CIS5371 – Cryptography **Fall 2025**

Scribe 2: Real Attacks on AE Schemes

Instructor: Viet Tung Hoang

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.*

2.1 ChopChop Attack on WiFi Security

WEP CONSTRUCTION. The WEP construction for WiFi security is given in Figure 2.1. It's based on the streamcipher RC4. WEP is an instance of the encrypt-with-redundancy paradigm. Specifically, to encrypt a message M with $\mathrm{IV} \in \{0,1\}^{24}$, one first appends some CRC32 checking to M to obtain a string $X \leftarrow M \| \mathrm{CRC}(M)$, generates a one-time pad P from $\mathrm{RC4}(K, \mathrm{IV})$, and then outputs $\mathrm{IV} \| (X \oplus P)$.

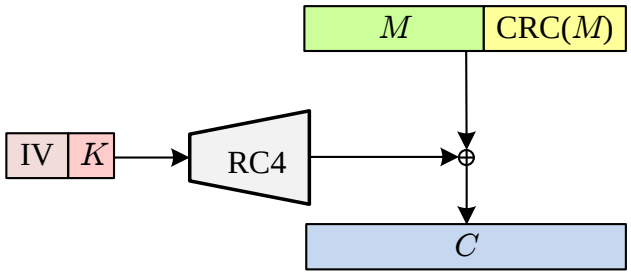

Figure 2.1: The WEP construction

ATTACK MODEL. We now describe a realistic scenario in which WEP is broken. Under our situation, an adversary observes a ciphertext C that a user sends to an access point, and wishes to recover the underlying message. The adversary will use the access point as a decryption oracle, querying several fake ciphertexts. For each query $\textsc{Dec}(C^*)$, the decryption oracle indicates whether the ciphertext C^* is valid, but nothing beyond it. That is, if C^* is valid, the adversary doesn't get to know the corresponding decrypted message.

CHOPCHOP ATTACK ON WEP. 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 $\mathrm{CRC}(M)$ to the message M, we will append the parity bit $\mathrm{parity}(M)$, the xor of all the bits of M. For example, if $M = 1011$ then $\mathrm{parity}(M) = 1$. For simplicity, assume that this variant allows encrypting the empty string ε, with $\mathrm{parity}(\varepsilon) = 0$. The parity bit is a weak form of redundancy checking: if there's one bit flip in $M \| \mathrm{parity}(M)$ then one can detect that.

Suppose that the secret $M = M_1 \cdots M_m$, where each $|M_i| = 1$. Initially, the adversary queries $\textsc{Enc}(\varepsilon)$, where ε 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 $M_1, \ldots, M_m$.

2.2 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 $\text{IV}\|C$. A correct implementation should apply the MAC on the entire $\text{IV}\|C$, meaning that the tag T is $F_{K_m}(\text{IV}\|C)$, and the ciphertext for the EtM composition is $(\text{IV}\|C, T)$. 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 $(\text{IV}\|C, T)$ the adversary can create a new ciphertext $((\text{IV} \oplus \Delta)\|C, T)$ for the EtM composition, where Δ 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 $M \oplus \Delta$.

2.3 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 $F : \mathcal{K} \times \{0,1\}^* \to \{0,1\}^{128}$.

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 $\text{pad}(M)$ 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 $M\|T\|\text{pad}(M)$.

Given a ciphertext C and key (K, K'), one first use CBC with key K to decrypt C to get a string X. Check the last byte of X to know how many more bytes we have to remove. Let the truncated string be $M\|T$. If $T \neq F_{K'}(M)$ then return $\perp$; otherwise return M.

ATTACK MODEL. We now define a security notion, called *Chosen Prefix Secret Suffix* (CPSS). Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be an authenticated encryption scheme. Under this attack, the game picks a secret message $M \xleftarrow{\$} \{0,1\}^\ell$ and key $K \xleftarrow{\$} \mathcal{K}$. 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 $\text{ENC}(P)$ to get back $\mathcal{E}_K(P\|M)$. On query

$\textsc{Dec}(C)$, if the ciphertext C is valid then the game returns 1 to the advesary, but the adversary doesn't get to see the decrypted message. Otherwise the game returns 0. Define

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

where game CPSS is defined in Figure 2.2.

Game CPSS^{A}_{Π}	**procedure** $\textsc{Enc}(P)$
$M \leftarrow\!\!\$\ \{0,1\}^{\ell}$; $K \leftarrow\!\!\$\ \mathcal{K}$	**return** $\mathcal{E}_K(P\|M)$
$M' \leftarrow A^{\textsc{Enc}(\cdot),\textsc{Dec}(\cdot)}$	**procedure** $\textsc{Dec}(C)$
return $(M' = M)$	**if** $\mathcal{D}_K(C) \neq \perp$ **then return** 1 **else return** 0

Figure 2.2: Game defining CPSS security.

The CPSS notion captures the following real-world scenario. In this setting, Alice first visits a legitimate website bank.com. Her browser then shares a secret cookie M with this website, and embeds this in every request it sends to bank.com. Suppose that later Alice is tricked into visiting a malicious website attacker.com. This site sends a Javascript program to the broswer, requesting some resource from bank.com. The browser will correspondingly send an encrypted HTTP GET request to bank.com. In other words, it sends $C \leftarrow\!\!\$\ \mathcal{E}_K(P\|M)$ 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 bank.com.

Attacking CPSS security of TLS encryption. For simplicity, suppose that the byte length of M is a multiple of 16. Let $M = M_1 M_2 \dots M_m$, 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 ε denote the empty string. The adversary first queries $\textsc{Enc}(\varepsilon)$, asking for an encryption of M. It then gets a CBC ciphertext $C = C_0 C_1 \cdots C_{m+2}$ for $M_1 \cdots M_m \| T \| \text{pad}(M)$, 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 Dec. Recall that in the implementation of Dec, the first step is to use CBC on key K to decrypt C', resulting in $M_1 \| \cdots \| M_m \| T \| V$, where $V = C_{m+1} \oplus E_K^{-1}(C_1) = C_{m+1} \oplus M_1 \oplus C_0$. See Figure 2.3 for an illustration.

After CBC-decrypting C' to get $X \leftarrow M_1 \cdots M_m \| T \| V$, 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 $\textsc{Enc}(\varepsilon)$, we would query $\textsc{Enc}(0^8)$, asking for encryption of $0^8 \| M$. Let the answer be $C'_0 \cdots C'_{m+2}$. Then, send $C_0 \cdots C_{m+1} C'_1$ to the decryption oracle. (Note that the all blocks except the last one are from the *old* ciphertext of M, not $0^8 \| M$.) In this process, we'll learn the last byte of the first block of $0^8 \| M$, 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

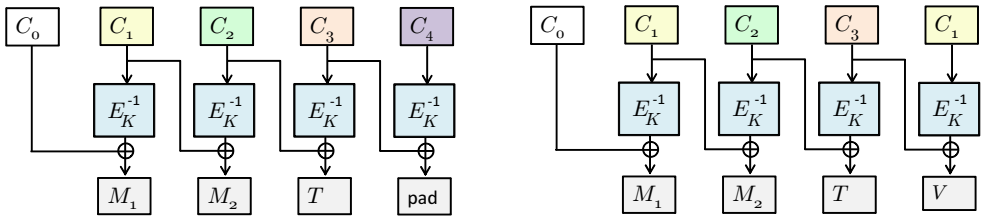

Figure 2.3: Illustration for the case $m = 2$ when we first make an encryption query on message $M = M_1 \cdots M_m$ to get ciphertext $C = C_0 C_1 \cdots C_{m+2}$, and then make a decryption query on $C' = C_0 C_1 \cdots C_{m+1} C_1$. On the right is the CBC-decryption of C' for $\textsc{Dec}(C')$. For comparison, on the left is what would happen for the CBC-decryption of C.

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 $\textsc{Enc}(\varepsilon)$ to get a ciphertext $C_0 C_1 \cdots C_{m+2}$ of M. Let B be an arbitrary 16-byte string, and query $\textsc{Dec}(C')$, 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.