CS4236 Cryptography Theory and Practice Topic 12 - Signatures and side channels

Hugh Anderson

National University of Singapore
School of Computing

November, 2022

Signatures...

Berlioz
Mahler
Schumann
Chopin
Haydn
Wagner
Mozart
Shostakovich
Vivaldi
Liszt
Beethoven
Ravel
Schubert
PICCOLLAGE

Outline

The story so far ... where are we?

The last 11 weeks: weekly steps we have taken

1. We had a historical/contextual introduction in Session 1.
2. We progressed from perfect *secrecy* to perfect *indistinguishability*.
3. *Perfectly* indistinguishable was relaxed to give *computationally* indistinguishable. An EAV-Secure system was constructed with a PRG.
4. The notion of CPA-secure was developed, and achieved with a PRF.
5. Modes, the “padding oracle”, and CCA-Security were outlined.
6. We looked at cryptographic MACs and *authenticated* encryption.
7. We began looking at hashes, with definitions, games, and applications.
8. We then looked at birthday attacks on hashes, and rainbow tables.
9. We looked at SP networks, and cryptanalysis of them.
10. We looked at definitions of PK systems, and the RSA construction.
11. Last week, we continued investigating PK systems based on Dlog: Key agreement, Elgamal, ECC and bilinear maps.

Public key variants of MAC schemes

Overview: MAC (Symmetric) vs SIG (Asymmetric)

The MAC systems we have seen before have symmetric keys: the same key to generate the mac, and to verify the tag. The corresponding system for public key systems would use asymmetric keys: private key to generate the signature, and the public key to verify the signature.

As for MAC, the security requirement is on forgery: the adversary can adaptively obtain pairs of valid message and signature, and yet should be unable to forge a valid and unseen pair.

A signature can also achieve non-repudiation. That is, a signer cannot deny that he/she had signed a message.

Definition

Definition 12.1. Signature $\mathrm{SIG}_{p_k} = (\mathrm{Gen}, \mathrm{Sign}, \mathrm{Vrfy})$

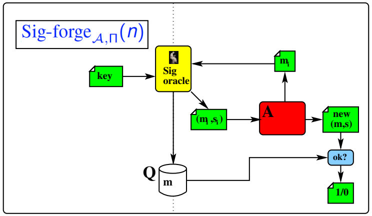
$\mathrm{Gen}(1^n)$:	Returns signing and verifying keys $\langle s_k, v_k \rangle$.
$\mathrm{Sign}_{s_k}(m)$:	Input message m and key s_k. Output: $(m, s) \leftarrow \mathrm{Sign}_{s_k}(m)$
$\mathrm{Vrfy}_{v_k}(m, s)$:	Verifies and outputs valid (1) or invalid (0). We will write this as $b := \mathrm{Vrfy}_{v_k}(m, s)$.

See pg 442 for details.

How to pose SIG security properties as a game...

An adversary has an oracle $O = \mathrm{Sign}_{s_k}(\cdot)$ that will generate a valid pair (m, s) of any query message m. The first security requirement is easier/weaker - can the adversary forge any message pair (m, s) where the message m was not seen before. This is called existential forgery under a chosen message, as we saw before.

Security requirement: Existential forgery



The first SIG game: $\text{Sig-forge}_{\mathcal{A},\Pi}$ (Secure Sig)

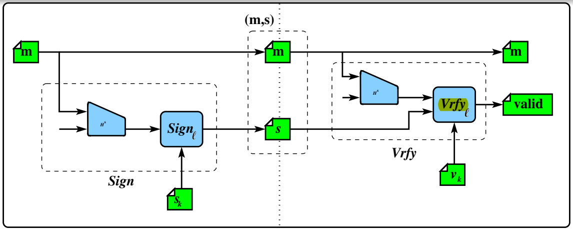
1. Adversary has oracle access. Adversary outputs (m, s). Let $\mathcal{Q}$ be all the messages the adversary had sent to the oracle in this experiment.
2. Adversary wins (output of experiment is 1) iff (m, s) is valid and $m \notin \mathcal{Q}$.

Definition 12.2 Existential unforgeability

We say that Π is existentially unforgeable under an adaptive chosen-message attack, or simply secure, iff for any $\mathcal{A}$, there is a negl, s.t.

$$\Pr[\text{Sig-forge}_{\mathcal{A},\Pi}(n) = 1] \leq \text{negl}(n)$$

Construction #1: Hash and sign



Construction 12.3: Hash-and-sign,..

Most practical constructions employ the hash-and-sign paradigm.

Suppose we have

1. a construction Sign_ℓ that takes in fixed size input, say ℓ bits, and
2. a collision resistant $\mathcal{H}$ that outputs an ℓ-bit digest.

We extend the sign to arbitrary input m by $\mathrm{Sign}(m) = \mathrm{Sign}_\ell(\mathcal{H}(m))$.

Theorem 12.4

if Sign_ℓ is a secure fixed-size signature construction, and $\mathcal{H}$ is collision resistant, then Sign is secure (for arbitrary size input).

Textbook RSA as a fixed size signature?

Construction #2: (an insecure construction)...

RSA could be used as a signature, where “signing” is encryption with the secret key, and “verifying” is decryption with the public key.

$\mathrm{Sign}_{s_k}(m)$: Given $m, \langle N, d \rangle$, output: $(m, s) \leftarrow (m, m^d \bmod N)$

$\mathrm{Vrfy}_{p_k}(m, s)$: Given $(m, s), \langle N, e \rangle$, if $m = s^e \bmod N$ then valid.

The above is not secure at all. There is a common misconception that a secure signature scheme can be achieved via “encryption” in this way:

1. Signing is done by “encrypting” the message using private key,
2. Verification is done by “decrypting” the signature using public key. [a]

This technique might not be secure. (Some textbooks use the term “encryption/decryption” in describing signatures. This is very misleading). Perhaps you can consider the effect of homomorphic properties on this.

[a]RSA has an interesting property that we can use private key for encryption and public key for decryption. This is not true for other PKC.

Construction #3: Textbook RSA, Hash-and-sign

Construction 12.6: Textbook RSA in hash-and-sign

If we employ textbook RSA in hash-and-sign, then it is secure. Key point: a secure (random oracle) hash is required.

$\mathrm{Sign}_{s_k}(m)$**:** Given $m, \langle N, d\rangle$, output: $(m, s) \leftarrow (m, \mathcal{H}(m)^d \bmod N)$

$\mathrm{Vrfy}_{p_k}(m, s)$**:** Given $(m, s), \langle N, e\rangle$, if $\mathcal{H}(m) = s^e \bmod N$ then valid.

Theorem 12.7: It can be shown that, if the RSA problem is hard, and $\mathcal{H}$ is a random oracle, then Construction 12.6 is a secure signature scheme. (Theorem 12.4, standard model, does not apply here)

Discrete Log based signature schemes

DSA is a fixed-size signature scheme. Together with a collision resistant hash, we can extend it to arbitrary message lengths. For a DL based signature, a group on an Elliptic Curve can be deployed.

Interestingly we can also build a signature scheme from a collision resistant hash (one-way is sufficient) and there isn't "Algebra" involved. However, each private key can only be used to generate a small number of signatures. See Lamport's Signature Scheme (pg 461).

Public Key Infrastructure

What is the purpose of PKI? To securely distribute a public key.

Emphasizing some properties

A common misconception about symmetric/public keys is that:

> *For symmetric keys, we need a secure channel to distribute the key from Alice to Bob. (ok)*
> *For public keys, we don't need a secure channel to distribute the key from Alice to Bob. (wrong)*

For both symmetric and public keys, we need a secure means to distribute the keys. The difference is:

> *Symmetric key: We need a secure channel for every pair.*
> *Public key: We need a secure broadcast channel for each entity to broadcast its public key.*

For public keys, because only a broadcast channel is needed, Alice doesn't need to know the recipients when she broadcast her key. (imagine that Alice is a website. The website does not know who is going to visit the website).

PKI and certificates are a means to securely broadcast public keys. We will not get into details.

Construction #4: Elgamal-style digital signature

Elgamal signature construction (not in book)

The algorithms are:

$\mathrm{Gen}(1^n)$**:** Signing key is $\langle \mathcal{G}, g, x \rangle$, verifying key is $\langle \mathcal{G}, g, h \rangle$. $(h = g^x)$.

$\mathrm{Sign}_{s_k}(m)$**:** Choose r with $\gcd(r, q-1) = 1$. Compute r^{-1}, $S_1 = g^r$, and $S_2 = r^{-1}(\mathcal{H}(m) - xS_1)$. Sig is: $(m, s) \leftarrow (m, \langle S_1, S_2 \rangle)$

$\mathrm{Vrfy}_{v_k}(m, s)$**:** Compute $V_1 = g^{\mathcal{H}(m)}$, $V_2 = h^{S_1} S_1^{S_2}$. Valid if $V_1 = V_2$.

Correctness

From the calculation of S_2 we have that $\mathcal{H}(m) = S_2 r + xS_1$ and so

$$\begin{aligned} V_1 = g^{\mathcal{H}(m)} &= g^{S_2 r + xS_1} \\ &= (g^r)^{S_2} (g^x)^{S_1} \\ &= (S_1)^{S_2} (h)^{S_1} = V_2 \end{aligned}$$

Elgamal signing is probabilistic. There are many valid signatures for a same message. Because it is probabilistic, two signatures of a same message (by the same signer) most probably are different. DSA is a variant of Elgamal.

Side channel attacks

Sometimes there are alternative routes...

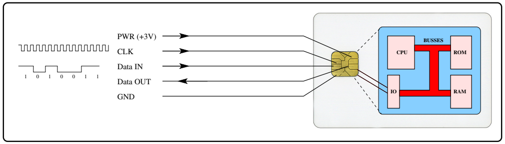
We have rigorously formulated the security of cryptographic primitives, for example CCA-secure for encryption, forgery for signatures and so on. In particular, the notion of an “experiment” captures the capability and goal of an adversary, and indistinguishability. We had also considered the strongest forms of attack, with access to oracles. and if a scheme is provably secure under the rigorous formulation, could it be broken in practical scenarios?

Attackers do not confine themselves to the playground decided by the designer. They can attempt to get out of the box, and gain capability and information not anticipated by the designer. In such scenarios, we typically call it a side-channel attack.

Many examples of side-channel attacks rely on physical information, for example the power usage during encryption, the time taken to encrypt and so on. It could also be software based: the amount of memory used, page faults. We will look at a few examples.

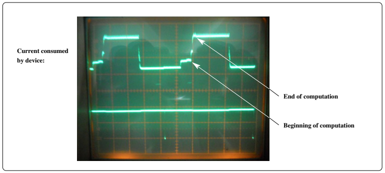
Typical hardware on a Smart/SIM card

A high-security Smart/SIM card controller



CPU, memory and IO are under the connector. They are all in one chip, connected externally with the connector.

Observe behaviour externally by measuring current...



Current trace gives evidence about computation - type and timing.

Timing attack on exponentiation

Efficient exponentiation

The first example of a timing attack on exponentiation (as in RSA) was introduced by Paul Kocher in 1995, while he was an undergraduate at Stanford. The attack attempts to find the key k, and exploits how efficient exponentiation is carried out.

Note that RSA does repeated multiplication, and Elgamal uses point addition, an analog of exponentiation. The algorithms use doubling and look the same:

RSA

```
To compute $c^d \bmod N$ in RSA
Q = 1;
foreach bit in key d {
    Q = Q*Q mod N;
    if (bit==1) {
        Q = Q*c mod N;
    }
}
return Q;
```

ECC

```
To compute $kP$ in ECC
Q = O;
foreach bit in key k {
    Q = 2Q;
    if (bit==1) {
        Q = Q + P;
    }
}
return Q;
```

Attacker's capability and goal

Assumptions on the attacker's capabilities

Consider the RSA algorithm, and multiplication. For any 2 integers x, y, the adversary can determine the time taken to compute $x \times y \bmod N$. The adversary can send any c to the decryption oracle and measure the time taken by the oracle. That is, the time taken to compute $c^d \bmod N$

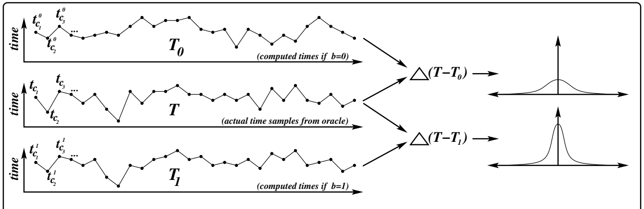
The main idea

First, note that for different values of the bits in the key, the loop takes different times. It takes longer if the bit is a 1.

The attacker sends large numbers of randomly drawn ciphertexts to the oracle, and measures the time taken for each computation. The attacker simulates the behaviour of the oracle for $b_0 = 0, b_0 = 1$ and tries to find timing correlations with the actual measured time. Once it has a best guess for b_0 it moves on to b_1 and so on through the bits, an iterative process.

`https://paulkocher.com/doc/TimingAttacks.pdf`

Side-channel attack #1: timing attack



Real and computed times for particular samples

Midway through the attack

The attacker knows $b_0, \ldots b_4$, and is trying to discover b_5 by repeatedly querying the oracle with a new c, and getting the time t. The adversary then

- Determines $t_{0\ldots4}$, with knowledge of the algorithm, and $b_0, \ldots b_4$.
- Determines t^0_5, the time if the bit in the key was 0.
- Determines t^1_5, the time if the bit in the key was 1.

Now either $t = t_{0\ldots4} + t^0_5 + x$ or $t = t_{0\ldots4} + t^1_5 + x$ (along with noise). The attacker builds two distributions T_0 and T_1. If the oracle's actual-time distribution T correlates better with T_0, then $b_5 = 0$, otherwise $b_5 = 1$.

The algorithm continues until all bits of the key are retrieved.

What do we do about this attack?

The Montgommery ladder:

Both functions return the same result as before, but now they run in a constant time. Note that RSA still does multiplication, and Elgamal still uses point addition, and the algorithms still use doubling and look the same:

RSA

```
To compute $c^d$ mod $N$ in RSA
R0 = c;
R1 = c * c mod N;
foreach bit in key d {
    if (bit==0) {
        R1 = R0 * R1 mod N;
        R0 = R0 * R0 mod N;
    } else {
        R0 = R0 * R1 mod N;
        R1 = R1 * R1 mod N;
    }
}
return R0;
```

ECC

```
To compute $kP$ in ECC
R0 = O;
R1 = P;
foreach bit in key k {
    if (bit==0) {
        R1 = R0 + R1;
        R0 = 2R0;
    } else {
        R0 = R0 + R1;
        R1 = 2R1;
    }
}
return R0;
```

.

Attack involves ECC, memory caches, timing...

OpenSSL: An important bit of security software

OpenSSL is used in about 60% of all web based ssl (https) servers. As a result of this we are *very* interested in bugs in openssl. It may be the software most closely examined for security errors - all 700,000 lines of code!

What is the (one second) attack?

From the paper at `http://eprint.iacr.org/2014/140.pdf`.

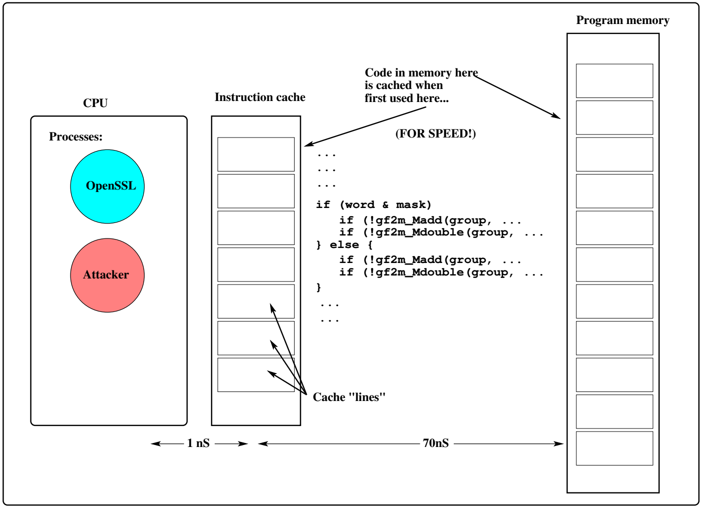
> *"Our attack recovers the scalar k and thus the secret key of the signer and would therefore allow unlimited forgeries."*

The attack is a side channel attack. An attacker can determine if code has been cached or not. If it has been used, then it will be cached.

OpenSSL is built with security in mind, so (of course) it uses the Montgommery ladder, that runs in constant time. Weirdly, the attack is a *timing* attack on the Montgomery ladder!

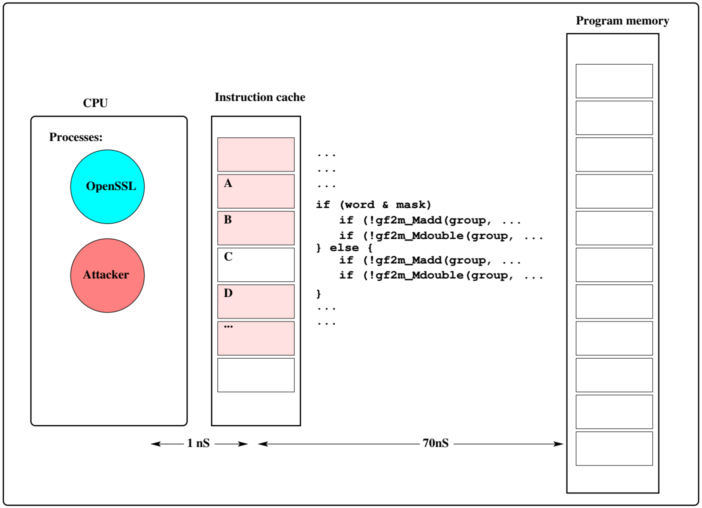
Side-channel attack #2: ...instruction cache...

As openssl program runs, it caches the code...



```
...
...
...
if (word & mask)
    if (!gf2m_Madd(group, ...
    if (!gf2m_Mdouble(group, ...
} else {
    if (!gf2m_Madd(group, ...
    if (!gf2m_Mdouble(group, ...
}
 ...
 ...
```

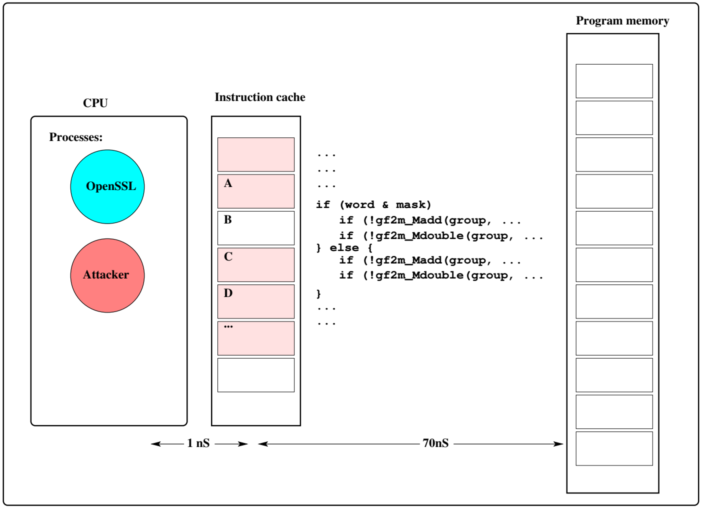
If bit is a “1” THIS is cached...



```
...
...
...
if (word & mask)
    if (!gf2m_Madd(group, ...
    if (!gf2m_Mdouble(group, ...
} else {
    if (!gf2m_Madd(group, ...
    if (!gf2m_Mdouble(group, ...
}
...
...
```

(and the attacker can determine this by seeing how fast the four cachelines A,B,C,D are loaded)

If bit is a “0” THIS is cached...



```
...
...
...
if (word & mask)
    if (!gf2m_Madd(group, ...
    if (!gf2m_Mdouble(group, ...
} else {
    if (!gf2m_Madd(group, ...
    if (!gf2m_Mdouble(group, ...
}
...
...
```

(and the attacker can determine this by seeing how fast the four cachelines A,B,C,D are loaded)

OpenSSL timing

Original openssl-1.0.1f/crypto/ec/ec2_mult.c...

```
if (word & mask) {
   if (!gf2m_Madd(..., x1, ... ) goto err;
   if (!gf2m_Mdouble(..., x2, ... ) goto err;
} else {
   if (!gf2m_Madd(..., x2, ... ) goto err;
   if (!gf2m_Mdouble(..., x1, ... ) goto err;
}
```

Each bit of a key is being tested, and depending on the bit we take two different paths. The paths operate on different (code) memory locations which would be cached, and an attacker can (afterwards) discover that, because that memory location would load faster. An attacker can determine the value of each bit.

Fixed openssl-1.0.1g/crypto/ec/ec2_mult.c...

```
BN_consttime_swap(word & mask, x1, x2,...
if (!gf2m_Madd(..., x1, ... ) goto err;
if (!gf2m_Mdouble(..., x2, ... ) goto err;
BN_consttime_swap(word & mask, x1, x2,...
```

A constant time swap to put variables in the same memory locations.

Fault injection

The idea of getting secrets through injecting faults

This idea was first discussed in D Boneh, RA DeMillo, RJ Lipton, *“On the importance of checking cryptographic protocols for faults”*, CRYPTO, 1997.

The adversary observes the behavior of a device when some components are faulty. From the behavior, the adversary infers a secret.

The fault could be injected by the adversary. Very often, although the adversary can inject a fault, the adversary cannot predict the outcome. For example, the adversary can flip some bits in a particular register but does not know which bits will be flipped and be unable to read the register.

Using CRT in RSA to speed up decryption about $4\times$

Rather than computing $c^d \bmod N$, it is possible to instead compute $e_1 = c^d \bmod p$, and $e_2 = c^d \bmod q$, and then

$$c^d \bmod N = (ae_1 + be_2) \bmod N$$

where $a = 1 \bmod p, a = 0 \bmod q, b = 0 \bmod p, b = 1 \bmod q$ (pre-computed). This is possible because the factorization of N could be kept around.

Side-channel attack #2: Fault injection

If exponentiation is using CRT...

Assume the attacker can obtain the output $m = (ae_1 + be_2) \bmod N$.

The attacker can introduce noise/errors during the computation of e_1, and no error during the computation of e_2. Hence the attacker obtains

$$m' = (ae_1' + be_2) \bmod N$$

Now, the attacker computes $\gcd(m – m', N)$. The gcd is either p or q.

Why does it work?

Note that $m - m' = a(e_1 - e_1') \bmod N$, and that $a = 0 \bmod q$ (but is not zero). If $m – m'$ is not divisible by p, then

$$\gcd(m - m', N) = q$$

Side channel attacks on crypto

Summary

These attacks are very real, and here we have only looked at a few of the known attacks. In general side-channel attacks are implementation or system dependant, not related to the *hardness* of the crypto problem. Caches, timing, power consumption, sound, light, radio waves, faults and so on - all have been used.

In any case they have been shown to achieve success with vastly smaller number of trials/attacks, than would be expected through (for example) root finding, discrete logs or factorization.

Though the underlying mathematical structures seem good, to implement crypto we must think out of the box, like the attackers, and handle unexpected side channel attacks.

Symmetric encryption schemes

$\mathrm{ENC}_k = (\mathrm{Gen}(1^n), \mathrm{Enc}_k(m), \mathrm{Dec}_k(c))$ **(ch1)**

Properties	Defs	Constructions	Proofs
Perfect secrecy	2.3	(One time pad)	Thm 2.9
$\updownarrow$			Lemma 2.6
Perfect indistinguishability	2.5		
$\downarrow$			
EAV-secure	3.8	3.17 (PRG)	Thm 3.18
$\uparrow$			Proof given
EAV-Multi-encryption	3.19		
$\uparrow$			
CPA-Multi-encryption	3.22		
$\updownarrow$			Thm 3.24
CPA-secure	3.23	3.30 (PRF)	Thm 3.31
$\uparrow$			
CCA-secure	3.33		
$+$			
Unforgeable	4.16		
$\updownarrow$			Thm 4.19
Authenticated	4.17	4.18 (Enc-then-Auth)	

MAC and HASH schemes

$$\mathrm{MAC}_k = (\mathrm{Gen}(1^n), \mathrm{Mac}_k(m), \mathrm{Vrfy}_k(m, t)) \quad \textbf{(4.1)}$$

Properties	Defs	Constructions	Proofs
Unforgeable	4.2		
$\uparrow$			Discussed
Strong	4.3	4.5 (Using PRF)	Thm 4.5
		4.7 (Variable length)	Thm 4.8
		4.11 (CBC-MAC)	Thm 4.12
		4.11(a) (Variable CBC-MAC)	

$$\mathrm{HASH} = (\mathrm{Gen}(1^n), \mathcal{H}^s(x)) \quad \textbf{(5.1)}$$

Properties	Defs	Constructions	Proofs
CollisionResistant	5.2	5.3 (Long message)	

Asymmetric encryption schemes

$$\text{ENC}_{p_k} = (\text{Gen}(1^n), \text{Enc}_{p_k}(m), \text{Dec}_{s_k}(c)) \quad \textbf{(11.1)}$$

Properties	Defs	Constructions	Proofs
		11.26 (Textbook RSA) (Elgamal) (ECC)	Thm 11.18
DDH-hard Factor-hard $\uparrow$ RSA-hard	8.63 8.45 8.46		
EAV-secure $\updownarrow$ CPA-secure $\uparrow$ CCA-secure	11.2	11.32 (1-Bit RSA)	

Commitment, KEM, IBE schemes

$\mathrm{COM} = (\mathrm{Setup}(1^n), \mathrm{Commit}(a), \mathrm{Open}(c_a))$ **(ch5)**

Properties	Defs	Constructions	Proofs
Secure (Binding+Hiding)	5.13	(Hash based system)	

$\mathrm{KEM}_{p_k} = (\mathrm{Gen}(1^n), \mathrm{Encaps}_{p_k}(1^n), \mathrm{Decaps}_{s_k}(c))$ **(11.9)**

Properties	Defs	Constructions	Proofs
CPA-secure		11.10 (CPA_KEM+EAV_ENC)	Thm 11.12
CCA-secure	11.13	11.10 (CCA_KEM+CCA_ENC)	Thm 11.13
		11.37 (CCA_KEM+TextbookRSA)	

$\mathrm{IBE}_{p_k} = (\mathrm{Setup}(1^n), \mathrm{Enc}_{\mathrm{nm},M_P}(m), \mathrm{Dec}_{k_U,M_P}(c))$ **(...)**

Properties	Defs	Constructions	Proofs
		(Boneh/Franklin)	

KA and Signature schemes

$$\mathrm{KA} = (\mathrm{Init}(1^n, \overline{p}), \mathrm{DoProt}(\overline{p})) \tag{...}$$

Properties	Defs	Constructions	Proofs
KA-EAV-secure	10.1	(DHKE)	Thm 10.3
		3-party Bilinear maps	

$$\mathrm{SIG}_{p_k} = (\mathrm{Gen}(1^n), \mathrm{Sign}_{s_k}(m), \mathrm{Vrfy}_{v_k}(m, t)) \tag{12.1}$$

Properties	Defs	Constructions	Proofs
Unforgeable	12.2	12.3 (Hash and Sign)	Thm 12.4
		12.36 (Hash and Sign, RSA)	
		(Elgamal Sign)	