CHAPTER

15

Implementation Issues (II)

The key negotiation protocol we designed leads to some new implementation issues.

15.1 Large Integer Arithmetic

The public-key computations all depend on large integer arithmetic. As we already mentioned, it is not easy to implement large integer arithmetic properly.

Large integer routines are almost always platform-specific in one way or another. The efficiencies that can be gained by using platform-specific features are just too great to pass up. For example, most CPUs have an add-with-carry operation to implement addition of multiword values. But in C or almost any other higher-level language, you cannot access this instruction. Doing large integer arithmetic in a higher-level language is typically several times slower than an optimized implementation for the platform. And these computations also form the bottleneck in public-key performance, so the gain is too important to ignore.

We won't go into the details of how to implement large integer arithmetic. There are other books for that. Knuth [75] is a good start, as is Chapter 14 of the *Handbook of Applied Cryptography* [90]. To us, the real question is how to *test* large integer arithmetic.

In cryptography, we have different goals from those of most implementers. We consider a failure rate of 2^{-64} (about one in 18 million trillion) unacceptable, whereas most engineers would be very happy to achieve this. Many

programmers seem to think that a failure rate of 2^{-20} (about one in a million) is acceptable, or even good. We have to do much better, because we're working in an adversarial setting.

Most block ciphers and hash functions are comparatively easy to test.[1] Very few implementation bugs lead to errors that are hard to find. If you make a mistake in the S-box table of AES, it will be detected by testing a few AES encryptions. Simple, random testing exercises all the data paths in a block cipher or hash function and quickly finds all systematic problems. The code path taken does not depend on the data provided, or only in a very limited way. Any decent test set for a symmetric primitive will exercise all the possible flows of control in the implementation.

Large integer arithmetic is different. The major difference is that in most implementations, the code path depends on the data. Code that propagates the last carry is used only rarely. Division routines often contain a piece of code that is used only once every 2^{32} divisions or even once every 2^{64} divisions. A bug in this part of the code will not be found by random testing. This problem gets worse as we use larger CPUs. On a 32-bit CPU, you could still run 2^{40} random test cases and expect that each 32-bit word value had occurred in each part of the data path. But this type of testing simply does not work for 64-bit CPUs.

The consequence is that you have to do extremely careful testing of your large integer arithmetic routines. You have to verify that every code path is in fact taken during the tests. To achieve this, you have to carefully craft test vectors: something that takes some care and precision. Not only do you have to use every code path, but you also need to run through all the boundary conditions. If there is a test with $a < b$, then you should test this for $a = b - 1$, $a = b$, and $a = b + 1$, but of course only as far as these conditions are possible to achieve.

Optimization makes this already bad situation even worse. As these routines are part of a performance bottleneck, the code tends to be highly optimized. This in turn leads to more special cases, more code path, etc., all of which make the testing even harder.

A simple arithmetic error can have catastrophic security effects. Here is an example. While Alice is computing an RSA signature, there is a small error in the exponentiation modulo p but not modulo q. (She is using the CRT to speed up her signature.) Instead of the proper signature σ, she sends out $\sigma + kq$ for some value of k. (The result Alice gets is correct modulo q but wrong modulo p, so it must be of the form $\sigma + kq$.) The attacker knows $\sigma^3 \bmod n$, which is the number Alice is computing a root of, and which only depends on the message. But $(\sigma + kq)^3 - \sigma^3$ is a multiple of q, and taking the greatest common divisor of this number and n will reveal q and thus the factorization of n. Disaster!

[1] Two notable exceptions are IDEA and MARS, which often use separate code for special cases.

So what are we to do? First of all, don't implement your own large integer routines. Get an existing library. If you want to spend any time on it, spend your time understanding and testing the existing library. Second, run really good tests on your library. Make sure you test every possible code path. Third, insert additional tests in the application. There are several techniques you can use.

By the way, we've discussed the testing problem in terms of different code paths. Of course, to avoid side-channel attacks (see Sections 8.5 and 15.3), the library should be written in such a way that the code path doesn't change depending on the data. Most of the code path differences that occur in large integer arithmetic can be replaced by masking operations (where you compute a mask from the ''if'' condition and use that to select the right result). This addresses the side-channel problem, but it has the same effect on testing. To test a masked computation, you have to test both conditions, so you have to generate test cases that achieve both conditions. This is exactly the testing problem we mentioned. We merely explained it in terms of code paths, as that seems to be easier to understand.

15.1.1 Wooping

The technique we describe in this section has the rather unusual name of *wooping*. During an intense discussion between David Chaum and Jurjen Bos, there was a sudden need to give a special verification value a name. In the heat of the moment, one of them suggested the name ''woop,'' and afterward the name stuck to the entire technique. Bos later described the details of this technique in his PhD thesis [18, ch. 6], but dropped the name as being insufficiently academic.

The basic idea behind wooping is to verify a computation modulo a randomly chosen small prime. Think of it as a cryptographic problem. We have a large integer library that tries to cheat and give us the wrong results. Our task is to check whether we get the right results. Just checking the results with the same library is not a good idea, as the library might make consistent errors. Using the wooping technique, we can verify the library computations, as long as we assume that the library is not actually malicious in the sense that it tries very hard to corrupt our verification computations.

First, we generate a relatively small random prime t, on the order of 64–128 bits long. The value of t should not be fixed or predictable, but that is what we have a PRNG for. The value of t is kept secret from all other parties. Then, for every large integer x that occurs in the computations, we also keep $\tilde{x} := (x \bmod t)$. The $\tilde{x}$ value is called the *woop* of x. The woop values have a fixed size, and are generally much smaller than the large integers. Computing the woop values is therefore not a great extra cost.

So now we have to keep woop values with every integer. For any input x to our algorithm, we compute $\tilde{x}$ directly as $x \bmod t$. For all our internal computations, we shadow the large integer computations in the woop values to compute the woop of the result without computing it from the large integer result.

A normal addition computes $c := a + b$. We can compute $\tilde{c}$ using $\tilde{c} = \tilde{a} + \tilde{b} \pmod{t}$. Multiplication can be handled in the same way. We could verify the correctness of $\tilde{c}$ after every addition or multiplication by checking that $c \bmod t = \tilde{c}$, but it is more efficient to do all of the checks at the very end.

Modular addition is only slightly more difficult. Instead of just writing $c = (a + b) \bmod n$, we write $c = a + b + k \cdot n$ where k is chosen such that the result c is in the range $0, \ldots, n - 1$. This is just another way to write the modulo reduction. In this case, k is either 0 or -1, assuming both a and b are in the range $0, \ldots, n - 1$. The woop version is $\tilde{c} = (\tilde{a} + \tilde{b} + \tilde{k} \cdot \tilde{n}) \bmod t$. Somewhere inside the modulo addition routine, the value of k is known. All we have to do is convince the library to provide us with k, so we can compute $\tilde{k}$.

Modular multiplication is somewhat more difficult to do. Again we have to write $c = a \cdot b + k \cdot n$; and to compute $\tilde{c} = \tilde{a} \cdot \tilde{b} + \tilde{k} \cdot \tilde{n} \pmod{t}$, we need $\tilde{a}$, $\tilde{b}$, $\tilde{n}$, and $\tilde{k}$. The first three are readily available, but $\tilde{k}$ will have to be teased out of the modular multiplication routine in some way. That can be done when you create the library, but it is very hard to retrofit to an existing library. A generic method is to first compute $a \cdot b$, and then divide that by n using a long division. The quotient of the division is the k we need for the woop computation. The remainder is the result c. The disadvantage of this generic method is that it is significantly slower.

Once you can keep the woop value with modular multiplications, it is easy to do so with the modular exponentiation as well. Modular exponentiation routines simply construct the modular exponentiation from modular multiplications. (Some use a separate modular squaring routine, but that can be extended with a woop value just like the modular multiplication routine.) Just keep a woop value with every large integer, and have every multiplication compute the woop of the result from the woops of the inputs.

The woop-extended algorithms compute the woop value of the results based on the woop values of the inputs: if one or more of the woop inputs is wrong, the woop output is almost certainly wrong, too. So once a woop value is wrong, the error propagates to the final result.

We check the woop values at the end of our computation. If the result is x, all you have to do is check that $(x \bmod t) = \tilde{x}$. If the library made any mistakes, the woop values will not match. We assume the library doesn't carefully craft its mistakes in a way that depends on the value t that we chose. After all, the library code was fixed long before we chose t, and the library code is not under control of the attacker. It is easy to show that any error the library might make

will be caught by the overwhelming majority of t values. So adding a woop verification to an existing library gives us an extremely good verification of the computations.

What we really want is a large integer library that has a built-in woop verification system. But we don't know of one.

How large should your woop values be? That depends on many factors. For random errors, the probability of the woop value not detecting the error is about $1/t$. But nothing is ever random in our world. Suppose there is a software error in our library. We've got to assume that the attacker knows this. She can choose the inputs to our computation, and not only trigger the error but also choose the difference that the error induces. This is why t must be a random, secret number; without knowing t, the attacker cannot target the error in the final result to a difference that won't be caught by our wooping.

So what would you do if you were an attacker? You would try to trigger the error, of course, but you would also try to force the difference to be zero modulo as many t's as you can. The simplest countermeasure is to require that t be a prime. If the attacker wants to cheat modulo 16 different 64-bit primes, then she will need to carefully select at least $16 \cdot 64 = 1024$ bits of the input. As most computations have a limited number of input bits that can be chosen by an attacker, this limits the probability of success of the attack.

Larger values for t are better. There are so many more primes of larger sizes that the probability of success rapidly disappears for the attacker. If we were to keep to our original goal of 128-bit security, we would need a 128-bit t, or something in that region.

Woop values are not the primary security of the system; they are only a backup. If a woop verification ever fails, we know we have a bug in our software that needs to be fixed. The program should abort whatever it is doing and report a fatal error. This also makes it much harder for an attacker to perform repeated attacks on the system. Therefore, we suggest using a 64-bit random prime for t. This will reduce the overhead significantly, compared to using a 128-bit prime, and in practice, it is good enough. If you cannot afford the 64-bit woop, a 32-bit woop is better than nothing. Especially on most 32-bit CPUs, a 32-bit woop can be computed very efficiently, as there are direct multiplication and division instructions available.

If you ever have a computation where the attacker could provide a large amount of data, you should check the intermediate woop values as well. Each check is simple: $(x \bmod t) \stackrel{?}{=} \tilde{x}$. By checking intermediate values that depend on only a limited number of bits from the attacker, you make it harder for her to cheat the woop system.

Using a large integer library with woop verifications is our strong preference. It is a relatively simple method of avoiding a large number of potential security problems. And we believe it is less work to add woop verification to the library

once than to add application-specific verifications to each of the applications that uses the library.

15.1.2 Checking DH Computations

If you don't have a woop-enabled library, you will have to work without one. The DH protocol we described already contains a number of checks; namely, that the result should not be 1 and that the order of the result should be q. Unfortunately, the checks are not performed by the party doing the computation, but by the party receiving the result of the computation. In general, you don't want to send out any erroneous results, because they could leak information, but in this particular case it doesn't seem to do much harm. If the result is erroneous, the protocol will fail in one way or another, so the error will be noticed. The protocol safety only breaks down when your arithmetic library returns x when asked to compute g^x, but that is a type of error that normal testing is very likely to find.

Where needed, we would probably run DH on a library without woop-verification. The type of very rare arithmetical errors that we worry about here are unlikely to reveal x from a g^x computation. Any other mistake seems harmless, especially since DH computations have no long-term secrets. Still, we prefer to use a wooping library wherever possible, just to feel safe.

15.1.3 Checking RSA Encryption

RSA encryption is more vulnerable and needs extra checks. If something goes wrong, you might leak the secret that you are encrypting, or even your secret key.

If woop-verification is not available, there are two other methods to check the RSA encryption. Suppose the actual RSA encryption consists of computing $c = m^5 \bmod n$, where m is the message and c the ciphertext. To verify this, we could compute $c^{1/5} \bmod n$ and compare it to m. The disadvantages are that this is a very slow verification of a relatively fast computation, and that it requires knowledge of the private key, which is typically not available when we do RSA encryption.

Probably a better method is to choose a random value z and check that $c \cdot z^5 = (m \cdot z)^5 \bmod n$. Here we have three computations of fifth powers: the $c = m^5$; the computation of z^5; and then finally the check that $(mz)^5$ matches $c \cdot z^5$. Random arithmetical errors are highly likely to be caught by this verification. By choosing a random value z, we make it impossible for any attacker to target the error-producing values. In our designs, we only use RSA encryption to encrypt random values, so the attacker cannot do any targeting at all.

15.1.4 Checking RSA Signatures

RSA signatures are really easy to check. The signer only has to run the signature verification algorithm. This is a relatively fast verification, and arithmetical errors are highly likely to be caught. Every RSA signature computation should verify the results by checking the signature just produced. There is no excuse not to do this.

15.1.5 Conclusion

Let us make something quite clear. The checks we have been talking about are *in addition* to the normal testing of the large integer libraries. They do not replace the normal testing that any piece of software, especially security software, should undergo.

If any of these checks ever fail, you know that your software just failed. There is not much you can do in that situation. Continuing with the work you are doing is unsafe; you have no idea what type of software error you have. The only thing you can really do is log the error and abort the program.

15.2 Faster Multiplication

There are a lot of ways in which you can do a modulo multiplication faster than a full multiply followed by a long division. If you have to do a lot of multiplications, then Montgomery's method [93] is the most widely used one; see [39] for a readable description.

The basic idea behind Montgomery's method is a technique to compute ($x \bmod n$) for some x much larger than n. The traditional ''long division'' method is to subtract suitable multiples of n from x. Montgomery's idea is simpler: divide x repeatedly by 2. If x is even, we divide x by two by shifting the binary representation one bit to the right. If x is odd, we first add n (which does not change the value modulo n, of course) and then divide the even result by 2. (This technique only works if n is odd, which is always the case in our systems. There is a simple generalization for even values of n.) If n is k bits long, and x is not more than $(n-1)^2$, we perform a total of k divisions by 2. The result will always be in the interval $0, \ldots, 2n-1$, which is an almost fully reduced result modulo n.

But wait! We've been dividing by 2, so this gives us the wrong answer. Montgomery's reduction does not actually give you ($x \bmod n$), but rather $x/2^k \bmod n$ for some suitable k. The reduction is faster, but you get an extra factor of 2^{-k}. There are various tricks to deal with this extra factor.

One bad idea is to simply redefine your protocol to include an extra factor 2^{-k} in the computations. This is bad because it mixes different levels. It modifies the cryptographic protocol specification to favor a particular implementation technique. Perhaps you'll want to implement the protocol on another platform later, where you'll find that you don't want to use Montgomery multiplication at all. (Maybe that platform is slow but has a large integer coprocessor that performs modular multiplication directly.) In that case, the 2^{-k} factors in the protocol become a real hindrance.

The standard technique is to change your number representation. A number x is represented internally by $x \cdot 2^k$. If you want to multiply x and y, you do a Montgomery multiplication on their respective representations. You get $x \cdot 2^k \cdot y \cdot 2^k$, but you also get the extra 2^{-k} factor from the Montgomery reduction, so the final result is $x \cdot y \cdot 2^k \bmod n$, which is exactly the representation of xy. The overhead cost of using Montgomery reduction therefore consists of the cost of converting the input numbers into the internal representation (multiplication by 2^k) and the cost of converting the output back to the real result (division by 2^k). The first conversion can be done by performing a Montgomery multiplication of x and $(2^{2k} \bmod n)$. The second conversion can be done by simply running the Montgomery reduction for another k bits, as that divides by 2^k. The final result of a Montgomery reduction is not guaranteed to be less than n, but in most cases it can be shown to be less than $2n - 1$. In those situations, a simple test and an optional subtraction of n will give the final correct result.

In real implementations, the Montgomery reduction is never done on a bit-by-bit basis, but per word. Suppose the CPU uses w-bit words. Given a value x, find a small integer z such that the least significant word of $x + zn$ is all zero. You can show that z will be one word, and can be computed by multiplying the least significant word of x with a single word constant factor that only depends on n. Once the least significant word of $x + zn$ is zero, you divide by 2^w by shifting the value a whole word to the right. This is much faster than a bit-by-bit implementation.

15.3 Side-Channel Attacks

We discussed timing attacks and other side-channel attacks briefly back in Section 8.5. The main reason we were brief there is not because these attacks are benign. It is, rather, that timing attacks are also useful against public-key computations and we now consider both.

Some ciphers invite implementations that use different code paths to handle special situations. IDEA [84, 83] and MARS [22] are two examples. Other ciphers use CPU operations whose timing varies depending on the data they

process. On some CPUs, multiplication (used by RC6 [108] and MARS) or data-dependent rotation (used by RC6 and RC5 [107]) has an execution time that depends on the input data. This can enable timing attacks. The primitives we have been using in this book do not use these types of operations. AES is, however, vulnerable to cache timing attacks [12], or attacks that exploit the difference in the amount of time it takes to retrieve data from cache rather than main memory.

Public-key cryptography is also vulnerable to timing attacks. Public-key operations often have a code path that depends on the data. This almost always leads to different processing times for different data. Timing information, in turn, can lead to attacks. Imagine a secure Web server for e-commerce. As part of the SSL negotiations, the server has to decrypt an RSA message chosen by the client. The attacker can therefore connect to the server, ask it to decrypt a chosen RSA value, and wait for the response. The exact time it takes the server to respond can give the attacker important information. Often it turns out that if some bit of the key is one, inputs from set A are slightly faster than inputs from set B, and if the key bit is zero, there is no difference. The attacker can use this difference to attack the system. She generates millions of queries from both sets A and B, and tries to find a statistical difference in the response times to the two groups. There might be many other factors that influence the exact response time, but she can average those out by using enough queries. Eventually she will gather enough data to measure whether the response times for A and B are different. This gives the attacker one bit of information about the key, after which the attack can proceed with the next bit.

This all sounds far-fetched, but it has been done in the laboratory, and could very well be done in practice [21, 78].

15.3.1 Countermeasures

There are several ways to protect yourself against timing attacks. The most obvious one is to ensure that every computation takes a fixed amount of time. But this requires the entire library be designed with this goal in mind. Furthermore, there are sources of timing differences that are almost impossible to control. Some CPUs have a multiplication instruction that is faster for some values than for others. Many CPUs have complicated cache systems, so as soon as your memory access pattern depends on nonpublic data, the cache delays might introduce a timing difference. It is almost impossible to rid operations of all timing differences. We therefore need other solutions.

An obvious idea is to add a random delay at the end of each computation. But this does not eliminate the timing difference. It just hides it in the noise of the delay. An attacker who can take more samples (i.e., get your machine to do more computations) can average the results and hope to average out the random delay that was added. The exact number of tries the attacker needs

depends on the magnitude of the timing difference the attacker is looking for, and the magnitude of the random delay that is added. In real timing attacks, there is almost always a lot of noise, so any attacker who tries a timing attack is already doing the averaging. The only question is the ratio of the signal to the noise.

A third method is to make an operation constant-time by forcing it to last a standardized amount of time. During development, you choose a duration d that is longer than the computation will ever take. You then mark the time t at which the computation started, and after the computation you wait until time $t + d$. This is slightly wasteful, but it is not too bad. We like this solution, but it only provides protection against pure timing attacks. If the attacker can listen in on the RF radiation that your machine emits or measure the power consumption, the difference between the computation and the delay is probably detectable, which in turn allows timing attacks as well as other attacks. Still, an RF-based attack requires the attacker to be physically close to the machine. That enormously reduces the threat, compared to timing attacks that can be done over the Internet.

You can also use techniques that are derived from blind signatures [78]. For some types of computations they can hide (almost) all of the timing variations.

There is no perfect solution to the problem of timing attacks. It is simply not possible to secure the computers you can buy against a really sophisticated attack such as an RF-based one. But although you can't create a perfect solution, you can get a reasonably good one. Just be really careful with the timing of your public-key operations. An even better solution than just making your public-key operations fixed-time is to make the entire transaction fixed-time, using the technique mentioned above. That is, you not only make the public-key operation fixed-time, but you also fix the time between when the request comes in and the response goes out. If the request comes in at a time t, you send the response at time $t + C$ for some constant C. But to make sure you never leak any timing information, you had better be sure that the response will be ready at time $t + C$. To guarantee this, you will probably have to limit the frequency at which you accept incoming requests to some fixed upper bound.

15.4 Protocols

Implementing cryptographic protocols is not that different from implementing communication protocols. The simplest method is to maintain the state in the program counter, and simply perform each of the steps of the protocol in turn. Unless you use multithreading, this stops everything else in the program while you wait for an answer. As the answer might not be forthcoming, this is often a bad idea.

A better solution is to keep an explicit protocol state, and update the state each time a message arrives. This message-driven approach is slightly more work to implement, but it provides much more flexibility.

15.4.1 Protocols Over a Secure Channel

Most cryptographic protocols are executed over insecure channels, but sometimes you run a cryptographic protocol over a secure channel. This makes sense in some situations. For example, each user has a secure channel to a key distribution center; the key distribution center uses a simple protocol to distribute keys to the users to allow them to communicate to each other. (The Kerberos protocol does something like this.) If you are running a cryptographic protocol with a party you have already exchanged a key with, you should use the full secure channel functionality. In particular, you should implement replay protection. This is very easy to do, and it prevents a large number of attacks on the cryptographic protocol.

Sometimes the secure channel allows the protocol to use shortcuts. For example, if the secure channel provides replay protection, the protocol itself does not have to. Still, the old modularization rule states that the protocol should minimize its dependency on the secure channel.

For the rest of our protocol implementation discussion, we are going to assume that the protocol runs over an insecure channel. Some of the discussion does not quite apply to the secure channel case, but the solutions can never hurt.

15.4.2 Receiving a Message

When a protocol state receives a message, there are several checks that have to be made. The first is to see if the message belongs to the protocol at all. Each message should start with the following fields:

Protocol identifier. Identifies exactly which protocol and protocol version this is. Version identifiers are important.

Protocol instance identifier. Identifies which instance of the protocol this message belongs to. Perhaps Alice and Bob are running two key negotiation protocols simultaneously, and we don't want to confuse the two runs.

Message identifier. Identifies the message within the protocol. The easiest method is to simply number them.

Depending on the situation, some of these identifiers can be implicit. For example, for protocols that run over their own TCP connection, the port number and its associated socket uniquely identify the protocol instance on

the local machine. The protocol identifier and version information only need to be exchanged once. Note that it is important to exchange them at least once to make sure they get included in any authentication or signature used in the protocol.

After checking the protocol identifier and instance identifier, we know which protocol state to send the message to. Let us assume that the protocol state has just received message $n - 1$ and is expecting to receive message n.

If the received message is indeed message n, things are easy. Just process it as the protocol rules specify. But what if it has a different number?

If the number is larger than n or less than $n - 1$, something very weird is going on. Such a message should not have been generated, and therefore must be a forgery of some kind. You must ignore the contents of the forged message.

If the received message is message $n - 1$, the reply message you sent might not have arrived. At least, this could happen if you are running the protocol over an unreliable transport system. As we want to minimize dependencies on other parts of the system, this is exactly what we will assume.

First of all, check that the newly received message $n - 1$ is absolutely identical to the previous message with number $n - 1$ that you received. If they are different, you must ignore the new message. Sending a second answer will break the security of many protocols. If the messages are identical, just resend your reply. Of course, the version that you resend must be identical to the previous reply that you sent.

If you ignored the received message due to any of these rules, you have a second decision to make. Should you abort the protocol? The answer depends to some extent on the application and situation. If you have been running a protocol over a secure channel, something is very wrong. Either the secure channel is compromised, or the party you are talking to is misbehaving. In either case, you should abort the protocol and the channel. Simply delete the protocol state and the channel state, including the channel key.

If you're running the protocol over an insecure channel, then any of the ignored messages could be from an attacker trying to interfere with the protocol. Ideally, you would ignore the attacker's messages and just complete the protocol. This is, of course, not always possible. For example, if the attacker's forged message $n - 1$ reaches you first, you will send a reply. If you later receive the ''real'' message $n - 1$, you are forced to ignore it. There is no recovery from this situation, as you cannot safely send a second reply. But you have no idea which of the two messages $n - 1$ you received was the real one, so in order to have the best chance of completing the protocol successfully, you should just log the second message $n - 1$ as an error and continue as usual. If the message you replied to came from the attacker, the protocol will fail eventually because cryptographic protocols are specifically designed to

prevent attackers from successfully completing the protocol with one of the participants.

15.4.3 Timeouts

Any protocol run includes timeouts. If you don't get a response to a message within a reasonable time, you can resend your last message. After a few resends, you have to give up. There is no point continuing with a protocol when you cannot communicate with the other party.

The easiest way to implement timeouts is to send timing messages to the protocol state. You can use timers explicitly set by the protocol, or use timing messages that are sent every few seconds or so.

One well-known attack is to send lots of ''start-of-protocol'' messages to a particular machine. Each time you receive a start-of-protocol message, you initialize a new protocol execution state. After receiving a few million of these, the machine runs out of memory, and everything stops. A good example is the SYN flood attack. There is no easy method to protect yourself against these flooding attacks in general, especially in the age of botnets and distributed attacks, but they do show that it is important to delete old protocol states. If a protocol is stalled for too long, you should delete it.

The proper timing for resends is debatable. In our experience, a packet on the Internet either arrives within a second or so, or is lost forever. Resending a message if you haven't received a reply within five seconds seems reasonable. Three retries should be enough; if the message loss rate is so high that you lose four consecutive messages spread out over 15 seconds, you're not going to get a whole lot done over that connection. We prefer to inform the user of a problem after 20 seconds, rather than require the user to sit there and wait for a minute or two.

15.5 Exercises

Exercise 15.1 Consider all the operations a computer might perform with a cryptographic key. Which ones might have timing characteristics that could leak information about the key?

Exercise 15.2 Find a new product or system that manipulates secret data. This might be the same product or system you analyzed for Exercise 1.8. Conduct a security review of that product or system as described in Section 1.12, this time focusing on issues surrounding side-channel attacks.