Chapter 8

Implementation Pitfalls

We have now learned about several very important cryptographic objects, including block ciphers, encryption schemes, message authentication schemes, and hash functions. Moreover, we discussed how to construct instances of some of these cryptographic objects such that the constructs are provably secure under reasonable assumptions. For example, in Theorem 5.19 we show that CBC$ is a secure encryption scheme under chosen-plaintext attacks if we assume that the base block cipher is a secure PRF or PRP with a large block size. We have also shown that some cryptographic objects are insecure, e.g., the attack against CBCC in Section 5.5.3.

Let us now step back for a moment and ask ourselves how to apply what we have learned in practice. Suppose that our employer asks us to design and implement the cryptographic portion of some software application. How should we proceed and, more importantly, what should we be careful about? Similarly, what should we look out for when deciding whether to use someone else's cryptographic product?

There are plenty of mistakes that one could accidentally make when designing and implementing the cryptographic portion of a system. Here we look at some of the most common pitfalls, and we discuss how these pitfalls relate to what we have already learned in class. At a very high level, to avoid the most common pitfalls, we suggest that people implementing cryptography:

1. Use widely accepted and believed to be secure cryptographic primitives (like AES).

2. Use a construction that is provably secure under reasonable assumptions (like CBC$; Theorem 5.19).

3. Do not assume that a construction has any security properties besides what you have proven (e.g., an encryption scheme may not provide authenticity).

4. Realize that simply the use of a secure scheme (like CBC$) does not immediately imply that your entire system will be secure.

5. Make sure that your implementation corresponds exactly to what you or someone else proved secure (e.g., be careful about how you generate "random" bits for keys and IVs).

8.1 Not using standard primitives

¿From our discussions in Chapters 3 and 6, it should be clear that it is very difficult to design strong cryptographic primitives like DES, AES, and SHA1. Classically, one of the biggest pitfalls in designing systems that use cryptography is to try to invent a new cryptographic primitive and to not subject the security of that new primitive to the scrutiny of the cryptographic community.

Let us consider one example: a block cipher standardized by the ITU and used in ATM-based passive optical networks [8]. The block cipher, CHURN: $\{0,1\}^8 \times \{0,1\}^4$, works as follows.

algorithm $\text{CHURN}_K(M)$
Parse M as bits $M[1]\|M[2]\|M[3]\|M[4]$
Parse K as bits $K[1]\|K[2]\|K[3]\|K[4]\|K[5]\|K[6]\|K[7]\|K[8]$
If $K[1] = 1$ **then** swap $M[1]$ and $M[2]$
If $K[2] = 1$ **then** swap $M[3]$ and $M[4]$
$N[1] = M[1] \oplus K[3]$; $N[2] = M[3] \oplus K[4]$
$N[3] = M[2] \oplus K[5]$; $N[4] = M[4] \oplus K[6]$
If $K[7] = 1$ **then** swap $N[1]$ and $N[2]$
If $K[8] = 1$ **then** swap $N[3]$ and $N[4]$
return $N[1]\|N[2]\|N[3]\|N[4]$

The inverse is defined in the natural way.

Can you spot what is wrong with this design? It's an incredibly simply design, and very insecure. The key is only 8-bits long. And the block size is only 4-bits long. Yet this scheme will supposedly see widespread use on ATM-based passive optical networks [8]. After reading Chapters 3 and 4, we doubt that anyone in this class would think of using the CHURN block cipher. Yet it might seem like an attractive design to someone who has not taken this course.

There are other block ciphers that might appear secure at first glance, but really are not. A wonderful example is the block cipher FEAL: $\{0,1\}^{64} \times \{0,1\}^{64} \to \{0,1\}^{64}$ [7]. We don't describe the cipher here, but note that it is possible to recover the key using only several thousand known plaintexts [3], or 8 chosen plaintexts [2]. This means that we can construct a practical adversary A such that $\mathbf{Adv}^{\text{prf}}_{\text{FEAL}}(A)$ is *very* close to 1. Let's follow the implications of this to CBC\$ built from FEAL. Theorem 5.19 tells us that given an adversary A against CBC\$, we can construct an adversary B against FEAL such that

$$\mathbf{Adv}^{\text{ind-cpa}}_{\text{CBC\$}}(A) \leq \mathbf{Adv}^{\text{prf}}_{\text{FEAL}}(B) + \frac{\sigma^2}{2^{65}} \ .$$

Here we assume that adversary A's oracle queries total at most σ 64-bit blocks. But since $\mathbf{Adv}^{\text{prf}}_{\text{FEAL}}(B)$ may be very close to 1, this theorem does not imply that $\mathbf{Adv}^{\text{ind-cpa}}_{\text{CBC\$}}(A)$ is small, as would be necessary for us to conclude that CBC\$ built from FEAL is secure. Therefore, we cannot use Theorem 5.19 to argue the security of CBC\$ built from FEAL. (This is as it should be since CBC\$ built from FEAL is not secure. Given only the information above, can you see why?)

(There have also been a number of published attacks against custom built stream ciphers for the cell phone industry. If there is interest, we can discuss these results too.)

The lesson here is to use widely accepted and believed-to-be secure cryptographic primitives, and to be wary of any product that does not. There are not too many such believed-to-be secure primitives. AES, DES, and SHA1 are among the select few.

8.2 Using a construct without proofs of security

There have been numerous cryptographic protocols, like encryption schemes and MACs, that did not come with proofs of security. That's not surprising since people did not start to prove the security of block cipher-based encryption schemes and MACs until the 1990s. What is unfortunate is that, without proofs of security, it is impossible to know whether a construction is actually secure or not. In fact, in the exercises you have already been asked to find attacks against constructions that do not come with proofs of security, even if they might appear secure at first sight.

Nowadays, many more people understand that it is important to use cryptographic protocols that come with proofs of security. Still, it is not uncommon to find homebrew security software using the ECB encryption mode, or something akin to CBCC. (We'll get back to some examples later.)

Hopefully, the discussions in Chapters 4 through 7 strongly motivate the fact that, whenever possible, software applications should use constructions that are provably secure under reasonable assumptions.

8.3 Not considering the security bounds

Another common implementation pitfall is to not fully understand the security bounds in the proofs. We saw a little bit of this when we discussed CBC$ with FEAL in Section 8.1 of this chapter. But now let's consider the case where we actually believe that the underlying cryptographic primitive is secure.

Let us define the encryption scheme CTRS$[$L$]$ $= (\mathcal{K}, \mathcal{E}, \mathcal{D})$ with a block cipher $E\colon \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$. Here L is an integer between 1 and $n-1$, and is a parameter of our construction. The key generation algorithm returns a randomly selected value from $\{0,1\}^k$. The encryption algorithm is shown below, and the decryption algorithm is defined in the natural way. This construction is very similar to CTR$ from Chapter 5.

algorithm $\mathcal{E}_K(M)$
  $m \leftarrow \lceil |M|/n \rceil$
  If $m \geq 2^{n-l}$ **then return** $\perp$
  $R \xleftarrow{\$} \{0,1\}^L$
  Pad $\leftarrow E_K(R\|\langle 1\rangle)\|E_K(R\|\langle 2\rangle)\|\cdots E_K(R\|\langle m\rangle)$
  Pad $\leftarrow$ the first $|M|$ bits of Pad
  $C' \leftarrow M \oplus$Pad
  $C \leftarrow R\|C'$
  return C

Here $\langle x\rangle$ denotes the $n-L$-bit encoding of the integer $x \in \{0, \ldots, 2^{n-L}-1\}$.

We can prove the following result about the above construction.

Theorem 8.3.1 *Let $E\colon \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a family of functions, let $L \in \{1, \ldots, n-1\}$ be an integer, and let CTRS$[$L$]$ $= (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be the corresponding CTRS$[$L$]$ symmetric encryption scheme as described above. Let A be an adversary (for attacking the IND-CPA security of CTRS$[$L$]$) that runs in time at most t and asks at most q queries, these totaling at most σ n-bit blocks. Then there exists an adversary B (attacking the PRF security of E) such that*

$$\mathbf{Adv}^{\text{ind-cpa}}_{CTRS\$[L]}(A) \leq \mathbf{Adv}^{\text{prf}}_{E}(B) + \frac{q^2}{2^{L+1}} \; . \tag{8.1}$$

Furthermore B runs in time at most $t' = t + O(q + n\sigma)$ and asks at most $q' = \sigma$ oracle queries. ∎

Do you think that this theorem means that CTRS$[L] is secure, assuming that the block cipher E is secure?

The answer is: it depends. Since L is a parameter of the above construction, those implementing the above construction may set L to any value of their choice, e.g., 24, 40, 64, or 80. The only requirement is that L should not change between successive invocations of $\mathcal{E}$ and $\mathcal{D}$. Regardless of what we choose to use for L, we need to plug L, and reasonable values of q, into Equation (8.1) before using Theorem 8.3.1 to argue that our implementation is secure.

The Wired Equivalent Privacy (WEP) encryption scheme for IEEE 802.11 wireless networks uses something like CTRS$[L], with $L = 24$ [4]. (WEP actually uses a stream cipher named RC4, but that detail does not affect our discussion here. Its worth noting that there are many other problems with WEP.) If we substitute the fact that WEP uses $L = 24$ into Equation (8.1), we get that

$$\mathbf{Adv}^{\text{ind-cpa}}_{\text{CTRS\$[24]}}(A) \leq \mathbf{Adv}^{\text{prf}}_{E}(B) + \frac{q^2}{2^{25}} \ .$$

If we continue by replacing q with 4096, the equation becomes

$$\mathbf{Adv}^{\text{ind-cpa}}_{\text{CTRS\$[24]}}(A) \leq \mathbf{Adv}^{\text{prf}}_{E}(B) + \frac{1}{2} \ .$$

This means that we cannot use Theorem 8.3.1 to argue that WEP will provide privacy against adversaries that make a small number of chosen-plaintext oracle queries. The above equation alone should be enough of an indication to us that it might not be a good idea to use CTRS$[L], for small L like $L = 24$. That intuition is correct. Indeed, one can construct an efficient adversary against the chosen-plaintext privacy of CTRS$[24] such that the adversary has high advantage. (Can you construct such an adversary?)

Using CTRS$[L] for large L, like $L = 80$, is still certainly reasonable.

8.4 Not using the right tool

Another very common pitfall is accidentally using the wrong tool for a task. In particular, it is very tempting to believe that encryption schemes provide some sort of message authenticity guarantees. Unfortunately, this is not necessarily true. We discussed this a lot in Section 7.2 when we motivated the need for message authentication codes. Still, confusing the properties of a cryptographic construct is serious enough and common enough to warrant re-mentioning here.

The general principle here is: it is very risky to assume that a cryptographic construct has properties other than what you or others can prove. Using a provably secure construction, but assuming the wrong properties, raises effectively the same concerns that we raised in Section 8.2, except that the problem here is more subtle: because the construction has a proof (of a different property), it is very easy to become confused about what properties the construction actually has.

Unfortunately, many systems have been attacked because the designers apparently assumed that an encryption scheme also provides message authenticity. For example, message authentication codes were not an integral part of the original IPsec specification, and this made IPsec vulnerable to certain attacks [1]. The first version of SSH also did not use a message authentication code [10]. And the IEEE 802.11 WEP construction does not use a message authentication code [4].

8.5 Not implementing *exactly* the construction that is provably secure

Another general class of failures is taking something that is provably secure, like CBC\$ with a believed-to-be-secure block cipher, but not implementing the construction *exactly* the way it was described when it was proven secure.

The problem is, unfortunately, that even the slightest tweak to a provably secure construct can render it insecure. This is again basically a rephrasing of our concern in Section 8.2 since, if one changes even a small portion of a construction that is provably secure, the original proof may no longer apply.

Let us consider some examples. The electronic voting machine manufacturer Diebold decided to encrypt their voting records using CBC\$, but instead of using a random IV, they chose to always use the all zero block as the IV [6]. Let's refer to this variant of CBC\$ as CBC0 since the IV is always zero. Unfortunately, Diebold's decision to remove the randomness from CBC\$ makes CBC0 stateless and deterministic. This has the same problems as encrypting a message with ECB mode — it will leak information. So the discussion earlier about not wanting to leak voter preferences via a stateless and deterministic encryption scheme was not very far off.

As another example, Microsoft was recently in the media because their Microsoft Word and Excel products use something like CTRS\$[L], except that they chose to use the same value for R each time they encrypt a file [9]. Unfortunately, by not implementing CTRS\$[L] exactly as described in Section 8.3 of this chapter, the encryption method in Microsoft Word and Excel fails to provide privacy under chosen-plaintext attacks.

As another example simple change that could render a implementation insecure, consider CTRC from Scheme 5.7. If a developer forgot to make the *ctr* variable static (i.e., if $\mathcal{E}$ reset *ctr* to 0 upon every invocation), then the implementation would not provide privacy under chosen-plaintext attacks.

8.6 Random numbers

Random numbers play an incredibly important role in cryptography. Consider any of the provably secure encryption or message authentication schemes from the previous chapters. Notice that they all use random coins in some way or another. For example, the encryption algorithm for CBC\$ uses random coins every time it encrypts a message. And all of the provably secure encryption schemes and message authentication schemes use random coins in their key generation algorithms.

For concreteness, let us consider the key generation and encryption algorithms for CBC\$ built with the block cipher E: $\{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ (copied here from Figure 5.2):

algorithm $\mathcal{K}$
  $K \xleftarrow{\$} \{0,1\}^k$
  return K

algorithm $\mathcal{E}_K(M)$
  if ($|M|$ mod $n \neq 0$ or $|M| = 0$) **then return** $\perp$
  Break M into n-bit blocks $M[1] \cdots M[m]$
  IV $\xleftarrow{\$} \{0,1\}^n$
  $C[0] \leftarrow$ IV
  for $i \leftarrow 1$ to m **do**
    $C[i] \leftarrow E_K(C[i-1] \oplus M[i])$
  $C \leftarrow C[i] \cdots C[m]$
  return $\langle \text{IV}, C \rangle$

Given this description, and a description of the block cipher E, any experienced C or Java programmer should be able to easily implement *most* parts of the above algorithms. This is because almost all of the operations in the above pseudocode are common to all popular languages. For example, the "$\leftarrow$" operation corresponds to the standard assignment operator ("`=`" in C and Java). The programmer might, however, be puzzled about how to implement the "$\xleftarrow{\$}$" randomized assignment operator from the lines

$$K \xleftarrow{\$} \{0,1\}^k$$

and

$$\mathrm{IV} \xleftarrow{\$} \{0,1\}^n \ .$$

It is worth thinking about how a programmer might instantiate the "$\xleftarrow{\$}$" operation in C or Java. In order to implement the algorithms *exactly* as described in the above pseudocode, the operation "$\xleftarrow{\$}$" must select bits independently and uniformly at random. If an implementation of CBC\$ does not do this, then the implementation is not *exactly* the object described above and in Figure 5.2. At a minimum, this means that the security of the software implementation of CBC\$ does not immediately follow from Theorem 5.19. In the worst case, not only might the security of the software implementation not follow from Theorem 5.19, but the software implementation may actually be insecure.

The first problem is that it is hard for computers, which are inherently deterministic, to select bits independently and uniformly at random. Therefore, people implementing cryptosystems are left to approximate the $\xleftarrow{\$}$ operation as best they can. The second problem is that there are many natural approaches for trying to implement the $\xleftarrow{\$}$ operation in C or Java, and some of these approaches can actually yield an insecure implementation. We consider some example (flawed) approaches for instantiating $\xleftarrow{\$}$ here.

8.6.1 The C random number generator

The C programming language has a built in "random number generator," called `rand`. Associated to `rand` is another function named `srand`. Therefore, a natural way to try to implement $\xleftarrow{\$}$ would be to use `rand` and `srand`.

At a high level, the way a programmer is supposed to use `rand` and `srand` is as follows. The program is first supposed to call `srand(seed)`, where `seed` is the "seed" to the C random number generator. After calling `srand`, the program can invoke `rand()` any number of times. Each time `rand()` will return a value that is supposed to appear random. Therefore, a natural way for trying to generate a large number of random bits is to invoke `rand()` as many times as necessary. Here we ignore how the programmer picks `seed`.

Let us look under the hood and see how `rand` and `srand` work. Although different systems implement these functions in slightly different ways, there is a lot of commonality between the code for these functions on different systems. Below we show the code for `srand` and `rand` on one popular system. For this system, `state` is a global 32-bit unsigned integer. We do not present actual C code, but try to capture as the essence of these functions in C-like pseudocode.

```
procedure srand(seed)
    state = seed;
```

```
function rand()
    state = ((state * 1103515245) + 12345)
        mod 2147483648;
    return state
```

Here 2147483648 is 2^{31}.

Let us now consider how a programmer might use `rand` to implement the CBC$ encryption scheme's key generation algorithm. We show our traditional pseudocode on the left, and a C-like pseudocode on the right.

algorithm $\mathcal{K}$	**function** `keygen()`
$K \xleftarrow{\$} \{0,1\}^k$	`key = rand();`
return K	**return** `key`

The first observation that we make is that, since `rand` returns a 32-bit integer, `key` must also be a 32-bit integer. If $k > 32$, then the implementation would clearly not be using a key selected randomly from the set of all strings k-bit strings $\{0,1\}^k$. The security implications of this should be clear. While we consider it impractical to exhaustively search a randomly selected 128-bit AES key, it would be practical to exhaustively search a 32-bit key generated via the above `keygen` code.

To fix the problem, one might try invoking `rand()` multiple times. For example, if the block cipher is AES with $k = 128$, the above pseudocode might change to:

algorithm $\mathcal{K}$	**function** `keygen()`
$K \xleftarrow{\$} \{0,1\}^{128}$	`key[0] = rand(); key[1] = rand();`
return K	`key[2] = rand(); key[3] = rand();`
	return `key`

where `key` is a now a four-element array of 32-bit unsigned integers.

But there is still something seriously wrong with the above implementation of `keygen` that could compromise the security of CBC$. To see the problem, let us return to how `rand` works. By looking at how `rand` works, we find that

$$\begin{aligned}
\texttt{key[1]} &= ((\texttt{key[0]} \cdot 1103515245) + 12345) \bmod 2^{31} \\
\texttt{key[2]} &= ((((\texttt{key[0]} \cdot 1103515245) + 12345) \cdot 1103515245) + \\
&\quad 12345) \bmod 2^{31} \\
\texttt{key[3]} &= ((((((\texttt{key[0]} \cdot 1103515245) + 12345) \cdot 1103515245) + \\
&\quad 12345) \cdot 1103515245) + 12345) \bmod 2^{31}
\end{aligned}$$

This means that now, even though `key` is now a 128-bit value (an array of four 32-bit elements), there are only 2^{32} possibilities for `key`. An adversary could therefore exhaustively search `key` using at most 2^{32} tries.

8.6.2 Key generation and the Netscape browser

¿From the above discussion, it should be clear that there are serious problems in one of the most natural approaches for trying to generate random numbers in software (using `rand`). There are two problems with `rand`. First, the `state` variable of rand is only 32-bits long, which means that the state can be exhaustively search using reasonable resources. Second, knowing one value of `state` (e.g., `Key[0]`) allows us to compute all previous or subsequent outputs of `rand` (e.g., `Key[1]`, `Key[2]`, and `Key[3]`).

Rather than use (`rand`), another natural approach for trying to create random numbers is to try to exploit properties of believed-to-be secure cryptographic objects, like AES or SHA1. This is exactly what version 1.1 of the Netscape browser did [5]. The following C-like pseudocode shows

the two main functions in Netscape's random number generator. We simplify the functions in order to capture the important properties.

```
procedure NetscapeRandSetup()
    pid = process ID;
    ppid = parent process ID;
    seconds = current time of day
        (seconds);
    microseconds = current time of day
        (microseconds);
    x = concatenation of pid, ppid,
        seconds, microseconds;
    NSseed = SHA1(x);
```

```
function NetscapeGetRand()
    rv = SHA1(NSseed);
    NSseed = NSseed + 1 mod 2^160;
    return rv;
```

Here `NSseed` is a global 160-bit (20 byte) string, which we sometimes interpret as a 160-bit unsigned integer; NS standards for Netscape, to avoid confusion with the `seed` variable used with C's standard `rand` function. As for why `NSseed` is 160-bits long, recall that SHA1 outputs a 160-bit value. This construction does seem better than `rand`. For example, given an output of `NetscapeGetRand`, and assuming reasonable properties of SHA1, it would seem hard to predict the next output of `NetscapeGetRand`. Or at least that's the intuition.

Below we show how Netscape 1.1 would instantiate $\mathcal{K}$ using the above functions:

```
algorithm K
    K <-$ {0,1}^128
    return K
```

```
function keygen();
    NetscapeRandSetup();
    tmp = NetscapeGetRand();
    key = first 128-bits of tmp;
    return key
```

Does the above `keygen` function generate keys uniformly and independently at random? Certain `keygen` uses a strong cryptographic object (SHA1) in its design, and it might be tempting to assume that the use of SHA1 "randomizes" the value of the output `key`.

Unfortunately, this reasoning is flawed. In particular, note that `key` ultimately depends only on the values of `pid`, `ppid`, `seconds`, and `microseconds`. If an adversary observes the time that it sees a user send an encrypted message, it would likely be able to guess `seconds`. Further, under many natural assumptions, the adversary would also be able to guess or exhaustively search the values for `pid`, `ppid`, and `microseconds`. Thus, an adversary would be able to exhaustively search `key` using a reasonable amount of resources, even though `key` is a 128-bit value.

8.6.3 Randomness during encryption

Let us now turn our attention to the CBC$ encryption algorithm. Recall that the CBC$ encryption algorithm is supposed to select the IV uniformly at random from $\{0,1\}^{128}$. It turns out that if the implementation of the encryption algorithm tries to do this, but instead selects the IV in a way that the adversary could predict, then the CBC$ implementation will fail to preserve the privacy of the encapsulated messages. (This is similar to Diebold's mistake, from Section 8.5, of always using the all zero block as the IV. However, in this case we assume that the designer is actually trying to implement CBC$ exactly as specified in Figure 5.2.)

Let us consider a simple example. Let us define a variant of CBC$, called CBCCIV. This variant uses the assumption that the last ciphertext block of a CBC$-encrypted message has "randomness" properties, and therefore can be used as the IV to encrypt the next message. This is a seductive assumption since we can prove that CBC MAC (Scheme 7.4) on fixed length messages is a secure PRF (the proof of this property is not stated in Chapter 7, but it is used in the proof of Theorem 7.5). For clarity, we show the encryption algorithm for CBCCIV below:

algorithm $\mathcal{E}_K(M)$
  if IV is undefined **then** IV $\xleftarrow{\$} \{0,1\}^n$
  if ($|M|$ mod $n \neq 0$ or $|M| = 0$) **then return** $\perp$
  Break M into n-bit blocks $M[1] \cdots M[m]$
  $C[0] \leftarrow$ IV
  for $i \leftarrow 1$ to m **do**
    $C[i] \leftarrow E_K(C[i-1] \oplus M[i])$
  $C \leftarrow C[i] \cdots C[m]$
  IV $\leftarrow C[m]$ // save last block as IV for next invocation
  return $\langle \text{IV}, C \rangle$

The designers of IPsec, SSL, and SSH all seem to have made the assumption that the last ciphertext block of an encrypted message is random, and therefore implement CBCCIV instead of CBC$.

Can you come up with an attack, in the IND-CPA setting, against the privacy of CBCCIV? Hint 1: at the time you encrypt the message M_{i+1}, the last ciphertext block for message M_i is *known* and, therefore, the IV used to encrypt the message M_{i+1} will not be chosen at random during the encryption process. Hint 2: try to generalize the attack in Section 5.5.3 against CBCC. Although this attack breaks CBCCIV in the IND-CPA setting, how easy do you think it would be to mount this attack in practice? If you don't think that it would be easy to mount in practice, do you think that you would be justified in still using CBCCIV?

Consider also instantiating CBC$, where the key generation algorithm somehow actually selects `key` uniformly at random from all 128-bit strings, but the encryption algorithm uses `NetscapeGetRand()` to generate the IVs. Can you come up with an attack against the privacy of this implementation? As above, try to generalize the attack in Section 5.5.3 against CBCC. And recall the earlier problems with `NetscapeGetRand`.

8.7 Not taking the security of the whole system into account

The cryptographic objects that we have learned about (e.g., encryption schemes and MACs) are very important tools for helping ensure the security of many systems. For example, if it were not for cryptography, many of us would never think of doing things like online banking or Internet shopping. That said, the provably secure cryptographic objects that we have learned about (like CBC$ or CBC MAC) are not, by themselves, sufficient to guarantee the security of a system that uses them. Said another way, one needs to be careful not to make the assumption that simply the use of a provably secure encryption scheme or MAC in a product will make that product secure. Let us consider two examples.

8.7.1 Combining cryptographic schemes

Just as the use of a believed-to-be-secure block cipher like AES in an encryption scheme does not necessarily mean that the encryption scheme is secure, the use of a provably secure encryption scheme or a MAC in a system does not necessarily mean that the system is secure.

For example, suppose that we wanted to implement something to protect *both* the privacy and the authenticity of encapsulated data. (Such objects are called *authenticated encryption schemes* in the literature.) Since we know that encryption schemes are designed to provide privacy (Section 7.2 and Section 8.4), and since we know that MACs are designed to provide authenticity, a natural approach for trying to create an authenticated encryption scheme would be to combine a provably secure encryption scheme with a provably secure MAC.

Given an encryption scheme $\mathcal{SE} = (\mathcal{K}_e, \mathcal{E}, \mathcal{D})$ and a MAC $\mathcal{MA} = (\mathcal{K}_m, \mathsf{MAC}, \mathsf{VF})$, we might try to construct an authenticated encryption $\mathcal{AE} = (\overline{\mathcal{K}}, \overline{\mathcal{E}}, \overline{\mathcal{D}})$ scheme as follows. The syntax for $\mathcal{AE}$ is exactly like a standard encryption scheme, and it has the same correctness requirement (if you encrypt a message under some key and then decrypt the resulting ciphertext with the same key, you will get back the original message).

Algorithm $\overline{\mathcal{K}}$	**Algorithm** $\overline{\mathcal{E}}_{\langle K_e, K_m\rangle}(M)$	**Algorithm** $\overline{\mathcal{D}}_{\langle K_e, K_m\rangle}(C)$
$K_e \xleftarrow{\$} \mathcal{K}_e$	$\sigma \xleftarrow{\$} \mathcal{E}_{K_e}(M)$	Parse C as $\langle \sigma, \tau\rangle$
$K_m \xleftarrow{\$} \mathcal{K}_m$	$\tau \xleftarrow{\$} \mathsf{MAC}_{K_m}(M)$	$M \leftarrow \mathcal{D}_{K_e}(\sigma)$
Return $\langle K_e, K_m\rangle$	$C \leftarrow \langle \sigma, \tau\rangle$	$v \leftarrow \mathsf{VF}_{K_m}(M, \tau)$
	Return C	**If** $v = 1$ **then return** M
		Else return $\perp$

The intuition for this construction is as follows: the encryption of the message M via the operation $\sigma \xleftarrow{\$} \mathcal{E}_{K_e}(M)$ is supposed to protect the privacy of the message. And the MACing of the message via $\tau \xleftarrow{\$} \mathsf{MAC}_{K_m}(M)$ is supposed to protect the authenticity of the message.

The above construction seems like a very natural way to combine an encryption scheme with a MAC. In fact, it is basically the composition method employed by recent versions of the popular SSH protocol (recall from Section 8.4 that the first version of the SSH protocol did not use a MAC). But does the above construction $\mathcal{AE}$ provide both privacy and authenticity assuming that $\mathcal{SE}$ is IND-CPA secure and that $\mathcal{MA}$ is UF-CMA secure?

The answer is: not necessarily. Consider, for example, the above construction where $\mathcal{SE}$ is CBC\$ and $\mathcal{MA}$ is CBC MAC. In this, if $\langle \sigma, \tau\rangle$ is the output of $\overline{\mathcal{E}}_{\langle K_e, K_m\rangle}(M)$, then because CBC MAC is stateless and deterministic, τ will leak information about the message M. This is basically an extension to the basic problem of using a stateless and deterministic encryption scheme, like ECB.

8.7.2 Key management

Key management is a critical part of any system that uses cryptography. We already know that it is important for cryptographic keys to be generated randomly. But how does a system control the distribution or updating of keys? For example, if Alice and Bob want to use a symmetric encryption scheme $\mathcal{SE}$ to communicate privately, they both have to know the same secret key K.

In an IEEE 802.11 wireless network using WEP encryption, all participants will use the same encryption key K. Similarly, to encrypt the voting records, all Diebold electronic voting machines (at least up until summer 2003) used the same encryption key. In fact, in the case of Diebold, the encryption key was hard-coded into the software via the following line of C code:

```
#define DESKEY ((des_key*)"F2654hD4")
```

Even if the WEP or Diebold encryption keys were initially generated randomly, giving them to all members of a wireless network or all voting machines is not a good idea. For example, if one of the participants in the network or one of the authors or maintainers of the voting machines turned out to be malicious or subvertable, he or she could compromise the privacy of the encrypted content.

Bibliography

[1] Steven M. Bellovin. Problem areas for the IP security protocols. In *Proceedings of the 6th USENIX Security Symposium*, San Jose, California, July 1996.

[2] Eli Biham and Adi Shamir. Differential cryptanalysis of Feal and N-Hash. In Donald W. Davies, editor, *Advances in Cryptology – EUROCRYPT'91*, volume 547 of *Lecture Notes in Computer Science*, Brighton, UK, April 8–11, 1991. Springer-Verlag, Berlin, Germany.

[3] Bert Den Boer. Cryptanalysis of F.E.A.L. In C. G. Günther, editor, *Advances in Cryptology – EUROCRYPT'88*, volume 330 of *Lecture Notes in Computer Science*, Davos, Switzerland, May 25–27, 1988. Springer-Verlag, Berlin, Germany.

[4] Nikita Borisov, Ian Goldberg, and David Wagner. Intercepting mobile communications: The insecurity of 802.11. In *Seventh Annual International Conference on Mobile Computing and Networking*, 2001.

[5] Ian Goldberg and David Wagner. Randomness and the Netscape browser. *Dr. Dobb's Journal*, January 1996.

[6] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy 2004*, pages 27–40. IEEE Computer Society, May 2004.

[7] Akihiro Shimizu and Shoji Miyaguchi. Fast data encipherment algorithm FEAL. In Walter Fumy, editor, *Advances in Cryptology – EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, Konstanz, Germany, May 11–15, 1997. Springer-Verlag, Berlin, Germany.

[8] Stephen Thomas and David Wagner. Insecurity in ATM-based passive optical networks. In *IEEE International Conference on Communications (ICC 2002), Optical Networking Symposium*, 2002.

[9] Hongjun Wu. The misuse of RC4 in Microsoft Word and Excel. Cryptology ePrint Archive, Report 2005/007, 2005. `http://eprint.iacr.org/`.

[10] Tatu Ylonen. SSH — Secure login connections over the Internet. In *Sixth USENIX Security Symposium*, pages 37–42, 1996.