CSCI 1515: Applied Cryptography

P. Miao

Spring 2023

These are lecture notes for CSCI 1515: Applied Cryptography taught at BROWN UNIVERSITY by Peihan Miao in the Spring of 2023.

These notes are taken by Jiahua Chen with gracious help and input from classmates. Please direct any mistakes/errata to me via email, post a thread on Ed, or feel free to pull request or submit an issue to the notes repository (https://github.com/BrownAppliedCryptography/notes).

FYI, todo's are marked like this.

Notes last updated February 7, 2023.

Contents

§1 February 7, 2023

§1.1 Message Integrity, *reviewed*

Last lecture, we discussed methods of authenticating a message. The symmetric-key version is called a MAC (message authentication code), the public-key version is called a digital signature. Let's review what we covered last time.

§1.1.1 Message Authentication Code

To authenticate a message, Alice will use the private key k to tag a message m with a tag t. Bob will verify that (m, t) is valid with key k.

diagram

§1.1.2 Digital Signature

In the public-key version, Alice has a secret key sk to sign message m with signature σ. Bob (or anyone) can verify with the public key pk that (m, σ) is a valid signature.

diagram

§1.1.3 Syntax

Recall the syntax of MAC and digital signatures (see **??**).

A message authentication code (MAC) scheme consists of $\pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Verify})$.

Generation. $k \leftarrow \mathsf{Gen}(1^\lambda)$.

Authentication. $t \leftarrow \mathsf{Mac}_k(m)$.

Verification $0/1 := \mathsf{Verify}_k(m, t)$.

A digital signature scheme consists of $\pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$.

Generation. $(sk, vk) \leftarrow \mathsf{Gen}(1^\lambda)$.

Authentication. $\sigma \leftarrow \mathsf{Sign}_{sk}(m)$.

Verification $0/1 := \mathsf{Verify}_{vk}(m, \sigma)$.

§1.1.4 Constructions

We can construct MAC pratically using

- Block Cipher: CBC-MAC
- Hash Functions: HMAC

We can construct digital signatures using

- RSA signature: RSA Assumption.
- DSA signature: Discrete Log Assumption
- Lattice-Based Encryption Schemes (post-quantum secure).

§1.2 RSA Signatures

Our RSA signatures algorithm works very similarly to RSA encryption.

We generate two n-bit primes p, q. Compute $N := p \cdot q$ and $\phi(N) = (p-1)(q-1)$. Again choose e with $\gcd(e, \phi(N)) = 1$ and invert $d = e^{-1} \mod \phi(N)$. Given N and a random $y \overset{\$}{\leftarrow} \mathbb{Z}_N^\times$, it's computationally hard to find x such that $x^e \equiv y \mod N$.

Similarly, $sk := d$ and $vk := (N, e)$. To sign, we compute

$$\mathsf{Sign}_{sk}(m) := m^d \mod N.$$

To verify, we compute

$$\mathsf{Verify}_{vk}(m, \sigma) := \sigma^e \overset{?}{\equiv} m \mod N.$$

Question. Are there any security issues with RSA as we have constructed it so far?

Thinking back to our definition of chosen-message attack, if Eve knows many messages and signatures

I missed this section to take a call will watch recording.

There is an easy solution, however. We can hash our message m before we sign, like so

$$\mathsf{Sign}_{sk}(m) := H(m)^d \mod N$$

$$\mathsf{Verify}_{vk}(m, \sigma) := \sigma^e \overset{?}{\equiv} H(m) \mod N$$

where H is a hash function[1]. This is a commonly known technique called 'hash-and-sign'.

[1]A hash function is, briefly, a function that produces some random output that is hard to compute the inverse of.

§1.3 DSA Signatures

DSA signatures are a bit more involved; they rely on the discrete log assumption and that it is hard to compute discrete logs in $\mathbb{Z}_p^\times$ and $\mathbb{Z}_q^\times$ a subgroup.

We give this definition using integer groups. There is also ECDSA which uses elliptic curve groups, which is used in cryptocurrencies[2]. Elliptic curves are much more efficient (especially when chosen correctly) and provide similar security guarantees.

$\mathsf{Gen}(1^k)$: We generate an n-bit[3] prime p and an m-bit[4] prime q such that $q \mid (p-1)$. We pick a random $\alpha \in \mathbb{Z}_p^\times$ such that $g = \alpha^{\frac{p-1}{q}} \mod p \neq 1$[5]

We pick $x \xleftarrow{\$} \mathbb{Z}_q^\times$ and compute $h = g^x \mod p$.

Our verification key is $vk := (p, q, g, h)$ and our signing key is $sk = x$.

Signing and verifying

§1.4 Authenticated Encryption

Generally, Alice and Bob will first perform a Diffie-Hellman key exchange, then use that shared key to conduct Symmetric-Key Encryption.

diagram

In reality, we want to achieve both message secrecy and integrity *at the same time*. For this, we can introduce Authenticated Encryption.

Our security definition is that our adversary can see the encryptions of many messages m_0, m_1, m_2. We want **CPA security**: given an encryption of either m_0, m_1, our adversary cannot distinguish between encryptions of the two. *Additionally*, we want the property of **unforgeability**, that our adversary cannot generate a c^* that is a valid encryption, such that $\mathsf{Dec}_k(c^*) \neq m_i$ for any i.

diagram

Now that we have two new primitives, we can construct Authenticated Encryption schemes.

[2]The specific curve used by Bitcoin, for example, is called **secp256k1**.
[3]Usually, 2048.
[4]Usually, 256. We assume discrete log in *both* of these groups.
[5]This means that $\langle g \rangle$ has order q, a subgroup in $\mathbb{Z}_p^{-1}$. $g^q \equiv \alpha^{p-1} \equiv 1 \mod p$.

§1.4.1 Encrypt-and-MAC?

Given a CPA-secure SKE scheme $\pi_1(\mathsf{Gen}_1, \mathsf{Enc}_1, \mathsf{Dec}_1)$ and a CMA-secure MAC scheme $\pi_2 = (\mathsf{Gen}_2, \mathsf{Mac}_2, \mathsf{Verify}_2)$.

We construct an AE scheme $\pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ by composing encryption and MAC. We encrypt first, then MAC the encrypted ciphertext.

Diagram

$\mathsf{Gen}(1^\lambda)$: $k_1 := \mathsf{Gen}_1(1^\lambda)$, $k_2 := \mathsf{Gen}_2(1^\lambda)$, output (k_1, k_2).

$\mathsf{Enc}(m)$: To encrypt, we first encrypt ciphertext $c_1 := \mathsf{Enc}_1(k_1, m)$ and sign message (in plaintext) $t_2 := \mathsf{Mac}_2(k_2, m)$ and output (c_1, t_2).

$\mathsf{Dec}(m)$: We have ciphertext $c = (c_1, t_2)$. Our message is $m := \mathsf{Dec}_1(k_1, c_1)$, and our verification bit $b := \mathsf{Verify}_1(k_2, (m, t_2))$. If $b = 1$, output m, otherwise we output $\perp$.

Question. Is this scheme secure? Assuming the CPA-secure SKE scheme and CMA-secure MAC scheme, does this give us both CPA-security and unforgeability?

MAC gives you *unforgeability*—it doesn't even try to hide the message at all. It's possible that the MAC scheme reveals the message in the clear. For example, we might have a MAC scheme that includes the message in the signature *in the clear* (which is still secure)!

Since MAC doesn't try to hide the message. If our MAC reveals something about our message, our composed scheme π doesn't give us CPA-security. You might still be able to infer something about the message.

We try something else...

§1.4.2 Encrypt-then-MAC

We encrypt first, then we MAC on the *ciphertext*.

diagram

$\mathsf{Gen}(1^\lambda)$: $k_1 := \mathsf{Gen}_1(1^\lambda)$, $k_2 := \mathsf{Gen}_2(1^\lambda)$, output (k_1, k_2).

$\mathsf{Enc}(m)$: To encrypt, we first encrypt ciphertext $c_1 := \mathsf{Enc}_1(k_1, m)$ and sign message (in plaintext) $t_2 := \mathsf{Mac}_2(k_2, c_1)$ and output (c_1, t_2).

$\mathsf{Dec}(m)$: We have ciphertext $c = (c_1, t_2)$. Our message is $m := \mathsf{Dec}_1(k_1, c_1)$, and our verification bit $b := \mathsf{Verify}_1(k_2, (c_1, t_2))$. If $b = 1$, output m, otherwise we output $\perp$.

You can prove that Encrypt-then-MAC schemes are CPA-secure and unforgeable.

§1.4.3 MAC-then-Encrypt

Similarly, we can also MAC first, encrypt the entire ciphertext and tag concatenated.

diagram

$\mathsf{Gen}(1^\lambda)$: $k_1 := \mathsf{Gen}_1(1^\lambda)$, $k_2 := \mathsf{Gen}_2(1^\lambda)$, output (k_1, k_2).

$\mathsf{Enc}(m)$: To encrypt, we first encrypt ciphertext $c_1 := \mathsf{Enc}_1(k_1, m)$ and sign message (in plaintext) $t_2 := \mathsf{Mac}_2(k_2, m||t_2)$ and output (c_1, t_2).

$\mathsf{Dec}(m)$: We have ciphertext $c = (c_1, t_2)$. Our message is $m||t_2 := \mathsf{Dec}_1(k_1, c_1)$, and our verification bit $b := \mathsf{Verify}_1(k_2, (m, t_2))$. If $b = 1$, output m, otherwise we output $\perp$.

Question. Is this secure?

This doesn't satisfy a stronger security definition called Chosen Ciphertext Attack (CCA) security. We might be able to forge ciphertexts that decrypt to valid message and tags.

§1.4.4 Chosen Ciphertext Attack Security

On top of CMA security, the adversary can now request Alice to decrypt ciphertexts $c_0, c_1, \ldots$.

diagram

We can prove that MAC-then-Encrypt is not CCA secure.

The moral of this is that **you should always use Encrypt-then-MAC.**

§1.5 A Summary So Far

To summarize, here's all we've covered so far:

	Symmetric-Key	Public-Key
Message Secrecy	Primitive: SKE Construction: Block Cipher	Primitive: PKE Constructions: RSA/ElGamal
Message Integrity	Primitive: MAC Constructions: CBC-MAC/HMAC	Primitive: Signature Constructions: RSA/DSA
Secrecy & Integrity	Primitive: AE Construction: Encrypt-then-MAC	
Key Exchange		Construction: Diffie-Hellman
Important Tool	Primitive: Hash function Construction: SHA	

§1.6 Hash Function

A hash function is a public function

$$H : 0, 1^* \to 0, 1^n$$

where n is order $\Theta(\lambda)$.

diagram

We want our hash function to be collision-resistant. That is, it's computationally hard to find $x, y \in \{0,1\}^*$ such that $x \neq y$ yet $H(x) = H(y)$ (which is called a collision).

How might one find a collision for function $H : \{0,1\}^* \to \{0,1\}^n$. We can try $H(x_1), H(x_2), \dots, H(x_q)$.

If $H(x_1)$ outputs a random value, $0, 1^n$, what is the probability of finding a collision?

If $q = 2^n + 1$, our probability is exactly 1 (by pigeon-hole). If $q = 2$, our probability is $\frac{1}{2^\lambda}$ (we have to get it right on the first try). What q do we need for a 'reasonable' probability?

> **Remark.** This is related to the birthday problem. If there are q students in a class, assume each student's birthday is a random day $y_i \xleftarrow{\$} [365]$. What is the probability of a collision? $q = 366$ gives 1, $q = 23$ gives around 50%, and $q = 70$ gives roughly 99.9%.
>
> We can apply this trick to our hash function. If $y_i \xleftarrow{\$} [N]$, then $q = N + 1$ gives us 100%, but $q = \sqrt{N}$ gives 50% probability.

Knowing this, we want $n = 2\lambda$ (output length of hash function). If $\lambda = 128$, we want n to be around 256.

§1.6.1 Random Oracle Model

Another way to model a hash function is the *Random Oracle Model.* We think of our hash function to be an oracle (in the sky) that can *only* take input and a random output (and if you give it the same input twice, the same output).

graphic

There are proofs that state that no hash functions can be a random oracle. There are schemes that can be secure in the random oracle model, but are not using hash functions[6].

In reality, hash functions are *about as good as*[7] random oracles. Thinking of our hash functions as random oracles gives us a good intuitive understanding of how hash functions can be used in our schemes.

In this model, the best thing that an attacker can do is to try inputs and query for outputs.

§1.6.2 Constructions for Hash Function

MDS. Output length 128-bit. Best known attack is in 2^{16}. A collision was found in 2004.

And we also have Secure Hash Functions (SHA), founded by NIST.

SHA-0. Standardized in 1993. Output length is 160-bit. Best known attack is in 2^{39}.

SHA-1. Standardized in 1995. Output length is 160-bit. Best known attack is in 2^{63}, and a collision *was* found in 2017.

SHA-2. Standardized in 2001. Output length of 224, 256, 284, 512-bit. The most commonly used is SHA-256.

SHA-3. There was a competition from 2007-2012 for new hash functions. SHA-3 was released in 2015, and has output length 224, 256, 2384, 512-bit. This is *completely different* from SHA-2.

> **Remark.** The folklore is that during a session at a cryptography conference, a mathematician, Xiaoyun Wang, presented slide-after-slide of attacks on MDS and SHA-0, astounding the audience.

[6]Some constructions don't rely on this model.
[7]But can never be...