14 Public-Key Encryption

So far, the encryption schemes that we've seen are **symmetric-key** schemes. The same key is used to encrypt and decrypt. In this chapter we introduce **public-key** (sometimes called *asymmetric*) encryption schemes, which use different keys for encryption and decryption. The idea is that the encryption key can be made *public*, so that anyone can send an encryption to the owner of that key, even if the two users have never spoken before and have no shared secrets. The decryption key is private, so that only the designated owner can decrypt.

We modify the syntax of an encryption scheme in the following way. A public-key encryption scheme consists of the following three algorithms:

KeyGen: Outputs a *pair* (pk, sk) where pk is a public key and sk is a private/secret key.

Enc: Takes the public key pk and a plaintext m as input, and outputs a ciphertext c.

Dec: Takes the secret key sk and a ciphertext c as input, and outputs a plaintext m.

We modify the correctness condition similarly. A public-key encryption scheme satisfies *correctness* if, for all $m \in \mathcal{M}$ and all $(pk, sk) \leftarrow \mathsf{KeyGen}$, we have $\mathsf{Dec}(sk, \mathsf{Enc}(pk, m)) = m$ (with probability 1 over the randomness of Enc).

14.1 Security Definitions

We now modify the definition of CPA security to fit the setting of public-key encryption. As before, the adversary calls a CHALLENGE subroutine with two plaintexts — the difference between the two libraries is which plaintext is actually encrypted. Of course, the encryption operation now takes the public key.

Then the biggest change is that we would like to make the public key *public*. In other words, the calling program should have a way to learn the public key (otherwise the library cannot model a situation where the public key is known to the adversary). To do this, we simply add another subroutine that returns the public key.

Definition 14.1 *Let Σ be a public-key encryption scheme. Then Σ is **secure against chosen-plaintext at-***

tacks (CPA secure) *if* $\mathcal{L}^{\Sigma}_{\text{pk-cpa-L}} \approx \mathcal{L}^{\Sigma}_{\text{pk-cpa-R}}$, *where:*

$\mathcal{L}^{\Sigma}_{\text{pk-cpa-L}}$

$(pk, sk) \leftarrow \Sigma.\text{KeyGen}$

GETPK():
  return pk

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
  return $\Sigma.\text{Enc}(pk, m_L)$

$\mathcal{L}^{\Sigma}_{\text{pk-cpa-R}}$

$(pk, sk) \leftarrow \Sigma.\text{KeyGen}$

GETPK():
  return pk

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
  return $\Sigma.\text{Enc}(pk, m_R)$

to-do *Re-iterate how deterministic encryption still can't be CPA-secure in the public-key setting.*

Pseudorandom Ciphertexts

We can modify/adapt the definition of pseudorandom ciphertexts to public-key encryption in a similar way:

Definition 14.2 *Let* Σ *be a public-key encryption scheme. Then* Σ *has* ***pseudorandom ciphertexts in the presence of chosen-plaintext attacks (CPA$ security)*** *if* $\mathcal{L}^{\Sigma}_{\text{pk-cpa\$-real}} \approx \mathcal{L}^{\Sigma}_{\text{pk-cpa\$-rand}}$, *where:*

$\mathcal{L}^{\Sigma}_{\text{pk-cpa\$-real}}$

$(pk, sk) \leftarrow \Sigma.\text{KeyGen}$

GETPK():
  return pk

CHALLENGE($m \in \Sigma.\mathcal{M}$):
  return $\Sigma.\text{Enc}(pk, m)$

$\mathcal{L}^{\Sigma}_{\text{pk-cpa\$-rand}}$

$(pk, sk) \leftarrow \Sigma.\text{KeyGen}$

GETPK():
  return pk

CHALLENGE($m \in \Sigma.\mathcal{M}$):
  $c \leftarrow \Sigma.C$
  return c

As in the symmetric-key setting, CPA$ security (for public-key encryption) implies CPA security:

Claim 14.3 *Let* Σ *be a public-key encryption scheme. If* Σ *has CPA$ security, then* Σ *has CPA security.*

The proof is extremely similar to the proof of the analogous statement for symmetric-key encryption (Theorem 7.3), and is left as an exercise.

14.2 One-Time Security Implies Many-Time Security

So far, everything about public-key encryption has been directly analogous to what we've seen about symmetric-key encryption. We now discuss a peculiar property that is different between the two settings.

In symmetric-key encryption, we saw examples of encryption schemes that are secure when the adversary sees only one ciphertext, but insecure when the adversary sees more ciphertexts. One-time pad is the standard example of such an encryption scheme.

Surprisingly, if a *public-key* encryption scheme is secure when the adversary sees just one ciphertext, then it is also secure for many ciphertexts! In short, there is no public-key one-time pad that is weaker than full-fledged public-key encryption — there is public-key encryption or nothing.

To show this property formally, we first adapt the definition of one-time secrecy (Definition 2.8) to the public-key setting. There is one small but important technical subtlety: in Definition 2.8 the encryption key is chosen at the last possible moment in the body of CHALLENGE. This ensures that the key is local to this scope, and therefore each value of the key is only used to encrypt one plaintext.

In the public-key setting, however, it turns out to be important to allow the adversary to see the public key before deciding which plaintexts to encrypt. (This concern is not present in the symmetric-key setting precisely because there is nothing public upon which the adversary's choice of plaintexts can depend.) For that reason, in the public-key setting we must sample the keys at initialization time so that the adversary can obtain the public key via GETPK. To ensure that the key is used to encrypt only one plaintext, we add a counter and a guard condition to CHALLENGE, so that it only responds once with a ciphertext.

Definition 14.4 *Let Σ be a public-key encryption scheme. Then Σ has **one-time secrecy** if $\mathcal{L}^{\Sigma}_{\text{pk-ots-L}} \approx \mathcal{L}^{\Sigma}_{\text{pk-ots-R}}$, where:*

$\mathcal{L}^{\Sigma}_{\text{pk-ots-L}}$

$(pk, sk) \leftarrow \Sigma.\text{KeyGen}$
$count := 0$

GETPK():
  return pk

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
  $count := count + 1$
  if $count > 1$: return null
  return $\Sigma.\text{Enc}(pk, m_L)$

$\mathcal{L}^{\Sigma}_{\text{pk-ots-R}}$

$(pk, sk) \leftarrow \Sigma.\text{KeyGen}$
$count := 0$

GETPK():
  return pk

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
  $count := count + 1$
  if $count > 1$: return null
  return $\Sigma.\text{Enc}(pk, m_R)$

Claim 14.5 *Let Σ be a public-key encryption scheme. If Σ has one-time secrecy, then Σ is CPA-secure.*

Proof Suppose $\mathcal{L}^{\Sigma}_{\text{pk-ots-L}} \approx \mathcal{L}^{\Sigma}_{\text{pk-ots-R}}$. Our goal is to show that $\mathcal{L}^{\Sigma}_{\text{pk-cpa-L}} \approx \mathcal{L}^{\Sigma}_{\text{pk-cpa-R}}$. The proof centers around the following hybrid library $\mathcal{L}_{\text{hyb-}h}$, which is designed to be linked to either

$\mathcal{L}_{\text{pk-ots-L}}$ or $\mathcal{L}_{\text{pk-ots-R}}$:

$\mathcal{L}_{\text{hyb-}h}$

```
count = 0
pk := GETPK()

CHALLENGE(m_L, m_R ∈ Σ.M):
  count := count + 1
  if count < h :
    return Σ.Enc(pk, m_R)
  elsif count = h :
    return CHALLENGE'(m_L, m_R)
  else:
    return Σ.Enc(pk, m_L)
```

Here the value h is an unspecified value that will be a hard-coded constant, and CHALLENGE′ (called by the "elsif" branch) and GETPK refer to the subroutine in $\mathcal{L}_{\text{pk-ots-}\star}$. Note that $\mathcal{L}_{\text{hyb-}h}$ is designed so that it only makes one call to CHALLENGE′ — in particular, only when its own CHALLENGE subroutine is called for the h^{th} time.

We now make a few observations:

$\mathcal{L}_{\text{hyb-1}} \diamond \mathcal{L}_{\text{pk-ots-L}} \equiv \mathcal{L}_{\text{pk-cpa-L}}$: In both libraries, every call to CHALLENGE encrypts the left plaintext. In particular, the first call to CHALLENGE in $\mathcal{L}_{\text{hyb-1}}$ triggers the "elsif" branch, so the challenge is routed to $\mathcal{L}_{\text{pk-ots-L}}$, which encrypts the left plaintext. In all other calls to CHALLENGE, the "else" branch is triggered and the left plaintext is encrypted explicitly.

$\mathcal{L}_{\text{hyb-}h} \diamond \mathcal{L}_{\text{pks-ots-L}} \equiv \mathcal{L}_{\text{hyb-}(h+1)} \diamond \mathcal{L}_{\text{pks-ots-R}}$, for all h. In both of these libraries, the first h calls to CHALLENGE encrypt the left plaintext, and all subsequent calls encrypt the right plaintext.

$\mathcal{L}_{\text{hyb-}h} \diamond \mathcal{L}_{\text{pks-ots-L}} \approx \mathcal{L}_{\text{hyb-}h} \diamond \mathcal{L}_{\text{pks-ots-R}}$, for all h. This simply follows from the fact that $\mathcal{L}_{\text{pks-ots-L}} \approx \mathcal{L}_{\text{pks-ots-R}}$.

$\mathcal{L}_{\text{hyb-}q} \diamond \mathcal{L}_{\text{pks-ots-R}} \equiv \mathcal{L}_{\text{pk-cpa-R}}$, where q is the number of times the calling program calls CHALLENGE. In particular, every call to CHALLENGE encrypts the right plaintext.

Putting everything together, we have that:

$$\begin{aligned}\mathcal{L}_{\text{pk-cpa-L}} &\equiv \mathcal{L}_{\text{hyb-1}} \diamond \mathcal{L}_{\text{pk-ots-L}} \approx \mathcal{L}_{\text{hyb-1}} \diamond \mathcal{L}_{\text{pk-ots-R}} \\ &\equiv \mathcal{L}_{\text{hyb-2}} \diamond \mathcal{L}_{\text{pk-ots-L}} \approx \mathcal{L}_{\text{hyb-2}} \diamond \mathcal{L}_{\text{pk-ots-R}} \\ &\vdots\end{aligned}$$

$$\equiv \mathcal{L}_{\text{hyb-}q} \diamond \mathcal{L}_{\text{pk-ots-L}} \approx \mathcal{L}_{\text{hyb-}q} \diamond \mathcal{L}_{\text{pk-ots-R}}$$
$$\equiv \mathcal{L}_{\text{pk-cpa-R}},$$

and so $\mathcal{L}_{\text{pk-cpa-L}} \approx \mathcal{L}_{\text{pk-cpa-R}}$. ■

The reason this proof goes through for public-key encryption but not symmetric-key encryption is that *anyone can encrypt* in a public-key scheme. In a symmetric-key scheme, it is not possible to generate encryptions without the key. But in a public-key scheme, the encryption key is public.

In more detail, the $\mathcal{L}_{\text{hyb-}h}$ library can indeed obtain pk from $\mathcal{L}_{\text{pk-ots-}\star}$. It therefore has enough information to perform the encryptions for all calls to CHALLENGE. Indeed, you can think of $\mathcal{L}_{\text{hyb-}0}$ as doing everything that $\mathcal{L}_{\text{pk-cpa-L}}$ does, even though it doesn't know the secret key. We let $\mathcal{L}_{\text{hyb-}h}$ designate the h^{th} call to CHALLENGE as a special one to be handled by $\mathcal{L}_{\text{pk-ots-}\star}$. This allows us to change the h^{th} encryption from using m_L to m_R.

14.3 ElGamal Encryption

ElGamal encryption is a public-key encryption scheme that is based on DHKA.

Construction 14.6 (ElGamal) *The public parameters are a choice of cyclic group $\mathbb{G}$ with n elements and generator g.*

	KeyGen:	Enc($A, M \in \mathbb{G}$):	
$\mathcal{M} = \mathbb{G}$	$sk := a \leftarrow \mathbb{Z}_n$	$b \leftarrow \mathbb{Z}_n$	Dec(a, (B, X)):
$C = \mathbb{G}^2$	$pk := A := g^a$	$B := g^b$	return $X(B^a)^{-1}$
	return (pk, sk)	return $(B, M \cdot A^b)$	

The scheme satisfies correctness, since for all M:

$$\begin{aligned}\text{Dec}(sk, \text{Enc}(pk, M)) &= \text{Dec}(sk, (g^b, M \cdot A^b)) \\ &= (M \cdot A^b)((g^b)^a)^{-1} \\ &= M \cdot (g^{ab})(g^{ab})^{-1} = M.\end{aligned}$$

Security

Imagine an adversary who is interested in attacking an ElGamal scheme. This adversary sees the public key $A = g^a$ and a ciphertext $(g^b, M\, g^{ab})$ go by. Intuitively, the Decisional Diffie-Hellman assumption says that the value g^{ab} looks random, even to someone who has seen g^a and g^b. Thus, the message M is masked with a pseudorandom group element — as we've seen before, this is a lot like masking the message with a random pad as in one-time pad. The only change here is that instead of the XOR operation, we are using the group operation in $\mathbb{G}$.

More formally, we can prove the security of ElGamal under the DDH assumption:

Claim 14.7 *If the DDH assumption in group $\mathbb{G}$ is true, then ElGamal in group $\mathbb{G}$ is CPA$-secure.*

Proof It suffices to show that ElGamal has pseudorandom ciphertexts when the calling program sees only a single ciphertext. In other words, we will show that $\mathcal{L}_{\text{pk-ots\$-real}} \approx \mathcal{L}_{\text{pk-ots\$-rand}}$, where these libraries are the $\mathcal{L}_{\text{pk-cpa\$-}\star}$ libraries from Definition 14.2 but with the single-ciphertext restriction used in Definition 14.4. It is left as an exercise to show that $\mathcal{L}_{\text{pk-ots\$-real}} \approx \mathcal{L}_{\text{pk-ots\$-rand}}$ implies CPA\$ security (which in turn implies CPA security); the proof is very similar to that of Claim 14.5.

The sequence of hybrid libraries is given below:

$\mathcal{L}_{\text{pk-ots\$-real}}$

$a \leftarrow \mathbb{Z}_n$
$A := g^a$
$count = 0$

GETPK():
  return A

QUERY($M \in \mathbb{G}$):
  $count : count + 1$
  if $count > 1$: return null
  $b \leftarrow \mathbb{Z}_n$
  $B := g^b$
  $X := M \cdot A^b$
  return (B, X)

The starting point is the $\mathcal{L}_{\text{pk-ots\$-real}}$ library, shown here with the details of ElGamal filled in.

$a \leftarrow \mathbb{Z}_n$; $b \leftarrow \mathbb{Z}_n$
$A := g^a$; $B := g^b$; $C := A^b$
$count = 0$

GETPK():
  return A

QUERY($M \in \mathbb{G}$):
  $count : count + 1$
  if $count > 1$: return null
  $X := M \cdot C$
  return (B, X)

The main body of QUERY computes some intermediate values B and A^b. But since those lines are only reachable one time, it does not change anything to precompute them at initialization time.

$(A, B, C) \leftarrow$ DHQUERY()
$count = 0$

GETPK():
  return A

QUERY($M \in \mathbb{G}$):
  $count : count + 1$
  if $count > 1$: return null
  $X := M \cdot C$
  return (B, X)

$\diamond$

$\mathcal{L}_{\text{dh-real}}$

DHQUERY():
  $a, b \leftarrow \mathbb{Z}_n$
  return (g^a, g^b, g^{ab})

We can factor out the generation of A, B, C in terms of the $\mathcal{L}_{\text{dh-real}}$ library from the Decisional Diffie-Hellman security definition (Definition 13.5).

$(A, B, C) \leftarrow$ DHQUERY()
$count = 0$

GETPK():
  return A

QUERY($M \in \mathbb{G}$):
  $count : count + 1$
  if $count > 1$: return null
  $X := M \cdot C$
  return (B, X)

$\diamond$

$\mathcal{L}_{\text{dh-rand}}$

DHQUERY():
  $a, b, c \leftarrow \mathbb{Z}_n$
  return (g^a, g^b, g^c)

Applying the security of DDH, we can replace $\mathcal{L}_{\text{dh-real}}$ with $\mathcal{L}_{\text{dh-rand}}$.

$a, b, c \leftarrow \mathbb{Z}_n$
$A := g^a;\ B := g^b;\ C := g^c$
$count = 0$

GETPK():
  return A

QUERY($M \in \mathbb{G}$):
  $count : count + 1$
  if $count > 1$: return null
  $X := M \cdot C$
  return (B, X)

The call to DHQUERY has been inlined.

$a \leftarrow \mathbb{Z}_n$
$A := g^a$
$count = 0$

GETPK():
  return A

QUERY($M \in \mathbb{G}$):
  $count : count + 1$
  if $count > 1$: return null
  $b, c \leftarrow \mathbb{Z}_n$
  $B := g^b$; $C := g^c$
  $X := M \cdot C$
  return (B, X)

As before, since the main body of QUERY is only reachable once, we can move the choice of B and C into that subroutine instead of at initialization time.

$\mathcal{L}_{\text{pk-ots\$-rand}}$

$a \leftarrow \mathbb{Z}_n$
$A := g^a$
$count = 0$

GETPK():
  return A

QUERY($M \in \mathbb{G}$):
  $count : count + 1$
  if $count > 1$: return null
  $b, x \leftarrow \mathbb{Z}_n$
  $B := g^b$; $X := g^x$
  return (B, X)

When b is sampled uniformly from $\mathbb{Z}_n$, the expression $B = g^b$ is a uniformly distributed element of $\mathbb{G}$. Also recall that when C is a uniformly distributed element of $\mathbb{G}$, then $M \cdot C$ is uniformly distributed — this is analogous to the one-time pad property (see Exercise 2.5). Applying this change gives the library to the left.

In the final hybrid, the response to QUERY is a pair of uniformly distributed group elements (B, X). Hence that library is exactly $\mathcal{L}_{\text{pk-ots\$-rand}}$, as desired. ■

14.4 Hybrid Encryption

As a rule, public-key encryption schemes are much more computationally expensive than symmetric-key schemes. Taking ElGamal as a representative example, computing g^b in a cryptographically secure cyclic group is considerably more expensive than one evaluation of AES. As the plaintext data increases in length, the difference in cost between public-key and symmetric-key techniques only gets worse.

A clever way to minimize the cost of public-key cryptography is to use a method called **hybrid encryption.** The idea is to use the expensive public-key scheme to encrypt a *temporary key* for a symmetric-key scheme. Then use the temporary key to (cheaply) encrypt the large plaintext data.

To decrypt, one can use the decryption key of the public-key scheme to obtain the temporary key. Then the temporary key can be used to decrypt the main payload.

Construction 14.8 (Hybrid Enc) *Let Σ_{pub} be a public-key encryption scheme, and let Σ_{sym} be a symmetric-key encryption scheme, where $\Sigma_{sym}.\mathcal{K} \subseteq \Sigma_{pub}.\mathcal{M}$ — that is, the public-key scheme is capable of encrypting keys of the symmetric-key scheme.*

Then we define Σ_{hyb} to be the following construction:

$\mathcal{M} = \Sigma_{sym}.\mathcal{M}$
$C = \Sigma_{pub}.C \times \Sigma_{sym}.C$

KeyGen:
$(pk, sk) \leftarrow \Sigma_{\text{pub}}.\text{KeyGen}$
return (pk, sk)

Enc(pk, m):
$tk \leftarrow \Sigma_{\text{sym}}.\text{KeyGen}$
$c_{\text{pub}} \leftarrow \Sigma_{\text{pub}}.\text{Enc}(pk, tk)$
$c_{\text{sym}} \leftarrow \Sigma_{\text{sym}}.\text{Enc}(tk, m)$
return $(c_{\text{pub}}, c_{\text{sym}})$

Dec$(sk, (c_{\text{pub}}, c_{\text{sym}}))$:
$tk := \Sigma_{\text{pub}}.\text{Dec}(sk, c_{\text{pub}})$
return $\Sigma_{\text{sym}}.\text{Dec}(tk, c_{\text{sym}})$

Importantly, the message space of the hybrid encryption scheme is the message space of the symmetric-key scheme (think of this as involving very long plaintexts), but encryption and decryption involves expensive public-key operations only on a small temporary key (think of this as a very short string).

The correctness of the scheme can be verified via:

$$\begin{aligned}
\text{Dec}(sk, \text{Enc}(pk, m)) &= \text{Dec}\Big(sk, \big(\Sigma_{\text{pub}}.\text{Enc}(pk, tk),\ \Sigma_{\text{sym}}.\text{Enc}(tk, m)\big)\Big) \\
&= \Sigma_{\text{sym}}.\text{Dec}\Big(\Sigma_{\text{pub}}.\text{Dec}\big(sk,\ \Sigma_{\text{pub}}.\text{Enc}(pk, tk)\big),\ \Sigma_{\text{sym}}.\text{Enc}(tk, m)\Big) \\
&= \Sigma_{\text{sym}}.\text{Dec}\Big(tk,\ \Sigma_{\text{sym}}.\text{Enc}(tk, m)\Big) \\
&= m.
\end{aligned}$$

To show that hybrid encryption is a valid way to encrypt data, we prove that it provides CPA security, when its two components have appropriate security properties:

Claim 14.9 *If Σ_{sym} is a one-time-secret symmetric-key encryption scheme and Σ_{pub} is a CPA-secure public-key encryption scheme, then the hybrid scheme Σ_{hyb} (Construction 14.8) is also a CPA-secure public-key encryption scheme.*

Note that Σ_{sym} does not even need to be CPA-secure. Intuitively, one-time secrecy suffices because each temporary key tk is used only once to encrypt just a single plaintext.

Proof As usual, our goal is to show that $\mathcal{L}^{\Sigma_{\text{hyb}}}_{\text{pk-cpa-L}} \approx \mathcal{L}^{\Sigma_{\text{hyb}}}_{\text{pk-cpa-R}}$, which we do in a standard sequence of hybrids:

$\mathcal{L}^{\Sigma_{\text{hyb}}}_{\text{pk-cpa-L}}$

$(pk, sk) \leftarrow \Sigma_{\text{pub}}.\text{KeyGen}$

GETPK():
  return pk

CHALLENGE(m_L, m_R):
  $tk \leftarrow \Sigma_{\text{sym}}.\text{KeyGen}$
  $c_{\text{pub}} \leftarrow \Sigma_{\text{pub}}.\text{Enc}(pk, tk)$
  $c_{\text{sym}} \leftarrow \Sigma_{\text{sym}}.\text{Enc}(tk, m_L)$
  return $(c_{\text{pub}}, c_{\text{sym}})$

The starting point is $\mathcal{L}_{\text{pk-cpa-L}}$, shown here with the details of Σ_{hyb} filled in.

Our only goal is to somehow replace m_L with m_R. Since m_L is only used as a plaintext for Σ_{sym}, it is tempting to simply apply the one-time-secrecy property of Σ_{sym} to argue that m_L can be replaced with m_R. Unfortunately, this cannot work because the *key* used for that ciphertext is tk, which is used elsewhere. In particular, it is used as an argument to $\Sigma_{\text{pub}}.\text{Enc}$.

However, using tk as the plaintext argument to $\Sigma_{\text{pub}}.\text{Enc}$ should *hide* tk to the calling program, if Σ_{pub} is CPA-secure. That is, the Σ_{pub}-encryption of tk should look like a Σ_{pub}-encryption of some unrelated dummy value. More formally, we can factor out the call to $\Sigma_{\text{pub}}.\text{Enc}$ in terms of the $\mathcal{L}_{\text{pk-cpa-L}}$ library, as follows:

CHALLENGE(m_L, m_R):
  $tk \leftarrow \Sigma_{\text{sym}}.\text{KeyGen}$
  $tk' \leftarrow \Sigma_{\text{sym}}.\text{KeyGen}$
  $c_{\text{pub}} \leftarrow$ CHALLENGE$'(tk, tk')$
  $c_{\text{sym}} \leftarrow \Sigma_{\text{sym}}.\text{Enc}(tk, m_L)$
  return $(c_{\text{pub}}, c_{\text{sym}})$

$\diamond$

$\mathcal{L}^{\Sigma_{\text{pub}}}_{\text{pk-cpa-L}}$

$(pk, sk) \leftarrow \Sigma_{\text{pub}}.\text{KeyGen}$

GETPK():
  return pk

CHALLENGE$'(tk_L, tk_R)$:
  return $\Sigma_{\text{pub}}.\text{Enc}(pk, tk_L)$

Here we have changed the variable names of the arguments of CHALLENGE$'$ to avoid unnecessary confusion. Note also that CHALLENGE now chooses *two* temporary keys — one which is actually used to encrypt m_L and one which is not used anywhere. This is because syntactically we must have two arguments to pass into CHALLENGE$'$.

Now imagine replacing $\mathcal{L}_{\text{pk-cpa-L}}$ with $\mathcal{L}_{\text{pk-cpa-R}}$ and then inlining subroutine calls. The result is:

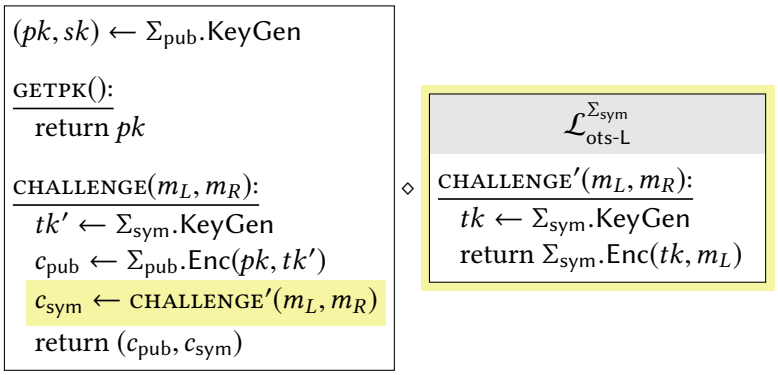
$(pk, sk) \leftarrow \Sigma_{\text{pub}}.\text{KeyGen}$

GETPK():
  return pk

CHALLENGE(m_L, m_R):
  $tk \leftarrow \Sigma_{\text{sym}}.\text{KeyGen}$
  $tk' \leftarrow \Sigma_{\text{sym}}.\text{KeyGen}$
  $c_{\text{pub}} \leftarrow \Sigma_{\text{pub}}.\text{Enc}(pk, tk')$
  $c_{\text{sym}} \leftarrow \Sigma_{\text{sym}}.\text{Enc}(tk, m_L)$
  return $(c_{\text{pub}}, c_{\text{sym}})$

At this point, it *does* now work to factor out the call to $\Sigma_{\text{sym}}.\text{Enc}$ in terms of the $\mathcal{L}_{\text{ots-L}}$ library. This is because the key tk is not used anywhere else in the library. The result of

factoring out in this way is:



At this point, we can replace $\mathcal{L}_{\text{ots-L}}$ with $\mathcal{L}_{\text{ots-R}}$. After this change the Σ_{sym}-ciphertext encrypts m_R instead of m_L. This is the "half-way point" of the proof, and the rest of the steps are a mirror image of what has come before. In summary: we inline $\mathcal{L}_{\text{ots-R}}$, then we apply CPA security to replace the Σ_{pub}-encryption of tk' with tk. The result is exactly $\mathcal{L}_{\text{pk-cpa-R}}$, as desired. ■

Exercises

14.1. Prove Claim 14.3.

14.2. Show that a 2-message key-agreement protocol exists if and only if CPA-secure public-key encryption exists.

In other words, show how to construct a CPA-secure encryption scheme from any 2-message KA protocol, and vice-versa. Prove the security of your constructions.

14.3. (a) Suppose you are given an ElGamal encryption of an unknown plaintext $M \in \mathbb{G}$. Show how to construct a different ciphertext that also decrypts to the same M.

(b) Suppose you are given two ElGamal encryptions, of unknown plaintexts $M_1, M_2 \in \mathbb{G}$. Show how to construct a ciphertext that decrypts to their product $M_1 \cdot M_2$.

14.4. Suppose you obtain two ElGamal ciphertexts $(B_1, C_1), (B_2, C_2)$ that encrypt unknown plaintexts M_1 and M_2. Suppose you also know the public key A and cyclic group generator g.

(a) What information can you infer about M_1 and M_2 if you observe that $B_1 = B_2$?

(b) What information can you infer about M_1 and M_2 if you observe that $B_1 = g \cdot B_2$?

★ (c) What information can you infer about M_1 and M_2 if you observe that $B_1 = (B_2)^2$?