter + 1
6. requested_bits = leftmost(no_of_bits_to_return) of temp
7. Return success and requested_bits
```

Figure 3.16 Algorithm: Hashgen

```
Input:
    requested_no_of_bits: Number of bits to return
    V: The current value of V
  Output:
    returned_bits: The number of bits returned

1. m = requested_no_of_bits / outlen (rounded up)
2. data = V
3. W = null
4. for j = 1 to m do
    1. w[j] = Hash(data)
    2. W = W || w[j]
    3. data = (data + 1) mod $2^{seedlen}$
    5. returned_bits = leftmost(requested_no_of_bits) of W
    6. return returned_bits
```

This function performs the stretching of the input seed for the generate function. It effectively is similar enough to Hash_df (Figure 3.15) to be confused with one another. The notable difference is that the counter is added to the seed instead of being concatenated with it.

Reseeding

Reseeding Hash_DRBG should follow similar rules as for Yarrow. Since there is only one pool, all sourced entropy is used immediately. This makes the long-term use of it less advisable over, say, Fortuna.

Statefulness

The state of this PRNG consists of the *V*, *C*, and *reseed_counter* variables. In this case, it is best to just generate a random string using the generate function and save that to your seedfile.

Pros and Cons

The Hash_DRBG function is certainly more complicated than Yarrow and less versatile than Fortuna. It addresses some concerns that Yarrow does not, such as state discovery attacks. However, it also is fairly rigid in terms of the lengths of inputs (e.g., seeds) and has several particularly useless duplications. We should note that NIST SP 800-90, at the time of writing, is still a draft and is likely to change.

On the positive side, the algorithm is more robust than Yarrow, and with some fine tuning and optimization, it could be made nearly as simple to describe and implement. This algorithm and the set of SP 800-90 are worth tracking. Unfortunately, at the time of this writing, the comment period for SP 800-90 is closed and no new drafts were available.

Putting It All Together

RNG versus PRNG

The first thing to tackle in designing a cryptosystem is to figure out if you need an RNG or merely a well-seeded PRNG for the generation of random bits. If you are making a completely self-contained product, you will definitely require an RNG. If your application is hosted on a platform such as Windows or Linux, a system RNG is the best choice for seeding an application PRNG.

Typically, an RNG is useful under two circumstances: lack of nonvolatile storage and the requirement for tight information theoretic properties. In circumstances where there is no proper nonvolatile storage, you cannot forward a seed from one runtime of the device (or application) to another. You could use what are known as fuse bits to get an initial state for the PRNG, but re-using it would result in an insecure process. In these situations, an RNG is required to at least seed the PRNG initially at every launch.

In other circumstances, an application will need to remove the assumption that the PRNG produces bits that are indistinguishable from truly random bits. For example, a certificate signing authority really ought to use an RNG (or several) as its source of random bits. Its task is far too important to assume that at some point the PRNG was well seeded.

Fuse Bits

Fuse bits are essentially a form of ROM that is generated after the masking stage of tape-out (tape-out is the name of the final stage of the design of an integrated circuit; the point at which the description of a circuit is sent for manufacture). Each instance of the circuit would have a unique random pattern of bits literally fused into themselves. These bits would serve as the initial state of the PRNG and allow traceability and diagnostics to be performed. The PRNG could be executed to determine correctness and a corresponding host device could generate the same bits (if need be).

Fuse bits are not modifiable. This means your device must never power off, or must have a form of nonvolatile storage where it can store updated copies of the PRNG state. There are ways to work around this limitation. For example, the fuse bits could be hashed with a real-time clock output to get a new PRNG working state. This would be insecure if the real-time clock cannot be trusted (e.g., rolled back to a previous time), but otherwise secure if it was left alone.

Use of PRNGs

For circumstances where an RNG is not a strict requirement, a PRNG may be a more suitable option. PRNGs are typically faster than RNGs, have lower latencies, and in most cases provide the same effective security as an RNG would provide.

The life of a PRNG within an application always begins with at least one reseed operation. On most platforms, a system-wide RNG is available. A short read of at least 256 bits provides enough seed material to start a PRNG in an unpredictable (from an adversary's point of view) state.

Although you need at least one reseed operation, it is not inadvisable to reseed during the life of an application. In practice, even seeds of little to no entropy should be safe to feed to any secure PRNG reseed function. It is ideal to force a reseed after any event that has a long lifetime, such as the generation of user credentials.

When the application is finished with the cryptographic services it's best to either save the state and/or wipe the state from memory. On platforms where an RNG is available, it does not always make sense to write to a seedfile—the exception being applications that need entropy prior to the RNG being available to produce output (during a boot up, for example). In this case, the seedfile should not contain the state of the PRNG verbatim; that is, do not write the state directly to the seedfile. Instead, it should either be the result of calling the PRNG's generate function to emit 256 bits (or more) or be a one-way hash of the current state. This means that an attacker who can read the saved state still can't go backward to recover previous outputs.

Notes from the Underground…

Seeding Do's and Don'ts

Do's

- Do reseed at least once with at least 256 bits from an RNG you trust.
- Do reseed often if possible.
- Do reseed after generating long-term credentials.
- Do hash the state or generate bits for a seedfile.
- Do wipe the state after use.

Don'ts

- Don't reseed with constants.
- Don't reseed with less than 256 bits of RNG entropy.
- Don't always trust user input for entropy (like capturing keystrokes).
- Don't use the same PRNG state for extended periods of time without reseeding.
- Don't leak the PRNG state at anytime.
- Don't use the PRNG state as a seedfile for future use.

Example Platforms

Desktop and Server

Desktop and Server platforms are often similar in their use of base components. Servers are usually differentiated by their higher specification and the fact they are typically multiway (e.g., multiprocessor). They tend to share the following components:

- High-resolution processor timers (e.g., RDTSC)
- PIT, ACPI, and HPET timers
- Sound codec (e.g., Intel HDA or AC'97 compatible)

- USB, PS/2, serial and parallel ports
- SATA, SCSI, and IDE ports (with respective storage devices attached)
- Network interface controllers

The load on most servers guarantees that a steady stream of storage and network interrupts is available for harvesting. Desktops typically will have enough storage interrupts to get an RNG going.

In circumstances where interrupts are sparse, the timer skew and ADC noise options are still open. It would be trivial as part of a boot script to capture one second of audio and feed it to the RNG. The supply of timer interrupts would ensure there is at least a gradual stream of entropy.

We mentioned the various ports readably available on any machine such as USB, PS/2, serial and parallel ports because various vendors sell RNG devices that feed the host random bits. One popular device is the SG100 (www.protego.se/sg100_en.htm) nine-pin serial port RNG. It uses the noise created by a diode at threshold. They also have a USB variant (www.protego.se/sg200_d.htm) for more modern systems. There are others, such as the RPG100B IC from FDK (www.fdk.co.jp/cyber-e/pi_ic_rpg100.htm), which is a small 32-pin chip that produces up to 250Kbit/sec of random data.

Seedfiles are often a requirement for these platforms, as immediately after boot there is usually not enough entropy to seed a PRNG. The seedfile must be either updated or removed during the shutdown process to avoid re-using the same seed. A safe way to proceed is to remove the seedfile upon boot; that way, if the machine crashes there is no way to reuse the seed.

Consoles

Most video game consoles are designed and produced well before security concerns are thought of. The Sony PS2 brought the broadband adapter without an RNG, which meant that most connections would likely be made in the clear. Similarly, the Sony PSP and Nintendo DS are network capable without any cryptographic facilities.

In these cases, it is easy to say, "just use an RNG," except that you do not have one readily available. What you do have is a decent source of interrupts and interaction, and non-volatile space to store seed data. Entropy on these platforms comes in the form of the user input, usually from a controller of some sort combined with timer data. The RNG presented earlier would be quick enough to minimize the performance impact of such operations. In the case of the Nintendo DS, a microphone is available that also can supply a decent amount of entropy from ADC noise.

For performance reasons, it is not ideal to leave the RNG running for the entire lifetime of the game. Users may tolerate some initial slowdown while the RNG gets fed, but will not tolerate slowdown during gameplay. In these cases, it is best to seed a PRNG with the RNG and use it for any entropy the game requires. Often, entropy is only required to establish connections, after which the cryptosystem is running and requires no further entropy.

Seed management is a bit tricky on these platforms. Ideally, you would like to save entropy for the next run, as the user would like to start playing quickly and ensure the transaction is secure. All popular consoles available as of June 2006 have the capability of storing data, whether an internal hard drive, an external flash memories, or internal to the game cartridge (like in the Nintendo DS cartridges). The trick with these platforms is to save a new seed when you have enough entropy and before the user randomly turns the power off. On most platforms, the power button is software driven and can signal an interrupt. However, we may have to deal with power outages or other shutdowns such as crashes.

In these cases, it is best to invalidate the seed after booting to avoid using it twice. As the program runs, the RNG will gather data, and as soon as enough entropy is available should initialize the PRNG. The PRNG generate function can then be used to generate a new seedfile.

Typically, data written locally is safe from prying eyes. That is, you can write the seed and then use it as your sole source of entropy in future runs. However, the entropy of the seed can degrade, and this process does not work for rentals (such as Nintendo DS games). It is always best to gather at least 256 bits of additional entropy in the RNG before seeding the PRNG and writing the new seedfile.

Network Appliances

Network appliances face the same battle as consoles. They usually do not have a lot of storage and are not in well-conditioned states upon first use. A typical network appliance could be something like a DSL router, probe, camera, or access devices (like RFID locks). They require cryptography to authenticate and encrypt data between some client or server and itself. For example, you may log in via SSL to your DSL router, your remote camera may SFTP files to your desktop, and your RFID door must authenticate an attempt with a central access control server.

These applications are better suited for hardware RNGs than software RNGs, as they are typically custom designs. It is much easier to design a diode RNG into the circuit than a complete finite state machine that requires a one-way hash function. Even worse, most network devices do not have enough interrupts to generate any measurable entropy.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: What is entropy?

A: Entropy is the measure of uncertainty in an observable event, and is typically expressed in terms of bits since it is usually mapped on to binary decision graphs. For example, a perfect coin toss is said to have one bit of entropy since the outcome will be heads or tails with equal probability

Q: What is an event?

A: An event is something the RNG or PRNG algorithm can observe, such as hardware interrupts. Data related to the event such as which event and when it occurred can be extracted for their entropy.

Q: Where do I find a standard RNG design?

A: Unfortunately, there is no (public) RNG design mandated by public governments. What they do standardize on are the RNG tests. Your design must pass them to be FIPS 140-2 certified. In fact, even this isn't enough. To pass certification, it must test itself *every time* it is started. This is to ensure the hardware hasn't failed.

Q: It seems like black art?

A: Yes, RNG design is essentially science and guestimation. How many bits of entropy are there in your mouse movements? It's hard to say exactly, which is why it is suggested to estimate conservatively.

Q: Why should I not just use an RNG all the time?

A: RNGs tend to be slower and block more often than PRNG algorithms (which rarely if ever use blocking). This means it is harder to use an RNG during the runtime of an application. For most purposes, the fact that the PRNG is guaranteed to return a result straight away is better.

Q: What is the practical significance between the short and long lifetime PRNGs?

A: Yarrow, at least as presented in this text, is safe to use for both short and long lifetimes provided it has been seeded frequently. Fortuna spreads its input entropy over a longer runtime, which makes it more suitable for longer run applications. In practice, however, provided the PRNG was well seeded, and you do not generate large amounts of bits, the output would be safe to use.

Q: What other PRNG standards are there?

A: NIST SP 800-90 specifies a cipher, HMAC and Elliptic Curve based PRNGs. ANSI X9.31 specifies a triple-DES bit generator, which has since been amended to include AES support.

Q: Is there anything that's truly random?

A: Yes, the standard model of quantum mechanics allows for processes that are truly random. The most commonly cited example of this is radioactive decay of elements.

It is possible to extract a great deal of entropy from radioactive decay by hooking up a Geiger-counter to a computer. Entropy is extracted by measuring the length of time between pairs of decays. If the time between the first pair of decays is shorter than the second pair, record a zero. If the time between the second pair of decays is longer than the second pair, record a one. Random bits generated in this way have an entropy very close to 1-bit per bit.

It is highly unlikely that there is an underlying pattern to radioactive decay. If there was, Quantum Mechanics as we know it would not be possible. Weigh this against the fact that Quantum Mechanics has been verified to a factor of one in many million. If one of the fundamental tenants of Quantum Mechanics were wrong, it would be deeply surprising.

Aside from quantum mechanics, it is quite possible to have entropy in systems that are deterministic. Consider a particle traveling at a speed exactly equal to the square root of five meters per second. We can never compute all the decimal places of the square root of five, because it is irrational. That means that for any finite expansion, there is always a small degree of entropy in the value. (Of course, we can always do more calculations to obtain the next decimal place, so such expansions are not suitable for cryptography.)

Chapter 4

Advanced Encryption Standard

Solutions in this chapter:

- What Is the Advanced Encryption Standard?
- Block Ciphers
- Design of AES
- Implementation
- Practical Attacks
- Chaining Modes
- Putting It All Together

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

The Advanced Encryption Standard (AES) began in 1997 with an announcement from NIST seeking a replacement for the aging and insecure Data Encryption Standard (DES). At that point, DES has been repeatedly shown to be insecure, and to inspire confidence in the new standard they asked the public to once again submit new designs. DES used a 56-bit secret key, which meant that brute force search was possible and practical. Along with a larger key space, AES had to be a 128-bit block cipher; that is, process 128-bit blocks of plaintext input at a time. AES also had to support 128-, 192-, and 256-bit key settings and be more efficient than DES.

Today it may seem rather redundant to force these limitations on a block cipher. We take AES for granted in almost every cryptographic situation. However, in the 1990s, most block ciphers such as IDEA and Blowfish were still 64-bit block ciphers. Even though they supported larger keys than DES, NIST was forward looking toward designs that had to be efficient and practical in the future.

As a result of the call, 15 designs were submitted, of which only five (MARS, Twofish, RC6, Serpent, and Rijndael) made it to the second round of voting. The other 10 were rejected for security or efficiency reasons. In late 2000, NIST announced that the Rijndael block cipher would be chosen for the AES. The decision was based in part on the third-round voting where Rijndael received the most votes (by a fair margin) and the endorsement of the NSA. Technically, the NSA stated that all five candidates would be secure choices as AES, not just Rijndael.

Rijndael is the design of two Belgian cryptographers Joan Daemen and Vincent Rijmen. It was actually a revisit of the Square block cipher and re-designed to address known attacks. It was particularly attractive during the AES process because of its efficiencies (it is one of the most commonly efficient designs) and the nice cryptographic properties. Rijndael is a *substitution-permutation network* that follows the work of Daemens Ph.D. *wide-trail* design philosophy. It was proven to resist both linear and differential cryptanalysis (attacks that broke DES) and has very good statistical properties in other regards. In fact, Rijndael was the only one of the five finalists to be able to prove such claims. The other security favorite, Serpent, was conjectured to also resist the same attacks but was less favored because it is much slower.

Rijndael (or simply AES now) is patent free, and the creators have given out various reference implementations as public domain. Their fast implementation is actually the basis of most software AES implementations, including those of OpenSSL, GnuPG, and LibTomCrypt. The fast and semi-fast implementations in this text are based off the Rijndael reference code.

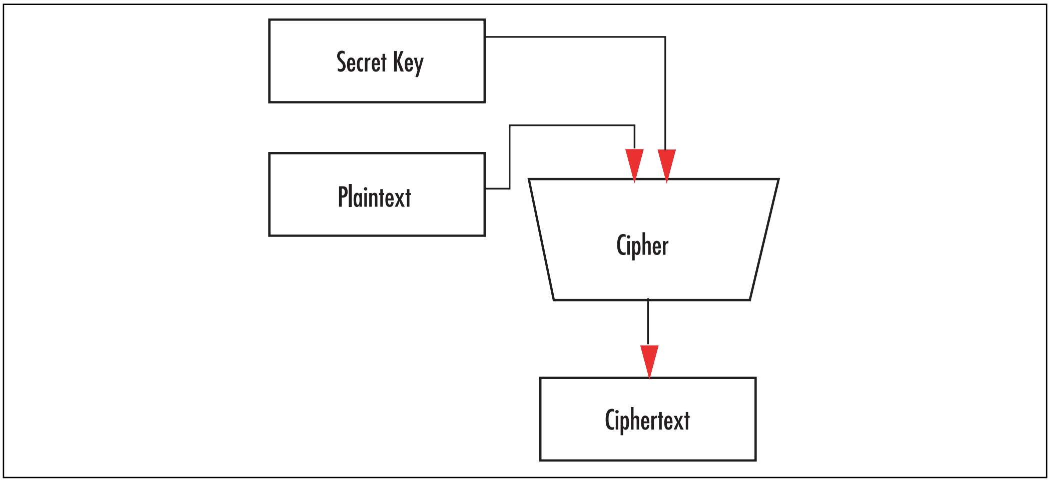
Block Ciphers

Before we go ahead into the design of AES, we should discuss what block ciphers are and the role they fill in cryptography.

The term *block cipher* actually arose to distinguish said algorithms from the normal stream ciphers that were commonplace. A cipher is simply a process to conceal the meaning of a message. Originally, these were in the form of simple substitution ciphers followed by stream ciphers, which would encode individual symbols of a message. In terms of practical use, this usually involved rotors and later shift registers (like LFSRs). The theory seemed to be to find a well-balanced generator of a provably long period, and filter the output through a nonlinear function to create a keystream. Unfortunately, many relatively recent discoveries have made most LFSR-based ciphers insecure.

A block cipher differs from a stream cipher in that it encodes a grouping of symbols in one step. The mapping from plaintext to ciphertext is fixed for a given secret key. That is, with the same secret key the same plaintext will map to the same ciphertext. Most commonly used block ciphers have block sizes of either 64 or 128 bits. This means that they process the plaintext in blocks of 64 or 128 bits. Longer messages are encoded by invoking the cipher multiple times, often with a *chaining mode* such as CTR to guarantee the privacy of the message. Due to the size of the mapping, block ciphers are implemented as algorithms as opposed to as a large lookup table (Figure 4.1).

Figure 4.1 Block Cipher Diagram



Early block ciphers include those of the IBM design team (DES and Lucifer) and eventually a plethora of designs in the 1980s and early 1990s. After AES started in 1997, design submissions to conferences drastically died off. The early series of block ciphers encoded 64-bit blocks and had short keys usually around 64 bits in length. A few designs such as IDEA and Blowfish broke the model and used much larger keys. The basic design of most ciphers was fairly consistent: find a somewhat nonlinear function and iterate it enough times over the plaintext to make the mapping from the ciphertext back to plaintext difficult without the key. For comparison, DES has 16 rounds of the same function, IDEA had 8 rounds, RC5 originally had 12 rounds, Blowfish had 16 rounds, and AES had 10 rounds in their respective designs, to name a few ciphers. (The current consensus is that RC5 is only secure with 16 rounds or more. While you should usually default to using AES, RC5 can be handy where code space is a concern.) By using an algorithm to perform the mapping, the cipher could be very compact, efficient, and used almost anywhere.

Technically speaking, a block cipher is what cryptographers call a Pseudo Random Permutation (PRP). That is, if you ran every possible input through the cipher, you would get as the output a random permutation of the inputs (a consequence of the cipher being a bijection). The secret key controls the order of the permutation, and different keys should choose seemingly random looking permutations.

Loosely speaking, a "good" cipher from a security point of view is one where knowing the permutation (or part of it) does not reveal the key other than by brute force search; that is, an attacker who gathers information about the order of the permutation does not learn the key any faster than trying all possible keys. Currently, this is believed to be the case for AES for all three supported key sizes.

Despite the fact that a block cipher behaves much like a random permutation, it should never be used on its own. Since the mapping is static for a given key the same plaintext block will map to the same ciphertext block. This means that a block cipher used to encrypted data directly leaks considerable data in certain circumstances.

Fortunately, it turns out since we assume the cipher is a decent PRP we can construct various things with it. First, we can construct chaining modes such as CBC and CTR (discussed later), which allow us to obtain privacy without revealing the nature of the plaintext. We can also construct hybrid encrypt and message authentication codes such as CCM and GCM (see Chapter 7, "Encrypt and Authenticate Modes") to obtain privacy and authenticity simultaneously. Finally, we can also construct PRNGs such as Yarrow and Fortuna.

Block ciphers are particularly versatile, which makes them attractive for various problems. As we shall see in Chapter 5, "Hash Functions," hashes are equally versatile, and knowing when to tradeoff between the two is dependent on the problem at hand.

AES Design

The AES (Rijndael) block cipher (see http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf) accepts a 128-bit plaintext, and produces a 128-bit ciphertext under the control of a 128-, 192-, or 256-bit secret key. It is a Substitution-Permutation Network design with a single collection of steps called a *round* that are repeated 9, 11, or 13 times (depending on the key length) to map the plaintext to ciphertext.

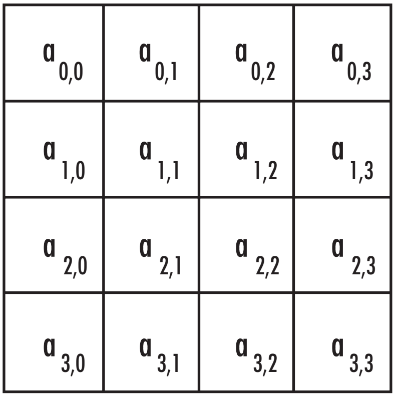
A single round of AES consists of four steps:

1. SubBytes
2. ShiftRows
3. MixColumns
4. AddRoundKey

Each round uses its own 128-bit *round key*, which is derived from the supplied secret key through a process known as a *key schedule*. Do not underestimate the importance of a properly designed key schedule. It distributes the entropy of the key across each of the round keys. If that entropy is not spread properly, it causes all kinds of trouble such as equivalent keys, related keys, and other similar distinguishing attacks.

AES treats the 128-bit input as a vector of 16 bytes organized in a column major (big endian) 4x4 matrix called the *state*. That is, the first byte maps to $a_{0,0}$, the third byte to $a_{3,0}$, the fourth byte to $a_{0,1}$, and the 16th byte maps to $a_{3,3}$ (Figure 4.2).

Figure 4.2 The AES State Diagram



$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

The entire forward AES cipher then consists of

```
1. AddRoundKey(round=0)
2. for round = 1 to Nr-1 (9, 11 or 13 depending on the key size) do
    1. SubBytes
    2. ShiftRows
    3. MixColumns
    4. AddRoundKey(round)
3. SubBytes
4. ShiftRows
5. AddRoundKey(Nr)
```

Finite Field Math

The AES cipher can actually be fully specified as a series of scalar and vector operations on elements of the field GF(2). It is not strictly important to master totally the subject of Galois fields to understand AES; however, a brief understanding does help tweak implementations. Do not fret if you do not understand the material of this section on the first read. For the most part, unless you are really optimizing AES (especially for hardware), you do not have to understand this section to implement AES efficiently.

GF(p) means a finite field of characteristic p. What does that mean? First, a quick tour of algebraic *groups.* We will cover this in more detail in the discussion of GCM and public key algorithms. A group is a collection of elements (elements can be numbers, polynomials, points on a curve, or anything else you could possibly place into an order with a group operator) with at least one well-defined group operation that has an identity, a zero, and an inverse.

A *ring* is a group for which addition is the group operation; note that this does not strictly mean integer addition. An example of a ring is the ring of integers denoted as $\mathbb{Z}$. This group contains all the negative and positive whole numbers. We can create a finite ring by taking the integers modulo an integer, the classic example of which is clock arithmetic—that is, the integers modulo 12. In this group, we have 12 elements that are the integers 0 through 11. There is an identity, 0, which also happens to be the zero. Each element *a* has an inverse—*a*. As we shall see with elliptic curve cryptography, addition of two points on a curve can be a group operation.

A *field* is a ring for which there also exists a multiplication operator. The field elements of a group are traditionally called *units*. An example of a field would be the field of rational numbers (numbers of the form *a/b*). A finite field can be created, for example, by taking the integers modulo a prime. For example, in the integers modulo 5 we have five elements (0 through 4) and four units (1 through 4). Each element follows the rules for a finite ring. Additionally, every unit follows the rules for a finite field. There exists a multiplicative identity, namely the element 1. Each unit *a* has a multiplicative inverse defined by *1/a* modulo 5. For instance, the inverse of 2 modulo 5 is 3, since 2*3 modulo 5 = 1.

Consicely, this means we have a group of elements that we can add, subtract, multiply, and divide. Characteristic p means a field of p elements (p–1 units since zero is not a unit) modulo the prime p. GF(p) is only defined when p is prime, but can be extended to extension fields denoted as GF(p^k). Extension fields are typically implemented in some form of polynomials basis; that is to say, an element of GF(p^k) is a polynomial of degree *k–1* with coefficients from GF(p).

AES uses the field GF(2), which effectively is the field of integers modulo 2. However, it extends it to the field of polynomials, usually denoted as GF(2^8), and at times treats it as a vector, denoted as GF(2)^8. As the smallest unit, AES deals with 8-bit quantities, which is often written as GF(2^8). This is technically incorrect, as it should be written as GF(2)[*x*] or GF(2)^8. This representation indicates a vector of eight GF(2) elements, or simply, eight bits.

Things get a bit messy for the newcomer with the polynomial representation denoted by GF(2)[*x*]. This is the set of polynomials in *x* with coefficients from GF(2). This representation means that we treat the vector of eight bits as coefficients to a seventh degree polynomial (e.g., $<a_0,a_1,a_2,a_3,a_4,a_5,a_6,a_7>$ turns into the polynomial $p(x) = a_0x^0 + a_1x^1 + \ldots + a_7x^7$).

To create a field, we have to use a polynomial that is not divisible by any lower degree polynomial from the same basis (GF(2)). Typically, such polynomials are known as *irreducible* polynomials. We define the entire field as $GF(2)[x]/v(x)$, where $v(x)$ is the irreducible polynomial. In the case of AES, the polynomial $v(x) = x^8 + x^4 + x^3 + x + 1$ was chosen. When the bits of $v(x)$ are stored as an integer, it represents the value 0x11B. The curious reader may have also heard of the term *primitive polynomial*, which means that the polynomial $g(x) = x$ would generate all units in the field. By *generate*, we mean that there is some value of k such that $g(x)^k = y(x)$ for all $y(x)$ in $GF(2)[x]/v(x)$. In the case of AES, the polynomial $v(x)$ is not primitive, but the polynomial $g(x) = x + 1$ is a generator of the field.

Addition in this field is a simple XOR operation. If both inputs are of degree 7 or less, no reduction is required. Otherwise, the result must be reduced modulo $v(x)$. Multiplication is just like multiplying any other polynomial. If we want to find $c(x) = a(x)b(x)$, we simply find the vector <c0, c1, c2, …, c15>, where c_n is equal to the sum of all the products for that coefficient. In C, the multiplication is

```
/* return ab mod v(x) */
unsigned gf_mul(unsigned a, unsigned b)
{
   unsigned res;
   res = 0;
   while (a) {
    /* if bit of a is set add b */
    if (a & 1) res ^= b;

    /* multiply b by x */
    b <<= 1;

    /* reduce it modulo 0x11B which is the AES poly */
    if (b & 0x100) b ^= 0x11B;

    /* get next bit of a */
    a >>= 1;
  }
  return res;
}
```

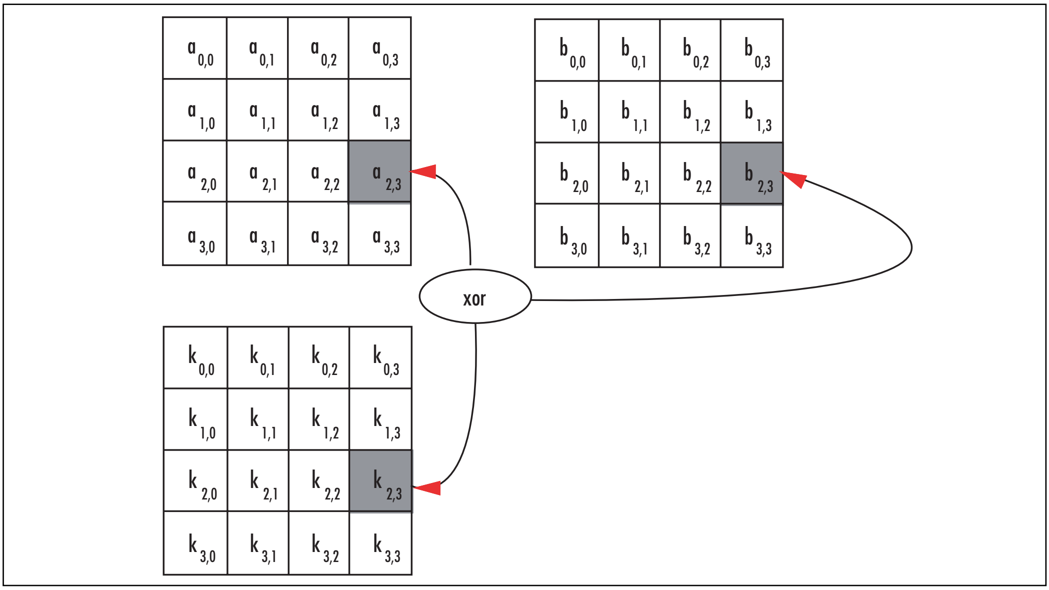
This algorithm is a trivial double-and-add algorithm (analogous to square-and-multiply as we shall see with RSA), which computes the product *ab* in the field that AES uses. In the loop, we first test if the least significant bit of *a* is set. If so, we add the value of *b* to *res*. Next, we multiply *b* by *x* with a shift. If a vector of bits represents the polynomial, then inserting a zero bit on the right is equivalent. Next, we reduce *b* modulo the AES polynomial. Essentially, if the eighth bit is set, we subtract (or XOR) the modulus from the value.

This polynomial field will be used both in the SubBytes and MixColumn stages. In practice, at least in software, we do not implement the multiplication this way, as it would be too slow.

AddRoundKey

This step of the round function adds (in GF(2)) the round key to the state. It performs 16 parallel additions of key material to state material. The addition is performed with the XOR operation (Figure 4.3).

Figure 4.3 AES AddRoundKey Function


$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$
xor
$k_{0,0}$ $k_{0,1}$ $k_{0,2}$ $k_{0,3}$
$k_{1,0}$ $k_{1,1}$ $k_{1,2}$ $k_{1,3}$
$k_{2,0}$ $k_{2,1}$ $k_{2,2}$ $k_{2,3}$
$k_{3,0}$ $k_{3,1}$ $k_{3,2}$ $k_{3,3}$

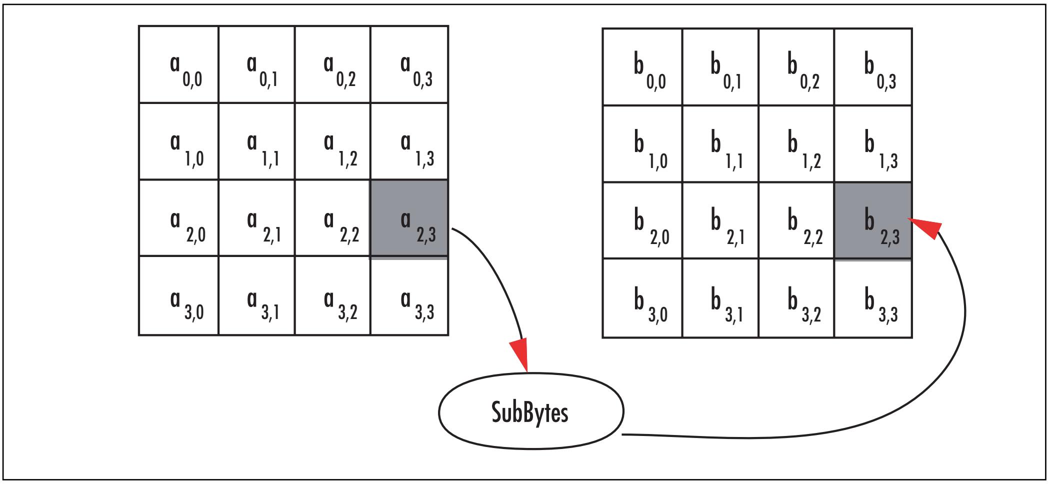
The *k* matrix is a round key and there is a unique key for each round. Since the key addition is a simple XOR, it is often implemented as a 32-bit XOR across rows in 32-bit software.

SubBytes

The SubBytes step of the round function performs the nonlinear confusion step of the SPN. It maps each of the 16 bytes in parallel to a new byte by performing a two-step substitution (Figure 4.4).

The substitution is composed of a multiplicative inversion in GF(2)[*x*]/v(*x*) followed by an affine transformation (Figure 4.5) in $GF(2)^8$. The multiplicative inverse of a unit *a* is another unit *b*, such that *ab* modulo the AES polynomial is congruent (equivalent to, or equal to when reduced by *v*(*x*)) to the polynomial p(*x*) = 1. For AES, we make the exception that the inverse of a(*x*) = 0 is itself.

Figure 4.4 AES SubBytes Function



$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$
SubBytes

Figure 4.5 AES Affine Transformation

$$\begin{bmatrix} 1&0&0&0&1&1&1&1\\ 1&1&0&0&0&1&1&1\\ 1&1&1&0&0&0&1&1\\ 1&1&1&1&0&0&0&1\\ 1&1&1&1&1&0&0&0\\ 0&1&1&1&1&1&0&0\\ 0&0&1&1&1&1&1&0\\ 0&0&0&1&1&1&1&1 \end{bmatrix} \begin{bmatrix} x_0\\ x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7 \end{bmatrix} + \begin{bmatrix} 1\\ 1\\ 0\\ 0\\ 0\\ 1\\ 1\\ 0 \end{bmatrix}$$

There are several ways to find the inverse. Since the field is small, brute force requiring on average 128 multiplications can find it. With this approach we simply multiply *a* by all units in the field until the product is one.

```
unsigned gf_inv_brute(unsigned x)
{
   unsigned y;
   if (x == 0) return 0;
   for (y = 1; y < 256; y++) {
      if (gf_mul(x, y) == 1) return y;
   }
}
```

It can also be found using the power rules. The order of the field GF(2^8) is $2^8 - 1 = 255$ and $a(x)^{254} = a(x)^{-1}$. Computing $a(x)^{254}$ can be accomplished with eight squarings and seven multiplications. We list them separately, since squaring in GF(2) is a O(n) time operation (as opposed to the O(n^2) that multiplication requires).

```
unsigned gf_inv_power(unsigned x)
{
   unsigned y, z;
   y = 1;
   for (z = 0; z < 7; z++) {
      y = gf_mul(gf_mul(y, y), x);
   }
   return gf_mul(y, y);
}
```

Here we used gf_mul to perform the squarings. However, there are faster and more hardware friendly ways of accomplishing this task. In GF(2)[x] the squaring operation is a simple bit shuffle by inserting zero bits between the input bits. For example, the value 1101_2 becomes 10100010_2. After the squaring, a reduction would be required. In software, for AES at least, the code space used to perform the function would be almost as large as the SubBytes function itself. In hardware, squaring comes up in another implementation trick we shall discuss shortly.

It can also be found by the Euclidean algorithm and finally by using log and anti-log (logarithm) tables. For software implementations, this is all overkill. Either the SubBytes step will be rolled into ShiftRows and MixColumns (as we will discuss shortly), or is implemented entirely as a single 8x8 lookup table.

After the inversion, the eight bits are sent through the affine transformation and the output is the result of the SubBytes function. The affine transform is denoted as

Where the vector $<x_0, x_1, x_2, \ldots, x_7>$ denotes the eight bits from least to most significant. Combined, the SubByte substitution table for eight bit values is as shown if Figure 4.6.

Figure 4.6 The AES SubBytes Table

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1x	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2x	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3x	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4x	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5x	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6x	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7x	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8x	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9x	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
Ax	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
Bx	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
Cx	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
Dx	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e

```
Ex |e1 f8 98 11 69 d9 8e 94 9b 1e 87 e9 ce 55 28 df
Fx |8c a1 89 0d bf e6 42 68 41 99 2d 0f b0 54 bb 16
```

The inverse of SubBytes is the inverse of the affine transform followed by the multiplicative inverse. It can be easily derived from the SubBytes table with a short loop.

```
int x;
for (x = 0; x < 256; x++) InvSubBytes[SubBytes[x]] = x;
```

Hardware Friendly SubBytes

This section discusses a trick for hardware implementations[1] that is not useful for software (information thanks to Elliptic Semiconductor—www.ellipticsemi.com). We include it here since the information is not that widespread and tends to be very handy if you are in the hardware market.

The hardware friendly SubBytes implementation takes advantage of the fact that you can compute the multiplicative inverse by computing a much smaller inverse and then applying some trivial GF(2) math to the output. This allows the circuit to be smaller and in most cases faster than an 8x8 ROM table implementation.

The general flow of this algorithm is the following.

1. Apply a forward linear mapping to the 8-bit input.
2. Split the 8-bit result into two 4-bit words *b* and *c* (*b* being the most significant nibble).
3. Compute d = $((b^2 * r(x)) \text{ XOR } (c*b) \text{ XOR } c^2)^{-1}$.
4. p = b * d.
5. q = (c XOR b) * d.
    1. Alternatively, you can use q = (c * d) XOR p.
6. Apply the inverse linear mapping to the 8-bit word p || q.

Where the multiplications are GF(2)[x]/t(x) multiplications modulo t(x) = $x^4 + x^1 + 1$, the value of r(x) is $x^3 + x^2 + x$, and the modular inversion (step 3) is a 4-bit inversion modulo t(x). The forward linear mapping is defined as the following 8x8 matrix (over GF(2)).

```
1  0  0  0  0  0  0  0
0  1  1  0  0  1  0  0
0  1  0  1  0  0  1  0
0  0  0  0  0  0  1  0
1  0  0  1  1  1  0  0
1  0  0  0  1  0  1  1
1  0  0  0  1  1  0  0
0  0  1  0  0  1  1  1
```

The inverse linear mapping is the inverse matrix transform over GF(2). The inverse mapping follows.

```
1  0  0  0  0  0  0  0
0  0  1  1  1  0  1  0
0  0  0  0  0  1  1  1
0  0  0  0  1  0  1  0
1  1  1  1  1  1  1  1
0  1  1  1  1  1  0  1
0  0  0  1  0  0  0  0
0  1  1  0  1  0  1  1
```

This trick takes advantage of the fact that squaring is virtually a free operation. If the input to the squaring is the vector $<x_0,x_1,x_2,x_3>$ (x_0 being the least significant bit), then these four equations can specify the output vector.

$y_0 = x_0 \text{ XOR } x_2$

$y_1 = x_2$

$y_2 = x_1 \text{ XOR } x_3$

$y_3 = x_3$

The output vector $<y_0,y_1,y_2,y_3>$ is the square of the input vector modulo the polynomial t(x). You can implement the 4-bit multiplications in parallel since they are very small. The multiplication by r(x) is constant so it does not have to be implemented as a full GF(2)[x] multiplier. The 4-bit inversion can either be performed by a ROM lookup, decomposed (using this trick recursively), or let the synthesizer attempt to optimize it.

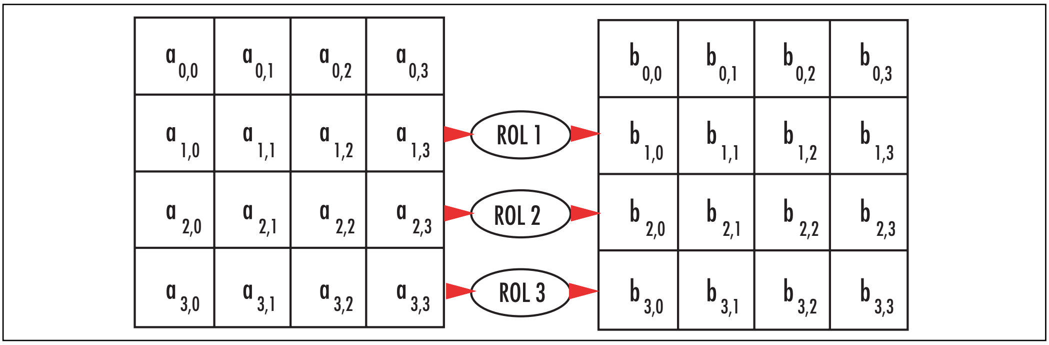
After applying the six-step algorithm to the 8-bit input, we now have the modular inverse. Applying the AES affine transformation completes the SubBytes transformation.

ShiftRows

The ShiftRows step cyclically shifts each row of the state to the left by 0, 1, 2, and 3 positions (respectively). It's entirely linear (Figure 4.7).

In practice, we will see this is implemented through *renaming* rather than an actual shift. That is, instead of moving the bytes around, we simply change from where we fetch them. In 32-bit software, we can combine shift rows fairly easily with SubBytes and MixColumns without having to swap bytes around.

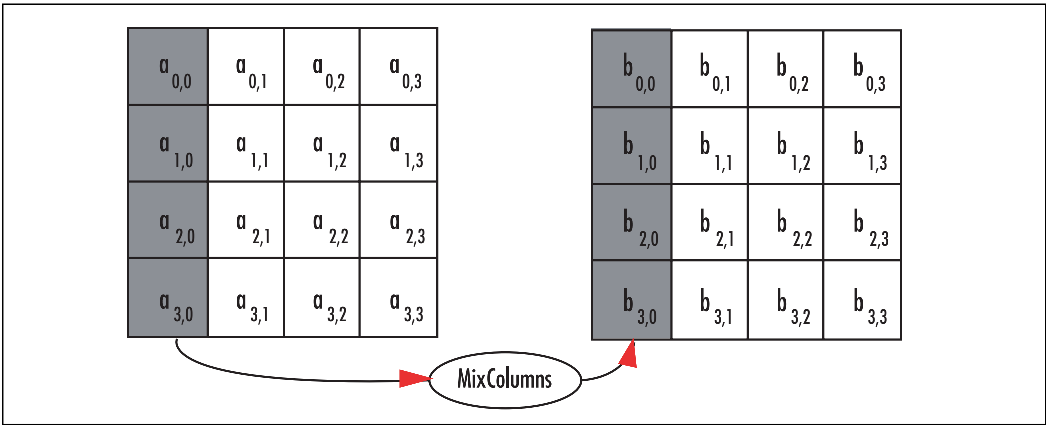
Figure 4.7 The AES ShiftRows Function


$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
ROL 1
ROL 2
ROL 3
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$

MixColumns

The MixColumns step multiplies each column of the state by a 4x4 transform known as a Maximally Distance Separable (MDS). The purpose of this step is to spread differences and make the outputs linearly dependant upon other inputs. That is, if a single input byte changes (and all other input bytes remain the same) between two plaintexts, the change will spread to other bytes of the state as fast as possible (Figure 4.8).

Figure 4.8 The AES MixColumns Function


$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$
MixColumns

In the case of AES, they chose an MDS matrix to perform this task. The MDS transforms actually form part of *coding theory* responsible for things such as error correction codes (Reed-Solomon). They have a unique property that between two different *k*-space inputs the sum of differing input and output co-ordinates is always at least *k+1*. For example, if we flip a bit of one of the input bytes between two four-byte inputs, we would expect the output to differ by at least (4+1)–1 bytes.

MDS codes are particularly attractive for cipher construction, as they are often very efficient and have nice cryptographic properties. Currently, several popular ciphers make use of MDS transforms, such as Square, Rijndael, Twofish, Anubis and Kazhad (the latter two are part of the NESSIE standardization project).

The SubBytes function provides the nonlinear component of the cipher, but it only operates on the bytes of the state in parallel. They have no affect on one another. If AES had MixColumns removed, the cipher would be trivial to break and totally useless.

The MDS matrix is generated by the vector <2,3,1,1>, which is expressed in Figure 4.9.

Figure 4.9 The AES MDS Matrix

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

The multiplications involved are performed in the field GF(2)[*x*]/v(*x*), which is where the SubBytes step is performed. The actual values of 1, 2, and 3 map directly to polynomials. For instance, 2 = *x* and 3 = *x* + 1.

The MixColumns step can be implemented in various manners depending on the target platform. The simplest implementation involves a function that performs a multiplication by *x*, traditionally called *xtime*.

```
unsigned xtime(unsigned x)
{
   /* multiply by x */
   x <<= 1;
   /* reduce */
   if (x & 0x100) x ^= 0x11B;
   return x;
}

void MixColumn(unsigned char *col)
{
   unsigned char tmp[4], xt[4];
   xt[0]    = xtime(col[0]);
   xt[1]    = xtime(col[1]);
   xt[2]    = xtime(col[2]);
   xt[3]    = xtime(col[3]);
   tmp[0]   = xt[0]  ^ xt[1]  ^ col[1] ^ col[2] ^ col[3];
   tmp[1]   = col[0] ^ xt[1]  ^ xt[2]  ^ col[2] ^ col[3];
   tmp[2]   = col[0] ^ col[1] ^ xt[2]  ^ xt[3]  ^ col[3];
   tmp[3]   = xt[0]  ^ col[0] ^ col[1] ^ col[2] ^ xt[3];
   col[0]   = tmp[0];
   col[1]  = tmp[1];
   col[2]  = tmp[2];
   col[3]  = tmp[3];
}
```

The function accesses data offset by four bytes, which corresponds to the width of the AES state matrix. The function would be called four times with col offset by one byte each time. In practice, the state would be placed in alternating buffers to avoid the double buffering required (copying from tmp[] to col[], for instance). In typical hardware implementations, this direct approach is often the method chosen, as it can be implemented in parallel and has a short critical path.

There is another way to implement MixColumn in software that actually allows us to combine SubBytes, ShiftRows, and MixColumns into a single set of operations. First, consider the round function without ShiftRows or SubBytes.

The MixColumn function is a 32x32 linear function, which can be implemented with four 8x32 lookup tables. We create four 8x32 tables where each byte of the output (for all 256 words) represents the product of MixColumn if the other three input bytes were zero. The following code would generate the table for us using the MixColumn() function.

```
unsigned char mc_tab[4][256][4];
void gen_tab(void)
{
   unsigned char col[16];
   int x, y, z;

   for (y = 0; y < 4; y++) {
     for (x = 0; x < 256; x++) {
       for (z = 0; z < 16; z++) col[z] = 0;
       col[y] = x;
       MixColumn(col);
       for (z = 0; z < 4; z++) {
           mc_tab[y][x][z] = col[z];
       }
     }
   }
}
```

Now, if we map mc_tab to an array of 4*256 32-bit words by loading each four-byte vector in little endian format, we can compute MixColumn as

```
unsigned long MixColumn32(unsigned long col)
{
   return mc_tab[0][col&255]        ^
          mc_tab[1][(col>>8)&255]   ^
          mc_tab[2][(col>>16)&255]  ^
          mc_tab[3][(col>>24)&255];
}
```

Note that we now assume mc_tab is an array of 32-bit words. This works because matrix algebra is commutative. Assume the MixColumns transform is the matrix C, and we have an input vector of <a,b,c,d>; what this optimization is saying is we can compute the product C<a,b,c,d> as C<a,0,0,0> + C<0,b,0,0> + … + C<0,0,0,d>. Effectively, *mc_tab[0][a]* is equivalent to [2, 1, 1, 3]**a* for all 256 values of *a*. Since each component of the vector is only an eight-bit variable, we can simply use a table lookup. That is, we precompute C<a,0,0,0> for all possible values of *a* (and call this mc_tab[0]). We then do the same for C<0,b,0,0>, C<0,0,c,0>, and C<0,0,0,d>. This approach requires one kilobyte of storage per table, and a total of four kilobytes to hold the four tables.

There is tradeoff in this approach. Due to the nature of the matrix, the words in mc_tab[1] are actually equal to the words in mc_tab[0] cyclically rotated by eight bits to the right. Similarly, the words in mc_tab[2] are rotated 16 bits, and mc_tab[3] are rotated 24 bits. This allows us to compress the table space to one kilobyte by trading time for memory. (This also will help against certain forms of timing attacks as discussed later.)

The inverse of MixColumns is the transform shown in Figure 4.10.

Figure 4.10 The AES InvMixColumns Matrix

$$\begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

```
void InvMixColumn(unsigned char *col)
{
   unsigned char tmp[4];
   tmp[0]  = gf_mul(col[0], 14) ^ gf_mul(col[1], 11) ^
             gf_mul(col[2], 13) ^ gf_mul(col[3], 9);
   tmp[1]  = gf_mul(col[1], 14) ^ gf_mul(col[2], 11) ^
             gf_mul(col[3], 13) ^ gf_mul(col[0], 9);
   tmp[2]  = gf_mul(col[2], 14) ^ gf_mul(col[3], 11) ^
             gf_mul(col[0], 13) ^ gf_mul(col[1], 9);
   tmp[3]  = gf_mul(col[3], 14) ^ gf_mul(col[0], 11) ^
             gf_mul(col[1], 13) ^ gf_mul(col[2], 9);
   col[0]  = tmp[0];
   col[1]  = tmp[1];
   col[2]  = tmp[2];
   col[3]  = tmp[3];
}
```

As we can see, the forward transform has simpler coefficients (fewer bits set), which comes into play when choosing how to use AES in the field. The inverse transform can also be implemented with tables, and the same compression trick can also be applied to the implementation. In total, only two kilobytes of tables are required to implement the compressed approach. On eight-bit platforms, the calls to gf_mul() can be replaced with table lookups. In total, one kilobyte of memory would be required.

Last Round

The last round of AES (round 10, 12, or 14 depending on key size) differs from the other rounds in that it applies the following steps:

1. SubBytes
2. ShiftRow
3. AddRoundKey

Inverse Cipher

The inverse cipher is composed of the steps in essentially the same order, except we replace the individual steps with their inverses.

1. AddRoundKey(Nr)
2. for round = Nr-1 downto 1 do
    1. InvShiftRow
    2. InvSubBytes
    3. AddRoundKey(round)
    4. InvMixColumns
3. InvSubBytes
4. InvShiftRow
5. AddRoundKey(0)

In theses steps, the "Inv" prefix means the inverse operation. The key schedule is slightly different depending on the implementation. We shall see that in the fast AES code, moving AddRoundKey to the last step of the round allows us to create a decryption routine similar to the encryption routine.

Key Schedule

The key schedule is responsible for turning the input key into the Nr+1 required 128-bit round keys. The algorithm in Figure 4.11 will compute the round keys.

Figure 4.11 The AES Key Schedule

```
Input:
    Nk      Number of 32-bit words in the key (4, 6 or 8)
    w       Array of 4*(Nk+1) 32-bit words
Output:
    w       Array setup with key

  1. Preload the secret key into the first Nk words of w in big endian fashion.
  2. i = Nk
  3. while (i < 4*(Nr+1)) do
      1. temp = w[i – 1]
      2. if (i mod Nk = 0)
          i. temp = SubWord(RotWord(temp)) XOR Rcon[i/Nk]
  3. else if (Nk > 6 and i mod Nk = 4)
          i. temp = SubWord(temp)
  4. w[i] = w[i-Nk] xor temp
  5. i = i + 1
```

The key schedule requires two additional functions. SubWord() takes the 32-bit input and sends each byte through the AES SubBytes substitution table in parallel. RotWord() rotates the word to the right cyclically by eight bits. The Rcon table is an array of the first 10 powers of the polynomial g(x) = x modulo the AES polynomial stored only in the most significant byte of the 32-bit words.

Implementation

There are already many public implementations of AES for a variety of platforms. From the most common reference, implementations are used on 32-bit and 64-bit desktops to tiny 8-bit implementations for microcontrollers. There is also a variety of implementations for hardware scenarios to optimize for speed or security (against side channel attacks), or both. Ideally, it is best to use a previously tested implementation of AES instead of writing your own. However, there are cases where a custom implementation is required, so it is important to understand how to implement it.

We are going to focus on a rather simple eight-bit implementation suitable for compact implementation on microcontrollers. Second, we are going to focus on the traditional 32-bit implementation common in various packages such as OpenSSL, GnuPG, and LibTomCrypt.

An Eight-Bit Implementation

Our first implementation is a direct translation of the standard into C using byte arrays. At this point, we are not applying any optimizations to make sure the C code is as clear as possible. This code will work pretty much anywhere, as it uses very little code and data space and works with small eight-bit data types. It is not ideal for deployment where speed is an issue, and as such is not recommended for use in fielded applications.

aes_small.c:

```
/* The AES Substitution Table */
static const unsigned char sbox[256] = {
0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
<snip>
0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68,
0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 };

/* The key schedule rcon table */
static const unsigned char Rcon[10] = {
0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,  0x1B, 0x36 };
```

These two tables form the constant tables. The first is the SubBytes function implemented as a single table called sbox. The second table is the Rcon table for the key schedule, which is the first 10 powers of g(x) = x.

```
/* The *x function */
static unsigned char xtime(unsigned char x)
{
   if (x & 0x80) { return ((x<<1)^0x1B) & 0xFF; }
   return x<<1;
}
```

This function computes the xtime required for MixColumns. One possible tradeoff would be to implement this as a single 256-byte table. It would avoid the XOR, shift and branch, making the code faster at a cost of more fixed data usage.

```
/* MixColumns:  Processes the entire block */
static void MixColumns(unsigned char *col)
{
  unsigned char tmp[4], xt[4];
  int x;

  for (x = 0; x < 4; x++, col += 4) {
     xt[0]  = xtime(col[0]);
     xt[1]  = xtime(col[1]);
     xt[2]  = xtime(col[2]);
     xt[3]  = xtime(col[3]);
     tmp[0] = xt[0] ^ xt[1] ^ col[1] ^ col[2] ^ col[3];
     tmp[1] = col[0] ^ xt[1] ^ xt[2] ^ col[2] ^ col[3];
     tmp[2] = col[0] ^ col[1] ^ xt[2] ^ xt[3] ^ col[3];
     tmp[3] = xt[0] ^ col[0] ^ col[1] ^ col[2] ^ xt[3];
     col[0] = tmp[0];
     col[1] = tmp[1];
     col[2] = tmp[2];
     col[3] = tmp[3];
  }
}
```

This is the MixColumn function we saw previously, except it has now been modified to work on all 16 bytes of the state. As previously noted, this function is also doubled buffered (copying to tmp[]) and can be optimized to avoid this. We are also using an array xt[] to hold copies of the xtime() output. Since it is used twice, caching it saves time. However, we do not actually need the array. If we first add all inputs, then the xtime() results, we only need a single byte of extra storage.

```
/* ShiftRows:  Shifts the entire block */
static void ShiftRows(unsigned char *col)
{
   unsigned char t;

   /* 2nd row */
   t = col[1]; col[1] = col[5];  col[5] = col[9];
   col[9] = col[13]; col[13] = t;

   /* 3rd row */
   t = col[2]; col[2] = col[10]; col[10] = t;
   t = col[6]; col[6] = col[14]; col[14] = t;

   /* 4th row */
   t = col[15]; col[15] = col[11]; col[11] = col[7];
   col[7] = col[3]; col[3] = t;
}
```

This function implements the ShiftRows function. It uses a single temporary byte *t* to swap around values in the rows. The second and fourth rows are implemented using essentially a shift register, while the third row is a pair of swaps.

```
/* SubBytes */
static void SubBytes(unsigned char *col)
{
   int x;
   for (x = 0; x < 16; x++) {
      col[x] = sbox[col[x]];
   }
}
```

This function implements the SubBytes function. Fairly straightforward, not much to optimize here.

```
/* AddRoundKey */
static void AddRoundKey(unsigned char *col,
                        unsigned char *key, int round)
{
   int x;
   for (x = 0; x < 16; x++) {
       col[x] ^= key[(round<<4)+x];
   }
}
```

This functions implements AddRoundKey function. It reads the round key from a single array of bytes, which is at most 15*16=240 bytes in size. We shift the round number by four bits to the left to emulate a multiplication by 16.

This function can be optimized on platforms with words larger than eight bits by XORing multiple key bytes at a time. This is an optimization we shall see in the 32-bit code.

```
/* Encrypt a single block with Nr rounds (10, 12, 14) */
void AesEncrypt(unsigned char *blk, unsigned char *key, int Nr)
{
   int x;

   AddRoundKey(blk, key, 0);
   for (x = 1; x <= (Nr - 1); x++) {
     SubBytes(blk);
     ShiftRows(blk);
     MixColumns(blk);
     AddRoundKey(blk, key, x);
   }

   SubBytes(blk);
   ShiftRows(blk);
   AddRoundKey(blk, key, Nr);
}
```

This function encrypts the block stored in *blk* in place using the scheduled secret key stored in key. The number of rounds used is stored in Nr and must be 10, 12, or 14 depending on the secret key length (of 128, 192, or 256 bits, respectively).

This implementation of AES is not terribly optimized, as we wished to show the discrete elements of AES in action. In particular, we have discrete steps inside the round. As we shall see later, even for eight-bit targets we can combine SubBytes, ShiftRows, and MixColumns into one step, saving the double buffering, permutation (ShiftRows), and lookups.

```
/* Schedule a secret key for use.
 * outkey[] must be 16*15 bytes in size
 * Nk == number of 32-bit words in the key, e.g., 4, 6 or 8
 * Nr == number of rounds, e.g., 10, 12, 14
 */
void ScheduleKey(unsigned char *inkey,
                 unsigned char *outkey, int Nk, int Nr)
{
   unsigned char temp[4], t;
   int           x, i;

   /* copy the key */
   for (i = 0; i < (4*Nk); i++) {
      outkey[i] = inkey[i];
   }

   i = Nk;
   while (i < (4 * (Nr + 1))) {
      /* temp = w[i-1] */
      for (x = 0; x < 4; x++) temp[x] = outkey[((i-1)<<2) + x];

      if (i % Nk == 0) {
         /* RotWord() */
         t = temp[0]; temp[0] = temp[1];
         temp[1] = temp[2]; temp[2] = temp[3]; temp[3] = t;

         /* SubWord() */
         for (x = 0; x < 4; x++) {
             temp[x] = sbox[temp[x]];
         }
         temp[0] ^= Rcon[(i/Nk)-1];
      } else if (Nk > 6 && (i % Nk) == 4) {
         /* SubWord() */
         for (x = 0; x < 4; x++) {
             temp[x] = sbox[temp[x]];
         }
      }

      /* w[i] = w[i-Nk] xor temp */
      for (x = 0; x < 4; x++) {
         outkey[(i<<2)+x] = outkey[((i-Nk)<<2)+x] ^ temp[x];
      }
      ++i;
   }
}
```

This key schedule is the direct translation of the AES standard key schedule into C using eight-bit data types. We have to emulate RotWords() with a shuffle, and all of the loads and stores are done with a four step for loop.

The obvious optimization is to create one loop per key size and do away with the remainder (%) operations. In the optimized key schedule, we shall see shortly a key can be scheduled in roughly 1,000 AMD64 cycles or less. A single division can take upward of 100 cycles, so removing that operation is a good starting point.

As with AddRoundKey on 32- and 64-bit platforms, we will implement the key schedule using full 32-bit words instead of 8-bit words. This allows us to efficiently implement RotWord() and the 32-bit XOR operations.

```
/** DEMO **/

#include <stdio.h>
int main(void)
{
  unsigned char blk[16], skey[15*16];
  int x, y;
  static const struct {
     int Nk, Nr;
     unsigned char key[32], pt[16], ct[16];
  } tests[] = {
    { 4, 10,
      { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
     { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
       0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff },
     { 0x69, 0xc4, 0xe0, 0xd8, 0x6a, 0x7b, 0x04, 0x30,
       0xd8, 0xcd, 0xb7, 0x80, 0x70, 0xb4, 0xc5, 0x5a }
   }, {
     6, 12,
     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
       0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17 },
     { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
       0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff },
     { 0xdd, 0xa9, 0x7c, 0xa4, 0x86, 0x4c, 0xdf, 0xe0,
       0x6e, 0xaf, 0x70, 0xa0, 0xec, 0x0d, 0x71, 0x91 }
   }, {
     8, 14,
     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
       0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
       0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f },
     { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
       0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff },
     { 0x8e, 0xa2, 0xb7, 0xca, 0x51, 0x67, 0x45, 0xbf,
       0xea, 0xfc, 0x49, 0x90, 0x4b, 0x49, 0x60, 0x89 }
   }
};
```

These three entries are the standard AES test vectors for 128, 192, and 256 key sizes.

```
for (x = 0; x < 3; x++) {
   ScheduleKey(tests[x].key, skey, tests[x].Nk, tests[x].Nr);

   for (y = 0; y < 16; y++) blk[y] = tests[x].pt[y];
   AesEncrypt(blk, skey, tests[x].Nr);
```

Here we are encrypting the plaintext (blk == pt), and are going to test if it equals the expected ciphertext.

Notes from the Underground…

Cipher Testing

A good idea for testing a cipher implementation is to encrypt the provided plaintext more than once; decrypt one fewer times and see if you get the expected result. For example, encrypt the plaintext, and then that ciphertext 999 more times. Next, decrypt the ciphertext repeatedly 999 times and compare it against the expected ciphertext.

Continued

Often, pre-computed table entries can be slightly off and still allow fixed vectors to pass. Its unlikely, but in certain ciphers (such as CAST5) it is entirely possible to pull off.

This test is more applicable to designs where tables are part of a *bijection*, such as the AES MDS transform. If the tables has errors in it, the resulting implementation should fail to decrypt the ciphertext properly, leading to the incorrect output.

Part of the AES process was to provide test vectors of this form. Instead of decrypting N–1 times, the tester would simply encrypt repeatedly N times and verify the output matches the expected value. This catches errors in designs where the elements of the design do not have to be a bijection (such as in Feistel ciphers).

```
    for (y = 0; y < 16; y++) {
       if (blk[y] != tests[x].ct[y]) {
          printf("Byte %d differs in test %d\n", y, x);
          for (y = 0; y < 16; y++) printf("%02x ", blk[y]);
          printf("\n");
          return -1;
       }
    }
 }
 printf("AES passed\n");
 return 0;
}
```

This implementation will serve as our reference implementation. Let us now consider various optimizations.

Optimized Eight-Bit Implementation

We can remove several hotspots from our reference implementation.

1. Implement xtime() as a table.
2. Combine ShiftRows and MixColumns in the round function.
3. Remove the double buffering.

The new xtime table is listed here.

aes_small_opt.c:

```
static const unsigned char xtime[256] = {
0x00, 0x02, 0x04, 0x06, 0x08, 0x0a, 0x0c, 0x0e,
0x10, 0x12, 0x14, 0x16, 0x18, 0x1a, 0x1c, 0x1e,
0x20, 0x22, 0x24, 0x26, 0x28, 0x2a, 0x2c, 0x2e,
<snip>
0xcb, 0xc9, 0xcf, 0xcd, 0xc3, 0xc1, 0xc7, 0xc5,
0xfb, 0xf9, 0xff, 0xfd, 0xf3, 0xf1, 0xf7, 0xf5,
0xeb, 0xe9, 0xef, 0xed, 0xe3, 0xe1, 0xe7, 0xe5 };
```

This lookup table will return the same result as the old function. Now we are saving on a function call, branch, and a few trivial logical operations.

Next, we mix ShiftRows and MixColumns into one function.

aes_small_opt.c:

```
static void ShiftMix(unsigned char *col, unsigned char *out)
{
   unsigned char xt;

#define STEP(i,j,k,l)                                       \
   out[0] = col[j] ^ col[k] ^ col[l];                        \
   out[1] = col[i] ^ col[k] ^ col[l];                        \
   out[2] = col[i] ^ col[j] ^ col[l];                        \
   out[3] = col[i] ^ col[j] ^ col[k];                        \
   xt = xtime[col[i]];  out[0] ^= xt; out[3] ^= xt;          \
   xt = xtime[col[j]];  out[0] ^= xt; out[1] ^= xt;          \
   xt = xtime[col[k]]; out[1] ^= xt; out[2] ^= xt;           \
   xt = xtime[col[l]]; out[2] ^= xt; out[3] ^= xt;           \
   out += 4;

   STEP(0,5,10,15);
   STEP(4,9,14,3);
   STEP(8,13,2,7);
   STEP(12,1,6,11);

#undef STEP
}
```

We did away with the double buffering *tmp* array and are outputting to a different destination. Next, we removed the *xt* array and replaced it with a single unsigned char.

The entire function has been unrolled to make the array indexing faster. In various processors (such as the 8051), accessing the internal RAM by constants is a very fast (one cycle) operation. While this makes the code larger, it does achieve a nice performance boost. Implementers should map *tmp* and *blk* to IRAM space on 8051 series processors.

The indices passed to the STEP macro are from the AES block offset by the appropriate amount. Recall we are storing values in column major order. Without ShiftRows, the selection patterns would be {0,1,2,3}, {4,5,6,7}, and so on. Here we have merged the ShiftRows function into the code by *renaming* the bytes of the AES state. Now byte 1 becomes byte 5 (position 1,1 instead of 1,0), byte 2 becomes byte 10, and so on. This gives us the following selection patterns {0,5,10,15}, {4,9,14,3}, {8, 13, 2, 7}, and {12, 1, 6, 11}.

We can roll up the loop as

```
for (x = 0; x < 16; x += 4) {
  STEP((x+0)&15,(x+5)&15,(x+10)&15,(x+15)&15);
}
```

This achieves a nearly 4x compression of the code when the compiler is smart enough to use CSE throughout the macro. For various embedded compilers, you may need to help it out by declaring i, j, k, and l as local ints. For example,

```
for (x = 0; x < 16; x += 4) {
  int i, j, k, l;
  i = (x+0)&15; j = (x+5)&15; k = (x+10)&15; l = (x+15)&15);
  STEP(i, j, k, l)
}
```

Now when the macro is expanded, the pre-computed values are used. Along with this change, we now need new SubBytes and AesEncrypt functions to accommodate the secondary output buffer.

aes_small_opt.c:

```
/* SubBytes */
static void SubBytes(unsigned char *col, unsigned char *out)
{
   int x;
   for (x = 0; x < 16; x++) {
      out[x] = sbox[col[x]];
   }
}

<snip>

/* Encrypt a single block with Nr rounds (10, 12, 14) */
void AesEncrypt(unsigned char *blk, unsigned char *key, int Nr)
{
   int x;
   unsigned char tmp[16];

   AddRoundKey(blk, key, 0);
   for (x = 1; x <= (Nr - 1); x++) {
     SubBytes(blk, tmp);
     ShiftMix(tmp, blk);
     AddRoundKey(blk, key, x);
   }

   SubBytes(blk, blk);
   ShiftRows(blk);
   AddRoundKey(blk, key, Nr);
}
```

Here we are still using a double buffering scheme (akin to page flipping in graphics programming), except we are not copying back the result without doing actual work. SubBytes stores the result in our local *tmp* array, and then ShiftMix outputs the data back to *blk*.

With all these changes, we can now remove the MixColumns function entirely. The code size difference is fairly trivial on x86 processors, where the optimized copy requires 298 more bytes of code space. Obviously, this does not easily translate into a code size delta on smaller, less capable processors. However, the performance delta should be more than worth it.

While not shown here, decryption can perform the same optimizations. It is recommended that if space is available, tables for the multiplications by 9, 11, 13, and 14 in GF(2)[*x*]/v(*x*) be performed by 256 byte tables, respectively. This adds 1,024 bytes to the code size but drastically improves performance.

TIP

When designing a cryptosystem, take note that many modes do not require the decryption mode of their underlying cipher. As we shall see in subsequent chapters, the CMAC, CCM, and GCM modes of operation only need the encryption direction of the cipher for both encryption and decryption.

This allows us to completely ignore the decryption routine and save considerable code space.

Key Schedule Changes

Now that we have merged ShiftRows and MixColumns, decryption becomes a problem. In AES decryption, we are supposed to perform the AddRoundKey *before* the InvMixColumns step; however, with this optimization the only place to put it afterward[2]. (Technically, this is not true. With the correct permutation, we could place AddRoundKey before InvShiftRows.) However, the presented solution leads into the fast 32-bit implementation. If we let S represent the AES block, K represent the round key, and C the InvMixColumn matrix, we are supposed to compute C(S + K) = CS + CK. However, now we are left with computing CS + K if we add the round key afterward.

The solution is trivial. If we apply InvMixColumn to all of the round keys except the first and last, we can add it at the end of the round and still end up with CS + CK. With this fix, the decryption implementation can use the appropriate variation of ShiftMix() to perform ShiftRows and MixColumns in one step. The reader should take note of this fix, as it arises in the fast 32-bit implementation as well.

Optimized 32-Bit Implementation

Our 32-bit optimized implementation achieves very high performance given that it is in portable C. It is based off the standard reference code provided by the Rijndael team and is public domain. To make AES fast in 32-bit software, we have to merge SubBytes, ShiftRows, and MixColumns into a single shorter sequence of operations. We apply renaming to achieve ShiftRows and use a single set of four tables to perform SubBytes and MixColumns at once.

Precomputed Tables

The first things we need for our implementation are five tables, four of which are for the round function and one is for the last SubBytes (and can be used for the inverse key schedule).

The first four tables are the product of SubBytes and columns of the MDS transform.

1. Te0[x] = S(x) * [2, 1, 1, 3]

2. Te1[x] = S(x) * [3, 2, 1, 1]
3. Te2[x] = S(x) * [1, 3, 2, 1]
4. Te3[x] = S(x) * [1, 1, 3, 2]

Where S(x) is the SubBytes transform and the product is a 1x1 * 1x4 matrix operation. From these tables, we can compute SubBytes and MixColumns with the following code:

```
unsigned long SubMix(unsigned long x)
{
   return Te0[x&255] ^
          Te1[(x>>8)&255] ^
          Te2[(x>>16)&255] ^
          Te3[(x>>24)&255];
}
```

The fifth table is simply the SubBytes function replicated four times; that is, Te4[x] = S(x) * [1, 1, 1, 1].

We note a space optimization (that also plays into the security of the implementation) is that the tables are simply rotated versions of each other's. For example, Te1[x] = RotWord(Te0[x]), Te2[x] = RotWord(Te1[x]), and so on. This means that we can compute Te1, Te2, and Te3 on the fly and save three kilobytes of memory (and possibly cache).

In our supplied code, we have Te0 and Te4 listed unconditionally. However, we provide the ability to remove Te1, Te2, and Te3 if desired with the define SMALL_CODE.

aes_tab.c:

```
static const unsigned long TE0[256] = {
    0xc66363a5UL, 0xf87c7c84UL, 0xee777799UL, 0xf67b7b8dUL,
    0xfff2f20dUL, 0xd66b6bbdUL, 0xde6f6fb1UL, 0x91c5c554UL,
    0x60303050UL, 0x02010103UL, 0xce6767a9UL, 0x562b2b7dUL,
    0xe7fefe19UL, 0xb5d7d762UL, 0x4dababe6UL, 0xec76769aUL,
    0x8fcaca45UL, 0x1f82829dUL, 0x89c9c940UL, 0xfa7d7d87UL,
<snip>
    0x038c8c8fUL, 0x59a1a1f8UL, 0x09898980UL, 0x1a0d0d17UL,
    0x65bfbfdaUL, 0xd7e6e631UL, 0x844242c6UL, 0xd06868b8UL,
    0x824141c3UL, 0x299999b0UL, 0x5a2d2d77UL, 0x1e0f0f11UL,
    0x7bb0b0cbUL, 0xa85454fcUL, 0x6dbbbbd6UL, 0x2c16163aUL,
};

static const unsigned long Te4[256] = {
    0x63636363UL, 0x7c7c7c7cUL, 0x77777777UL, 0x7b7b7b7bUL,
    0xf2f2f2f2UL, 0x6b6b6b6bUL, 0x6f6f6f6fUL, 0xc5c5c5c5UL,
    0x30303030UL, 0x01010101UL, 0x67676767UL, 0x2b2b2b2bUL,
    0xfefefefeUL, 0xd7d7d7d7UL, 0xababababUL, 0x76767676UL,
<snip>
    0xcecececeUL, 0x55555555UL, 0x28282828UL, 0xdfdfdfdfUL,
    0x8c8c8c8cUL, 0xa1a1a1a1UL, 0x89898989UL, 0x0d0d0d0dUL,
    0xbfbfbfbfUL, 0xe6e6e6e6UL, 0x42424242UL, 0x68686868UL,
    0x41414141UL, 0x99999999UL, 0x2d2d2d2dUL, 0x0f0f0f0fUL,
    0xb0b0b0b0UL, 0x54545454UL, 0xbbbbbbbbUL, 0x16161616UL,
};
```

These two tables are our Te0 and Te4 tables. Note that we have named it TE0 (uppercase), as we shall use macros (below) to access the tables.

```
#ifdef SMALL_CODE

#define Te0(x) TE0[x]
#define Te1(x) RORc(TE0[x], 8)
#define Te2(x) RORc(TE0[x], 16)
#define Te3(x) RORc(TE0[x], 24)

#define Te4_0 0x000000FF & Te4
#define Te4_1 0x0000FF00 & Te4
#define Te4_2 0x00FF0000 & Te4
#define Te4_3 0xFF000000 & Te4

#else

#define Te0(x) TE0[x]
#define Te1(x) TE1[x]
#define Te2(x) TE2[x]
#define Te3(x) TE3[x]

static const unsigned long TE1[256] = {
    0xa5c66363UL, 0x84f87c7cUL, 0x99ee7777UL, 0x8df67b7bUL,
    0x0dfff2f2UL, 0xbdd66b6bUL, 0xb1de6f6fUL, 0x5491c5c5UL,
    0x50603030UL, 0x03020101UL, 0xa9ce6767UL, 0x7d562b2bUL,
    0x19e7fefeUL, 0x62b5d7d7UL, 0xe64dababUL, 0x9aec7676UL,
<snip>
```

Here we see the definitions for our four tables. We have also split Te4 into four tables in the large code variation. This saves the logical AND operation required to extract the desired byte.

In the small code variation, we do not include TE1, TE2, or TE3, and instead use our cyclic rotation macro RORc (defined later) to emulate the tables required. We also construct the four Te4 tables by the required logical AND operation.

Decryption Tables

For decryption mode, we need a similar set of five tables, except they are the inverse.

1. Td0[x] = $S^{-1}(x)$ * [14, 9, 13, 12];
2. Td1[x] = $S^{-1}(x)$ * [12, 14, 9, 13];
3. Td2[x] = $S^{-1}(x)$ * [13, 12, 14, 9];
4. Td3[x] = $S^{-1}(x)$ * [9, 13, 12, 14];
5. Td4[x] = $S^{-1}(x)$ * [1, 1, 1, 1];

Where $S^{-1}(x)$ is InvSubBytes and the row matrices are the columns of InvMixColumns. From this, we can construct InvSubMix() using the previous technique.

```
unsigned long InvSubMix(unsigned long x)
{
   return Td0[x&255] ^
          Td1[(x>>8)&255] ^
          Td2[(x>>16)&255] ^
          Td3[(x>>24)&255];
}
```

Macros

Our AES code uses a series of portable C macros to help work with the data types. Our first two macros, STORE32H and LOAD32H, were designed to help store and load 32-bit values as an array of bytes. AES uses big endian data types, and if we simply loaded 32-bit words, we would not get the correct results on many platforms. Our third macro, RORc, performs a cyclic shift right by a specified (nonconstant) number of bits. Our fourth and last macro, byte, extracts the n'th byte out of a 32-bit word.

aes_large.c:

```
/* Helpful macros */
#define STORE32H(x, y)                                 \
     { (y)[0] = (unsigned char)(((x)>>24)&255);   \
       (y)[1] = (unsigned char)(((x)>>16)&255);   \
       (y)[2] = (unsigned char)(((x)>>8)&255);    \
       (y)[3] = (unsigned char)((x)&255); }

#define LOAD32H(x, y)                                \
     { x = ((unsigned long)((y)[0] & 255)<<24) | \
           ((unsigned long)((y)[1] & 255)<<16) | \
           ((unsigned long)((y)[2] & 255)<<8)  | \
           ((unsigned long)((y)[3] & 255)); }

#define RORc(x, y) \
(((((unsigned long)(x)&0xFFFFFFFFUL)>> \
(unsigned long)((y)&31)) | \
((unsigned long)(x)<< \
(unsigned long)(32-((y)&31)))) & 0xFFFFFFFFUL)

#define byte(x, n) (((x) >> (8 * (n))) & 255)
```

These macros are fairly common between our cryptographic functions so they are handy to place in a common header for your cryptographic source code. These macros are actually the portable macros from the LibTomCrypt package. LibTomCrypt is a bit more advanced than this, in that it can autodetect various platforms and use faster equivalent macros (loading little endian words on x86 processors, for example) where appropriate.

On the ARM (and similar) series of processors, the byte() macro is not terribly efficient. The ARM7 (our platform of choice) can perform byte loads and stores into 32-bit registers. The previous macro can be safely changed to

```
#define byte(x, n) (unsigned long)((unsigned char *)&x)[n]
```

on little endian platforms. On big endian platforms, replace [n] with [3–n].

Key Schedule

Our key schedule takes advantage of the fact that you can easily unroll the loop. We also perform all of the operations using (at least) 32-bit data types.

aes_large.c:

```
static unsigned long setup_mix(unsigned long temp)
{
   return (Te4_3[byte(temp, 2)]) |
          (Te4_2[byte(temp, 1)]) |
          (Te4_1[byte(temp, 0)]) |
          (Te4_0[byte(temp, 3)]);
}
```

This computes the SubWord() function of the key schedule. It applies the SubBytes function to the bytes of temp in parallel. The Te4_n arrays are values from the Te4 array with all but the n'th byte masked off. For example, all of the words in Te4_3 only have the top eight bits nonzero.

This function performs RotWord() as well to the input by renaming the bytes of temp. Note, for example, how the byte going into Te4_3 is actually the third byte of the input (as opposed to the fourth byte).

```

void ScheduleKey(const unsigned char *key, int keylen,
                       unsigned long *skey)
{
    int i, j;
    unsigned long temp, *rk;
```

This function differs from the eight-bit implementation in two ways. First, we pass the key length (keylen) in bytes, not 32-bit words. That is, valid values for keylen are 16, 24, and 32. The second difference is the output is stored in an array of 15*4 words instead of 15*16 bytes.

```
    /* setup the forward key */
    i                 = 0;
    rk                = skey;
    LOAD32H(rk[0], key     );
    LOAD32H(rk[1], key +  4);
    LOAD32H(rk[2], key +  8);
    LOAD32H(rk[3], key + 12);
```

We always load the first 128 bits of the key regardless of the actual key size.

```
    if (keylen == 16) {
        j = 44;
        for (;;) {
            temp  = rk[3];
            rk[4] = rk[0] ^ setup_mix(temp) ^ rcon[i];
            rk[5] = rk[1] ^ rk[4];
            rk[6] = rk[2] ^ rk[5];
            rk[7] = rk[3] ^ rk[6];
            if (++i == 10) {
               break;
            }
            rk += 4;
        }
```

This loop computes the round keys for the 128-bit key mode. It is fully unrolled to produce one round key per iteration and avoids division completely.

```
    } else if (keylen == 24) {
        j = 52;
        LOAD32H(rk[4], key + 16);
        LOAD32H(rk[5], key + 20);
        for (;;) {
            temp = rk[5];
            rk[ 6] = rk[ 0] ^ setup_mix(temp) ^ rcon[i];
            rk[ 7] = rk[ 1] ^ rk[ 6];
            rk[ 8] = rk[ 2] ^ rk[ 7];
            rk[ 9] = rk[ 3] ^ rk[ 8];
            if (++i == 8) {
                break;
            }
            rk[10] = rk[ 4] ^ rk[ 9];
            rk[11] = rk[ 5] ^ rk[10];
            rk += 6;
        }
    } else if (keylen == 32) {
        j = 60;
        LOAD32H(rk[4], key + 16);
        LOAD32H(rk[5], key + 20);
        LOAD32H(rk[6], key + 24);
        LOAD32H(rk[7], key + 28);
        for (;;) {
            temp = rk[7];
            rk[ 8] = rk[ 0] ^ setup_mix(temp) ^ rcon[i];
            rk[ 9] = rk[ 1] ^ rk[ 8];
            rk[10] = rk[ 2] ^ rk[ 9];
            rk[11] = rk[ 3] ^ rk[10];
            if (++i == 7) {
                break;
            }
            temp = rk[11];
            rk[12] = rk[ 4] ^ setup_mix(RORc(temp, 8));
            rk[13] = rk[ 5] ^ rk[12];
            rk[14] = rk[ 6] ^ rk[13];
            rk[15] = rk[ 7] ^ rk[14];
            rk += 8;
        }
    } else {
       /* this can't happen */
       return;
    }
}
```

The last two compute the 192- and 256-bit round keys, respectively. At this point, we now have our round keys required in the *skey* array. We will see later how to compute keys for decryption mode. The rest of the AES code implements the encryption mode.

TIP

The AES key schedule was actually designed to be efficient to compute in environments with limited storage. For example, if you look at the key schedule for 128-bit keys, the unrolled loop we only use rk[0...7]. Where rk[0...3] is the current round key, rk[4...7] would be the round key for the next round.

In fact, the key schedule can be computed in place; once we overwrite rk[4] for instance, we no longer need rk[0]. We can just allow rk[4] to be rk[0]. The same is true for rk[5,6,7]. This allows us to integrate the key schedule with the encryption process using only 4*4 = 16 bytes of memory instead of the default minimum of 11*4*4 = 176 bytes.

The same trick applies to the 192- and 256-bit key schedules.

```
void AesEncrypt(const unsigned char *pt,
                      unsigned char *ct,
                      unsigned long *skey, int Nr)
{
```

Again, we deviate from the eight-bit code in that we read the plaintext from the *pt* array and store the ciphertext in the *ct* array. This implementation allows pt == ct so they can overlap if the caller chooses.

```
    unsigned long s0, s1, s2, s3, t0, t1, t2, t3, *rk;
    int r;

    rk = skey;

    /*
     * map byte array block to cipher state
     * and add initial round key:
     */
    LOAD32H(s0, pt      ); s0 ^= rk[0];
    LOAD32H(s1, pt  +  4); s1 ^= rk[1];
    LOAD32H(s2, pt  +  8); s2 ^= rk[2];
    LOAD32H(s3, pt  + 12); s3 ^= rk[3];
```

Here we load the block into the array [s0, s1, s2, s3] and then apply the first AddRoundKey at the same time.

```
    /*
     * Nr - 1 full rounds:
     */
    r = Nr >> 1;
    for (;;) {
        t0 =
            Te0(byte(s0, 3)) ^
            Te1(byte(s1, 2)) ^
            Te2(byte(s2, 1)) ^
            Te3(byte(s3, 0)) ^
            rk[4];
        t1 =
            Te0(byte(s1, 3)) ^
            Te1(byte(s2, 2)) ^
            Te2(byte(s3, 1)) ^
            Te3(byte(s0, 0)) ^
            rk[5];
        t2 =
            Te0(byte(s2, 3)) ^
            Te1(byte(s3, 2)) ^
            Te2(byte(s0, 1)) ^
            Te3(byte(s1, 0)) ^
            rk[6];
        t3 =
            Te0(byte(s3, 3)) ^
            Te1(byte(s0, 2)) ^
            Te2(byte(s1, 1)) ^
            Te3(byte(s2, 0)) ^
            rk[7];
```

This is one complete AES round; we are encrypting the data in [s0,s1,s2,s3] into the set [t0,t1,t2,t3]. We can clearly see the four applications of SubBytes and MixColumns in the pattern of four Te array lookups and XORs.

The ShiftRows function is accomplished with the use of renaming. For example, the first output (for t0) is byte three of s0 (byte zero of the AES input), byte two of s1 (byte five of the AES input), and so on. The next output (for t1) is the same pattern but shifted by four columns.

This same pattern of rounds is what we will use for decryption. We will get into the same trouble as the optimized eight-bit code in that we need to modify the round keys so we can apply it after MixColumns and still achieve the correct result.

```
        rk += 8;
        if (--r == 0) {
            break;
        }
```

This "break" allows us to exit the loop when we hit the last full round. Recall that AES has 9, 11, or 13 full rounds. We execute this loop up to Nr/2–1 times, which means we have to exit in the middle of the loop, not the end.

```

        s0 =
            Te0(byte(t0, 3)) ^
            Te1(byte(t1, 2)) ^
            Te2(byte(t2, 1)) ^
            Te3(byte(t3, 0)) ^
            rk[0];
        s1 =
            Te0(byte(t1, 3)) ^
            Te1(byte(t2, 2)) ^
            Te2(byte(t3, 1)) ^
            Te3(byte(t0, 0)) ^
            rk[1];
        s2 =
            Te0(byte(t2, 3)) ^
            Te1(byte(t3, 2)) ^
            Te2(byte(t0, 1)) ^
            Te3(byte(t1, 0)) ^
            rk[2];
        s3 =
            Te0(byte(t3, 3)) ^
            Te1(byte(t0, 2)) ^
            Te2(byte(t1, 1)) ^
            Te3(byte(t2, 0)) ^
            rk[3];
```

This code handles the even rounds. We are using [t0,t1,t2,t3] as the source and feeding back into [s0,s1,s2,s3]. The reader may note that we are using rk[0,1,2,3] as the round keys. This is because we enter the loop offset by 0 but should be by 4 (the first AddRoundKey). So, the first half of the loop uses rk[4,5,6,7], and the second half uses the lower words.

A simple way to collapse this code is to only use the first loop and finish each iteration with

```
s0 = t0; s1 = t1; s2 = t2; s3 = t3;
```

This allows the implementation of the encryption mode to be roughly half the size at a slight cost in speed.

```
    }

    /*
     * apply last round and
     * map cipher state to byte array block:
     */
    s0 =
        (Te4_3[byte(t0, 3)]) ^
        (Te4_2[byte(t1, 2)]) ^
        (Te4_1[byte(t2, 1)]) ^
        (Te4_0[byte(t3, 0)]) ^
        rk[0];
    STORE32H(s0, ct);
    s1 =
        (Te4_3[byte(t1, 3)]) ^
        (Te4_2[byte(t2, 2)]) ^
        (Te4_1[byte(t3, 1)]) ^
        (Te4_0[byte(t0, 0)]) ^
        rk[1];
    STORE32H(s1, ct+4);
    s2 =
        (Te4_3[byte(t2, 3)]) ^
        (Te4_2[byte(t3, 2)]) ^
        (Te4_1[byte(t0, 1)]) ^
        (Te4_0[byte(t1, 0)]) ^
        rk[2];
    STORE32H(s2, ct+8);
    s3 =
        (Te4_3[byte(t3, 3)]) ^
        (Te4_2[byte(t0, 2)]) ^
        (Te4_1[byte(t1, 1)]) ^
        (Te4_0[byte(t2, 0)]) ^
        rk[3];
    STORE32H(s3, ct+12);
}
```

Here we are applying the last SubBytes, ShiftRows, and AddRoundKey to the block. We store the output to the *ct* array in big endian format.

Performance

This code achieves very respectable cycles per block counts with various compilers, including the GNU C and the Intel C compilers (Table 4.1).

Table 4.1 Comparisons of AES on Various Processors (GCC 4.1.1)

Processor	Cycles per block encrypted [128-bit key]
AMD Opteron	247
Intel Pentium 540J	450
Intel Pentium M	396
ARM7TDMI	3300 (Measured on a Nintendo GameBoy, which contains an ARM7TDMI processor at 16MHz. We put the AES code in the internal fast memory (IWRAM) and ran it from there.)
ARM7TDMI + Byte Modification	1780

Even though the code performs well, it is not the best. Several commercial implementations have informally claimed upward of 14 cycles per byte (224 cycles per block) on Intel Pentium 4 processors. This figure seems rather hard to achieve, as AES-128 has at least 420 opcodes in the execution path. A result of 224 cycles per block would mean an instruction per cycle count of roughly 1.9, which is especially unheard of for this processor.

x86 Performance

The AMD Opteron achieves a nice boost due to the addition of the eight new general-purpose registers. If we examine the GCC output for x86_64 and x86_32 platforms, we can see a nice difference between the two (Table 4.2).

Table 4.2 First Quarter of an AES Round

x86_64	x86_32
movq %r10, %rdx	**movl 4(%esp), %eax**
movq %rbp, %rax	**movl (%esp), %edx**
shrq $24, %rdx	**movl (%esp), %ebx**
shrq $16, %rax	shrl $16, %eax
andl $255, %edx	shrl $24, %edx
andl $255, %eax	andl $255, %eax
movq TE1(,%rax,8), %r8	movl TE1(,%eax,4), %edi
movzbq %bl,%rax	movzbl %cl,%eax
xorq TE0(,%rdx,8), %r8	xorl TE0(,%edx,4), %edi
xorq TE3(,%rax,8), %r8	xorl TE3(,%eax,4), %edi
movq %r11, %rax	**movl 8(%esp), %eax**
movzbl %ah, %edx	movzbl %ah, %edx
movq (%rdi), %rax	movl (%esi), %eax
xorq TE2(,%rdx,8), %rax	xorl TE2(,%edx,4), %eax
movq %rbp, %rdx	**movl 4(%esp), %edx**
shrq $24, %rdx	shrl $24, %edx
andl $255, %edx	xorl %eax, %edi
xorq %rax, %r8	

Both snippets accomplish (at least) the first MixColumns step of the first round in the loop. Note that the compiler has scheduled part of the second MixColumns during the first to achieve higher parallelism. Even though in Table 4.2 the x86_64 code looks longer, it executes faster, partially because it processes more of the second MixColumns in roughly the same time and makes good use of the extra registers.

From the x86_32 side, we can clearly see various spills to the stack (in bold). Each of those costs us three cycles (at a minimum) on the AMD processors (two cycles on most Intel processors). The 64-bit code was compiled to have zero stack spills during the main loop of rounds. The 32-bit code has about 15 stack spills during each round, which incurs a penalty of at least 45 cycles per round or 405 cycles over the course of the 9 full rounds.

Of course, we do not see the full penalty of 405 cycles, as more than one opcode is being executed at the same time. The penalty is also masked by parallel loads that are also on the critical path (such as loads from the *Te* tables or round key). Those delays occur anyways, so the fact that we are also loading (or storing to) the stack at the same time does not add to the cycle count.

In either case, we can improve upon the code that GCC (4.1.1 in this case) emits. In the 64-bit code, we see a pairing of "shrq $24, %rdx" and "andl $255,%edx". The *andl* operation is not required since only the lower 32 bits of %rdx are guaranteed to have anything in them. This potentially saves up to 36 cycles over the course of nine rounds (depending on how the andl operation pairs up with other opcodes).

With the 32-bit code, the double loads from *(%esp)* (lines 2 and 3) incur a needless three-cycle penalty. In the case of the AMD Athlon (and Opterons), the load store unit will short the load operation (in certain circumstances), but the load will always take at least three cycles. Changing the second load to "*movl %edx,%ebx*" means that we stall waiting for *%edx*, but the penalty is only one cycle, not three. That change alone will free up at most 9*2*4 = 72 cycles from the nine rounds.

ARM Performance

On the ARM platform, we cannot mix memory access opcodes with other operations as we can on the x86 side. The default byte() macro is actually pretty slow, at least with GCC 4.1.1 for the ARM7. To compile the round function, GCC tries to perform all quarter rounds all at once. The actual code listing is fairly long. However, with some coaxing, we can approximate a quarter round in the source.

Oddly enough, GCC is fairly smart. The first attempt commented out all but the first quarter round. GCC correctly identified that it was an endless loop and optimized the function to a simple

```
.L2: b .L2
```

which endlessly loops upon itself. The second attempt puts the following code in the loop.

```
if (--r) break;
```

Again, GCC optimized this since the source variables s0, s1, s2, and s3 are not modified. So, we simply copied t0 over them all and got the following code, which is for exactly one quarter round.

```
mov     r3, lr, lsr #16
ldr     lr, [sp, #32]
mov     r0, r0, lsr #24
ldr     r2, [lr, r0, asl #2]
ldr     r0, [sp, #36]
mov     r1, r4, lsr #8
and     r3, r3, #255
ldr     lr, [r0, r3, asl #2]
and     ip, r5, #255
and     r1, r1, #255
ldr     r0, [r8, ip, asl #2]
ldr     r3, [r7, r1, asl #2]
eor     r2, r2, lr
eor     r2, r2, r0
eor     r3, fp, r3
eor     sl, r2, r3
```

Here is an AES quarter round with the byte code optimization we mentioned earlier in the text.

```
ldrb    r1, [r5, #0]
ldrb    r2, [sp, #43]
ldrb    ip, [r9, #0]
ldr     r0, [r6, r1, asl #2]
ldr     r3, [r7, r2, asl #2]
ldrb    r2, [sp, #33]
ldr     r1, [r4, ip, asl #2]
```

```
eor     r3, r3, r0
ldr     ip, [r8, r2, asl #2]
eor     r3, r3, r1
ldr     r2, [lr, #32]
eor     r3, r3, ip
eor     r3, r3, r2
```

We can see the compiler easily uses the "load byte" *ldrb* instruction to isolate bytes of the 32-bit words to get indexes into the tables. Since this is ARM code, it can make use of the inline "*asl #2*", which multiplies the indices by four to access the table. Overall, the optimized code has three fewer opcodes per quarter round. So, why is it faster? Consider the number of memory operations per round (Table 4.3).

Table 4.3 Memory Operations with the ARM7 AES Code

	Load	Store	Total Memory Operations
Normal C Code	30	65	95
Optimized C Code	36	6	42

Even though we have placed our data in the fast 32-bit internal memory on the GameBoy (our test platform), it still takes numerous cycles to access it. According to the various nonofficial datasheets covering this platform, a load requires three cycles if the access are not sequential, and a store requires two cycles. The memory operations alone contribute 100 cycles per round above the optimized code; this accounts for 1000 cycles over the entire enciphering process.

The code could, in theory, be made faster by using each source word before moving on to the next. In our reference code, we compute whole 32-bit words of the round function at a time by reading bytes from the four other words in a row. For example, consider this quarter round

```
t0 = rk[4];
t1 = rk[5];
t2 = rk[6];
t3 = rk[7];
t1 ^= Te3(byte(s0,0));
t2 ^= Te2(byte(s0,1));
t3 ^= Te1(byte(s0,2));
t0 ^= Te0(byte(s0,3));
```

Ideally, the compiler aliases *t0*, *t1*, *t2*, and *t3* to ARM processor registers. In this approach, we are accessing the bytes of the words sequentially. The next quarter round would use the bytes of *s1* in turn. In this way, all memory accesses are sequential.

Performance of the Small Variant

Now we consider the performance using the smaller tables and a rolled up encrypt function. This code is meant to be deployed where code space is at a premium, and works particularly well with processors such as the ARM series (Table 4.4).

Table 4.4 Comparison of AES with Small and Large Code on an AMD Opteron

Mode	Cycles per block encrypted (128-bit key)
Large Code	247
Small Code	325

An interesting thing to point out is how GCC treats our rotations. With the aes_large.c code and the SMALL_CODE symbol defined, we get the following quarter round.

```
aes_large.s:
movq    %rbp, %rax
movq    %rdi, %rcx
shrq    $16, %rax
andl    $255, %eax
movq    TE0(,%rax,8), %rdx
movzbl  %ch, %eax
movq    TE0(,%rax,8), %rsi
movzbq  %bl,%rax
movq    TE0(,%rax,8), %rcx
movq    %r11, %rax
movq    %rdx, %r10
shrq    $24, %rax
salq    $24, %rdx
andl    $4294967295, %r10d
andl    $255, %eax
shrq    $8, %r10
orq     %rdx, %r10
andl    $4294967295, %r10d
xorq    TE0(,%rax,8), %r10
movq    %rcx, %rax
andl    $4294967295, %eax
salq    $8, %rcx
shrq    $24, %rax
orq     %rcx, %rax
andl    $4294967295, %eax
xorq    %rax, %r10
```

As we can see, GCC is doing a 32-bit rotation with a 64-bit data type. The same code compiled in 32-bit mode yields the following quarter round.

```
aes_large.s (32-bit):
movl    4(%esp), %eax
movl    (%esp), %ebx
shrl    $16, %eax
shrl    $24, %ebx
andl    $255, %eax
movl    TE0(,%eax,4), %esi
movzbl  %cl,%eax
movl    TE0(,%eax,4), %eax
rorl    $8, %esi
xorl    TE0(,%ebx,4), %esi
movl    %ebp, %ebx
rorl    $24, %eax
xorl    %eax, %esi
movzbl  %bh, %eax
movl    4(%esp), %ebx
movl    TE0(,%eax,4), %eax
shrl    $24, %ebx
rorl    $16, %eax
xorl    (%edx), %eax
xorl    %eax, %esi
```

Here we can clearly see that GCC picks up on the rotation and uses a nice constant "rorl" instruction in place of all the shifts, ANDs, and ORs. The solution for the 64-bit case is simple; use a 32-bit data type. At least for GCC, the symbol __x86_64__ is defined for 64-bit x86_64 mode. If we insert the following code at the top and replace all "unsigned long" references with "ulong32," we can support both 32- and 64-bit modes.

```
#if defined(__x86_64__) || (defined(__sparc__) && defined(__arch64__))
   typedef unsigned ulong32;
#else
   typedef unsigned long ulong32;
#endif
```

We have also added a common define combination for SPARC machines to spruce up the code snippet. Now with the substitutions in place, we see that GCC has a very good time optimizing our code.

aes_large_mod.s:

```
movl    %ebp, %eax
movl    %edi, %edx
movq    %rdi, %rcx
shrl    $16, %eax
shrl    $24, %edx
movzbl  %al, %eax
movzbl  %dl, %edx
movq    TE0(,%rax,8), %rax
movl    %eax, %r8d
movzbl  %bl, %eax
rorl    $8, %r8d
movq    TE0(,%rax,8), %rax
xorl    TE0(,%rdx,8), %r8d
movq    %r11, %rdx
rorl    $24, %eax
xorl    %eax, %r8d
movzbl  %dh, %eax
movl    %ebp, %edx
movq    TE0(,%rax,8), %rax
```

```
shrl    $24, %edx
movzbl  %dl, %edx
rorl    $16, %eax
xorl    (%r10), %eax
xorl    %eax, %r8d
```

Notes from the Underground…

Know Your Data Types

We saw in the aes_large.c example that using the wrong data type can lead to code that performs poorly. So, how do you know when you fall into this trap?

For starters, it is good to know how your platform compares against the C reference. For example, "unsigned long" is at *least* 32-bits long. It does not have to be that small, and indeed, on many 64-bit platforms it is actually 64-bits long.

The second way to know your code is using the wrong type is to examine the assembler output. By invoking GCC (for example) with the –S option, the compiler will emit assembler that you can audit and examine for performance traps. A surefire sign that you are using the wrong type is if you are getting calls to internal helper routines (working with 64-bit values on a 32-bit target).

However, internal functions are not always a catch-all, as GCC is smart enough to inline many simple operations, such as 64-bit additions on a 32-bit host. In our case, we noted that GCC was using shifts, ORs, and ANDs to accomplish what it can (and knows how to) with a single x86 opcode.

Inverse Key Schedule

So far, we have only considered the forward mode. The inverse cipher looks exactly the dame, at least for this style of implementation. The key difference is that we substitute the forward tables with the inverse. This leaves us with the key schedule.

In the standard AES reference code, the key schedule does not change for the inverse cipher, since we must apply AddRoundKey before InvMixColumns. However, in this style of implementation, we are performing the majority of the round function in a single short sequence of operations. We cannot insert AddRoundKey in between so we must place it afterward.

The solution is to perform two steps to the forward round keys.

1. Reverse the array of round keys.
    1. Group the round keys into 128-bit words.
    2. Reverse the list of 128-bit words.

3. Ungroup the round keys back into 128-bit words.

2. Apply InvMixColumns to all but the first and last round keys.

To reverse the keys, we do not actually form 128-bit words; instead, we do the swaps logically with 32-bit words. The following C code will reverse the keys in *rk* to *drk*.

```
rk += 10*4; /* last 128-bit round key for AES-128 */
for (x = 0; x < 11; x++) {
   drk[0] = rk[0];
   drk[1] = rk[1];
   drk[2] = rk[2];
   drk[3] = rk[3];
   rk -= 4; drk += 4;
}
```

Now we have the keys in the opposite order in the *drk* array. Next, we have to apply InvMixColumns. At this point, we do not have a cheap way to implement that function. Our tables *Td0*, *Td1*, *Td2*, and *Td3* actually implement InvSubBytes and InvMixColumns. However, we do have a SubBytes table handy (Te4). If we first pass the bytes of the key through Te4, then through the optimized inverse routine, we end up with

drk[x] := InvMixColumns(InvSubWord(SubWord(drk[x]))
drk[x] := InvMixColumns(drk[x])

which can be implemented in the following manner.

```
for (x = 4; x < 10*4; x++) {
    drk[x] = Td0(255 & Te4[byte(drk[x], 3)]) ^
             Td1(255 & Te4[byte(drk[x], 2)]) ^
             Td2(255 & Te4[byte(drk[x], 1)]) ^
             Td3(255 & Te4[byte(drk[x], 0)]);
}
```

Now we have the proper inverse key schedule for AES-128. Substitute "10*4" by "12*4" or "14*4" for 192- or 256-bit keys, respectively.

Practical Attacks

As of this writing, there are no known breaks against the math of the AES block cipher. That is, given no other piece of information other than the traditional plaintext and ciphertext mappings, there is no known way of determining the key faster than brute force.

However, that does not mean AES is perfect. Our "classic" 32-bit implementation, while very fast and efficient, leaks considerable side channel data. Two independent attacks developed by Bernstein and Osvik (et al.) exploit this implementation in what is known as a side channel attack.

Side Channels

To understand the attacks we need to understand what a side channel is. When we run the AES cipher, we produce an output called the ciphertext (or plaintext depending on the direction). However, the implementation also produces other measurable pieces of information. The execution of the cipher does not take fixed time or consume a fixed amount of energy.

In the information theoretic sense, the fact that it does not take constant time means that the implementation leaks information (entropy) about the internal state of the algorithm (and the device it is running on). If the attacker can correlate runtimes with the knowledge of the implementation, he can, at least in theory, extract information about the key.

So why would our implementation not have a constant execution time? There are two predominantly exploitable weaknesses of the typical processor cache.

Processor Caches

A processor cache is where a processor stores recently written or read values instead of relying on main system memory. Caches are designed in all sorts of shapes and sizes, but have several classic characteristics that make them easy to exploit. Caches typically have a low set associativity, and make use of bank selectors.

Associative Caches

Inside a typical processor cache, a given physical (or logical depending on the design) address has to map to a location within the cache. They typically work with units of memory known as *cache lines*, which range in size from small 16-byte lines to more typical 64- and even 128-byte lines. If two source addresses (or cache lines) map to the same cache address, one of them has to be *evicted* from the cache. The eviction means that the lost source address must be fetched from memory the next time it is used.

In a *fully associated* cache (also known as a completely associated memory or CAM), a source address can map anywhere inside the cache. This yields a high cache hit rate as evictions occur less frequently. This type of cache is expensive (in terms of die space) and slower to implement. Raising the latency of a cache hit is usually not worth the minor savings in the cache miss penalties you would have otherwise.

In a *set-associative* cache, a given source address can map to one of *N* unique locations; this is also known as a *N*-way associative cache. These caches are cheaper to implement, as you only have to compare along the *N* ways of the cache for a cache line to evict. This lowers the latency of using the cache, but makes a cache eviction more likely.

The cheapest way to compute a cache address in the set-associative model is to use consecutive bits of the address as a cache address. These are typically taken from the middle of the address, as the lower bits are used for index and bank selection. These details depend highly on the configuration of the cache and the architecture in general.

Cache Organization

The organization of the cache affects how efficiently you can access it. We will consider the AMD Opteron cache design (AMD Software Optimization Guide for AMD64 Processors, #25112, Rev 3.06, September 2005) for this section, but much of the discussion applies to other processors.

The AMD Opteron splits addresses into several portions:

1. Index is addr[14:6] of the address
2. Bank is addr[5:3] of the address
3. Byte is addr[2:0] of the address

These values come from the L1 cache design, which is a 64-kilobyte two-way set-associative cache. The cache is actually organized as two linear arrays of 32 kilobytes in two ways (hence 64 kilobytes in total). The *index* value selects which 64 byte cache line to use, *bank* selects which eight-byte group of the cache line, and *byte* indicates the starting location of the read inside the eight-byte group.

The cache is dual ported, which means two reads can be performed per cycle; that is, unless a bank conflict occurs. The processor moves data in the cache on parallel busses. This means that all bank 0 transactions occur on one bus, bank 1 transactions on another, and so on. A conflict occurs when two reads are from the same bank but different index. What that means is you are reading from the same bank offset of two different cache lines. For example, using AT&T syntax, the following code would have a bank conflict.

```
movl (%eax),%ebx
movl 64(%eax),%ecx
```

Assuming that both addresses are in the L1 cache and %eax is aligned on a four-byte boundary, this code will exhibit a bank conflict penalty. Effectively, we want to avoid reads that a proper multiple of 64 offsets, which, as we will see, is rather unfortunate for our implementation.

Bernstein Attack

Bernstein was the first to describe a complete cache attack against AES[3] (although it is seemingly bizarre, it does in fact work) using knowledge of the implementation and the platform it was running on. (He later wrote a follow-up paper at http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.) His attack works as follows.

1. Pick one of the 16 plaintext bytes, call that pt[n].
2. Go through all 256 possible values of pt[n] and see which takes the longest to encrypt.

This attack relies on the fact that pt[n] XOR key[n] will be going into the first round tables directly. You first run the attack with a known key. You deduce a value T = pt[n] XOR key[n], which occurs the most often. That is, you will see other values of T, but one value will eventually emerge as the most common. Now, you run the attack against the victims by having them repeatedly encrypt plaintexts of your choosing. You will end up with a value of pt[n] for which the delay is the longest and then can deduce that pt[n] XOR T = key[n].

The attack seems like it would not work and indeed it is not without its flaws. However, over sufficiently long enough runs it will yield a stable value of T for almost all key bytes. So why does the attack work? There are two reasons, which can both be true depending on the circumstances.

- A cache line was evicted from the cache, causing a load from L2 or system memory.
- A(n additional) bank conflict occurred in the first round.

In the first case, one of our tables had a cache line evicted due to another process competing for the cache line. This can arise due to another task (as in the case of the Osvik attack) or kernel process interrupting the encryption. Most processors (including the Pentium 4 with its small 8-16KB L1 cache) can fit the entire four kilobytes of tables in the L1 cache. Therefore, back-to-back executions should have few cache line evictions if the OS (or the calling process) does not evict them.

The more likely reason the attack succeeds is due to a bank conflict, changing the value of pt[n] until it conflicts with another lookup in the same quarter round of AES. All of the tables are 1024 bytes apart, which is a proper multiple of 64. This means that, if you access the same 32-bit word (or its neighbor) as another access, you will incur the stall.

Bernstein proposed several solutions to the problem, but the most practical is to serialize all loads. On the AMD processors are three integer pipelines, each of which can issue a load or store per cycle (a maximum of two will be processed, though). The fix is then to bundle instructions in groups of three and ensure that the loads and stores all occur in the same location within the bundles. This fix also requires that instructions do not cross the 16-byte boundary on which the processor fetches instructions. Padding with various lengths of no operation (NOP) opcodes can ensure the alignment is maintained.

This fix is the most difficult to apply, as it requires the implementer to dive down in to assembler. It is also not portable, even across the family of x86 processors. It also is not even an issue on certain processors that are single scalar (such as the ARM series).

Osvik Attack

The Osvik (Dag Arne Osvik, Adi Shamir, and Eran Tromer: *Cache Attacks and Countermeasures: the Case of AES*) attack is a much more active attack than the Bernstein attack. Where in the Bernstein attack you simply try to observe encryption time, in this attack you are actively trying to evict things out of the cache to influence timing.

Their attack is similar to the attack of Colin Percival (Colin Percival: *Cache Missing For Fun and Profit*) where a second process on the victim machine is actively populating specific cache lines in the hopes of influencing the encryption routine. The goal is to know which lines to evict and how to correlate them to key bits. In their attack, they can read 47 bits of key material in one minute.

Countering this attack is a bit harder than the Bernstein attack. Forcing serialized loads is not enough, since that will not stop cache misses. In their paper, they suggest a variety of changes to mitigate the attack; that is, not to defeat the attack but make it impractical.

Defeating Side Channels

Whether these side channels are even a threat depends solely on your application's threat model and use cases. It is unwise to make a blanket statement we must defend against all side channels at all times. First, it's impractically nonportable to do so. Second, it's costly to defend against things that are not threats, and often you will not get the time to do so. Remember, you have customers to provide a product to.

The attacks are usually not that practical. For example, Bernstein's attack, while effective in theory, is hard to use in practice, as it requires the ability to perform millions of chosen queries to an encryption routine, and a low latency channel to observe the encryptions on. As we will see with authenticated channels, requesting encryptions from an unauthorized party can easily be avoided. We simply force the requestors to authenticate their requests, and if the authentication fails, we terminate the session (and possibly block the attackers from further communication). The attack also requires a low latency channel so the timings observed are nicely correlated. This is possible in a lab setting where the two machines (attacker and victim) are on the same backplane. It is completely another story if they are across the Internet through a dozen hops going from one transmission medium to another.

The Osvik attack requires an attacker to have the ability to run processes locally on the victim machine. The simplest solution if you are running a server is not to allow users to have a shell on the machine.

NOTE

The significance of these attacks has stirred debate in various cryptographic circles. While nobody debates whether the attacks work, many question how effective the attacks are in realistic work situations. Bernstein's attack requires an application to process millions of un-authorized encryption requests. Osvik's attack requires an attacker to load applications on the victim machine.

The safest thing to do with this information is *be aware* of it but not especially *afraid* of it.

Little Help from the Kernel

Suppose you have weeded out the erroneous threat elements and still think a cache smashing attack is a problem. There is still a feasible solution without designing new processors or other impractical solutions. Currently, no project provides this solution, but there is no real reason why it would not work.

Start with a kernel driver (module, device driver, etc.) that implements a serialized AES that preloads the tables into L1 before processing data. That is, all accesses to the tables are forced into a given pipeline (or load store unit pipeline). This is highly specific to the line of processor and will vary from one architecture to another.

Next, implement a double buffering scheme where you load the text to be processed into a buffer that is cacheable and will not compete with the AES tables. For example, on the AMD processors all you have to do is ensure that your buffer does not have an overlap with the tables modulo 32768. For example, the tables take 5 kilobytes, leaving us with 27 kilobytes to store data.

Now to process a block of data, the kernel is passed the data to encrypt (or decrypt), and locks the machine (stopping all other processors). Next, the processor crams as much as possible into the buffer (looping as required to fulfill the request), preloads it into the L1, and then proceeds to process the data with AES (in a chaining mode as appropriate).

This approach would defeat both Bernstein's and Osvik's attacks, as there are no bank conflicts and there is no way to push lines out of the L1. Clearly, this solution would lead to a host of denial of service (DoS) problems, as an attacker could request very large encryptions to lock up the machine. A solution to that problem would be to either require certain users' identifiers to use the device or restrict the maximum size of data allowed. A value as small as 4 to 16 kilobytes would be sufficient to make the device practical.

Chaining Modes

Ciphers on their own are fairly useless as privacy primitives. Encrypting data directly with the cipher is a mode known as Electronic Codebook (ECB). This mode fails to achieve privacy, as it leaks information about the plaintext. The goal of a good chaining mode is therefore to provide more privacy than ECB mode. One thing a good chaining mode is *not* for is authenticity. This is a point that cannot be stressed enough and will be repeated later in the text.

Repeatedly, we see people write security software where they implement the privacy primitive but fail to ensure the authenticity—either due to ignorance or incompetence. Worse, people use modes such as CBC and assume it provides authenticity as well.

ECB mode fails to achieve privacy for the simple fact that it leaks information if the same plaintext block is ever encrypted more than once. This can occur within the same session (e.g., a file) and across different sessions. For example, if a server responds to a query with a limited subset of responses all encrypted in ECB mode, the attacker has a very small mapping to decode to learn the responses given out.

To see this, consider a credit-card authorization message. At its most basic level, the response is either success or failure; 0 or 1. If we encrypted these status messages with ECB, it would not take an attacker many transactions to work out what the code word for success or failure was.

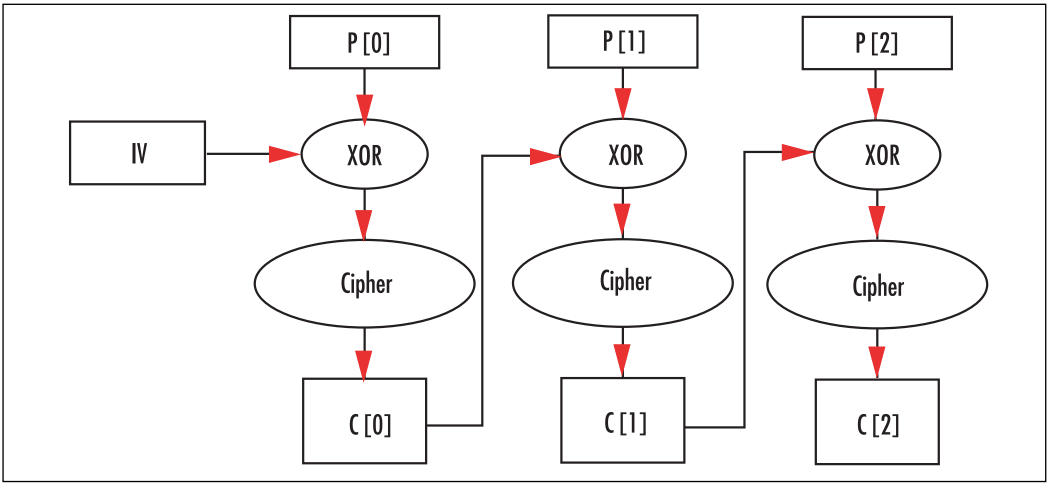
Back in the days of DES, there were many popular chaining modes, from Ciphertext Feedback (CFB), Output Feedback (OFB), and Cipher Block Chaining (CBC). Today, the most common modes are cipher block chaining and counter mode (CTR). They are part of the NIST SP 800-38A standard (and OFB and CFB modes) and are used in other protocols (CTR mode is used in GCM and CCM, CBC mode is used in CCM and CMAC).

Cipher Block Chaining

Cipher block chaining (CBC) mode is most common legacy encryption mode. It is simple to understand and trivial to implement around an existing ECB mode cipher implementation. It is often mistakenly attributed with providing authenticity for the reason that "a change in the ciphertext will make a nontrivial change in the plaintext." It is true that changing a single bit of ciphertext will alter two blocks of plaintext in a nontrivial fashion. It is *not* true that this provides authenticity.

We can see in Figure 4.12 that we encrypt the plaintext blocks P[0,1,2] by first XORing a value against the block and then encrypting it. For blocks P[1,2], we use the previous ciphertext C[0,1] as the chaining value. How do we deal with the first block then, as there is no previous ciphertext? To solve this problem, we use an initial value (IV) to serve as the chaining value. The message is then transmitted as the ciphertext blocks C[0,1,2] and the IV.

Figure 4.12 Cipher Block Chaining Mode



What's in an IV?

An initial value, at least for CBC mode, must be *randomly chosen* but it need not be secret. It must be the same length of the cipher block size (e.g., 16 bytes for AES).

There are two common ways of dealing with the storage (or transmission) of the IV. The simplest method is to generate an IV at random and store it with the message. This increases the size of the message by a single block.

The other common way of dealing with the IV is to make it during the key negotiation process. Using an algorithm known as PKCS #5 (see Chapter 5), we can see that from a randomly generated shared secret we can derive both encryption keys and chaining IVs.

Message Lengths

The diagram (Figure 4.12) suggests that all messages have to be a proper multiple of the cipher block size. This is not true. There are two commonly suggested (both equally valid) solutions. FIPS 81 (which is the guiding doctrine for SP 800-38A) does not mandate a solution, unfortunately.

The first common solution is to use a technique known as *ciphertext stealing*. In this approach, you pass the last ciphertext block through the cipher in ECB mode and XOR the output of that against the remaining message bytes (or bits). This avoids lengthening the message (beyond adding the IV).

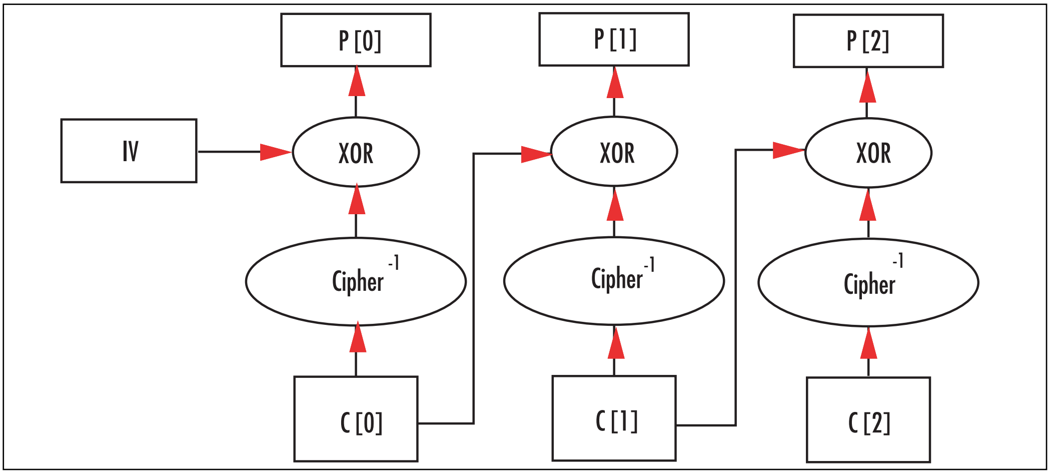
The second solution is to pad the last block with enough zero bits to make the message length a proper multiple of the cipher block size. This increases the message size and really has no benefits over ciphertext stealing.

In practice, no standard from NIST mandates one solution over the other. The best practice is to default to ciphertext stealing, but document the fact you chose that mode, unless you have another standard to which you are trying to adhere.

Decryption

To decrypt in CBC mode, pass the ciphertext through the inverse cipher and then XOR the chaining value against the output (Figure 4.13).

Figure 4.13 Cipher Block Chaining Decryption



Performance Downsides

CBC mode suffers from a serialization problem during encryption. The ciphertext C[n] cannot be computed until C[n–1] is known, which prevents the cipher from being invoked in parallel. Decryption can be performed in parallel, as the ciphertexts are all known at that point.

Implementation

Using our fast 32-bit AES as a model, we can implement CBC mode as follows.

cbc.c:

```
009   void AesCBCEncrypt(const unsigned char *IV,
010                            unsigned char *pt,
011                            unsigned char *ct,
012                            unsigned long  size,
013                            unsigned long *skey, int Nr)
014   {
015      unsigned char buf[16];
016      unsigned long x;
017
018      for (x = 0; x < 16; x++) buf[x] = IV[x];
019      while (size--) {
020         /* create XOR of pt and chaining value */
021         for (x = 0; x < 16; x++) buf[x] ^= pt[x];
022
023         /* encrypt it */
024         AesEncrypt(buf, buf, skey, Nr);
025
026         /* copy it out */
027         for (x = 0; x < 16; x++) ct[x] = buf[x];
028
029         /* advance */
030         pt += 16; ct += 16;
031      }
032   }
```

This code performs the CBC updates on a byte basis. In practice, it's faster and mostly portable to perform the XORs as whole words at once. On x86 processors, it's possible to use 32- or 64-bit words to perform the XORs (line 21). This dramatically improves the performance of the CBC code and reduces the overhead over the raw ECB mode.

Decryption is a bit trickier if we allow *pt* to equal *ct*.

cbc.c:

```
034   void AesCBCDecrypt(const unsigned char *IV,
035                            unsigned char *ct,
036                            unsigned char *pt,
037                            unsigned long  size,
038                            unsigned long *skey, int Nr)
039   {
040      unsigned char buf[16], buf2[16], t;
041      unsigned long x;
042
043      for (x = 0; x < 16; x++) buf[x] = IV[x];
044      while (size--) {
045         /* decrypt it */
046         AesDecrypt(ct, buf2, skey, Nr);
047
048         /* copy current ct, create pt and then update buf */
049         for (x = 0; x < 16; x++) {
050            t       = ct[x];
051            pt[x]   = buf2[x] ^ buf[x];
052            buf[x]  = t;
053         }
054
055         /* advance */
056         pt += 16; ct += 16;
057      }
058   }
```

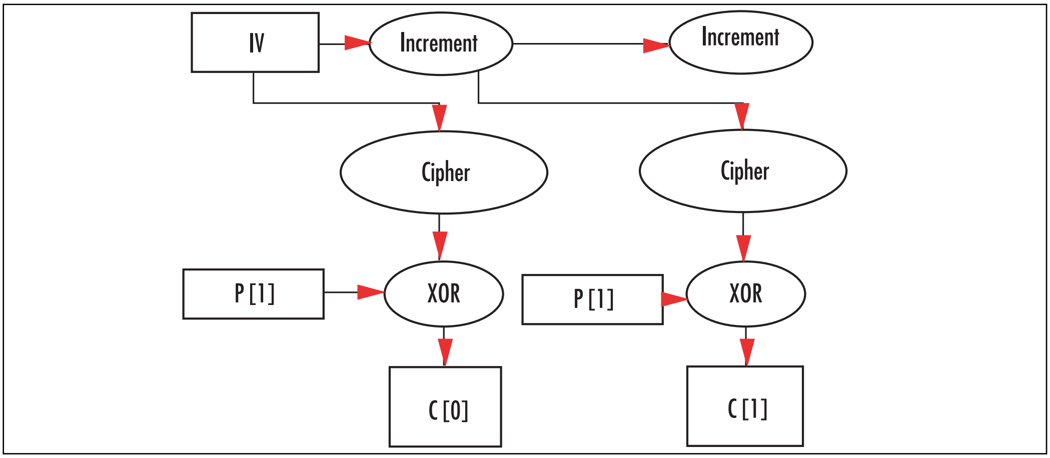
Here we again use *buf* as the chaining buffer. We decrypt through AES ECB mode to *buf2*, since at this point we cannot write to *pt* directly. Inside the inner loop (lines 49 to 53), we fetch a byte of *ct* before we overwrite the same position in *pt*. This allows the buffers to overlap without losing the previous ciphertext block required to decrypt in CBC mode.

Counter Mode

Counter mode (CTR) is designed to extract the maximum efficiency of the cipher to achieve privacy. In this mode, we make use of the fact that the cipher should be a good pseudo random permutation (PRP). Instead of encrypting the message by passing it through the cipher, we encrypt it as we would with a stream cipher.

In CTR mode, we have an IV as we would for CBC, except it does not have to be random, merely unique. The IV is encrypted and XORed against the first plaintext block to produce the first ciphertext. The IV is then incremented (as if it were one large integer), and then the process is repeated for the next plaintext block (Figure 4.14).

Figure 4.14 Counter Cipher Mode



The "increment" function as specified by SP800-38A is a simple binary addition of the value "1." The standard does not specify if the string is big or little endian. Experience shows that most standards assume it is big endian (that is, you increment byte 15 of the chaining value and the carries propagate toward byte 0).

The standard SP800-38A allows permutation functions other than the increment. For example, in hardware, a LFSR stepping (see Chapter 3) would be faster than an addition since there are no carries. The standard is clear, though, that all chaining values used with the same secret key *must* be unique.

Message Lengths

In the case of CTR, we are merely XORing the output of the cipher against the plaintext. This means there is no inherit reason why the plaintext has to be a multiple of the cipher block size. CTR mode is just as good for one-bit messages as it is terabit messages.

CTR mode requires that the IV be transmitted along with the ciphertext—so there is still that initial message expansion to deal with. In the same way that CBC IVs can be derived from key material, the same is true for CTR mode.

Decryption

In CTR mode, encryption and decryption are the same process. This makes implementations simpler, as there is less code to deal with. Also, unlike CBC mode, only the forward direction of the cipher is required. An implementation can further save space by not including the ECB decryption mode support.

Performance

CTR mode allows for parallel encryption or decryption. This is less important in software designs, but more so in hardware where you can easily have one, two, or many cipher "cores" creating a linear improvement in throughput.

Security

CTR mode strongly depends on the chaining values being unique. This is because if you re-use a chaining value, the XOR of two ciphertext blocks will be the XOR of the plaintext blocks. In many circumstances, for example, when encrypting English ASCII text, this is sufficient to learn the contents of both blocks. (You would expect on average that 32 bytes of English ASCII have 41.6 bits of entropy, which is less than the size of a single plaintext block.)

The simplest solution to this problem, other than generating IVs at random, is never to reuse the same key. This can be accomplished with fairly modest key generation protocols (see Chapter 5).

Implementation

Again, we have used the 32-bit fast AES code to construct another chaining mode; this time, the CTR chaining mode.

ctr.c:

```
006   void AesCTRMode(unsigned char *IV,
007                   unsigned char *in,
008                   unsigned char *out,
009                   unsigned long  size,
010                   unsigned long *skey, int Nr);
011   {
012      unsigned char buf[16];
013      unsigned long x, y;
014      int           z;
015
016      while (size) {
017         /* encrypt counter */
018         AesEncrypt(IV, buf, skey, Nr);
019
020         /* increment it */
021         for (z = 15; z >= 0; z--) {
022            if (++IV[z] & 255) break;
023         }
024
025         /* process input */
026         y = (size > 16) ? 16 : size;
027         for (x = 0; x < y; x++) {
028            *in++ = *out++ ^ buf[x];
029         }
030         size -= y;
031      }
032   }
```

In this implementation, we are again working on the byte level to demonstrate the mode. The XORs on line 28 could more efficiently be accomplished with a set of XORs with larger data types. The increment (lines 21 through 23) performs a big endian increment by one. The increment works because a carry will only occur if the value of (*++IV[z] & 255*) is zero. Therefore, if it is nonzero, there is no carry to propagate and the loop should terminate.

Here we also deviate from CBC mode in that we do not have specific plaintext and ciphertext variables. This function can be used to both encrypt and decrypt. We also update the IV value, allowing the function to be called multiple times with the same key as need be.

Choosing a Chaining Mode

Choosing between CBC and CTR mode is fairly simple. They are both relatively the same speed in most software implementations. CTR is more flexible in that it can natively support any length plaintext without padding or ciphertext stealing. CBC is a bit more established in existing standards, on the other hand.

A simple rule of thumb is, unless you have to use CBC mode, choose CTR instead.

Putting It All Together

The first thing to keep in mind when using ciphers, at least in either CTR or CBC mode, is to ignore the concept of authenticity. Neither mode will grant you any guarantee. A common misconception is that CBC is safer in this regard. This is *not* true. What you will want to use CBC or CTR mode for is establishing a private communication medium.

Another important lesson is that you almost certainly do need authenticity. This means you will be using a MAC algorithm (see Chapter 6) along with your cipher in CBC or CTR mode. Authentication is vital to protect your assets, and to protect against a variety of random oracle attacks (Bernstein and Osviks attacks). The only time authenticity is not an issue is if there is no point at which an attacker can alter the message. That said, it is a nice safety net to ensure authenticity for the user and in the end it usually is not that costly.

Now that we have that important lesson out of the way, we can get to deploying our block cipher and chaining mode(s). The first thing we need to be able to do is key the cipher, and to do so securely. Where and how we derive the cipher key is a big question that depends on how the application is to be used. The next thing we have to be able to do is generate a suitable IV for the chaining mode, a step that is often tied to the key generation step.

Keying Your Cipher

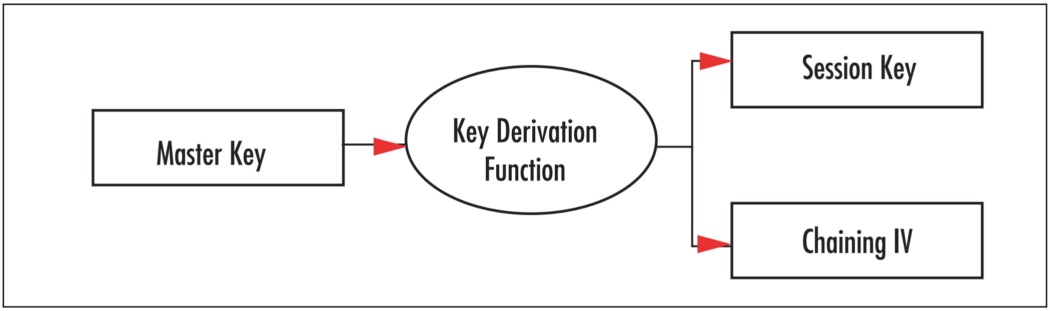
It is very important that the cipher be keyed with as much entropy as possible. There is little point of using AES if your key has 10 bits of entropy. Brute force would be trivial and you would be providing no privacy for your users. Always use the maximum size your application can tolerate. AES mandates you use at least a 128-bit key. If your application can tolerate the performance hit, you are better off choosing a larger key. This ensures any bias in your PRNG (or RNG) will still allow for a higher amount of entropy in the cipher key.

Forget about embedding the key in your application. It just will not work.

The most common way of keying a cipher is from what is known as a *master key*, but is also known as a *shared secret*. For example, a salted password hash (see Chapter 4) can provide a master key, and a public key encrypted random key (see Chapter 9) can provide a shared secret.

Once we have this master key, we can use a *key derivation algorithm* such as PKCS #5 (see Chapter 4) to derive a key for our cipher and an IV. Since the master key should be random for every session (that is, each time a new message is to be processed), the derived cipher key and IV should be random as well (Figure 4.15).

Figure 4.15 Key Derivation Function



It is important that you do not derive your cipher key (session key) from something as trivial as a password directly. Some form of key derivation should always be in your cryptosystems pipeline.

ReKeying Your Cipher

Just as important as safely keying your cipher is the process of rekeying it. It is generally not a safe practice to use the same cipher key for a significant length of time. Formally, the limitations fall along the birthday paradox boundaries (2^{64} blocks for AES). However, the value of all the secrets you can encrypt in that time usually outweighs the cost of supporting secure rekeying.

Rekeying can take various forms depending on if the protocol is online or offline. If the protocol is online, the rekeying should be triggered both on a timed and traffic basis; that is, after an elapsed time and after an elapsed traffic volume count has been observed. All signals to rekey should be properly authenticated just like any other message in the system. If the system is offline, the process is much simpler, as only one party has to agree to change keys. Reasonable guidelines depend on the product and the threat vectors, especially since the threat comes not from the cipher being broken but the key being determined through other means such as side channel attacks or protocol breaks.

Once the decision to rekey has been accepted (or just taken), it is important to properly rekey the cipher. The goal of rekeying is to mitigate the damages that would arise if an attacker learned the cipher key. This means that the new key cannot be derived from the previous cipher key. However, if you are clever and derived your cipher key from a master key, your world has become much easier.

At this point, an attacker would have at most a short amount of key derivation output, very unlikely to be enough to break the key derivation function. Using the key derivation function to generate a new cipher key at given intervals solves our problem nicely.

Bi-Directional Channels

A useful trick when initiating bi-directional channels is to have two key and IV pairs generated (ideally from a suitable key derivation function). Both parties would generate the same pair of keys, but use them in the opposite order. This has both cryptographic and practical benefits to it.

On the practical side, this allows both parties to send at once and to keep the IVs synchronized. It also makes dealing with lost packets (say from a UDP connection) easier.

On the cryptographic side, this reduces the traffic used under a given symmetric key. Assuming both sides transmit roughly an equal amount of traffic, the attacker now has half the ciphertext under a given key to look at.

Lossy Channels

On certain channels of communication such as UDP, packets can be lost, or worse, arrive out of order. First, let's examine how to deal with packet data.

In the ideal protocol, especially when using UDP, the protocol should be able to cope with a minor amount of traffic loss, either by just ignoring the missing data or requesting re-transmissions. In cases where latency is an issue, UDP is usually the protocol of choice for transmitting data. Don't be mistaken that TCP is a secure transport protocol. Even though it makes use of checksums and counters, packets can easily be modified by an attacker. All data received should be suspect. To cope with loss, each packet must be independent. This means that each packet should have its own IV attached to it.

If we are using CBC mode, this means that each packet needs a fresh random IV. Since the IV is random, we will have to also attach a counter to it. The counter would be unique and incremented for every packet. It allows the receiver to know if it has seen the packet before, and if not, where in the stream of packets it belongs. At the very least, this implies a packet overhead of at least 20 bytes (for AES) per packet; that is, 16 bytes for the IV and 4 bytes (say) for the counter.

If we are using CTR mode, each packet needs a *unique* IV. There is no reason why the IV itself cannot be the packet counter. There is also no reason why the IV needs to be large if the key was chosen at random. For example, if we know we are going to send less than 2^{32} packets, we could use a 4 byte counter, leaving the lower 12 bytes of the counter as implicitly zero (to be incremented during the encryption of the packet). This would add an overhead of only four bytes per packet. For more details on streaming protocols, see Chapter 6.

Now that we have counters on all of our packets, we can do two things with them. We can reject old or replayed packets, and we can sort out-of-order packets. There are two logical ways of dealing with out-of-order packets. The first is to just bump the counter up to the highest valid packet (disregarding any packets with lower counter values). This is the simplest solution and in many applications totally feasible. (Despite the bad reputation, UDP packets rarely arrive out of order.) So, in the rare event that it does occur, tolerating the error should be minor. The more difficult solution is to maintain a sliding window. The window would count upward from a given base, sliding the base as the situation arises.

For example, consider the following code:

```
static unsigned long window, window_base
int isValidCtr(unsigned long ctr)
{
   /* below the window? */
   if (ctr < window_base) return 0;

   /* out of the window? */
   if (ctr - window_base > 31) { window_base = ctr; window = 0; return 1; }

   /* already seen? */
   if (window & (1UL << (ctr - window_base))) { return 0; }

   /* not seen */
   window |= 1UL << (ctr – window_base);

   /* shift window */
   while (window & 1) { window >>= 1; ++window_base; }
   return 1;
}
```

The preceding code allows the caller to keep track up to 32 numbered packets at once. It rejects old and previously seen packets, adjusts to bumps in the counter, and slides the window when possible. The caller would pass this function a counter from an otherwise valid function (a packet that has been authenticated; see Chapter 6), and it would return 0 if the counter is valid or 1 if not.

Myths

Here are some popular myths about block ciphers.

- CBC provides authenticity (or integrity).
- CBC requires only a unique IV, not a random one.
- CTR mode is not secure because the plaintext does not pass through the cipher itself.
- You can use data hidden in the application as a cipher key.
- Modifying the algorithm (to make the details obfuscated) makes it more secure.
- Using the largest key size is "more secure."
- Encrypting multiple times is "safer."

CBC mode does not provide authenticity (sorry for the repetition, but this is a point many people get it wrong). If you want authenticity (and you usually do), apply a MAC to the message (or ciphertext) as well. CBC mode also requires unpredictable IVs, not merely unique ones.

CTR mode provably reduces the security of the block cipher. That is, if CTR mode is insecure so, are the CBC, OFB, and CFB modes.

Embedding a cipher key in the application is a trick people seem to like. Diebold was one of these types; except the source code was leaked and the key was found (and put on display, it was "F2654hD4" btw). Do not do this.

Modifying the cipher to achieve some proprietary tweak is another faux pas. It frustrates users by disallowing interoperability, can make the algorithm less secure, and in the end someone is going to reverse engineer it anyway.

Using the largest key size is not always "more secure" or better practice. It is a good idea, at least in theory. For example, if your RNG has a slight bias, then a 256-bit string it generates will have more entropy (at least on average) than a 128-bit string. However, 256-bit keys are slower in AES (14 rounds versus 10 rounds), and often RNGs or PRNGs are working as desired. Brute forcing a 128-bit key remains out of the realm of possibility for the foreseeable future.

The key reason why bigger is not always better is because it is easier to find attacks faster than brute-force for ciphers with larger keys. Consider using AES with a 64-bit key. To break the algorithm, you have to find an attack that works faster than 2^{64} encryptions. This is a much harder proposition than breaking the full-length key. The key length of the cipher sends an expectation of how much security the designer claims the algorithm can offer.

Encrypting multiple times, possibly with different ciphers, does not create a safer mix; it just makes a slower design. The same logic that says "different ciphers applied multiple times makes it more secure" can also say, "The specific combination of these ciphers makes it insecure." There is no reason to believe one argument over the other, and in the end you just create more proprietary junk.

Again, this is a easy to see. A block cipher is just shorthand notation for a huge substitution table. Given any particular substitution, it is always possible to find another complement substitution such that when applied one after another together will result in a linear transform. This construction is trivially breakable.

Providers

There are many readily available providers of AES encryption and decryption. In general, unless your application has very specific needs, you are better served (and serving your users) by not dwelling on the implementation. LibTomCrypt and OpenSSL are common providers in C callable applications. We shall use the former to create a simple CTR mode example.

The reader is encouraged to read the user manual that comes with LibTomCrypt to fully appreciate what the library has to offer. What follows is a simple program that will encrypt and decrypt a short string with AES in CTR mode.

ctr_example.c:

```
#include <tomcrypt.h>

int main(void)
{
    symmetric_CTR ctr;
    unsigned char secretkey[16], IV[16], plaintext[32],
                  ciphertext[32], buf[32];
    int           x;

    /* setup LibTomCrypt */
    register_cipher(&aes_desc);
```

This statement tells LibTomCrypt to register the AES plug-in with its internal tables. LibTomCrypt uses a modular plug-in system to allow the developer to substitute one implementation with another (say for added hardware support). In this case, we are using the built-in AES software implementation.

```
    /* somehow fill secretkey and IV ... */

```

Obviously, we left this section blank. The key and IV can be derived in many ways, most of which we haven't shown you how to use yet. Chapter 5 shows how to use PKCS #5 as a key derivation function, and Chapter 9 shows how to use a public key algorithm to distribute a shared secret.

```
    /* start CTR mode */
    assert(
      ctr_start(find_cipher("aes"), IV, secretkey, 16, 0,
                CTR_COUNTER_BIG_ENDIAN, &ctr) == CRYPT_OK);

```

This function call initializes the *ctr* structure with the parameters for an AES-128 CTR mode encryption with the given IV. We have chosen a big endian counter to be nice and portable. This function can fail, and while there are many ways to deal with errors (such as more graceful reporting), here we simply used an assertation. In reality, this code should never fail, but as you put code like this in larger applications, it is entirely possible that it may fail.

```
    /* create a plaintext */
    memset(plaintext, 0, sizeof(plaintext));
    strncpy(plaintext, "hello world how are you?",
            sizeof(plaintext));

```

We zero the entire buffer *plaintext* before copying our shorter string into it. This ensures that we have zero bytes in the eventual decryption we intend to display to the user.

```
    /* encrypt it */
    ctr_encrypt(plaintext, ciphertext, 32, &ctr);
```

This function call performs AES-128 CTR mode encryption of *plaintext* and stores it in *ciphertext*. In this case, we are encrypting 32 bytes. Note, however, the CTR mode is not restricted to dealing with multiples of the block size. We could have easily resized the buffers to 30 bytes and still call the function (substituting 30 for 32).

The *ctr_encrypt* function can be called as many times as required to encrypt the plaintext. Each time the same CTR structure is passed in, it is updated so that the next call will proceed from the point the previous call left off. For example,

```
ctr_encrypt("hello", ciphertext, 5, &ctr);
ctr_encrypt(" world", ciphertext+5, 6, &ctr);
```

and

```
ctr_encrypt("hello world", ciphertext, 11, &ctr);
```

perform the same operation.

```
   /* reset the IV */
   ctr_setiv(IV, 16, &ctr);

   /* decrypt it */
   ctr_decrypt(ciphertext, buf, 32, &ctr);
```

Before we can decrypt the text with the same CTR structure, we have to reset the IV. This is because after encrypting the plaintext the chaining value stored in the CTR structure has changed. If we attempted to decrypt it now, it would not work.

We use the *ctr_decrypt* function to perform the decryption from *ciphertext* to the *buf* array. For the curious, *ctr_decrypt* is just a placeholder that eventually calls *ctr_encrypt* to perform the decryption.

```
   /* print it */
   for (x = 0; x < 32; x++) printf("%c", buf[x]);
   printf("\n");

   return EXIT_SUCCESS;
}
```

At this point, the user should be presented with the string "hello world how are you?" and the program should terminate normally.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: What is a cipher?

A: A cipher is an algorithm that transforms an input (plaintext) into an output (ciphertext) with a secret key.

Q: What is the purpose of a cipher?

A: The first and foremost purpose of a cipher is to provide privacy to the user. This is accomplished by controlling the mapping from plaintext to ciphertext with a secret key.

Q: What standards are there for ciphers?

A: The Advanced Encryption Standard (AES) is specified in FIPS 197. The NIST standard SP 800-38A specifies five chaining modes, including CBC and CTR mode.

Q: What about the other ciphers?

A: Formally, NIST still recognizes Skipjack (FIPS 185) as a valid cipher. It is slower than AES, but well suited for small 8- and 16-bit processors due to the size and use of 8-bit operations. In Canada, the CSE (Communication Security Establishment) formally recognizes CAST[4] (CSE Web site of approved ciphers is at www.cse-cst.gc.ca/services/crypto-services/crypto-algorithms-e.html) in addition to all NIST approved modes. CAST5 is roughly as fast as AES, but nowhere near as flexible in terms of implementation. It is larger and harder to implement in hardware. Other common ciphers such as RC5, RC6, Blowfish, Twofish, and Serpent are parts of RFCs of one form or another, but are not part of official government standards. In the European Union, the NESSIE project selected Anubis and Khazad as its 128-bit and 64-bit block ciphers. Most countries formally recognize Rijndael (or often even AES) as their officially standardized block cipher.

Q: Where can I find implementations of ciphers such as AES?

A: Many libraries already support vast arrays of ciphers. LibTomCrypt supports a good mix of standard ciphers such as AES, Skipjack, DES, CAST5, and popular ciphers such as

Blowfish, Twofish, and Serpent. Similarly, Crypto++ supports a large mix of ciphers. OpenSSL supports a few, including AES, CAST5, DES, and Blowfish.

Q: What is a pseudo random permutation (PRP)?

A: A pseudo random permutation is a re-arrangement of symbols (in the case of AES, the integers 0 through $2^{128} - 1$) created by an algorithm (hence the pseudo random bit). The goal of a secure PRP is such that knowing part of the permutation is insufficient to have a significant probability of determining the rest of the permutation.

Q: How do I get authenticity with AES?

A: Use the CMAC algorithm explained in Chapter 6.

Q: Isn't CBC mode an authentication algorithm?

A: It can be, but you have to know what you are doing. Use CMAC.

Q: I heard CTR is insecure because it does not guarantee authenticity.

A: You heard wrong.

Q: Are you sure?

A: Yes.

Q: What is an IV?

A: The initial vector (IV) is a value used in chaining modes to deal with the first block. Usually, previous ciphertext (or counters) is used for every block after the first. IVs must be stored along with the ciphertext and are not secret.

Q: Does my CBC IV have to be random, or just unique, or what?

A: CBC IVs must be random.

Q: What about the IVs for CTR mode?

A: CTR IVs must only be unique. More precisely, they must never collide. This means that through the course of encrypting one message, the intermediate value of the counter must not equal the value of the counter while encrypting another message. That is, assuming you used the same key. If you change keys per message, you can re-use the same IV as much as you wish.

Q: What are the advantages of CTR mode over CBC mode?

A: CTR is simpler to implement in both hardware and software. CTR mode can also be implemented in parallel, which is important for hardware projects looking for gigabit per second speeds. CTR mode also is easier to set up, as it does not require a random IV, which makes certain packet algorithms more efficient as they have less overhead.

Q: Do I need a chaining mode? What about ECB mode?

A: Yes, you most likely need a chaining mode if you encrypt messages longer than the block size of the cipher (e.g., 16 bytes for AES). ECB mode is not really a mode. ECB means to apply the cipher independently to blocks of the message. It is totally insecure, as it allows frequency analysis and message determination.

Q: What mode do you recommend?

A: Unless there is some underlying standard you want to comply with, use CTR mode for privacy, if not for the space savings, then for the efficiency of the mode in terms of overhead and execution time.

Q: What are Key Derivation Functions?

A: Key Derivation Functions (KDF) are functions that map a secret onto essential parameters such as keys and IVs. For example, two parties may share a secret key K and wish to derive keys to encrypt their traffic. They might also need to generate IVs for their chaining modes. A KDF will allow them to generate the keys and IVs from a single shared secret key. They are explained in more detail in Chapter 5.

Chapter 5

Hash Functions

Solutions in this chapter:

- What Are Hash Functions?
- Designs of SHS and Implementation
- PKCS #5 Key Derivation
- Putting It All Together

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

Secure one-way hash functions are recurring tools in cryptosystems just like the symmetric block ciphers. They are highly flexible primitives that can be used to obtain privacy, integrity and authenticity. This chapter deals solely with the integrity aspects of hash functions.

A hash function (formally known as a *pseudo random function* or PRF) maps an arbitrary sized input to a fixed size output through a process known as *compression*. This form of compression is not your typical data compression (as you would see with a .zip file), but a noninvertible mapping. Loosely speaking, checksum algorithms are forms of "hash functions," and in many independent circles they are called just that. For example, mapping inputs to *hash buckets* is a simple way of storing arbitrary data that is efficiently searchable. In the cryptographic sense, hash functions must have two properties to be useful: they must be one-way and must be collision resistant. For these reasons, simple checksums and CRCs are not good hash functions for cryptography.

Being one-way implies that given the output of a hash function, learning anything useful about the input is nontrivial. This is an important property for a hash, since they are often used in conjunction with RNG seed data and user passwords. Most trivial checksums are not one-way, since they are linear functions. For short enough inputs, deducing the input from the output is often a simple computation.

Being collision resistant implies that given an output from the hash, finding another input that produces the same output (called a collision) is nontrivial. There are two forms of collision resistance that we require from a useful hash function. Pre-image collision resistance (Figure 5.1) states that given an output Y, finding another input M' such that the hash of M' equals Y is nontrivial. This is an important property for digital signatures since they apply their signature to the hash only. If collisions of this form were easy to find, an attacker could substitute one signed message for another message. Second pre-image collision resistance (Figure 5.2) states that finding two messages M1 (given) and M2 (chosen at random), whose hatches match is nontrivial.

Figure 5.1 Pre-Image Collision Resistance

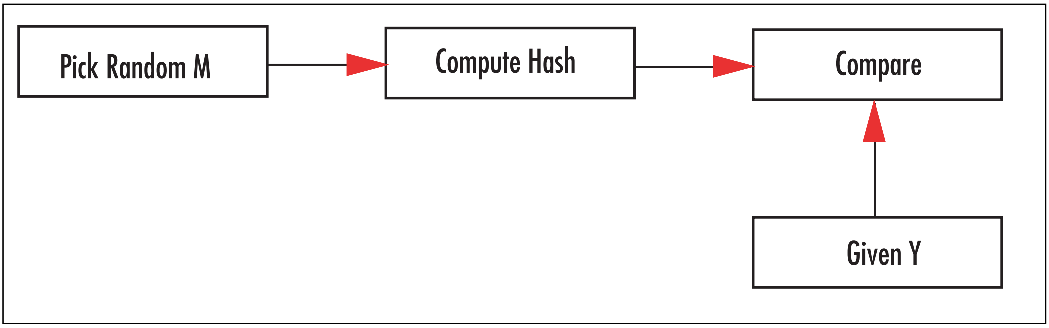
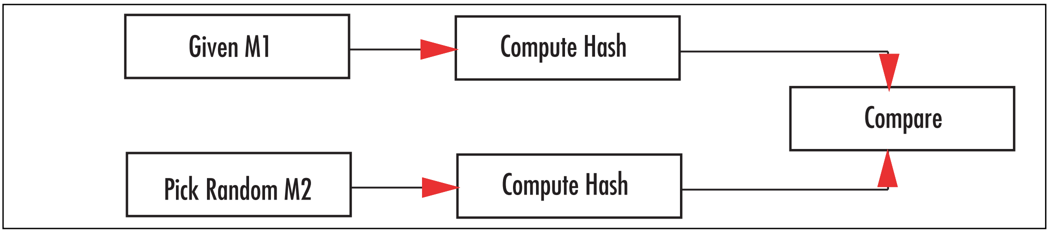


Figure 5.2 Second Pre-Image Collision Resistance



Throughout the years, there have been multiple proposals for secure hash functions. The reader may have even heard of algorithms such as MD4, MD5, or HAVAL. All of these algorithms have held their place in cryptographic tools and all have been broken.

MD4 and MD5 have shown to be fairly insecure as they are not collision resistant. HAVAL is suffering a similar fate, but the designers were careful enough to over design the algorithm. So far, it is still secure to use. NIST has provided designs for what it calls the Secure Hash Standard (FIPS 180-2), which includes the older SHA-1 hash function and newer SHA-2 family of hashes (SHA stands for Secure Hash Algorithm). We will refer to these SHS algorithms only in the rest of the text.

SHA-1 was the first family of hashes proposed by NIST. Originally, it was called SHA, but a flaw in the algorithm led to a tweak that became known as SHA-1 (and the old standard as SHA-0). NIST only recommends the use of SHA-1 and not SHA-0.

SHA-1 is a 160-bit hash function, which means that the output, also known as the *digest*, is 160 bits long. Like HAVAL, there are attacks on reduced variants of SHA-1 that can produce collisions, but there is no attack on the full SHA-1 as of this writing. The current recommendation is that SHA-1 is not insecure to use, but people instead use one of the SHA-2 algorithms.

SHA-2 is the informal name for the second round of SHS algorithms designed by NIST. They include the SHA-224, SHA-256, SHA-384, and SHA-512 algorithms. The number preceding SHA indicates the digest length. In the SHA-2 series, there are actually only two algorithms. SHA-224 uses SHA-256 with a minor modification and truncates the output to 224 bits. Similarly, SHA-384 uses SHA-512 and truncates the output. The current recommendation is to use at least SHA-256 as the default hash algorithm, especially if you are using AES-128 for privacy (as we shall see shortly).

Hash Digests Lengths

You may be wondering where all these sizes for hash digests come from. Why did SHA-2 start at 256 bits and go all the way up to 512 (SHA-224 was added to the SHS specification after the initial release)?

It turns out the resistance of a hash to collision is not as linear as one would hope. For example, the probability of a second pre-image collision in SHA-256 is not $1/2^{256}$ as one may think; instead, it is only at least $1/2^{128}$. An observation known as the *birthday paradox* states (roughly) that the probability of 23 people in a room sharing a birthday is roughly 50 percent.

This is because there are 23C2 = 253 (that is read as "23 choose 2") unique pairs. Each pair has a chance of 364/365 that the birthday is *not* the same. The chance that all the pairs are not the same is given by raising the fraction to the power of 253. Noticing the probability of an event and its negation must sum to one, we take this last result and deduct it from one to get a decent estimate for the probability that any birthdays match. It turns out to be fractionally over 50 percent.

As the number *n* grows, the nC2 operation is closely approximated by n^2, so with 2^{128} hashes we have 2^{256} pairs and expect to find a collision.

In effect, our hashes have half of their digest size in strength. SHA-256 takes 2^{128} work to find collisions; SHA-512 takes 2^{256} work; and so on. One important design guideline is to ensure that all of your primitives have equal "bit strength." There is no sense using AES-256 with SHA-1 (at least directly), as the hash only emits 160 bits; birthday paradoxes play less into this problem. They do, however, affect digital signatures, as we shall see.

SHA-1 output size of 160 bits actually comes from the (then) common use of RSA-1024 (see Chapter 9, "Public Key Algorithms"). Breaking a 1024-bit RSA key takes roughly 2^{86} work, which compares favorably to the difficulty of finding a hash collision of 2^{80} work. This means that an attacker would spend about as much time trying to find another document that collides with the victim's document, then breaking the RSA key itself.

What one should avoid is getting into a situation where you have a mismatch of strength. Using RSA-1024 with SHA-256 is not a bad idea, but you should be clearly aware that the strength of the combination is only 86 bits and not 128 bits. Similarly, using RSA-2048 (112 bits of strength) with SHA-1 would imply the attacker would only have to find a collision and not break the RSA key (which is much harder).

Table 5.1 indicates which standards apply to a given bit strength desired. It is important to note that the column indicating which SHS to use is only a minimum suggestion. You can safely use SHA-256 if your target is only 112 bits of security. The important thing to note is you do not gain strength by using a larger hash. For instance, if you are using ECC-192 and choose SHA-512, you still only have at most 96 bits of security (provided all else is going right). Choose your primitives wisely.

Table 5.1 Bit Strength and Hash Standard Matching

Bit Strength	ECC Strength	RSA Strength	SHS To Use
80	ECC-192*	RSA-1024	SHA-1
112	ECC-224	RSA-2048	SHA-224
128	ECC-256		SHA-256
192	ECC-384		SHA-384
256	ECC-521		SHA-512

*Technically, ECC-192 requires 2^{96} work to break, but it is the smallest standard ECC curve NIST provides.

Many (smart) people have written volumes on what key size to strive for. We will simplify it for you. Aim for at least 128 bits and use more if your application can tolerate it. Usually, larger keys mean slower algorithms, so it is important to take timing constraints in consideration. Smaller keys just mean you are begging for someone to put a cluster together and break your cryptography. In the end, if you are worried more about the key sizes you use and less about how the entire application works together, you are not doing a good job as a cryptographer.

Notes from the Underground…

MD5CRK Attack of the Hashes

A common way to find a collision in a fixed function without actually storing a huge list of values and comparing is *cycle finding*. The attack works by iterating the function on its output. You start with two or more different initial values and cycle until two of them collide; for example, if user A starts with A[–1] and user B starts with B[–1] such that A[–1] does not equal B[–1], we compute

A[i] = Hash(A[i-1])
B[i] = Hash(B[i-1])

Until A[i] equals B[i]. Clearly, comparing online is annoying if you want to distribute this attack. However, storing the entire list of A[i] and B[i] for comparison is very inefficient. A clever optimization is to store *distinguished points*. Usually, they are distinguished by a particular bit pattern. For example, only store the hash values for which the first *l*-bits are zero.

Now, if they collide they will produce colliding distinguished points as well. The value of *l* provides a tradeoff between memory on the collection side and efficiency. The more bits, the smaller your tables, but the longer it takes users to report distinguished points. The fewer bits you use, the larger the tables, and the slower the searches.

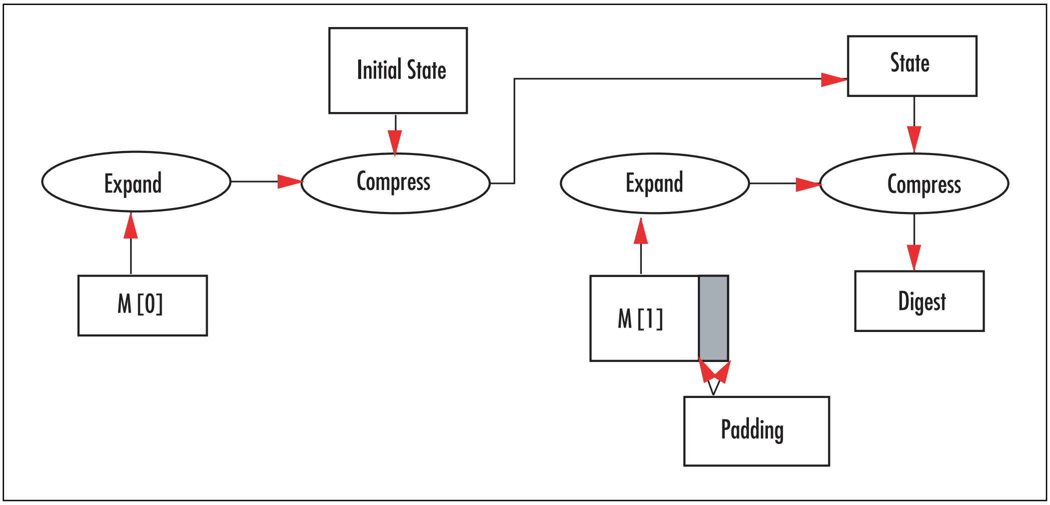
Designs of SHS and Implementation

As mentioned earlier, SHS FIPS 180-2 is comprised of three distinct algorithms: SHA-1, SHA-256, and SHA-512. From the last two, the alternate algorithms SHA-224 and SHA-384 can be constructed. We will first consider the three unique algorithms.

All three algorithms follow the same basic design flow. A block of the message is extracted, expanded, and then passed through a compression function that updates an internal state (which is the size of the message digest). All three algorithms employ padding to the message using a technique known as *MD strengthening*.

In Figure 5.3, we can see the flow of how the hash of the two block message M[0,1] is computed. M[0] is first expanded, and then compressed along with the existing hash state. The output of this is the new hash state. Next, we apply the Message Digest (MD) strengthening padding to M[1], expand it, and compress it with the hash state. Since this was the last block, the output of the compression is the hash digest.

Figure 5.3 Hash of a Two-Block Message



All three hashes are fairly easy to describe in terms of block cipher terminology. The message block (*key*) is expanded to a set of round keys. The round keys are then used to encrypt the current hash state, which performs a compression of the message to the smaller hash state size. This construction can turn a normal block cipher into a hash as well. In terms of AES, for example, we have

S[i] := S[i-1] xor AES(M[I], S[i-1])

where AES(M[i], S[i–1]) is the encryption of the previous state S[i–1] under the key M[i]. We use a fixed known value for S[–1], and by padding the message with MD strengthening we have constructed a secure hash. The problem with using traditional ciphers for this is that the key and ciphertext output are too small. For example, with AES-256 we can compress 32 bytes per call and produce a 128-bit digest. SHA-1, on the other hand, compresses 64 bytes per call and produces a 160-bit digest.

MD Strengthening

The process of MD strengthening was originally invented as part of the MD series of hashes by Dr. Rivest. The goal was to prevent a set of prefix and suffix attacks by encoding the length as part of the message.

The padding works as follows.

1. Append a single one bit to the message.
2. Append enough zero bits so the length of the message is congruent to *w-l* modulo *w*.
3. Append the length of the message in big endian format as an *l*-bit integer.

Where *w* is the block size of the hash function and *l* is the number of bits to encode the maximum message size. In the case of SHA-1, SHA-224, and SHA-256, we have *w* = 512 and *l* = 64. In the case of SHA-384 and SHA-512, we have *w* = 1024 and *l* = 128.

SHA-1 Design

SHA-1 is by far the most common hash function next to the MD series. At the time when it was invented, it was among the only hashes that provided a 160-bit digest and still maintained a decent level of efficiency. We will break down the SHA-1 design into three components: the state, the expansion function, and compression function.

SHA-1 State

The SHA-1 state is an array of five 32-bit words denoted as S[0...4], which hold the initial values of

```
S[0] = 0x67452301;
S[1] = 0xefcdab89;
S[2] = 0x98badcfe;
S[3] = 0x10325476;
S[4] = 0xc3d2e1f0;
```

Each invocation of the compression function updates the SHA-1 state. The final value of the SHA-1 state is the hash digest.

SHA-1 Expansion

SHA-1 processes the message in blocks of 64 bytes. Regardless of which block we are expanding (the first or the last with padding), the expansion process is the same. The expansion makes use of an array of 80 32-bit words denoted as W[0...79]. The first 16 words are loaded in big endian fashion from the message block.

The next 64 words are produced with the following code.

```
for (x = 16; x < 80; x++) {
   W[i] = ROL(W[i-3] ^ W[i-8] ^ W[i-14] ^ W[i-16], 1);
}
```

Where ROL(x, 1) is a left cyclic 32-bit rotation by 1 bit. At this point, we have fully expanded the 64-byte message block into round keys required for the compression function. The curious reader may wish to know that SHA-0 was defined without the rotation.

Without the rotation, the hash function does not have the desired 80 bits' worth of strength against collision searches.

SHA-1 Compression

The compression function is a type of Feistel network (do not worry if you do not know what that is) with 80 rounds. In each round, three of the words from the state are sent through a round function and then combined with the remaining two words. SHA-1 uses four types of rounds, each iterated 20 times for a total of 80 rounds.

The hash state must first be copied from the S[] array to some local variables. We shall call them {a,b,c,d,e}, such that a = S[0], b = S[1], and so on. The round structure resembles the following.

```
FFx(a,b,c,d,e,x) \
   e = (ROL(a, 5) + Fx(b,c,d) + e + W[x] + Kx); b = ROL(b, 30);
```

where *Fx(x,y,z)* is the round function for the *x*'th grouping. After each round, the words of the state are swapped such that *e* becomes *d*, *d* becomes *c*, and so on. *Kx* denotes the round constant for the *x*'th round. Each group of rounds has its own constant. The four round functions are given by the following code.

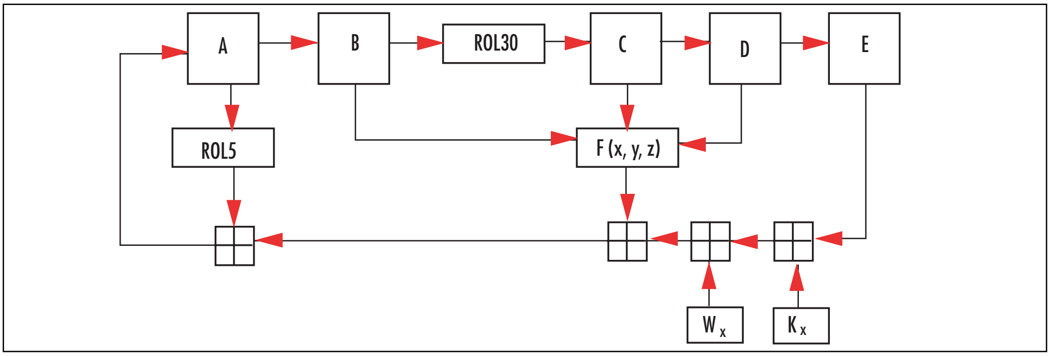
```
#define F0(x,y,z)  (z ^ (x & (y ^ z)))
#define F1(x,y,z)  (x ^ y ^ z)
#define F2(x,y,z)  ((x & y) | (z & (x | y)))
#define F3(x,y,z)  (x ^ y ^ z)
```

The round constants are the following.

```
K0...19  = 0x5a827999
K20...39 = 0x6ed9eba1
K40...59 = 0x8f1bbcdc
K60...79 = 0xca62c1d6
```

After 20 rounds of each type of round, the five words of {a,b,c,d,e} are added (as integers modulo 2^{32}) to their counterparts in the hash state. Another view of the round function is in Figure 5.4.

Figure 5.4 SHA-1 Round Function



SHA-1 Implementation

Our SHA-1 implementation is a direct translation of the standard and avoids a couple common optimizations that we will mention inline with the source.

sha1.c:

```
#if defined(__x86_64__)
   typedef unsigned ulong32;
#else
   typedef unsigned long ulong32;
#endif

/* Helpful macros */
#define STORE32H(x, y)                              \
     { (y)[0] = (unsigned char)(((x)>>24)&255);    \
       (y)[1] = (unsigned char)(((x)>>16)&255);    \
       (y)[2] = (unsigned char)(((x)>>8)&255);     \
       (y)[3] = (unsigned char)((x)&255); }

#define LOAD32H(x, y)                              \
     { x = ((ulong32)((y)[0] & 255)<<24) | \
           ((ulong32)((y)[1] & 255)<<16) | \
           ((ulong32)((y)[2] & 255)<<8)  | \
           ((ulong32)((y)[3] & 255)); }

#define ROL(x, y) \
((((ulong32)(x)<<(ulong32)((y)&31)) |  \
(((ulong32)(x)&0xFFFFFFFFUL)>> \
(ulong32)(32-((y)&31)))) & 0xFFFFFFFFUL)
```

Our familiar macros come up again in this implementation. These macros are a lifesaver in portable coding, as they totally eliminate any endianess issues we may have had.

```
#define F0(x,y,z)  (z ^ (x & (y ^ z)))
#define F1(x,y,z)  (x ^ y ^ z)
#define F2(x,y,z)  ((x & y) | (z & (x | y)))
#define F3(x,y,z)  (x ^ y ^ z)
```

These are the SHA-1 round functions.

```
typedef struct {
   unsigned char buf[64];
   unsigned long buflen, msglen;
   ulong32       S[5];
} sha1_state;
```

This structure holds an SHA-1 state. It allows us to process messages with multiple calls to sha1_process. For example, if we are hashing a message that is streaming or too large to fit in memory at once, we can use multiple calls to the process function to handle the entire message.

```
void sha1_init(sha1_state *md)
{
    md->S[0] = 0x67452301;
    md->S[1] = 0xefcdab89;
    md->S[2] = 0x98badcfe;
    md->S[3] = 0x10325476;
    md->S[4] = 0xc3d2e1f0;
    md->buflen = md->msglen = 0;
}
```

This function initializes the SHA-1 state to the default state. We set the S[] array to the SHA-1 defaults and set the buffer and message lengths to zero. The buffer length (buflen) variable counts how many bytes in the current block we have so far. When it reaches 64, we have to call the compression function to compress the data. The message length (msglen) variable counts the size of the entire message. In our case, we are counting bytes, which means that this routine is limited to 2^{32}–1 byte messages.

```
static void sha1_compress(sha1_state *md)
{
   ulong32 W[80], a, b, c, d, e, t;
   unsigned x;

   /* load W[0..15] */
   for (x = 0; x < 16; x++) {
       LOAD32H(W[x], md->buf + 4 * x);
   }
```

This function expands and compresses the message block into the state. This first loop loads the 64-byte block into W[0..15] in big endian format using the LOAD32H macro.

```
   /* compute W[16...79] */
   for (x = 16; x < 80; x++) {
       W[x] = ROL(W[x-3] ^ W[x-8] ^ W[x-14] ^ W[x-16], 1);
   }
```

This loop produces the rest of the W[] array entries. Like the AES key schedule (see Chapter 4, "Advanced Encryption Standard"), it is a form of shift register.

TIP

Like the AES key schedule, we can optimize SHA-1 in limited memory environments. By running the expansion on the fly, we only need 16 32-bit words (64 bytes) and not the full 80 32-bit words (320 bytes). The optimization takes advantage of the fact that W[x] and W[x-16] can overlap in memory.

```
   /* load a copy of the state */
   a = md->S[0]; b = md->S[1]; c = md->S[2];
   d = md->S[3]; e = md->S[4];

   /* 20 rounds */
   for (x = 0; x < 20; x++) {
```

```
067          e = (ROL(a, 5) + F0(b,c,d) + e + W[x] + 0x5a827999);
068          b = ROL(b, 30);
069          t = e; e = d; d = c; c = b; b = a; a = t;
070       }
```

This loop implements the first 20 rounds of the SHA-1 compression function. Here we have rolled the loop up to save space. There are two other common ways to implement it. One is partially to unroll it by copying the body of the loop five times. This allows us to use register *renaming* instead of the swaps (line 69). With a macro such as

```
#define FF0(a,b,c,d,e,i) \
e = (ROLc(a, 5) + F0(b,c,d) + e + W[i] + 0x5a827999UL); \
b = ROLc(b, 30);
```

the loop could be implemented as follows.

```
for (x = 0; x < 20; ) {
  FF0(a,b,c,d,e,x++);
  FF0(e,a,b,c,d,x++);
  FF0(d,e,a,b,c,x++);
  FF0(c,d,e,a,b,x++);
  FF0(b,c,d,e,a,x++);
}
```

This saves on all of the swaps and is more efficient (albeit five times larger) to execute. The next level of optimization is that we can unroll the entire loop replacing the "x++" with the constant round numbers. This level of unrolling rarely produces much benefit on modern processors (with good branch prediction) and solely serves to pollute the instruction cache.

```
072       /* 20 rounds */
073       for (; x < 40; x++) {
074          e = (ROL(a, 5) + F1(b,c,d) + e + W[x] + 0x6ed9eba1);
075          b = ROL(b, 30);
076          t = e; e = d; d = c; c = b; b = a; a = t;
077       }
078
079       /* 20 rounds */
080       for (; x < 60; x++) {
081          e = (ROL(a, 5) + F2(b,c,d) + e + W[x] + 0x8f1bbcdc);
082          b = ROL(b, 30);
083          t = e; e = d; d = c; c = b; b = a; a = t;
084       }
085
086       /* 20 rounds */
087       for (; x < 80; x++) {
088          e = (ROL(a, 5) + F3(b,c,d) + e + W[x] + 0xca62c1d6);
089          b = ROL(b, 30);
090          t = e; e = d; d = c; c = b; b = a; a = t;
091       }
```

These three loops implement the last 60 rounds of the SHA-1 compression. We can unroll them with the same style of code as the first round for extra performance.

```
   /* update state */
   md->S[0] += a;
   md->S[1] += b;
   md->S[2] += c;
   md->S[3] += d;
   md->S[4] += e;
}
```

This last bit of code updates the SHA-1 state by adding the copies of the state to the state itself. The additions are meant to be modulo 2^{32}. We do not have to reduce the results, as are rotate macros explicitly mask bits in the 32-bit range.

```
void sha1_process(         sha1_state *md,
                  const unsigned char *buf,
                        unsigned long  len)
{
```

This function is what the caller will use to add message bytes to the SHA-1 hash. The function copies message bytes from buf to the internal buffer and then proceeds to compress them when 64 bytes have accumulated.

```
   unsigned long x, y;

   while (len) {
       x    = (64 - md->buflen) < len ? 64 - md->buflen : len;
       len -= x;
```

Our buffer is only 64 bytes long, so we must prevent copying too much into it. After this point, *x* contains the number of bytes we have to copy to proceed.

```
       /* copy x bytes from buf to the buffer */
       for (y = 0; y < x; y++) {
           md->buf[md->buflen++] = *buf++;
       }

       if (md->buflen == 64) {
          sha1_compress(md);
          md->buflen  = 0;
          md->msglen += 64;
       }
   }
}
```

After copying the bytes, we check if our buffer has 64 bytes, and if so we call the compression function. After compression, we reset the buffer length and update the message length. There is an optimization we can apply to this process function, which is a method of performing *zero-copy compression*.

Suppose you enter the function with buflen equal to zero and you want to hash more than 63 bytes of data. Why does the data have to be copied to the buf[] array before the hash compresses it? By hashing out of the user supplied buffer directly, we avoid the costly double buffering that would otherwise be required. The optimization can be applied further. Suppose buflen was not zero, but we were hashing more than 64 bytes. We could double buffer until we fill buflen; then, if enough bytes remain we could zero-copy as many 64 byte blocks as remained.

This trick is fairly simple to implement and can easily give a several percentage point boost in performance.

```
124   void sha1_done(   sha1_state *md,
125                     unsigned char *dst)
126   {
```

This function terminates the hash and outputs the digest to the caller in the dst buffer.

```
127       ulong32 l1, l2, i;
128
129       /* compute final length as 8*md->msglen */
130       md->msglen += md->buflen;
131       l2 = md->msglen >> 29;
132       l1 = (md->msglen << 3) & 0xFFFFFFFF;
```

Technically, SHA-1 supports messages up to $2^{64}–1$ bits; however, as a matter of practicality we limit ourselves to $2^{32}–1$ bytes. Before we can encode the length, however, we have to encode it as the number of bits. We extract the upper bits of the message length (line 131) and then shift up by three, emulating a multiplication by 8. At this point, the 64-bit value $l2*2^{32} + l1$ represents the message length in bits as required for the padding scheme.

```
134       /* add the padding bit */
135       md->buf[md->buflen++] = 0x80;
```

This is the leading padding bit. Since we are dealing with byte units, we are always aligned on a byte boundary. SHA-1 is big endian, so the one bit turns into 0x80 (with seven padding zero bits).

```
137       /* if the current len > 56 we have to finish this block */
138       if (md->buflen > 56) {
139          while (md->buflen < 64) {
140              md->buf[md->buflen++] = 0x00;
141          }
142          sha1_compress(md);
143          md->buflen = 0;
144       }
```

If our current block is too large to accommodate the 64-bit length, we must pad with zero bytes until we hit 64 bytes. We compress and reset the buffer length counter.

```
146       /* now pad until we are at pos 56 */
147       while (md->buflen < 56) {
148           md->buf[md->buflen++] = 0x00;
149       }
```

At this point, buflen < 56 is guaranteed. We pad with enough zero bytes until we hit position 56.

```
151       /* store the length */
152       STORE32H(l2, md->buf + 56);
```

```
   STORE32H(l1, md->buf + 60);
```

We store the length of the message as a 64-bit big endian number. We are emulating this by performing two big endian 32-bit stores.

```
   /* compress */
   sha1_compress(md);
```

A final compression including the padding terminates the hash.

```
   /* extract the state */
   for (i = 0; i < 5; i++) {
      STORE32H(md->S[i], dst + i*4);
   }
}
```

Now we extract the digest, which is the five 32-bit words of the SHA-1 state. At this point, dst[0...19] contains the message digest of the message that was hashed.

```
void sha1_memory(const unsigned char *in,
                       unsigned long  len,
                       unsigned char *dst)
{
   sha1_state md;
   sha1_init(&md);
   sha1_process(&md, in, len);
   sha1_done(&md, dst);
}
```

This function is a nice helper function to have around. It hashes a complete message and outputs the digest in a single function call. This is useful if you are hashing small buffers that can fit in memory—for example, for hashing passwords and salts.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{
  static const struct {
      char *msg;
      unsigned char hash[20];
  } tests[] = {
    { "abc",
      { 0xa9, 0x99, 0x3e, 0x36, 0x47, 0x06, 0x81, 0x6a,
        0xba, 0x3e, 0x25, 0x71, 0x78, 0x50, 0xc2, 0x6c,
        0x9c, 0xd0, 0xd8, 0x9d }
    },
    { "abcdbcdecdefdefgefghfghighi"
      "jhijkijkljklmklmnlmnomnopnopq",
      { 0x84, 0x98, 0x3E, 0x44, 0x1C, 0x3B, 0xD2, 0x6E,
        0xBA, 0xAE, 0x4A, 0xA1, 0xF9, 0x51, 0x29, 0xE5,
        0xE5, 0x46, 0x70, 0xF1 }
    }
  };
  int i;
  unsigned char tmp[20];

  for (i = 0; i < 2; i++) {
     sha1_memory((unsigned char *)tests[i].msg,
                 strlen(tests[i].msg), tmp);
     if (memcmp(tests[i].hash, tmp, 20)) {
         printf("Failed test %d\n", i);
         return EXIT_FAILURE;
     }
  }
  printf("SHA-1 Passed\n");
  return EXIT_SUCCESS;
}
```

Our demo program uses two standard SHA-1 test vectors to try to determine if our code actually works. There are a few corner cases we are missing from the test suite (which are not part of the FIPS standard anyway). They:

1. A zero length (null) message.
    1. That is: **da39a3ee5e6b4b0d3255bfef95601890afd80709**
2. A message that is exactly 64 bytes long.
    1. 64 zero bytes: **c8d7d0ef0eedfa82d2ea1aa592845b9a6d4b02b7**
3. A message that is exactly 56 bytes long.
    1. 56 zero bytes: **9438e360f578e12c0e0e8ed28e2c125c1cefee16**
4. A message that is a proper multiple of 64 bytes long.
    1. 128 zero bytes: **0ae4f711ef5d6e9d26c611fd2c8c8ac45ecbf9e7**

The FIPS 800-2 standard provides three test vectors, two of which we used in our example. The third is a million a's, which should result in the digest 34aa973c d4c4daa4 f61eeb2b dbad2731 6534016f.

As we will see with other NIST standards, it is not uncommon for their test vectors to lack thoroughness. As an aside, how much do FIPS validation certificates cost these days anyway?

SHA-256 Design

SHA-256 follows a similar design (overall) as SHA-1. It uses a 256-bit state divided into eight 32-bit words denoted as S[0...7]. While it has the same expansion and compression feel as SHA-1, the actual operations are more complicated. SHA-256 uses a single round function that is repeated 64 times.

SHA-256 makes use of a set of eight simple functions defined as follows.

```
#define Ch(x,y,z)      (z ^ (x & (y ^ z)))
#define Maj(x,y,z)     (((x | y) & z) | (x & y))
```

These are the nonlinear functions used within the SHA-256 round function. They provide the linear complexity to the design required to ensure the function is one way and differences are hard to control. Without these functions, SHA-256 would be entirely linear (with the exception of the integer additions we will see shortly) and collisions would be trivial to find.

```
#define S(x, n)          ROR((x),(n))
#define R(x, n)          (((x)&0xFFFFFFFFUL)>>(n))
```

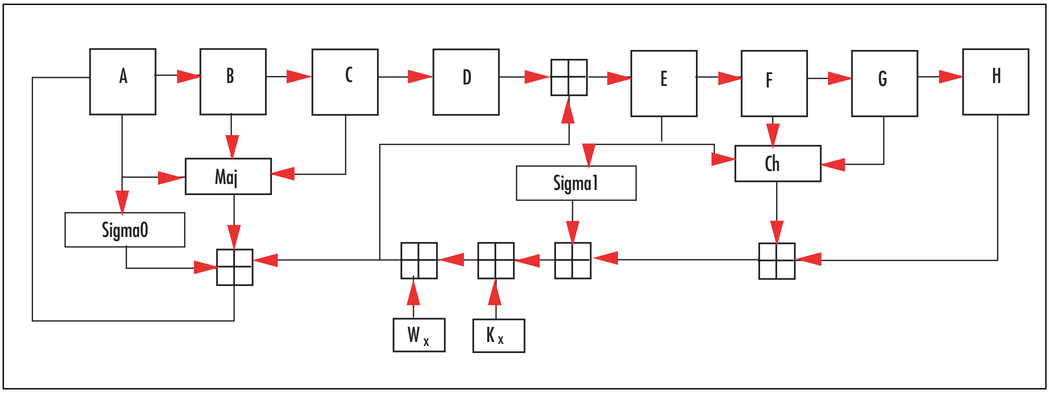
These are a right cyclic shift and a right logical shift, respectively. A curious reader may note that the S() macro performs a rotation and the R() macro performs a shift. These were macros based off the early drafts of the SHA-256 design. We left them like this partially because it's amusing and partially because it is what LibTomCrypt uses.

```
#define Sigma0(x)        (S(x, 2) ^ S(x, 13) ^ S(x, 22))
#define Sigma1(x)        (S(x, 6) ^ S(x, 11) ^ S(x, 25))
```

These two functions are used within the round function to promote diffusion. They replace the rotations by 5 and 30 bits used in SHA-1. Now, a single bit difference in the input will spread to two other bits in the output. This helps promote rapid diffusion through the design by being placed at just the right spot. If we examine the SHA-256 block diagram (Figure 5.5), we can see that the output of Sigma0 and Sigma1 eventually feeds back into the same word of the state the input came from. This means that after one round, a single bit affects three bits of the neighboring word. After two rounds, it affects at least nine bits, and so on. This feedback scheme has so far proven very effective at making the designs immune to cryptanalysis.

```
#define Gamma0(x)        (S(x, 7) ^ S(x, 18) ^ R(x, 3))
#define Gamma1(x)        (S(x, 17) ^ S(x, 19) ^ R(x, 10))
```

Figure 5.5 SHA-256 Compression Diagram



These two functions are used within the expansion phase to promote collision resistance. Within the SHA-256 (and FIPS 180-2 in general) standard, they use the upper- and lowercase Greek sigma for the Sigma and Gamma functions. We renamed the lowercase sigma functions to Gamma to prevent confusion in the source code.

SHA-256 State

The initial state of SHA-256 is 8 32-bit words denoted as S[0...7], specified as follows.

```
S[0] = 0x6A09E667;
S[1] = 0xBB67AE85;
S[2] = 0x3C6EF372;
S[3] = 0xA54FF53A;
S[4] = 0x510E527F;
S[5] = 0x9B05688C;
S[6] = 0x1F83D9AB;
S[7] = 0x5BE0CD19;
```

SHA-256 Expansion

Like SHA-1, the message block is expanded before it is used. The message block is expanded to 64 32-bit words.The first 16 words are loaded in big endian format from the 64 bytes of the message block. The next 48 words are computed with the following recurrence.

```
for (i = 16; i < 64; i++) {
   W[i] = Gamma1(W[i - 2]) + W[i - 7] +
          Gamma0(W[i - 15]) + W[i - 16];
}
```

SHA-256 Compression

Compression begins by making a copy of the SHA-256 state from S[0...7] to {a,b,c,d,e,f,g,h}. Next, 64 identical rounds are executed. The x'th round structure is as follows.

```
t0 = h + Sigma1(e) + Ch(e, f, g) + K[x] + W[x];
t1 = Sigma0(a) + Maj(a, b, c);
d += t0;
h  = t0 + t1;
```

After each round, the words are rotated such that *h* becomes *g*, *g* becomes *f*, *f* becomes *e*, and so on.The *K* array is a fixed array of 64 32-bit words (see the Implementation section). After the 64th round, the words {a,b,c,d,e,f,g,h} are added to the hash state in their respective locations.The hash state of after compressing the last message block is the message digest.

Comparing this diagram to that of SHA-1 (Figure 5.4), we can see that they apply two parallel nonlinear functions, two more complicated diffusion primitives (Sigma0 and Sigma1), and more feedback (into the A and E words). These changes make the new SHS algorithms have much higher diffusion and are less likely to be susceptible to the same class of attacks that are plaguing the MD5 and SHA-1 algorithms currently.

SHA-256 Implementation

Our SHA-256 implementation is directly adopted from the framework of the SHA-1 implementation. In a way, it highlights the similarities between the two hashes. It also highlights the fact that we did not want to write two different hash interfaces, partly out of laziness and partly because it would make the code harder to use.

```
#if defined(__x86_64__)
   typedef unsigned ulong32;
#else
   typedef unsigned long ulong32;
#endif

/* Helpful macros */
#define STORE32H(x, y)                            \
     { (y)[0] = (unsigned char)(((x)>>24)&255);   \
       (y)[1] = (unsigned char)(((x)>>16)&255);   \
       (y)[2] = (unsigned char)(((x)>>8)&255);    \
       (y)[3] = (unsigned char)((x)&255); }

#define LOAD32H(x, y)                            \
     { x = ((ulong32)((y)[0] & 255)<<24) | \
           ((ulong32)((y)[1] & 255)<<16) | \
           ((ulong32)((y)[2] & 255)<<8)  | \
           ((ulong32)((y)[3] & 255)); }

#define ROR(x, y) \
((((ulong32)(x)>>(ulong32)((y)&31)) |  \
(((ulong32)(x)&0xFFFFFFFFUL)<< \
(ulong32)(32-((y)&31)))) & 0xFFFFFFFFUL)
```

So far, very familiar with the exception we use a right cyclic rotation, not left.

```
#define Ch(x,y,z)       (z ^ (x & (y ^ z)))
#define Maj(x,y,z)      (((x | y) & z) | (x & y))
#define S(x, n)         ROR((x),(n))
#define R(x, n)         (((x)&0xFFFFFFFFUL)>>(n))
#define Sigma0(x)       (S(x, 2) ^ S(x, 13) ^ S(x, 22))
#define Sigma1(x)       (S(x, 6) ^ S(x, 11) ^ S(x, 25))
#define Gamma0(x)       (S(x, 7) ^ S(x, 18) ^ R(x, 3))
#define Gamma1(x)       (S(x, 17) ^ S(x, 19) ^ R(x, 10))
```

These are our SHA-256 macros for the expansion and compression phases.

```
typedef struct {
   unsigned char buf[64];
   unsigned long buflen, msglen;
   ulong32       S[8];
} sha256_state;
```

Our SHA-256 state with the longer S[] array. The variables otherwise have the same purposes as in the SHA-1 code.

```
static const ulong32 K[64] = {
    0x428a2f98UL, 0x71374491UL, 0xb5c0fbcfUL, 0xe9b5dba5UL,
    0x3956c25bUL, 0x59f111f1UL, 0x923f82a4UL, 0xab1c5ed5UL,
    0xd807aa98UL, 0x12835b01UL, 0x243185beUL, 0x550c7dc3UL,
    0x72be5d74UL, 0x80deb1feUL, 0x9bdc06a7UL, 0xc19bf174UL,
    0xe49b69c1UL, 0xefbe4786UL, 0x0fc19dc6UL, 0x240ca1ccUL,
    0x2de92c6fUL, 0x4a7484aaUL, 0x5cb0a9dcUL, 0x76f988daUL,
    0x983e5152UL, 0xa831c66dUL, 0xb00327c8UL, 0xbf597fc7UL,
    0xc6e00bf3UL, 0xd5a79147UL, 0x06ca6351UL, 0x14292967UL,
    0x27b70a85UL, 0x2e1b2138UL, 0x4d2c6dfcUL, 0x53380d13UL,
    0x650a7354UL, 0x766a0abbUL, 0x81c2c92eUL, 0x92722c85UL,
    0xa2bfe8a1UL, 0xa81a664bUL, 0xc24b8b70UL, 0xc76c51a3UL,
    0xd192e819UL, 0xd6990624UL, 0xf40e3585UL, 0x106aa070UL,
    0x19a4c116UL, 0x1e376c08UL, 0x2748774cUL, 0x34b0bcb5UL,
    0x391c0cb3UL, 0x4ed8aa4aUL, 0x5b9cca4fUL, 0x682e6ff3UL,
    0x748f82eeUL, 0x78a5636fUL, 0x84c87814UL, 0x8cc70208UL,
    0x90befffaUL, 0xa4506cebUL, 0xbef9a3f7UL, 0xc67178f2UL
};
```

This is the complete K[] array required by the compression function. NIST states they are the 32 bits of the fractional part of the cube root of the first 32 primes. For example, $2^{1/3}$ truncated to just the fractional part is 0.2599210498948731647672106072782, which when multiplied by 2^{32} is 1116352408.8404644807431890114033, and rounded down produces our first table entry 0x428A2F98.

We guess NIST never thought it may be a good idea to be able to generate the K[] values on the fly. In their defense, they chose such an odd table generation function so they could claim there are no "trap doors" in the table.

```
void sha256_init(sha256_state *md)
{
    md->S[0] = 0x6A09E667UL;
    md->S[1] = 0xBB67AE85UL;
    md->S[2] = 0x3C6EF372UL;
    md->S[3] = 0xA54FF53AUL;
    md->S[4] = 0x510E527FUL;
    md->S[5] = 0x9B05688CUL;
    md->S[6] = 0x1F83D9ABUL;
    md->S[7] = 0x5BE0CD19UL;
    md->buflen = md->msglen = 0;
}
```

This initializes the state to the default SHA-256 state.

```
static void sha256_compress(sha256_state *md)
{
   ulong32 W[64], a, b, c, d, e, f, g, h, t, t0, t1;
   unsigned x;

   /* load W[0..15] */
   for (x = 0; x < 16; x++) {
       LOAD32H(W[x], md->buf + 4 * x);
   }

   /* compute W[16...63] */
   for (x = 16; x < 64; x++) {
       W[x] = Gamma1(W[x - 2]) + W[x - 7] +
              Gamma0(W[x - 15]) + W[x - 16];
   }
```

At this point, we have fully expanded the message block to W[0...63] and can begin compressing the data. Note that the additions are modulo 2^{32}, but as it turns out, we do not have to explicitly reduce them, as the round function only uses the rotation macros. Like SHA-1, we can expand the block on the fly and in place if memory is tight.

```
/* load a copy of the state */
a = md->S[0]; b = md->S[1]; c = md->S[2];
d = md->S[3]; e = md->S[4]; f = md->S[5];
g = md->S[6]; h = md->S[7];

/* perform 64 rounds */
for (x = 0; x < 64; x++) {
  t0 = h + Sigma1(e) + Ch(e, f, g) + K[x] + W[x];
  t1 = Sigma0(a) + Maj(a, b, c);
  d += t0;
  h  = t0 + t1;

  /* swap */
  t = h; h = g; g = f; f = e; e = d;
  d = c; c = b; b = a; a = t;
}
```

Like the SHA-1 implementation, we have fully rolled the loop. We can unroll it either eight times or fully to gain a boost in speed. Unrolling it fully also allows us to embed the K[] constants in the code, which avoids a table lookup during the round function. The benefits of fully unrolling depend on the platform. On most x86 platforms, the gains are slight (if any) and the code size increase can be dramatic.

By defining a round macro, we could perform the unrolling nicely.

```
#define RND(a,b,c,d,e,f,g,h,i)                             \
     t0 = h + Sigma1(e) + Ch(e, f, g) + K[i] + W[i];   \
     t1 = Sigma0(a) + Maj(a, b, c);                    \
     d += t0;                                          \
     h  = t0 + t1;

for (i = 0; i < 64; ) {
    RND(a,b,c,d,e,f,g,h,i); ++i;
    RND(h,a,b,c,d,e,f,g,i); ++i;
    RND(g,h,a,b,c,d,e,f,i); ++i;
    RND(f,g,h,a,b,c,d,e,i); ++i;
    RND(e,f,g,h,a,b,c,d,i); ++i;
    RND(d,e,f,g,h,a,b,c,i); ++i;
    RND(c,d,e,f,g,h,a,b,i); ++i;
    RND(b,c,d,e,f,g,h,a,i); ++i;
}
```

This will perform the SHA-256 compression without performing all of the swaps of the previous routine. Note that we increment *i* after the macro, as it is used multiple times before we hit a *sequence point* in the source.

```
   /* update state */
   md->S[0] += a;
   md->S[1] += b;
   md->S[2] += c;
   md->S[3] += d;
   md->S[4] += e;
   md->S[5] += f;
   md->S[6] += g;
   md->S[7] += h;
}
```

Like SHA-1, we add the copies to the SHA-256 state to terminate the compression function.

```
void sha256_process(     sha256_state *md,
                  const unsigned char *buf,
                        unsigned long  len)
{
   unsigned long x, y;

   while (len) {
       x    = (64 - md->buflen) < len ? 64 - md->buflen : len;
       len -= x;

       /* copy x bytes from buf to the buffer */
       for (y = 0; y < x; y++) {
           md->buf[md->buflen++] = *buf++;
       }

       if (md->buflen == 64) {
          sha256_compress(md);
          md->buflen  = 0;
          md->msglen += 64;
       }
   }
}
```

This is a direct copy of the SHA-1 function, with the sha1_compress function swapped for sha256_compress.

NOTE

The "process" functions of most popular hashes such as MD4, MD5, SHA-1, SHA-256, and so on are so similar that it is possible to use a single macro to expand out to the required process function for the given hash.

This technique is used in the LibTomCrypt library so that optimizations applied to the macro (such as zero-copy hashing) apply to all hashes that use the macro.

```
void sha256_done(   sha256_state *md,
               unsigned char *dst)
{
   ulong32 l1, l2, i;

   /* compute final length as 8*md->msglen */
   md->msglen += md->buflen;
   l2 = md->msglen >> 29;
   l1 = (md->msglen << 3) & 0xFFFFFFFF;

   /* add the padding bit */
   md->buf[md->buflen++] = 0x80;

   /* if the current len > 56 we have to finish this block */
   if (md->buflen > 56) {
      while (md->buflen < 64) {
          md->buf[md->buflen++] = 0x00;
      }
      sha256_compress(md);
      md->buflen = 0;
   }

   /* now pad until we are at pos 56 */
   while (md->buflen < 56) {
       md->buf[md->buflen++] = 0x00;
   }

   /* store the length */
   STORE32H(l2, md->buf + 56);
   STORE32H(l1, md->buf + 60);

   /* compress */
   sha256_compress(md);

   /* extract the state */
   for (i = 0; i < 8; i++) {
      STORE32H(md->S[i], dst + i*4);
   }
}
```

This function is also copied from the SHA-1 implementation with the function name changes and we store eight 32-bit words instead of five.

```
void sha256_memory(const unsigned char *in,
                       unsigned long  len,
                       unsigned char *dst)
{
   sha256_state md;
   sha256_init(&md);
   sha256_process(&md, in, len);
   sha256_done(&md, dst);
}
```

Our helper function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{
  static const struct {
      char *msg;
      unsigned char hash[32];
  } tests[] = {
    { "abc",
      { 0xba, 0x78, 0x16, 0xbf, 0x8f, 0x01, 0xcf, 0xea,
        0x41, 0x41, 0x40, 0xde, 0x5d, 0xae, 0x22, 0x23,
        0xb0, 0x03, 0x61, 0xa3, 0x96, 0x17, 0x7a, 0x9c,
        0xb4, 0x10, 0xff, 0x61, 0xf2, 0x00, 0x15, 0xad }
    },
    { "abcdbcdecdefdefgefghfghighijhi"
      "jkijkljklmklmnlmnomnopnopq",
      { 0x24, 0x8d, 0x6a, 0x61, 0xd2, 0x06, 0x38, 0xb8,
        0xe5, 0xc0, 0x26, 0x93, 0x0c, 0x3e, 0x60, 0x39,
        0xa3, 0x3c, 0xe4, 0x59, 0x64, 0xff, 0x21, 0x67,
        0xf6, 0xec, 0xed, 0xd4, 0x19, 0xdb, 0x06, 0xc1 }
    },
  };
  int i;
  unsigned char tmp[32];

  for (i = 0; i < 2; i++) {
     sha256_memory((unsigned char *)tests[i].msg,
                 strlen(tests[i].msg), tmp);
     if (memcmp(tests[i].hash, tmp, 32)) {
         printf("Failed test %d\n", i);
         return EXIT_FAILURE;
     }
  }
  printf("SHA-256 Passed\n");
  return EXIT_SUCCESS;
}
```

Our test program to ensure our SHA-256 implementation is working correctly.

SHA-512 Design

SHA-512 was designed after the SHA-256 algorithm. It differs in the message block size, round constants, number of rounds, and the Sigma/Gamma functions. It has a state comprised of eight 64-bit words and follows the same expansion and compression workflow as the other hashes. SHA-512 uses 128-byte blocks instead of the 64-byte blocks SHA-1 and SHA-256 use. The new macros are the following.

```
#define Sigma0(x)      (S(x, 28) ^ S(x, 34) ^ S(x, 39))
#define Sigma1(x)      (S(x, 14) ^ S(x, 18) ^ S(x, 41))
```

These are the round function macros; note that the rotations and shifts are of 64-bit words, not 32-bit words as in the case of SHA-256.

```
#define Gamma0(x)        (S(x, 1) ^ S(x, 8) ^ R(x, 7))
#define Gamma1(x)        (S(x, 19) ^ S(x, 61) ^ R(x, 6))
```

These are the expansion function macros; again, they are 64-bit operations.

SHA-512 State

The SHA-512 state is eight 64-bit words denoted as S[0...7], initially set to the following values.

```
S[0] = 0x6a09e667f3bcc908;
S[1] = 0xbb67ae8584caa73b;
S[2] = 0x3c6ef372fe94f82b;
S[3] = 0xa54ff53a5f1d36f1;
S[4] = 0x510e527fade682d1;
S[5] = 0x9b05688c2b3e6c1f;
S[6] = 0x1f83d9abfb41bd6b;
S[7] = 0x5be0cd19137e2179;
```

SHA-512 Expansion

SHA-512 expansion works the same as SHA-256 expansion, except it uses the 64-bit macros and we must produce 80 words instead. The first 16 64-bit words are loaded from the 128-byte message block as big endian format. The next 64 words are generated with the following recurrence.

```
for (i = 16; i < 80; i++) {
   W[i] = Gamma1(W[i - 2]) + W[i - 7] +
          Gamma0(W[i - 15]) + W[i - 16];
}
```

SHA-512 Compression

SHA-512 compression is similar to SHA-256 compression. It uses the same pattern for the round function, except there are 80 rounds instead of 64. It also uses 80 64-bit constant words denoted K[0...79], which are derived in much the same manner as the 64 words used in the SHA-256 algorithm.

SHA-512 Implementation

Following is our implementation of SHA-512 derived from the SHA-256 source code. It has the same calling conventions as the SHA-1 and SHA-256 implementation, with the exception that it produces a 64-byte message digest instead of a 20- or 32-byte message.

sha512.c:

```
#ifdef _MSC_VER
```

```
   #define CONST64(n) n ## ui64
   typedef unsigned __int64 ulong64;
#else
   #define CONST64(n) n ## ULL
   typedef unsigned long long ulong64;
#endif
```

First, we need a way to use 64-bit data types. SHA-512 is for the most part an adaptation of SHA-256 with 64-bit words instead of 32-bit words. In C99, a data type that is *at least* 64-bits was defined to be *unsigned long long*, which works well with most UNIX CC and the GNU CC. Unfortunately, Microsoft's compiler does not support C99 and implements this differently.

Our CONST64 macro gives us a reasonably portable manner of defining 64-bit constants. Our ulong64 typedef allows us to use 64-bit words in an efficient fashion.

```
/* Helpful macros */
#define STORE64H(x, y)                            \
   { (y)[0] = (unsigned char)(((x)>>56)&255);     \
     (y)[1] = (unsigned char)(((x)>>48)&255);     \
     (y)[2] = (unsigned char)(((x)>>40)&255);     \
     (y)[3] = (unsigned char)(((x)>>32)&255);     \
     (y)[4] = (unsigned char)(((x)>>24)&255);     \
     (y)[5] = (unsigned char)(((x)>>16)&255);     \
     (y)[6] = (unsigned char)(((x)>>8)&255);      \
     (y)[7] = (unsigned char)((x)&255); }

#define LOAD64H(x, y)                         \
   { x = (((ulong64)((y)[0] & 255))<<56)| \
         (((ulong64)((y)[1] & 255))<<48)| \
         (((ulong64)((y)[2] & 255))<<40)| \
         (((ulong64)((y)[3] & 255))<<32)| \
         (((ulong64)((y)[4] & 255))<<24)| \
         (((ulong64)((y)[5] & 255))<<16)| \
         (((ulong64)((y)[6] & 255))<<8)|  \
         (((ulong64)((y)[7] & 255))); }

#define ROR(x, y) \
((((ulong64)(x)>>(ulong64)((y)&63)) |  \
(((ulong64)(x)&CONST64(0xFFFFFFFFFFFFFFFF))<< \
(ulong64)(64-((y)&63)))) & CONST64(0xFFFFFFFFFFFFFFFF))
```

These are our friendly macros adapted for 64-bit data types. Best to place these in a shared header.

```
#define Ch(x,y,z)   (z ^ (x & (y ^ z)))
#define Maj(x,y,z)  (((x | y) & z) | (x & y))
#define S(x, n)     ROR((x),(n))
#define R(x, n)     (((x)&CONST64(0xFFFFFFFFFFFFFFFF)) \
                     >>((ulong64)n))
#define Sigma0(x)   (S(x, 28) ^ S(x, 34) ^ S(x, 39))
#define Sigma1(x)   (S(x, 14) ^ S(x, 18) ^ S(x, 41))
#define Gamma0(x)   (S(x, 1) ^ S(x, 8) ^ R(x, 7))
#define Gamma1(x)   (S(x, 19) ^ S(x, 61) ^ R(x, 6))
```

These are the SHA macros for the 512-bit hash. Note that we are performing 64-bit operations here (including the shifts and rotations).

```
typedef struct {
   unsigned char buf[128];
   unsigned long buflen, msglen;
   ulong64       S[8];
} sha512_state;
```

This is our SHA-512 state. Note that SHA-512 uses a 128-byte block so our buffer is now larger. There are still eight chaining variables, but they are now 64 bits.

```
static const ulong64 K[80] = {
CONST64(0x428a2f98d728ae22), CONST64(0x7137449123ef65cd),
CONST64(0xb5c0fbcfec4d3b2f), CONST64(0xe9b5dba58189dbbc),
CONST64(0x3956c25bf348b538), CONST64(0x59f111f1b605d019),
CONST64(0x923f82a4af194f9b), CONST64(0xab1c5ed5da6d8118),
CONST64(0xd807aa98a3030242), CONST64(0x12835b0145706fbe),
CONST64(0x243185be4ee4b28c), CONST64(0x550c7dc3d5ffb4e2),
CONST64(0x72be5d74f27b896f), CONST64(0x80deb1fe3b1696b1),
CONST64(0x9bdc06a725c71235), CONST64(0xc19bf174cf692694),
CONST64(0xe49b69c19ef14ad2), CONST64(0xefbe4786384f25e3),
CONST64(0x0fc19dc68b8cd5b5), CONST64(0x240ca1cc77ac9c65),
CONST64(0x2de92c6f592b0275), CONST64(0x4a7484aa6ea6e483),
CONST64(0x5cb0a9dcbd41fbd4), CONST64(0x76f988da831153b5),
CONST64(0x983e5152ee66dfab), CONST64(0xa831c66d2db43210),
CONST64(0xb00327c898fb213f), CONST64(0xbf597fc7beef0ee4),
CONST64(0xc6e00bf33da88fc2), CONST64(0xd5a79147930aa725),
CONST64(0x06ca6351e003826f), CONST64(0x142929670a0e6e70),
CONST64(0x27b70a8546d22ffc), CONST64(0x2e1b21385c26c926),
CONST64(0x4d2c6dfc5ac42aed), CONST64(0x53380d139d95b3df),
CONST64(0x650a73548baf63de), CONST64(0x766a0abb3c77b2a8),
CONST64(0x81c2c92e47edaee6), CONST64(0x92722c851482353b),
CONST64(0xa2bfe8a14cf10364), CONST64(0xa81a664bbc423001),
CONST64(0xc24b8b70d0f89791), CONST64(0xc76c51a30654be30),
CONST64(0xd192e819d6ef5218), CONST64(0xd69906245565a910),
CONST64(0xf40e35855771202a), CONST64(0x106aa07032bbd1b8),
CONST64(0x19a4c116b8d2d0c8), CONST64(0x1e376c085141ab53),
CONST64(0x2748774cdf8eeb99), CONST64(0x34b0bcb5e19b48a8),
CONST64(0x391c0cb3c5c95a63), CONST64(0x4ed8aa4ae3418acb),
CONST64(0x5b9cca4f7763e373), CONST64(0x682e6ff3d6b2b8a3),
CONST64(0x748f82ee5defb2fc), CONST64(0x78a5636f43172f60),
CONST64(0x84c87814a1f0ab72), CONST64(0x8cc702081a6439ec),
CONST64(0x90befffa23631e28), CONST64(0xa4506cebde82bde9),
CONST64(0xbef9a3f7b2c67915), CONST64(0xc67178f2e372532b),
CONST64(0xca273eceea26619c), CONST64(0xd186b8c721c0c207),
CONST64(0xeada7dd6cde0eb1e), CONST64(0xf57d4f7fee6ed178),
CONST64(0x06f067aa72176fba), CONST64(0x0a637dc5a2c898a6),
CONST64(0x113f9804bef90dae), CONST64(0x1b710b35131c471b),
CONST64(0x28db77f523047d84), CONST64(0x32caab7b40c72493),
CONST64(0x3c9ebe0a15c9bebc), CONST64(0x431d67c49c100d4c),
CONST64(0x4cc5d4becb3e42b6), CONST64(0x597f299cfc657e2a),
CONST64(0x5fcb6fab3ad6faec), CONST64(0x6c44198c4a475817)
};
```

These are the 80 64-bit round constants for the compression function. Note we are using our CONST64 macro for portability.

```
void sha512_init(sha512_state *md)
{
    md->S[0] = CONST64(0x6a09e667f3bcc908);
    md->S[1] = CONST64(0xbb67ae8584caa73b);
    md->S[2] = CONST64(0x3c6ef372fe94f82b);
    md->S[3] = CONST64(0xa54ff53a5f1d36f1);
    md->S[4] = CONST64(0x510e527fade682d1);
    md->S[5] = CONST64(0x9b05688c2b3e6c1f);
    md->S[6] = CONST64(0x1f83d9abfb41bd6b);
    md->S[7] = CONST64(0x5be0cd19137e2179);
    md->buflen = md->msglen = 0;
}
```

This function initializes our SHA-512 state to the default state.

```
static void sha512_compress(sha512_state *md)
{
   ulong64 W[80], a, b, c, d, e, f, g, h, t, t0, t1;
   unsigned x;

   /* load W[0..15] */
   for (x = 0; x < 16; x++) {
       LOAD64H(W[x], md->buf + 8 * x);
   }

   /* compute W[16...80] */
   for (x = 16; x < 80; x++) {
       W[x] = Gamma1(W[x - 2]) + W[x - 7] +
              Gamma0(W[x - 15]) + W[x - 16];
   }
```

At this point, we have expanded the 128 bytes of message input into 80 64-bit words. Just like the SHA-1 and SHA-256 functions, we can compute the expanded words on the fly.

```
   /* load a copy of the state */
   a = md->S[0]; b = md->S[1]; c = md->S[2];
   d = md->S[3]; e = md->S[4]; f = md->S[5];
   g = md->S[6]; h = md->S[7];

   /* perform 80 rounds */
   for (x = 0; x < 80; x++) {
     t0 = h + Sigma1(e) + Ch(e, f, g) + K[x] + W[x];
     t1 = Sigma0(a) + Maj(a, b, c);
     d += t0;
     h  = t0 + t1;

     /* swap */
     t = h; h = g; g = f; f = e; e = d;
     d = c; c = b; b = a; a = t;
   }
```

This performs the compression round functions. We can unroll this either eight times or fully. Usually, eight times will net a sizeable performance boost, while unrolling fully will not pay off as much. On the AMD64 series, unrolling fully does not improve performance and wastes cache space.

```
   /* update state */
   md->S[0] += a;
   md->S[1] += b;
   md->S[2] += c;
   md->S[3] += d;
   md->S[4] += e;
   md->S[5] += f;
   md->S[6] += g;
   md->S[7] += h;
}

void sha512_process(     sha512_state *md,
                  const unsigned char *buf,
                        unsigned long  len)
{
   unsigned long x, y;

   while (len) {
       x    = (128 - md->buflen) < len ? 128 - md->buflen : len;
       len -= x;

       /* copy x bytes from buf to the buffer */
       for (y = 0; y < x; y++) {
           md->buf[md->buflen++] = *buf++;
       }

       if (md->buflen == 128) {
          sha512_compress(md);
          md->buflen  = 0;
          md->msglen += 128;
       }
   }
}
```

The process function resembles the process functions from SHA-1 and SHA-256, with the exception that we use 128-byte blocks. As in the other algorithms, we can use a zero-copy mechanism here as well, and it is highly recommended.

```
void sha512_done(   sha512_state *md,
               unsigned char *dst)
{
   ulong64 l1, l2, i;

   /* compute final length as 8*md->msglen */
   md->msglen += md->buflen;
   l2 = md->msglen >> 29;
   l1 = (md->msglen << 3) & 0xFFFFFFFF;

   /* add the padding bit */
   md->buf[md->buflen++] = 0x80;

   /* if the current len > 112 we have to finish this block */
   if (md->buflen > 112) {
      while (md->buflen < 128) {
          md->buf[md->buflen++] = 0x00;
      }
      sha512_compress(md);
      md->buflen = 0;
   }

   /* now pad until we are at pos 112 */
   while (md->buflen < 112) {
       md->buf[md->buflen++] = 0x00;
   }

   /* store the length */
   STORE64H(l2, md->buf + 112);
   STORE64H(l1, md->buf + 120);

   /* compress */
   sha512_compress(md);

   /* extract the state */
   for (i = 0; i < 8; i++) {
     STORE64H(md->S[i], dst + i*8);
   }
}
```

This function terminates the hash and outputs the digest. Note we have to store a 128-bit length. Since our block size is 128 bytes, we have to pad until our message is 112 modulo 128 bytes long.

```
void sha512_memory(const unsigned char *in,
                       unsigned long  len,
                       unsigned char *dst)
{
   sha512_state md;
   sha512_init(&md);
   sha512_process(&md, in, len);
   sha512_done(&md, dst);
}
```

This is our familiar helper function to perform a SHA-512 compression of an in-memory buffer.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{
  static const struct {
      char *msg;
      unsigned char hash[64];
  } tests[] = {
    { "abc",
     { 0xdd, 0xaf, 0x35, 0xa1, 0x93, 0x61, 0x7a, 0xba,
       0xcc, 0x41, 0x73, 0x49, 0xae, 0x20, 0x41, 0x31,
       0x12, 0xe6, 0xfa, 0x4e, 0x89, 0xa9, 0x7e, 0xa2,
       0x0a, 0x9e, 0xee, 0xe6, 0x4b, 0x55, 0xd3, 0x9a,
       0x21, 0x92, 0x99, 0x2a, 0x27, 0x4f, 0xc1, 0xa8,
       0x36, 0xba, 0x3c, 0x23, 0xa3, 0xfe, 0xeb, 0xbd,
       0x45, 0x4d, 0x44, 0x23, 0x64, 0x3c, 0xe8, 0x0e,
       0x2a, 0x9a, 0xc9, 0x4f, 0xa5, 0x4c, 0xa4, 0x9f }
    },
    { "abcdefghbcdefghicdefghijdefghijkefghijklfghijkl"
      "mghijklmnhijklmnoijklmnopjklmnopqklmnopqrlmnopq"
      "rsmnopqrstnopqrstu",
     { 0x8e, 0x95, 0x9b, 0x75, 0xda, 0xe3, 0x13, 0xda,
       0x8c, 0xf4, 0xf7, 0x28, 0x14, 0xfc, 0x14, 0x3f,
       0x8f, 0x77, 0x79, 0xc6, 0xeb, 0x9f, 0x7f, 0xa1,
       0x72, 0x99, 0xae, 0xad, 0xb6, 0x88, 0x90, 0x18,
       0x50, 0x1d, 0x28, 0x9e, 0x49, 0x00, 0xf7, 0xe4,
       0x33, 0x1b, 0x99, 0xde, 0xc4, 0xb5, 0x43, 0x3a,
       0xc7, 0xd3, 0x29, 0xee, 0xb6, 0xdd, 0x26, 0x54,
       0x5e, 0x96, 0xe5, 0x5b, 0x87, 0x4b, 0xe9, 0x09 }
    },
  };
  int i;
  unsigned char tmp[64];
  for (i = 0; i < 2; i++) {
     sha512_memory((unsigned char *)tests[i].msg,
                 strlen(tests[i].msg), tmp);
     if (memcmp(tests[i].hash, tmp, 64)) {
         printf("Failed test %d\n", i);
         return EXIT_FAILURE;
     }
  }
  printf("SHA-512 Passed\n");
  return EXIT_SUCCESS;
}
```

This is our demo application that tests the implementation. We have only implemented the first two of three NIST standard test vectors.

SHA-224 Design

SHA-224 is a hash that produces a 224-bit message digest and uses SHA-256 to perform the hashing. SHA-224 begins by initializing the state with eight different words listed here.

```
S[0] = 0xc1059ed8;
S[1] = 0x367cd507;
S[2] = 0x3070dd17;
S[3] = 0xf70e5939;
S[4] = 0xffc00b31;
S[5] = 0x68581511;
S[6] = 0x64f98fa7;
S[7] = 0xbefa4fa4;
```

SHA-224 uses the SHA-256 expansion and compression function on all the message blocks. The final digest as produced by SHA-256 is truncated by copying the first 28 bytes to the caller.

From a performance point of view, SHA-224 is no faster than SHA-256, and from a security standpoint, it is less secure. There are times when truncating a hash is a good idea, as we will see with digital signatures, but there are simpler ways to accomplish it than modifying an existing hash.

SHA-224 is a useful choice if your application is working with ECC-224, as it produces an output that has roughly the same size as the order of the curve—a property that will be shown to be useful in Chapter 9.

For completeness, here are the test vectors for SHA-224 similar to those from SHA-256.

```
static const struct {
    char *msg;
    unsigned char hash[28];
} tests[] = {
  { "abc",
    { 0x23, 0x09, 0x7d, 0x22, 0x34, 0x05, 0xd8,
      0x22, 0x86, 0x42, 0xa4, 0x77, 0xbd, 0xa2,
      0x55, 0xb3, 0x2a, 0xad, 0xbc, 0xe4, 0xbd,
      0xa0, 0xb3, 0xf7, 0xe3, 0x6c, 0x9d, 0xa7 }
  },
  { "abcdbcdecdefdefgefghfghighijh"
    "ijkijkljklmklmnlmnomnopnopq",
    { 0x75, 0x38, 0x8b, 0x16, 0x51, 0x27, 0x76,
      0xcc, 0x5d, 0xba, 0x5d, 0xa1, 0xfd, 0x89,
      0x01, 0x50, 0xb0, 0xc6, 0x45, 0x5c, 0xb4,
      0xf5, 0x8b, 0x19, 0x52, 0x52, 0x25, 0x25 }
  },
};
```

SHA-384 Design

Like SHA-224, SHA-384 was designed after an SHS algorithm. In this case, SHA-384 is based on SHA-512. Like SHA-224, we begin with a modified state.

```
S[0] = CONST64(0xcbbb9d5dc1059ed8);
S[1] = CONST64(0x629a292a367cd507);
S[2] = CONST64(0x9159015a3070dd17);
S[3] = CONST64(0x152fecd8f70e5939);
S[4] = CONST64(0x67332667ffc00b31);
S[5] = CONST64(0x8eb44a8768581511);
S[6] = CONST64(0xdb0c2e0d64f98fa7);
S[7] = CONST64(0x47b5481dbefa4fa4);
```

SHA-384 uses the SHA-512 algorithm to perform the actual message hashing. The output of SHA-384 is the output of SHA-512 truncated by copying the first 48 bytes of the output to the caller. SHA-384 is a useful choice of hash when working with the ECC-384 curve.

For completeness, here are the test vectors for SHA-384.

```
static const struct {
    char *msg;
    unsigned char hash[48];
} tests[] = {
  { "abc",
    { 0xcb, 0x00, 0x75, 0x3f, 0x45, 0xa3, 0x5e, 0x8b,
      0xb5, 0xa0, 0x3d, 0x69, 0x9a, 0xc6, 0x50, 0x07,
      0x27, 0x2c, 0x32, 0xab, 0x0e, 0xde, 0xd1, 0x63,
      0x1a, 0x8b, 0x60, 0x5a, 0x43, 0xff, 0x5b, 0xed,
      0x80, 0x86, 0x07, 0x2b, 0xa1, 0xe7, 0xcc, 0x23,
      0x58, 0xba, 0xec, 0xa1, 0x34, 0xc8, 0x25, 0xa7 }
  },
  { "abcdefghbcdefghicdefghijdefghi"
    "jkefghijklfghijklmghijklmnhijk"
    "lmnoijklmnopjklmnopqklmnopqrlm"
    "nopqrsmnopqrstnopqrstu",
    { 0x09, 0x33, 0x0c, 0x33, 0xf7, 0x11, 0x47, 0xe8,
      0x3d, 0x19, 0x2f, 0xc7, 0x82, 0xcd, 0x1b, 0x47,
      0x53, 0x11, 0x1b, 0x17, 0x3b, 0x3b, 0x05, 0xd2,
      0x2f, 0xa0, 0x80, 0x86, 0xe3, 0xb0, 0xf7, 0x12,
      0xfc, 0xc7, 0xc7, 0x1a, 0x55, 0x7e, 0x2d, 0xb9,
      0x66, 0xc3, 0xe9, 0xfa, 0x91, 0x74, 0x60, 0x39 }
  },
};
```

Zero-Copying Hashing

Earlier we alluded to a zero-copying technique to boost hash algorithm throughput. We shall now present this technique as applied to the SHA-1 *process* function. We shall only present the modified code to save space.

sha1zc.c:

```
static void sha1_compress(         sha1_state *md,
                          const unsigned char *buf)
{
   ulong32 W[80], a, b, c, d, e, t;
   unsigned x;

   /* load W[0..15] */
   for (x = 0; x < 16; x++) {
       LOAD32H(W[x], buf + 4 * x);
   }
```

We first have to modify the compression function to accept the message block from the caller, instead of implicitly from the SHA-1 state.

```
void sha1_process(         sha1_state *md,
                  const unsigned char *buf,
                        unsigned long  len)
{
   unsigned long x, y;

   while (len) {
       /* zero copy 64 byte chunks */
       while (len >= 64 && md->buflen == 0) {
          sha1_compress(md, buf);
          buf        += 64;
          md->msglen += 64;
          len        -= 64;
       }

       x    = (64 - md->buflen) < len ? 64 - md->buflen : len;
       len -= x;

       /* copy x bytes from buf to the buffer */
       for (y = 0; y < x; y++) {
           md->buf[md->buflen++] = *buf++;
       }

       if (md->buflen == 64) {
          sha1_compress(md, md->buf);
          md->buflen  = 0;
          md->msglen += 64;
       }
   }
}
```

This is our newly modified process function. Inside the inner loop, we perform the zero-copy optimization by passing 64-byte blocks directly to the compression function without first copying it to the internal state. We can only do this if the SHA-1 state is empty (buflen is zero) and there are at least 64 bytes of message bytes left to process.

This optimization may not seem like much on a processor like the AMD Opteron with its huge L1 data cache. However, on much more compact processors such as the ARM series, extra data movement over, say, a 16-MHz data bus can mean a real performance loss.

```
void sha1_done(   sha1_state *md,
               unsigned char *dst)
{
   ulong32 l1, l2, i;

   /* compute final length as 8*md->msglen */
   md->msglen += md->buflen;
   l2 = md->msglen >> 29;
   l1 = (md->msglen << 3) & 0xFFFFFFFF;

   /* add the padding bit */
   md->buf[md->buflen++] = 0x80;

   /* if the current len > 56 we have to finish this block */
   if (md->buflen > 56) {
      while (md->buflen < 64) {
          md->buf[md->buflen++] = 0x00;
      }
      sha1_compress(md, md->buf);
      md->buflen = 0;
   }

   /* now pad until we are at pos 56 */
   while (md->buflen < 56) {
       md->buf[md->buflen++] = 0x00;
   }

   /* store the length */
   STORE32H(l2, md->buf + 56);
   STORE32H(l1, md->buf + 60);

   /* compress */
   sha1_compress(md, md->buf);

   /* extract the state */
   for (i = 0; i < 5; i++) {
      STORE32H(md->S[i], dst + i*4);
   }
}
```

For completeness, this is the modified *done* function with the new calls to the compression function.

PKCS #5 Key Derivation

Hash functions can be used for many different problems, from integrity and authenticity (see Chapter 6, "Message Authentication Code Algorithms") to pseudo random number generation (see Chapter 3, "Random Number Generation") and key derivation. Now we shall explore the latter property.

Key derivation functions (KDF) derive key material from another source of entropy while preserving the entropy of the input and being one-way. Key derivation is often used for more than generating key material. It is also used to derive initial values (IV) and *nonces* (see Chapter 6) for cryptographic sessions. The typical usage of a key derivation function is to take a secret such as a password or a shared secret (see Chapter 9) and a salt to produce a key and IV. The salt is generated at random when the session is first created to prevent dictionary attacks. It's not as important when a random shared secret is being used, as a dictionary attack will not apply.

PKCS #5 (ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf) is a standard put forth by the former RSA Data Security Corporation as one of a series of standards of public key cryptography. It defines two KDF algorithms called PBKDF1 and PBKDF2. The former is an old standard that was originally meant to be used with DES, and as such has fixed size inputs. PBKDF2 is more flexible and the recommended KDF from the PKCS #5 standard

The algorithm we are going to present uses an algorithm known as HMAC (hash message authentication code), which we have not discussed yet. Confused readers are encouraged to read the discussion of HMAC in Chapter 6 first before coming back to this algorithm (Figure 5.6).

Figure 5.6 PKCS #5 PBKDF2 Algorithm

```
Input:
    secret:        The secret used as a master key to derive into a session key
    salt:          The random nonsecret string used to prevent dictionary attacks
    iterations     The number of iterations in the main loop
    w:             The digest size of the hash being used
    outlen:        The desired amount of KDF data requested
Output:
    out:           The KDF data

 1. for blkNo from 1 to ceil(outlen/w) do
     1. T = HMAC(secret, salt || blkNo)
     2. U = T
     3. for i from 1 to iterations do
         i.  T  = HMAC(secret, T)
         ii. U = U xor T
     4. out = out || U
 2. Truncate out to outlen bytes
```

The value of *blkNo* is appended to the salt as a 32-bit big endian number. The algorithm begins by computing the HMAC of the salt with the blkNo value appended. This gives us our starting value for this pass of the algorithm. We make a copy of this value into U and then repeatedly HMAC the value of T, XORing the output to U in each iteration. The purpose of the *iterations* count is to make dictionary and exhaustive search attacks even slower. For example, if it is set to 1024, we are effectively adding 10 bits to our secret key.

The data generated by this function is meant to be used as both cipher keys and session initial values such as IVs and nonces. For example, to use AES-128 in CTR mode, the caller would use this KDF to generate 32 bytes of data. The first 16 bytes could be the AES key, and the second 16 bytes could be the CTR initial value. If we used SHA-256 as the hash for the HMAC, we would loop only once, as at step 4 we would have generated the required 32 bytes. Had we used SHA-1, for example, we would have to loop twice, producing 40 bytes that would then be truncated to 32 bytes.

The function requires that the *secret* be unpredictable from an attacker. The *salt* should be random, but cannot be a secret, as the recipient will need to know it to generate the same session values. The *salt* can be any length. In practice, it should be no larger than *secret* and no smaller than eight bytes.

A curious reader may wish to use the often mistaken construction of hash (secret||data) as a message authentication like primitive. However, as we shall see in Chapter 6, this is not a secure construction and should be avoided. As it applies to PKCS #5, it may very well be secure; however, for the purposes of interoperability, one should choose to use a proper HMAC construction.

Putting It All Together

As we have seen, the SHS algorithms are easy to implement, and as we have hinted at, are convenient to have around. What we will now consider are more concrete examples of using hashes in fielded systems. What not to use them for, how to optimize them for space and time, and finally give a concrete example of using PKCS #5 with our previous AES CTR example (see Chapter 4).

It is easy to use a hash as the tool it was meant to be if we simply keep in mind it is a pseudo-random function (PRF). Just as how ciphers are pseudo random permutations (PRP) and have their own usage limitations, so do hashes.

What Hashes Are For

Hashes are pseudo-random functions. What this means is that they pseudo randomly map the input to the output. Unlike ciphers, which are PRPs, the input (domain) and output (co-domain) are not the same size. The typical hash algorithm can allow inputs as large as 2^{64}–1 bits and produce fixed size outputs of hundreds of bits. The pseudo-random mapping suggests one useful feature, one-wayness. It also suggests the capability of being collision resistant if the mapping is hard to control.

One-Wayness

A function is considered one-way if computing the function is trivial in one sense and nontrivial in the reverse sense. In terms of hash functions, this means that the message digest is trivial to compute. Finding the message given only the message digest is nontrivial.

Passwords

Working with user passwords and pass phrases is one example of using this property. Ideally, a network application that requires users to authenticate themselves will ask for a credential such as a password. Sending the password in the clear would mean attackers could read it. Similarly, a password list for a multi-user environment (e.g., school network) would need to store something to compare against user credentials. Hash algorithms provide a way of proving ownership of the credential without leaking the credential itself.

Random Number Generators

Another useful task that takes advantage of this property is random number generators, and the pseudo random variety. By hashing seeding data, the output does not leak the seed data the generator is privy to. This seed data often will contain private information such as network traffic or keystrokes.

Collision Resistance

Hashes must also be collision resistant to be cryptographically useful. Often, this property is exploited in the process of performing digital signatures (see Chapter 9). Typical public key signature algorithms are slower by magnitudes of cycles than the typical hash function. To optimize the signature process, a cryptosystem could choose to sign a hash of the message instead of the full message itself.

This procedure can be as effectively secure as signing the message, provided the hash algorithm has two properties. It must be collision resistant and it must produce a digest of respective size. Recall the discussion of the birthday paradox earlier in this chapter. If you are using a public key algorithm that takes, say, 2^{80} work ("work" often refers to the number of operations required; in this sense, work could mean public key operations or hash operations) to break, your hash should produce no less than 160 bits of message digest.

File Manifests

Collision resistance is useful for integrity tasks as well. In the most innocent of cases, computing the message digest of files offered for download helps ensure users download the file correctly. This assumes, of course, the absence of an attacker. There is an often cited defense that because the message digest would be stored on a secure server, the attackers would have to modify it there. This is not correct. An attacker would only have to get between the client and server to perform the attack. This is typically far easier than breaking into a secured server.

Intrusion Detection

Hashes are also in intrusion detection software (IDS), which scans for modifications to files as a sign of an intruder. They compute the message digest of executables and compare on the fly against the cached value. This works well provided the IDS software itself has not been exploited. However, one can never be too vigilant. In 2005, MD5 (the most common hash for this task) was broken. Various people, including Dan Kaminsky, championed the Trojan payload attack.

Their attack is both clever and effective. They produce a program that has two payloads attached. Then, they insert a block of data that will collide with MD5. They can then swap the block out depending on whether they want to activate the payload. The program works by first checking which block was included and acting differently accordingly. The two files would have the same MD5 message digest, but they would act very differently.

Many Linux distributions have moved to using multiple hashes as a result. Gentoo Linux, for example, checks MD5, RMD-160, and SHA-256 checksums on all files in their Portage repository. While this is at best as strong as a perfect 256-bit hash (we will not assume SHA-256 is perfect), it has the durability that in case SHA-256 is broken, at least checking the RMD-160 hash is a good fallback. ((RMD (also RIPEMD) stands for RIPE-Message Digest, where RIPE (RACE Integrity Primitives Evaluation) is a European standards process. The original RIPEMD algorithm was generally not considered strong and was broken in 2004. The RIPEMD-160, 256, and 320 algorithms are mostly based off of the MD4 design and are not competitively efficient.)

What Hashes Are Not For

We have seen some clear examples of what hashes can be safely used for. There is also a variety of tasks that crop up and are mistakenly used in fielded systems. Part of the problem stems from the fact that people think that hashes are universally magical random functions. They often forget that we can prefix data, append data, and pre-compute things.

Unsalted Passwords

This is one of the cases where a little salt is healthy (sorry, we could not resist the pun). As we will see in more detail shortly (we are almost there), password hashing is a tricky business. Suppose we want to store your password for future comparison.

Clearly, we cannot store the password on its own, as an attacker who can see the password list will own the system. The logical solution that pops to mind is to hash the passwords. After all, it is one-way. What could possibly go wrong?

Hashes Make Bad Ciphers

It's often tempting to turn a hash into a cipher. Hashes are actually ciphers in disguise if you look at them. For example, SHACAL is a 160-bit block cipher that uses SHA-1. It treats the W[] array as the cipher key and the state as the plaintext (or ciphertext). You can also just use a hash in CTR mode to create a stream cipher.

While technically sound and most likely secure, neither construction is part of a standard. They are also slower than block ciphers. For example, MD5 routinely clocks in at eight cycles per byte hashes on an AMD Opteron. That means eight cycles per byte of *input* not output. MD5 has 64-byte blocks; therefore, the compression takes at least 512 cycles, which would produce a CTR key stream at a rate of 32 cycles per byte. AES CTR requires slightly over 17 cycles per byte.

It gets no better for the other hashes. SHA-512 requires 12 cycles per byte of input on the Opteron, which translates into 24 cycles per byte of output. SHA-1 requires 18 cycles per byte on an Intel Pentium 4 (to give examples from other processors), which translates into 58 cycles per byte of output (over twice as slow as AES on this processor).

The lack of standards and the fact it is inefficient to do so makes ciphering with a hash a bad idea in general. About the only place it would make sense is where you are limited for software or hardware space.

Hashes Are Not MACs

Hashes are not message authentication code algorithms. The most common construction that is not actually secure is to make the MAC as follows.

tag := hash(*key* || *message*)

The attack takes advantage of the fact that an attacker does not have to start with the same initial value as the standard specifies. In the preceding case, we are effectively hashing the message blocks: key, message, MD-strengthening. This produces a tag that is effectively the hash state after hashing all the data. An attacker could append a block (and then another MD-strengthening block) and use the tag as the initial state. The computed hash would appear to be a valid message from a victim with the key.

The *fix* is often to use the following combination:

tag := hash(hash(*key* || *message*))

which does not leak the internal state of the inner hash. However, it violates one of the goals of MAC algorithms. It is attackable in an offline sense, as we will see in the discussion of HMAC in Chapter 6.

Hashes Don't Double

A common trick to double the size of the message digest is to concatenate two message digests of slightly different messages. For instance,

digest := hash(1 || *message*) || hash(2 || *message*)

The problem with this technique is that a collision in the first message block will immediately make any following blocks collide. For example, consider the input as 1 || *block* || *message*, where 1 || *block* fits in one message block in the hash (e.g., 64 bytes for SHA-1). If we can find a *block* for which hash(1 || *block*) equals hash(2 || *block*), we would only have to find two *message* values that collide through the remainder of the hash.

Hashes Don't Mingle

Failing the double-up trick, the next idea is to concatenate two message digests from different hashes. For instance,

digest := hash1(*message*) || hash2(*message*)

Unfortunately, this is no better than the strongest hash in the mix—at least not with the typical hash construction. Suppose M and M' collide in hash1. Then, M || Q and M' || Q will collide since their hash states collide after M (M', respectively). Thus, an attacker would only have to find a pair M and M' and proceed to attack the second hash. It is conceivable that two properly constructed hashes offer more security as a pair; however, with the common MD style hash this is not believed to be the case.

Working with Passwords

As we have mentioned, hashes are good when working with passwords. They do not allow determining the input from the output and are collision resistant. In effect, if users can produce the correct message digest, chances are they know the correct input.

Using this nice property securely is a tricky challenge. First, let us consider the offline world, and then we shall explore the online world.

Offline Passwords

Offline password verification is what you may find on an account login (say, locally to your machine). The goal is to compare what you produce against a *proof* stored locally. The proof is essentially producible only by someone in possession of the credential required.

In the offline world, we are typically dealing with a privileged process such as login (typical to UNIX-like platforms), so the proof generation process is tamper safe.

If we simply stored the hash of the password, we would prevent an attacker from directly finding out the password. Unfortunately, we do not stop a practical and often fruitful attack—the dictionary attack. Users tend to pick passwords and pass phrases that are easy to memorize. This means they pick dictionary words, and occasionally mix two or more together. A dictionary attack literally runs through a dictionary, hashes the generated strings, and compares it against the password list.

If we simply hash the passwords, if two or more users choose the same password this will immediately be revealed. What is worse is that an attacker could pre-compute a hash list (another use of the word *hash*) that associates hashes with passwords. The entire "attack" could take only seconds on the victim's machine.

Salts

The most common and satisfactory solution (letting aside educating the users) is to *salt* the passwords. Salting means we "add flavor" to the hash specific for the user. In terms of cryptography, it means we add random digits to the string we hash. Those random digits, the salt, become associated with the user for the given credential proof list. If a user moves to another system, he should have a different salt totally unrelated to the previous instance. What the salt prevents the attacker from doing is pre-computing a dictionary list. The attacker would to compute it only after learning the user's salt, and would have to re-compute it for every user he chooses to attack.

Salt Sizes

Now that we are salting our passwords, what size should the salt be? Typically, it does not have to be large. It should be unique for all the users in the given credential list. A safe guideline is to use salts no less than 8 bytes and no larger than 16 bytes. Even 8 bytes is overkill, but since it is not likely to hurt performance (in terms of storage space or computation time), it's a good low bound to use.

Technically, you need at least the square of the number of credentials you plan to store. For example, if your system is meant to accommodate 1000 users, you need a 20-bit salt. This is due to the birthday paradox.

Our suggestion of eight bytes would allow you to have slightly over four billion credentials in your list.

Rehash

Another common trick is to not use the hash output directly, but instead re-apply the hash to the hash output a certain number of times. For example:

$$\textit{proof} := \text{hash}(\text{hash}(\text{hash}(\text{hash}(\ldots(\text{hash}(\textit{salt}\,||\,\textit{password})))))\ldots)$$

While not highly scientific, it is a valid way of making dictionary attacks slower. If you apply the hash, say 1024 times, then you make a brute force search 1024 times harder. In practice, the user will not likely notice. For example, on an AMD Opteron, 1024 invocations of SHA-1 will take roughly 720,000 CPU cycles. At the average clock rate of 2.2GHz, this amounts to a mere 0.32 milliseconds. This technique is used by PKCS #5 for the same purpose.

Online Passwords

Online password checking is a different problem from the offline word. Here we are not privileged, and attackers can intercept and modify packets between the client and server.

The most important first step is to establish an anonymous secure session. An SSL session between the client and server is a good example. This makes password checking much like the offline case. Various protocols such as IKE and SRP (Secure Remote Passwords: http://srp.stanford.edu/) achieve both password authentication and channel security (see Chapter 9).

In the absence of such solutions, it is best to use a challenge-response scheme on the password. The basic challenge response works by having the server send a random string to the client. The client then must produce the message digest of the password and challenge to pass the test. It is important to always use random challenges to prevent replay attacks. This approach is still vulnerable to meet in the middle attacks and is not a safe solution.

Two-Factor Authentication

Two-factor authentication is a user verification methodology where multiple (at least two in this case) *different* forms of credentials are used for the authentication process.

A very popular implementation of this are the RSA SecurID tokens. They are small, keychain size computers with a six-to-eight digit LCD. The computer has been keyed to a given user ID. Every minute, it produces a new number on the LCD that only the token and server will now. The purpose of this device is to make guessing the password insufficient to break the system.

Effectively, the device is producing a hash of a secret (which the server knows) and time. The server must compensate for drift (by allowing values in the previous, current, and next minutes) over the network, but is otherwise trivial to develop.

Performance Considerations

Hashes typically do not use as many table lookups or complicated operations as the typical block cipher. This makes implementation for performance (or space) a rather nice and short job.

All three (distinct) algorithms in the SHS portfolio are subject to the same performance tweaks.

Inline Expansion

The expanded values (the W[] arrays) do not have to be fully computed before compression. In each case, only 16 of the values are required at any given time. This means we can save memory by only storing them and compute 16 new expanded values as required.

In the case of SHA-1, this saves 256 bytes; SHA-256 saves 192 bytes; and SHA-512 saves 512 bytes of memory by using this trick.

Compression Unrolling

All three algorithms employ a shift register like construction. In a fully rolled loop, this requires us to manually shift data from one word to another. However, if we fully unroll the loops, we can perform *renaming* to avoid the shifts. All three algorithms have a round count that is a multiple of the number of words in the state. This means we always finish the compression with the words in the same spot they started in.

In the case of SHA-1, we can unroll each of the four groups either 5-fold or the full 20-fold. Depending on the platform, the performance gains of 20-fold can be positive or negative over the 5-fold unrolling. On most desktops, it is not faster, or faster by a large enough margin to be worth it.

In SHA-256 and SHA-512, loop unrolling can proceed at either the 8-fold or the full 64-fold (80, resp.) steps. Since SHA-256 and SHA-512 are a bit more complicated than SHA-1, the benefits differ in terms of unrolling. On the Opteron, process unrolling SHA-256 fully usually pays off better than 8-fold, whereas SHA-512 is usually better off unrolled only 8-fold.

Unrolling in the latter hashes also means the possibility of embedding the round constants (the K[] array) into the code instead of performing a table lookup. This pays off less on platforms like the ARM, which cannot embed 32-bit (or 64-bit for that matter) constants in the instruction flow.

Zero-Copy Hashing

Another useful optimization is to zero-copy the data we are hashing. This optimization basically loads the message block directly from the user-passed data instead of buffering it internally. This hash is most important on platforms with little to no cache. Data in these cases is usually going over a relatively slower data bus, often competing for system devices for traffic.

For example, if a 32-bit load or store requires (say) six cycles, which is typical for the average low power embedded device, then storing a message block will take 96 cycles. A compression may only take 1000 to 2000 cycles, so we are adding between 4.5% and 9 percent more cycles to the operation that we do not have to.

This optimization usually adds little to the code size and gives us a cheap boost in performance.

PKCS #5 Example

We are now going to consider the example of AES CTR from Chapter 4. The reader may be a bit upset at the comment "somehow fill secretkey and IV ..." found in the code with that section missing. We now show one way to fill it in.

The reader should keep in mind that we are putting in a *dummy* password to make the example work. In practice, you would fetch the password from the user, or by first turning off the console echo and so on.

Our example again uses the LibTomCrypt library. This library also provides a nice and handy PKCS #5 function that in one call produces the output from the secret and salt.

pkcs5ex.c:

```
#include <tomcrypt.h>

void dumpbuf(const unsigned char *buf,
                   unsigned long  len,
                   unsigned char *name)
{
   unsigned long i;
   printf("%20s[0...%3lu] = ",name, len-1);
   for (i = 0; i < len; i++)  {
      printf("%02x ", *buf++);
   }
   printf("\n");
}
```

This is a handy debugging function for dumping arrays. Often in cryptographic protocols, it is useful to see intermediate outputs before the final output. In particular, in multistep protocols, it will let us debug at what point we deviated from the test vectors. That is, provided the test vectors list such things.

```
int main(void)
{
   symmetric_CTR ctr;
   unsigned char secretkey[16], IV[16], plaintext[32],
                 ciphertext[32], buf[32], salt[8];
   int           x;
   unsigned long buflen;
```

Similar list of variables from the CTR example. Note we now have a salt[] array and a buflen integer.

```
   /* setup LibTomCrypt */
   register_cipher(&aes_desc);
   register_hash(&sha256_desc);
```

Now we have registered SHA-256 in the crypto library. This allows us to use SHA-256 by name in the various functions (such as PKCS #5 in this case). Part of the benefit of the LibTomCrypt approach is that many functions are agnostic to which cipher, hash, or other function they are actually using. Our PKCS #5 example would work just as easily with SHA-1, SHA-256, or even the Whirlpool hash functions.

```
   /* somehow fill secretkey and IV ... */
      /* read a salt */
      rng_get_bytes(salt, 8, NULL);
```

In this case, we read the RNG instead of setting up a PRNG. Since we are only reading eight bytes, this is not likely to block on Linux or BSD setups. In Windows, it will never block.

```
      /* invoke PKCS #5 on our password "passwd" */
      buflen = sizeof(buf);
      assert(pkcs_5_alg2("passwd", 6,
                         salt, 8,
                         1024, find_hash("sha256"),
                         buf, &buflen) == CRYPT_OK);
```

This function call invokes PKCS #5. We pass the *dummy* password "passwd" instead of a properly entered one from the user. Please note that this is just an example and not the type of password scheme you should employ in your application.

The next line specifies our salt and its length—in this case, eight bytes. Follow by the number of iterations desired. We picked 1024 simply because it's a nice round nontrivial number.

The find_hash() function call may be new to some readers unfamiliar with the LibTomCrypt library. This function searches the tables of registered hashes for the entry matching the name provided. It returns an integer that is an index into the table. The function (PKCS #5 in this case) can then use this index to invoke the hash algorithm.

The tables LibTomCrypt uses are actually an array of a C "struct" type, which contains pointers to functions and other parameters. The functions pointed to implement the given hash in question. This allows the calling routine to essentially support any hash without having been designed around it first.

The last line of the function call specifies where to store it and how much data to read. LibTomCrypt uses a "caller specified" size for buffers. This means the caller must first say the size of the buffer (in the pointer to an unsigned long), and then the function will update it with the number of bytes stored.

This will become useful in the public key and ASN.1 function calls, as callers do not always know the final output size, but do know the size of the buffer they are passing.

```
      /* copy out the key and IV */
      memcpy(secretkey, buf,    16);
      memcpy(IV,        buf+16, 16);
```

At this point, buf[0..31] contains 32 pseudo random bytes derived from our password and salt. We copy the first 16 bytes as the secret key and the second 16 bytes as the IV for the CTR mode.

```
   /* start CTR mode */
   assert(
     ctr_start(find_cipher("aes"), IV, secretkey, 16, 0,
               CTR_COUNTER_BIG_ENDIAN, &ctr) == CRYPT_OK);

   /* create a plaintext */
   memset(plaintext, 0, sizeof(plaintext));
   strncpy(plaintext, "hello world how are you?",
           sizeof(plaintext));

   /* encrypt it */
   ctr_encrypt(plaintext, ciphertext, 32, &ctr);

   printf("We give out salt and ciphertext as the 'output'\n");
   dumpbuf(salt, 8, "salt");
   dumpbuf(ciphertext, 32, "ciphertext");

   /* reset the IV */
   ctr_setiv(IV, 16, &ctr);

   /* decrypt it */
   ctr_decrypt(ciphertext, buf, 32, &ctr);

   /* print it */
   for (x = 0; x < 32; x++) printf("%c", buf[x]);
   printf("\n");

   return EXIT_SUCCESS;
}
```

The example can be built, provided LibTomCrypt has already been installed, with the following command.

```
gcc pkcs5ex.c -ltomcrypt -o pkcs5ex
```

The example output would resemble the following.

```
We give out salt and ciphertext as the 'output'

              salt[0...  7] = 58 56 52 f6 9c 04 b5 72
          ciphertext[0... 31] = e2 3f be 1f 1a 0c f8 96 0c e5 50 04 c0 a8 f7 f0
c4 27 60 ff b5 be bb bc f4 dc 88 ec 0e 0a f4 e6
hello world how are you?
```

Each run should choose a different salt and respectively produce a different ciphertext. As the demonstration states, we would only have to be given the salt and ciphertext to be able to decrypt it (provided we knew the password). We do not have to send the IV bytes since they are derived from the PKCS #5 algorithm.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to www.syngress.com/solutions and click on the "Ask the Author" form.

Q: What is a hash function?

A: A hash function accepts as input an arbitrary length string of bits and produces as output a fixed size string of bits known as the message digest. The goal of a cryptographic hash function is to perform the mapping as if the function were a random function.

Q: What is a message digest?

A: A message digest is the output of a hash function. Usually, it is interpreted as a representative of the message.

Q: What does one-way and collision resistant mean?

A: A function that is one-way implies that determining the output given the input is a hard problem to solve. In this case, given a message digest, finding the input should be hard. An ideal hash function is one-way. Collision resistant implies that finding pairs of unique inputs that produce the same message digest is a hard problem. There are two forms of collision resistance. The first is called pre-image collision resistance, which implies given a fixed message we cannot find another message that collides with it. The second is simply called second pre-image collision resistance and implies that finding two random messages that collide is a hard problem.

Q: What are hash functions used for?

A: Hash functions form what are known as Pseudo Random Functions (PRFs). That is, the mapping from input to output is indistinguishable from a random function. Being a PRF, a hash function can be used for integrity purposes. Including a message digest with an archive is the most direct way of using a hash. Hashes can also be used to create message authentication codes (see Chapter 6) such as HMAC. Hashes can also be used to collect entropy for RNG and PRNG designs, and to produce the actual output from the PRNG designs.

Q: What standards are there?

A: Currently, NIST only specifies SHA-1 and the SHA-2 series of hash algorithms as standards. There are other hashes (usually unfortunately) in wide deployment such as MD4 and MD5, both of which are currently considered broken. The NESSIE process in Europe has provided the Whirlpool hash, which competes with SHA-512.

Q: Where can I find implementations of these hashes?

A: LibTomCrypt currently supports all NIST standard hashes (including the newer SHA-224), and the NESSIE specifies Whirlpool hash. LibTomCrypt also supports the older hash algorithms such as RIPEMD, MD2, MD4, and so on, but generally users are warned to avoid them unless they are trying to implement an older standard (such as the NT hash). OpenSSL supports SHA-1 and RIPEMD, and Crypto++ supports a variety of hashes including the NIST standards.

Q: What are the patent claims on these hashes?

A: SHA-0 (the original SHA) was patented by the NSA, but irrevocably released to the public for all purposes. SHA-2 series and Whirlpool are both public domain and free for all purposes.

Q: What length of digest should I use? What is the birthday paradox?

A: In general, you should use twice the number of bits in your message digest as the target *bit strength* you are looking for. If, for example, you want an attacker to spend no less than 2^{128} work breaking your cryptography, you should use a hash that produces at least a 256-bit message digest. This is a result of the birthday paradox, which states that given roughly the square root of the message digest's domain size of outputs, one can find a collision. For example, with a 256-bit message digest, there are 2^{256} possible outcomes. The square root of this is 2^{128}, and given 2^{128} pairs of inputs and outputs from the hash function, an attacker has a good probability of finding a collision among the entries of the set.

Q: What is MD strengthening?

A: MD (Message Digest) strengthening is a technique of padding a message with an encoding of the message length to avoid various prefix and extension attacks.

Q: What is key derivation?

A: Key derivation is the process of taking a shared secret key and producing from it various secret and public materials to secure a communication session. For instance, two parties could agree on a secret key and then pass that to a key derivation function to produce keys for encryption, authentication, and the various IV parameters. Key derivation is preferable over using shared secrets directly, as it requires sharing fewer bits and also mitigates the damages of key discovery. For example, if an attacker learns your authentication key, he should not learn your encryption key.

Q: What is PKCS #5?

A: PKCS #5 is the RSA Security Public Key Cryptographic Standard that addresses password-based encryption. In particular, their updated and revised algorithm PBEKDF2 (also known as PKCS #5 Alg2) accepts a secret salt and then expands it to any length required by the user. It is very useful for deriving session keys and IVs from a single (shorter) shared secret. Despite the fact that the standard was meant for password-based cryptography, it can also be used for randomly generated shared secrets typical of public key negotiation algorithms.

Chapter 6

Message - Authentication Code Algorithms

Solutions in this chapter:

- What Are MAC Functions?
- Purpose of a MAC
- Security Guidelines
- Standards
- CMAC Algorithm
- HMAC Algorithm
- Putting It All Together

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

Message Authentication Code (MAC) algorithms are a fairly crucial component of most online protocols. They ensure the *authenticity* of the message between two or more parties to the transaction. As important as MAC algorithms are, they are often overlooked in the design of cryptosystems.

A typical mistake is to focus solely on the privacy of the message and disregard the implications of a message modification (whether by transmission error or malicious attacker).

An even more common mistake is for people to not realize they need them. Many people new to the field assume that not being sure of the contents of a message means you cannot change it. The logic goes, "if they have no idea what is in my message, how can they possibly introduce a useful change?"

The error in the logic is the first assumption. Generally, an attacker can get a very good idea of the rough content of your message, and this knowledge is more than enough to mess with the message in a meaningful way. To illustrate this, consider a very simple banking protocol. You pass a transaction to the bank for authorization and the bank sends a single bit back: 0 for declined, 1 for a successful transaction.

If the transmission isn't authenticated and you can change messages on the communication line, you can cause all kinds of trouble. You could send fake credentials to the merchant that the bank would duly reject, but since you know the message is going to be a rejection, you could change the encrypted zero the bank sends back to a one—just by flipping the value of the bit. It's these types of attacks that MACs are designed to stop.

MAC algorithms work in much the same context as symmetric ciphers. They are fixed algorithms that accept a secret key that controls the mapping from input to the output (typically called the *tag*). However, MAC algorithms do not perform the mapping on a fixed input size basis; in this regard, they are also like hash functions, which leads to confusion for beginners.

Although MAC functions accept arbitrary large inputs and produce a fixed size output, they are not equivalent to hash functions in terms of security. MAC functions with fixed keys are often not secure one-way hash functions. Similarly, one-way functions are not secure MAC functions (unless special care is taken).

Purpose of A MAC Function

The goal of a MAC is to ensure that two (or more) parties, who share a secret key, can communicate with the ability (in all likelihood) to detect modifications to the message in transit. This prevents an attacker from modifying the message to obtain undesirable outcomes as discussed previously.

MAC algorithms accomplish this by accepting as input the message and secret key and producing a fixed size MAC *tag*. The message and tag are transmitted to the other party, who can then re-compute the tag and compare it against the tag that was transmitted. If they match, the message is almost certainly correct. Otherwise, the message is incorrect and should be ignored, or drop the connection, as it is likely being tampered with, depending on the circumstances.

For an attacker to forge a message, he would be required to break the MAC function. This is obviously not an easy thing to do. Really, you want it be just as hard as breaking the cipher that protects the secrecy of the message.

Usually for reasons of efficiency, protocols will divide long messages into smaller pieces that are independently authenticated. This raises all sorts of problems such as replay attacks. Near the end of this chapter, we will discuss protocol design criteria when using MAC algorithms. Simply put, it is *not* sufficient to merely throw a properly keyed MAC algorithm to authenticate a stream of messages. The protocol is just as important.

Security Guidelines

The security goals of a MAC algorithm are different from those of a one-way hash function. Here, instead of trying to ensure the integrity of a message, we are trying to establish the authenticity. These are distinct goals, but they share a lot of common ground. In both cases, we are trying to determine correctness, or more specifically the purity of a message. Where the concepts differ is that the goal of authenticity tries also to establish an origin for the message.

For example, if I tell you the SHA-1 message digest of a file is the 160-bit string *X* and then give you the file, or better yet, you retrieve the file yourself, then you can determine if the file is original (unmodified) if the computed message digest matches what you were given. You will not know who made the file; the message digest will not tell you that. Now suppose we are in the middle of communicating, and we both have a shared secret key *K*. If I send you a file and the MAC tag produced with the key *K*, you can verify if the message originated from my side of the channel by verifying the MAC tag.

Another way MAC and hash functions differ is in the notion of their bit security. Recall from Chapter 5, "Hash Functions," that a birthday attack reduces the bit security strength of a hash to half the digest size. For example, it takes 2^{128} work to find collisions in SHA-256. This is possible because message digests can be computed *offline*, which allows an attacker to pre-compute a huge dictionary of message digests without involving the victim. MAC algorithms, on the other hand, are *online* only. Without access to the key, collisions are not possible to find (if the MAC is indeed secure), and the attacker cannot arbitrarily compute tags without somehow tricking the victim into producing them for him.

As a result, the common line of thinking is that birthday attacks do not apply to MAC functions. That is, if a MAC tag is *k*-bits long, it should take roughly 2^k work to find a collision to that specific value. Often, you will see protocols that greatly truncated the MAC tag length, to exploit this property of MAC functions.

IPsec, for instance, can use 96-bit tags. This is a safe optimization to make, since the bit security is still very high at 2^{96} work to produce a forgery.

MAC Key Lifespan

The security of a MAC depends on more than just on the tag length. Given a single message and its tag, the length of the tag determines the probability of creating a forgery. However, as the secret key is used to authenticate more and more messages, the advantage—that is, the probability of a successful forgery—rises.

Roughly speaking, for example, for MACs based on block ciphers the probability of a forgery is 0.5 after hitting the birthday paradox limit. That is, after 2^{64} blocks, with AES an attacker has an even chance of forging a message (that's still 512 exabytes of data, a truly stupendous quantity of information).

For this reason, we must think of security not from the ideal tag length point of view, but the probability of forgery. This sets the upper bound on our MAC key lifespan. Fortunately for us, we do not need a very low probability to remain secure. For instance, with a probability of 2^{-40} of forgery, an attacker would have to guess the correct tag (or contents to match a fixed tag) on his first try. This alone means that MAC key lifespan is probably more of an academic discussion than anything we need to worry about in a deployed system

Even though we may not need a very low probability of forgery, this does not mean we should truncate the tag. The probability of forgery only rises as you authenticate more and more data. In effect, truncating the tag would save you space, but throw away security at the same time. For short messages, the attacker has learned virtually nothing required to compute forgeries and would rely on the probability of a random collision for his attack vector on the MAC.

Standards

To help developers implement interoperable MAC functions in their products, NIST has standardized two different forms of MAC functions. The first to be developed was the hash-based HMAC (FIPS 198), which described a method of *safely* turning a one-way collision resistant hash into a MAC function. Although HMAC was originally intended to be used with SHA-1, it is appropriate to use it with other hash function. (Recent results show that collision resistance is not required for the security of NMAC, the algorithm from which HMAC was derived (http://eprint.iacr.org/2006/043.pdf for more details). However, another paper (http://eprint.iacr.org/2006/187.pdf) suggests that the hash has to behave securely regardless.)

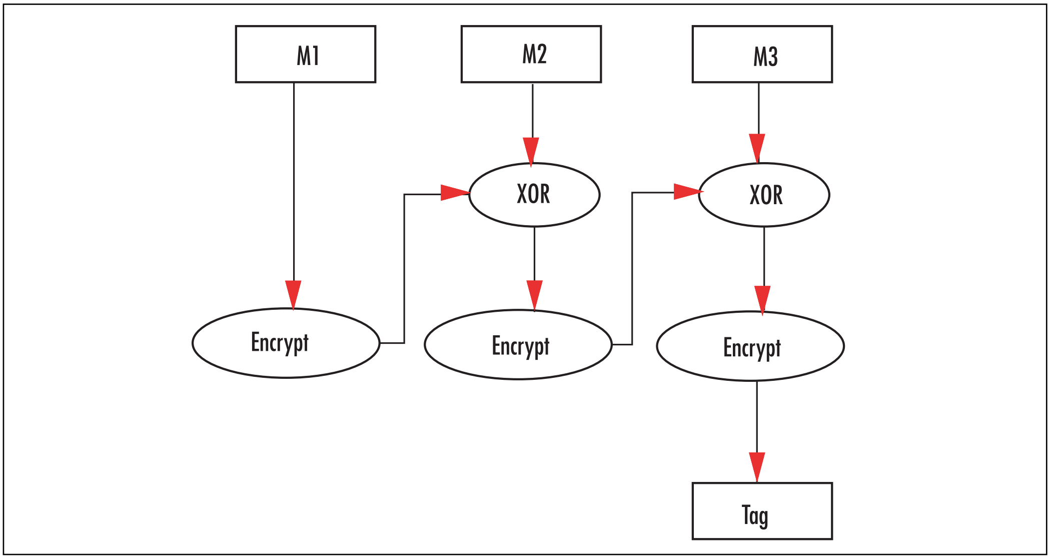
The second standard developed by NIST was the CMAC (SP 800-38B) standard. Oddly enough, CMAC falls under "modes of operations" on the NIST Web site and not a message authentication code. That discrepancy aside, CMAC is intended for message authenticity. Unlike HMAC, CMAC uses a block cipher to perform the MAC function and is ideal in space-limited situations where only a cipher will fit.

Cipher Message Authentication Code

The cipher message authentication code (CMAC, SP 800-38B) algorithm is actually taken from a proposal called OMAC, which stands for "One-Key Message Authentication Code" and is historically based off the three-key cipher block chaining MAC. The original cipher-based MAC proposed by NIST was informally known as CBC-MAC.

In the CBC-MAC design, the sender simply chooses an independent key (not easily related to the encryption key) and proceeds to encrypt the data in CBC mode. The sender discards all intermediate ciphertexts except for the last, which is the MAC tag. Provided the key used for the CBC-MAC is not the same (or related to) the key used to encrypt the plaintext, the MAC is secure (Figure 6.1).

Figure 6.1 CBC-MAC



That is, for all fixed length messages under the same key. When the messages are packets of varying lengths, the scheme becomes insecure and forgeries are possible; specifically, when messages are not an even multiple of the cipher's block length.

The fix to this problem came in the form of XCBC, which used three keys. One key would be used for the cipher to encrypt the data in CBC-MAC mode. The other two would be XOR'ed against the last message block depending on whether it was complete. Specifically, if the last block was complete, the second key would be used; otherwise, the block was padded and the third key used.

The problem with XCBC was that the proof of security, at least originally, required three totally independent keys. While trivial to provide with a key derivation function such as PKCS #5, the keys were not always easy to supply.

After XCBC mode came TMAC, which used two keys. It worked similarly to XCBC, with the exception that the third key would be linearly derived from the first. They did trade some security for flexibility. In the same stride, OMAC is a revision of TMAC that uses a single key (Figures 6.2 and 6.3).

Figure 6.2 OMAC Whole Block Messages

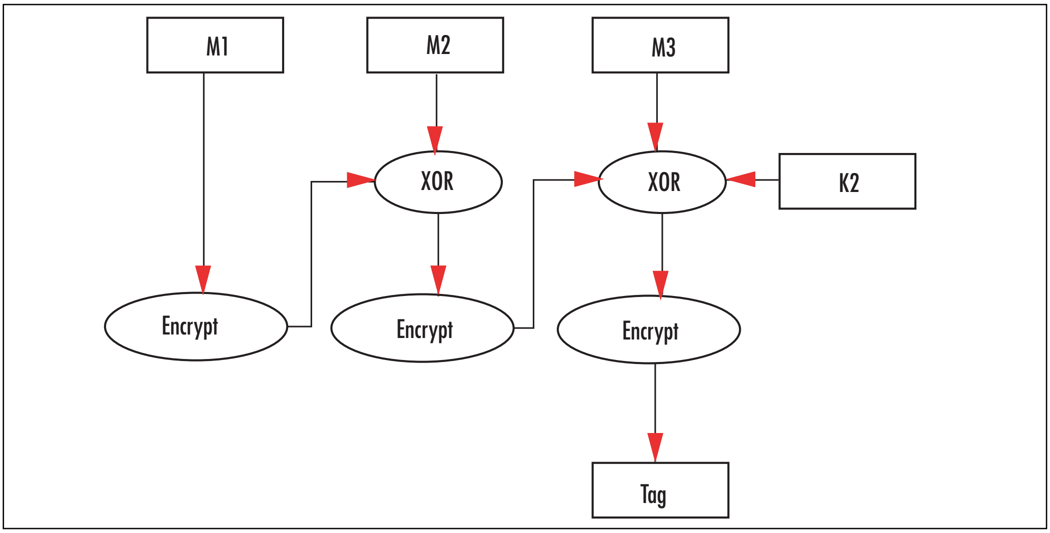
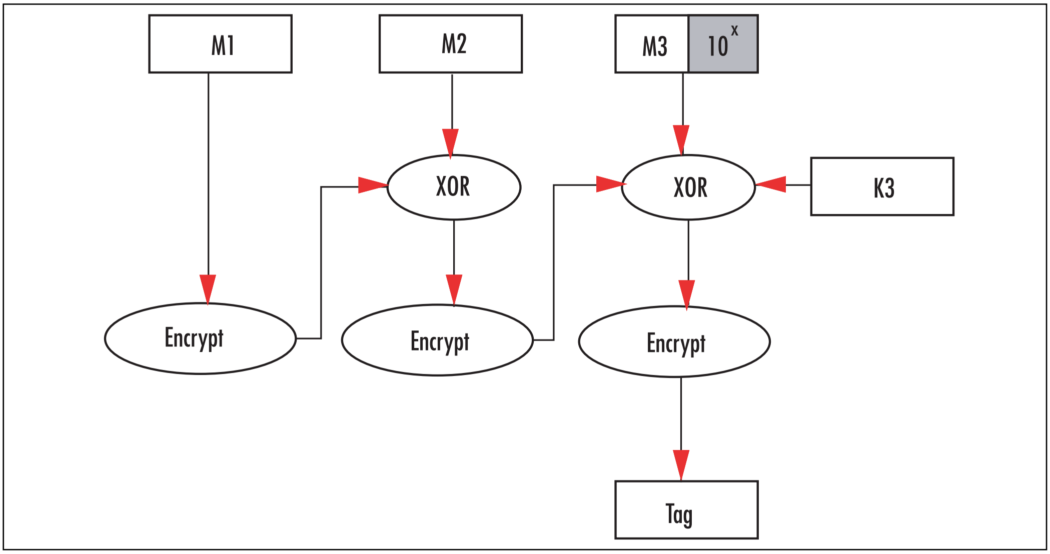


Figure 6.3 OMAC Partial Block Messages



Security of CMAC

To make these functions easier to use, they made the keys dependent on one another. This falls victim to the fact that if an attacker learns one key, he knows the others (or all of them in the case of OMAC). We say the *advantage* of an attacker is the probability that his forgery will succeed after witnessing a given number of MAC tags being produced.

1. Let Adv^{MAC} represent the probability of a MAC forgery.
2. Let Adv^{PRP} represent the probability of distinguishing the cipher from a random permutation.
3. Let *t* represent the time the attacker spends.
4. Let *q* represent the number of MAC tags the attacker has seen (with the corresponding inputs).
5. Let *n* represent the size of the block cipher in bits.
6. Let *m* represent the (average) number of blocks per message authenticated.

The advantage of an attacker against OMAC is then (roughly) no more than:

$$\text{Adv}^{\text{OMAC}} < (mq)^2/2^{n-2} + \text{Adv}^{\text{PRP}}(t + O(mq), mq + 1)$$

Assuming that mq is much less than $2^{n/2}$, then AdvPRP() is essentially zero. This leaves us with the left-hand side of the equation. This effectively gives us a limit on the CMAC algorithm. Suppose we use AES (n = 128), and that we want a probability of forgery of no more than 2^{-96}. This means that we need

$$2^{-96} > (mq)^2/2^{126}$$

If we simplify that, we obtain the result

$$2^{30} > (mq)^2$$
$$2^{15} > mq$$

What this means is that we can process no more than 2^{15} blocks with the same key, while keeping a probability of forgery below 2^{-96}. This limit seems a bit too strict, as it means we can only authenticate 512 kilobytes before having to change the key. Recall from our previous discussion on MAC security that we do not need such strict requirements. The attacker need only fail once before the attack is detected. Suppose we use the upper bound of 2^{-40} instead. This means we have the following limits:

$$2^{-40} > (mq)^2/2^{12}$$
$$2^{86} > (mq)^2$$
$$2^{43} > mq$$

This means we can authenticate 2^{43} blocks (1024 terabytes) before changing the key. An attacker having seen all of that traffic would have a probability of 2^{-40} of forging a packet, which is fairly safe to say not going to happen. Of course, this does not mean that one should use the same key for that length of traffic.

Notes from the Underground…

Online versus Offline Attacks

It is important to understand the distinction between online and offline attack vectors. Why is 40 bits enough for a MAC and not for a cipher key?

In the case of a MAC function, the attacks are *online*. That is, the attacker has to engage the victim and stimulate him to give information. We call the victim an oracle in traditional cryptographic literature. Since all traffic should be authenticated, an attacker cannot easily query the device. However, he may see known data fed to the MAC. In any event, the attack on the MAC is online. The attacker has only one shot to forge a message without being detected. A sufficiently low probability of success such as 2^{-40} means that you can safely mitigate that issue.

In the case of a cipher, the attacks are *offline*. The attacker can repeatedly perform a given computation (such as decryption with a random key) without involving the victim. A 40-bit key in this sense would provide barely any lasting security at all. For instance, an AMD Opteron can test an AES-128 key in roughly 2,000 processor cycles. Suppose you used a 40-bit key by zeroing the other 88 bits. A 2.2-GHz Opteron would require 11.6 days to find the key. A fully complemented AMD Opteron 885 setup (four processors, eight cores total at 2.6 GHz) could accomplish the goal in roughly 1.23 days for a cost less than $20,000.

It gets even worse in custom hardware. A pipelined AES-128 engine could test one key per cycle, and depending on the FPGA and to a larger degree the expected composition of the plaintext (e.g., ASCII) at rates approaching 100 MHz. That turns into a search time of roughly three hours. Of course, this is a bit simplistic, since any reasonably fast filtering on the keys will have many false positives. A secondary (and slower) screening process would be required for them. However, it too can work in parallel and since there are fewer false positives than keys, to test would not become much of a bottleneck.

Clearly, in the offline sense, bit security matters much more.

CMAC Design

CMAC is based off the OMAC design; more specifically, off the OMAC1 design. The designer of OMAC designed two very related proposals. OMAC1 and OMAC2 differ only in how the two additional keys are generated. In practice, people should only use OMAC1 if they intend to comply with the CMAC standard.

CMAC Initialization

CMAC accepts as input during initialization a secret key *K*. It uses this key to generate two additional keys *K1* and *K2*. Formally, CMAC uses a multiplication by p(x) = x in a GF(2)[x]/v(x) field to accomplish the key generation. Fortunately, there is a much easier way to explain it (Figure 6.4).

Figure 6.4 CMAC Initialization

```
Input
    K:              Secret key
Output
    K1, K2:         Additional CMAC keys

1. $L = \text{Encrypt}_K(0)$
2. If $\text{MSB}(L) = 0$, then $K1 = L << 1$
   else $K1 = (L << 1)$ XOR $R_b$
3. If $\text{MSB}(K1) = 0$, then $K2 = K1 << 1$
   else $K2 = (K1 << 1)$ XOR $R_b$
4. Return $K1, K2$
```

The values are interpreted in big endian fashion, and the operations are all on either 64- or 128-bit strings depending on the block size of the block cipher being used. The value of R_b depends on the block size. It is 0x87 for 128-bit block ciphers and 0x1B for 64-bit block ciphers. The value of L is the encryption of the all zero string with the key *K*.

Now that we have K1 and K2, we can proceed with the MAC. It is important to keep in mind that K1 and K2 must remain secret. Treat them as you would a ciphering key.

CMAC Processing

From the description, it seems that CMAC is only useful for packets where you know the length in advance. However, since the only deviations occur on the last block, it is possible to implement CMAC as a streaming MAC function without advanced knowledge of the data size. For zero length messages, CMAC treats them as incomplete blocks (Figure 6.5).

Figure 6.5 CMAC Processing

```
Input
    $K$:            Secret Key
    $K1$, $K2$:     Additional CMAC keys
    $M$:            Message
    $L$:            Number of bits in the message
    $Tlen$:         Desired length of the MAC tag
    $w$:            Bits per block
Output
    $T$:            The tag

1. If $L = 0$, let $n = 1$, else $n = \text{ceil}(L/w)$
2. Let $M_1, M_2, M_3, \ldots, M_n$ represent the blocks of the message.
3. If $L > 0$ and $L \bmod w = 0$ then
    1. $M_n := M_n$ XOR $K1$
4. if $L = 0$ or $L \bmod w > 0$ then
    1. Append a '1' bit then enough '0' bits to fill $w$ bits
    2. $M_n := M_n$ XOR $K2$
5. $C_0 = 0$
6. for $i$ from 1 to $n$ do
    1. $C_i = \text{Encrypt}_K(C_{i-1}$ XOR $M_i)$
7. $T = \text{MSB}_{Tlen}(C_n)$
8. Return $T$
```

It may look tempting to give out C_i values as ciphertext for your message. However, that invalidates the proof of security for CMAC. You will have to encrypt your plaintext with a different (unrelated) key to maintain the proof of security for CMAC.

CMAC Implementation

Our implementation of CMAC has been hardcode to use the AES routines of Chapter 4 with 128-bit keys. CMAC is not limited to such decisions, but to better demonstrate the MAC we decided to simplify it. The CMAC routines in LibTomCrypt (under the OMAC directory) demonstrate how to write a very flexible CMAC routine that can accept any 64- or 128-bit block cipher.

cmac.c:

```
/* poor linker for AES code */
#include "aes_large_mod.c"
```

We copied the AES code to our directory for Chapter 6. At this stage, we want to keep the code simple, so to this end, we simply include the AES code directly in our application.

Obviously, in the field the best practice would be to write an AES header and link the two files against each other properly.

```
typedef struct {
   unsigned char L[2][16],
                  C[16];
   ulong32        AESkey[15*4];
   unsigned       buflen;
   int            first;
} cmac_state;
```

This is our CMAC state function. Our implementation will process the CMAC message as a stream instead of a fixed sized block. The *L* array holds our two keys *K1* and *K2*, which we compute in the cmac_init() function. The *C* array holds the CBC chaining block. We buffer the message into the C array by XOR'ing the message against it. The *buflen* integer counts the number of bytes pending to be sent through the cipher.

```
void cmac_init(const unsigned char *key, cmac_state *cmac)
{
   int i, m;
```

This function initializes our CMAC state. It has been hard coded to use 128-bit AES keys.

```
   /* schedule the key */
   ScheduleKey(key, 16, cmac->AESkey);
```

First, we schedule the input key to the array in the CMAC state. This allows us to invoke the cipher on demand throughout the rest of the algorithm.

```
   /* encrypt 0 byte string */
   for (i = 0; i < 16; i++) {
      cmac->L[0][i] = 0;
   }
   AesEncrypt(cmac->L[0], cmac->L[0], cmac->AESkey, 10);
```

At this point, our L[0] array (equivalent to K1) contains the encryption of the zero byte string. We will multiply this by the polynomial *x* next to compute the final value of K1.

```
   /* now compute K1 and K2 */
      /* multiply K1 by x */
      m = cmac->L[0][0] & 0x80 ? 1 : 0;

      /* shift */
      for (i = 0; i < 15; i++) {
         cmac->L[0][i] = ((cmac->L[0][i] << 1) |
                          (cmac->L[0][i+1] >> 7)) & 255;
      }
      cmac->L[0][15] = (cmac->L[0][15] << 1) ^ (m ? 0x87 : 0);
```

We first grab the MSB of L[0] (into *m*) and then proceed with the left shift. The shift is equivalent to a multiplication by *x*. The last byte is shifted on its own and the value of 0x87 XORed in if the MSB was nonzero.

```
      /* multiple K2 by x */
      for (i = 0; i < 16; i++) {
          cmac->L[1][i] = cmac->L[0][i];
      }
      m = cmac->L[1][0] & 0x80 ? 1 : 0;

      /* shift */
      for (i = 0; i < 15; i++) {
         cmac->L[1][i] = ((cmac->L[1][i] << 1) |
                          (cmac->L[1][i+1] >> 7)) & 255;
      }
      cmac->L[1][15] = (cmac->L[1][15] << 1) ^ (m ? 0x87 : 0);
```

This copies L[0] (K1) into L[1] (K2) and performs the multiplication by *x* again. At this point, we have both additional keys required to process the message with CMAC.

```
    /* setup buffer */
    cmac->buflen = 0;
    cmac->first  = 1;

    /* CBC buffer */
    for (i = 0; i < 16; i++) {
        cmac->C[i] = 0;
    }
}
```

This final bit of code initializes the buffer and CBC chaining variable. We are now ready to process the message through CMAC.

```
void cmac_process(const unsigned char *in, unsigned inlen,
                           cmac_state *cmac)
{
```

Our "process" function is much like the process functions found in the implementations of the hash algorithms. It allows the caller to send in an arbitrary length message to be handled by the algorithm.

```
    while (inlen–) {
      cmac->first = 0;
```

This turns off the first block flag telling the CMAC functions that we have processed at least one byte in the function.

```
      /* we have 16 bytes, encrypt the buffer */
      if (cmac->buflen == 16) {
        AesEncrypt(cmac->C, cmac->C, cmac->AESkey, 10);
        cmac->buflen = 0;
      }
```

If we have filled the CBC chaining block, we must encrypt it and clear the counter. We must do this for every 16 bytes we process, since we assume we are using AES, which has a 16-byte block size.

```
      /* xor in next byte */
      cmac->C[cmac->buflen++] ^= *in++;
    }
  }
```

The last statement XORs a byte of the message against the CBC chaining block. Notice, how we check for a full block *before* we add the next byte. The reason for this becomes more apparent in the next function.

This loop can be optimized on 32- and 64-bit platforms by XORing larger words of input message against the CBC chaining block. For example, on a 32-bit platform we could use the following:

```
if (cmac->buflen == 0) {
  while (inlen >= 16) {
     *((ulong32*)&cmac->C[0]) ^= *((ulong32*)&in[0]);
     *((ulong32*)&cmac->C[4]) ^= *((ulong32*)&in[4]);
     *((ulong32*)&cmac->C[8]) ^= *((ulong32*)&in[8]);
     *((ulong32*)&cmac->C[12]) ^= *((ulong32*)&in[12]);
     if (inlen > 16) AesEncrypt(cmac->C, cmac->C, cmac->AESKey, 10);
     inlen -= 16;
     in    += 16;
  }
}
```

This loop XORs 32-bit words at a time, and for performance reasons assumes that the input buffer is aligned on a 32-bit boundary. Note that it is endianess neutral and only depends on the mapping of four unsigned chars to a single ulong32. That is, the code is not entirely portable but will work on many platforms. Note that we only process if the CMAC buffer is empty, and we only encrypt if there are more than 16 bytes left.

The LibTomCrypt library uses a similar trick that also works well on 64-bit platforms. The OMAC routines in that library provide another example of how to optimize CMAC.

NOTE

The x86-based platforms tend to create "slackers" in terms of developers. The CISC instruction set makes it fairly effective to write decently efficient programs, especially with the ability to use memory operands as operands in typical RISC like instructions—whereas on a true RISC platforms you must load data before you can perform an operation (such as addition) on it.

Another feature of the x86 platform is that unaligned are tolerated. They are sub-optimal in terms of performance, as the processor must issue multiple memory commands to fulfill the request. However, the processor will still allow it.

On other platforms, such as MIPS and ARM, word memory operations must always be word aligned. In particular, on the ARM platform, you cannot actually perform unaligned memory operations without manually emulating them, since the processor zero bits of the address.

This causes problems for C applications that try to cast a pointer to another type. As in our example, we cast an *unsigned char* pointer to a *ulong32* pointer. This will work well on x86 platforms, but only work on ARM and MIPS if the pointer is 32-bit aligned. The C compiler will not detect this error at compile time and the user will only be able to tell there is an error at runtime.

```
void cmac_done(   cmac_state *cmac,
                unsigned char *tag, unsigned taglen)
{
   unsigned i;
```

This function terminates the CMAC and outputs the MAC tag value.

```
   /* do we have a partial block? */
   if (cmac->first || cmac->buflen & 15) {
       /* yes, append the 0x80 byte */
       cmac->C[cmac->buflen++] ^= 0x80;

       /* xor K2 */
       for (i = 0; i < 16; i++) {
           cmac->C[i] ^= cmac->L[1][i];
       }
```

If we have zero bytes in the message or an incomplete block, we first append a one bit follow by enough zero bits. Since we are byte based, the padding is the 0x80 byte followed by zero bytes. We then XOR K2 against the block.

```
    } else {
        /* no, xor K1 */
        for (i = 0; i < 16; i++) {
            cmac->C[i] ^= cmac->L[0][i];
        }
    }
```

Otherwise, if we had a complete block we XOR K1 against the block.

```
    /* encrypt pad */
    AesEncrypt(cmac->C, cmac->C, cmac->AESkey, 10);
```

We encrypt the CBC chaining block one last time. The ciphertext of this encryption will be the MAC tag. All that is left is to truncate it as requested by the caller.

```
    /* copy tag */
    for (i = 0; i < 16 && i < taglen; i++) {
        tag[i] = cmac->C[i];
    }
}

void cmac_memory(const unsigned char *key,
                  const unsigned char *in, unsigned inlen,
                         unsigned char *tag, unsigned taglen)
{
    cmac_state cmac;
    cmac_init(key, &cmac);
    cmac_process(in, inlen, &cmac);
    cmac_done(&cmac, tag, taglen);
}
```

This simple function allows the caller to compute the CMAC tag of a message with a single function call. Very handy to have.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    static const struct {
        int keylen, msglen;
        unsigned char key[16], msg[64], tag[16];
} tests[] = {
{ 16, 0,
  { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
    0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c },
  { 0x00 },
  { 0xbb, 0x1d, 0x69, 0x29, 0xe9, 0x59, 0x37, 0x28,
    0x7f, 0xa3, 0x7d, 0x12, 0x9b, 0x75, 0x67, 0x46 }
},
{ 16, 16,
  { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
    0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c },
  { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
    0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a },
  { 0x07, 0x0a, 0x16, 0xb4, 0x6b, 0x4d, 0x41, 0x44,
    0xf7, 0x9b, 0xdd, 0x9d, 0xd0, 0x4a, 0x28, 0x7c }
},
{ 16, 40,
  { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
    0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c },
  { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
    0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
    0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
    0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
    0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11 },
  { 0xdf, 0xa6, 0x67, 0x47, 0xde, 0x9a, 0xe6, 0x30,
    0x30, 0xca, 0x32, 0x61, 0x14, 0x97, 0xc8, 0x27 }
},
{ 16, 64,
  { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
    0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c },
  { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
    0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
    0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
    0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
    0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11,
    0xe5, 0xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
    0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17,
    0xad, 0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10 },
  { 0x51, 0xf0, 0xbe, 0xbf, 0x7e, 0x3b, 0x9d, 0x92,
    0xfc, 0x49, 0x74, 0x17, 0x79, 0x36, 0x3c, 0xfe }
}
};
```

These arrays are the standard test vectors for CMAC with AES-128. An implementation must at the very least match these vectors to claim CMAC AES-128 compliance.

```
    unsigned char tag[16];
    int i;

    for (i = 0; i < 4; i++) {
       cmac_memory(tests[i].key, tests[i].msg,
                tests[i].msglen, tag, 16);
       if (memcmp(tag, tests[i].tag, 16)) {
          printf("CMAC test %d failed\n", i);
          return -1;
      }
    }
    printf("CMAC passed\n");
    return 0;
}
```

This demonstration program computes the CMAC tags for the test messages and compares the tags. Keep in mind this test program only uses AES-128 and not the full AES suite. Although, in general, if you can comply to the AES-128 CMAC test vectors, you should comply with the AES-192 and AES-256 vectors as well.

CMAC Performance

Overall, the performance of CMAC depends on the underlying cipher. With the feedback optimization for the process function (XORing words of data instead of bytes), the overhead can be minimal.

Unfortunately, CMAC uses CBC mode and cannot be parallelized. This means in hardware, the best performance will be achieved with the fastest AES implementation and not many parallel instances.

Hash Message Authentication Code

The Hash Message Authentication Code standard (FIPS 198) takes a cryptographic one-way hash function and turns it into a message authentication code algorithm. Remember how earlier we said that hashes are not authentication functions? This section will tell you how to turn your favorite hash into a MAC.

The overall HMAC design was derived from a proposal called NMAC, which turns any pseudo random function (PRF) into a MAC function with provable security bounds. In particular, the focus was to use a hash function as the PRF. NMAC was based on the concept of prefixing the message with a key smf then hashing the concatenation. For example, *tag* = hash(*key* || *message*)

However, recall from Chapter 5 that we said that such a construction is not secure. In particular, an attacker can extend the message by using the *tag* as the initial state of the hash. The problem is that the attacker knows the message being hashed to produce the tag. If we could somehow hide that, the attacker could not produce valid tag, Effectively, we have

tag = hash(*key* || PRF(*message*))

Now an attacker who attempts to extend the message by using *tag* as the initial hash state, the result of the PRF() mapping will not be predictable. In this configuration, an attacker can no longer use *tag* as the initial state. The only question now is how to create a PRF? It turns out that the original construction is a decent PRF to use. That is, hash functions are by definition pseudo random functions that map their inputs to difficult to otherwise predict outputs. The outputs are also hard to invert (that is, the hash is one-way). Keying the hash function by pre-pending secret data to the message should by definition create a keyed PRF.

The complete construction is then

tag = hash(*key1* || hash(*key2* || message))

Note that NMAC requires two keys, one for in the *inner* hash and one for the *outer* hash. While the constructions is simple, it lacks efficiency as it requires two independent keys.

The HMAC construction is based on NMAC, except that the two keys are linearly related. The contribution of HMAC was to prove that the construction with a single key is also secure. It requires that the hash function be a secure PRF, and while it does not have to be collision resistant (New Proofs for NMAC and HMAC: Security without Collision Resistant: http://eprint.iacr.org/2006/043.pdf), it must be resistant to differential cryptanalysis (On The Security of HMAC and NMAC Based on HAVAL, MD4, MD5, SHA-0, and SHA-1: http://eprint.iacr.org/2006/187.pdf).

HMAC Design

Now that we have a general idea of what HMAC is, we can examine the specifics that make up the FIPS 198 standard. HMAC, like CMAC, is not specific to a given algorithm underneath. While HMAC was originally intended to be used with SHA-1 it can safely be used with any otherwise secure hash function, such as SHA-256 and SHA-512.

HMAC derives the two keys from a single secret key by XORing two constants against it. First, before we can do this we have to make sure the key is the size of the hashes compression block size. For instance, SHA-1 and SHA-256 compress 64-byte blocks, whereas SHA-512 compresses 128-byte blocks.

If the secret key is larger than the compression block size, the standard requires the key to be hashed first. The output of the hash is then treated as the secret key. The secret key is padded with zero bytes to ensure it is the size of the compression block input size.

The result of the padding is then copied twice. One copy has all the bytes XORed with 0x36; this is the outer key. The other copy has all its bytes XORed with 0x5C; this is the inner key. For simplicity we shall call the string of 0x36 bytes the *opad*, and the string of 0x5C bytes the *ipad*.

You may wonder why the key is padded to be the size of a compression block. Effectively, this turns the hash into a keyed hash by making the initial hash state key dependent. From a cryptographic standpoint, HMAC is equivalent to picking the hashes initial state at random based on a secret key (Figures 6.6 and 6.7).

Figure 6.6 HMAC Block Diagram

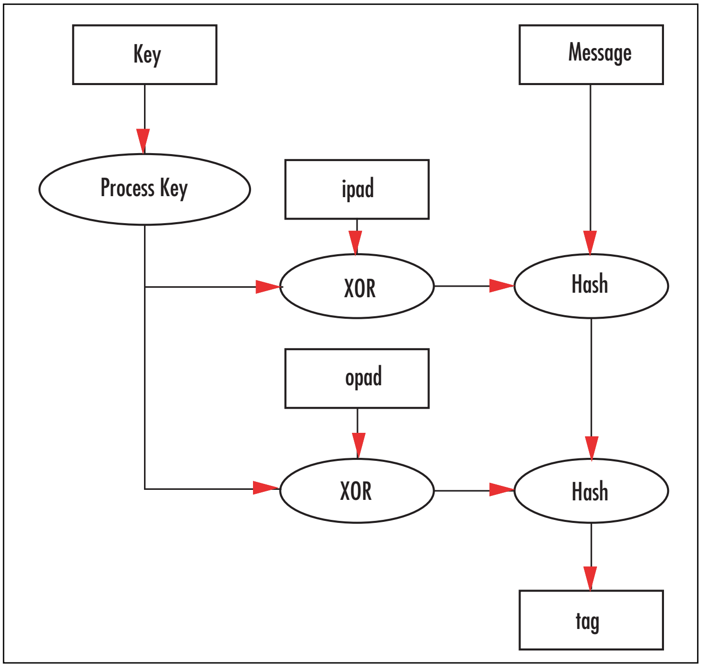


Figure 6.7 HMAC Algorithm

```
Input
    K:          Secret key
    message:    Message to determine the MAC for
    w:          Compression block size
    Tlen:       Desired MAC tag length
Output
    Tag:        The MAC Tag

  1. if length(K) > w then
      1. K = hash(K)
  2. Pad K with zeros until it is w bytes long
  3. Tag = hash((opad XOR K) || hash((ipad XOR K) || message))
  4. Truncate Tag to Tlen bytes by keeping only the first Tlen bytes.
  5. Return Tag.
```

As we can see, HMAC is a simple algorithm to describe. With a handy hash function implementation, it is trivial to implement HMAC. We note that, since the message is only hashed in the inner hash, we can effectively HMAC a stream on the fly without the need forthe entire message at once (provided the hash function does not require the entire message at once).

Since this algorithm uses a hash as the PRF, and most hashes are not able to process message blocks in parallel, this will become the significant critical path of the algorithm in terms of performance. It is possible to compute the hash of (*opad* XOR *K*) while computing the inner hash, but this will save little time and require two parallel instances of the hash. It is not a worthwhile optimization for messages that are more than a few message blocks in length.

HMAC Implementation

Our implementation of HMAC has been tied to the SHA-1 hash function. Like CMAC, we decided to simplify the implementation to ensure the implementation is easy to absorb.

hmac.c:

```
/* poor linker */
#include “sha1.c”
```

We directly include the SHA-1 source code to provide our hash function. Ideally, we would include a proper header file and link against SHA-1. However, at this point we just want to show off HMAC working.

```
typedef struct {
   sha1_state    hash;
```

```
  unsigned char K[64];
} hmac_state;
```

This is our HMAC state. It is fairly simple, since all of the data buffering is handled internally by the SHA-1 functions. We keep a copy of the outer key *K* to allow the HMAC implementation to apply the outer hash. Note that we would have to change the size of *K* to suit the hash. For instance, with SHA-512 it would have to be 128 bytes long. We would have to make that same change to the following function.

```
void hmac_init(const unsigned char *key,
                 unsigned  keylen,
                hmac_state *hmac)
{
  unsigned char K[64];
  unsigned i;
```

This function initializes the HMAC state by processing the key and starting the inner hash.

```
  /* if keylen > 64 hash it */
  if (keylen > 64) {
    sha1_memory(key, keylen, K);
    i = 20;
  } else {
    /* copy key */
    for (i = 0; i < keylen; i++) {
       K[i] = key[i];
    }
  }
```

If the secret key is larger than 64 bytes (the compression block size of SHA-1), we hash it using the helper SHA-1 function. If it is not, we copy it into our local *K* array. In either case, at this point *i* will indicate the number of bytes in the array that have been filled in. This is used in the next loop to pad the key.

```
  /* pad with zeros */
  for (; i < 64; i++) {
     K[i] = 0x00;
  }
```

This pads the keys with zero bytes so that it is 64 bytes long.

```
  /* copy key to structure, this is out outer key */
  for (i = 0; i < 64; i++) {
    hmac->K[i] = K[i] ^ 0x5C;
  }
```

```

   /* XOR inner key with 0x36 */
   for (i = 0; i < 64; i++) {
      K[i] ^= 0x36;
   }
```

The first loop creates the outer key and stores it in the HMAC state. The second loop creates the inner key and stores it locally. We only need it for a short period of time, so there is no reason to copy it to the HMAC state.

```
   /* start hash */
   sha1_init(&hmac->hash);

   /* hash key */
   sha1_process(&hmac->hash, K, 64);

   /* wipe key */
   for (i = 0; i < 64; i++) {
      K[i] = 0x00;
   }
}
```

At this point we have initialized the HMAC state. We can process data to be authenticated with the following function.

```
void hmac_process(const unsigned char *in,
                               unsigned  inlen,
                             hmac_state *hmac)
{
   sha1_process(&hmac->hash, in, inlen);
}
```

This function processes data we want to authenticate. Take a moment to appreciate the vast complexity of HMAC. Done? This is one of the reasons HMAC is a good standard. It is ridiculously simple to implement.

```
void hmac_done(   hmac_state *hmac,
                unsigned char *tag,
                     unsigned  taglen)
{
```

This function terminates the HMAC and outputs the tag.

```
    unsigned char T[20];
    unsigned      i;

```

The *T* array stores the message digest from the hash function. You will have to adjust it to match the output size of the hash function.

```
    /* terminate inner hash */
    sha1_done(&hmac->hash, T);

    /* start outer hash */
    sha1_init(&hmac->hash);

    /* hash the outer key */
    sha1_process(&hmac->hash, hmac->K, 64);

    /* hash the inner hash */
    sha1_process(&hmac->hash, T, 20);

    /* get the output (tag) */
    sha1_done(&hmac->hash, T);

    /* copy out */
    for (i = 0; i < 20 && i < taglen; i++) {
       tag[i] = T[i];
    }
}
```

At this point, we have all the prerequisites to begin using HMAC to process data. We can borrow from our hash implementations a bit to help out here.

```
void hmac_memory(const unsigned char *key,
                              unsigned  keylen,
                  const unsigned char *in, unsigned inlen,
                        unsigned char *tag, unsigned taglen)
{
   hmac_state hmac;
   hmac_init(key, keylen, &hmac);
   hmac_process(in, inlen, &hmac);
   hmac_done(&hmac, tag, taglen);
}
```

As in the case of CMAC, we have provided a simple to use HMAC function that produces a tag with a single function call.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

```
int main(void)
{
    static const struct {
        unsigned char key[128];
        unsigned long keylen;
        unsigned char data[128];
        unsigned long datalen;
        unsigned char tag[20];
    } tests[] = {
    {
      {0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c,
       0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c,
       0x0c, 0x0c, 0x0c, 0x0c}, 20,
      “Test With Truncation”, 20,
      {0x4c, 0x1a, 0x03, 0x42, 0x4b, 0x55, 0xe0, 0x7f,
       0xe7, 0xf2, 0x7b, 0xe1, 0xd5, 0x8b, 0xb9, 0x32,
       0x4a, 0x9a, 0x5a, 0x04} },
    {
      {0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
       0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
       0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
       0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
       0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
       0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
       0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
       0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
       0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
       0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa}, 80,
      “Test Using Larger Than Block-Size Key - “
      “Hash Key First”, 54,
      {0xaa, 0x4a, 0xe5, 0xe1, 0x52, 0x72, 0xd0, 0x0e,
       0x95, 0x70, 0x56, 0x37, 0xce, 0x8a, 0x3b, 0x55,
       0xed, 0x40, 0x21, 0x12} },
   {
      {0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
       0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
       0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
       0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
       0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
       0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
       0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
       0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
       0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
       0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa}, 80,
      "Test Using Larger Than Block-Size Key and Larger "
      "Than One Block-Size Data", 73,
      {0xe8, 0xe9, 0x9d, 0x0f, 0x45, 0x23, 0x7d, 0x78,
       0x6d, 0x6b, 0xba, 0xa7, 0x96, 0x5c, 0x78, 0x08,
       0xbb, 0xff, 0x1a, 0x91} } };
   unsigned char tag[20];
   unsigned i;

   for (i = 0; i < 3; i++) {
      hmac_memory(tests[i].key, tests[i].keylen,
                   tests[i].data, tests[i].datalen,
                   tag, 20);
      if (memcmp(tag, tests[i].tag, 20)) {
         printf("HMAC-SHA1 Test %u failed\n", i);
         return -1;
      }
   }
   printf("HMAC-SHA1 passed\n");
   return 0;
}
```

The test vectors in our implementation are from RFC 2202[1] published in 1997 (HMAC RFC: www.faqs.org/rfcs/rfc2104.html, HMAC Test Cases: www.faqs.org/rfcs/rfc2202.html). The Request for Comments (RFC) publication was the original standard for the HMAC algorithm. We use these vectors since they were published before the test vectors listed in FIPS 198. Strictly speaking, FIPS 198 is not dependent on RFC 2104; that is, to claim standard compliance with FIPS 198, you must pass the FIPS 198 test vectors.

Fortunately, RFC 2104 and FIPS 198 specify the same algorithm. Oddly enough, in the NIST FIPS 198 specification they claim their standard is a "*generalization*" of RFC 2104. We have not noticed any significant difference between the standards. While HMAC was originally intended to be used with MD5 and SHA-1, the RFC does not state that it is limited in such a way. In fact, quoting from the RFC, "This document specifies HMAC using a generic cryptographic hash function (denoted by H)," we can see clearly the intended scope of the standard was not limited to a given hash function.

Putting It All Together

Now that we have seen two standard MAC functions, we can begin to work on how to use the MAC functions to accomplish useful goals in a secure manner. We will first examine the basic tasks MAC functions are for, and what they are not for. Next, we will compare CMAC and HMAC and give advice on when to use one over the other.

After that point, we will have examined which MAC function to use and why we want you to use them. We will then proceed to examine how to use them securely. It is not enough to simply apply the MAC function to your data; you must also apply it within a given context for the system to be secure.

What MAC Functions Are For?

First and foremost, MAC functions were designed to provide authenticity between parties on a communication channel. If all is working correctly, all of the parties can both send and receive (and verify) authenticated messages between one another—that is, without the significant probability that an attacker is forging messages.

So, what exactly do we use MACs for in real-world situations? Some classic examples include SSL and TLS connections on the Internet. For example, HTTPS connections use either TLS or SSL to encrypt the channel for privacy, and apply a MAC to the data being sent in either direction to ensure its authenticity. SSH is another classic example. SSH (Secure Shell) is a protocol that allows users to remotely log in to other machines in much the same way telnet would. However, unlike telnet, SSH uses cryptography to ensure both the privacy and authentication of the data being sent in both directions. Finally, another example is the GSM cellular standard. It uses an algorithm known as COMP128 to authenticate users on the network. Unlike SSL and SSH, COMP128 is used to authenticate the user by providing a challenge for which the user must provide the correct MAC tag in response. COMP128 is not used to authenticate data in transit.

Consequences

Suppose we had not used MAC authenticators in the previous protocols, what threats would the users face?

In the typical use case of TLS and SSL, users are trying to connect to a legitimate host and transfer data between the two points. The data usually consists of HTTPS traffic; that is, HTML documents and form replies. The consequences of being able to insert or modify replies depends on the application. For instance, if you are reading e-mail, an attacker could re-issue previous commands from the victim to the server. These commands could be innocuous such as "read next e-mail," but could also be destructive such as "delete current e-mail." Online banking is another important user of TLS cryptography. Suppose you issued an HTML form reply of "pay recipient $100." An attacker could modify the message and change the dollar amount, or simpler yet, re-issue the command depleting the bank account.

Modification in the TLS and SSL domains are generally less important, as the attacker is modifying ciphertext, not plaintext. Attackers can hardly make modifications in a protocol legible fashion. For instance, a user connected to SMTP through TLS must send properly formatted commands to have the server perform work. Modifying the data will likely modify the commands.

NOTE

One of the consequences of not using a MAC with protocols such as TLS is that the attacker could modify the ciphertext. With ciphering modes such as CTR, a single bit change of ciphertext leads to only one bit changed in the plaintext.

CBC is not a forgery resistant mode of operation. An attacker who knows plaintext and ciphertext pairings, and who can control the CBC IV used, can *forge* a message that will decrypt sensibly.

If you need authentication, and chances are good that you do, CBC mode is not the way to get it.

In the SSH context, without a MAC in place, an attacker could re-issue many commands sent by the users. This is because SSH uses a Nagle-like protocol to buffer out going data. For instance, if SSH sent a packet for every keystroke you made, it would literally be sending kilobytes of data to simply enter a few paragraphs of text. Instead, SSH, like the TCP/IP protocol, buffers outgoing data until a reply is received from the other end. In this way, the protocol is *self-clocking* and will only give as low latency as the network will allow while using an optimal amount of bandwidth.

With the buffering of input comes a certain problem. For instance, if you typed "rm –f *" at your shell prompt to empty a directory, chances are the entire string (along with the newline) would be sent out as one packet to the SSH server. An attacker who retrieves this packet can then replay it, regardless of where you are in your session.

In the case of GSM's COMP128 protocol, if it did not exist (and no replacement was used), a GSM client would simply pop on a cellular network, say "I'm Me!" and be able to issue any command he wanted such as dialing a number or initiating a GPRS (data) session.

In short, without a MAC function in place, attackers can modify packets and *replay* old packets without immediate and likely detection. Modifying packets may seem hard at first; however, on various protocols it is entirely possible if the attacker has some knowledge of the likely characteristics of the plaintext. People tend to think that modifications will be caught further down the line. For instance, if the message is text, a human reading it will notice. If the message is an executable, the processor will throw an exception at invalid instructions. However, not all modifications, even random ones (without an attacker), are that noticeable.

For instance, in many cases on machines with broken memory—that is, memory that is either not storing values correctly, or simply failing to meet timing specifications—errors received by the processor go unnoticed for quite some time. The symptoms of problematic memory could be as simple as a program terminating, or files and directories not appearing correctly. Servers typically employ the use of error correcting code (ECC) based memory, which sends redundant information along with the data. The ECC data is then used to correct errors on the fly. Even with ECC, however, it's possible to have memory failures that are not correctable.

Message replay attacks can be far more dangerous. In this case, an attacker resends an old valid packet with the hopes of causing harm to the victim. The replayed packets are not modified so the decryption should be valid, and depending on the current context of the protocol could be very dangerous.

What MAC Functions Are Not For?

MAC functions usually are not abused in cryptographic applications, at least not in the same way hash and cipher functions are. Sadly, this is not because of the technical understanding of MAC functions by most developers. In fact, most amateur protocols omit authentication threats altogether. As an aside, that is usually a good indicator of snake oil products. MAC functions are not usually abused by those who know about them, mostly because MAC functions are made out of useful primitives like ciphers and hashes. With that said, we shall still briefly cover some basics.

MACs are not meant to be used as hash functions. Occasionally, people will ask about using CMAC with a fixed key to create a hash function. This construction is not secure because the "compression" function (the cipher) is no longer a PRP. None of the proofs applies at that point. There is a secure way of turning a cipher into a hash:

```
1. $H[0] = \text{Encrypt}_0(0)$
2. Pad the message with MD Strengthening and divide into blocks M[1], M[2], ..., M[n]
3. for i from 1 to n do
    1. $H[i] = \text{Encrypt}_{H[i-1]}(M[i]) \text{ XOR } M[i]$
4. Return $H[n]$ as the message digest
```

In this mode, EncryptK(P) means to encrypt the plaintext *P* with key *K*. The initial value, H[0], is chosen by encrypting the zero string to make the protocol easier to specify. This method of hashing is not part of any NIST standard, although it has been used in a variety of products. It is secure, provided the cipher itself is immune to differential attacks and the key schedule is immune to related key attacks.

MACs are also not meant for RNG processing or even to be turned into a PRNG. With a random secret key, the MAC function should behave as a PRF to an attacker. This implies that it could be used as a PRNG, which is indeed true. In practice, a properly keyed MAC could make a secure PRNG function if you ran a counter as the input and used the output as the random numbers. However, such a construction is slow and wastes resources. A properly keyed cipher in CTR mode can accomplish the same goal and consume fewer resources.

CMAC versus HMAC

Comparing CMAC against HMAC for useful comparison metrics depends on the problem you are trying to solve, and what means you have to solve it. CMAC is typically a good choice if you already have a cipher lying around (say, for privacy). You can re-use the same code (or hardware) to accomplish authentication. CMAC is also based on block ciphers, which while having small inputs (compared to hashes) can return an output in short order. This reduces the latency of the operation. For short messages, say as small as 16 or 32 bytes, CMAC with AES can often be faster and lower latency than HMAC.

Where HMAC begins to win out is on the larger messages. Hashes typically process data with fewer cycles per byte when compared to ciphers. Hashes also typically create larger outputs than ciphers, which allows for larger MAC tags as required. Unfortunately, HMAC requires you to have a hash function implemented; however, on the upside, the HMAC code that wraps around the hash implementation is trivial to implement.

From a security standpoint, provided the respective underlying primitives (e.g., the cipher or the hash) are secure on their own (e.g., a PRP or PRF, respectively), the MAC construction can be secure as well. While HMAC can typically put out larger MAC tags than CMAC (by using a larger hash function), the security advantage of using larger tags is not significant.

Bottom line: If you have the space or already have a hash, and your messages are not trivially small, use HMAC. If you have a cipher (or want a smaller footprint) and are dealing with smaller messages, use CMAC. Above all, do not re-invent the wheel; use the standard that fits your needs.

Replay Protection

Simply applying the MAC function to your message is not enough to guarantee the system as a whole is safe from replays. This problem arises due to a need for efficiency and necessity. Instead of sending a message as one long chunk of data, a program may divide it into smaller, more manageable pieces and send them in turn. Streaming applications such as SSH essentially demand the use of data packets to function in a useful capacity.

The cryptographic vulnerabilities of using packets, even individually authenticated packets, is that they are meant to form, as a whole, a larger message. That is, the individual packets themselves have to be correct but also the order of the packets over the entire session. An attacker may exploit this venue by replaying old packets or modifying the order of the packets. If the system has no way to authenticate the order of the packets, it could accept the packets as valid and interpret the re-arrangement as the entire message.

The classic example of how this is a problem is a purchase order. For example, a client wishes to buy 100 shares of a company. So he packages up the message M = "I want to buy 100 shares." The client then encrypts the message and applies a MAC to the ciphertext. Why is this not secure? An attacker who sees the ciphertext and MAC tag can retransmit the pair. The server, which is not concerned with replay attacks, will read the pair, validate the MAC, and interpret the ciphertext as valid. Now an attacker can deplete the victim's funds by reissuing the purchase order as often as he desires.

The two typical solutions to replay attacks are timestamps and counters. Both solutions must include extra data as part of the message that is authenticated. They give the packets a *context*, which helps the receiver interpret their presence in the stream.

Timestamps

Timestamps can come in various shapes and sizes, depending on the precision and accuracy required. The goal of a timestamp is to ensure that a message is valid only for a given period of time. For example, a system may only allow a packet to be valid for 30 seconds since the timestamp was applied. Timestamps sound wonderfully simple, but there are various problems.

From a security standpoint, they are hard to get narrow enough to avoid replay attacks within the window of validity. For instance, if you set the window of validity to (say) five minutes, an attacker can replay the message during a five-minute period. On the other hand, if you make the window only three seconds, you may have a hard time delivering the packet within the window; worse yet, clock drift between the two end points may make the system unusable.

This last point leads to the practical problems with timestamps. Computers are not terribly accurate time keepers. The system time can drift due to inaccuracies in the timing hardware, system crashes, and so on. Usually, most operating systems will provide a mechanism for updating system time, usually via the Network Time Protocol (NTP). Getting two computers, especially remotely, to agree on the current time is a difficult challenge.

Counters

Counters are the more ideal solution against replay attacks. At its most fundamental level, you need a concept of what the "next" value should be. For instance, you could store a LFSR state in one packet and expect the clocked LFSR in the next packet. If you did not see the clocked LFSR value, you could assume that you did not actually receive the packet desired. A simpler approach that is also more useful is to use an integer as the counter. As long as the counter is not reused during the same session (without first changing the MAC key), they can be used to put the packets in to context.

Counter values are agreed upon by both parties and do not have to be random or unique if a new MAC key is used during the session (for example, a MAC key derived from a key derivation function). Each packet sent increments the counter; the counter itself is included in the message and is part of the MAC function input. That is, you MAC both the ciphertext and the counter. You could optionally encrypt the counter, but there is often little security benefit in doing so. The recipient checks the counter before doing any other work. If the counter is less than or equal to the newest packet counter, chances are it is a replay and you should act accordingly. If it is equal, you should proceed to the next step.

Often, it is wise to determine the size of your counter to be no larger than the maximum number of packets you will send. For example, it is most certainly a waste to use a 16-byte counter. For most applications, an 8-byte counter is excessive as well. For most networked applications, a 32-bit counter is sufficient, but as the situation varies, a 40- or 48-bit counter may be required.

Another useful trick with counters in bidirectional mediums is to have one counter for each direction. This makes the incoming counter independent of the outgoing counter. In effect, it allows both parties to transmit at the same time without using a *token passing* scheme.

Encrypt then MAC?

A common question regarding the use of both encryption and MAC algorithms is which order to apply them in? Encrypt then MAC or MAC then Encrypt? That is, do you MAC the plaintext or ciphertext? Fundamentally, they seem they would provide the same level of security, but there are subtle differences.

Regardless of the order chosen, it is very important that the ciphering and MAC keys are not easily related to one another. The simplest and most appropriate solution is to use a key derivation function to stretch a shared (or known) shared secret into ciphering and MAC keys.

Encrypt then MAC

In this mode, you first encrypt the plaintext and then MAC the ciphertext (along with a counter or timestamp). Since the encryption is a proper random mapping of the plaintext (requires an appropriately chosen IV), the MAC is essentially of a random message. This mode is generally preferred on the basis that the MAC does not leak information about the plaintext. Also, one does not have to decrypt to reject invalid packets.

MAC then Encrypt

In this mode, you first MAC the plaintext (along with a counter or timestamp) and then encrypt the plaintext. Since the input to the MAC is not random and most MAC algorithms (at least CMAC and HMAC) do not use IVs, the output of the MAC will be nonrandom—that is, if you are not using replay protection. The common objection is that the MAC is based on the plaintext, so you are giving an attacker the ciphertext and the MAC tag of the plaintext. If the plaintext remained unchanged, the ciphertext may change (due to proper selection of an IV), but the MAC tag would remain the same.

However, one should always include a counter or timestamp as part of the MAC function input. Since the MAC is a PRF, it would defeat this attack even if the plaintext remained the same. Better yet, there is another way to defeat this attack altogether: simply apply the encryption to the MAC tag as well as the plaintext. This *will not* prevent replay attacks (you still need some variance in the MAC function input), but will totally obscure the MAC tag value from the attacker.

The one downside to this mode is that it requires a victim to decrypt before he can compare the MAC tag for a forgery. Oddly enough, this downside has a surprisingly useful upside. It does not leak, at least on its own, timing information to the attacker. That is, in the first mode we may stop processing once the MAC fails. This tells the attacker when the forgery fails. In this mode, we always perform the same amount of work to verify the MAC tag. How useful this is to an attacker depends on the circumstances. At the very least, it is an edge over the former mode.

Encryption and Authentication

We will again draw upon LibTomCrypt to implement an example system. The example is meant for bidirectional channels where the threat vectors include privacy and authenticity violations, and stream modifications such as re-ordering and replays.

The example code combats these problems with the use of AES-CTR for privacy, HMAC-SHA256 for authenticity, and numbered packets for stream protection. The code is not quite optimal, but does provide for a useful foundation upon which to improve the code as the situation warrants. In particular, the code is not thread safe. That is, if two treads attempt to send packets at the same time, they will corrupt the state.

encmac.c:

```
#include <tomcrypt.h>
```

We are using LibTomCrypt to provide the routines. Please have it installed if you wish to attempt to run this demonstration.

```
#define ENCKEYLEN    16
#define MACKEYLEN    16
```

These are our encryption and MAC key lengths. The encrypt key length must be a valid AES key length, as we have chosen to use AES. The MAC key length can be any length, but practically speaking, it might as well be no larger than the encryption key.

```
/* Our Per Packet Sizes, CTR len and MAC len */
#define CTRLEN       4
#define MACLEN       12
#define OVERHEAD     (CTRLEN+MACLEN)
```

These three macros define our per packet sizes. CTRLEN defines the size of the packet counter. The default, four bytes, allows one to send 2^{32} packets before an overflow occurs and the stream becomes unusable. On the upside, this is a simple way to avoid using the same key settings for too long.

MACLEN defines the length of the MAC tag we wish to store. The default, 12 bytes (96 bits), should be large enough to make forgery difficult. Since we are limited to 2^{32} packets, the advantage of a forger will remain fairly minimal.

Together, both values contribute OVERHEAD bytes to each packet in addition to the ciphertext. With the defaults, the data expands by 16 bytes per packet. With a typical packet size of 1024 bytes, this represents a 1.5-percent overhead.

```
/* errors */
#define MAC_FAILED    -3
#define PKTCTR_FAILED -4
```

MAC_FAILED indicates when the message MAC tag does not compare to what the recipient generates; for instance, if an attacker modifies the payload or the counter (or both). PKTCTR_FAILED indicates when a counter has been replayed or out of order.

```
/* our nice containers */
typedef struct {
   unsigned char PktCTR[CTRLEN],
                  enckey[ENCKEYLEN],
                  mackey[MACKEYLEN];
   symmetric_CTR skey;
} encauth_channel;
```

This structure contains all the details we need about a single unidirectional channel. We have the packet counter (PktCTR), encryption key (enckey), and MAC key (mackey). We have also scheduled a key in advance to reduce the processing latency (skey).

```
typedef struct {
   encauth_channel channels[2];
} encauth_stream;
```

This structure simply encapsulates two unidirectional streams into one structure. In our notation, channel[0] is always the outgoing channel, and channel[1] is always the incoming channel. We will see shortly how we can have two peers communicating with this convention.

```
void register_algorithms(void)
{
   register_cipher(&aes_desc);
   register_hash(&sha256_desc);
}
```

This function registers AES and SHA256 with LibTomCrypt so we can use them in the plug-in driven functions.

```
int init_stream(const unsigned char *masterkey,
                             unsigned  masterkeylen,
                 const unsigned char *salt,
                             unsigned  saltlen,
                       encauth_stream *stream,
                                   int  node)
{
```

This function initiates a bi-directional stream with a given master key and salt. It uses PKCS #5 key derivation to obtain a pair of key for each direction of the stream.

The node parameter allows us to swap the meaning of the streams. This is used by one of the parties so that their outgoing stream is the incoming stream of the other party (and vice versa for their incoming).

```
   unsigned char tmp[2*(ENCKEYLEN+MACKEYLEN)];
   unsigned long tmplen;
   int            err;
   encauth_channel tmpswap;

   /* derive keys */
   tmplen = sizeof(tmp);
   if ((err = pkcs_5_alg2(masterkey, masterkeylen,
                            salt, saltlen,
                            16, find_hash("sha256"),
                            tmp, &tmplen)) != CRYPT_OK) {
      return err;
   }
```

This call derives the bytes required for the two pairs of keys. We use only 16 iterations of PKCS #5 since we will make the assumption that masterkey is randomly chosen.

```
   /* copy keys */
   memcpy(stream->channels[0].enckey,
          tmp, ENCKEYLEN);
   memcpy(stream->channels[0].mackey,
          tmp + ENCKEYLEN, MACKEYLEN);
   memcpy(stream->channels[1].enckey,
          tmp + ENCKEYLEN + MACKEYLEN, ENCKEYLEN);
   memcpy(stream->channels[1].mackey,
          tmp + ENCKEYLEN + MACKEYLEN + ENCKEYLEN, MACKEYLEN);
```

This snippet extracts the keys from the PKCS #5 output buffer.

```
   /* reset counters */
   memset(stream->channels[0].PktCTR, 0,
          sizeof(stream->channels[0].PktCTR));
   memset(stream->channels[1].PktCTR, 0,
          sizeof(stream->channels[1].PktCTR));
```

With each new session, we start the packet counters at zero.

```
   /* schedule keys+setup mode */
   /* clear an IV */
   memset(tmp, 0, 16);
   if ((err = ctr_start(find_cipher("aes"), tmp,
                          stream->channels[0].enckey, ENCKEYLEN,
                          0, CTR_COUNTER_BIG_ENDIAN,
                          &stream->channels[0].skey))!= CRYPT_OK) {
      return err;
   }

   if ((err = ctr_start(find_cipher("aes"), tmp,
                          stream->channels[1].enckey, ENCKEYLEN,
                          0, CTR_COUNTER_BIG_ENDIAN,
                          &stream->channels[1].skey))!= CRYPT_OK) {
      return err;
   }
```

At this point, we have scheduled the encrypt keys for use. This means as we process keys, we do not have to run the (relatively slow) AES key schedule to initialize the CTR context.

```
   /* do we swap? */
   if (node != 0) {
       tmpswap              = stream->channels[0];
       stream->channels[0] = stream->channels[1];
       stream->channels[1] = tmpswap;
       zeromem(&tmpswap, sizeof(tmpswap));
   }
```

If we are not node 0, we swap the meaning of the streams. This allows two parties to talk to one another.

```
   zeromem(tmp, sizeof(tmp));
```

Wipe the keys off the stack. Note that we use the LTC zeromem() function, which will not be optimized by the compiler (well at least, very likely will not be) to a no-operation (which would be valid for the compiler).

```
   return 0;
}

int encode_frame(const unsigned char *in,
                              unsigned  inlen,
                         unsigned char *out,
                        encauth_stream *stream)
```

```
{
```

This function encodes a frame (or packet) by numbering, encrypting, and applying the HMAC. It stores inlen+OVERHEAD bytes in the out buffer. Note that in and out may not overlap in memory.

```
   int            x, err;
   unsigned char IV[16];
   unsigned long maclen;

   /* increment counter */
   for (x = CTRLEN-1; x >= 0; x—) {
       if (++(stream->channels[0].PktCTR[x])) break;
   }
```

We increment our packet counter in big endian format. This coincides with how we initialized the CTR sessions previously and will shortly come in handy.

```
   /* construct an IV */
   for (x = 0; x < CTRLEN; x++) {
       IV[x] = stream->channels[0].PktCTR[x];
   }
   for (; x < 16; x++) {
       IV[x] = 0;
   }

   /* set IV */
   if ((err = ctr_setiv(IV, 16,
                          &stream->channels[0].skey)) != CRYPT_OK) {
      return err;
   }
```

Our packet counter is only CTRLEN bytes long (default: 4), and an AES CTR mode IV is 16 bytes. We pad the rest with 0 bytes, but what does that mean in this context?

Our CTR counters are in big endian mode. The first CTRLEN bytes will be most significant bytes of the CTR IV. The last bytes (where we store the zeroes) are the least significant bytes. This means that while we are encrypting the text, the CTR counter is incremented only in the lower portion of the IV, preventing an overlap.

For instance, if CTRLEN was 15 and inlen was 257 * 16 = 4112, we would run into problems. The last 16 bytes of the first packet would be encrypted with the IV 0000000000000000000000000000**01**00, while the first 16 bytes of the second packet would be encrypted with the same IV.

As we recall from Chapter 4, CTR mode is as secure as the underlying block cipher (assuming it has been keyed and implemented properly) only if the IVs are unique. In this case, they would not be unique and an attacker could exploit the overlap.

This places upper bounds on implementation. With CTRLEN set to 4, we can have only 2^{32} packets, but each could be 2^{100} bytes long. With CTRLEN set to 8, we can have 2^{64} packets, each limited to 2^{68} bytes. However, the longer the CTRLEN setting, the larger the overhead. Longer packet counters do not always help; on the other hand, short packet counters can be ineffective if there is a lot of traffic.

```
   /* Store counter */
   for (x = 0; x < CTRLEN; x++) {
       out[x] = IV[x];
   }

   /* encrypt message */
   if ((err = ctr_encrypt(in, out+CTRLEN, inlen,
                           &stream->channels[0].skey)) != CRYPT_OK) {
      return err;
   }
```

At this point, we have stored the packet counter and the ciphertext in the output buffer. The first CTRLEN bytes are the counter, followed by the ciphertext.

```
   /* HMAC the ctr+ciphertext */
   maclen = MACLEN;
   if ((err = hmac_memory(find_hash("sha256"),
                           stream->channels[0].mackey, MACKEYLEN,
                           out, inlen + CTRLEN,
                           out + inlen + CTRLEN, &maclen)) != CRYPT_OK)
{
      return err;
   }
```

Our ordering of the data is not haphazard. One might wonder why we did not place the HMAC tag after the packet counter. This function call answers this question. In one fell swoop, we can HMAC both the counter and the ciphertext.

LibTomCrypt does provide a hmac_memory_multi() function, which is similar to hmac_memory() except that it uses a va_list to HMAC multiple regions of memory in a single function call (very similar to scattergather lists). That function has a higher caller overhead, as it uses va_list functions to retrieve the parameters.

```
   /* packet out[0...inlen+CTRLEN+MACLEN-1] now
      contains the authenticated ciphertext */
   return 0;
}
```

At this point, we have the entire packet ready to be transmitted. All packets that come in as inlen bytes in length come out as inlen+OVERHEAD bytes in length.

```
int decode_frame(const unsigned char *in,
                                     unsigned  inlen,
                               unsigned char *out,
                              encauth_stream *stream)
{
```

This function decodes and authenticates an encoded frame. Note that inlen is the size of the packet created by encode_frame() and *not* the original plaintext length.

```
   int            err;
   unsigned char IV[16], tag[MACLEN];
   unsigned long maclen;

   /* restore our original inlen */
   if (inlen < MACLEN+CTRLEN) { return -1; }
   inlen -= MACLEN+CTRLEN;
```

We restore the plaintext length to make the rest of the function comparable with the encoding. The first check is to ensure that the input length is actually valid. We return –1 if it is not.

```
   /* first compute the mactag */
   maclen = MACLEN;
   if ((err = hmac_memory(find_hash(“sha256”),
                          stream->channels[1].mackey, MACKEYLEN,
                          in, inlen + CTRLEN,
                          tag, &maclen)) != CRYPT_OK) {
      return err;
   }

   /* compare */
   if (memcmp(tag, in+inlen+CTRLEN, MACLEN)) {
      return MAC_FAILED;
   }
```

At this point, we have verified the HMAC tag and it is valid. We are not out of the woods yet, however. The packet could be a replay or out of order.

There is a choice of how the caller can handle a MAC failure. Very likely, if the medium is something as robust as Ethernet, or the underlying transport protocol guarantees delivery such as TCP, then a MAC failure is a sign of tampering. The caller should look at this as an active attack. On the other hand, if the medium is not robust, such as a radio link or watermark, a MAC failure could just be the result of noise overpowering the signal.

The caller must determine how to proceed based on the *context* of the application.

```
    /* compare CTR */
    if (memcmp(in, stream->channels[1].PktCTR, CTRLEN) <= 0) {
       return PKTCTR_FAILED;
    }
```

This memcmp() operation performs our nice big endian packet counter comparison. It will return a value <= 0 if the packet counter in the packet is not larger than the packet counter in our stream structure. We allow out of order packets, but only in the forward direction. For instance, receiving packets 0, 3, 4, 7, and 8 (in that order) would be valid; however, the packets 0, 3, 4, 1, 2 (in that order) would not be.

Unlike MAC failures, a counter failure can occur for various legitimate reasons. It is valid for UDP packets, for instance, to arrive in *any* order. While they will most likely arrive in order (especially over traditional IPv4 links), unordered packets are not always a sign of attack. Replayed packets, on the other hand, are usually not part of a transmission protocol.

The reader may wish to augment this function to distinguish between replay and out of order packets (such as using the sliding window trick).

```
    /* good to go, decrypt and copy the CTR */
    memset(IV, 0, 16);
    memcpy(IV, in, CTRLEN);
    memcpy(stream->channels[1].PktCTR, in, CTRLEN);

    /* set IV */
    if ((err = ctr_setiv(IV, 16,
                           &stream->channels[1].skey)) != CRYPT_OK) {
       return err;
    }

    /* encrypt message */
    if ((err = ctr_decrypt(in+CTRLEN, out, inlen,
&stream->channels[1].skey)) != CRYPT_OK) {
       return err;
    }
    return 0;
 }
```

Our test program will initialize two streams (one in either direction) and proceed to try to decrypt the same packet three times. It should work the first time, and fail the second and third times. On the second attempt, it should fail with a PKTCTR_FAILED error as we replayed the packet. On the third attempt, we have modified a byte of the payload and it should fail with a MAC_FAILED error.

```
int main(void)
{
   unsigned char  masterkey[16], salt[8];
   unsigned char  inbuf[32], outbuf[32+OVERHEAD];
   encauth_stream incoming, outgoing;
   int             err;

   /* setup lib */
   register_algorithms();
```

This sets up LibTomCrypt for use by our demonstration.

```
   /* pick master key */
   rng_get_bytes(masterkey, 16, NULL);
   rng_get_bytes(salt, 8, NULL);
```

Here we are using the system RNG for our key and salt. In a real application, we need to get our master key from somewhere a bit more useful. The salt should be generated in this manner.

Two possible methods of deriving a master key could be by hashing a user's password, or sharing a random key by using a public key encryption scheme.

```
   /* setup two streams */
   if ((err = init_stream(masterkey, 16,
                            salt, 8,
                            &incoming, 0)) != CRYPT_OK) {
      printf("init_stream error: %d\n", err);
      return EXIT_FAILURE;
   }
```

This initializes our incoming stream. Note that we used the value 0 for the node parameter.

```
      /* other side of channel would use this one */
   if ((err = init_stream(masterkey, 16,
                            salt, 8,
                            &outgoing, 1)) != CRYPT_OK) {
      printf("init_stream error: %d\n", err);
      return EXIT_FAILURE;
   }
```

This initializes our outgoing stream. Note that we used the value 1 for the node parameter. In fact, it does not matter which order we pick the node values in; as long as we are consistent, it will work fine.

Note also that each side of the communication has to generate only *one* stream structure to both encode and decode. In our example, we generate two because we are both encoding and decoding data we generate.

```
/* make a sample message */
memset(inbuf, 0, sizeof(inbuf));
strcpy((char*)inbuf, “hello world”);
```

Our traditional sample message.

```
if ((err = encode_frame(inbuf, sizeof(inbuf),
                          outbuf, &outgoing)) != CRYPT_OK) {
   printf(“encode_frame error: %d\n”, err);
   return EXIT_FAILURE;
}
```

At this point, outbuf[0...sizeof(inbuf)+OVERHEAD-1] contains the packet. By transmitting the entire buffer to the other party, they can authenticate and decrypt it.

```
/* now let’s try to decode it */
memset(inbuf, 0, sizeof(inbuf));
if ((err = decode_frame(outbuf, sizeof(outbuf),
                          inbuf, &incoming)) != CRYPT_OK) {
   printf(“decode_frame error: %d\n”, err);
   return EXIT_FAILURE;
}
printf(“Decoded data: [%s]\n”, inbuf);
```

We first clear the inbuf array to show that the routine did indeed decode the data. We decode the buffer using the incoming stream structure. At this point we should see the string

```
Decoded data: [hello world]
```

on the terminal.

```
/* now let’s try to decode it again (should fail) */
memset(inbuf, 0, sizeof(inbuf));
if ((err = decode_frame(outbuf, sizeof(outbuf),
                          inbuf, &incoming)) != CRYPT_OK) {
   printf(“decode_frame error: %d\n”, err);
   if (err != PKTCTR_FAILED) {
       printf(“We got the wrong error!\n”);
       return EXIT_FAILURE;
   }
}
```

This represents a replayed packet. It should fail with PKTCTR_FAILED, and we should see

```
decode_frame error: -4
```

on the terminal.

```
   /* let's modify a byte and try again */
   memset(inbuf, 0, sizeof(inbuf));
   outbuf[CTRLEN] ^= 0x01;
   if ((err = decode_frame(outbuf, sizeof(outbuf),
                           inbuf, &incoming)) != CRYPT_OK) {
      printf("decode_frame error: %d\n", err);
      if (err != MAC_FAILED) {
         printf("We got the wrong error!\n");
         return EXIT_FAILURE;
      }
   }
```

This represents both a replayed and forged message. It should fail the MAC test before getting to the packet counter check. We should see

```
decode_frame error: -3
```

on the terminal.

```
   return EXIT_SUCCESS;
}
```

This demonstration represents code that is not entirely optimal. There are several methods of improving it based on the context it will be used.

The first useful optimization ensures the ciphertext is aligned on a 16-byte boundary. This allows the LibTomCrypt routines to safely use word-aligned XOR operations to perform the CTR encryption. A simple way to accomplish this is to pad the message with zero bytes between the packet counter and the ciphertext (include it as part of the MAC input).

The second optimization involves knowledge of how LibTomCrypt works; the CTR structure exposes the IV nicely, which means we can directly set the IV instead of using ctr_setiv() to update it.

The third optimization is also a security optimization. By making the code thread safe, we can decode or encode multiple packets at once. This combined with a sliding window for the packet counter can ensure that even if the threads are executed out of order, we are reasonable assured that the decoder will accept them.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: What is a MAC function?

A: A MAC or message authentication code function is a function that accepts a secret key and message and reduces it to a MAC tag.

Q: What is a MAC tag?

A: A tag is a short string of bits that is used to prove that the secret key and message were processed together through the MAC function.

Q: What does that mean? What does authentication mean?

A: Being able to prove that the message and secret key were combined to produce the tag can directly imply one thing: that the holder of the key produced vouches for or simply wishes to convey an unaltered original message. A forger not possessing the secret key should have no significant *advantage* in producing verifiable MAC tags for messages. In short, the goal of a MAC function is to be able to conclude that if the MAC tag is correct, the message is intact and was not modified during transit. Since only a limited number of parties (typically only one or two) have the secret key, the ownership of the message is rather obvious.

Q: What standards are there?

A: There are two NIST standards for MAC functions currently worth considering. The CMAC standard is SP 800-38B and specifies a method of turning a block cipher into a MAC function. The HMAC standard is FIPS-198 and specifies a method of turning a hash function into a MAC. An older standard, FIPS-113, specifies CBC-MAC (a precursor to CMAC) using DES, and should be considered insecure.

Q: Should I use CMAC or HMAC?

A: Both CMAC and HMAC are secure when keyed and implemented safely. CMAC is typically more efficient for very short messages. It is also ideal for instances where a cipher is already deployed and space is limited. HMAC is more efficient for larger messages, and ideal when a hash is already deployed. Of course, you should pick whichever matches the standard you are trying to adhere to.

Q: What is advantage?

A: We have seen the term *advantage* several times in our discussion already. Essentially, the advantage of an attacker refers to the probability of forgery gained by a forger through analysis of previously authenticated messages. In the case of CMAC, for instance, the advantage is roughly approximate to $(mq)^2/2^{126}$ for CMAC-AES—where m is the number of messages authenticated, and q is the number of AES blocks per message. As the ratio approaches one, the probability of a successful forgery approaches one as well.

Advantage is a little different in this context than in the symmetric encryption context. An advantage of 2^{-40} is not the same as using a 40-bit encryption key. An attack on the MAC must take place online. This means, an attacker has but one chance to guess the correct MAC tag. In the latter context, an attacker can guess encryption keys offline and does not run the risk of exposure.

Q: How do key lengths play into the security of MAC functions?

A: Key lengths matter for MAC functions in much the same way they matter in symmetric cryptography. The longer the key, the longer a brute force key determination will take. If an attacker can guess a message, he can forge messages.

Q: How does the length of the MAC tag play into the security of MAC functions?

A: The length of the MAC tag is often variable (at least it is in HMAC and CMAC) and can limit the security of the MAC function. The shorter the tag, the more likely a forger is to guess it correctly. Unlike hash functions, the birthday paradox attack does not apply. Therefore, short MAC tags are often ideally secure for particular applications.

Q: How do I match up key length, MAC tag length, and advantage?

A: Your key length should ideally be as large as possible. There is often little practical value to using shorter keys. For instance, padding an AES-128 key with 88 bits of zeroes, effectively reducing it to a 40-bit key, may seem like it requires fewer resources. In fact, it saves no time or space and weakens the system. Ideally, for a MAC tag length of *w*-bits, you wish to give your attacker an advantage of no more than 2^{-w}. For instance, if you are going to send 2^{40} blocks of message data with CMAC-AES, the attacker's advantage is no less than 2^{-46}. In this case, a tag longer than 46 bits is actually wasteful as you approach the $2^{40\text{ th}}$ block of message data. On the other hand, if you are sending a trivial amount of message blocks, the advantage is very small and the tag length can be customized to suit bandwidth needs.

Q: Why can I not use hash(key || message) as a MAC function?

A: Such a construction is not resistant to offline attacks and is also vulnerable to message extension attacks. Forging messages is trivial with this scheme.

Q: What is a replay attack?

A: A replay attack can occur when you break a larger message into smaller independent pieces (e.g., packets). The attacker exploits the fact that unless you correlate the order of the packets, the attacker can change the meaning of the message simply by re-arranging the order of the packets. While each individual packet may be authenticated, it is not being modified. Thus, the attack goes unnoticed.

Q: Why do I care?

A: Without replay protection, an attacker can change the meaning of the overall message. Often, this implies the attacker can re-issue statements or commands. An attacker could, for instance, re-issue shell commands sent by a remote login shell.

Q: How do I defeat replay attacks?

A: The most obvious solution is to have a method of correlating the packets to their overall (relative) order within the larger stream of packets that make up the message. The most obvious solutions are timestamp counters and simple incremental counters. In both cases, the counter is included as part of the message authenticated. Filtering based on previously authenticated counters prevents an attacker from re-issuing an old packet or issuing them out of stream order.

Q: How do I deal with packet loss or re-ordering?

A: Occasionally, packet loss and re-ordering are part of the communication medium. For example, UDP is a lossy protocol that tolerates packet loss. Even when packets are not lost, they are not guaranteed to arrive in any particular order (this is often a warning that does not arise in most networks). Out of order UDP is fairly rare on non-congested IPv4 networks. The meaning of the error depends on the context of the application. If you are working with UDP (or another lossy medium), packet loss and re-ordering are usually not malicious acts. The best practice is to reject the packet, possibly issue a synchronization message, and resume the protocol. Note that an attacker may exploit the resynchronization step to have a victim generate authenticated messages. On a relatively stable medium such as TCP, packet loss and reordering are usually a sign of malicious interference and should be treated as hostile. The usual action here is to drop the connection. (Commonly, this is argued to be a denial of service (DoS) attack vector. However, anyone with the ability to modify packets between you and another host can also simply filter all packets anyways.) There is no *added* threat by taking this precaution.

In both cases, whether the error is treated as hostile or benign, the packet should be dropped and not interpreted further up the protocol stack.

Q: What libraries provide MAC functionality?

A: LibTomCrypt provides a highly modular HMAC function for C developers. Crypto++ provides similar functionality for C++ developers. Limited HMAC support is also found in OpenSSL. LibTomCrypt also provides modular support for CMAC. At the time of this writing, neither Crypto++ or OpenSSL provide support for CMAC. By "modular," we mean that the HMAC and CMAC implementations are not tied to underlying algorithms. For instance, the HMAC code in LibTomCrypt can use any hash function that LibTomCrypt supports without changes to the API. This allows future upgrades to be performed in a more timely and streamlined fashion.

Q: What patents cover MAC functions?

A: Both HMAC and CMAC are patent free and can be used for any purpose. Various other MAC functions such as PMAC are covered by patents but are also not standard.

Chapter 7

Encrypt and Authenticate Modes

Solutions in this chapter:

- **Encrypt and Authenticate Modes**
- **Security Goals**
- **Standards**
- **Design of GCM and CCM Modes**
- **Putting It All Together**

☑ **Summary**

☑ **Solutions Fast Track**

☑ **Frequently Asked Questions**

Introduction

In Chapter 6,"Message Authentication Code Algorithms," we saw how we could use message authentication code (MAC) functions to ensure the authenticity of messages between two or more parties. The MAC function takes a message and secret key as input and produces a MAC tag as output. This tag, combined with the message, can be verified by any party who has the same secret key.

We saw how MAC functions are integral to various applications to avoid various attacks. That is, if an attacker can forge messages he could perform tasks we would rather he could not. We also saw how to secure a message broken into smaller packets for convenience. Finally, our example program combined both encryption and authentication into a frame encoder to provide both privacy and authentication. In particular, we use PKCS #5, a key derivation function to accept a master secret key, and produce a key for encryption and another key for the MAC function.

Would it not be nice, if we had some function F(K, P) that accepts a secret key K and message P and returns the pair of (C, T) corresponding to the ciphertext and MAC tag (respectively)? Instead of having to create, or otherwise supply, two secret keys to accomplish both goals, we could defer that process to some encapsulated standard.

Encrypt and Authenticate Modes

This chapter introduces a relatively new set of standards in the cryptographic world known as encrypt and authenticate modes. These modes of operations encapsulate the tasks of encryption and authentication into a single process. The user of these modes simply passes a single key, IV (or nonce), and plaintext. The mode will then produce the ciphertext and MAC tag. By combining both tasks into a single step, the entire operation is much easier to implement.

The catalyst for these modes is from two major sources. The first is to extract any performance benefits to be had from combining the modes. The second is to make authentication more attractive to developers who tend to ignore it. You are more likely to find a product that encrypts data, than to find one that authenticates data.

Security Goals

The security goals of encrypt and authenticate modes are to ensure the privacy and authenticity of messages. Ideally, breaking one should not weaken the other. To achieve these goals, most combined modes require a secret key long enough such that an attacker could not guess it. They also require a unique IV per invocation to ensure replay attacks are not possible. These unique IVs are often called nonces in this context. The term *nonce* actually comes from N_{once}, which means to use N once and only once.

We will see later in this chapter that we can use the nonce as a packet counter when the secret key is randomly generated. This allows for ease of integration into existing protocols.

Standards

Even though encrypt and authenticate modes are relatively new, there are still a few good standards covering their design. In May 2004, NIST specified CCM as SP 800-38C, the first NIST encrypt and authenticate mode. Specified as a mode of operation for block ciphers, it was intended to be used with a NIST block cipher such as AES. CCM was selected as the result of a design contest in which various proposals were sought out. Of the more likely contestants to win were Galois Counter Mode (GCM), EAX mode, and CCM.

GCM was designed originally to be put to use in various wireless standards such as 802.16 (WiMAX), and later submitted to NIST for the contest. GCM is not yet a NIST standard (it is proposed as SP 800-38D), but as it is used through IEEE wireless standards it is a good algorithm to know about. GCM strives to achieve hardware performance by being massively parallelizable. In software, as we shall see, GCM can achieve high performance levels with the suitable use of the processor's cache.

Finally, EAX mode was proposed after the submission of CCM mode to address some of the shortcomings in the design. In particular, EAX mode is more flexible in terms of how it can be used and strives for higher performance (which turns out to not be true in practice). EAX mode is actually a properly constructed wrapper around CTR encryption mode and CMAC authentication mode. This makes the security analysis easier, and the design more worthy of attention. Unfortunately, EAX was not, and is currently not, considered for standardization. Despite this, EAX is still a worthy mode to know about and understand.

Design and Implementation

We shall consider the design, implementation, and optimization of three popular algorithms. We will first explore the GCM algorithm, which has already found practical use in the IEEE 802 series of standards. The reader should take particular interest in this design, as it is also likely to become a NIST standard. After GCM, we will explore the design of CCM, the only NIST standardized mode at the time of this writing. CCM is both efficient and secure, making it a mode worth using and knowing about. `

Additional Authentication Data

All three algorithms include an input known as the additional authentication data (AAD, also known as header data in CCM). This allows the implementer to include data that accompanies the ciphertext, and must be authenticated but does not have to be encrypted; for example, metadata such as packet counters, timestamps, user and host names, and so on.

AAD is unique to these modes and is handled differently in all three. In particular, EAX has the most flexible AAD handling, while GCM and CCM are more restrictive. All three modes accept empty AAD strings, which allows developers to ignore the AAD facilities if they do not need them.

Design of GCM

GCM (Galois Counter Mode) is the design of David McGraw and John Viega. It is the product of universal hashing and CTR mode encryption for security. The original motivation for GCM mode was fast hardware implementation. As such, GCM employs the use of GF(2^{128}) multiplication, which can be efficient in typical FPGA and other hardware implementations.

To properly discuss GCM, we have to unravel an implementer's worst nightmare—bit ordering. That is, which bit is the most significant bit, how are they ordered, and so on. It turns out that GCM is not one of the most straightforward designs in this respect. Once we get past the Galois field math, the rest of GCM is relatively easy to specify.

GCM GF(2) Mathematics

GCM employs multiplications in the field GF(2^{128})$[x]/\mathrm{v}(x)$ to perform a function it calls GHASH. Effectively, GHASH is a form of universal hashing, which we will discuss next. The multiplication we are performing here is not any different in nature than the multiplications used within the AES block cipher. The only differences are the size of the field and the irreducible polynomial used.

GCM uses a bit ordering that does not seem normal upon first inspection. Instead of storing the coefficients of the polynomials from the least significant bit upward, they store them backward. For example, from AES we would see that the polynomial $\mathrm{p}(x) = x^7 + x^3 + x + 1$ would be represented by 0x8B. In the GCM notation, the bits are reversed. In GCM notation, x^7 would be 0x01 instead of 0x80, so our polynomial $\mathrm{p}(x)$ would be represented as 0xD1 instead. In effect, the bytes are in little endian fashion. The bytes themselves are arranged in big endian fashion, which further complicates things. That is, byte number 15 is the least significant byte, and byte number 0 is the most significant byte.

The multiplication routine is then implemented with the following routines:

```
static void gcm_rightshift(unsigned char *a)
{
   int x;
   for (x = 15; x > 0; x--) {
       a[x] = (a[x]>>1) | ((a[x-1]<<7)&0x80);
   }
   a[0] >>= 1;
}
```

This performs what GCM calls a *right shift operation*. Numerically, it is equivalent to a left shift (multiplication by 2), but since we order the bits in each byte in the opposite direction, we use a right shift to perform this. We are shifting from byte 15 down to byte 0.

```
static const unsigned char mask[] = {
    0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01
};
static const unsigned char poly[] = { 0x00, 0xE1 };
```

The mask is a simple way of masking off bits in the byte in reverse order. The poly array is the least significant byte of the polynomial, the first element is a zero, and the second element is the byte of the polynomial. In this case, 0xE1 maps to $p(x) = x^{128} + x^7 + x^2 + x + 1$ where the x^{128} term is implicit.

```
void gcm_gf_mult(const unsigned char *a,
                 const unsigned char *b,
                       unsigned char *c)
{
   unsigned char Z[16], V[16];
   unsigned x, y, z;

   memset(Z, 0, 16);
   memcpy(V, a, 16);
   for (x = 0; x < 128; x++) {
       if (b[x>>3] & mask[x&7]) {
          for (y = 0; y < 16; y++) {
              Z[y] ^= V[y];
          }
       }
       z     = V[15] & 0x01;
       gcm_rightshift(V);
       V[0] ^= poly[z];
   }
   memcpy(c, Z, 16);
}
```

This routine accomplishes the operation $c = ab$ in the Galois field chosen by GCM. It effectively is the same algorithm we used for the multiplication in AES, except here we are using an array of bytes to represent the polynomials. We use Z to accumulate the product as we produce it. We use V as a copy of a, which we can double and selectively add to Z based on the bits of b.

This multiplication routine accomplishes numerically what we require, but is horribly slow. Fortunately, there is more than one way to multiply field elements. As we shall see during the implementation phase, a table-based multiplication routine will be far more profitable.

The curious reader may wish to examine the GCM source of LibTomCrypt for the variety of tricks that are optionally used depending on the configuration. In addition to the previous routine, LibTomCrypt provides an alternative to gcm_gf_mult() (see src/encauth/gcm/gcm_gf_mult.c in LibTomCrypt) that uses a windowed multiplication on whole words (Darrel Hankerson, Alfred Menezes, Scott Vanstone, "Guide to Elliptic Curve Cryptography," p. 50, Algorithm 2.36). This becomes important during the setup phase of GCM, even when we use a table-based multiplication routine for bulk data processing. Before we can show you a table-based multiplication routine, we must show you the constraints on GCM that make this possible.

Universal Hashing

Universal hashing is a method of creating a function f(x) such that for distinct values of x and y, the probability of f(x) = f(y) is that of any proper random function. The simplest example of such a universal hash is the mapping

$$f(x) = (ax + b \bmod p) \bmod n$$

for random values of a and b and random primes p and n ($n < p$). Universal MAC functions, such as those in GCM (and other algorithms such as Daniel Bernstein's Poly1305) use a variation of this to achieve a secure MAC function

$$H[i] = (H[i - 1] * K) + M[i]$$

where the last H[i] value is the tag, K is a unit in a finite field and the secret key, and M[i] is a block of the message. The multiplication and addition must be performed in a finite field of considerable size (e.g., 2^{128} units or more). In the case of GCM, we will create the MAC functionality, called GHASH, with this scheme using our GF(2^{128}) multiplication routine.

GCM Definitions

The entire GCM algorithm can be specified by a series of equations. First, let us define the various symbols we will be using in the equations (Figure 7.1).

- Let K represent the secret key.
- Let A represent the additional authentication data, there are m blocks of data in A.
- Let P represent the plaintext, there are n blocks of data in P.
- Let C represent the ciphertext.
- Let Y represent the CTR counters.
- Let T represent the MAC tag.
- Let E(K, P) represent the encryption of P with the secret key K and block cipher E (e.g., E = AES).
- Let IV represent the IV for the message to be processed.

Figure 7.1 GCM Data Processing

```
Input
    $P$:     Plaintext
    $K$:     Secret Key
    $A$:     Additional Authentication Data
    $IV$:    GCM Initial Vector
Output
    $C$:     Ciphertext
    $T$:     MAC Tag

1. $H = E(K, 0)$
2. If length($IV$) = 96
       1. $Y_0 = IV \,||\, 0^{31}1$
   else
       2. $Y_0 = \text{GHASH}(H, \{\}, IV)$
3. $Y_i = Y_{i-1} + 1$, for i = 1, ..., n
4. $C_i = P_i$ XOR E(K, $Y_i$), for i = 1, ..., n – 1
5. $C_n = P_n$ XOR E(K, $Y_n$), truncated to the length of $P_n$
6. $T = \text{GHASH}(H, A, C)$ XOR E($K$, $Y_0$)
7. Return C and T.
```

The first step is to generate the universal MAC key *H*, which is used solely in the GHASH function. Next, we need an IV for the CTR mode. If the user-supplied IV is 96 bits long, we use it directly by padding it with 31 zero bits and 1 one bit. Otherwise, we apply the GHASH function to the IV and use the returned value as the CTR IV.

Once we have *H* and the initial Y_0 value, we can encrypt the plaintext. The encryption is performed in CTR mode using the counter in big endian fashion. Oddly enough, the bits per byte of the counter are treated in the normal ordering. The last block of ciphertext is not expanded if it does not fill a block with the cipher. For instance, if P_n is 32 bits, the output of $E(K,Y_n)$ is truncated to 32 bits, and C_n is the 32-bit XOR of the two values.

Finally, the MAC tag is produced by performing the GHASH function on the additional authentication data and ciphertext. The output of GHASH is then XORed with the encryption of the initial Y_0 value. Next, we examine the GHASH function (Figure 7.2).

Figure 7.2 GCM GHASH Function

```
Input
    $H$:    Secret Parameter (derived from the secret key)
    $A$:    Additional Authentication Data ($m$ blocks)
    $C$:    Ciphertext (also used as an additional input source, $n$ blocks)
Output
    $T$:    GHASH Output
1. $X_0 = 0$
2. For $i$ from 1 to $m$ do
    1. $X_i = (X_{i-1}$ XOR $A_i) * H$
3. For $i$ from 1 to $n$ do
    1. $X_{i+m} = (X_{i+m-1}$ XOR $C_i) * H$
4. $T = (X_{m+n}$ XOR $(\text{length}(A)||\text{length}(C)) * H$
5. Return $T$
```

The GHASH function compresses the additional authentication data and ciphertext to a final MAC tag. The multiplication by H is a GF(2^{128})[x] multiplication as mentioned earlier. The length encodings are 64-bit big endian strings concatenated to one another. The length of A stored in the first 8 bytes and the length of C in the last 8.

Implementation of GCM

While the description of GCM is nice and concise, the implementation is not. First, the multiplication requires careful optimization to get decent performance. Second, the flexibility of the IV, AAD, and plaintext processing requires careful state transitions. Originally, we had planned to write a GCM implementation from scratch for this text. However, we later decided our readers would be better served if we simply used an existing optimized implementation.

To demonstrate GCM, we used the implementation of LibTomCrypt. This implementation is public domain, freely accessible on the project's Web site, optimized, and easy to follow. We will omit various administrative portions of the code to reduce the size of the code listings. Readers are strongly encouraged to use the routines found in LibTomCrypt (or similar libraries) instead of rolling their own if they can get away with it.

Interface

Our GCM interface has several functions that we will discuss in turn. The high level of abstraction allows us to use the GCM implementation to the full flexibility warranted by the GCM specification. The functions we will discuss are:

1.	gcm_gf_mult()	Generic GF(2^{128})[*x*] multiplication
2.	gcm_mult_h()	Multiplication by H (usually optimized since H is fixed after setup)
3.	gcm_init()	Initialize a GCM state
4.	gcm_add_iv()	Add IV data to the GCM state
5.	gcm_add_aad()	Add AAD to the GCM state
6.	gcm_process()	Add plaintext to the GCM state
7.	gcm_done()	Terminate the GCM state and return the MAC tag

These functions all combine to allow a caller to process a message through the GCM algorithm. For any message, the functions 3 through 7 are meant to be called in that order to process the message. That is, one must add the IV before the AAD, and the AAD before the plaintext. GCM does not allow for processing the distinct data elements in other orders. For example, you cannot add AAD before the IV. The functions can be called multiple times as long as the order of the appearance is intact. For example, you can call gcm_add_iv() twice before calling gcm_add_aad() for the first time.

All the functions make use of the structure *gcm_state*, which contains the current working state of the GCM algorithm. It fully determines how the functions should behave, which allows the functions to be fully thread safe (Figure 7.3).

Figure 7.3 GCM State Structure

```
typedef struct {
   symmetric_key       K;
   unsigned char       H[16],    /* multiplier */
                       X[16],    /* accumulator */
                       Y[16],    /* counter */
                       Y_0[16],  /* initial counter */
                       buf[16];  /* buffer for stuff */

   int                 cipher,   /* which cipher */
                       ivmode,   /* Which mode is the IV in? */
                       mode,     /* mode the GCM code is in */
                       buflen;  /* length of data in buf */

   ulong64             totlen,   /* 64-bit counter used for IV and AAD */
                       pttotlen; /* 64-bit counter for the PT */

#ifdef GCM_TABLES
   unsigned char       PC[16][256][16];  /* 16 tables of 8x128 */
#endif
} gcm_state;
```

As we can see, the state has quite a few members. Table 7.1 explains their function.

Table 7.1 *gcm_state* Members and Their Functions

Member Name	Purpose
K	Scheduled cipher key, used to encrypt counters.
H	GHASH multiplier value.
X	GHASH accumulator.
Y	CTR mode counter value (incremented as text is processed).
Y_0	The initial counter value used to encrypt the GHASH output.
buf	Used in various places; for example, holds the encrypted counter values.
cipher	ID of which cipher we are using with GCM.
ivmode	Specifies whether we are working with a short IV. It is set to nonzero if the IV is longer than 12 bytes.
mode	Current mode GCM is in. Can be one of the following: GCM_MODE_IV GCM_MODE_AAD GCM_MODE_TEXT
buflen	Current length of data in the *buf* array.
totlen	Total length of the IV and AAD data.
pttotlen	Total length of the plaintext.
PC	A 16x256x16 table such that *PC*[*i*][*j*][*k*] is the *k*th byte of $H * j * x^{8i}$ in GF(2^{128})[*x*] This table is pre-computed by gcm_init() based on the secret H value to accelerate the multiplication by *H* required by the GHASH function.

The PC table is an optional table only included if GCM_TABLES was defined at build time. As we will see shortly, it can greatly speed up the processing of data through GHASH; however, it requires a 64 kilobyte table, which could easily be prohibitive in various embedded platforms.

GCM Generic Multiplication

The following code implements the generic GF(2^{128})[*x*] multiplication required by GCM. It is designed to work with any multiplier values and is not optimized to the GHASH usage pattern of multiplying by a single value (*H*).

gcm_gf_mult.c:

```
/* this is x*2^128 mod p(x) ... the results are 16 bytes
 * each stored in a packed format.  Since only the
 * lower 16 bits are not zero'ed I removed the upper 14 bytes */
const unsigned char gcm_shift_table[256*2] = {
0x00, 0x00, 0x01, 0xc2, 0x03, 0x84, 0x02, 0x46,
0x07, 0x08, 0x06, 0xca, 0x04, 0x8c, 0x05, 0x4e,
0x0e, 0x10, 0x0f, 0xd2, 0x0d, 0x94, 0x0c, 0x56,
0x09, 0x18, 0x08, 0xda, 0x0a, 0x9c, 0x0b, 0x5e,
<snip>
0xb5, 0xe0, 0xb4, 0x22, 0xb6, 0x64, 0xb7, 0xa6,
0xb2, 0xe8, 0xb3, 0x2a, 0xb1, 0x6c, 0xb0, 0xae,
0xbb, 0xf0, 0xba, 0x32, 0xb8, 0x74, 0xb9, 0xb6,
0xbc, 0xf8, 0xbd, 0x3a, 0xbf, 0x7c, 0xbe, 0xbe };
```

This table contains the residue of the value of k * x^{128} mod p(x) for all 256 values of k. Since the value of p(x) is sparse, only the lower two bytes of the residue are nonzero. As such, we can compress the table. Every pair of bytes are the lower two bytes of the residue for the given value of *k*. For instance, gcm_shift_table[3] and gcm_shift_table[4] are the value of the least significant bytes of 2 * x^{128} mod p(x).

This table is only used if LTC_FAST is defined. This define instructs the implementation to use a fast parallel XOR operations on words instead of on the byte level. In our case, we can exploit it to perform the generic multiplication much faster.

```
#ifndef LTC_FAST
/* right shift */
static void gcm_rightshift(unsigned char *a)
{
   int x;
   for (x = 15; x > 0; x--) {
       a[x] = (a[x]>>1) | ((a[x-1]<<7)&0x80);
   }
   a[0] >>= 1;
}
```

This function performs the right shift (multiplication by *x*) using GCM conventions.

```
/* c = b*a */
static const unsigned char mask[] =
   { 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01 };
static const unsigned char poly[] =
   { 0x00, 0xE1 };


/**
  GCM GF multiplier (internal use only)  bitserial
  @param a   First value
  @param b   Second value
  @param c   Destination for a * b
 */
void gcm_gf_mult(const unsigned char *a,
                 const unsigned char *b,
                       unsigned char *c)
{
   unsigned char Z[16], V[16];
   unsigned x, y, z;

   zeromem(Z, 16);
   XMEMCPY(V, a, 16);
   for (x = 0; x < 128; x++) {
       if (b[x>>3] & mask[x&7]) {
          for (y = 0; y < 16; y++) {
              Z[y] ^= V[y];
          }
       }
       z     = V[15] & 0x01;
       gcm_rightshift(V);
       V[0] ^= poly[z];
   }
   XMEMCPY(c, Z, 16);
}
```

This is the slow bit serial approach. We use the LibTomCrypt functions zeromem (similar to memset) and XMEMCPY (defaults to memcpy) for portability issues. Many small C platforms do not have full C libraries. These functions (and macros) allow developers to work around such limitations in a safe manner.

```
#else

/* map normal numbers to "ieee" way ... e.g. bit reversed */
#define M(x) (((x&8)>>3) | ((x&4)>>1) | ((x&2)<<1) | ((x&1)<<3))
#define BPD (sizeof(LTC_FAST_TYPE) * 8)
#define WPV (1 + (16 / sizeof(LTC_FAST_TYPE)))
```

These are some macros we use in the faster generic multiplier. The M() macro maps a four-bit nibble to the GCM convention (e.g., reverse).

The LTC_FAST_TYPE symbol refers to a data type defined in LibTomCrypt that represents an optimal data type that is a multiple of eight bits in length. For example, on 32-bit platforms, it is a *unsigned long*. The data type has to overlap perfectly with the *unsigned char* data type. It is used to allow parallel XOR operations.

The BPD macro is the number of bytes per LTC_FAST_TYPE. Clearly, this only works if CHAR_BIT is 8, which is why LTC_FAST is not enabled by default. The WPV macro is the number of words per 128-bit value plus a word.

```
/**
  GCM GF multiplier (internal use only)  word oriented
  @param a   First value
  @param b   Second value
  @param c   Destination for a * b
 */
void gcm_gf_mult(const unsigned char *a,
                 const unsigned char *b,
                       unsigned char *c)
{
   int i, j, k, u;
   LTC_FAST_TYPE   B[16][WPV],
                 tmp[32 / sizeof(LTC_FAST_TYPE)],
                  pB[16 / sizeof(LTC_FAST_TYPE)],
                  zz, z;
   unsigned char pTmp[32];
```

The *B* array contains the computed values of *ka* for *k*=0...15. It allows us to perform a 4x128 multiplication with a table lookup. The *tmp* array contains the product (before it has been reduced). The *pB* array contains the loaded and converted copy of *b* with the appropriate treatment for the GCM order of the bits.

```
   /* create simple tables */
   zeromem(B[0],       sizeof(B[0]));
   zeromem(B[M(1)],    sizeof(B[M(1)]));

#ifdef ENDIAN_32BITWORD
   for (i = 0; i < 4; i++) {
       LOAD32H(B[M(1)][i], a + (i<<2));
       LOAD32L(pB[i],      b + (i<<2));
   }
#else
   for (i = 0; i < 2; i++) {
       LOAD64H(B[M(1)][i], a + (i<<3));
       LOAD64L(pB[i],      b + (i<<3));
   }
#endif
```

The preceding code loads the bytes of *a* and *b* into their respective arrays. A curious reader may note that we load *a* with a big endian macro and *b* with a little endian macro. The *a* value is loaded in big endian fashion to adhere to the GCM specs. The *b* value is loaded in the opposite fashion so we can use a more straightforward digit extraction expression.

In fact, we could load both as big endian, and merely rewrite the order in which we fetch nibbles to compensate.

```
   /* now create 2, 4 and 8 */
   B[M(2)][0] = B[M(1)][0] >> 1;
   B[M(4)][0] = B[M(1)][0] >> 2;
   B[M(8)][0] = B[M(1)][0] >> 3;
   for (i = 1; i < (int)WPV; i++) {
   B[M(2)][i] =(B[M(1)][i-1] << (BPD-1)) | (B[M(1)][i] >> 1);
   B[M(4)][i] =(B[M(1)][i-1] << (BPD-2)) | (B[M(1)][i] >> 2);
   B[M(8)][i] =(B[M(1)][i-1] << (BPD-3)) | (B[M(1)][i] >> 3);
   }
```

This block of code creates the entries for *ax*, ax^2, and ax^3. Note that we do not perform any reductions. This is why WPV has an extra word appended to it, since we are dealing with values that have more than 128 bits in them.

```
   /*  now all values with two bits which are
    *  3, 5, 6, 9, 10, 12 */
   for (i = 0; i < (int)WPV; i++) {
      B[M(3)][i]  = B[M(1)][i] ^ B[M(2)][i];
      B[M(5)][i]  = B[M(1)][i] ^ B[M(4)][i];
      B[M(6)][i]  = B[M(2)][i] ^ B[M(4)][i];
      B[M(9)][i]  = B[M(1)][i] ^ B[M(8)][i];
      B[M(10)][i] = B[M(2)][i] ^ B[M(8)][i];
      B[M(12)][i] = B[M(8)][i] ^ B[M(4)][i];

   /* now all 3 bit values and the only 4 bit value:
    * 7, 11, 13, 14, 15 */
      B[M(7)][i]  = B[M(3)][i] ^ B[M(4)][i];
      B[M(11)][i] = B[M(3)][i] ^ B[M(8)][i];
      B[M(13)][i] = B[M(1)][i] ^ B[M(12)][i];
      B[M(14)][i] = B[M(6)][i] ^ B[M(8)][i];
      B[M(15)][i] = B[M(7)][i] ^ B[M(8)][i];
   }
```

These two blocks construct the rest of the entries word per word. We first construct the values that only have two bits set (3, 5, 6, 9, 10, and 12), and then from those we construct the values that have three bits set. Note the use of the M() macro, which evaluates to a constant at compile time.

```
    zeromem(tmp, sizeof(tmp));

    /* compute product four bits of each word at a time */
    /* for each nibble */
    for (i = (BPD/4)-1; i >= 0; i--) {
        /* for each word */
        for (j = 0; j < (int)(WPV-1); j++) {
         /* grab the 4 bits recall the nibbles are
            backwards so it's a shift by (i^1)*4 */
            u = (pB[j] >> ((i^1)<<2)) & 15;
```

Here we are extracting a nibble of *b* to multiply *a* by. Note the use of (*i*^1) to extract the nibbles in reverse order since GCM stores bits in each byte in reverse order.

```
         /* add offset by the word count the table
            looked up value to the result */
            for (k = 0; k < (int)WPV; k++) {
                tmp[k+j] ^= B[u][k];
            }
        }
```

This loop multiplies each nibble of each word of *b* by *a,* and adds it to the appropriate offset within *tmp.* The product of the *i*th nibble of the *j*th word is added to tmp[*j*...*j*+WPV–1].

```
      /* shift result up by 4 bits */
        if (i != 0) {
        for (z=j=0; j < (int)(32 / sizeof(LTC_FAST_TYPE)); j++) {
           zz = tmp[j] << (BPD-4);
           tmp[j] = (tmp[j] >> 4) | z;
           z = zz;
        }
        }
    }
```

After we have added all of the products regarding the *i*th nibbles of each word, we shift the entire product (tmp) up by four bits.

```
    /* store product */
 #ifdef ENDIAN_32BITWORD
    for (i = 0; i < 8; i++) {
        STORE32H(tmp[i], pTmp + (i<<2));
    }
 #else
    for (i = 0; i < 4; i++) {
        STORE64H(tmp[i], pTmp + (i<<3));
    }
 #endif

    /* reduce by taking most significant byte and adding the
       appropriate two byte sequence 16 bytes down */
    for (i = 31; i >= 16; i--) {
        pTmp[i-16] ^= gcm_shift_table[((unsigned)pTmp[i]<<1)];
        pTmp[i-15] ^= gcm_shift_table[((unsigned)pTmp[i]<<1)+1];
    }
```

This reduction makes use of the fact that for any $j > 15$, the value of kx^j mod p(x) is congruent to $(kx^{16})x^{j-16}$. Since we have a nice table for kx^{16} mod p(x), we can compute $(kx^{16})x^{j-16}$ by a table look up and shift. This routine adds the residue of the product from the high byte to the lower bytes.

Each loop of the preceding for loop removes one byte from the product at a time. We perform the shift inline by adding the lookup values to pTmp[i–16] and pTmp[i–15].

```
   for (i = 0; i < 16; i++) {
       c[i] = pTmp[i];
   }

}

#endif
```

Both implementations of gcm_gf_mult() accomplish the same goal and are numerically equivalent. The latter implementation is much faster on 32- and 64-bit processors but is not 100-percent portable. It requires a data type that is a multiple of a *unsigned char* data type in size, which is not always guaranteed.

Now that we have a generic multiplier, we have to implement an optimized multiplier to be used by GHASH.

GCM Optimized Multiplication

The following multiplication routine is optimized solely for performing a multiplication by the secret *H* value. It takes advantage of the fact we can precompute tables for the multiplication.

gcm_mult_h.c:

```
/**
  GCM multiply by H
  @param gcm   The GCM state which holds the H value
  @param I     The value to multiply H by
 */
void gcm_mult_h(gcm_state *gcm, unsigned char *I)
{
   unsigned char T[16];
#ifdef GCM_TABLES
```

```
   int x, y;
   XMEMCPY(T, &gcm->PC[0][I[0]][0], 16);
```

If GCM_TABLES has been defined, we will use the tables approach. The PC table contains 16 8x128 tables, one for each byte of the input and for each of their respective possible values. The first thing we must do is copy the 0th entry to T (our accumulator). The rest of the lookups will be XORed into this value.

```
   for (x = 1; x < 16; x++) {
#ifdef LTC_FAST
       for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
           *((LTC_FAST_TYPE *)(T + y)) ^=
           *((LTC_FAST_TYPE *)(&gcm->PC[x][I[x]][y]));
       }
#else
       for (y = 0; y < 16; y++) {
           T[y] ^= gcm->PC[x][I[x]][y];
       }
#endif
```

Here we see the use of LTC_FAST to optimize parallel XOR operations. For each byte of I, the input, we look up the 128-bit value and XOR it against the accumulator. Since the entries in the table have already been reduced, our accumulator never grows beyond 128 bits in size.

```
   }
#else
   gcm_gf_mult(gcm->H, I, T);
#endif
   XMEMCPY(I, T, 16);
}
```

If we are not using tables, we use the slower gcm_gf_mult() to achieve the operation.

Now that we have both of our multipliers out of the way, we can move on to the rest of the GCM algorithm starting with the initialization routine.

GCM Initialization

The first function is gcm_init(), which accepts a secret key and initializes the GCM state.

gcm_init.c:

```
/**
  Initialize a GCM state
  @param gcm     The GCM state to initialize
  @param cipher  The index of the cipher to use
  @param key     The secret key
  @param keylen  The length of the secret key
  @return CRYPT_OK on success
 */
int gcm_init(gcm_state *gcm, int cipher,
             const unsigned char *key,  int keylen)
{
   int           err;
   unsigned char B[16];
#ifdef GCM_TABLES
   int           x, y, z, t;
#endif

   LTC_ARGCHK(gcm != NULL);
   LTC_ARGCHK(key != NULL);

#ifdef LTC_FAST
   if (16 % sizeof(LTC_FAST_TYPE)) {
      return CRYPT_INVALID_ARG;
   }
#endif
```

This is a simple sanity check to make sure the code will actually work with LTC_FAST defined. It does not catch all error cases, but in practice is enough.

```
   /* is cipher valid? */
   if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
      return err;
   }
   if (cipher_descriptor[cipher].block_length != 16) {
      return CRYPT_INVALID_CIPHER;
   }

   /* schedule key */
   if ((err = cipher_descriptor[cipher].setup(key, keylen,
                                              0, &gcm->K)) !=
       CRYPT_OK) {
      return err;
   }
```

This code schedules the secret key to be used by the GCM code.

```
   /* H = E(0) */
   zeromem(B, 16);
   if ((err =
        cipher_descriptor[cipher].ecb_encrypt(B, gcm->H,
                                              &gcm->K)) !=
       CRYPT_OK) {
      return err;
   }
```

We encrypt the zero string to compute our secret multiplier value *H*.

```
   /* setup state */
   zeromem(gcm->buf, sizeof(gcm->buf));
   zeromem(gcm->X,   sizeof(gcm->X));
   gcm->cipher   = cipher;
   gcm->mode     = GCM_MODE_IV;
   gcm->ivmode   = 0;
   gcm->buflen  = 0;
   gcm->totlen   = 0;
   gcm->pttotlen = 0;
```

This block of code initializes the GCM state to the default empty and zero state. After this point, we are ready to process IV, AAD, or plaintext (provided GCM_TABLES was not defined).

```
#ifdef GCM_TABLES
   /* setup tables */

   /* generate the first table as it has no shifting
    * (from which we make the other tables) */
   zeromem(B, 16);
   for (y = 0; y < 256; y++) {
        B[0] = y;
        gcm_gf_mult(gcm->H, B, &gcm->PC[0][y][0]);
   }
```

If we are using tables, we first compute the lowest table, which is simply yH mod p(x) for all 256 values of y. We use the slower multiplier, since at this point we do not have tables to work with.

```
   /* now generate the rest of the tables
    * based the previous table */
   for (x = 1; x < 16; x++) {
      for (y = 0; y < 256; y++) {
         /* now shift it right by 8 bits */
         t = gcm->PC[x-1][y][15];
         for (z = 15; z > 0; z--) {
             gcm->PC[x][y][z] = gcm->PC[x-1][y][z-1];
         }
         gcm->PC[x][y][0]  = gcm_shift_table[t<<1];
         gcm->PC[x][y][1] ^= gcm_shift_table[(t<<1)+1];
     }
  }
```

This code block generates the 15 other 8x128 tables. Since the only difference between the 0th 8x128 table and the 1st 8x128 table is that we multiplied the values by x^8, we can perform this with a simple shift and XOR with the reduction table values (as we saw with the fast gcm_gf_mult() function). We can repeat the process for the 2nd, 3rd, 4th, and so on tables.

Using this shift and reduce trick is much faster than naively using gcm_gf_mult() to produce all 16 * 256 = 4096 multiplications required.

```
#endif

   return CRYPT_OK;
}
```

At this point, we are good to go with using GCM to process IV, AAD, or plaintext.

GCM IV Processing

The GCM IV controls the initial value of the CTR value used for encryption, and indirectly the MAC tag value since it is based on the ciphertext. Each packet should have a unique IV if the same key is being used. It is safe to use a packet counter as the IV, as long as it never repeats while using the same key.

gcm_add_iv.c:

```
/**
  Add IV data to the GCM state
  @param gcm    The GCM state
  @param IV     The initial value data to add
  @param IVlen  The length of the IV
  @return CRYPT_OK on success
 */
int gcm_add_iv(gcm_state *gcm,
               const unsigned char *IV,     unsigned long IVlen)
{
   unsigned long x, y;
   int           err;

   LTC_ARGCHK(gcm != NULL);
   if (IVlen > 0) {
      LTC_ARGCHK(IV  != NULL);
   }
```

This is a bit odd, but we do allow null IVs, which can also have IV == NULL.

```
   /* must be in IV mode */
   if (gcm->mode != GCM_MODE_IV) {
      return CRYPT_INVALID_ARG;
   }
```

We must be in IV mode to call this function. If we are not, this means we have called the AAD or process function (to handle plaintext).

```
   if (gcm->buflen >= 16 || gcm->buflen < 0) {
      return CRYPT_INVALID_ARG;
   }

   if ((err = cipher_is_valid(gcm->cipher)) != CRYPT_OK) {
      return err;
   }


   /* trip the ivmode flag */
   if (IVlen + gcm->buflen > 12) {
      gcm->ivmode |= 1;
   }
```

If we have more than 12 bytes of IV, we set the *ivmode* flag. The processing of the IV changes based on whether this flag is set.

```
   x = 0;
#ifdef LTC_FAST
   if (gcm->buflen == 0) {
      for (x = 0; x < (IVlen & ~15); x += 16) {
          for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
              *((LTC_FAST_TYPE*)(&gcm->X[y])) ^=
              *((LTC_FAST_TYPE*)(&IV[x + y]));
          }
          gcm_mult_h(gcm, gcm->X);
          gcm->totlen += 128;
      }
      IV += x;
   }
#endif
```

If we can use LTC_FAST, we will add bytes of the IV to the state a word at a time. We only use this optimization if there are 16 or more bytes of IV to add. Usually, IVs are short, so this should not be invoked, but it does allow a user to base the IV on a larger string if desired.

```
   /* start adding IV data to the state */
   for (; x < IVlen; x++) {
       gcm->buf[gcm->buflen++] = *IV++;

       if (gcm->buflen == 16) {
         /* GF mult it */
         for (y = 0; y < 16; y++) {
             gcm->X[y] ^= gcm->buf[y];
         }
         gcm_mult_h(gcm, gcm->X);
         gcm->buflen = 0;
         gcm->totlen += 128;
      }
   }
```

This block of code handles adding any leftover bytes of the IV to the state. It adds one byte at a time; once we accumulate 16, we XOR them against the state and call gcm_mult_h().

```
   return CRYPT_OK;
}
```

This function handles add IV data to the state. Note carefully that it does not actually terminate the GHASH or compute the initial counter value. That is handled in the next function, gcm_add_aad().

GCM AAD Processing

Additional Authentication Data (AAD) is metadata you can add to the stream being authenticated that is not encrypted. It must be added after the IV and before the plaintext (or ciphertext) has been processed. The AAD step can be skipped if there is no data to add to the stream. Typically, the AAD would be nonprivacy, but stream or packet related data such as unique identifiers or placement markers.

gcm_add_aad.c:

```
/**
  Add AAD to the GCM state
  @param gcm       The GCM state
  @param adata     The AAD to add to the GCM state
  @param adatalen  The length of the AAD data.
  @return CRYPT_OK on success
 */
int gcm_add_aad(          gcm_state *gcm,
                const unsigned char *adata,
                      unsigned long adatalen)
{
   unsigned long x;
   int           err;
#ifdef LTC_FAST
   unsigned long y;
#endif

   LTC_ARGCHK(gcm    != NULL);
   if (adatalen > 0) {
      LTC_ARGCHK(adata  != NULL);
   }

   if (gcm->buflen > 16 || gcm->buflen < 0) {
      return CRYPT_INVALID_ARG;
   }

   if ((err = cipher_is_valid(gcm->cipher)) != CRYPT_OK) {
      return err;
   }
```

So far, this is all the same style of error checking. The only reason we do not place this in another function is to make sure the LTC_ARGCHK macros report the correct function name (they work much like the assert macro).

```
   /* in IV mode? */
   if (gcm->mode == GCM_MODE_IV) {
      /* let's process the IV */
```

This block of code finishes processing the IV, if any.

```
      if (gcm->ivmode || gcm->buflen != 12) {
```

If we have tripped the IV flag or the IV accumulated length is not 12, we have to apply GHASH to the IV data to produce our starting *Y* value.

```
         for (x = 0; x < (unsigned long)gcm->buflen; x++) {
             gcm->X[x] ^= gcm->buf[x];
         }
         if (gcm->buflen) {
            gcm->totlen += gcm->buflen * CONST64(8);
            gcm_mult_h(gcm, gcm->X);
         }
```

At this point, we have emptied the IV buffer and multiplied the GCM state by the secret *H* value.

```
         /* mix in the length */
         zeromem(gcm->buf, 8);
         STORE64H(gcm->totlen, gcm->buf+8);
         for (x = 0; x < 16; x++) {
             gcm->X[x] ^= gcm->buf[x];
         }
         gcm_mult_h(gcm, gcm->X);
```

Next, we append the length of the IV and multiply that by *H*. The result is the output of GHASH and the starting *Y* value.

```
         /* copy counter out */
         XMEMCPY(gcm->Y, gcm->X, 16);
         zeromem(gcm->X, 16);
      } else {
         XMEMCPY(gcm->Y, gcm->buf, 12);
         gcm->Y[12] = 0;
         gcm->Y[13] = 0;
         gcm->Y[14] = 0;
         gcm->Y[15] = 1;
      }
```

This block of code handles the case that our IV is 12 bytes (because the flag has not been tripped and the buflen is 12). In this case, processing the IV means simply copying it to the GCM state and setting the last 32 bits to the big endian version of "1".

Ideally, you want to use the 12-byte IV in GCM since it allows fast packet processing. If your GCM key is randomly derived per session, the IV can simply be a packet counter. As long as they are all unique, we are using GCM properly.

```
      XMEMCPY(gcm->Y_0, gcm->Y, 16);
      zeromem(gcm->buf, 16);
      gcm->buflen = 0;
      gcm->totlen = 0;
      gcm->mode   = GCM_MODE_AAD;
   }

   if (gcm->mode != GCM_MODE_AAD || gcm->buflen >= 16) {
      return CRYPT_INVALID_ARG;
   }
```

At this point, we check that we are indeed in AAD mode and that our buflen is a legal value.

```
   x = 0;
#ifdef LTC_FAST
   if (gcm->buflen == 0) {
      for (x = 0; x < (adatalen & ~15); x += 16) {
          for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
              *((LTC_FAST_TYPE*)(&gcm->X[y])) ^=
              *((LTC_FAST_TYPE*)(&adata[x + y]));
          }
          gcm_mult_h(gcm, gcm->X);
          gcm->totlen += 128;
      }
      adata += x;
   }
```

```
#endif
```

If we have LTC_FAST enabled, we will process AAD data in 16-byte increments by quickly XORing it into the GCM state. This avoids all the manual single-byte XOR operations.

```
   /* start adding AAD data to the state */
   for (; x < adatalen; x++) {
       gcm->X[gcm->buflen++] ^= *adata++;

       if (gcm->buflen == 16) {
         /* GF mult it */
         gcm_mult_h(gcm, gcm->X);
         gcm->buflen = 0;
         gcm->totlen += 128;
      }
   }
```

This is the default processing of AAD data. In the event LTC_FAST has been enabled, it handles any lingering bytes. It is ideal to have AAD data that is always a multiple of 16 bytes in length. This way, we can avoid the slower manual byte XORs.

```
   return CRYPT_OK;
}
```

GCM Plaintext Processing

Processing the plaintext is the final step in processing a GCM message after processing the IV and AAD. Since we are using CTR to encrypt the data, the ciphertext is the same length as the plaintext.

gcm_process.c:

```
/**
  Process plaintext/ciphertext through GCM
  @param gcm       The GCM state
  @param pt        The plaintext
  @param ptlen     The plaintext length
  @param ct        The ciphertext
  @param direction Encrypt or Decrypt mode
  @return CRYPT_OK on success
 */
int gcm_process(    gcm_state *gcm,
                unsigned char *pt, unsigned long ptlen,
                unsigned char *ct,
                           int direction)
{
   unsigned long x, y;
   unsigned char b;
   int           err;

   LTC_ARGCHK(gcm != NULL);
   if (ptlen > 0) {
      LTC_ARGCHK(pt  != NULL);
      LTC_ARGCHK(ct  != NULL);
   }

   if (gcm->buflen > 16 || gcm->buflen < 0) {
      return CRYPT_INVALID_ARG;
   }

   if ((err = cipher_is_valid(gcm->cipher)) != CRYPT_OK) {
      return err;
   }
```

Same error checking code. We can call this function with an empty plaintext just like we can call the previous two functions with empty IV and AAD blocks. GCM is valid with zero length plaintexts and can be used to produce a MAC tag for the AAD data. It is not a good idea to use GCM this way, as CMAC is more efficient for the purpose (and does not require huge tables for efficiency). In short, if all you want is a MAC tag, do not use GCM.

```
   /* in AAD mode? */
   if (gcm->mode == GCM_MODE_AAD) {
      /* let's process the AAD */
      if (gcm->buflen) {
         gcm->totlen += gcm->buflen * CONST64(8);
         gcm_mult_h(gcm, gcm->X);
      }
```

At this point, we have finished processing the AAD. Note that unlike processing the IV, we do not terminate the GHASH.

```
      /* increment counter */
      for (y = 15; y >= 12; y--) {
          if (++gcm->Y[y] & 255) { break; }
      }
      /* encrypt the counter */
      if ((err =
         cipher_descriptor[gcm->cipher].ecb_encrypt(gcm->Y,
                                                     gcm->buf,
                                                     &gcm->K))
         != CRYPT_OK) {
         return err;
      }
```

We increment the initial value of Y and encrypt it to the buf array. This buffer is our CTR key stream used to encrypt or decrypt the message.

```
      gcm->buflen = 0;
      gcm->mode   = GCM_MODE_TEXT;
   }

   if (gcm->mode != GCM_MODE_TEXT) {
      return CRYPT_INVALID_ARG;
   }
```

At this point, we are ready to process the plaintext. We again will use a LTC_FAST trick to handle the plaintext efficiently. Since the GHASH is applied to the ciphertext, we must handle encryption and decryption differently. This is also because we allow the plaintext and ciphertext buffers supplied by the user to overlap.

```
   x = 0;
#ifdef LTC_FAST
   if (gcm->buflen == 0) {
      if (direction == GCM_ENCRYPT) {
         for (x = 0; x < (ptlen & ~15); x += 16) {
```

Again, we attempt to process the plaintext 16 bytes at a time. For this reason, ensuring your plaintext is a multiple of 16 bytes is a good idea.

```
             /* ctr encrypt */
             for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
                 *((LTC_FAST_TYPE*)(&ct[x + y])) =
                    *((LTC_FAST_TYPE*)(&pt[x+y])) ^
                    *((LTC_FAST_TYPE*)(&gcm->buf[y]));
                 *((LTC_FAST_TYPE*)(&gcm->X[y])) ^=
                    *((LTC_FAST_TYPE*)(&ct[x+y]));
             }
```

This loop XORs the CTR key stream against the plaintext, and then XORs the ciphertext into the GHASH accumulator. The loop may look complicated, but GCC does a good job to optimize the loop. In fact, the loop is usually fully unrolled and turned into simple load and XOR operations.

```
             /* GMAC it */
             gcm->pttotlen += 128;
             gcm_mult_h(gcm, gcm->X);
```

Since we are processing 16-byte blocks, we always perform the multiplication by H on the accumulated data.

```
             /* increment counter */
             for (y = 15; y >= 12; y--) {
                 if (++gcm->Y[y] & 255) { break; }
             }
             if ((err =
             cipher_descriptor[gcm->cipher].ecb_encrypt(
                    gcm->Y, gcm->buf, &gcm->K)) != CRYPT_OK){
                return err;
             }
```

We next increment the CTR counter and encrypt it to generate another 16 bytes of key stream.

```
         }
      } else {
         for (x = 0; x < (ptlen & ~15); x += 16) {
             /* ctr encrypt */
             for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
                 *((LTC_FAST_TYPE*)(&gcm->X[y])) ^=
                    *((LTC_FAST_TYPE*)(&ct[x+y]));
                 *((LTC_FAST_TYPE*)(&pt[x + y])) =
                    *((LTC_FAST_TYPE*)(&ct[x+y])) ^
                    *((LTC_FAST_TYPE*)(&gcm->buf[y]));
             }
             /* GMAC it */
             gcm->pttotlen += 128;
             gcm_mult_h(gcm, gcm->X);
             /* increment counter */
             for (y = 15; y >= 12; y--) {
                 if (++gcm->Y[y] & 255) { break; }
             }
             if ((err =
                  cipher_descriptor[gcm->cipher].ecb_encrypt(
                  gcm->Y, gcm->buf, &gcm->K)) != CRYPT_OK){
                return err;
             }
         }
     }
   }
#endif
```

This second block handles decryption. It is similar to encryption, but since we are shaving cycles, we do not merge them. The code size increase is not significant, especially compared to the time savings.

```
   /* process text */
   for (; x < ptlen; x++) {
```

This loop handles any remaining bytes for both encryption and decryption. Since we are processing the data a byte at a time, it is best to avoid needing this section of code in performance applications.

```
       if (gcm->buflen == 16) {
```

Every 16 bytes, we must accumulate them in the GHASH tag and update the CTR key stream.

```
          gcm->pttotlen += 128;
          gcm_mult_h(gcm, gcm->X);

          /* increment counter */
          for (y = 15; y >= 12; y--) {
              if (++gcm->Y[y] & 255) { break; }
          }
          if ((err=cipher_descriptor[gcm->cipher].ecb_encrypt(
               gcm->Y, gcm->buf, &gcm->K)) != CRYPT_OK) {
             return err;
          }
          gcm->buflen = 0;
       }

       if (direction == GCM_ENCRYPT) {
          b = ct[x] = pt[x] ^ gcm->buf[gcm->buflen];
       } else {
          b = ct[x];
```

```
136             pt[x] = ct[x] ^ gcm->buf[gcm->buflen];
137          }
138          gcm->X[gcm->buflen++] ^= b;
```

This last bit is seemingly overly complicated but done so by design. We allow ct = pt, which means we cannot overwrite the buffer without copying the ciphertext byte. We could move line 138 into the two cases of the if statement, but that enlarges the code for no savings in time.

```
139      }
140
141      return CRYPT_OK;
142   }
```

At this point, we have enough functionality to start a GCM state, add IV, add AAD, and process plaintext. Now we must be able to terminate the GCM state to compute the final MAC tag.

Terminating the GCM State

Once we have finished processing our GCM message, we will want to compute the GHASH output and retrieve the MAC tag.

gcm_done.c:

```
001   /**
002     Terminate a GCM stream
003     @param gcm     The GCM state
004     @param tag     [out] The destination for the MAC tag
005     @param taglen  [in/out]  The length of the MAC tag
006     @return CRYPT_OK on success
007    */
008   int gcm_done(    gcm_state *gcm,
009                 unsigned char *tag, unsigned long *taglen)
010   {
011      unsigned long x;
012      int err;
013
014      LTC_ARGCHK(gcm     != NULL);
015      LTC_ARGCHK(tag     != NULL);
016      LTC_ARGCHK(taglen  != NULL);
017
018      if (gcm->buflen > 16 || gcm->buflen < 0) {
019         return CRYPT_INVALID_ARG;
020      }
021
022      if ((err = cipher_is_valid(gcm->cipher)) != CRYPT_OK) {
023         return err;
024      }
025
026
027      if (gcm->mode != GCM_MODE_TEXT) {
028         return CRYPT_INVALID_ARG;
029      }
```

This is again the same sanity checking. It is important that we do this in all the GCM functions, since the context of the state is important in using GCM securely.

```
   /* handle remaining ciphertext */
   if (gcm->buflen) {
      gcm->pttotlen += gcm->buflen * CONST64(8);
      gcm_mult_h(gcm, gcm->X);
   }

   /* length */
   STORE64H(gcm->totlen, gcm->buf);
   STORE64H(gcm->pttotlen, gcm->buf+8);
   for (x = 0; x < 16; x++) {
       gcm->X[x] ^= gcm->buf[x];
   }
   gcm_mult_h(gcm, gcm->X);
```

This terminates the GHASH of the AAD and ciphertext. The length of the AAD and the length of the ciphertext are added to the final multiplication by H. The output is the GHASH final value, but not the MAC tag.

```
   /* encrypt original counter */
   if ((err =
        cipher_descriptor[gcm->cipher].ecb_encrypt(
            gcm->Y_0, gcm->buf, &gcm->K)) != CRYPT_OK) {
      return err;
   }
   for (x = 0; x < 16 && x < *taglen; x++) {
       tag[x] = gcm->buf[x] ^ gcm->X[x];
   }
   *taglen = x;
```

We encrypt the original *Y* value and XOR the output against the GHASH output. Since GCM allows truncating the MAC tag, we do so. The user may request any length of MAC tag from 0 to 16 bytes. It is not advisable to truncate the GCM tag.

```

   cipher_descriptor[gcm->cipher].done(&gcm->K);

   return CRYPT_OK;
}
```

The last call terminates the cipher state. With LibTomCrypt, the ciphers are allowed to allocate resources during their initialization such as heap or hardware tokens. This call is required to release the resources if any. Upon completion of this function, the user has the MAC tag and the GCM state is no longer valid.

GCM Optimizations

Now that we have seen the design of GCM, and a fielded implementation from the LibTomCrypt project, we have to specifically address how to best use GCM.

The default implementation in LibTomCrypt with the GCM_TABLES macro defined uses 64 kilobytes per GCM state. This may be no problem for a server or desktop application; however, it does not serve well for platforms that may not have 64 kilobytes of memory total. There exists a simple compromise that uses a single 8x128 table, which is four kilobytes in size. In fact, it uses the zero'th table from the 64-kilobyte variant, and we produce a 32-byte product and reduce it the same style as our fast variant of the gcm_gf_mult() function.

```
void gcm_mult_h(gcm_state *gcm, unsigned char *I)
{
   unsigned char T[32];
   int           i, j;

   /* produce 32 byte product */
   for (i = 0; i < 32; i++) T[i] = 0;
   for (i = 0; i < 16; i++)
      for (j = 0; j < 16; j++)
         T[i+j] ^= gcm->PC[I[i]][j];

   /* reduce it */
   for (i = 31; i >= 16; i--) {
     T[i-16] ^= gcm_shift_table[((unsigned)T[i]<<1)];
     T[i-15] ^= gcm_shift_table[((unsigned)T[i]<<1)+1];
   }

   /* copy out result */
   for (i = 0; i < 16; i++) I[i] = T[i];
}
```

We can optimize the nested loop by using word-oriented XOR operations. The final reduction is less amenable to optimization, unfortunately. The fallback position from this algorithm is to use the LTC_FAST variant of gcm_gf_mult(), the generic GF(2) multiplier, to perform the multiply by H function.

Use of SIMD Instructions

Processors such as the recent x86 series and Power PC based G4 and G5 have instructions known as Single Instruction Multiple Data (SIMD). These allow developers to perform multiple parallel operations such as 128-bit wide XORs. This comes in very handy, indeed. For example, consider the function gcm_mult_h() with SSE2 support.

gcm_mult_h.c: (From LibTomCrypt v1.14)

```
void gcm_mult_h(gcm_state *gcm, unsigned char *I)
{
   unsigned char T[16];
#ifdef GCM_TABLES
   int x, y;
#ifdef GCM_TABLES_SSE2
   asm("movdqa (%0),%%xmm0"::"r"(&gcm->PC[0][I[0]][0]));
   for (x = 1; x < 16; x++) {
      asm("pxor (%0),%%xmm0"::"r"(&gcm->PC[x][I[x]][0]));
   }
   asm("movdqa %%xmm0,(%0)"::"r"(&T));
<snip>
   XMEMCPY(I, T, 16);
}
#endif
```

As we can see, the loop becomes much more efficient. Particularly, we are issuing fewer load operations and consuming fewer decoder resources. For example, even though the Opteron FPU issues the 128-bit operations as two 64-bit operations behind the scenes, since we only decode one x86 opcode, the processor can pack the instructions more effectively. Table 7.2 compares the performance of three implementations of GCM on an Opteron and Intel P4 Prescott processors.

Table 7.2 GCM Implementation Performance Observations

Implementation	Cycles per message byte (4KB blocks) Opteron	Cycles per message byte (4KB blocks) P4 Prescott
LTC_FAST only	69	
LTC_FAST and GCM_TABLES	27	53
LTC_FAST and GCM_TABLES_SSE2	25	49

Design of CCM

CCM is the current NIST endorsed combined mode (SP 800-38C) based on CBC-MAC and CTR chaining mode. The purpose of CCM is to use a single secret key *K* and block cipher, to both authenticate and encrypt a message. Unlike GCM, CCM is much simpler and only requires the block cipher to operate. For this reason, it is often a better fit where GF(2) multiplication is too cumbersome.

While CCM is simpler than GCM, it does carry several disadvantages. In hardware, GF(2) multiplication is usually faster than an AES encryption (AES being the usual block cipher paired with GCM). CCM can process AAD (which they call header data), but is less flexible with the order of operations. Where in GCM you can process arbitrary length AAD and plaintext data, CCM must know in advance the length of these elements before initializing the state. This means you cannot use CCM for streaming calls. Fortunately, in practice this is usually not a problem. You simply apply CCM to your individual packets of a known or computable length and everything is fine.

The design of CCM can be split into three distinct phases. First, we must generate a B_0 block, which is built out of the user supplied nonce and message length. The B_0 block is used as the IV for the CBC-MAC and counter for the CTR chaining mode. Next, the MAC tag generation, which is the CBC-MAC of B_0, the header (AAD) data, and finally the plaintext, is performed. Unlike GCM, CCM authenticates the plaintext, which is why a unique nonce is important for security. Finally, the plaintext is encrypted with the CTR chaining mode using a modified B_0 block as the counter.

CCM B_0 Generation

The B_0 block is a special block in CCM mode that is prefixed to the message (before the header data) that is derived from the user supplied nonce value. The layout of the B_0 block depends on the length of the plaintext, especially the length of the encoding of the length. For example, if the plaintext is 129 bytes long, it will require one byte to encode the length of the length. We call this length q and the value of the length Q. For example, in our previous example, we would have q=1 and Q=129.

The nonce data can be up to 13 bytes in length and its contents are represented by N. The length of the desired MAC tag is represented by t, and must be even and larger than 2 (Figure 7.4).

Figure 7.4 CCM B_0 Format

Octet Number	0	1 ... 15–q	16–q ...	15
Contents	Flags	N	Q	

Note how the nonce (N) is truncated due to the length of the plaintext. Usually, this is not a problem in most network protocols where Q is typically smaller than 65536 (leaving us with up to a 13-byte nonce). The flags require a single byte and are packed as shown in Figure 7.5.

Figure 7.5 CCM Flags Format

Bit Number	7	6	5	4	3	2	1	0
Contents	Reserved	Adata	(t–2)/2			q–1		

The *Adata* flag is set when there is header (AAD) data present. We encode the MAC tag length by subtracting 2 and dividing it by 2, and t must be an element of the set {4, 6, 8, 10, 12, 14, 16}. MAC lengths that short are usually not a good idea. If you really want to shave a few bytes per packet, try using a MAC tag length no shorter than 12 bytes. The plaintext length is encoded by subtracting one from the length of the plaintext. q must be a member of the set {2, 3, 4, 5, 6, 7, 8}. Zero is not a valid value for either (t–2)/2 or q–1.

CCM MAC Tag Generation

The CCM MAC Tag is generated by the CBC-MAC of B_0, any header data (if present), and the plaintext (if any). We prefix the header data with its length. If the header data is less than 65,280 bytes in length, the length is simply encoded as a big endian two-byte value. If the length is larger, the value 0xFF FE is inserted followed by the big endian four-byte encoding of the header length. Technically, CCM supports larger header lengths; however, if you are using more than four gigabytes for your header data, you ought to rethink your application's used of cryptography.

The header data is padded to a multiple of 16 bytes by inserting zero bytes at the end. The plaintext is similarly padded as required.

After the CBC-MAC output has been generated, it will be pre-pended to the plaintext and encrypted in CTR mode along with the plaintext.

CCM Encryption

Encryption is performed in CTR mode using the B_0 block as the counter. The only difference here is that the B_0 block will be modified by zeroing the *flags* and *Q* parameters (leaving only the nonce). The last q bytes of the B0 block are incremented in big endian fashion for each 16-byte block of Tag and plaintext. The ciphertext is the same length as the plaintext and not padded.

CCM Implementation

We use the CCM implementation from LibTomCrypt as reference. It is a compact and efficient implementation of CCM that performs the entire CCM encoding with a single function call. Since it accepts all the parameters in a single call, it has quite a few parameters. Table 7.3 matches the implementation names with the design variable names.

Table 7.3 CCM Implementation Guide

Design Name	Implementation Name	Function
	cipher	Index into LTC tables for 128-bit cipher to use, makes CCM agnostic to the cipher choice
K	key	Secret key
	keylen	Length of key in octets
	uskey	Previously scheduled key, used to save time by not using key schedule
N	none	The CCM nonce
	noncelen	Length of nonce in octets
	header	The header or AAD data
	headerlen	Length of header in octets
P	pt	Plaintext
Q	ptlen	Length of plaintext in octets
C	ct	Ciphertext
T	tag	The MAC tag
t	taglen	The length of the MAC tag desired

ccm_memory.c:

```
/**
   CCM encrypt/decrypt and produce an authentication tag
   @param cipher     The index of the cipher desired
   @param key        The secret key to use
   @param keylen     The length of the secret key (octets)
   @param uskey      A previously scheduled key [can be NULL]
   @param nonce      The session nonce [use once]
   @param noncelen   The length of the nonce
   @param header     The header for the session
   @param headerlen  The length of the header (octets)
   @param pt         [out] The plaintext
   @param ptlen      The length of the plaintext (octets)
   @param ct         [out] The ciphertext
   @param tag        [out] The destination tag
   @param taglen     [in/out] The max size of the
                     authentication tag
   @param direction  Encrypt or Decrypt direction (0 or 1)
   @return CRYPT_OK if successful
*/
int ccm_memory(int cipher,
    const unsigned char *key,    unsigned long keylen,
    symmetric_key       *uskey,
    const unsigned char *nonce,  unsigned long noncelen,
    const unsigned char *header, unsigned long headerlen,
          unsigned char *pt,     unsigned long ptlen,
          unsigned char *ct,
          unsigned char *tag,    unsigned long *taglen,
                    int  direction)
{
   unsigned char  PAD[16], ctr[16], CTRPAD[16], b;
   symmetric_key *skey;
   int            err;
   unsigned long  len, L, x, y, z, CTRlen;

   if (uskey == NULL) {
      LTC_ARGCHK(key    != NULL);
   }
```

We allow the caller to schedule a key prior to the call. This is a good idea in practice, as it shaves a fair chunk of cycles per call. Even if you are not using LibTomCrypt, you should provide or use this style of optimization in your code.

```
   LTC_ARGCHK(nonce  != NULL);
   if (headerlen > 0) {
      LTC_ARGCHK(header != NULL);
   }
```

We allow empty headers so in those cases header can have the value of NULL.

```
   LTC_ARGCHK(pt     != NULL);
   LTC_ARGCHK(ct     != NULL);
   LTC_ARGCHK(tag    != NULL);
   LTC_ARGCHK(taglen != NULL);

```

```
#ifdef LTC_FAST
   if (16 % sizeof(LTC_FAST_TYPE)) {
      return CRYPT_INVALID_ARG;
   }
#endif
```

We provide a word-oriented optimization later in this implementation. This check is a simple sanity check to make sure it will actually work.

```
   /* check cipher input */
   if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
      return err;
   }
   if (cipher_descriptor[cipher].block_length != 16) {
      return CRYPT_INVALID_CIPHER;
   }

   /* make sure the taglen is even and <= 16 */
   *taglen &= ~1;
   if (*taglen > 16) {
      *taglen = 16;
   }
```

We force the tag length to even and truncate it if it is larger than 16 bytes. We do not error out on excessively long tag lengths, as the caller may simply have passed something such as

```
unsigned char tag[MAXTAGLEN];
unsigned long taglen;

taglen = sizeof(tag);
ccm_memory(..., &taglen, ...);
```

In which case, MAXTAGLEN could be larger than 16 if they are supporting more than one algorithm.

```
   /* can't use < 4 */
   if (*taglen < 4) {
      return CRYPT_INVALID_ARG;
   }
```

We must reject tag lengths less than four bytes per the CCM specification. We have no choice but to error out, as the caller has not indicated we can store at least four bytes of MAC tag.

```
   /* is there an accelerator? */
   if (cipher_descriptor[cipher].accel_ccm_memory != NULL) {
       return cipher_descriptor[cipher].accel_ccm_memory(
           key,    keylen,
           uskey,
           nonce,  noncelen,
           header, headerlen,
           pt,     ptlen,
           ct,
           tag,    taglen,
           direction);
   }
```

LibTomCrypt has the capability to "overload" functions—in this case, CCM. If the pointer is not NULL, the computation is offloaded to it automatically. In this way, a developer can take advantage of accelerators without re-writing their application. This technically is not part of CCM, so you can avoid looking at this chunk if you want.

```
/* let's get the L value */
len = ptlen;
L   = 0;
while (len) {
   ++L;
   len >>= 8;
}
if (L <= 1) {
   L = 2;
}
```

Here we compute the value of *L* (*q* in the CCM design). *L* was the original name for this variable and is why we used it here. We make sure that *L* is at least 2 as per the CCM specification.

```
/* increase L to match the nonce len */
noncelen = (noncelen > 13) ? 13 : noncelen;
if ((15 - noncelen) > L) {
   L = 15 - noncelen;
}
if (15 < (noncelen + L)) noncelen = 15 - L;
```

This resizes the nonce if it is too large and the *L* parameter as required. The caller has to be aware of the tradeoff. For instance, if you want to encrypt one-megabyte packets, you will need at least three bytes to encode the length, which means the nonce can only be 12 bytes long. One could add a check to ensure that *L* is never too small for the plaintext length.

```
/* allocate mem for the symmetric key */
if (uskey == NULL) {
   skey = XMALLOC(sizeof(*skey));
   if (skey == NULL) {
      return CRYPT_MEM;
   }

   /* initialize the cipher */
   if ((err =
        cipher_descriptor[cipher].setup(
           key, keylen, 0, skey)) != CRYPT_OK) {
      XFREE(skey);
      return err;
   }
} else {
   skey = uskey;
}
```

If the caller does not supply a key, we must schedule one. We avoid placing the scheduled key structure on the stack by allocating it from the heap. This is important for embedded and kernel applications, as the stacks can be very limited in size.

```
120      /* form B_0 == flags | Nonce N | l(m) */
121      x = 0;
122      PAD[x++] = ((headerlen > 0) ? (1<<6) : 0) |
123               (((*taglen - 2)>>1)<<3)        |
124               (L-1);
125
126      /* nonce */
127      for (y = 0; y < (16 - (L + 1)); y++) {
128          PAD[x++] = nonce[y];
129      }
130
131      /* store len */
132      len = ptlen;
133
134      /* shift len so the upper bytes of len are
135       * the contents of the length */
136      for (y = L; y < 4; y++) {
137          len <<= 8;
138      }
139
140      /* store l(m) (only store 32-bits) */
141      for (y = 0; L > 4 && (L-y)>4; y++) {
142          PAD[x++] = 0;
143      }
144      for (; y < L; y++) {
145          PAD[x++] = (len >> 24) & 255;
146          len <<= 8;
147      }
```

This section of code creates the B_0 value we need for the CBC-MAC phase of CCM. The PAD array holds the 16 bytes of CBC data for the MAC, while CTRPAD, which we see later, holds the 16 bytes of CTR output.

The first byte (line 122) of the block is the flags. We set the *Adata* flag based on *headerlen*, encode the tag length by dividing *taglen* by two, and finally the length of the plaintext length is stored.

Next, the nonce is copied to the block. We use $16 - L + 1$ bytes of the nonce since we must store the flags and *L* bytes of the plaintext length value.

To make things a bit more practical, we only store 32 bits of the plaintext length. If the user specifies a short nonce, the value of *L* has to be increased to compensate. In this case, we pad with zero bytes before encoding the actual length.

```
149      /* encrypt PAD */
150      if ((err =
151           cipher_descriptor[cipher].ecb_encrypt(
152              PAD, PAD, skey)) != CRYPT_OK) {
153          goto error;
154      }
```

We are using CBC-MAC effectively with a zeroed IV, so the first thing we must do is encrypt PAD. The ciphertext is now the IV for the CBC-MAC of the rest of the header and plaintext data.

```
   /* handle header */
   if (headerlen > 0) {
      x = 0;
```

We only get here and do any of the following code if there is header data to process.

```
      /* store length */
      if (headerlen < ((1UL<<16) - (1UL<<8))) {
         PAD[x++] ^= (headerlen>>8) & 255;
         PAD[x++] ^= headerlen & 255;
      } else {
         PAD[x++] ^= 0xFF;
         PAD[x++] ^= 0xFE;
         PAD[x++] ^= (headerlen>>24) & 255;
         PAD[x++] ^= (headerlen>>16) & 255;
         PAD[x++] ^= (headerlen>>8) & 255;
         PAD[x++] ^= headerlen & 255;
      }
```

The encoding of the length of the header data depends on the size of the header data. If it is less than 65,280 bytes, we use the short two-byte encoding. Otherwise, we emit the escape sequence 0xFF FE and then the four-byte encoding. CCM supports larger header sizes, but you are unlikely to ever need to support it.

Note that instead of XORing the PAD (as an IV) against another buffer, we simply XOR the lengths into the PAD. This avoids any double buffering we would otherwise have to use.

```
      /* now add the data */
      for (y = 0; y < headerlen; y++) {
          if (x == 16) {
             /* full block so let's encrypt it */
             if ((err =
                  cipher_descriptor[cipher].ecb_encrypt(
                     PAD, PAD, skey)) != CRYPT_OK) {
                goto error;
             }
             x = 0;
          }
          PAD[x++] ^= header[y];
      }
```

This loop processes the entire header data. We do not provide any sort of LTC_FAST optimizations, since headers are usually empty or very short. Every 16 bytes of header data, we encrypt the PAD to emulate CBC-MAC properly.

```
      /* remainder? */
      if (x != 0) {
         if ((err =
              cipher_descriptor[cipher].ecb_encrypt(
                 PAD, PAD, skey)) != CRYPT_OK) {
            goto error;
         }
      }
   }
```

If we have leftover header data (that is, headerlen is not a multiple of 16), we pad it with zero bytes and encrypt it. Since XORing zero bytes is a no-operation, we simply ignore that step and invoke the cipher.

```
   /* setup the ctr counter */
   x = 0;

   /* flags */
   ctr[x++] = L-1;

   /* nonce */
   for (y = 0; y < (16 - (L+1)); ++y) {
      ctr[x++] = nonce[y];
   }
   /* offset */
   while (x < 16) {
      ctr[x++] = 0;
   }
```

This code creates the initial counter for the CTR encryption mode. The flags only contain the length of the plaintext length. The nonce is copied much as it is for the CBC-MAC, and the rest of the block is zeroed. The bytes after the nonce are incremented during the encryption.

```
   x      = 0;
   CTRlen = 16;

   /* now handle the PT */
   if (ptlen > 0) {
      y = 0;
#ifdef LTC_FAST
      if (ptlen & ~15)  {
          if (direction == CCM_ENCRYPT) {
             for (; y < (ptlen & ~15); y += 16) {
```

If we are encrypting, we handle all complete 16-byte blocks of plaintext we have.

```
                /* increment the ctr? */
                for (z = 15; z > 15-L; z--) {
                    ctr[z] = (ctr[z] + 1) & 255;
                    if (ctr[z]) break;
                }
```

The CTR counter is big endian and stored at the end of the ctr array. This code increments it by one.

```
                if ((err =
                     cipher_descriptor[cipher].ecb_encrypt(
                        ctr, CTRPAD, skey)) != CRYPT_OK) {
                   goto error;
                }
```

We must encrypt the CTR counter before using it to encrypt plaintext.

```
               /* xor the PT against the pad first */
               for (z = 0; z < 16; z += sizeof(LTC_FAST_TYPE)) {
                   *((LTC_FAST_TYPE*)(&PAD[z]))  ^=
                      *((LTC_FAST_TYPE*)(&pt[y+z]));
                   *((LTC_FAST_TYPE*)(&ct[y+z])) =
                      *((LTC_FAST_TYPE*)(&pt[y+z])) ^
                      *((LTC_FAST_TYPE*)(&CTRPAD[z]));
               }
```

This loop XORs 16 bytes of plaintext against the CBC-MAC pad, and then creates 16 bytes of ciphertext by XORing CTRPAD against the plaintext. We do the encryption second (after the CBC-MAC), since we allow the plaintext and ciphertext to point to the same buffer.

```
               if ((err =
                    cipher_descriptor[cipher].ecb_encrypt(
                       PAD, PAD, skey)) != CRYPT_OK) {
                  goto error;
               }
            }
```

Encrypting the CBC-MAC pad performs the required MAC operation for this 16-byte block of plaintext.

```
        } else {
            for (; y < (ptlen & ~15); y += 16) {
               /* increment the ctr? */
               for (z = 15; z > 15-L; z--) {
                   ctr[z] = (ctr[z] + 1) & 255;
                   if (ctr[z]) break;
               }
               if ((err =
                    cipher_descriptor[cipher].ecb_encrypt(
                       ctr, CTRPAD, skey)) != CRYPT_OK) {
                  goto error;
               }

               /* xor the PT against the pad last */
               for (z = 0; z < 16; z += sizeof(LTC_FAST_TYPE)) {
                   *((LTC_FAST_TYPE*)(&pt[y+z])) =
                      *((LTC_FAST_TYPE*)(&ct[y+z])) ^
                      *((LTC_FAST_TYPE*)(&CTRPAD[z]));
                   *((LTC_FAST_TYPE*)(&PAD[z]))  ^=
                      *((LTC_FAST_TYPE*)(&pt[y+z]));
               }
               if ((err =
                    cipher_descriptor[cipher].ecb_encrypt(
                       PAD, PAD, skey)) != CRYPT_OK) {
                  goto error;
```

```
                }
             }
         }
     }
```

We handle decryption similarly, but distinctly, since we allow the plaintext and ciphertext to point to the same memory. Since this code is likely to be unrolled, we avoid having redundant conditional code inside the main loop where possible.

```
#endif

      for (; y < ptlen; y++) {
          /* increment the ctr? */
          if (CTRlen == 16) {
             for (z = 15; z > 15-L; z--) {
                 ctr[z] = (ctr[z] + 1) & 255;
                 if (ctr[z]) break;
             }
             if ((err =
                  cipher_descriptor[cipher].ecb_encrypt(
                     ctr, CTRPAD, skey)) != CRYPT_OK) {
                goto error;
             }
             CTRlen = 0;
          }

          /* if we encrypt we add the bytes to the MAC first */
          if (direction == CCM_ENCRYPT) {
             b     = pt[y];
             ct[y] = b ^ CTRPAD[CTRlen++];
          } else {
             b     = ct[y] ^ CTRPAD[CTRlen++];
             pt[y] = b;
          }

          if (x == 16) {
             if ((err =
                  cipher_descriptor[cipher].ecb_encrypt(
                     PAD, PAD, skey)) != CRYPT_OK) {
                goto error;
             }
             x = 0;
          }
          PAD[x++] ^= b;
      }
```

This block performs the CCM operation for any bytes of plaintext not handled by the LTC_FAST code. This could be because the plaintext is not a multiple of 16 bytes, or that LTC_FAST was not enabled. Ideally, we want to avoid needing this code as it is slow and over many packets can consume a fair amount of processing power.

```
      if (x != 0) {
         if ((err =
              cipher_descriptor[cipher].ecb_encrypt(
                 PAD, PAD, skey)) != CRYPT_OK) {
            goto error;
         }
      }
   }
```

We finish the CBC-MAC if there are bytes left over. As in the processing of the header, we implicitly pad with zeros by encrypting the PAD as is. At this point, the PAD now contains the CBC-MAC value but not the CCM tag as we still have to encrypt it.

```
/* setup CTR for the TAG */
for (z=15;z>15-L;z++) ctr[z] = 0x00;
if ((err =
     cipher_descriptor[cipher].ecb_encrypt(
        ctr, CTRPAD, skey)) != CRYPT_OK) {
   goto error;
}
```

The CTR pad for the CBC-MAC tag is computed by zeroing the last L bytes of the CTR counter and encrypting it to CTRPAD.

```
if (skey != uskey) {
   cipher_descriptor[cipher].done(skey);
}
```

If we scheduled our own key, we will now free any allocated resources.

```
/* store the TAG */
for (x = 0; x < 16 && x < *taglen; x++) {
    tag[x] = PAD[x] ^ CTRPAD[x];
}
*taglen = x;
```

CCM allows a variable length tag, from 4 to 16 bytes in length in 2-byte increments. We encrypt and store the CCM tag by XORing the CBC-MAC tag with the last encrypted CTR counter.

```
#ifdef LTC_CLEAN_STACK
   zeromem(skey,   sizeof(*skey));
   zeromem(PAD,    sizeof(PAD));
   zeromem(CTRPAD, sizeof(CTRPAD));
#endif
```

This block zeroes memory on the stack that could be considered sensitive. We hope the stack has not been swapped to disk, but this routine does not make this guarantee. By clearing the memory, any further potential stack leaks will not be sharing the keys or CBC-MAC intermediate values with the attacker. We only perform this operation if the user requested it by defining the LTC_CLEAN_STACK macro.

TIP

In most modern operating systems, the memory used by a program (or process) is known as *virtual memory*. The memory has no fixed physical address and can be moved between locations and even swapped to disk (through page invalidation). This latter action is typically known as *swap memory*, as it allows users to emulate having more physical memory than they really do.

The downside to swap memory, however, is that the process memory could contain sensitive information such as private keys, usernames, passwords, and other credentials. To prevent this, an application can *lock* memory. In operating systems such as those based on the NT kernel (e.g., Win2K, WinXP), locking is entirely voluntary and the OS can choose to later swap nonkernel data out.

In POSIX compatible operating systems, such as those based on the Linux and the BSD kernels, a set of functions such as mlock(), munlock(), mlockall(), and so forth have been provided to facilitate locking. Physical memory in most systems can be costly, so the polite and proper application will request to lock as little memory as possible. In most cases, locked memory will span a region that contains *pages* of memory. On the x86 series of processors, a page is four kilobytes. This means that all locked memory will actually lock a multiple of four kilobytes.

Ideally, an application will pool its related credentials to reduce the number of physical pages required to lock them in memory.

```
error:
   if (skey != uskey) {
      XFREE(skey);
   }

   return err;
}
```

Upon successful completion of this function, the user now has the ciphertext (or plaintext depending on the direction) and the CCM tag. While the function may be a tad long, it is nicely bundled up in a single function call, making its deployment rather trivial.

Putting It All Together

This chapter introduced the two standard encrypt and authenticate modes as specified by both NIST and IEEE. They are both designed to take a single key and IV (nonce) and produce a ciphertext and message authentication code tag, thereby simplifying the process for developers by reducing the number of different standards they must support, and in practice the number of functions they have to call to accomplish the same results.

Knowing how to use these modes is a matter of properly choosing an IV, making ideal use of the additional authentication data (AAD), and checking the MAC tag they produce. Neither of these two modes will manage any of these properties for the developer, so they must look after them carefully.

For most applications, it is highly advisable to use these modes over an ad hoc combination of encryption and authentication, if not solely for the reason of code simplicity, then also for proper adherence to cryptographic standards.

What Are These Modes For?

We saw in the previous chapter how we could accomplish both privacy and authentication of data through the combined use of a symmetric cipher and chaining mode with a MAC algorithm. Here, the goal of these modes is to combine the two. This accomplishes several key goals simultaneously. As we alluded to in the previous chapter, CCM and GCM are also meant for small packet messages, ideal for securing a stream of messages between parties. CCM and GCM can be used for offline tasks such as file encryption, but they are not meant for such tasks (especially CCM since it needs the length of the plaintext in advance).

First, combining the modes makes development simpler—there is only one key and one IV to keep track of. The mode will handle using both for both tasks. This makes key derivation easier and quicker, as less session data must be derived. It also means there are fewer variables floating around to keep track of.

These combined modes also mean it's possible to perform both goals with a single function call. In code where we specifically must trap error codes (usually by looking at the return codes), having fewer functions to call means the code is easier to write *safely*. While there are other ways to trap errors such as signals and internal masking, making threadsafe global error detection in C is rather difficult.

In addition to making the code easier to read and write, combined modes make the security easier to analyze. CCM, for instance, is a combination of CBC-MAC and CTR encryption mode. In various ways, we can reduce the security of CCM to the security of these modes. In general, with a full length MAC tag, the security of CCM reduces to the security of the block cipher (assuming a unique nonce and random key are used).

What we mean by *reduce* is that we can make an argument for equivalence. For example, if the security of CTR is reducible to the security of the cipher, we are saying it is as secure as the latter. By this reasoning, if one could break the cipher, he could also break CTR mode. (Strictly speaking, the security of CTR reduces to the determination of whether the cipher is a PRP.)

So, in this context, if we say CCM reduces to the security of the cipher in terms of being a proper pseudo-random permutation (PRP), then if we can break the cipher (by showing it is not a PRP), we can likely break CCM. Similarly, GCM reduces to the security of the cipher for privacy and to universal hashing for the MAC. It is more complicated to prove that it *can* be secure.

Choosing a Nonce

Both CCM and GCM require a unique nonce (N used once) value to maintain their privacy and authenticity goals. In both cases, the value need not be random, but merely unique for a given key. That is, you can safely use the same nonce (only once, though) between two different keys. Once you use the nonce for a particular key, you cannot use it again.

GCM Nonces

GCM was designed to be most efficient with 12-byte nonce values. Any longer or shorter and GHASH is used to create an IV for the mode. In this case, we can simply use the 12-byte nonce as a packet counter. Since we have to send the nonce to the other party anyway, this means we can double up on the purpose of this field. Each packet would get its own 12-byte nonce (by incrementing it), and the receiver can check for replays and out of order packets by checking the nonce as if it were a 96-bit number.

You can use the 12-byte number as either a big or little endian value, as GCM will not truncate the nonce.

CCM Nonces

CCM was not designed as favorably to the nonces as GCM was. Still, if you know all your packets will be shorter than 65,536 bytes, you can safely assume your nonce is allowed to be up to 13 bytes. Like GCM, you can use it as a 104-bit counter and increment it for every packet you send out.

If you cannot determine the length of your packets ahead of time, it is best to default to a shorter nonce (say 11 bytes, allowing up to four-gigabyte packets) as your counter. Remember, there is no magic property to the length of the nonce other than you have to have a long enough nonce to have unique values for every packet you send out under the same key.

CCM will truncate the nonce if the packet is too long (to have room to store the length), so in practice it is best to treat it as a little endian counter. The most significant bytes would be truncated. It is even better to just use a shorter nonce than worry about it.

Additional Authentication Data

Both CCM and GCM support a sort of side channel known as additional authentication data (AAD). This data is meant to be nonprivate data that should influence the MAC tag output. That is, if the plaintext and AAD are not present together and unmodified, the tag should reflect that.

The usual use for AAD is to store session metadata along with the packet. Things such as username, session ID, and transaction ID are common. You would never use a user credential, since it would not really be something you need on a per-packet basis.

Both protocols support empty AAD strings. Only GCM is optimized to handle AAD strings that are a multiple of 16 bytes long. CCM inserts a four- or six-byte header that offsets the data and makes it harder to optimize for. In general, if you are using CCM, try to have very short AAD strings, preferably less than 10 bytes, as you can cram that into a single encrypt invocation. For GCM, try to have your AAD strings a multiple of 16 bytes even if you have to pad with zero bytes (the implementation cannot do this padding for you, as it would change the meaning of the AAD).

MAC Tag Data

The MAC tag produced by both implementations is not checked internally. The typical usage would involve transmitting the MAC tag with the ciphertext, and the recipient would compare it against the one he generated while decrypting the message.

In theory at least, you can truncate the MAC tag to short lengths such as 80 or 96 bits. However, some evidence points to the contrary with GCM, and in reality the savings are trivial. As the research evolves, it would be best to read up on the latest analysis papers of GCM and CCM to see if short tags are still in fact secure.

In practice, you can save more space if you aggregate packets over a stable channel. For example, instead of sending 128-byte packets, send 196- or 256-byte packets. You will send fewer nonces (and protocol data), and the difference can allow you to use a longer MAC tag. Clearly, this does not work in low latency switching cases (e.g., VoIP), so it is not a bullet-proof suggestion.

Example Construction

For our example, we will borrow from the example of Chapter 6, except instead of using HMAC and CTR mode, we will use CCM. The rest of the demo works the same. Again, we will be using LibTomCrypt, as it provides a nice CCM interface that is very easy to use in fielded applications.

encmac.c:

```
#include <tomcrypt.h>

#define KEYLEN    16
```

We only have one key and its 16 bytes long.

```
/* Our Per Packet Sizes, Nonce len and MAC len */
#define NONCELEN     4
#define MACLEN       12
#define OVERHEAD     (NONCELEN+MACLEN)
```

As in the previous example, we have a packet counter length (the length of our nonce), MAC tag length, and the composite overhead. Here we have a 32-bit counter and a 96-bit MAC.

```
/* errors */
#define MAC_FAILED    -3
#define PKTCTR_FAILED -4

```

```
/* our nice containers */
typedef struct {
   unsigned char PktCTR[NONCELEN],
                 key[KEYLEN];
   symmetric_key skey;
} encauth_channel;
```

Our encrypted and authenticated channel is similar to the previous example. The differences here are that we are using a generic scheduled key, and we do not have a MAC key.

```
typedef struct {
   encauth_channel channels[2];
} encauth_stream;


void register_algorithms(void)
{
   register_cipher(&aes_desc);
   register_hash(&sha256_desc);
}

int init_stream(const unsigned char *masterkey,
                           unsigned  masterkeylen,
                const unsigned char *salt,
                           unsigned  saltlen,
                     encauth_stream *stream,
                                int  node)
{
   unsigned char tmp[2*KEYLEN];
   unsigned long tmplen;
   int           err;
   encauth_channel tmpswap;

   /* derive keys */
   tmplen = sizeof(tmp);
   if ((err = pkcs_5_alg2(masterkey, masterkeylen,
                          salt, saltlen,
                          16, find_hash("sha256"),
                          tmp, &tmplen)) != CRYPT_OK) {
      return err;
   }

   /* copy keys */
   memcpy(stream->channels[0].key,
          tmp, KEYLEN);
   memcpy(stream->channels[1].key,
          tmp + KEYLEN, KEYLEN);

   /* reset counters */
   memset(stream->channels[0].PktCTR, 0,
          sizeof(stream->channels[0].PktCTR));
   memset(stream->channels[1].PktCTR, 0,
          sizeof(stream->channels[1].PktCTR));

   /* schedule keys+setup mode */
   if ((err = rijndael_setup(stream->channels[0].key,
                             KEYLEN, 0,
                             &stream->channels[0].skey))
      != CRYPT_OK) {
      return err;
   }

   if ((err = rijndael_setup(stream->channels[1].key,
                             KEYLEN, 0,
                             &stream->channels[1].skey))
      != CRYPT_OK) {
      return err;
   }

   /* do we swap? */
   if (node != 0) {
       tmpswap             = stream->channels[0];
       stream->channels[0] = stream->channels[1];
       stream->channels[1] = tmpswap;
       zeromem(&tmpswap, sizeof(tmpswap));
   }

   zeromem(tmp, sizeof(tmp));

   return 0;
}
```

This initialization function is essentially the same. We only change how much PKCS #5 data is to be generated.

```
int encode_frame(const unsigned char *in,
                            unsigned  inlen,
                       unsigned char *out,
                      encauth_stream *stream)
{
   int           x, err;
   unsigned long taglen;

   /* increment counter */
   for (x = NONCELEN-1; x >= 0; x--) {
       if (++(stream->channels[0].PktCTR[x]) & 255) break;
   }

   /* store counter */
   for (x = 0; x < NONCELEN; x++) {
     out[x] = stream->channels[0].PktCTR[x];
   }
```

We use our packet counter PktCTR as the a method of keeping the packets in order. We also use it as a nonce for the CCM algorithm. The output generated will first consist of the nonce, followed by the ciphertext, and then the MAC tag.

```
   /* encrypt it */
   taglen = MACLEN;
   if ((err = ccm_memory(
```

This function call will generate the ciphertext and MAC tag for us. Libraries are nice.

```
                 find_cipher("aes"),
```

First, we need the index of the cipher we want to use. In this case, AES with CCM. We could otherwise use any other 128-bit block cipher such as Twofish, Anubis, or Serpent.

```
                 stream->channels[0].key,    KEYLEN,
```

This is the byte array of the key we want to use and the key length.

```
                 &stream->channels[0].skey,
```

This is the pre-scheduled key. Our CCM routine will use this instead of scheduling the key on the fly. We pass the byte array for completeness (accelerators may need it).

```
                 stream->channels[0].PktCTR, NONCELEN,
```

This is the nonce for the packet.

```
                 NULL, 0,
```

This is the AAD; in this case, we are sending none. We pass NULL and 0 to signal this to the library.

```
                 (unsigned char*)in, inlen, out + NONCELEN,
                 out+inlen+NONCELEN, &taglen, CCM_ENCRYPT))
```

These are the plaintext and ciphertext buffers. The plaintext is always specified first regardless of the direction we are using. It may get a bit confusing for those expecting an "input" and "output" order of arguments.

```
       != CRYPT_OK) {
      return err;
   }
   return CRYPT_OK;
}
```

The ccm_memory() function call produces the ciphertext and MAC tag. The format of our packet contains the nonce, followed by the ciphertext, and then the MAC tag. From a security standpoint, this function provides for the same security goals as the previous chapter's incarnation of this function. The improvement in this case is that we have shortened the code and removed a few variables from our structure.

```
int decode_frame(const unsigned char *in,
                            unsigned  inlen,
                       unsigned char *out,
                      encauth_stream *stream)
{
   int           err;
   unsigned char tag[MACLEN];
   unsigned long taglen;

   /* restore our original inlen */
```

```
      if (inlen < MACLEN+NONCELEN) { return -1; }
      inlen -= MACLEN+NONCELEN;
```

As before, we assume that *inlen* is the length of the *entire* packet and not just the plaintext. We first make sure it is a valid length, and then subtract the MAC tag and nonce from the length.

```
      /* decrypt it */
      taglen = sizeof(tag);
      if ((err = ccm_memory(
                    find_cipher("aes"),
                    stream->channels[1].key, KEYLEN,
                    &stream->channels[1].skey,
                    (unsigned char*)in, NONCELEN,
                    NULL, 0,
                    out, inlen, (unsigned char*)in + NONCELEN,
                    tag, &taglen, CCM_DECRYPT))
          != CRYPT_OK) {
         return CRYPT_OK;
      }
```

For both encryption and decryption, the ccm_memory() function is used. In decrypt mode, it is used much the same, except our *out* buffer goes in the plaintext argument position. Therefore, it appears before the input, which seems a bit awkward.

Note that we store the MAC tag locally since we need to compare it next.

```
      /* compare */
      if (memcmp(tag, in+inlen+NONCELEN, MACLEN)) {
         return MAC_FAILED;
      }

      /* compare CTR */
      if (memcmp(in, stream->channels[1].PktCTR, NONCELEN) <= 0) {
         return PKTCTR_FAILED;
      }
```

These checks are the same as they were in the previous incarnation. The first avoids alterations, and the second avoids replay attacks. We still do not handle out of order packets (e.g., with a window), but that should be trivial to accommodate.

```
      /* copy the pktctr */
      memcpy(stream->channels[1].PktCTR, in, NONCELEN);
```

At this point, our packet is valid, so we copy the nonce out and store it locally, thus preventing future replay attacks.

```
      return CRYPT_OK;
   }

<snip>
```

The rest of our demonstration program is the same as our previous incarnation, as we have not changed the function interfaces.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: What is an Encrypt and Authenticate algorithm?

A: These are algorithms designed to accept a single key and IV, and allow the caller to both encrypt and authenticate his message. Unlike distinct encrypt and authenticate protocols, these modes do not require two independent keys for security.

Q: What are the benefits of these modes over the distinct modes?

A: There are two essential benefits to these modes. First, they are easier to incorporate into cryptosystems, as they require fewer keys, and perform two functions at the same time. This means a developer can accomplish two goals with a single function call. It also means they are more likely to start authenticating traffic. The second benefit is that they are often reducible to the security of the underlying cipher or primitive.

Q: What does reducible mean?

A: When we say one problem is reducible to another, we imply that solving the former involves a solution to the latter. For instance, the security of CTR-AES reduces to the problem of whether AES is a true pseudo random permutation (PRP). If AES is not a PRP, CTR-AES is not secure.

Q: What Encrypt and Authenticate modes are standardized?

A: CCM is specified in the NIST publication SP 800-38C. GCM is an IEEE standard and will be specified in the NIST publication SP 800-38D in the future.

Q: Should I use these modes over distinct modes such as AES-CTR and SHA1-HMAC?

A: That depends on whether you are trying to adhere to an older standard. If you are not, most likely GCM or CCM make better use of system resources to accomplish equivalent goals. The distinct modes can be just as secure in practice, but are generally harder to implement and verify in fielded applications. They also rely on more security assumptions, which poses a security risk.

Q: What is a nonce and what is the importance of it?

A: A nonce is a parameter (like an initial value) that is meant to be used only once. Nonces are used to introduce entropy into an otherwise static and fully deterministic process (that is, after the key has been chosen). Nonces must be unique under one key, but may be reused across different keys. Usually, they are simply packet counters. Without a unique nonce, both GCM and CCM have no provable security properties.

Q: What is additional authentication data? AAD? Header data?

A: Additional authentication Data (AAD), also known as header data and associated data (in EAX), is data related to a message, that must be part of the MAC tag computation but not encrypted. Typically, it is simple metadata related to the session that is not private but affects the interpretation of the messages. For instance, an IP datagram encoder could use parts of the IP datagram as part of the header data. That way, if the routing is changed it is detectable, and since the IP header cannot be encrypted (without using another IP standard), it must remain in plaintext form. That is only one example of AAD; there are others. In practice, it is not used much but provided nonetheless.

Q: Which mode should I use? GCM or CCM?

A: That depends on the standard you are trying to comply with (if any). CCM is currently the only NIST standard of the two. If you are working with wireless standards, chances are you will need GCM. Outside of those two cases, it depends on the platform. GCM is very fast, but requires a huge 64-kilobyte table to achieve this speed (at least in software). CCM is a slight bit slower, but is much simpler in design and implementation. CCM also does not require huge tables to achieve its performance claims. If you removed the tables from GCM, it would be much slower than CCM.

Q: Are there any patents on these modes?

A: Neither of GCM and CCM is patented. They are free for any use.

Q: Where can I find implementations?

A: Currently, only LibTomCrypt provides both GCM and CCM in a library form. Brian Gladman has independent copyrighted implementations available. Crypto++ has neither, but the author is aware of this.

Q: What do I do with the MAC tag?

A: If you are encrypting, transmit it along with your ciphertext and AAD data (if any). If you are decrypting, the decryption will generate a MAC tag; compare that with the value stored along with the ciphertext. If they do not compare as equal, the message has been altered or the stored MAC tag is invalid.

Chapter 8

Large Integer Arithmetic

Solutions in this chapter:

- **What Are BigNums?**
- **Why Do We Need Them for Cryptography?**
- **What Algorithms Are Most Important?**
- **Implementation Details**
- **Where to Get More Resources?**

☑ **Summary**

☑ **Solutions Fast Track**

☑ **Frequently Asked Questions**

Introduction

So far, we have been examining symmetric key algorithms that rely solely on secret keys for security. Now we are going to explore the realm of public key cryptography, but before we can do this, we have a significant piece of mathematics to cover.

Most standard public key algorithms are based on problems that are hard to solve in general. For example, the RSA algorithm is (loosely speaking) as secure as factoring is hard. That is, if factoring is hard, breaking RSA is, too (in practice). Similarly, elliptic curve algorithms are as hard to break as inverting point multiplication on the given curve.

In both cases, the "problem" becomes harder as you increase the size of the problem. In the case of RSA, as you increase the composite (public key), factoring becomes harder. Similarly, as you increase the order of the elliptic curve (do not worry if you do not know what that means at this point), the difficulty of inverting point multiplication increases.

To accommodate these larger parameters, we must deploy algorithms known collectively as BigNum algorithms.

What Are BigNums?

As developers, you are most likely aware of the size limitations of your preferred data types. In C, for example, the *int* data type can represent only up to 32767 in a portable fashion. Even though on certain platforms it may be 32 or even 64 bits in length, you cannot always count on that.

Most programming languages *at best* have a 64-bit data type at their disposal. If we had to live within these constraints directly, we would have very insecure public key algorithms. Factoring 64-bit numbers is a trivial task, for instance.

To work around these limitations, we use algorithms typically known as either multiple or fixed precision algorithms. These algorithms construct representations of larger integers using the supported data type (such as *unsigned long*) as a digit—much like we construct larger numbers in decimal by appending more digits in the range 0–9.

The digits (also known as *limbs* in various libraries) form the basis of all arithmetic. They are typically chosen to be efficient on the target platform. For example, if your machine can perform operations on *unsigned long* data types efficiently, that will likely be your digit type. The math we must perform works much the same as was taught in school with decimal. The only difference being that instead of using 10 as our radix we use a power of 2, such as 2^{32}.

Fixed and multiple precision algorithms differ little in theory, and mostly in implementation. In multiple precision algorithms, we seek to accommodate any size integer by allocating new memory as required to represent the result of an operation. This is nice to have if we are dealing with numbers of unknown dimensions. However, it has the overhead of performing heap operations in the middle of calculations, which tend to be rather slow. Fixed precision algorithms only have a limited (fixed) space to store the representation of a number. As such, there is no need for heap operations in the middle of a calculation. Fixed precision is well suited for tasks where the dimensions of the inputs are known in advance. For example, if you are implementing ECC P-192, you know your largest number will be 384 bits (2*192) and you can adjust accordingly.

In this text, we focus on fixed precision implementation details, as they are more efficient. We dispense with much of the discussion of BigNum math and instead defer the reader to various sources.

Further Resources

For the curious reader, various other sources cover BigNum mathematics in greater depth. The book *BigNum Math* covers the topic by presenting the development of a public domain math library, LibTomMath. It is certainly worth a read for those new to the subject. The text *The Art of Computer Programming Volume 2* also covers the topic from a theoretical angle, presenting key mathematical concepts behind efficient algorithms.

The reader should also examine the freely available source code for the LibTomMath and TomsFastMath packages, which implement multiple and fixed (respectively) precision arithmetic. The source codes for both are well documented and free to use for any purpose.

Key Algorithms

Certain key algorithms when implemented correctly can make a BigNum library perform very well. Even though there are dozens of algorithms involved in something like an elliptic curve point addition, there are only four algorithms of critical importance: multiplication, squaring, reduction, and modular inversion.

When performing typical public key operations, the processor time usage can usually broken down from most to least to reduction, multiplication, squaring, and modular inversion—which gives you a good idea of where to spend the most amount of time when optimizing.

In practice, we are talking about the *same* algorithms that are taught to young children for things like multiplication. The key difference is *how* they are implemented. Things like accumulator structure, loop unrolling, and opcode selection can make all the difference. We will present the four algorithms using code suitable for x86, PPC, and ARM processors using GCC as our development tool of choice.

The Algorithms

Represent!

Before we can dive into the algorithms required for efficient public key cryptography, we must discuss the structure we will use to represent these integers. For our purposes, we will borrow from the TomsFastMath package, which uses a structure similar to the following:

```
typedef struct {
   fp_digit dp[FP_SIZE];
   int      used,
            sign;
} fp_int;
```

where *fp_digit* is a type for a single digit, usually equivalent to *unsigned long* but can change to suit the platform. Along with that type is an *fp_word* type that we have not seen yet. It is required to be twice the size of an *fp_digit* and be able to hold the product of two maximum valued *fp_digit* variables. The *FP_SIZE* macro is the default maximum size of an integer. It is based on the size of a digit and the number of bits required. For instance, if *fp_digit* is a 32-bit type and you wish to represent up to 384-bit integers, *FP_SIZE* must be at least 12.

The *used* flag indicates how many digits of the *dp* array are nonzero. This allows us to manipulate shorter integers more efficiently. For example, just because your integers can be up to (say) 12 digits does not mean they all are going to be that big. Performing 12-digit operations on numbers that may only be 6 digits long wastes time and resources.

The *sign* flag denotes the sign of the integer. It is 0 for non-negative, and nonzero for negative. This means that our integers are always unsigned and only this flag controls the behavior.

The digits of the *fp_int* type are used in little endian order fashion. For example, with a 32-bit *fp_digit* if the *dp* array contained {1, 2, 3, 4, 0, 0, 0, ..., 0}, that would represent the integer $1 + 2*2^{32} + 3*2^{64} + 4*2^{96}$.

Multiplication

Multiplication, much like most BigNum problems, can be solved with a variety of algorithms. Algorithms such as Karatsuba and Toom-Cook linearized multiplication claim efficient asymptotic time requirements, but are in fact not that useful in practice—at least, not with typical software platforms. They do come in handy with hardware platforms, however.

It turns out the most profitable way to multiply two numbers of the size we use in public key algorithms is to use the basic $O(n^2)$ long-hand algorithm. That is, if we are multiplying *A* by *B*, we would multiply every digit of *B* by every digit of *A* and accumulate (add up) all of the products. Assuming that *A* and *B* have the same number of digits, *n*, this process requires n^2 single precision multiplications (that is, multiplications of *fp_digit* types) (Figure 8.1).

Figure 8.1 Multiplication Algorithm

```
Input
    A, B:   Integers
Output
    C:      Product of A and B

1. Zero C
2. $pa = A.used + B.used$
3. if $pa >= FP\_SIZE$ then
   1. pa = FP_SIZE – 1
4. $\{c0, c1, c2\} = \{0, 0, 0\}$
5. for $ix$ from 0 to $pa - 1$ do
   1. $ty = MIN(ix, B.used - 1)$
   2. $tx = ix - ty$
   3. $iy = MIN(A.used - tx, ty + 1)$
   4. for $iz$ from 0 to $iy - 1$ do
      i. {c0, c1, c2} = {c0, c1, c2} + A.dp[tx + iz] * B.dp[ty – iz]
   5. C.dp[ix] = c0
   6. $\{c0, c1, c2\} = \{c1, c2, 0\}$
6. $C.sign = A.sign$ XOR $B.sign$
7. $C.used = pa$
8. Return C
```

This algorithm produces one digit of the product per iteration of the loop (started on step 5). We use a three-digit accumulator {*c0*, *c1*, *c2*} to accumulate the products inside the inner loop. This is accomplished by adding the two-digit product to the accumulator, and then propagating the carry to the third digit. It may sound complicated but, as we will see shortly, it is entirely efficient.

Outside of the inner loop, the lowest digit of the accumulator holds the digit of the product in question. We store *c0* to *C* and then shift the accumulator down—*c1* becomes *c0*, *c2* becomes *c1*, and then *c2* is zeroed.

The MIN() macro is used to determine the smallest of the two operands. Using it in the loop may seem like a branch hazard, and in practice, it is. However, as we will see, loop unrolling can remove this from our code. The performance of the implementation depends mostly on the ability to perform the single inner loop step efficiently.

Lets examine the generic multiply from TomsFastMath before we look at the macros that make this feasible.

Ripped from **fp_mul_comba.c:**

```
/* generic PxQ multiplier */
void fp_mul_comba(fp_int *A, fp_int *B, fp_int *C)
{
   int       ix, iy, iz, tx, ty, pa;
   fp_digit  c0, c1, c2, *tmpx, *tmpy;
   fp_int    tmp, *dst;

   COMBA_START;
   COMBA_CLEAR;
```

These two macros start our accumulator and get the rest of the algorithm ready. On platforms on which we use special registers (such as XMM with x86-sse2), we may need to do something special (save them) before executing this code. It is entirely possible that COMBA_START is an empty macro.

```
   /* get size of output and trim */
   pa = A->used + B->used;
   if (pa >= FP_SIZE) {
      pa = FP_SIZE-1;
   }

   if (A == C || B == C) {
      fp_zero(&tmp);
      dst = &tmp;
   } else {
      fp_zero(C);
      dst = C;
   }
```

So far, this is much like our pseudo code (which was actually derived from this code).

```
   for (ix = 0; ix < pa; ix++) {
      /* get offsets into the two bignums */
      ty = MIN(ix, B->used-1);
      tx = ix - ty;

      /* setup temp aliases */
      tmpx = A->dp + tx;
      tmpy = B->dp + ty;

      /* this is the number of times the loop will iterrate
         while (tx++ < a->used && ty-- >= 0) { ... }
       */
      iy = MIN(A->used-tx, ty+1);
```

At this point, our inner loop is ready to execute. We use pointer aliases *tmpx* and *tmpy* to point to the digits of A and B, respectively. This makes our inner loop simpler.

```
      /* execute loop */
      COMBA_FORWARD;
      for (iz = 0; iz < iy; ++iz) {
          MULADD(*tmpx++, *tmpy--);
      }
```

COMBA_FORWARD performs the operation of shifting the accumulator. MULADD performs the operation of {*c0*, *c1*, *c2*} += **tmpx* * **tmpy*. At this point, we have no idea how this will be done. The point of showing this coding technique is to illustrate the power of macros.

```
      /* store term */
      COMBA_STORE(dst->dp[ix]);
```

COMBA_STORE stores the least significant digit of the accumulator. It is a macro, since the accumulator could be in a special register that GCC cannot easily see.

```
  }
  COMBA_FINI;
```

COMBA_FINI is a macro that lets us clean up. For example, if we had used MMX, we would have to put an EMMS opcode here. Like COMBA_START, it can be an empty macro.

```
  dst->used = pa;
  dst->sign = A->sign ^ B->sign;
  fp_clamp(dst);
  fp_copy(dst, C);
}
```

Multiplication Macros

At this point, we have C code for multiplication, but we have no idea how the most vital portions of the code actually work. We have cleverly hidden their details in a series of C preprocessor macros. Why would we do this? To make the code harder to follow? To make it harder for others to use? The answer is *diversity*.

Our macro scheme is the basis of a library known as TomsFastMath. It currently targets four *different* architectures without re-writing significant portions of the code. Our macros are flexible enough to allow us to work with x86 in 32- and 64-bit mode (as well with SSE2), 32 and 64-bit PPC, and ARMv4 platforms—even though the instruction sets of the x86, PPC, and ARM do not resemble one another.

First, let's examine the portable C implementation of these macros just to get a good idea of what they are supposed to accomplish.

```
#define COMBA_START
```

This macro has no use inside the logic of the multiplication other than to make it easier to support various platforms.

```
#define COMBA_CLEAR \
   c0 = c1 = c2 = 0;
```

This macro zeros the accumulator.

```
#define COMBA_FORWARD \

   do { c0 = c1; c1 = c2; c2 = 0; } while (0);
```

This macro shifts the accumulator to the right and inserts a zero in the most significant digits place.

```
#define COMBA_STORE(x) \
   x = c0;
#define COMBA_STORE2(x) \
   x = c1;
```

These two macros store values out of the accumulator. We have seen COMBA_STORE, but not COMBA_STORE2, as it is part of the unrolled code.

```
#define COMBA_FINI
```

COMBA_FINI is here for support only and not required. In this case, it is simply an empty macro.

```
#define MULADD(i, j) \
   do { fp_word t; \
   t = (fp_word)c0 + ((fp_word)i) * ((fp_word)j); c0 = t; \
   t = (fp_word)c1 + (t >> DIGIT_BIT);c1 = t; c2 += t >> DIGIT_BIT; \
   } while (0);
```

This performs the inner loop multiplication and accumulate. We use a double precision *fp_word* to hold the product and then add it to the accumulator. By using double precision types, we avoid the otherwise required test for overflow. This is because the C language lacks an "add with carry" operation otherwise required to propagate carry bits.

So far, of all the platforms we support with our code, the only macro that changes is MULADD. Table 8.1 lists various platforms and their respective macros.

Table 8.1 MULADD Macros for Various Platforms

x86_32	#define MULADD(i, j)\ asm(\ "movl %6,%%eax \n\t"\ "mull %7 \n\t"\ "addl %%eax,%0 \n\t"\ "adcl %%edx,%1 \n\t"\ "adcl $0,%2 \n\t"\ :"=r"(c0), "=r"(c1), "=r"(c2) :"0"(c0),"1"(c1),"2"(c2),"m"(i),"m"(j) :"%eax","%edx","%cc");

Continued

Table 8.1 continued MULADD Macros for Various Platforms

x86_64	#define MULADD(i, j)\ asm(\ "movq %6,%%rax \n\t"\ "mulq %7 \n\t"\ "addq %%rax,%0 \n\t"\ "adcq %%rdx,%1 \n\t"\ "adcq $0,%2 \n\t"\ :"=r"(c0), "=r"(c1), "=r"(c2) :"0"(c0),"1"(c1),"2"(c2),"m"(i),"m"(j) :”%rax”,”%rdx”,”%cc”);
x86_32 + SSE2	#define MULADD(i, j)\ asm(\ "movd %6,%%mm0 \n\t"\ "movd %7,%%mm1 \n\t"\ "pmuludq %%mm1,%%mm0\n\t"\ "movd %%mm0,%%eax \n\t"\ "psrlq $32,%%mm0 \n\t"\ "addl %%eax,%0 \n\t"\ "movd %%mm0,%%eax \n\t"\ "adcl %%eax,%1 \n\t"\ "adcl $0,%2 \n\t"\ :"=r"(c0), "=r"(c1), "=r"(c2) :"0"(c0),"1"(c1),"2"(c2),"m"(i),"m"(j) :"%eax","%cc");
PPC32	#define MULADD(i, j)\ asm(\ " mullw 16,%6,%7 \n\t" \ " addc %0,%0,16 \n\t" \ " mulhwu 16,%6,%7 \n\t" \ " adde %1,%1,16 \n\t" \ " addze %2,%2 \n\t" \ :"=r"(c0), "=r"(c1), "=r"(c2) :"0"(c0),"1"(c1),"2"(c2),"r"(i),"r"(j) :"16");

Continued

Table 8.1 MULADD Macros for Various Platforms

ARMv4	#define MULADD(i, j)\ asm(\ "UMULL r0,r1,%6,%7 \n\t"\ "ADDS %0,%0,r0 \n\t"\ "ADCS %1,%1,r1 \n\t"\ "ADC %2,%2,#0 \n\t"\ :"=r"(c0), "=r"(c1), "=r"(c2) :"0"(c0),"1"(c1),"2"(c2),"r"(i),"r"(j) :"r0", "r1", "%cc");

All five macros accomplish the same goal, but with different architectures in mind. They all multiply *i* by *j* and accumulate the product in {*c0*, *c1*, *c2*}.

The x86_32 and x86_64 macros use the MUL instruction from the x86 instruction set. They load the *i* operand into the EAX (RAX, resp.) register and then perform a multiply against the *fp_digit* pointed to by *j*. The product in the x86 instruction set is always stored in the EDX:EAX (RDX:RAX resp.) registers. This product is then accumulated into the three-digit accumulator {*c0*, *c1*, *c2*}, which we ask GCC to alias to processor registers. When GCC has parsed this macro, it turns into assembler output resembling the following.

```
movq  (%rsp),%rax
mulq  8(%rsp)
addq  %rax,%rcx
adcq  %rdx,%rsi
adcq  $0,%rdi
```

For those who cannot read x86_64 assembler, GCC has assigned {*c0*, *c1*, *c2*} to {rcx, rsi, rdi}, three processor registers. This is partly what makes the code so efficient. The products are accumulated without additional memory access.

The x86_32 SSE2 code is meant for Pentium 4 Northwood (before Prescott) processors. In these processors, the FPU is used for all integer MUL instructions, which means that if you simply use the FPU directly, you can get the product faster. Intel later improved their cores and this is no longer the case. This code is not efficient on AMD processors, at least not compared to the integer multiplier AMD has.

It would seem that using SSE2 to perform two 32-bit multiplications in parallel would be a faster way to perform multiplication. This is not true, however. The AMD64 (and Opteron) series of processors can perform a single 64-bit multiplication in roughly five processor cycles. You would need four times as many 32-bit multiplications to accomplish what you could with 64-bit multiplications. Therefore, even if you are doing two at once, you have to accomplish them in less than half the time to become faster. Currently, the SSE2 multiplication is not a single cycle on AMD64 processors, nor will it be in the near future.

The PPC32 code is another straight adaptation to the instruction set. The PPC differs from other instruction sets in that only half of the product is produced per opcode. The *mullw* instruction produces the lower 32 bits of the product, while the *mulhwu* produces higher 32 bits. We could place both multiplications back to back; however, we want to avoid clobbering[1] as many registers as possible (*clobbering* is the GCC term for destroying the contents of a register inside an assembler macro). This macro clobbers only one register and requires three others for the accumulator.

The PPC64 instruction set has similar opcodes; for instance, *mulld* and *mulhdu* perform the equivalent 64-bit multiplications. Those are currently not in the project due to lack of access to a PPC64 machine.

The ARMv4 code requires a v4 core with the M features (e.g., ARM7TDMI) or a v5 core or higher. It is perhaps the most elegant of all the code. We store the product in *r0* and *r1* and then accumulate it. The astute reader may notice we do not use the ARMv4 "multiply and accumulate" instruction. This is because it does not set the carry flag. Therefore, we could not pack 32 bits per digit if we used it.

Code Unrolling

So, now we have the general technique, and macros to plug into the code. Now we need to organize the code for real raw performance. The real boost comes from unrolling the entire inner and outer loops. If we know in advance what size numbers we are multiplying, this can pay off big.

Now, instead of unrolling a multiply by hand, or telling GCC implicitly the size of our numbers, we will craft a routine that is fully explicitly unrolled. To do this, we will write a program that emits C source code for a multiplier. For reference, we will use another source file from the TomsFastMath project. This program accepts a single dimension N as input and produces the C source code for a N-by-N multiply.

ripped from **comba_mult_gen.c:**

```
/* program emits a NxN comba multiplier */
#include <stdio.h>

int main(int argc, char **argv)
{
   int N, x, y, z;
   N = atoi(argv[1]);

   /* print out preamble */
printf(
"void fp_mul_comba%d(fp_int *A, fp_int *B, fp_int *C)\n"
```

We will name the routine after how many digits it handles. For instance, *fp_mul_comba16*() would perform a 16-by-16 multiplication.

```
"{\n"
"   fp_digit c0, c1, c2, at[%d];\n"
"\n"
"   memcpy(at, A->dp, %d * sizeof(fp_digit));\n"
"   memcpy(at+%d, B->dp, %d * sizeof(fp_digit));\n"
```

We copy both inputs into a single array, an optimization that is not always required but is a nice way to lower the number of registers used on 32-bit x86 platforms (since offsetting into the second half of *at* is free).

```
"   COMBA_START;\n"
"\n"
"   COMBA_CLEAR;\n", N, N+N, N, N, N);

   /* now do the rows */
   for (x = 0; x < (N+N-1); x++) {
printf(
"   /* %d */\n", x);
```

This out loop controls the construction of the inner loop for the *x*'th digit of the product.

```
if (x > 0) {
printf(
"   COMBA_FORWARD;\n");
}
```

If we are not on the first product, we will shift the accumulator, an optimization only really possible with a fully unrolled loop.

```
      for (y = 0; y < N; y++) {
      for (z = 0; z < N; z++) {
          if ((y+z)==x) {
             printf("   MULADD(at[%d], at[%d]); ", y, z+N);
          }
      }
      }
```

This constructs the inner loop in a brute force fashion. Fortunately, we only have to execute this once to create the source code. Essentially, we step through both inputs, and whenever their location adds to *x*, we perform a MULADD.

```
printf(
"\n"
"   COMBA_STORE(C->dp[%d]);\n", x);
   }
printf(
"   COMBA_STORE2(C->dp[%d]);\n"
"   C->used = %d;\n"
"   C->sign = A->sign ^ B->sign;\n"
"   fp_clamp(C);\n"
"   COMBA_FINI;\n"
"}\n\n\n", N+N-1, N+N);

  return 0;
}
```

This preamble closes off the function and we are finished. The output of this program resembles the following.

```
tom@box ~/libtom/tomsfastmath $ ./comba_mult_gen 2
void fp_mul_comba2(fp_int *A, fp_int *B, fp_int *C)
{
   fp_digit c0, c1, c2, at[4];
   memcpy(at, A->dp, 2 * sizeof(fp_digit));
   memcpy(at+2, B->dp, 2 * sizeof(fp_digit));
   COMBA_START;

   COMBA_CLEAR;
   /* 0 */
   MULADD(at[0], at[2]);
   COMBA_STORE(C->dp[0]);
   /* 1 */
   COMBA_FORWARD;
   MULADD(at[0], at[3]);    MULADD(at[1], at[2]);
   COMBA_STORE(C->dp[1]);
   /* 2 */
   COMBA_FORWARD;
   MULADD(at[1], at[3]);
   COMBA_STORE(C->dp[2]);
   COMBA_STORE2(C->dp[3]);
   C->used = 4;
   C->sign = A->sign ^ B->sign;
   fp_clamp(C);
   COMBA_FINI;
}
```

This function performs a 2-by-2 multiplication in a fully unrolled fashion. The payoff of unrolling the code depends on the size of the numbers and the platform in question. On the AMD64 and Opteron processors, it always pays off, especially since memory is usually in abundance. For example, compare the figures for the Opteron in Table 8.2.

Table 8.2 AMD Opteron Cycle Counts for Integer Multiplication

Size (bits)	Unrolled (cycles)	Looped (cycles)
128	53	229
256	104	346
512	272	876
1024	934	2,248

From Table 8.2, we can see that the unrolled code behaves very nicely on the Opteron. In fact, it works so well that the processor is executing more than one opcode per cycle overall, which is a nice property to have. For example, the 1024-bit multiplication requires 256 64-bit multiplications, which should on their own require 1280 cycles to process. The reason for the quicker return time is that the Opteron multiplier is pipelined, allowing it to process more than one product at once.

On platforms such as the PPC and ARM, memory typically is not abundant and one must measure carefully. While the unrolled code will always win over in terms of performance, due to the lack of control structure (e.g., the for loops) it is often difficult to pack the code. For example, a 6-by-6 multiplier, something you may use for ECC P-192, requires 1088 bytes when fully unrolled on the ARMv4 processors (tested with GCC 4.1.1 on an ARM7TDMI platform). This may not seem like a lot in most cases; however, the growth of the function is quadratic. For example, a 12-by-12 multiplier, twice the dimension, requires four times the space (4036 bytes).

The RSA algorithm (see Chapter 9, "Public Key Algorithms") requires numbers starting in the 32-digit (on 32-bit platforms) range. A 32-by-32 multiplier on the ARM when fully unrolled requires 27,856 bytes. This is very likely far too much memory for most environments. Fortunately, algorithms such as ECC (see Chapter 9) use small numbers, making loop unrolling for multiplication tractable.

As a general rule of thumb, on most efficient RISC cores, if you are dealing 10 or fewer digits it is a good tradeoff to fully unroll the loop. Of course, if you have the memory to spare and require performance, it is almost always a good tradeoff.

Squaring

Squaring is a special case of multiplication that arises often in public key operations. To the casual observer, squaring is nothing more than multiplying a number against itself. However, there is a very simple optimization to be performed.

Suppose we are multiplying two two-digit numbers *ab* and *cd*. To compute the product, we need to find the products *ac*, *ad*, *bc*, and *bd*. Seems simple. Now, what if *cd* equals *ab*? Then we are computing the products *aa*, *ab*, *ba*, and *bb*. From this, we can see that *ab* and *ba* are the same value and only need to be computed once. Overall, for any *n*-digit squaring, $(n^2 + n)/2$ unique products have to be computed. This makes squaring roughly twice as fast as multiplying for equal length integers.

Like the multiplication algorithms, we use a set of portable macros to make the code more versatile. Squaring is a bit more complicated than multiplication, so we will use more macros. First, let us examine the rolled generic squaring code (taken from TomsFastMath).

ripped from **fp_sqr_comba_generic.c:**

```
/* generic comba squarer */
void fp_sqr_comba(fp_int *A, fp_int *B)
{
  int       pa, ix, iz;
  fp_digit  c0, c1, c2;
  fp_int    tmp, *dst;
#ifdef TFM_ISO
  fp_word   tt;
#endif

  /* get size of output and trim */
  pa = A->used + A->used;
  if (pa >= FP_SIZE) {
     pa = FP_SIZE-1;
  }
```

Here we are computing the number of digits in the final product. We truncate as required as to not overflow the destination.

```
    /* number of output digits to produce */
    COMBA_START;
    CLEAR_CARRY;
```

As before, we have macros to initialize the routine (COMBA_START), and another to clear the accumulator (CLEAR_COMBA). We are using the same three register accumulator as before, except this time we are putting a slight twist on the usage.

```
    if (A == B) {
       fp_zero(&tmp);
       dst = &tmp;
    } else {
       fp_zero(B);
       dst = B;
    }

    for (ix = 0; ix < pa; ix++) {
        int      tx, ty, iy;
        fp_digit *tmpy, *tmpx;

        /* get offsets into the two bignums */
        ty = MIN(A->used-1, ix);
        tx = ix - ty;

        /* setup temp aliases */
        tmpx = A->dp + tx;
        tmpy = A->dp + ty;

        /* this is the number of times the loop will iterrate,
           while (tx++ < a->used && ty-- >= 0) { ... }
         */
        iy = MIN(A->used-tx, ty+1);

        /* now for squaring tx can never equal ty
         * we halve the distance since they approach
         * at a rate of 2x and we have to round because
         * odd cases need to be executed
         */
        iy = MIN(iy, (ty-tx+1)>>1);
```

At this point, we know we have to step *iy* times to produce the *ix*'th digit of the squaring.

```
        /* forward carries */
        CARRY_FORWARD;
```

This macro forwards the carry by shifting the three-register accumulator to the right by one register.

```
        /* execute loop */
        for (iz = 0; iz < iy; iz++) {
            SQRADD2(*tmpx++, *tmpy--);
        }
```

This is a similar inner loop to the MULADD we saw previously, except in this case, the product of *tmpx and *tmpy are added *twice* to the accumulator. This loops handles all the duplicate terms in the final product by performing the multiplication once and doubling the outcome.

The astute reader may notice that we could keep separate accumulators and double outside the loop. This will arise shortly.

```
      /* even columns have the square term in them */
      if ((ix&1) == 0) {
          SQRADD(A->dp[ix>>1], A->dp[ix>>1]);
      }
```

Even digit outputs have a single square term included in the product. This is effectively the MULADD macro from before.

```
      /* store it */
      COMBA_STORE(dst->dp[ix]);
```

This macro stores the least significant digit of the accumulator. It does not shift the accumulator.

```
  }

  COMBA_FINI;
```

This macro cleans up the machine state (e.g., if you were using MMX).

```
  /* setup dest */
  dst->used = pa;
  fp_clamp (dst);
  if (dst != B) {
     fp_copy(dst, B);
  }
}
```

This function is our generic rolled squaring code. It can be very compact when compiled without loop unrolling, and as such is useful for memory limited embedded platforms. It is, however, none too speedy. Simply unrolling it through a compiler switch will not get us the full speed we are seeking. Again, we will write a simple C source generator for squaring.

ripped from **comba_sqr_gen.c:**

```
#include <stdio.h>

int main(int argc, char **argv)
{
   int x, y, z, N, f;
   N = atoi(argv[1]);

printf(
"void fp_sqr_comba%d(fp_int *A, fp_int *B)\n"
"{\n"
"   fp_digit *a, b[%d], c0, c1, c2, sc0, sc1, sc2;\n"
```

```
"\n"
"   a = A->dp;\n"
"   COMBA_START; \n"
"\n"
"   /* clear carries */\n"
"   CLEAR_CARRY;\n"
"\n"
"   /* output 0 */\n"
"   SQRADD(a[0],a[0]);\n"
"   COMBA_STORE(b[0]);\n", N, N+N);

   for (x = 1; x < N+N-1; x++) {
printf(
"\n   /* output %d */\n"
"   CARRY_FORWARD;\n   ", x);

       for (f = y = 0; y < N; y++) {
           for (z = 0; z < N; z++) {
               if (z != y && z + y == x && y <= z) {
                  ++f;
               }
           }
       }

   if (f <= 2) {
       for (y = 0; y < N; y++) {
           for (z = 0; z < N; z++) {
               if (y<=z && (y+z)==x) {
                  if (y == z) {
                     printf("SQRADD(a[%d], a[%d]); ", y, y);
                  } else {
                     printf("SQRADD2(a[%d], a[%d]); ", y, z);
                  }
               }
           }
       }
   } else {
      // new method
      /* do evens first */
       f = 0;
       for (y = 0; y < N; y++) {
           for (z = 0; z < N; z++) {
               if (z != y && z + y == x && y <= z) {
                  if (f == 0) {
                     // first double
                     printf("SQRADDSC(a[%d], a[%d]); ", y, z);
                     f = 1;
                  } else {
                     printf("SQRADDAC(a[%d], a[%d]); ", y, z);
                  }
               }
           }
       }
       // forward the carry
       printf("SQRADDDB; ");
       if ((x&1) == 0) {
```

```
078             // add the square
079             printf("SQRADD(a[%d], a[%d]); ", x/2, x/2);
080          }
081       }
082   printf("\n   COMBA_STORE(b[%d]);\n", x);
083      }
084   printf("   COMBA_STORE2(b[%d]);\n", N+N-1);
085
086   printf(
087   "   COMBA_FINI;\n"
088   "\n"
089   "   B->used = %d;\n"
090   "   B->sign = FP_ZPOS;\n"
091   "   memcpy(B->dp, b, %d * sizeof(fp_digit));\n"
092   "   fp_clamp(B);\n"
093   "}\n\n\n", N+N, N+N);
094
095     return 0;
096   }
```

This program generates fully unrolled squaring routines for various input sizes. We can see some of the same macro names as from the rolled squaring code. There are a few new ones now, though.

Recall how we earlier noted that we are doubling all the products we accumulate? Well, it turns out we can optimize that out in certain circumstances. The loop on line 38 counts the number of multiplications required for the given output digit (the counting method is brute force; fortunately, our inputs are small so brute force is a sensible approach given the simplicity). If there are more than two, we accumulate all of the double products, and then double them once afterward.

To accomplish this, we added new macros. SQRADDSC performs a multiplication and the product in a *second* accumulator (zeroing the third register at the same time). SQRADDAC does what MULADD does, except adds the product to the second accumulator. Finally, SQRADDDB doubles the second accumulator and adds the outcome to the first accumulator.

Let us examine the output for an eight-digit squaring.

```
void fp_sqr_comba8(fp_int *A, fp_int *B)
{
   fp_digit *a, b[16], c0, c1, c2, sc0, sc1, sc2;
   a = A->dp;
   COMBA_START;

   /* clear carries */
   CLEAR_CARRY;

   /* output 0 */
   SQRADD(a[0],a[0]);
   COMBA_STORE(b[0]);

   /* output 1 */
   CARRY_FORWARD;
   SQRADD2(a[0], a[1]);
   COMBA_STORE(b[1]);
<snip>
   /* output 5 */
   CARRY_FORWARD;
   SQRADDSC(a[0], a[5]); SQRADDAC(a[1], a[4]); SQRADDAC(a[2], a[3]); SQRADDDB;
   COMBA_STORE(b[5]);
<snip>
```

As we can see from code, the program recognizes that the 5th output requires three products and uses the secondary registers. You may wonder why we only perform this optimization for three or more terms. This is because we always have to perform at least two additions—one to double the second accumulator, and another to add it to the first accumulator. Recall, we cannot simply double the first accumulator, as it has the square terms (even columns, using SQRADD) that must not be doubled.

Let us compare rolled and unrolled code in Table 8.3.

Table 8.3 AMD Opteron Cycle Counts for Integer Squaring

Size (bits)	Unrolled	Looped
128	40	223
256	78	338
512	231	722
1024	652	2,145

Again, there is a huge savings in cycle counts by using code unrolling. There is also quadratic growth in code size. In this case, the code growth follows $n^2/2$; that is, with an n-fold increase in input size, you expect roughly an $n^2/2$-fold increase in code size (Table 8.4).

Table 8.4 Unrolled Squaring Code Size for 64-bit x86

Number of Digits	Code Size (bytes)
4	458
8	1,169
16	3,490
32	13,138

Table 8.4 shows that the growth of the squaring code size is still close to quadratic in practice.

Squaring Macros

Our macros for squaring are trivial to derive from the multiplication macros. In the interest of saving trees, we will refer the reader to the free TomsFastMath package. Within the file fp_sqr_comba.c, the reader will find the equivalent macros for all the platforms supported. However, for completeness, we will show the ISO C implementation of these macros.

```
#define CLEAR_CARRY \
   c0 = c1 = c2 = 0;
```

This clears the three accumulator registers (in this case, they are variables on the stack).

```
#define COMBA_STORE(x) \
   x = c0;
```

This stores the least significant register of the accumulator.

```
#define COMBA_STORE2(x) \
   x = c1;
```

This stores the middle register of the accumulator.

```
#define CARRY_FORWARD \
   do { c0 = c1; c1 = c2; c2 = 0; } while (0);
```

This shifts the accumulator to the right by one register, inserting a zero in the most significant register position. Note the use of the do-while construction to make the macro safe to insert anywhere.

```
/* multiplies point i and j, updates carry "c1" and digit c2 */
#define SQRADD(i, j)                                \
   do { fp_word t;                                   \
   t = c0 + ((fp_word)i) * ((fp_word)j);  c0 = t;    \
   t = c1 + (t >> DIGIT_BIT);             c1 = t; c2 += t >> DIGIT_BIT; \
   } while (0);
```

This multiplies *i* by *j* and adds the result to the accumulator once.

```
/* for squaring some of the terms are doubled... */
#define SQRADD2(i, j)                              \
   do { fp_word t;                                  \
   t  = ((fp_word)i) * ((fp_word)j);                \
   tt = (fp_word)c0 + t;                 c0 = tt;   \
   tt = (fp_word)c1 + (tt >> DIGIT_BIT); c1 = tt;   \
   c2 += tt >> DIGIT_BIT;                           \
   tt = (fp_word)c0 + t;                 c0 = tt;   \
   tt = (fp_word)c1 + (tt >> DIGIT_BIT); c1 = tt;   \
   c2 += tt >> DIGIT_BIT;                           \
   } while (0);
```

This performs the same operation as SQRADD, but instead of adding the produce once, it does it twice. We do not double the product, since it would not fit in an fp_word variable. We are forced to add twice instead.

```
#define SQRADDSC(i, j)                     \
   do { fp_word t;                          \
      t =  ((fp_word)i) * ((fp_word)j);     \
      sc0 = (fp_digit)t;                    \
      sc1 = (t >> DIGIT_BIT); sc2 = 0;      \
   } while (0);
```

This macro performs multiplies the inputs and stores it in a new second accumulator. We use the variables {*sc0*, *sc1*, *sc2*} instead of the defaults, since we want to accumulate products individually before doubling it.

```
#define SQRADDAC(i, j)                  \
   do { fp_word t;                      \
   t = sc0 + ((fp_word)i) * ((fp_word)j);\
   sc0 = t;                             \
   t = sc1 + (t >> DIGIT_BIT);          \
   sc1 = t;                             \
   sc2 += t >> DIGIT_BIT;               \
   } while (0);
```

This macro performs the SQRADD macro, but adds the product to the second accumulator.

```
#define SQRADDDB                                                \
   do { fp_word t;                                              \
   t = ((fp_word)sc0) + ((fp_word)sc0) + c0;                    \
   c0 = t;                                                      \
   t = ((fp_word)sc1) + ((fp_word)sc1) + c1 + (t >> DIGIT_BIT);  \
   c1 = t;                                                      \
   c2 = c2 + ((fp_word)sc2) + ((fp_word)sc2) + (t >> DIGIT_BIT); \
   } while (0);
```

Finally, this macro doubles the second accumulator and adds it to the first. These secondary macros allow us to accumulate the double products without doubling them redundantly.

Montgomery Reduction

The last performance critical algorithm we need for a competent BigNum library is Montgomery reduction. Modular reduction refers to the process of computing the remainder of a division; specifically, we are looking for the remainder of one integer divided by another called the modulus.

In the case of public key algorithms, we are going to have special circumstances on our side. The numbers we are dividing are always no larger than the square of the modulus. From this fact, we can construct various optimized reduction algorithms. In this case, we will examine Montgomery reduction, as it is efficient and versatile. The following code is taken from TomsFastMath and performs generic rolled Montgomery reduction.

```
/* computes x/R == x (mod N) via Montgomery Reduction */
void fp_montgomery_reduce(fp_int *a, fp_int *m, fp_digit mp)
{
   fp_digit c[FP_SIZE], *_c, *tmpm, mu;
   int      oldused, x, y, pa;

   /* bail if too large */
   if (m->used > (FP_SIZE/2)) {
      return;
   }
```

Since we are working with fixed precision, we limit the input modulus length to half of the maximum BigNum length.

```
    pa = m->used;

    /* copy the input */
    oldused = a->used;
    for (x = 0; x < oldused; x++) {
        c[x] = a->dp[x];
    }
    for (; x < 2*pa+1; x++) {
        c[x] = 0;
    }
```

At this point, we have copied the lower half of the input we want to find the residue of. This is equivalent to reducing the input modulo a power of two, which is the first step of Montgomery reduction.

```
    MONT_START;
```

This macro allows us to do any initialization required for the inner loop.

```
    for (x = 0; x < pa; x++) {
        fp_digit cy = 0;
        /* get Mu for this round */
        LOOP_START;
```

The LOOP_START macro multiplies *c[x]* by *mp* to create the Mu digit for this iteration of the loop. It can be done by assembler code, but as we will see, it is simpler to just use C on most platforms.

```
        _c   = c + x;
        tmpm = m->dp;
        y = 0;

        for (; y < pa; y++) {
           INNERMUL;
           ++_c;
        }
```

Our inner loop performs a simple single-digit multiply and accumulate. We can unroll it on x86_64 platforms somewhat by deferring memory loads; for now, though, we will leave it simple.

```
        LOOP_END;
```

The LOOP_END macro is used only to fetch the *cy* register. That is, if LOOP_START and INNERMUL alias *cy* to a machine register the C compiler cannot see, this macro will fetch it. After this statement, the *cy* variable must be in sync with the machine register contents (if any).

```
        while (cy) {
            PROPCARRY;
            ++_c;
        }
```

This loop propagates the carry upward. The PROPCARRY macro performs the job of propagating the carry of *cy* into *_c*[0]. The *_c* pointer is incremented and the loop re-iterated as long as *cy* is not zero.

```
  }

  /* now copy out */
  _c   = c + pa;
  tmpm = a->dp;
  for (x = 0; x < pa+1; x++) {
     *tmpm++ = *_c++;
  }

  for (; x < oldused; x++)   {
     *tmpm++ = 0;
  }
```

At this point, we have copied out the residue and zeroed the high numbers.

```
  MONT_FINI;
```

This macro is placed to clear up the machine state as required.

```
  a->used = pa+1;
  fp_clamp(a);

  /* if A >= m then A = A - m */
  if (fp_cmp_mag (a, m) != FP_LT) {
    s_fp_sub (a, m, a);
  }
}
```

The final bit of code clamps the BigNum to remove unused digits and subtracts the modulus for the "off by one" case that is possible with Montgomery reduction.

Montgomery Reduction Unrolling

It turns out that this code coupled with optimal platform macros does not yield favorable results with code unrolling. That is, the performance gains rarely warrant the code size increase. However, if you are dead set on performance, you can examine the source for TomsFastMath, which has unrolled Montgomery reduction to edge out that last 2 percent of performance.

Montgomery Macros

Table 8.5 lists the Montgomery reduction macros for x86, ARM, and PPC. All of them use a C-based LOOP_START macro defined as follows.

```
#define LOOP_START \
   mu = c[x] * mp
```

The only exception so far is the x86_32 SSE2 code, which uses a SSE2 multiplication instead of the standard integer ALU multiplication.

Table 8.5 Montgomery Reduction Macros for x86

INNERMUL	#define INNERMUL \ asm(\ "movl %5,%%eax \n\t" \ "mull %4 \n\t" \ "addl %1,%%eax \n\t" \ "adcl $0,%%edx \n\t" \ "addl %%eax,%0 \n\t" \ "adcl $0,%%edx \n\t" \ "movl %%edx,%1 \n\t" \ :"=g"(_c[LO]), "=r"(cy) \ :"0"(_c[LO]), "1"(cy), "g"(mu), "g"(*tmpm++) \ :"%eax", "%edx", "%cc")
PROPCARRY	#define PROPCARRY \ asm(\ "addl %1,%0 \n\t" \ "**setb** %%al \n\t" \ "**movzbl** %%al,%1 \n\t" \ :"=g"(_c[LO]), "=r"(cy) \ :"0"(_c[LO]), "1"(cy) \ :"%eax", "%cc")

Our x86 code makes use of the setb and movzbl instructions to perform a branchless carry store into *cy*. The first instruction, setb, will set the *al* register to 1 if the carry flag was set, or 0 otherwise. The second instruction, movzbl**,** will zero extend the byte register *al* into the register holding a copy of the variable *cy*. These instructions are very useful, as they allow us to avoid branching, which leaks side channel information and hinders performance (Table 8.6).

Table 8.6 Montgomery Reduction Macros for ARMv4

INNERMUL	#define INNERMUL \ asm(\ " LDR r0,%1 \n\t" \ " ADDS r0,r0,%0 \n\t" \

Continued

Table 8.6 continued Montgomery Reduction Macros for ARMv4

```
                  " MOVCS   %0,#1               \n\t" \
                  " MOVCC   %0,#0               \n\t" \
                  " UMLAL   r0,%0,%3,%4         \n\t" \
                  " STR     r0,%1               \n\t" \
              :"=r"(cy),"=m"(_c[0])
              :"0"(cy),"r"(mu),"r"(*tmpm++),"1"(_c[0])
              :"r0","%cc");
PROPCARRY     #define  PROPCARRY                        \
              asm(                                      \
                  " LDR    r0,%1                \n\t" \
                  " ADDS   r0,r0,%0             \n\t" \
                  " STR    r0,%1                \n\t" \
                  " MOVCS  %0,#1                \n\t" \
                  " MOVCC  %0,#0                \n\t" \
              :"=r"(cy),"=m"(_c[0])
              :"0"(cy),"1"(_c[0])
              :"r0","%cc");
```

Like the x86 case, we use conditional move instructions MOVCS and MOVCC to avoid branches in the code. This pair works much like the CMOV class of x86 instructions. MOVCS will perform the move if the carry flag is set, while MOVCC will perform the move if the carry flag is unset. While branching is less of a hazard on the ARM processors, it still leaks side channel data and ought to be avoided.

As a performance note, we make use of the UMLAL instruction to perform a 32x32 => 64 multiply and accumulate operation. It allows us to combine three instructions into one and save a few cycles along the way. We could not have used this in our multiplying and squaring code, as UMLAL will not set the carry flag—which is rather unfortunate as it would speed up ARM BigNum math operations (Table 8.7).

Table 8.7 Montgomery Reduction Macros for PPC

```
INNERMUL      #define  INNERMUL                          \
              asm(                                      \
                 " mullw    16,%3,%4        \n\t"     \
                 " mulhwu   17,%3,%4        \n\t"     \
                 " addc     16,16,%0        \n\t"     \
                 " addze    17,17           \n\t"     \
```

Continued

Table 8.7 continued Montgomery Reduction Macros for PPC

	" lwz 18,%1 \n\t" \ " addc 16,16,18 \n\t" \ " addze %0,17 \n\t" \ " stw 16,%1 \n\t" \ :"=r"(cy),"=m"(_c[0]) :"0"(cy),"r"(mu),"r"(tmpm[0]),"1"(_c[0]) :"16", "17", "18","%cc"); ++tmpm;
PROPCARRY	#define PROPCARRY \ asm(\ " lwz 16,%1 \n\t" \ " addc 16,16,%0 \n\t" \ " stw 16,%1 \n\t" \ " xor %0,%0,%0 \n\t" \ " addze %0,%0 \n\t" \ :"=r"(cy),"=m"(_c[0]) :"0"(cy),"1"(_c[0]) :"16","%cc");

Putting It All Together

Core Algorithms

We have only explored three core algorithms that occupy the dominant amount of processor time when performing public key operations. They by no means form an exhaustive list of all functions required to perform the public key operations.

Aside from the basic addition, subtraction, and comparisons, operations such as multiplicative inversion, modular exponentiation, and greatest common divisor are required. These algorithms are less platform specific and well covered in several texts.

There are several algorithms for multiplicative inversion, such as those based on the extended Euclidean algorithm and the almost inverse. The Euclidean algorithm is by far the most versatile and commonplace. It is however, not the fastest. The almost inverse algorithm is optimal for cases where the modulus is odd (specifically has the least significant bit), and is ideal for hardware implementations. Usually credited to Kaliski as the "Almost Montgomery Inversion," it was also part of a 1995 Crypto paper[2] whereby they presented an algorithm that quickly computes a value that is roughly congruent to the true modular inverse (Richard Schroeppel, Hilarie Orman, Sean O'Malley, Oliver Spatscheck, "Fast Key Exchange with Elliptic Curve Systems," 1995, Advances in Cryptology—Crypto '95, Edited by Don Coppersmith, Springer-Verlag). The final inversion is computed from the rough estimate efficiently with shifts and additions.

Modular exponentiation is another problem best solved with a variety of algorithms depending on the situation. *The Handbook of Applied Cryptography* outlines several, such as basic left-to-right exponentiation, windowed exponentiation, and vector chain exponentiation. These are also covered in a more academic setting in *The Art Of Computer Programming Volume 2* by Knuth. His text discusses the asymptotic behaviors of various exponentiation algorithms; this knowledge is fundamental to properly develop a math library of versatile use. A more practical treatment of exponentiation is explored in the text *BigNum Math: Implementing Cryptographic Multiple Precision Arithmetic* by Tom St Denis. The latter text includes a vast array of source code useful for implementing independent BigNum libraries.

Size versus Speed

Aside from picking platform suitable algorithms, one is often presented with the choice of how to best use code and data memory. As we have seen, loop unrolling can greatly accelerate integer multiplication and squaring. However, this added speed comes at a price of size. The algorithms we use for multiplication and squaring are quadratic in nature[3] (e.g., $O(n^2)$) and the tradeoff follows this. (Technically, algorithm such as Karatsuba and Toom-Cook multiplication are not quadratic. However, they are completely inefficient with the size of numbers we will be using.)

Various algorithms such as those for addition, subtraction, and shifting are linear by nature (e.g., $O(n)$) and can be sped up relatively cheaply with code unrolling. On the whole, this will save cycles when performing public key operations. However, the savings (like the cost) will be minimal at best with most algorithms. Where unrolling these algorithms does pay off is with algorithms like the almost inverse that use no multiplications and perform a $O(n^2)$ amount of shifts and additions. A good sign if unrolling the linear operations is a good idea is to count the time spent in modular inversion. If it is significant by comparison, obviously unrolling them is a good idea.

A general rule of thumb is if you have 10 or fewer digits in your numbers (e.g., 320-bit numbers on a 32-bit platform), strongly consider loop unrolling as a good performance tradeoff. Usually at this point, the setup code required for multiplication (e.g., before and after the inner loop) eats into the cycles actually spent performing the multiplications. Keeping small multipliers rolled will save space, but the efficiency of the process will decrease tremendously. Of course, this still depends on the available memory, but experience shows it is usually a good starting point. Above 10 digits and the unrolled code usually becomes far too large to manage on embedded platforms, and you will be spending more time in the inner loop than setting it up.

Roughly speaking, if we let c represent the cycles spent managing the inner loop, and n represent the number of digits, the performance loss of rolling the loop follows nc/n^2, or simply *c/n*. This does not take into account the performance loss due to actually performing the loop (e.g., branching, decrementing counters, etc.).

Performance BigNum Libraries

Fortunately, for most developers there are a score of performance math libraries available. In all but the most extreme cases there is already a library or two to pick from, as developers are strongly discouraged from writing their own for anything other than academic exercises.

Writing a complete and functional math library takes experience and is a tricky proposition, as there are many corner cases to test for in each routine. As performance demands rise, the code usually suffers from being less direct than desired, especially when assembler is introduced to the equation.

GNU Multiple Precision Library

The GNU Multiple Precision (GMP) library is by far the oldest and most well known library for handling arbitrary length integers, rationals, and floating point numbers. The library is released under the GPLv2 license and is hosted at www.swox.com/gmp/.

This library was designed for a variety of tasks, few of which are cryptographic by nature. The goal of GMP is to efficiently, in turns of asymptotic bounds, handle as wide variety of input sizes as possible. It has algorithms that only become useful when the input size approaches tens of thousands of bits in length.

Even with the flexibility in hand, the library still performs well on cryptographic sized numbers. It has well-optimized multiplication, squaring, and exponentiation code that make it highly competitive. It has also been ported to a wide variety of machines, making it more platform independent.

The most notable failing of GMP is size. The library weighs in at a megabyte and is hard to hand tune for size. Another glaring omission is the lack of a publicly accessible Montgomery reduction function. This makes the implementation of elliptic curve cryptography harder, as one has to write his own reduction function to perform point operations.

LibTomMath Library

LibTomMath is a fairly well established library in its own right. It was designed with teaching in mind and was written using portable ISO C syntax. While it is not the fastest math library in the world, it is very platform independent and compact. It achieves roughly 30 to 50 percent of the performance of libraries such as GMP and TomsFastMath depending on the size of numbers and operation in question.

The LibTomMath package is hosted at http://math.libtomcrypt.com and is public domain. That is, it is free for all purposes and there are no license arrangements required to use the library. Due to the ISO C compliance of the library, it forms integral parts of various portable projects, including, for example, the Tcl scripting language.

LibTomMath has fewer functions than GMP, but has enough functions that constructing most public key algorithms is practical. The code uses multiple precision representations like GMP, which allows it to address a variety of tasks by accommodating sizes of numbers not known at compile time.

TomsFastMath Library

TomsFastMath is a newer math library designed by the author of LibTomMath. It features a very similar API but has been trimmed down and optimized solely for fast cryptographic mathematics. The library uses considerable code unrolling, which makes it both fast and large. Fortunately, it is configurable at build time to suit a given problem (e.g., RSA-1024 or ECC P-192) in memory limited platforms.

The TomsFastMath package is hosted at http://tfm.libtomcrypt.com and is public domain. It features optimizations for 32- and 64-bit x86 and PPC platforms andARMv4 and above processors. The package can be built in ISO C mode, but is not very fast in that mode. This project does not strive for as much portability as LibTomMath, but makes up for this failing with raw speed.

Unlike LibTomMath and GMP, TomsFastMath is not a generic purpose library. It was designed solely for cryptographic tasks, and as such makes various design decisions such as using fixed precision representations. This means, for instance, you must know in advance the largest number you will be working with before you compile the library. Moreover, routines such as the modular exponentiation and inversion only accept odd moduli values. This is because even moduli are not used in any standard public key algorithm and as such are not worth spending time thinking about in this project.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to www.syngress.com/solutions and click on the "Ask the Author" form.

Q: What is BigNum mathematics?

A: Multiple or fixed precision mathematics is the set of algorithms that allow the representation and manipulation of large integers, typically designed to compensate for the lack of intrinsic support for large integers. These algorithms use the smaller, typically fixed, integers (usually called *limbs* or *digits*) to represent large integers.

Q: Why is knowing about BigNum mathematics important?

A: Large integers form the basis of public key algorithms such as RSA, ElGamal, and Elliptic Curve Cryptography. These algorithms require large numbers to make attacks such as factoring and discrete logarithms ineffective. RSA, for example, requires numbers that are at least in the 1024-bit range, while ECC requires numbers in at least the 192-bit range. These values are not possible with the typical built-in variables supported by languages such as C, C++, and Java.

Q: What sort of algorithms are the most important to optimize?

A: The answer to this question depends on the type of public key algorithm you are using. In most case, you will need fast Montgomery reduction, multiplication, and squaring. Where the optimizations differ is in the size of the numbers. Algorithms such as ECC benefit from small unrolled algorithms, while algorithms such as RSA and ElGamal benefit from large unrolled algorithms when the memory is available. In the case of ECC, we will want to use fast fixed point algorithms, whereas with RSA, we will use sliding window exponentiation algorithms (see Chapter 9).

Q: What libraries provide the algorithms required for public key algorithms?

A: GNU MP (GMP) provides a wide variety of mathematical algorithms for a wide range of input sizes. It is provided under the GPL license at the Web site www.swox.com/gmp/. LibTomMath provides a variety of cryptography related algorithms for a variable range of input sizes. It is not as general purpose as GMP, designed mostly for cryptographic tasks. It is provided as public domain at the Web site http://math.libtomcrypt.com. TomsFastMath provides a more limited subset of cryptographic related algorithms designed solely for speed. It is much faster than LibTomMath and usually on par with or better than GMP in terms of speed. It is provided as public domain at the Web site http://tfm.libtomcrypt.com.

Chapter 9

Public Key Algorithms

Solutions in this chapter:

- What Is Public Key Cryptography?
- Goals of Public Key Cryptography
- Standard RSA Cryptography
- Standard Elliptic Curve Cryptography
- Public Key Algorithms
- Further References

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

So far, we have been discussing *symmetric* key algorithms such as AES, HMAC, CMAC, GCM, and CCM. These algorithms are known as symmetric (or shared secret) algorithms, since all parties share the same key values. Revealing this key would compromise the security of the system. This means we have been assuming that we somehow shared a key, and now we are going to answer the how part.

Public key algorithms, also known as *asymmetric* key algorithms, are used (primarily) to solve two problems that symmetric key algorithms cannot: key distribution and nonrepudiation. The first helps solve privacy problems, and the latter helps solve authenticity problems.

Public key algorithms accomplish these goals by operating asymmetrically; that is, a key is split into two corresponding parts, a public key and a private key. The public key is so named as it is secure to give out publicly to all those who ask for it. The public key enables people to encrypt messages and verify signatures. The private key is so named as it must remain private and cannot be given out. The private key is typically owned by a single person or device in most circumstances, but could technically be shared among a trusted set of parties. The private key allows for decrypting messages and the generation of signatures.

The first publicly disclosed public key algorithm was the Diffie-Hellman key exchange, which allowed, at least initially, only for key distribution between known parties. It was extended by ElGamal to a full encrypt and signature public key scheme, and is used for ECC encryption, as we will see shortly. Shortly after Diffie-Hellman was published, another algorithm known as RSA (Rivest Shamir Adleman) was publicly presented. RSA allowed for both encryption and signatures while using half of the bandwidth as ElGamal. Subsequently, RSA became standardized in various forms.

Later, in the 1980s, elliptic curves were proposed as an abelian group over which ElGamal encryption and DSA (variant of ElGamal) could be performed, and throughout the 1990s and 2000s, various algorithms were proposed that make elliptic curve cryptography an attractive alternative to RSA and ElGamal.

For the purposes of this text, we will discuss PKCS #1 standard RSA and ANSI standard ECC cryptography. They represent two of the three standard algorithms specified by NIST for public key cryptography, and in general are representative of the commercial sector demands.

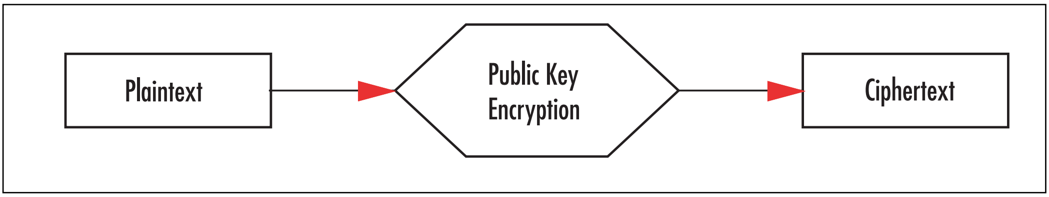
Goals of Public Key Cryptography

Public key cryptography is used to solve various problems that symmetric key algorithms cannot. In particular, it can be used to provide privacy, and nonrepudiation. Privacy is usually provided through *key distribution*, and a symmetric key cipher. This is known as *hybrid encryption*. Nonrepudiation is usually provided through *digital signatures*, and a hash function.

Privacy

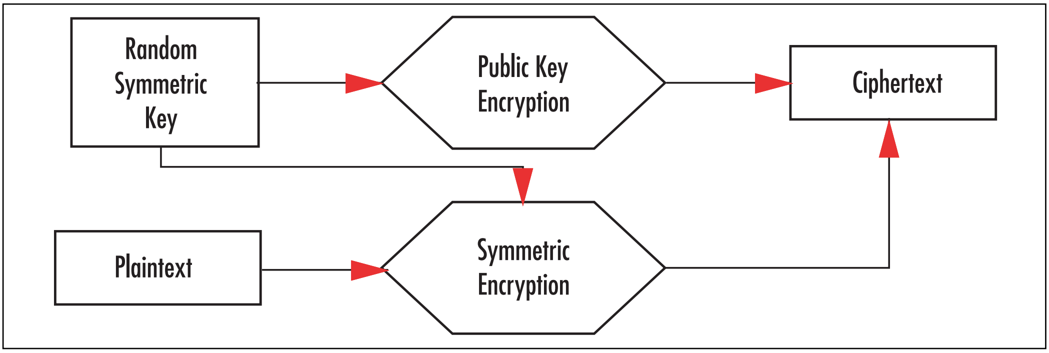
Privacy is accomplished with public key algorithms in one of two fashions. The first method is to only use the public key algorithm to encode plaintext into ciphertext (Figure 9.1). For example, RSA can accept a short plaintext and encrypt it directly. This is useful if the application must only encrypt short messages. However, this convenience comes at a price of speed. As we will see shortly, public key operations are much slower than their symmetric key counterparts.

Figure 9.1 Public Key Encryption



The second useful way of accomplishing privacy is in a mode known as *hybrid-encryption* (Figure 9.2). This mode leverages the key distribution benefits of public key encryption, and the speed benefits of symmetric algorithms. In this mode, each encrypted message is processed by first choosing a random symmetric key, encrypting it with the public key algorithm, and finally encrypting the message with the random symmetric key. The ciphertext is then the combination of the random public key and random symmetric key ciphertexts.

Figure 9.2 Hybrid Encryption



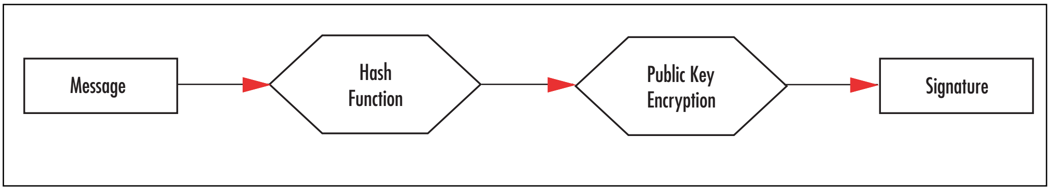
Nonrepudiation and Authenticity

Nonrepudiation is the quality of being unable to deny or refuse commitment to an agreement. In the paper world, this is accomplished with hand signatures on contracts. They have the quality that in practice they are nontrivial to forge, at least by a layperson. In the digital world, they are produced with public key signatures using a private key. The corresponding public key can verify the signature. Signatures are also used to test the authenticity of a message by verifying a signature from a trusted party.

In the Public Key Infrastructure (PKI) commonly deployed in the form of X.509 certificates (SSL, TLS), and PGP keys, a public key can be signed by an *authority* common to all users as a means of guaranteeing identity. For example, VeriSign is one of many *root certificate authorities* in the SSL and TLS domains. Applications that use SSL typically have the public key from VeriSign installed locally so they can verify other public keys VeriSign has vouched for.

Regardless of the purpose of the signature, they are all generated in a common fashion. The message being authenticated is first hashed with a secure cryptographic hash function, and the message digest is then sent through the public key signature algorithm (Figure 9.3).

Figure 9.3 Public Key Signatures



This construction of public key signatures requires the hash function be collision resistant. If it were easy to find collisions, signatures could be forged by simply producing documents that have the same hash. For this reason, we usually match the hash size and strength with the strength of the underlying public key algorithm. There is, for instance, no point in using a public key algorithm that takes only 2^{64} operations to break with SHA-256. An attacker could just break the public key and produce signatures without finding collisions.

RSA Public Key Cryptography

RSA public key cryptography is based on the problem of taking e'th roots modulo a composite. If you do not know what that means, that is ok; it will be explained shortly. RSA is unique among public key algorithms in that the message (or message digest in the case of signatures) is transformed directly by the public key primitive. As difficult as inverting this transform is (this is known as the trapdoor), it cannot be used directly in a secure fashion.

To solve this problem, RSA Security (the company) invented a standard known as Public Key Cryptographic Standards (PKCS), and their very first standard #1 details how to use RSA. The standard includes how to *pad* data such that you can apply the RSA transform in a secure fashion. There are two popular versions of this standard: v1.5 and the current v2.1. The older version is technically still secure in most circumstances, but is less favorable as the newer version addresses a couple of cases where v1.5 is not secure. For this reason, it is not recommended to implement v1.5 in new systems. Sadly, this is not always the case. SSL still uses v1.5 padding, as do a variety of PKI based applications.

RSA used to fall under a U.S. patent, which has since expired. The original RSA, including the PKCS #1 padding, are now entirely patent free. There are patents on variants of RSA such as multiprime RSA; however, these are not as popularly deployed and often are avoided for various security concerns.

RSA in a Nutshell

We will now briefly discuss the mathematics behind RSA so the rest of this chapter makes proper sense.

Key Generation

Key generation for RSA is conceptually a simple process. It is not trivial in practice, especially for implementations seeking speed (Figure 9.4).

Figure 9.4 RSA Key Generation

```
Input:
    $e$:    Public exponent
    $n$:    Desired bit length of the modulus
Output:
    $n$:    Public modulus
    $d$:    Private exponent

  1. Choose a random prime $p$ of length $n/2$ bits, such that $\gcd(p-1, e) = 1$
  2. Choose a random prime $q$ of length $n/2$ bits, such that $\gcd(q-1, e) = 1$
  3. Let $n = pq$
  4. Compute $d = e^{-1} \bmod (p - 1)(q - 1)$
  5. Return $n$, $d$
```

The e exponent must be odd, as $p - 1$ and $q - 1$ will both have factors of two in them. Typically, e is equal to 3, 17, or 65537, which are all efficient to use as a power for exponentiation. The value of d is such that for any m that does not divide n, we will have the property that $(m^e)^d$ is congruent to m mod n; similarly, $(m^d)^e$ is also congruent to the same value.

The pair e and n form the public key, while the pair d and n form the private key. A party given e and n cannot trivially compute d, nor trivially reverse the computation $c = m^e$ mod n.

For details on how to choose primes, one could consider reading various sources such as *BigNum Math*, (Tom St Denis, Greg Rose, *BigNum Math—Implementing Cryptographic Multiple Precision Arithmetic*, Syngress, 2006), which discuss the matters in depth. That text also discusses other facets required for fast RSA operations such as fast modular exponentiation. The reader could also consider *The Art Of Computer Programming*" (Donald Knuth, *The Art of*

Computer Programming, *Volume 2*, third edition, Addison Wesley) as another reference on the subject matter. The latter text treats the arithmetic from a very academic view point and is useful for teaching students the finer points of efficient multiple precision arithmetic.

RSA Transform

The RSA transform is performed by converting a message (interpreted as an array of octets) to an integer, exponentiating it with one of the exponents, and finally converting the integer back to an array of octets. The private transform is performed using the d exponent and is used to decrypt and sign messages. The public transform is performed using the e exponent and is used to encrypt and verify messages.

PKCS #1

PKCS #1 is an RSA standard[1] that specifies how to correctly encrypt and sign messages using the RSA transform as a *trapdoor* primitive (PKCS #1 is available at www.rsasecurity.com/rsalabs/node.asp?id=2125). The padding applied as part of PKCS #1 addresses various vulnerabilities that raw RSA applications would suffer.

The PKCS #1 standard is broken into four parts. First, the data conversion algorithms are specified that allow for the conversion from message to integer, and back again. Next are the cryptographic primitives, which are based on the RSA transform, and provide the trapdoor aspect to the public key standard. Finally, it specifies a section for a proper encryption scheme, and another for a proper signature scheme.

PKCS #1 makes very light use of ASN.1 primitives and requires a cryptographic hash function (even for encryption). It is easy to implement without a full cryptographic library underneath, which makes it suitable for platforms with limited code space.

PKCS #1 Data Conversion

PKCS #1 specifies two functions, OS2IP and I2OSP, which perform octet string and integer conversion, respectively. They are used to describe how a string of octets (bytes) is transformed into an integer for processing through the RSA transform.

The OS2IP function maps an octet string to an integer by loading the octets in big endian fashion. That is, the first byte is the most significant. The I2OSP functions perform the opposite without padding. That is, if the input octet string had leading zero octets, the I2OSP output will not reflect this. We will see shortly how the PKCS standard addresses this.

PKCS #1 Cryptographic Primitives

The PKCS #1 standard specifies four cryptographic primitives, but technically, there are only two unique primitives. The RSAEP primitive performs the public key RSA transform by raising the integer to e modulo n.

The RSADP primitive performs a similar operation, except it uses the d exponent instead. With the exception of leading zeros and inputs that divide the modulus n, the

RSADP function is the inverse of RSAEP. The standard specifies how RSADP can be performed with the Chinese Remainder Theorem (CRT) to speed up the operation. This is not technically required for numerical accuracy, but is generally a good idea.

The standard also specifies RSASP1 and RSAVP1, which are equivalent to RSADP and RSAEP, respectively. These primitives are used in the signature algorithms, but are otherwise not that special to know about.

PKCS #1 Encryption Scheme

The RSA recommended encryption scheme is known as RSAES-OAEP, which is simply the combination of OAEP padding and the RSAEP primitive. The OAEP padding is what provides the scheme with security against a variety of active adversarial attacks. The decryption is performed by first applying RSADP followed by the inverse OAEP padding (Figure 9.5).

Figure 9.5 RSA Encryption Scheme with OAEP

```
Input:
    $(n, e)$:     Recipients public key, $k$ denotes the length of $n$ in octets.
    $M$:          Message to be encrypted of $mLen$ octets in length
    $L$:          Optional label (salt), if not provided it is the empty string
    $hLen$:       Length of the message digest produced by the chosen hash
Output:
    $C$:          Ciphertext

 1. If $mLen > k - 2*hLen - 2$ then output "message too long" and return
 2. lHash = hash($L$)
 3. Let $PS$ be a string of $k - mLen - 2*hLen - 2$ zero octets
 4. Concatenate $lHash$, $PS$, the single octet 0x01, and M into DB
 5. Generate a random string seed of length $hLen$ octets
 6. $dbMask$ = MGF(seed, $k - hLen - 1$)
 7. $maskedDB$ = DB XOR $dbMask$
 8. $seedMask$ = MGF($maskedDB, hLen$)
 9. $maskedSeed$ = seed XOR $seedMask$
10. Concatenate the single octet 0x00, $maskedSeed$, and $maskedDB$ into $EM$
11. $m$ = OS2IP($EM, k$)
12. $c$ = RSAEP(($e, n$), $m$)
13. $C$ = I2OSP($c, k$)
14. return $C$
```

The hash function is not specified in the standard, and users are free to choose one. The MGF function is specified as in Figure 9.6.

Figure 9.6 MGF

```
Input:
    mgfSeed:    The mask generation seed data
    maskLen:    The length of the mask data required
Output:
    mask:       The output mask

1. Let $T$ be the empty string
2. For counter from 0 to ceil(maskLen / hLen) – 1 do
    1. $C$ = I2OSP(counter, 4)
    2. $T = T || \text{hash}(mgfSeed || C)$
3. Return the leading maskLen octets of $T$
```

RSA decryption is performed by first applying RSADP, and then the inverse of the OAEP padding scheme. Decryptions must be rejected if they are missing any of the constant bytes, the *PS* string, or the *lHash* string.

NOTE

The RSA OAEP padding places limits on the size of the plaintext you can encrypt. Normally, this is not a problem, as hybrid mode is used; however, it is worthy to know about it. The limit is defined by the RSA modulus length and the size of the message digest with the hash function chosen.

With RSA-1024—that is, RSA with a 1024-bit modulus—and SHA-1, the payload limit for OAEP is 86 octets. With the same modulus and SHA-256, the limit is 62 bytes. The limit is generically stated as $k - 2*hLen - 2$.

For hybrid mode, this poses little problem, as the typical largest payload would be 32 bytes corresponding to a 256-bit AES key.

PKCS #1 Signature Scheme

Like the encryption scheme, the signature scheme employs a padding algorithm before using the RSA primitive. In the case of signatures, this padding algorithm is known as PSS, and the scheme as EMSA-PSS.

The EMSA-PSS signature algorithm is specified as shown in Figure 9.7.

Figure 9.7 RSA Signature Scheme with PSS

```
Input:
    (n, d):     RSA Private Key
        M:      Message to be signed
    emBits:     The size of the modulus in bits
    emLen:      Equal to ceil(embits/8)
    sLen:       Salt length
Output:
    S:          Signature

 1. mHash = hash(M)
 2. If emLen < hLen + sLen + 2, output "encode error", return.
 3. Generate a random string salt of length sLen octets.
 4. M' = 0x00 00 00 00 00 00 00 00 || mHash || salt
 5. H = Hash(M')
 6. Generate an octet string PS consisting of emLen – sLen – hLen – 2 zero octets.
 7. DB = PS || 0x01 || salt
 8. dbMask = MGF(H, emLen – hLen – 1)
 9. maskedDB = DB XOR dbMask
10. Set the leftmost 8*emLen – emBits of the leftmost octet of maskedDB to zero
11. EM = maskedDB || H || 0xBC
12. s = OS2IP(EM, emLen)
13. s' = RSASP1((d, n), s)
14. S = I2OSP(s', emLen)
15. return 15
```

The signature is verified by first applying RSAVP1 to the signature, which returns the value of *EM*. We then look for the 0xBC octet and check if the upper 8**emLen* – *emBits* bits of the leftmost octet are zero. If either test fails, the signature is invalid. At this point, we extract *maskedDB* and *H* from *EM*, re-compute *dbMask*, and decode *maskedDB* to *DB*. *DB* should then contain the original *PS* zero octets (all *emLen* – *sLen* – *hLen* – 2 of them), the 0x01 octet, and the *salt*. From the salt, we can re-compute *H* and compare it against the *H* we extracted from *EM*. If they match, the signature is valid; otherwise, it is not.

SECURITY

It is very important that you ensure the decoded strings after performing the RSA primitive (RSADP or RSAVP1) are the size of the modulus and not shorter. The PSS and OAEP algorithms assume the decoded values will be the size of the modulus and will not work correctly otherwise. A good first test after decoding is to check the length and abort if it is incorrect.

PKCS #1 Key Format

PKCS #1 specifies two key formats for RSA keys; one is meant for public keys, and the other is meant for private keys. The public key format is the following.

```
RSAPublicKey ::= SEQUENCE {
   modulus            INTEGER, – n
   publicExponent     INTEGER, – e
}
```

While the private key format is the following.

```
RSAPrivateKey ::= SEQUENCE {
   version            Version,
   modulus            INTEGER, – n
   publicExponent     INTEGER, – e
   privateExponent    INTEGER, – d
   prime1             INTEGER, – p
   prime2             INTEGER, – q
   exponent1          INTEGER, – d mod (p - 1)
   exponent2          INTEGER, – d mod (q - 1)
   coefficient        INTEGER, – (1/q) mod p
   otherPrimeInfos    OtherPrimeInfos OPTIONAL
}
Version ::= INTEGER { two-prime(0), multi(1) }
OtherPrimeInfos ::= SEQUENCE SIZE(1..MAX) OF OtherPrimeInfo
OtherPrimeInfo ::= SEQUENCE {
    prime             INTEGER, – ri
    exponent          INTEGER, – di, d mod prime
     coefficient      INTEGER, – ti
}
```

The private key stores the CRT information used to speed up private key operations. It is not optional that it be stored in the SEQUENCE, but it is optional that you use it. The format also allows for what is known as *multi-prime* RSA by specifying a Version of 1 and providing OtherPrimeInfos. This is optional to support and generally is a good idea to ignore, as it is covered by a patent owned by HP.

RSA Security

The security of RSA depends mostly on the inability to factor the modulus into the two primes it is made of. If an attacker could factor the modulus, he could compute the private exponent and abuse the system. To this end, extensive research has been undertaken in the field of algebraic number theory to discover factoring algorithms. The first useful algorithm was the quadratic sieve invented in 1981 by Carl Pomerance.

The quadratic sieve had a running time of $O(\exp(\text{sqrt}(\log n \log \log n)))$ for an integer n. For example, to factor a 1024-bit number, the expected average runtime would be on the order of 2^{98} operations. The quadratic sieve works by gathering relations of the form $X^2 = Y \pmod{n}$ and then factoring Y using a small set of primes known as a factor bound. These relations are gathered and analyzed by looking for combinations of relations such that their product forms a square on both sides; for instance, if $X_1^2 * X_2^2 = Y_1Y_2 \pmod{n}$ and Y_1Y_2 is a square, then the pair can be used to attempt a factorization. If we let $P = X_1^2 * X_2^2$ and $Q = Y_1Y_2$, then if $x_1 * x_2$ is not congruent to sqrt(Q) modulo n, we can factor n. If $P = Q \pmod{n}$, then $P - Q = 0 \pmod{n}$, and since P and Q are squares, it may be possible to factor n with a difference of squares.

A newer factoring algorithm known as the Number Field Sieve attempts to construct the same relations but in a much different fashion. It was originally meant for numbers of a specific (non-RSA) form, and was later improved to become the generalized number field sieve. It has a running time of $O(\exp(64/9 * \log(n))^{1/3} * \log(\log(n))^{2/3}))$. For example, to factor a 1024-bit number the expected average runtime would be on the order of 2^{86} operations (Table 9.1).

Table 9.1 RSA Key Strength

RSA Modulus Size (bits)	Security from Factoring (bits)
1024	86
1536	103
1792	110
2048	116
2560	128
3072	138
4096	156

This implies if you want your attacker to expend at least 2^{112} work breaking your key, you must use at least a 2048-bit RSA modulus. In practice, once you pass the 2048-bit mark the algorithm becomes very inefficient, often impossible to implement in embedded systems. There is also no practical RSA key size that will match AES-256 or SHA-512 in terms of bit strength. To obtain 256 bits of security from RSA, you would require a 13,500-bit RSA modulus, which would tax even the fastest desktop processor. One of the big obstacles in making factoring more practical is the size requirements. In general, the number field sieve requires the square root of the work effort in storage to work. For example, a table with 2^{43} elements would be required to factor a 1024-bit composite. If every element were a single bit, that would be a terabyte of storage, which may not seem like a lot until you realize that for this algorithm to work efficiently, it would have to be in random access memory, not fixed disk storage.

In practice, RSA-1024 is still *safe* to use today, but new applications should really be looking at using at least RSA-1536 or higher—especially where a key may be used for several years to come. Many in the industry simply set the minimum to 2048-bit keys as a way to avoid eventual upgrades in the near future.

RSA References

There are many methods to make the modular exponentiation step faster. The first thing the reader will want to seek out is the use of the *Chinese Remainder Theorem* (CRT), which allows the private key operation to be split into two half-size modular exponentiations. The PKCS #1 standard explains how to achieve CRT exponentiation with the RSADP and RSAVP1 primitives.

After that optimization, the next optimization is to choose a suitable exponentiation algorithm. The most basic of all is the *square and multiply* (Figure 7.1, page 192 of *BigNum Math*), which uses the least amount of memory but takes nearly the maximum amount of time. Technically, square and multiply is vulnerable to timing attacks, as it only multiplies when there is a matching bit in the exponent. A *Montgomery Powering Ladder* approach can remove that vulnerability, but is also the slowest possible exponentiation algorithm (Marc Joye and Sung-Ming Yen, "The Montgomery Powering Ladder," Hardware and Embedded Systems, CHES 2002, vol. 2523 of Lecture Notes in Computer Science, pp. 291–302, Springer-Verlag, 2003). Unlike *blinded exponentiation* techniques, the Montgomery powering ladder is fully deterministic and does not require entropy at runtime to perform the operation. The ladder is given by the algorithm in Figure 9.8.

Figure 9.8 The Montgomery Powering Ladder

```
Input:
    $g$:    The base
    $k$:    The exponent, bits denoted as $(k_{t-1}, ..., k_0)_2$
Output:
    $y$:    $y = g^k$

 1. $R_0 = 1, R_1 = g$
 2. for $j = t - 1$ downto 0 do
     1. if $(k_j = 0)$ then
     $R_1 = R_0 * R_1, R_0 = R_0^2$
```

```
  else
    $R_0 = R_0 * R_1, R_1 = R_1^2$
  3. Return $R_0$
```

This algorithm performs the same operations, at least at the high level, regardless of the bits of the exponent—which can be essential for various threat models given adversaries who can gather side channel information. It can be sped up by realizing that the two operations per iteration can be performed in parallel. This allows hardware implementations to reclaim some speed at a significant cost in area. This observation pays off better with elliptic curve algorithms, as we will see shortly, as the numbers are smaller, and as a consequence, the hardware can be smaller as well.

A simple blinding technique is to pre-compute g^{-r} for a random *r*, and compute g^k as $g^{k+r} * g^{-r}$. As long as you never re-use *r*, it is random, and it is the same length as *k* the technique blinds timing information that would have been leaked by *k*.

If memory is abundant, *windowed exponentiation* is a possible alternative (Figure 7.4, page 196 of *BigNum Math*). It allows the combination of several multiplications into one step at an expense of pre-compute time and storage. It can also be made to run in fixed time by careful placement of dummy operations.

Elliptic Curve Cryptography

Over the last two decades, a different form of public key cryptography has been gaining ground—*elliptic curve cryptography*, or ECC. ECC encompasses an often hard to comprehend set of algebraic operations to perform cryptography that is faster than RSA, and more secure.

ECC makes use of the mathematical construct known as an elliptic curve to construct a *trapdoor* in a manner not vulnerable to sub-exponential attacks (as of yet). This means that every bit used for ECC goes toward the end security of the primitives more than it would with RSA. For this reason, the numbers (or polynomials) used in ECC are smaller than those in RSA. This in turn allows the integer operations to be faster and use less memory.

For the purposes of this text, we will discuss what are known as the *prime field* ECC curves as specified by NIST and used in the NSA Suite B protocol. NIST also specifies a set of *binary field* ECC curves, but are less attractive for software. In either case, an excellent resource on the matter is the text *Guide to Elliptic Curve Cryptography* (Darrel Hankerson, Alfred Menezes, Scott Vanstone, *Guide to Elliptic Curve Cryptography*, Springer, 2004). That text covers the software implementation of ECC math for both binary and prime field curves in great depth. Readers are strongly encouraged to seek out that text if they are interested in implementing ECC, as it will give them pointers not included here toward high optimization and implementation ease.

What Are Elliptic Curves?

An elliptic curve is typically a two-space graph defined by the square roots of a cubic equation. For instance, $y^2 = x^3 - x$ is an elliptic curve over the set of real numbers. Elliptic curves can also be defined over other fields such as the field of integers modulo a prime, denoted as *GF(p)*, and over the extension field of various bases, such as *GF(2^k)* (this is known as binary field ECC).

Given an elliptic curve such as $E_p : y^2 = x^3 - 3x + b$ (typical prime field definition) defined in a finite field modulo *p*, we can compute *points* on the curve. A point is simply a pair (*x*, *y*) that satisfies the equation of the curve. Since there is a finite number of units in the field, there must be a finite number of unique points on the curve. This number is known as the *order* of the curve. For various cryptographic reasons, we desire that the order be a large prime, or have a large prime as one of its factors.

In the case of the NIST curves, they specify five elliptic curves with fields of sizes 192, 224, 256, 384, and 521 bits in length. All five of the curves have orders that are themselves large primes. They all follow the definition of E_p listed earlier, except they have unique *b* values. The fact that they have the same form of equation for the curve is important, as it allows us to use the same basic mathematics to work with all the curves. The only difference is that the modulus and the size of the numbers change (it turns out that you do not need *b* to perform ECC operations).

From our elliptic curve, we can construct an algebra with operations known as *point addition*, *point doubling*, and *point multiplication*. These operations allow us to then create a trapdoor function useful for both DSA signatures and Diffie-Hellman-based encryption.

Elliptic Curve Algebra

Elliptic curves possess an algebra that allows the manipulation of points along the curve in controlled manners. Point addition takes two points on the curve and constructs another. If we subtract one of the original points from the sum, we will have computed the other original point. Point doubling takes a single point and computes what would amount to the addition of the point to itself. Finally, point multiplication combines the first two operations and allows us to multiply a scalar integer against a point. This last operation is what creates the trapdoor, as we shall see.

Point Addition

Point addition is defined as computing the slope through two points and finding where it strikes the curve when starting from either point. This third point is negated by negating the *y* portion of the point to compute the addition. Given $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, two different points on the curve, we can compute the point addition with the following.

$$P + Q = (x_3, y_3)$$

$$x_3 = ((y_2 - y_1) / (x_2 - x_1))^2 - x_1 - x_2$$

$$y_3 = ((y_2 - y_1) / (x_2 - x_1)) * (x_1 - x_3) - y_1$$

All of these operations are performed in the finite field; that is, modulo some prime. The equations are only defined for the case where P and Q do not lie at the same *x* co-ordinate.

Point Doubling

Point doubling is defined as computing the tangent of a point on the curve and finding where it strikes the curve. There should be only one unique answer, and it is the double of the point. Point doubling can be thought of as adding a point to itself. By using the tangent instead of the slope, we can obtain well-defined behavior. Given $P = (x_1, y_1)$, the point double is computed as follows.

$$2P = (x_3, y_3)$$

$$x_3 = ((3x_1^2 + a) / (2y_1))^2 - 2x_1$$

$$y_3 = ((3x_1 + a) / (2y_1)) * (x_1 - x_3) - y_1$$

Again, all of these operations are performed in a finite field. The *a* value comes from the definition of the curve, and in the case of the NIST curves is equal to –3.

Point Multiplication

Point multiplication is defined as adding a point to itself a set number of times, typically denoted as *kP* where *k* is the scalar number of times we wish to add *P* to itself. For instance, if we wrote 3*P*, that literally is equivalent to *P* + *P* + *P*, or specifically, a point double and addition.

TIP

It is very easy to be confused by elliptic curve mathematics at first. The notation is typically very new for most developers and the operations required are even stranger. To this end, most authors of elliptic curve material try to remain somewhat consistent in their notation.

When someone reads "*kG*", the lowercase letter is almost always the scalar and the uppercase letter the point on the curve. The letter *k* is occasionally reserved for *random* scalars; for instance, as required by the Diffie-Hellman protocol. The letter *G* is occasionally reserved for the standard *base point* on the curve.

The *order* of the curve plays an important role in the use of point multiplication. The order specifies the maximum number of unique points that are possible given an ideal fixed point G on the curve. That is, if we cycle through all possible values of *k*, we will hit all of the points. In the case of the NIST curves, the order is prime; this has a neat side effect that all points on the curve have maximal order.

Elliptic Curve Cryptosystems

To construct a public key cryptosystem from elliptic curves, we need a manner of creating a public and private key. For this, we use the point multiplier and a standards specified *base point* that lies on the curve and is shared among all users. NIST has provided one base point for each of the five different curves they support (on the prime field side of ECC).

Elliptic Curve Parameters

NIST specifies five prime field curves for use with encryption and signature algorithms. A PDF copy of the standard is available through NIST and is very handy to have around (NIST recommended curves: http://csrc.nist.gov/CryptoToolkit/dss/ecdsa/NISTReCur.pdf).

NIST specifies five curves with the sizes 192, 224, 256, 384, and 521 bits, respectively. When we say *size*, we really mean the order of the curve. For instance, with the 192-bit curve (also known as P-192), the order is a 192-bit number. For any curve over the prime fields, four parameters describe the curve. First is the modulus, *p*, which defines the field in which the curve lies. Next is the *b* parameter, which defines the curve in the finite field. Finally is the order of the curve *n* and the base point, *G*, on the curve with the specified order. The tuple of (*p*, *n*, *b*, *G*) is all a developer requires to implement an elliptic curve cryptosystem.

For completeness, following is the P-192 curve settings so you can see what they look like. We will not list all five of them, since printing a page full of random-looking numbers is not useful. Instead, consider reading the PDF that NIST provides, or look at LibTomCrypt in the file *src/pk/ecc/ecc.c*. That file has all five curves in an easy to steal format to include in other applications (LibTomCrypt is public domain). The following is a snippet from that file.

```
#ifdef ECC192
{
        24,
        "ECC-192",
        "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFF",
        "64210519E59C80E70FA7E9AB72243049FEB8DEECC146B9B1",
        "FFFFFFFFFFFFFFFFFFFFFFFF99DEF836146BC9B1B4D22831",
        "188DA80EB03090F67CBF20EB43A18800F4FF0AFD82FF1012",
        "7192B95FFC8DA78631011ED6B24CDD573F977A11E794811",
},
#endif
```

The first line of the structure describes the field size in octets. The second line is a nice string literal name of the curve (for diagnostic support). Next are *p*, *b*, *r*, and finally the base point (*x*, *y*). All the numbers are stored in hexadecimal format.

Key Generation

The algorithm in Figure 9.9 describes the key generation process.

Figure 9.9 ECC Key Generation

```
Input:
    $G$:    Standard base point
    $n$:    Order of the curve (NIST provides this)
Output:
    $Y$:    Public key (ECC point)
    $x$:    Private key (integer)

1. Pick a random integer $x$ in the range of $1 < x < n - 1$
2. Compute $Y = xG$
3. Return $(x, Y)$
```

We can now give out Y to anyone we want, and the recipient can use this to encrypt messages for us, or verify messages we sign. The format—that is, byte format of how we store this public key—is specified as part of ANSI X9.63 (Section 4.3.6); however, there are better formats. For the benefit of the reader, we will show the ANSI method.

ANSI X9.63 Key Storage

An elliptic curve point can be represented in one of three forms in the ANSI standard.

- Compressed
- Uncompressed
- Hybrid

When storing $P = (x, y)$, first convert x to an octet string X. In the case of prime curves, this means store the number as a big endian array of octets.

1. If using the compressed form, then
    1. Compute $t = \text{compress}(x, y)$
    2. The output is **0x02** || X if t is 0; otherwise, the output is **0x03** || X
2. If using the uncompressed form
    1. Convert y to an octet string Y
    2. The output is **0x04** || X || Y
3. If using the hybrid form

1. Convert y to an octet string Y
2. Compute t = compress(x, y)
3. The output is **0x04** || X || Y if t is 0; otherwise, the output is **0x05** || X || Y

The compression function is computing which of the square roots y is. Taking from ANSI X9.63 (Section 4.2.1), we can compress a point (x, y) by taking the rightmost bit of y and returning that as the output. Decompressing the point is fairly simple, given x and the compressed bit t, the decompression is:

1. Compute a, the square root of $x^3 - 3x + b \pmod{p}$

If the rightmost bit of a is equal to t, then return a; otherwise, return $p - a$

NOTE

Readers should be careful around point compression. Certicom holds U.S. patent #6,141,420, which describes the method of compressing a full point down to the *x* co-ordinate and an additional bit. For this reason alone, it is usually a good idea to avoid point compression.

Generally, ECC keys are so small that point compression is not required. However, if you still want to compress your points, there is a clever trick you can use that seems to avoid the Certicom patent (Figure 9.10).

Figure 9.10 Elliptic Curve Key Generation with Point Compression

```
Input:
    G:      Standard base point
    p:      Modulus
    n:      Order of curve
Output:
    k:      Secret key (integer)
    x:      Public key (only x co-ordinate of public key, integer)

1. Pick a random integer k in the range of $1 < k < n - 1$
2. Compute $Y = kG = (x, y)$
3. Compute $z = sqrt(x3 - 3x + b) \pmod{p}$
4. If z does not equal y, goto step 1
5. return (k, x)
```

This method of key generation is roughly twice as slow as the normal key generation, but generates public keys you can compress and avoids the Certicom patent. Since the compressed bit will *always* be 0, there is no requirement to transmit it. The user only has to give out his *x* co-ordinate of his public key and he is set. What is more, this key generation produces keys compatible with other algorithms such as EC-DH and EC-DSA.

Elliptic Curve Encryption

Encryption with elliptic curves uses a variation of the ElGamal algorithm, as specified in section 5.8 of ANSI X9.63. The algorithm is as shown in Figure 9.11).

Figure 9.11 Elliptic Curve Encryption

```
Input:
    Y:            The recipients public key
    EncData:      Plaintext of length encdatalen
    mackeylen:    The length of the desired MAC key
Output:
    c:            The ciphertext

 1. Generate a random key $Q = (k_q, (x_q, y_q))$
 2. Compute the shared secret, z, the x co-ordinate of $k_qY$
 3. Use the key derivation (ANSI X9.63 Section 5.6.3, equivalent to PKCS #1 MGF)
    to transform z into string KeyData of length encdatalen + mackeylen bits
 4. Split KeyData into two strings, EncKey and MacKey of encdatalen and mackeylen
    bits in length, respectively
 5. MaskedEncData = EncData XOR EncKey
 6. MacTAG = MAC(MacKey, MaskedEncData) using an ANSI approved MAC (such
    as X9.71, which is HMAC)
 7. Store the public key $(x_q, y_q)$ using the key storage algorithm, call it QE
 8. Return c = QE || MaskedEncData || MacTag
```

ANSI allows for additional shared data in the encryption, and it is optional. The key derivation function is, without this optional shared data, equivalent to the recommended mask generation function (MGF) from the PKCS #1 standard. The MAC required (step 6) must be an ANSI approved MAC with a security level of at least 80 bits. They recommend X9.71, which is the ANSI standard for HMAC. HMAC-SHA1 would provide the minimum security required for X9.63.

Decryption is performed by computing z as the x co-ordinate of $k(x_q, y_q)$, where k is the recipient's private key. From z, the recipient can generate the encryption and MAC keys, verify the MAC, and decrypt the ciphertext.

Elliptic Curve Signatures

Elliptic curve signatures use an algorithm known as EC-DSA, which is derived from the NIST standard digital signature algorithm (DSA, FIPS-186-2), and from ElGamal. Signatures are in section 7.3 of ANSI X9.62 (Figure 9.12).

Figure 9.12 Elliptic Curve Signatures

```
Input:
    $k$:       Private key (integer)
    $n$:       Order of the curve
    $M$:       Message to be signed
Output:
    $(r, s)$:  The signature

1. Generate a random key $Q = (k_q, (x_q, y_q))$.
2. Convert $x_q$ to an integer j. This step is omitted for prime curves since xq is already an integer.
3. $r = j \bmod n$, if $r = 0$, go to step 1.
4. $H = hash(M)$
5. Truncate H to the leftmost $ceil(\log_2 n)$ bits. For instance, with P-192, truncate H to 192 bits.
6. Convert H to an integer e by loading it in big endian fashion.
7. Compute $s = k_q^{-1}(e + kr) \bmod n$; if $s = 0$, go to step 1.
8. Return $(r, s)$.
```

The signature is actually two integers of the size of the order. For instance, with P-192, the signature would be roughly 384 bits in size. ANSI X9.62 specifies that the signatures be stored with the following ASN.1 SEQUENCE when transporting them (Section E.8):

```
ECDSA-Sig-Value ::= SEQUENCE {
   r   INTEGER,
   s   INTEGER
}
```

which is a welcome change from the X9.63 key storage format given that it is *uniquely decodable*. Verification is given as shown in Figure 9.13 (ANSI X9.62 Section 7.4).

Figure 9.13 Elliptic Curve Signature Verification

```
Input:
    $Y$:       Signers public key
    $G$:       Base point for the curve
    $M$:       Signed message
    $n$:       Order of the curve
    $(r, s)$:  The signature
Output:
    Validity of signature

 1. If $r$ and $s$ are both not in the interval $[1, n - 1]$, then return invalid.
 2. $H = \text{hash}(M)$
 3. Truncate $H$ to the leftmost ceil($\log_2 n$) bits. For instance, with P-192, truncate $H$ to
    192 bits.
 4. Convert $H$ to an integer $e$ by loading it in big endian fashion.
 5. $u_1 = e/s \bmod n$
 6. $u_2 = r/s \bmod n$
 7. $R = u_1G + u_2Y = (x_r, y_r)$
 8. Convert $x_r$ to an integer $j$. This step is omitted for prime curves since $x_r$ is already
    an integer.
 9. $r' = j \bmod n$
10. If $r$ equals $r'$ return valid; otherwise, return invalid.
```

For clarification, step 7 performs a point addition of the two point multiplications.

TIP

There are two methods to speed up verification. The first is to compute *R* using what is known as "Shamir's Trick." You can find a description on page 109 of the "Guide to Elliptic Curve Cryptography," algorithm 3.48. This allows you to adhere to the standards but compute the *R* point in much less time.

If you will be performing verifications on a resource starved platform and can tolerate a slight deviation from the standard, you can create the signatures as (*r*, 1/*s*) instead. This will omit the slow modular inversion required during verification by making signature generation slower. This is *not* ANSI compliant, however.

Elliptic Curve Performance

So far, we have been only looking at the very basics of elliptic curve operations such as point addition and doubling using affine co-ordinates. If we implemented a cryptosystem using the functions as so far described, we would have a very *slow* cryptosystem. The reason being that modular inversions—that is, computing $1/x$ modulo a prime—is a very slow operation, often comparable to dozens of field multiplications.

To solve this problem, various projective systems have been described that map the two-space operations to three-space or more. The goal is to trade off the inversions with field multiplications. Ideally, if you minimize the field multiplications, you can break even and achieve the point addition and doubling faster. In turn, this makes point multiplication faster, and finally the dependent operations such as encryption and signatures.

Another important optimization is *how* we perform the point multiplication itself. If we simply added the point to itself the requisite number of times, it would take forever to perform a point multiplication. There are various windowed and fixed point algorithms useful for this as well.

Jacobian Projective Points

In the Jacobian projective co-ordinate system, we represent a standard point with three variables. The mapping is fairly simple going into projective format. The forward mapping turns (x, y) into (x, y, z), with $z = 1$ to start with. To map back from projective, we compute $(x/z^2, y/z^3, 1)$, where the division is performed in the finite field. Typically, one would compute $1/z$ first, and from this derive $1/z^2$ and $1/z^3$, using only one inversion.

For example, point doubling then becomes the algorithm in Figure 9.14 (taken from page 91, "Guide to Elliptic Curve Cryptography," algorithm 3.21). Point addition is on the same page, and listed as algorithm 3.22.

Figure 9.14 Jacobian Projective Point Doubling

```
Input:
    P:                $(x_1, y_1, z_1)$ in Jacobian co-ordinates
Output:
    $(x_3, y_3, z_3)$:    The value of 2P

  1. $t_1 = z_1^2$
  2. $t_2 = z_1 - t_1$
  3. $t_1 = x_1 + t_1$
  4. $t_2 = t_2 * t_1$
  5. $t_2 = 3 * t_2$
  6. $y_3 = 2 * y_1$
```

```
7.  $z_3 = y_3 * z_1$
8.  $y_3 = y_3^2$
9.  $t_3 = y_3 * x_1$
10. $y_3 = y_3^2$
11. $y_3 = y_3/2$
12. $x_3 = t_2^2$
13. $t_1 = 2 * t_3$
14. $x_3 = x_3 - t_1$
15. $t_1 = t_3 - x_3$
16. $t_1 = t_1 * t_2$
17. $y_3 = t_1 - y_3$
18. Return $(x_3, y_3, z_3)$
```

The division by two require the value to be even; otherwise, you must first add the modulus (forcing it to be even) and then perform the division (usually by performing the right shift). As we can see, this doubling has a fair bit more steps than the original point double algorithm. In fact, there are more field multiplications as well. However, there are no inversions, and most of the operations are trivial additions or subtractions.

The point addition algorithm assumes the second input is in affine co-ordinates ($z = 1$). This is important to keep in mind when performing the point multiplication. If you would rather not map to affine, you can use the algorithm from section 4.2 of the "A Practical Implementation of Elliptic Curve Cryptosystems over GF(p) on a 16-bit Microprocessor," by T. Hasegawa, J. Nakajima, and M. Matsui. It describes both the Jacobian-Jacobian and the Jacobian-affine addition techniques. Mixed additions are slightly faster and consume less memory (as your 2nd operand only needs two co-ordinates of storage), but cost more to set up initially.

Point Multiplication Algorithms

There are a variety of useful manners in which one could accomplish point multiplication, the most basic being the double and add method. It is essentially the square and multiply technique for exponentiation converted to point multiplication (Algorithm 3.27 of the Guide, page 97).

That method works, but is not typically used because it is too slow. Instead, most developers opt instead for a sliding window approach (Algorithm 3.38 of the Guide, page 101). It has fewer point additions and consumes little memory. This algorithm is similar to the exponentiation trick listed in Figure 7.7, page 198 of *BigNum Math*.

Another approach for when the point is known (or fixed) is the fixed point technique. A variation of this algorithm is described as algorithm 3.41 on page 104 of the Guide. It was also recently added to LibTomCrypt, but as a slightly different variation. This technique leads to very fast point multiplication but at a cost in memory. It is also only useful for encryption, signature generation, and verification. It cannot be used for decryption, as the points are always random. It is therefore important to always have a fast random point multiplier (when decryption is required) before spending resources on a fast fixed point multiplier.

The Guide describes various methods based on nonadjacent forms (NAF) encodings (Algorithm 3.31, page 99), which seem ideal for limited memory platforms. In practice, they often are not faster than the normal windowed approach. It is worth the investigation, though, if memory is a problem.

Putting It All Together

There are two competent public key algorithms in the standards bodies: RSA, which is older and more traditionally used throughout the industry, and ECC, which is slightly newer, more efficient, and making quite a few new appearances in the wild.

The choice between RSA or ECC is often easy given no logistical restrictions such as standards compliance.

ECC versus RSA

Speed

ECC with Jacobian co-ordinates and a proper point multiplier leads to much faster private key operations when compared to RSA. The fact that RSA uses larger integers and must perform a slow modular exponentiation make any private key operation very slow. ECC, by using smaller integers, can perform its primitive operations such as field multiplications much faster.

Where RSA wins over ECC is with public key operations (such as encryption and verification). Since the public exponent, *e*, is typically small, the modular exponentiation can be computed very efficiently. For instance, with $e = 65537$, only 16 field squarings and 1 field multiplication are required to compute the power. Granted, the field is larger, but it still wins out.

In Table 9.2, we see that RSA-1024 encryption on the AMD Opteron requires 131,176 cycles, while encryption with ECC-192 (the closest ECC match in terms of bit strength) requires two point multiplications and would consume at least 780,000 cycles. On the other hand, signature generation would require at least 1.2 million cycles with RSA-1024, and only 390,000 cycles with ECC-192.

Table 9.2 Public Key Performance (cycles per operation)

RSA Tests (Encrypt/Decrypt)	**AMD Opteron**	**Intel Pentium 4**	**Intel Pentium M**	**Intel Core 2 Duo**
RSA-1024	131,176/1,243,530	694,262/7,875,960	467,330/4,851,959	146,020/1,549,781
RSA-1536	225,191/3,562,010	1,090,842/13,470,711	933,843/13,816,129	254,741/4,314,161
RSA-2048	348,006/6,144,918	2,143,433/47,124,161	1,495,344/29,705,622	396,468/8,294,762
ECC Tests (Fixed Point Multiplication)				
ECC-192	390,615	1,470,551	989,938	367,879
ECC-224	468,685	1,952,344	1,280,312	454,996
ECC-256	583,723	2,464,947	1,547,887	586,797
ECC-384	1,123,152	5,916,616	3,572,755	1,240,034
ECC-521	1,986,757	12,877,997	7,743,301	2,304,239

Size

ECC uses smaller numbers (the reason for which will become apparent shortly), which allows it to use less memory and storage when transmitting data.

For example, while aiming for 80 bits of security, ECC would only have to use a 160-bit curve, whereas RSA would have to use a 1024-bit modulus. This means our ECC public key is only 320 bits in size. It also means our EC-DSA signature is only 320 bits, compared to the 1024 bits required for RSA-PSS.

It also pays off during the implementation phase. In the same space required for a single RSA sized integer, one could store roughly eight or more ECC sized parameters. We saw from the Jacobian point doubling algorithm that we require 11 integers (one for the modulus, one for overflow, three for the input, three for the output, and three temporaries). This would mean that for ECC-192, we would require 264 bytes of storage for that operation, and 360 for point addition. RSA-1024 with a simple square and multiply would require at least four integers, requiring 512 bytes of storage. (In practice, to get any sort of speed out of RSA, you would require a few more integers. The size estimates are just that—estimates.)

Security

The security of RSA for the most part rests on the difficulty of factoring. While this is not entirely true, it is the current only line of attack against RSA public keys that works. On the other hand, ECC is not vulnerable to factoring or other attacks in the sub-exponential range.

The only known attack against the current elliptic curves chosen by NIST is a exponential cycle finding attack known as Pollard-Rho; that is, assuming the order of the curve is prime. If the order of a curve is n, then it would require roughly $O(n^{1/2})$ operations to break the ECC public key and find the secret key. For example, with P-192, an attacker would have to expend roughly 2^{96} time breaking the key on average. This means that P-192 is actually about 1024 times harder to break than RSA-1024 when mounting offline attacks on the public key. In practice, RSA-1024 pairs with ECC P-192, RSA-2048 with ECC P-224, and RSA-3072 with ECC P-256.

This is where ECC really shines. At the 112 bits of security level (RSA-2048), the ECC operations are not much harder than from P-192. On the other hand, RSA-2048 is much harder to work with. The field operations are four times slower and there are twice as many of them. On the AMD Opteron when moving from RSA-1024 to RSA-2048, the private key operations become roughly 600-percent slower, whereas when moving from ECC P-192 to P-224, the point multiplication only becomes 20-percent slower.

Standards

Both IEEE and ANSI specify standards for elliptic curve cryptography. The IEEE P1363a is the current IEEE standard, while X9.62 and X9.63 are the current ANSI standards. NIST specifies the use of the ANSI X9.62 EC-DSA as part of the digital signature standard (DSS, FIPS 180-2).

While ANSI specifies RSA standards (X9.31), the PKCS #1 standard is the reference most other standards default to. In particular, quite a few older systems use v1.5 of PKCS #1, which should be avoided. This is not because v1.5 of PKCS #1 is entirely insecure; it is simply less optimal from a security standpoint than v2.1. New applications should avoid both X9.31 (which is very similar to v1.5 of PKCS #1) and v1.5 of PKCS #1 in favor of v2.1.

References

Text References

When implementing RSA, the PKCS #1 standard (v2.1) is by far the most important resource. It describes the OAEP and PSS padding techniques, CRT exponentiation, and the ASN.1 definitions required for interoperability.

For FIPS 180-2 DSS compliance, the ANSI X9.31 standard must be used.

When implementing ECC, the ANSI X9.62 standard specifies EC-DSA and is used by FIPS 180-2 DSS. The ANSI X9.63 standard specifies ECC encryption, key storage, and a few authentication schemes (a couple of which have patents). Currently, NIST is working on SP800-56A, which specifies ANSI X9.42 using discrete logarithm systems (like ElGamal), and X9.63 using ECC. An additional specification SP800-56B specifies ANSI X9.44 (RSA encryption). It is more likely that SP800-56A will become more popular in the future, as it uses ECC as oppose to RSA.

A good reference for the large integer operations is *BigNum Math* (Tom St Denis, Greg Rose, *BigNum Math: Implementing Cryptographic Multiple Precision Arithmetic*, Syngress, 2006, ISBN 1-59749-112-8), which discusses the creation of a portable and efficient multiple precision large integer operations. That book uses both pseudo code and real production C source code to demonstrate the math to the reader. It is by no means a hard read and is well suited for the target audience of this text.

For implementing ECC math, the reader is strongly encouraged to obtain a copy of the *Guide to Elliptic Curve Cryptography* (D. Hankerson, A. Menezes, S. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer, 2004, ISBN 0-387-95273-X). That text discusses both binary and prime field ECC operations, and detailed explanations of point operations and multiplication. It spends less time on the field operations (such as integer multiplication), but a good deal of time on the binary field operations (which are typically harder to understand at first for most). It is an invaluable resource on point multiplication techniques.

Source Code References

The LibTomCrypt package provides PKCS #1 compliant RSA and ANSI X9.62 compliant EC-DSA. It uses a modified key derivation function and key storage that is incompatible with X9.63. LibTomCrypt employs the use of CRT exponentiation for RSA. It uses Jacobian-affine co-ordinates for the ECC math. It provides both a sliding window random point multiplier and a fixed point multiplier. Since the code is well commented and public domain, it is a valuable source of implementation insight.

The Crypto++ package also provides PKCS #1, and ANSI X9.62 and X9.63 support. Its code base is larger than LibTomCrypt and is not public domain. It includes a variety of patented algorithms as well, such as EC-MQV. The reader should be careful when exploring this code base, as it is not all free and clear to use.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to www.syngress.com/solutions and click on the "Ask the Author" form.

Q: What are public key algorithms?

A: Public key algorithms are algorithms where a key can be split into a public and private portion. This then allows the holder of the private key to perform operations that are not possible with the public key. For example, the private key can sign messages while the public key can only verify them.

Q: What else are they good for?

A: Public key cryptography solves, or at least, offers a solution to, the key distribution problem that plagues symmetric algorithms. For example, one may encrypt a message with a public key, and only the private key can decrypt it. In this way, without first securing a shared secret, we can still transmit encrypted data.

Q: What public key algorithms are out there?

A: There are many, from Diffie-Hellman, ElGamal, RSA, ECC, and NTRU. Most standards focus on RSA, and recently more often on ECC. ElGamal is still considered secure; it is merely too inefficient to contend with RSA (which is now patent free).

Q: What standards are out there?

A: ANSI X9.62 specifies ECC signatures, while ANSI X9.63 specifies ECC encryption. PKCS #1 specifies RSA encryption and signatures. ANSI X9.31 specifies RSA signatures, and X9.42 specifies RSA encryption. The FIPS 180-2 specifies X9.62 for ECC and X9.31 for RSA, along with its own DSA (over integers).

Q: What is good/bad about RSA?

A: Conceptually, RSA is very simple. It is fairly easy to develop against, and PKCS #1 is not too painful to digest. RSA public key operations (encrypt and verify) are typically very fast. On the other hand, RSA private key operations are very slow. RSA also requires large keys for typically demanded bit security. For example, for 112 bits of security, one must use a 2048-bit RSA key. This can make RSA very inefficient very quickly.

Q: What is good/bad about ECC?

A: The security of ECC is much better than that of RSA. With smaller parameters, ECC provides more security than RSA. This means that ECC math can be both memory efficient and fast. ECC public key operations are slower than their private key counterparts, but are still typically faster than RSA at the nontrivial bit security levels. ECC is also more complicated than RSA to implement and more prone to implementation errors. It's conceptually hard to get going very fast and requires more time to complete than RSA.

Q: Are there libraries I can look at?

A: Both LibTomCrypt and Crypto++ implement public key algorithms. The latter implements many more algorithms than just RSA and ECC. LibTomCrypt does not fully adhere to the X9.63 standard, as it deviates to solve a few problems X9.63 has. It does, however, implement X9.62, and PKCS #1 for RSA.

Q: Why should I buy *Guide to Elliptic Curve Cryptography*, if I just bought this book?

A: The Guide was written by some of the best minds in cryptography, from Certicom, and are essentially the professionals in elliptic curve mathematics. The book also fully covers the NIST suite of ECC curves with algorithms that are efficient, patent free, and easy to implement. Duplicating the content of that book here seems like a waste of time and does not serve the audience well. Trust us, it's worth the purchase.

Index