In this chapter we will look at choosing and optimizing cryptographic algorithms, particularly for resource-constrained systems, such as embedded systems. We will look at various strategies for choosing algorithms for specific applications, and look at some specific algorithms as well as some strategies to avoid some of the more expensive cryptographic operations without compromising security.

First of all, we will look at whether cryptography is even necessary. Some applications can actually get away without using traditional cryptography. These applications utilize other mechanisms, such as hashing algorithms, in order to provide some assurance about data. The big advantage here is that hashing algorithms are typically many orders of magnitude faster than symmetric or public-key cryptography.

One classification that will help us make the decision is whether or not we care about eavesdropping. If we only care that the data is reliable, and do not care who gets it, we can definitely avoid cryptography. We will look at these types of applications and come up with some other general rules for determining if an application requires the more expensive operations, or can get away with less.

We will look at cryptographic hashing, and discuss some optimizations and other tricks that can be used to speed up commonly used algorithms. Due to recent discoveries of weaknesses in the commonly used algorithms MD5 and SHA-1, we will also look at the future of hashing, and see if there is an alternative method, or new algorithms we can choose.

For those applications that require absolute secrecy, there are many options that will help the embedded developer meet the goals of performance, cost, and security. We will look at some specific
algorithms, such as DES/3DES, which is slow and obsolete but still necessary for some applications. AES, the replacement for DES, will also be covered, and specifically, some of the hardware options available. Additionally, we will look at RC4 and other ciphers that provide a marked performance advantage over the bulkier DES and AES, but are not as provably secure.

Finally, we will cover public-key algorithms, and specifically, RSA, by far the most commonly used public-key cipher. Unfortunately, RSA and other public-key algorithms are extremely slow, requiring hardware assistance on many platforms—not just small embedded systems. However, these algorithms are essential to certain protocols (primarily SSL and SSH), so we will look at ways of handling the computationally intense public-key operations, including software tricks and hardware assistance.

**Do We Need Cryptography?**

### 12.1 Do We Need Cryptography?

One of the first steps in building a secure embedded system is to see if cryptography is actually needed. Whenever security is discussed, many engineers will immediately think of cryptography as the solution, when in fact, many options may exist that do not strictly require cryptography. Sure, cryptography is an important part of securing applications, but it is not a security panacea, nor is it always necessary for building a secure system.

To many people, cryptography is a mysterious and confusing, almost magical area of computer science that is reserved for cloistered geniuses at top academic institutions and the US National Security Agency. The NSA is mysterious and secretive, and it is believed to employ some of the greatest cryptographers in the world, only adding to the idea that cryptography is an untouchable discipline.

There have been some attempts over the last couple of decades to bring cryptography out from behind its cloak of mystery, most notably with the publication of *Applied Cryptography* by renowned security expert Bruce Schneier. Schneier, more than any other individual, has brought cryptography into mainstream computer science by unveiling the techniques employed in most common cryptographic algorithms.

That being said, a stream of inaccurate portrayals of cryptography in Hollywood and television, combined with unrelenting advertising campaigns for various security products, have only served to keep those not involved directly in computer science in the dark. For this reason, it can be difficult to explain to managers or hardware engineers what exactly cryptography is, let alone convince them that it will not solve their security problems.

Given the confusion surrounding cryptography, it can be extremely difficult for a systems engineer to determine what type of cryptography, if any, is needed. The truth is that in some circumstances, no cryptography is needed at all. For instance, if you do not care about anyone seeing the information being transmitted, but rather you are only concerned that it is not tampered with in transit, you really do not need a complete cryptographic suite to suit your needs.

For example, let’s take a look at publicly available stock market reports. For many investors, keeping tabs on the latest price of a stock can mean thousands or millions of dollars being made or lost. The price of stock in a public company is readily available, but if someone were able to change that price
in transit during a trading session, that person would be able to wreak havoc upon unknowing investors. Obviously, there is a security risk for any system that transmits this data, but we do not need to hide it from anyone (unless your goal is to provide only your customers with the most up-to-date information, but that is another matter). Instead, we can just use a simple hash algorithm to verify the integrity of the data being transported. Not being a full-blown encryption scheme, the system's requirements can be lowered to support just the hashing (or those resources could be used for improving the performance of other parts of the system).

In order to see what type of security an application will need, we can divide applications into several distinct categories based upon the type of information each application deals with. After all, the purpose of computer security is to protect information, so this is a good place to start. The following categories should be sufficient for most applications, ordered from the lowest level of security to the highest level of security:

1. **No security required (may be optional)**—Applications such as streaming video, noncritical data monitoring, and applications without networking in a controlled environment.

2. **Low-level security (hashing only, plaintext data)**—Applications delivering publicly available information such as stock market data, data monitoring, networked applications in a controlled environment, and applications requiring an extra level of robustness without concern about eavesdropping. Another example of this would be the distribution of (usually open-source) source code or executables with an accompanying hash value for verifying integrity, usually an MD5 or SHA-1 hash of the files that is to be calculated by the end user and compared to the provided hash.

3. **Low-medium security (hashing plus authentication)**—General industrial control, some web sites, and internal corporate network communications.

4. **Medium-level security (cryptography such as RC4 with authentication, small key sizes)**—Applications dealing with general corporate information, important data monitoring, and industrial control in uncontrolled environments.

5. **Medium-high security (SSL with medium key sizes, RSA and AES, VPNs)**—Applications dealing with e-commerce transactions, important corporate information, and critical data monitoring.

6. **High-level security (SSL with maximum-sized keys, VPNs, guards with guns)**—Applications dealing with important financial information, noncritical military communications, medical data, and critical corporate information.

7. **Critical-level security (physical isolation, maximum-size keys, dedicated communications systems, one-time pads, guards with really big guns)**—Used to secure information such as nuclear launch codes, root Certificate Authority private keys.
(hopefully!), critical financial data, and critical military communications.

- Absolute security (one-time pads, possibly quantum cryptography, guards with an entire military behind them) used to secure information including Cold War communications between the United States and the Soviet Union regarding nuclear weapons, the existence of UFOs, the recipe for Coca-Cola, and the meaning of life.

The above categories are intended as a rule-of-thumb, not as strict guidelines for what level of security you will want for a particular application. Obviously, your particular application may fit into one of the above categories and require a higher level of security than indicated.

You could also reduce the recommended level of security, but remember that security levels are always decreasing as hardware improves; what we consider to be "high" security today may not be very secure at all in 10 years. The prime example of this is the fact that 56-bit DES was considered nearly "unbreakable" when it was first introduced, but was broken just 20 or so years later with a concerted effort. Now, more than 10 years after that, DES is practically obsolete, and even the beefed-up triple-DES is showing its age.

If you haven't figured it out by this point, this chapter is primarily focused on applications requiring security up to medium level and some medium-high level (as described in the categories above). If your application falls into the medium-high category or higher, you will definitely want to do some serious research before jumping into implementation. In fact, you should do a lot of research if your application needs security at all.

This book and others will be useful to determine what you need for your application, but you shouldn't ever take one person's or book's viewpoint as gospel. It is highly likely (pretty much guaranteed) that any single viewpoint on security will miss something important that can come back to haunt you later. That being said, we will move forward and attempt to make some sense of cryptography for systems not strictly designed for it.

**Hashing - Low Security, High Performance**

12.2 Hashing - Low Security, High Performance

Ideally, we would not need any security in our applications at all, instead having the luxury to focus all resources on performance. Unfortunately, this is not reality and we have to balance security with performance. Fortunately, the security-performance tradeoff is relatively linear; higher security means some performance loss, but the balance can be adjusted fairly easily by choosing different algorithms, methods, and key sizes.

For the best performance with some level of security, you can't beat the hash algorithms. As far as resources are concerned, they use far less than other types of cryptographic algorithms. Not strictly cryptography in the classic sense, hash algorithms provide integrity without protection from eavesdropping; in many applications this is all the security that is really needed.
How do we actually make hashing work for us in our applications? Probably the most common use of hashing is to provide a guarantee of integrity. The concept behind cryptographically secure hashing is that hash collisions are very difficult to create. Therefore, if you hash some information and compare it to a hash of that information provided to you by a trusted source and the hashes match, you have a fairly high certainty that the information has not changed or been tampered with in transit.

Another mechanism that employs hashing to provide a level of security is called "challenge-response," after the way the mechanism works. In a challenge-response operation, the system or user requesting the information provides a "challenge" to the sender of the requested information. The sender must then provide an appropriate "response" to the requestor, or the information is deemed invalid.

There are many variants of this simple method for verifying the authenticity of information, but for our purposes, the variant we will talk about uses hashing to prove that the requestor and sender (client and server) both know some shared secret only known to those systems. The challenge is typically some cryptographically secure random number that is difficult to predict, and it is sent by the client to the server (or from the server to the client, as is the case with some HTTP applications). Once the server receives the challenge, it calculates a hash of the challenge and the secret, and sends the hash back to the client. The client also performs the same operation on its copy of the secret, so when it receives the hash it will know that the server knows the same secret.

The reason the mechanism works is because the server could not produce the correct hash value unless it knew the secret. The random number for the challenge ensures that every hash sent back to the client is different; otherwise an attacker could simply send that value to the client and there would be no security.

The challenge-response hashing described here does suffer from a man-in-the-middle vulnerability, since an attacker could intercept the message in transit from the server back to the client. This attack requires the attacker to have access to the routing system of the network in order to spoof the address of the server and the client so it can intercept and retransmit the messages. However, for security at this level, it can usually be assumed that breaking the network routing will be sufficiently difficult to provide a decent level of security.

So now we know how hashing can be used to provide different types of security, how do we actually use it in practice? There are several hashing algorithms available, but the most common are MD5 and SHA-1. Unfortunately, the MD5 algorithm has been shown to have some weaknesses, and there is a general feeling of uneasiness about the algorithm. SHA-1 has been the focus of some suspicion as well, but it has not had the same negative press received by MD5. In any case, both algorithms are still heavily used in the absence of anything better. For this reason, we will look at MD5 and SHA-1 for our examples. Though there is some risk that these particular algorithms will become obsolete, the methods described here for their use in cryptographic systems can be applied to any similarly structured hash algorithms.
In practice, many embedded development tool suites provide libraries for MD5, SHA-1, or both hashing algorithms. The algorithms themselves are not too difficult to implement (they actually consist of a number of quite repetitive operations), but the convenience of having a pre-tested implementation to use is quite nice. Hashing algorithms are fairly easy to optimize as well, so it is quite likely that the provided implementations will already be fairly optimal for your target hardware.

Using hash algorithms is quite easy, since they have only three basic operations that are provided in the user API:

1. Initialization, which sets up the state data structure used to actually perform the hash.

2. Hashing, which operates on the incoming data, typically in raw bytes (may also be text).

3. Finalization, which finishes up the hash, and copies the result into an output buffer.

In this chapter, we focus primarily on C-based development tools, so the following examples of hashing in practice directly apply to the C language, but the principles can be applied to other similar languages (such as Java or C++).

The basic operation of hashing algorithms uses a buffer, in C an array of `char` type, as a workspace for the hashing. The algorithms utilize a structure that provides state information across hashing operations, allowing for the hash to be added to in multiple operations, rather than all at once in a single hashing operation. This feature is the reason that each of the hashing APIs has three distinct operations.

The algorithms are set up so that they can utilize the state structure to keep the in-progress hash intact without requiring separate support from the application designer. With this setup, once the user application is ready, it can provide an output buffer for the finalization operation. This is important for networked environments where several hashes may be in progress simultaneously—the user need only keep track of the hash state structures.

So what does hashing look like in a real application? In the following C program, the user input is hashed into an output buffer (actual SHA-1 API may vary):

```c
#include < sha1.h >
```
```c
#include <stdio.h>

main () {

    char input_buf[128], output_buf[20];
    struct SHA1_state sha_state;
    int i, input_len;

    // Initialize the state structure, requires only a reference to the struct
    SHA1_init(&sha_state);

    // Get user input, make sure buffer is cleared first
    memset(input_buf, 0, sizeof(input_buf));
    scanf("%127s", input_buf);

    // Hash the input, with a length equal to the size of the user input. Note that
    // the input to the SHA1_hash function can be of any length
    // !!! Danger, strlen can overflow, so we terminate the buffer for safety
    input_buf[127] = 0;
    input_len = strlen(input_buf);

    SHA1_hash(&sha_state, input_buf, input_len);

    // Finally, finalize the hash and copy it into a buffer and display
    SHA1_finish( & sha_state, output_buf);
    for(i = 0; i < 20; ++i) {

        printf("%X", output_buf[i]);
    }
    printf("n");

} // End program
```

Listing 12.1: Hashing with SHA-1

That's it! Hashing is a very simple operation in code. Notice that we do some
defensive programming when using the strlen function. Unfortunately, the C
programming language does not have very good standard library support for
protecting against buffer overflow.

In our little program example, if the user entered enough data to fill up the
buffer to the end (more than 127 characters), we are relying on scanf to be
sure that the last element of the array contains a null-terminator. In our
program, the scanf "%s" type is used in the format string with the optional
width format parameter, so it should not cause any issues for the call to strlen later. However, if someone was to change the scanf to some other form of input, then the assumption may be violated. For this reason, we add an unconditional null-terminator to the end of the array to be sure that strlen will terminate appropriately.

The hashing example above illustrates the use of a cryptographic algorithm to protect data, but it also highlights the fact that anything in the program can become a security issue. The use of standard C library functions such as strlen can lead to unexpected and unintentional behavior. Sometimes this behavior goes unnoticed; sometimes it leads to a crash. All it takes is one malicious attacker to find the flaw in your program and exploit it somehow. It may not be that the attacker gains access to the whole bank, but shutting down a few hundred automated teller machines could do a lot of financial damage. All that the attacker needs is for you, the developer, to stop paying attention.

The example has a trick or two that help to keep the program safe, such as terminating the buffer (a simple operation that could be easily overlooked), but what if the algorithms themselves were the problem. In the next section we will look at some recent developments with two major hash algorithms (MD5 and SHA-1) that cast doubt on their continued utility.

**Is Hashing Considered Dangerous?**

**12.2.1 Is Hashing Considered Dangerous?**

In the past few years, cryptography has come into a lot of visibility, and the old faithful algorithms that have served us well for years are now being put to the test. The vast amount of information on the public Internet that needs to be protected has led to a virtual stampede of corporations, governments, organizations, and individuals studying the security mechanisms that form the foundation of that protection.

People on both sides of the law (and with varying levels of ethics) are racing to discover flaws in the most commonly used algorithms. After all, boatloads of money can be made if a security breach is discovered. For the "good" guys, the rewards are recognition and (hopefully) prompt fixing of the issue. The "bad" guys profit in numerous different ways. The end result is always the same, however: If an algorithm is broken, it usually means it's useless from that point on.

This insane virtual arms race has revealed that it is extremely hard to develop secure cryptographic algorithms (it's easy to write broken cryptographic algorithms), and it appears that hashing may be among the most difficult. The two major hash algorithms in use today (notably by SSL and TLS) are MD5 and SHA-1. At the time of the writing of this text, MD5 is considered "mostly broken" and SHA-1 is "sorta broken." What does that mean? Well, there are various ways a hash algorithm could be broken from a cryptographic standpoint. Some of these are:
• Take two arbitrary but different messages and hash them. If you can easily calculate a hash value that is the same for these different messages (a "hash collision"), then the algorithm is somewhat broken, and potentially seriously broken.

• Given a hash value, compute an arbitrary message to hash to that value. If this is easy, then the algorithm is a little more broken, since this starts to get into the area where the flaw can be exploited.

• Generate a meaningful message that generates a hash collision with another meaningful message. If this is easy, then the algorithm is mostly broken, and it is highly likely it provides no security whatsoever. If this is true for an algorithm, it is very easy to fool someone into accepting incorrect information (or worse, damaging information such as a virus or Trojan horse).

Each of the above levels of compromise is based on the idea that performing these operations on the algorithm is "hard" (infeasible given current technology and future technology for at least a few years). They all feed into one another as well, so if you can find an arbitrary hash collision, it is often easier to discover the other attacks.

Unfortunately, both MD5 and SHA-1 have been discovered to have vulnerabilities. For MD5 there are several demonstrations of ways to generate different meaningful messages that generate the same MD5 hash value. Granted, these operations are currently somewhat contrived and generally a little tricky, but it is only a matter of time until someone figures out how to do it fast and easy. Generally speaking, we don't need to rush out and pull MD5 out of all our applications, but if it isn't too difficult to do so, it is recommended. MD5 should not be used in new applications whenever possible.

The MD5 algorithm is fairly broken, but fortunately for us (and the rest of the world), SHA-1 is not as broken (yet). Researchers have discovered something akin to the first vulnerability (the arbitrary hash collision) in SHA-1, but as yet, there does not seem to be a way to translate that vulnerability into a security breach (as seems to be the case with MD5). Possibly by the time you read this, however, both SHA-1 and MD5 will be considered obsolete and will be replaced by something new (or at least in the process of being replaced).

SHA-1 and MD5, albeit broken, are still in heavy use today and will continue to be for some time. They are so integrated into the
security mechanisms that we have come to rely on that it will take years to pull them all out and replace them. Even then, many legacy systems may still require them in some capacity. This scenario obviously assumes there is a decent replacement. There are some contenders in the same family as SHA-1, but if that algorithm fails, it may be hard to tell if the SHA-1 vulnerabilities translate to its brethren.

One ray of hope, however, is that there may be another competition to develop a new cryptographic algorithm as was done with AES. Only this time, the target would be a hashing algorithm to replace the aging and ailing MD5 and the slightly less damaged SHA-1. Only time will tell what ends up happening with this. Heck, we might see a quantum computer implemented in the next few years that could make all of our current strategies obsolete overnight. We still need something in the meantime, however, and it worked for AES, so it may work for this too.

Coming up in Part 2: To Optimize or Not to Optimize...

Printed with permission from Newnes, a division of Elsevier. Copyright 2008. "Wireless Security: Know it All" by Praphul Chandra, Alan Bensky et al. For more information about this title and other similar books, please visit www.elsevierdirect.com.

Related links:
What you need to know about embedded systems security
Fundamentals of Embedded Systems Security
Security Protocols for the Embedded Internet
Implementing secure digital data transfer in portable handheld embedded devices: Part 1 | Part 2 | Part 3