Disclaimer: These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.

4.1 Security notions and their relation

CCA security. Recall that an authenticated encryption scheme $\Pi = (K, \mathcal{E}, \mathcal{D})$ must satisfy the privacy and authenticity requirements. However, it’s sometimes more convenient if we can identify a single definitional requirement, instead of two. We now define what’s known as Chosen-Ciphertext Attack (CCA) security. One can prove that if $\Pi$ meets both the privacy and authenticity requirements then $\Pi$ must satisfy CCA security; this is left as an exercise.

In the CCA notion, an adversary $A$ is given two oracles $\mathcal{E}$ and $\mathcal{D}$. Initially, a uniformly random key $K \leftarrow K$ is picked. The adversary is then dropped into either the real world or a random world. In the real world, when the adversary queries $\mathcal{E}(M)$, the game returns $E_K(M)$. In the random world, the game instead picks a random message $M' \leftarrow \{0, 1\}^{|M|}$ and returns $E_K(M')$. In both worlds, the oracle $\mathcal{D}(\cdot)$ implements $D_K(\cdot)$. To prevent the adversary from trivially winning, if it gets $C$ from the oracle $\mathcal{E}$ then it can’t query $\mathcal{D}(C)$. The adversary then has to guess which world it’s in. Define

$$Adv_{\Pi}^{\text{CCA}}(A) = \Pr[\text{REAL}^A_{\Pi} \Rightarrow 1] - \Pr[\text{RAND}^A_{\Pi} \Rightarrow 1],$$

where games $\text{REAL}$ and $\text{RAND}$ are defined in Figure 4.1.

![Game REAL](image)

**Figure 4.1:** Games defining CCA security. Once the adversary receives $C$ from the first oracle, it’s prohibited from querying $C$ to the second oracle.

WEP construction. The WEP construction for WiFi security is given in Figure 4.2. It’s based on the stream cipher RC4, a tool for generating a stream of pseudorandom bits. WEP is an instance of the encrypt-with-redundancy paradigm. Specifically, to encrypt a message $M$ with IV $\in \{0, 1\}^{24}$, one first appends some CRC32 checking to $M$ to obtain a string $X \leftarrow M||\text{CRC}(M)$, generates a one-time pad $P$ from $\text{RC4(IV||K})$, and then outputs $\text{IV||(X \oplus P)}$. 

4-1
We now show that WEP doesn’t meet CCA security. The adversary first picks an arbitrary message \( M \in \{0, 1\}^\ell \) and queries it to the encryption oracle to get ciphertext \( IV \parallel C \). Pick an arbitrary message \( M^* \in \{0, 1\}^\ell \) such that \( M^* \neq M \). Let \( \Delta \leftarrow (M^* \parallel CRC(M^*)) \oplus (M \parallel CRC(M)) \). The adversary then queries \( IV \parallel (C \oplus \Delta) \) to the decryption oracle. If the answer is \( M^* \) then the adversary outputs 1, indicating that it’s in the real world. Otherwise, it outputs 0. In the real world, the adversary will always output 1. In the random world, let \( M' \) be the random message chosen by the encryption oracle. The adversary will output 1 only if \( M' = M \), which happens with probability \( 2^{-\ell} \). Hence the adversary wins with advantage \( 1 - 2^{-\ell} \).

**CPSS security.** We now define a weaker security notion, called *Chosen Prefix Secret Suffix* (CPSS). Let \( \Pi = (K, E, D) \) be an authenticated encryption scheme. Under this attack, the game picks a secret message \( M \leftarrow \{0, 1\}^\ell \) and key \( K \leftarrow K \). The adversary is given two oracles \( Enc \) and \( Dec \), and its goal is to guess the message \( M \). The adversary can choose a prefix \( P \) and query \( Enc(P) \) to get back \( E_K(P \parallel M) \). On query \( Dec(C) \), if the ciphertext \( C \) is valid, i.e. \( Dec_K(C) \neq \perp \), then the game returns 1 to the adversary, but the adversary doesn’t get to see the decrypted message. Otherwise the game returns 0. Define

\[
\text{Adv}_{\Pi}^{\text{CPSS}}(A) = \Pr[\text{CPSS}_A^{\Pi} \Rightarrow 1],
\]

where game CPSS is defined in Figure 4.3.

```plaintext
Game CPSS_{\Pi}^{A}
M \leftarrow \{0, 1\}^\ell; K \leftarrow K \\
M' \leftarrow A^{Enc(), Dec()} \\
return (M' = M)
```

```plaintext
procedure Enc(P)
return \( E_K(P \parallel M) \)
```

```plaintext
procedure Dec(C)
if \( D_K(C) \neq \perp \) then return 1 else return 0
```

![Figure 4.2: The WEP construction](image)

**Figure 4.3**: Game defining CPSS security.

The CPSS notion captures the following real-world scenario. In this setting, Alice first visits a legitimate website \( \text{bank.com} \). Her browser then shares a secret cookie \( M \) with this website, and embeds this in every request it sends to \( \text{bank.com} \). Suppose that later Alice is tricked into visiting a malicious website \( \text{attacker.com} \). This site sends a Javascript program to the browser, requesting some resource from \( \text{bank.com} \). The browser will correspondingly send an encrypted HTTP GET request to \( \text{bank.com} \). In other words, it sends \( C \leftarrow E_K(P \parallel M) \) with a prefix \( P \) that the adversary can partially control. The adversary then can intercept this request, replace the ciphertext with another \( C' \) of its choice, and observe the error message from \( \text{bank.com} \).
The code of \( \text{B} \) Dec otherwise. Due to the assumption above, \( \text{DecSim} \) queries and provides the latter with access to two (simulated) oracles \( \text{EncSim} \) oracle, it won’t query \( \text{C} \) to the decryption oracle, because it knows that it’ll get 1 anyway. We will construct an adversary \( B \) breaking CCA security of \( \Pi \). The adversary \( B^{\text{Enc,Dec}} \) first picks \( M \leftarrow \{0,1\}^t \) and runs \( A \) and provides the latter with access to two (simulated) oracles \( \text{EncSim} \) and \( \text{DecSim} \). Each time \( A \) queries \( \text{EncSim}(P) \), meaning that it wants an encryption of \( P \| M \), adversary \( B \) simply returns \( \text{Enc}(P||M) \). If \( A \) queries \( \text{DecSim}(C) \), \( B \) will query \( \text{Dec}(C) \). Adversary \( B \) will return 0 to \( A \) if it gets a \( \bot \)-answer, and return 1 otherwise. Due to the assumption above, \( B \) will never query a ciphertext \( C \) that it received from \( \text{Enc} \) to the oracle \( \text{Dec} \). Finally, when \( A \) outputs a message \( M' \), if \( M' = M \) then \( B \) outputs 1; otherwise it outputs 0. The code of \( B \) is given below.

**Game \( G_0 \)**

\[
\begin{align*}
K & \leftarrow K; \quad M \leftarrow \{0,1\}^t \\
M' & \leftarrow A^{\text{EncSim,DecSim}} \\
\text{return} & \ (M' = M) \\
\text{procedure} & \ \text{EncSim}(P) \\
\text{return} & \ \mathcal{E}_K(P\|M) \\
\text{procedure} & \ \text{DecSim}(C) \\
\text{if} & \ \mathcal{D}_K(C) \neq \bot \ \text{then return} \ 1 \ \text{else return} \ 0
\end{align*}
\]

**Game \( G_1 \)**

\[
\begin{align*}
K & \leftarrow K; \quad M \leftarrow \{0,1\}^t \\
M' & \leftarrow A^{\text{EncSim,DecSim}} \\
\text{return} & \ (M' = M) \\
\text{procedure} & \ \text{EncSim}(P) \\
M^* & \leftarrow \{0,1\}^{|P|+t}; \quad \text{return} \ \mathcal{E}_K(M^*) \\
\text{procedure} & \ \text{DecSim}(C) \\
\text{if} & \ \mathcal{D}_K(C) \neq \bot \ \text{then return} \ 1 \ \text{else return} \ 0
\end{align*}
\]

Figure 4.4: Games \( G_0 \) and \( G_1 \) for the proof that CCA security implies CPSS security.

CCA IMPLIES CPSS. We now show that CCA security implies CPSS security. Suppose that there’s an adversary \( A \) breaking CPSS security of \( \Pi \). Assume that if \( A \) receives a ciphertext \( C \) from its encryption oracle, it won’t query \( C \) to the decryption oracle, because it knows that it’ll get 1 anyway. We will construct an adversary \( B \) breaking CCA security of \( \Pi \). The adversary \( B^{\text{Enc,Dec}} \) first picks \( M \leftarrow \{0,1\}^t \) and runs \( A \) and provides the latter with access to two (simulated) oracles \( \text{EncSim} \) and \( \text{DecSim} \). Each time \( A \) queries \( \text{EncSim}(P) \), meaning that it wants an encryption of \( P \| M \), adversary \( B \) simply returns \( \text{Enc}(P||M) \). If \( A \) queries \( \text{DecSim}(C) \), \( B \) will query \( \text{Dec}(C) \). Adversary \( B \) will return 0 to \( A \) if it gets a \( \bot \)-answer, and return 1 otherwise. Due to the assumption above, \( B \) will never query a ciphertext \( C \) that it received from \( \text{Enc} \) to the oracle \( \text{Dec} \). Finally, when \( A \) outputs a message \( M' \), if \( M' = M \) then \( B \) outputs 1; otherwise it outputs 0. The code of \( B \) is given below.

**Adversary \( B^{\text{Enc,Dec}} \)**

\[
\begin{align*}
\text{M} & \leftarrow \{0,1\}^t; \quad \text{M'} \leftarrow A^{\text{EncSim,DecSim}} \\
\text{if} & \ (M' = M) \ \text{then return} \ 1 \ \text{else return} \ 0 \\
\text{procedure} & \ \text{EncSim}(P) \\
\text{return} & \ \text{Enc}(P\|M) \\
\text{procedure} & \ \text{DecSim}(C) \\
\text{if} & \ \text{Dec}(C) \neq \bot \ \text{then return} \ 1 \ \text{else return} \ 0
\end{align*}
\]

For analysis, consider games \( G_0 \) and \( G_1 \) in Figure 4.4. Game \( G_0 \) corresponds to game \( \text{REAL}^B_\Pi \), and the other corresponds to game \( \text{RAND}^B_\Pi \). Then

\[
\text{Adv}_\Pi^{\text{cca}}(B) = \Pr[G_0 \Rightarrow 1] - \Pr[G_1 \Rightarrow 1] .
\]

On the other hand, note that game \( G_0 \) also coincides with game \( \text{CPSS}^A_\Pi \). Hence

\[
\text{Adv}_\Pi^{\text{cpss}}(A) = \Pr[G_0 \Rightarrow 1] .
\]

Moreover, note that in game \( G_1 \), whatever \( A \) receives is independent of the message \( M \), yet \( A \) has to guess \( M \). Since \( M \) is uniformly distributed in \( \{0,1\}^t \), the chance that \( A \) can guess \( M \) correctly is at most \( 2^{-t} \). Hence

\[
\Pr[G_1 \Rightarrow 1] \leq 2^{-t} .
\]

Summing up, \( \text{Adv}_\Pi^{\text{cca}}(B) \geq \text{Adv}_\Pi^{\text{cpss}}(A) - 2^{-t} \). Moreover, adversary \( B \) is about as efficient as \( A \).

CHOPCHOP ATTACK ON WEP. The WEP construction doesn’t meet the CPSS notion, even if the adversary makes a single \( \text{Enc} \) in which the prefix is the empty string, that is, the adversary is given just a ciphertext of the secret \( M \). In the context of WiFi security, it means the adversary observes a ciphertext, and wants to recover the message, by using the Access Point as a decryption oracle. This is known as the ChopChop attack.
Understanding the ChopChop attack for WEP requires some knowledge of finite-field multiplication. Here we describe how to attack the following variant of WEP: instead of appending CRC($M$) to the message $M$, we will append the parity bit parity($M$), the xor of all the bits of $M$. For example, if $M = 1011$ then parity($M$) = 1. For simplicity, assume that this variant allows encrypting the empty string $\varepsilon$, with parity($\varepsilon$) = 0. The parity bit is a weak form of redundancy checking: if there’s one bit flip in $M ||$ parity($M$) then one can detect that.

Suppose that the secret $M = M_1 \cdots M_m$, where each $|M_i| = 1$. Initially, the adversary queries ENC($\varepsilon$), where $\varepsilon$ is the empty string, to receive a ciphertext $C_0$. Note that now the adversary knows $m$, by observing the length of the ciphertext $C_0$. It then chops the last bit of $C_0$ to obtain a string $C_1$, and queries $C_1$ to the decryption oracle. If the answer of the decryption oracle is 1 then $M_m = \text{parity}(M_1 \cdots M_{m-1})$. In other words, $M_1 \oplus \cdots \oplus M_m = 0$. Otherwise, if the answer of the decryption oracle is 0 then we must have $M_1 \oplus \cdots \oplus M_m = 1$. In other words, we have obtained $M_1 \oplus \cdots \oplus M_m$. By repeating the process above for “ciphertext” $C_1$ (instead of $C_0$), we obtain $M_1 \oplus \cdots \oplus M_{m-1}$, and so on. At the end, we have a system of $m$ linear equations of the variables $M_1, \ldots, M_m$, which we can solve to recover $M_1, \ldots, M_m$.

4.2 Pitfalls in implementing EtM

Suppose that one uses the Encrypt-then-MAC composition, in which the encryption scheme is CBC mode and the MAC is a good PRF $F$ such as Encrypted CBC-MAC. Suppose that when we first encrypt a one-block message $M$ via CBC, we obtain a ciphertext IV$||C$. A correct implementation should apply the MAC on the entire IV$||C$, meaning that the tag $T$ is $F_{K_m}(IV||C)$, and the ciphertext for the EtM composition is (IV$||C,T$). However, it is very common for people to (incorrectly) apply the MAC on just $C$, meaning $T \leftarrow F_{K_m}(C)$. This happened on ISO 1972 standard, and also in RNCryptor facility in iOS. This buggy implementation completely destroys authenticity. For example, given (IV$||C,T$) the adversary can create a new ciphertext ((IV $\oplus \Delta$)$||C,T$) for the EtM composition, where $\Delta$ is an arbitrary string. If we feed this ciphertext to the EtM decryption then it will say that this is a valid ciphertext, and the decrypted message is $M \oplus \Delta$.

4.3 Pitfalls in implementing MtE: Padding-oracle attacks

Recall that the MAC-then-Encrypt composition is not generally secure. One can come up with a simple but artificial counter-example: the encryption scheme adds some redundant bits to the ciphertexts, and decryption ignores those. But this says nothing about the security of MAC-then-Encrypt if you use some specific MAC and encryption schemes. We now study a well known attack on the authenticated encryption scheme in TLS 1.0, where the encryption scheme is CBC. The specific MAC doesn’t matter here; we can treat it as a good PRF $F : K \times \{0,1\}^* \rightarrow \{0,1\}^{128}$.

The scheme. In TLS, people only deal with byte strings, so the message space is $(\{0,1\}^8)^*$. Still, raw CBC can only handle strings whose byte lengths are multiple of 16. We therefore need a padding mechanism. The simplest way is to add $10^*\$, but TLS chooses a different method. If you want to add $p$ bytes to the message, the last byte must encode the number $p - 1$, and the remaining $p - 1$ bytes are arbitrary. For example, if your message has byte length 30, then you need to pad two bytes, the last one encoding 1. If your message has byte length 32, then you need to pad 16 bytes, the last one encoding 15. Given a message $M$, we write pad($M$) to denote the padding string.

Given a message $M$ and key $(K,K')$, one can MAC-then-Encrypt as follows. First, use the MAC to create a tag $T \leftarrow F_{K'}(M)$. Recall that the tag $T$ has byte length 16. Then use CBC with key $K$ to encrypt
ATTACKING CPSS SECURITY OF TLS ENCRYPTION. For simplicity, suppose that the byte length of $M$ is a multiple of 16. Let $M = M_1 M_2 \cdots M_m$, where each $|M_i| = 16$. We now recover the first block $M_1$ of $M$; the same method can be used to recover any block of $M$.

We first recover the last byte of $M_1$. Let $\varepsilon$ denote the empty string. The adversary first queries $\text{Enc}(\varepsilon)$, asking for an encryption of $M$. It then gets a CBC ciphertext $C = C_0 C_1 \cdots C_{m+2}$ for $M_1 \cdots M_m \parallel T \parallel \text{pad}(M)$, where $T \leftarrow F_{K'}(M)$ and $C_0$ is a random IV. Now modify the last block of $C$, replacing $C_{m+2}$ by $C_1$. Let $C'$ be the resulting ciphertext, and query $C'$ to $\text{Dec}$. Recall that in the implementation of $\text{Dec}$, the first step is to use CBC on key $K$ to decrypt $C'$, resulting in $M_1 \parallel \cdots \parallel M_m \parallel T \parallel V$, where $V = C_{m+1} \oplus E^{-1}_K(C_1) = C_{m+1} \oplus M_1 \oplus C_0$. See Figure 4.5 for an illustration.

After CBC-decrypting $C'$ to get $X \leftarrow M_1 \cdots M_m \parallel T \parallel V$, one would need to look at the last byte of $V$ to know how many bytes we need to truncate from $X$, before doing a tag comparison to check for validity. Note that (1) if the last byte of $V$ encodes 15 then the ciphertext $C'$ will be deemed valid, and (2) if the last byte of $V$ is not 15 then it’s unlikely that $C'$ is valid. Hence by observing the output of the decryption oracle, the adversary can tell if the last byte of $M_1 \oplus C_0 \oplus C_{m+1}$ is the encoding of 15. Hence with probability about 1/256, one can find the value of the last byte of $M_1$.

By repeating the whole process above, the adversary can check if the last byte of $M_1$ is the same as the last byte of another randomly chosen IV. If the adversary iterates $t$ times then it can find the exact value of the last byte of $M_1$ with probability around $1 - (1 - 1/256)^t \geq 1 - e^{-t/256}$.

So far we’ve learned just the last byte of $M_1$. To learn the second last byte of $M_1$, instead of querying $\text{Enc}(\varepsilon)$, we would query $\text{Enc}(0^8)$, asking for encryption of $0^8 \parallel M$. Let the answer be $C_0' \cdots C_{m+2}'$, where $C_0' \cdots C_{m+2}'$ to the decryption oracle. (Note that the all blocks except the last one are from the old ciphertext of $M$, not $0^8 \parallel M$.) In this process, we’ll learn the last byte of the first block of $0^8 \parallel M$, which is exactly the second last byte of $M_1$. We can do the same trick to learn any other byte in $M_1$.

A FIX AND ITS NEW PROBLEM. After the security disaster of TLS encryption, here’s what people actually tried to fix it. Now if you need to pad $p$ bytes, every single byte must encode $p - 1$. In decryption, after CBC-decryption, you need to check if the padding is well-formed. If yes, then we’ll do the tag checking.

Now there are two possible reasons that we might reject a ciphertext as invalid: (1) bad padding, and (2) bad tag. Some implementations actually explicitly tell users if their ciphertexts fall into (1) or (2). While this might be a good software engineering practice elsewhere, here it’s a security issue.
Now suppose that for the CPSS attack, $M = M_1 \cdots M_m$ with $m \geq 2$. Let’s try to recover the last byte of, say $M_2$. The adversary first queries $\text{Enc}(\varepsilon)$ to get a ciphertext $C_0C_1 \cdots C_{m+2}$ of $M$. Let $B$ be an arbitrary 16-byte string, and query $\text{Dec}(C')$, with $C' \leftarrow C_0||(C_1 \oplus B)||C_2$. After the CBC-decryption of $C'$, the last block of the decrypted string will be $(C_1 \oplus B) \oplus E_K^{-1}(C_2) = M_2 \oplus B$. If the last byte of $M_2 \oplus B$ encodes 0 then we would receive an error for bad tag. Otherwise, it’s likely that we would receive an error for bad padding. By trying this for many different $B$’s, eventually we’ll recover the last byte of $M_2$.

You might think that to fix the bug above, one should simply return the same error signal for both (1) and (2), but even then the problem still remains. For a naive implementation, if you have a bad padding then you’ll immediately reject the ciphertext without checking the tag, and if you have a well-formed padding, you’ll do a MAC for tag comparison. But then there’s quite a difference in the running time between the two types of error, and thus an adversary can still tell what kind of error it has. It requires some expertise to do a constant-time implementation to ensure that for both types of error, the running time is the same, especially when one also runs some compiler optimizations.