Reminder: Security of Plain RSA

Is plain RSA secure (assuming the RSA assumption)?

Observation 1:

- The RSA assumption is defined with respect to the RSA-ind experiment
- In the RSA-ind experiment the adversary wins by computing the e-th root of a **random group element** $x \in Z_N^*$
- If m is a message chosen u.a.r. then, since $f_e(x) = x^e$ is a permutation, m^e is also uniformly distributed in Z_N^*

Reminder: Security of Plain RSA

Is plain RSA secure (assuming the RSA assumption)?

Observation 1:

- The RSA assumption is defined with respect to the RSA-ind experiment
- In the RSA-ind experiment the adversary wins by computing the e-th root of a **random group element** $x \in Z_N^*$
- If m is a message chosen u.a.r. then, since $f_e(x) = x^e$ is a permutation, m^e is also uniformly distributed in Z_N^*
- In general m is not a random message! $\implies$ We can't rely on the RSA assumption.

Reminder: Security of Plain RSA

Is plain RSA secure (assuming the RSA assumption)?

Observation 1:

- The RSA assumption is defined with respect to the RSA-ind experiment
- In the RSA-ind experiment the adversary wins by computing the e-th root of a **random group element** $x \in Z_N^*$
- If m is a message chosen u.a.r. then, since $f_e(x) = x^e$ is a permutation, m^e is also uniformly distributed in Z_N^*
- In general m is not a random message! $\implies$ We can't rely on the RSA assumption.

Observation 2:

- The plain RSA scheme is **deterministic**!
- It can't be CPA-secure!

(and since CPA-security and EAV-security concide in the public-key world, it is not even secure agaisnt an eavesdropper)

Reminder: Security of Plain RSA

Is plain RSA secure (assuming the RSA assumption)?

Observation 1:

- The RSA assumption is defined with respect to the RSA-ind experiment
- In the RSA-ind experiment the adversary wins by computing the e-th root of a **random group element** $x \in Z_N^*$
- If m is a message chosen u.a.r. then, since $f_e(x) = x^e$ is a permutation, m^e is also uniformly distributed in Z_N^*
- In general m is not a random message! $\implies$ We can't rely on the RSA assumption.

Observation 2:

- The plain RSA scheme is **deterministic**!
- It can't be CPA-secure!

(and since CPA-security and EAV-security concide in the public-key world, it is not even secure agaisnt an eavesdropper)

Plain RSA should never be used!

CCA Security of Plain RSA

Since Plain RSA is not CPA-Secure, we already know that it can't be CCA-Secure

CCA Security of Plain RSA

Since Plain RSA is not CPA-Secure, we already know that it can't be CCA-Secure

We can explicitly show that Plain-RSA is malleable:

- Let $c = m^e \pmod{N}$ a Plain RSA ciphertext and choose $\alpha \in \mathbb{Z}_N^*$

CCA Security of Plain RSA

Since Plain RSA is not CPA-Secure, we already know that it can't be CCA-Secure

We can explicitly show that Plain-RSA is malleable:

- Let $c = m^e \pmod N$ a Plain RSA ciphertext and choose $\alpha \in \mathbb{Z}_N^*$
- What does the ciphertext $c' = \alpha^e \cdot c \pmod N$ decrypt to?

CCA Security of Plain RSA

Since Plain RSA is not CPA-Secure, we already know that it can't be CCA-Secure

We can explicitly show that Plain-RSA is malleable:

- Let $c = m^e \pmod N$ a Plain RSA ciphertext and choose $\alpha \in \mathbb{Z}_N^*$
- What does the ciphertext $c' = \alpha^e \cdot c \pmod N$ decrypt to?

$$(c')^d = \alpha^{ed} m^{ed} = \alpha \cdot m \pmod N$$

CCA Security of Plain RSA

Since Plain RSA is not CPA-Secure, we already know that it can't be CCA-Secure

We can explicitly show that Plain-RSA is malleable:

- Let $c = m^e \pmod N$ a Plain RSA ciphertext and choose $\alpha \in \mathbb{Z}_N^*$
- What does the ciphertext $c' = \alpha^e \cdot c \pmod N$ decrypt to?

$$(c')^d = \alpha^{ed} m^{ed} = \alpha \cdot m \pmod N$$

- c' is a valid encryption of $\alpha \cdot m$. We can build a polynomial-time adversary that wins the $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$ experiment with non-negligible probability:

CCA Security of Plain RSA

Since Plain RSA is not CPA-Secure, we already know that it can't be CCA-Secure

We can explicitly show that Plain-RSA is malleable:

- Let $c = m^e \pmod{N}$ a Plain RSA ciphertext and choose $\alpha \in \mathbb{Z}_N^*$
- What does the ciphertext $c' = \alpha^e \cdot c \pmod{N}$ decrypt to?

$$(c')^d = \alpha^{ed} m^{ed} = \alpha \cdot m \pmod{N}$$

- c' is a valid encryption of $\alpha \cdot m$. We can build a polynomial-time adversary that wins the $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$ experiment with non-negligible probability:

- Receive the public key $pk = \langle N, e \rangle$
- Output two messages m_0, m_1 and receive the challenge ciphertext c
- Query the decryption oracle with $c' = 2^e \cdot c \pmod{N}$ to obtain $m' = 2 \cdot m_b$ of
- If $m' = 2 \cdot m_0 \pmod{N}$ return $b' = 0$, otherwise return $b' = 1$

“Fixing” RSA

How do we make RSA secure?

Option 1: Use randomized encoding

- Make sure that the message is “random enough”
- Choose some **randomized encoding** between messages and group elements
- To encrypt: encode the message, then encrypt the group element
- To decrypt: decrypt the ciphertext to recover the group element, then decode the group element

Option 2: Use RSA as a KEM **with a key derivation function**

“Fixing” RSA

How do we make RSA secure?

Option 1: Use randomized encoding

- Make sure that the message is “random enough”
- Choose some **randomized encoding** between messages and group elements
- To encrypt: encode the message, then encrypt the group element
- To decrypt: decrypt the ciphertext to recover the group element, then decode the group element

Option 2: Use RSA as a KEM **with a key derivation function**

Padded RSA

Idea: prepend a random "padding" string to the message

Padded RSA

Idea: prepend a random "padding" string to the message

Let $\eta = \lfloor \log N \rfloor$ (one less than the number of bits of N)

We will use a padding of length $\ell(n) < \eta$

Padded RSA

Idea: prepend a random "padding" string to the message

Let $\eta = \lfloor \log N \rfloor$ (one less than the number of bits of N)

We will use a padding of length $\ell(n) < \eta$

To encrypt a message in $\{0,1\}^{\eta - \ell(n)}$:

Enc$_{pk}(m)$: (where $pk = \langle N, e \rangle$)

- Choose a padding r u.a.r. from $\{0,1\}^{\ell(n)}$
- Interpret $\hat{m} = r \| m$ as an element of Z_N^*
- Output $c = \hat{m}^e \pmod{N}$

Padded RSA

Idea: prepend a random "padding" string to the message

Let $\eta = \lfloor \log N \rfloor$ (one less than the number of bits of N)

We will use a padding of length $\ell(n) < \eta$

To encrypt a message in $\{0,1\}^{\eta - \ell(n)}$:

Enc$_{pk}(m)$: (where $pk = \langle N, e \rangle$)

- Choose a padding r u.a.r. from $\{0,1\}^{\ell(n)}$
- Interpret $\hat{m} = r \| m$ as an element of Z_N^*
- Output $c = \hat{m}^e \pmod N$

Almost all $\hat{m}$ are binary encodings of an integer in $\mathbb{Z}_N^*$

Padded RSA

Idea: prepend a random "padding" string to the message

Let $\eta = \lfloor \log N \rfloor$ (one less than the number of bits of N)

We will use a padding of length $\ell(n) < \eta$

To encrypt a message in $\{0,1\}^{\eta-\ell(n)}$:

Enc$_{pk}(m)$: (where $pk = \langle N, e \rangle$)

- Choose a padding r u.a.r. from $\{0,1\}^{\ell(n)}$
- Interpret $\hat{m} = r \| m$ as an element of Z_N^*
- Output $c = \hat{m}^e \pmod N$

Almost all $\hat{m}$ are binary encodings of an integer in $\mathbb{Z}_N^*$

To decrypt c:

Dec$_{sk}(c)$: (where $sk = \langle N, d \rangle$)

- Compute $\hat{m} = c^d \pmod N$
- Return the $\eta - \ell(n)$ least significant bits of $\hat{m}$

Security of Padded RSA

The security of padded RSA depends on the choice of $\ell(n)$

- An attacker can win the $\mathsf{PubK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n)$ experiment by guessing $r \in \{0,1\}^{\ell(n)}$ using a brute-force attack

Security of Padded RSA

The security of padded RSA depends on the choice of $\ell(n)$

- An attacker can win the $\mathsf{PubK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n)$ experiment by guessing $r \in \{0,1\}^{\ell(n)}$ using a brute-force attack

  $\implies \ell(n)$ need to be long enough

Security of Padded RSA

The security of padded RSA depends on the choice of $\ell(n)$

- An attacker can win the $\mathsf{PubK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n)$ experiment by guessing $r \in \{0,1\}^{\ell(n)}$ using a brute-force attack

  $\Longrightarrow$ $\ell(n)$ need to be long enough

For example, if $\ell(n) = O(\log n)$ then $|\{0,1\}^{\ell(n)}| = 2^{\ell(n)} \leq 2^{k \log n} = n^k$ (for some constant k)

Security of Padded RSA

The security of padded RSA depends on the choice of $\ell(n)$

- An attacker can win the $\mathsf{PubK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n)$ experiment by guessing $r \in \{0,1\}^{\ell(n)}$ using a brute-force attack

  $\implies$ $\ell(n)$ need to be long enough

For example, if $\ell(n) = O(\log n)$ then $|\{0,1\}^{\ell(n)}| = 2^{\ell(n)} \leq 2^{k \log n} = \boxed{n^k}$ (for some constant k)

Polynomial

Security of Padded RSA

The security of padded RSA depends on the choice of $\ell(n)$

- An attacker can win the $\mathsf{PubK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n)$ experiment by guessing $r \in \{0,1\}^{\ell(n)}$ using a brute-force attack

  $\Longrightarrow$ $\ell(n)$ need to be long enough

For example, if $\ell(n) = O(\log n)$ then $|\{0,1\}^{\ell(n)}| = 2^{\ell(n)} \leq 2^{k \log n} = \boxed{n^k}$ (for some constant k)

Polynomial

Good news: If $\ell = \eta - 1$, i.e., the message is just a single bit, then it is possible to show that Padded RSA is CPA-secure if the RSA assumption holds

Security of Padded RSA

The security of padded RSA depends on the choice of $\ell(n)$

- An attacker can win the $\text{PubK}^{\text{cpa}}_{\mathcal{A},\Pi}(n)$ experiment by guessing $r \in \{0,1\}^{\ell(n)}$ using a brute-force attack

  $\implies$ $\ell(n)$ need to be long enough

For example, if $\ell(n) = O(\log n)$ then $|\{0,1\}^{\ell(n)}| = 2^{\ell(n)} \leq 2^{k \log n} = \boxed{n^k}$ (for some constant k)

Polynomial

Good news: If $\ell = \eta - 1$, i.e., the message is just a single bit, then it is possible to show that Padded RSA is CPA-secure if the RSA assumption holds

The situation is less clear for intermediate values of $\ell(n)$

- No proof of security
- No known polynomial-time attacks

Security of Padded RSA

The security of padded RSA depends on the choice of $\ell(n)$

- An attacker can win the $\mathsf{PubK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n)$ experiment by guessing $r \in \{0,1\}^{\ell(n)}$ using a brute-force attack

  $\implies \ell(n)$ need to be long enough

For example, if $\ell(n) = O(\log n)$ then $|\{0,1\}^{\ell(n)}| = 2^{\ell(n)} \leq 2^{k \log n} = \boxed{n^k}$ (for some constant k)

Polynomial

Good news: If $\ell = \eta - 1$, i.e., the message is just a single bit, then it is possible to show that Padded RSA is CPA-secure if the RSA assumption holds

The situation is less clear for intermediate values of $\ell(n)$

- No proof of security
- No known polynomial-time attacks

In any case, Padded RSA is not CCA-secure!

PKCS #1 v1.5

Variant of Padded RSA standardized in 1993 (PKCS stands for Public-Key Cryptography Standard)

It uses paddings of specific lengths and formats

- For a public key $pk = \langle N, e \rangle$, let k be the length of N in bytes
- The message m is required to have an integral number D of bytes between 1 and $k - 11$
- The remaining $k - D \geq 11$ bytes form the padding

PKCS #1 v1.5

Variant of Padded RSA standardized in 1993 (PKCS stands for Public-Key Cryptography Standard)

It uses paddings of specific lengths and formats

- For a public key $pk = \langle N, e \rangle$, let k be the length of N in bytes
- The message m is required to have an integral number D of bytes between 1 and $k - 11$
- The remaining $k - D \geq 11$ bytes form the padding
  — The first two bytes of the padding are (the binary encoding of) 0 and 2, respectively
  — The last byte of the padding is 0

PKCS #1 v1.5

Variant of Padded RSA standardized in 1993 (PKCS stands for Public-Key Cryptography Standard)

It uses paddings of specific lengths and formats

- For a public key $pk = \langle N, e \rangle$, let k be the length of N in bytes
- The message m is required to have an integral number D of bytes between 1 and $k - 11$
- The remaining $k - D \geq 11$ bytes form the padding

— The first two bytes of the padding are (the binary encoding of) 0 and 2, respectively

— The last byte of the padding is 0

— Each of the remaining bytes is (the binary encoding of) a value chosen u.a.r. from $\{1, \ldots, 255\}$

PKCS #1 v1.5

Variant of Padded RSA standardized in 1993 (PKCS stands for Public-Key Cryptography Standard)

It uses paddings of specific lengths and formats

- For a public key $pk = \langle N, e \rangle$, let k be the length of N in bytes
- The message m is required to have an integral number D of bytes between 1 and $k - 11$
- The remaining $k - D \geq 11$ bytes form the padding

— The first two bytes of the padding are (the binary encoding of) 0 and 2, respectively

— The last byte of the padding is 0

— Each of the remaining bytes is (the binary encoding of) a value chosen u.a.r. from $\{1, \dots, 255\}$

$$\hat{m} = \texttt{0x00} \| \texttt{0x02} \| r \| \texttt{0x00} \| m$$

random string of (at least 8) non-zero bytes

PKCS #1 v1.5

Variant of Padded RSA standardized in 1993 (PKCS stands for Public-Key Cryptography Standard)

It uses paddings of specific lengths and formats

- For a public key $pk = \langle N, e \rangle$, let k be the length of N in bytes
- The message m is required to have an integral number D of bytes between 1 and $k - 11$
- The remaining $k - D \geq 11$ bytes form the padding

— The first two bytes of the padding are (the binary encoding of) 0 and 2, respectively

— The last byte of the padding is 0

— Each of the remaining bytes is (the binary encoding of) a value chosen u.a.r. from $\{1, \ldots, 255\}$

$$\hat{m} = \texttt{0x00} \| \texttt{0x02} \| r \| \texttt{0x00} \| m$$

random string of (at least 8) non-zero bytes

- The choice of the padding ensures that $\hat{m} < N$ and that m can be unambiguously recovered from $\hat{m}$

Security of PKCS #1 v1.5

Is it secure?

Security of PKCS #1 v1.5

Is it secure? **No!**

Security of PKCS #1 v1.5

Is it secure? **No!**

The PKCS #1 v1.5 standard allows for padding that is too short!

Security of PKCS #1 v1.5

Is it secure? **No!**

The PKCS #1 v1.5 standard allows for padding that is too short!

As an example consider a message that is as long as possible, and consists of all 0s except for the most significant bit b, which is unknown

$$m = b \,\|\, 0^L \quad \text{for } L = 8(k-11) - 1$$

Security of PKCS #1 v1.5

Is it secure? **No!**

The PKCS #1 v1.5 standard allows for padding that is too short!

As an example consider a message that is as long as possible, and consists of all 0s except for the most significant bit b, which is unknown

$$m = b \,\|\, 0^L \quad \text{for } L = 8(k-11)-1$$

The resulting ciphertext is

$$c = (\texttt{0x00} \| \texttt{0x02} \| r \| \texttt{0x00} \| \, b \,\|\, 0^L)^e \pmod{N}$$

Security of PKCS #1 v1.5

Is it secure? **No!**

The PKCS #1 v1.5 standard allows for padding that is too short!

As an example consider a message that is as long as possible, and consists of all 0s except for the most significant bit b, which is unknown

$$m = b \,\|\, 0^L \quad \text{for } L = 8(k-11)-1$$

The resulting ciphertext is

$$c = (\texttt{0x00} \| \texttt{0x02} \| r \| \texttt{0x00} \| \, b \,\|\, 0^L)^e \pmod N$$

An attacker can compute:

$$c' = c \cdot (2^{-L})^e = ((\texttt{0x00} \| \texttt{0x02} \| r \| \texttt{0x00} \| \, b \,\|\, 0^L) \cdot 2^{-L})^e \pmod N$$

Security of PKCS #1 v1.5

Is it secure? **No!**

The PKCS #1 v1.5 standard allows for padding that is too short!

As an example consider a message that is as long as possible, and consists of all 0s except for the most significant bit b, which is unknown

$$m = b \,\|\, 0^L \quad \text{for } L = 8(k-11)-1$$

The resulting ciphertext is

$$c = (\texttt{0x00}\|\texttt{0x02}\|r\|\texttt{0x00}\|\, b \,\|\, 0^L)^e \pmod N$$

An attacker can compute:

$$\begin{aligned} c' = c \cdot (2^{-L})^e &= ((\texttt{0x00}\|\texttt{0x02}\|r\|\texttt{0x00}\|\, b \,\|\, 0^L) \cdot 2^{-L})^e \pmod N \\ &= (\texttt{0x00}\|\texttt{0x02}\|r\|\texttt{0x00}\|\, b)^e \pmod N \end{aligned}$$

Security of PKCS #1 v1.5

Is it secure? **No!**

The PKCS #1 v1.5 standard allows for padding that is too short!

As an example consider a message that is as long as possible, and consists of all 0s except for the most significant bit b, which is unknown

$$m = b \,\|\, 0^L \qquad \text{for } L = 8(k-11)-1$$

The resulting ciphertext is

$$c = (\mathtt{0x00} \| \mathtt{0x02} \| r \| \mathtt{0x00} \| \, b \, \| \, 0^L)^e \pmod N$$

An attacker can compute:

$$c' = c \cdot (2^{-L})^e = ((\mathtt{0x00} \| \mathtt{0x02} \| r \| \mathtt{0x00} \| \, b \, \| \, 0^L) \cdot 2^{-L})^e \pmod N$$

$$= (\mathtt{0x00} \| \mathtt{0x0}\underbrace{\mathtt{2} \| r \| \mathtt{0x00} \| \, b}_{\text{only 75 bits}})^e \pmod N$$

Security of PKCS #1 v1.5

Is it secure? **No!**

The PKCS #1 v1.5 standard allows for padding that is too short!

As an example consider a message that is as long as possible, and consists of all 0s except for the most significant bit b, which is unknown

$$m = b \,\|\, 0^L \qquad \text{for } L = 8(k-11) - 1$$

The resulting ciphertext is

$$c = (\texttt{0x00}\|\texttt{0x02}\|r\|\texttt{0x00}\|\, b \,\|\, 0^L)^e \pmod N$$

An attacker can compute:

$$\begin{aligned} c' = c \cdot (2^{-L})^e &= ((\texttt{0x00}\|\texttt{0x02}\|r\|\texttt{0x00}\|\, b \,\|\, 0^L) \cdot 2^{-L})^e \pmod N \\ &= (\texttt{0x00}\|\texttt{0x0}\underbrace{\texttt{2}\|r\|\texttt{0x00}\|\, b}_{\text{only 75 bits}})^e \pmod N \end{aligned}$$

If $(2^{75})^e \leq N$ (for example when N has more than $75e$ bits) the “short message” attack applies.

Security of PKCS #1 v1.5

To avoid these attacks we need to ensure that the number of bits in r is $\Omega(\frac{\log N}{e})$

Security of PKCS #1 v1.5

To avoid these attacks we need to ensure that the number of bits in r is $\Omega(\frac{\log N}{e})$

- In PKCS #1 v1.5 the number of random bits can be as small as a **constant!**

Security of PKCS #1 v1.5

To avoid these attacks we need to ensure that the number of bits in r is $\Omega(\frac{\log N}{e})$

- In PKCS #1 v1.5 the number of random bits can be as small as a **constant!**

The variant that uses long padding strings is **conjectured** to be CPA-secure (if the RSA assumption holds) but no security proof is known

Security of PKCS #1 v1.5

To avoid these attacks we need to ensure that the number of bits in r is $\Omega(\frac{\log N}{e})$

- In PKCS #1 v1.5 the number of random bits can be as small as a **constant!**

The variant that uses long padding strings is **conjectured** to be CPA-secure (if the RSA assumption holds) but no security proof is known

This is still not CCA-secure

Security of PKCS #1 v1.5

To avoid these attacks we need to ensure that the number of bits in r is $\Omega(\frac{\log N}{e})$

- In PKCS #1 v1.5 the number of random bits can be as small as a **constant!**

The variant that uses long padding strings is **conjectured** to be CPA-secure (if the RSA assumption holds) but no security proof is known

This is still not CCA-secure

A word of caution:

- The recipient needs to check that the leading bytes of the decoded plaintext are `0x00 0x02`
- If this is not the case, the decryption returns an error
- If not done properly, this might provide access to a **padding oracle**!
- The whole message m can be recovered!

RSA-OAEP (Encryption)

Optimal asymmetric encryption padding is a (more complex) randomized encoding of messages that results in a **CCA-secure** version of RSA, called **RSA-OAEP**

- Uses a two-round Feistel network with round functions G and H, where G and H are Hash functions modeled as two independent random oracles

RSA-OAEP (Encryption)

Optimal asymmetric encryption padding is a (more complex) randomized encoding of messages that results in a **CCA-secure** version of RSA, called **RSA-OAEP**

- Uses a two-round Feistel network with round functions G and H, where G and H are Hash functions modeled as two independent random oracles
- Let $\ell(n)$ and $k(n)$ be two functions such that $k(n) = \Theta(n)$ and $\ell(n) + 2k(n)$ is less than the number of bits of the modulus N output by $\mathsf{GenRSA}(1^n)$

$$G : \{0,1\}^{k(n)} \to \{0,1\}^{\ell(n)+k(n)} \qquad \text{and} \qquad H : \{0,1\}^{\ell(n)+k(n)} \to \{0,1\}^{k(n)}$$

RSA-OAEP (Encryption)

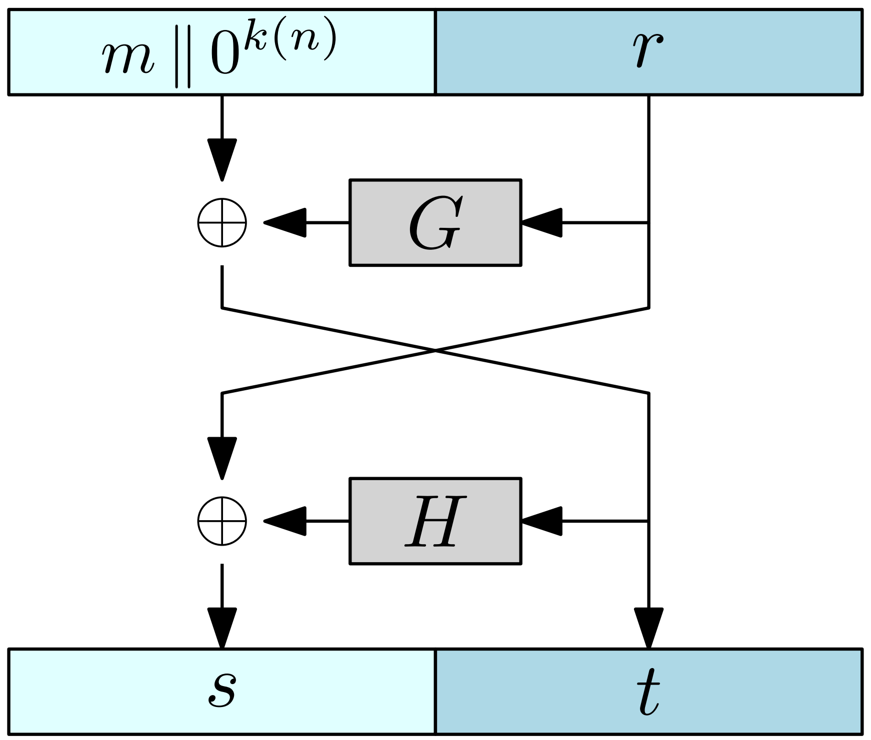
Optimal asymmetric encryption padding is a (more complex) randomized encoding of messages that results in a **CCA-secure** version of RSA, called **RSA-OAEP**

- Uses a two-round Feistel network with round functions G and H, where G and H are Hash functions modeled as two independent random oracles
- Let $\ell(n)$ and $k(n)$ be two functions such that $k(n) = \Theta(n)$ and $\ell(n) + 2k(n)$ is less than the number of bits of the modulus N output by GenRSA(1^n)

$$G : \{0,1\}^{k(n)} \to \{0,1\}^{\ell(n)+k(n)} \qquad \text{and} \qquad H : \{0,1\}^{\ell(n)+k(n)} \to \{0,1\}^{k(n)}$$

To encrypt a message m of length $\ell(n)$ using the public key $pk = \langle N, e \rangle$:

- Pick r u.a.r. from $\{0,1\}^{k(n)}$
- $t = (m \,\|\, 0^{k(n)}) \oplus G(r)$
- $s = r \oplus H(t)$
- $\hat{m} = s \,\|\, t$
- Return $c = \hat{m}^e \pmod N$



RSA-OAEP (Encryption)

Optimal asymmetric encryption padding is a (more complex) randomized encoding of messages that results in a **CCA-secure** version of RSA, called **RSA-OAEP**

- Uses a two-round Feistel network with round functions G and H, where G and H are Hash functions modeled as two independent random oracles
- Let $\ell(n)$ and $k(n)$ be two functions such that $k(n) = \Theta(n)$ and $\ell(n) + 2k(n)$ is less than the number of bits of the modulus N output by GenRSA(1^n)

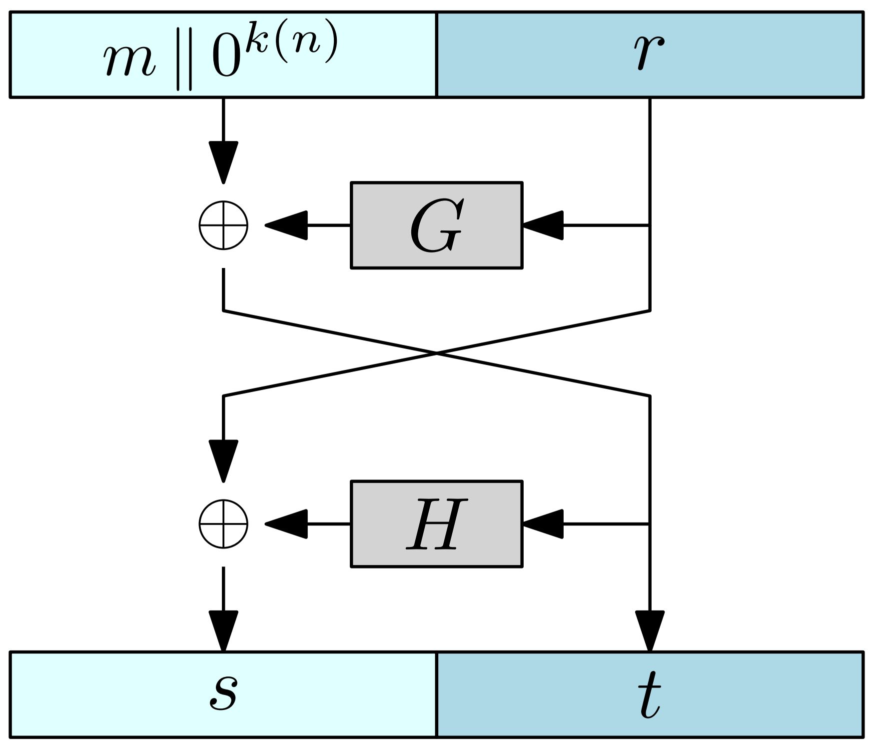
$$G : \{0,1\}^{k(n)} \to \{0,1\}^{\ell(n)+k(n)} \qquad \text{and} \qquad H : \{0,1\}^{\ell(n)+k(n)} \to \{0,1\}^{k(n)}$$

To encrypt a message m of length $\ell(n)$ using the public key $pk = \langle N, e \rangle$:

- Pick r u.a.r. from $\{0,1\}^{k(n)}$
- $t = (m \,\|\, 0^{k(n)}) \oplus G(r)$
- $s = r \oplus H(t)$
- $\hat{m} = s \,\|\, t$
- Return $c = \hat{m}^e \pmod N$

padding

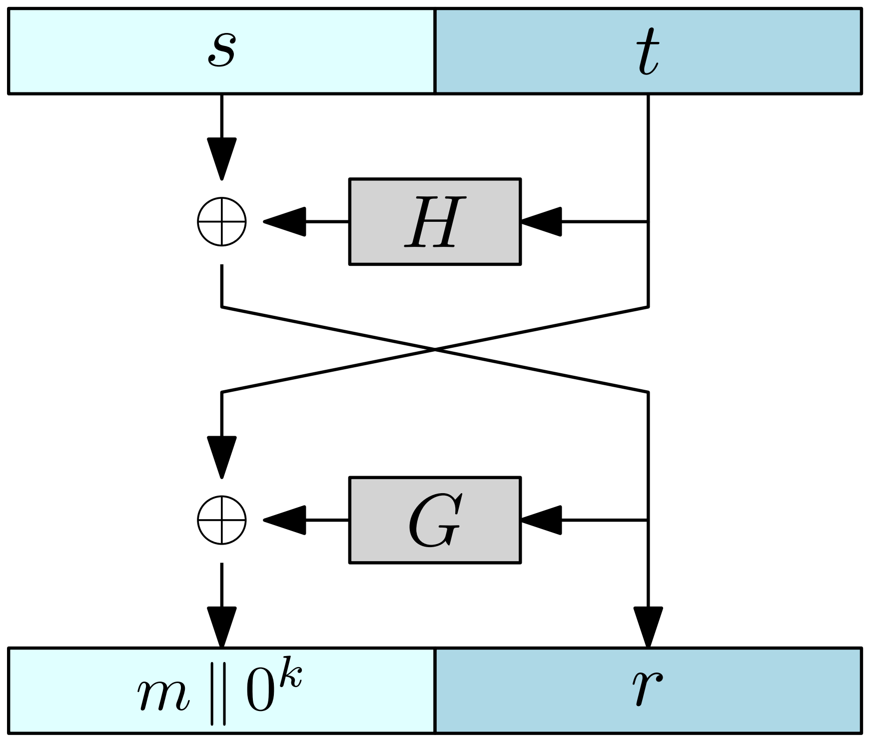
actual encryption



RSA-OAEP (Decryption)

To decrypt c, the recipient can compute $\hat{m} = c^d \pmod{N}$ and then invert the Feistel network to recover $(m \,\|\, 0^k) \,\|\, r$

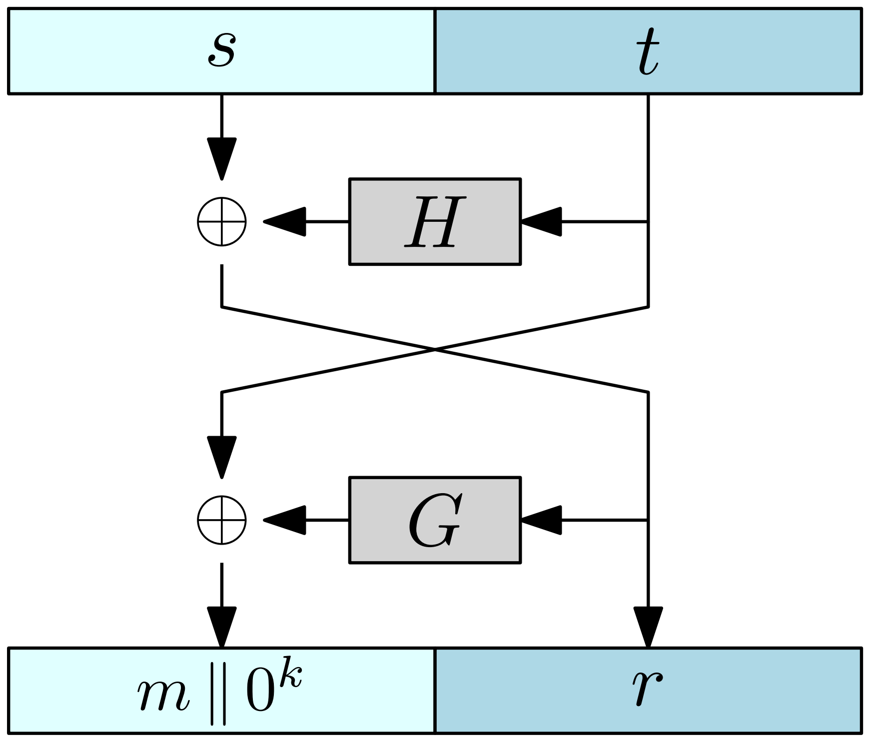
- $\hat{m} = c^d \pmod{N}$
- If $\hat{m}$ has more than $\ell(n) + 2k(n)$ bits: return $\perp$
- Interpret $\hat{m}$ as $s \,\|\, t$, where s has $k(n)$ bits and t has $\ell(n) + k(n)$ bits
- $r = s \oplus H(t)$
- $m' = t \oplus G(r)$
- If m' is of the form $m \,\|\, 0^k$: return m
- Otherwise return $\perp$



RSA-OAEP (Decryption)

To decrypt c, the recipient can compute $\hat{m} = c^d \pmod N$ and then invert the Feistel network to recover $(m \,\|\, 0^k) \,\|\, r$

- $\hat{m} = c^d \pmod N$
- If $\hat{m}$ has more than $\ell(n) + 2k(n)$ bits: return $\perp$
- Interpret $\hat{m}$ as $s \,\|\, t$, where s has $k(n)$ bits and t has $\ell(n) + k(n)$ bits
- $r = s \oplus H(t)$
- $m' = t \oplus G(r)$
- If m' is of the form $m \,\|\, 0^k$: return m
- Otherwise return $\perp$

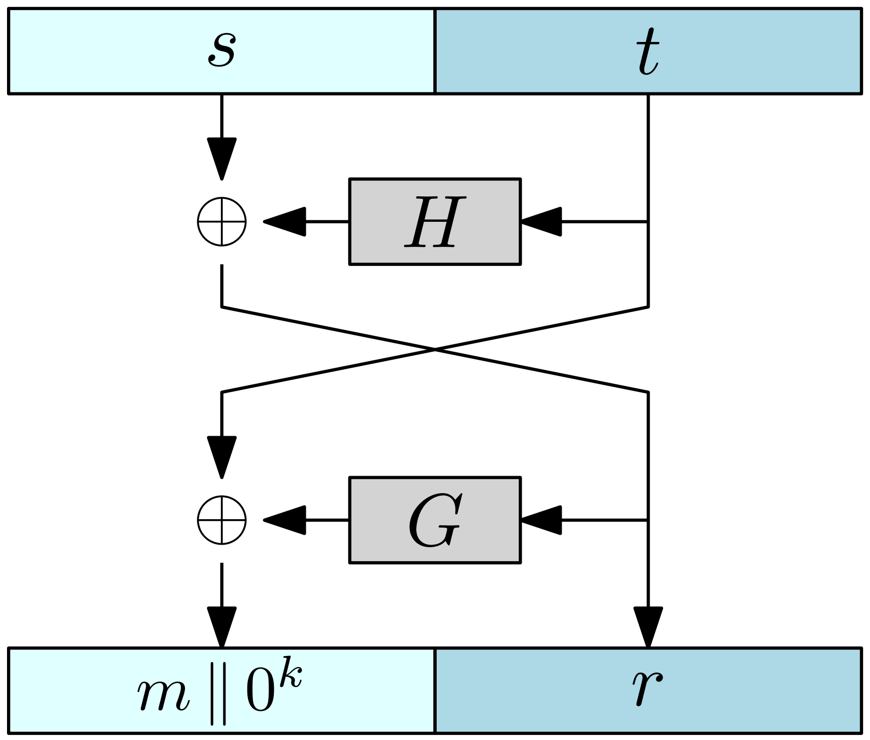


Can be proven CCA-Secure if (i) the RSA assumption holds, and (ii) G and H are modeled as independent random oracles

RSA-OAEP (Decryption)

To decrypt c, the recipient can compute $\hat{m} = c^d \pmod N$ and then invert the Feistel network to recover $(m \,\|\, 0^k) \,\|\, r$

- $\hat{m} = c^d \pmod N$
- If $\hat{m}$ has more than $\ell(n) + 2k(n)$ bits: return $\perp$
- Interpret $\hat{m}$ as $s \,\|\, t$, where s has $k(n)$ bits and t has $\ell(n) + k(n)$ bits
- $r = s \oplus H(t)$
- $m' = t \oplus G(r)$
- If m' is of the form $m \,\|\, 0^k$: return m
- Otherwise return $\perp$



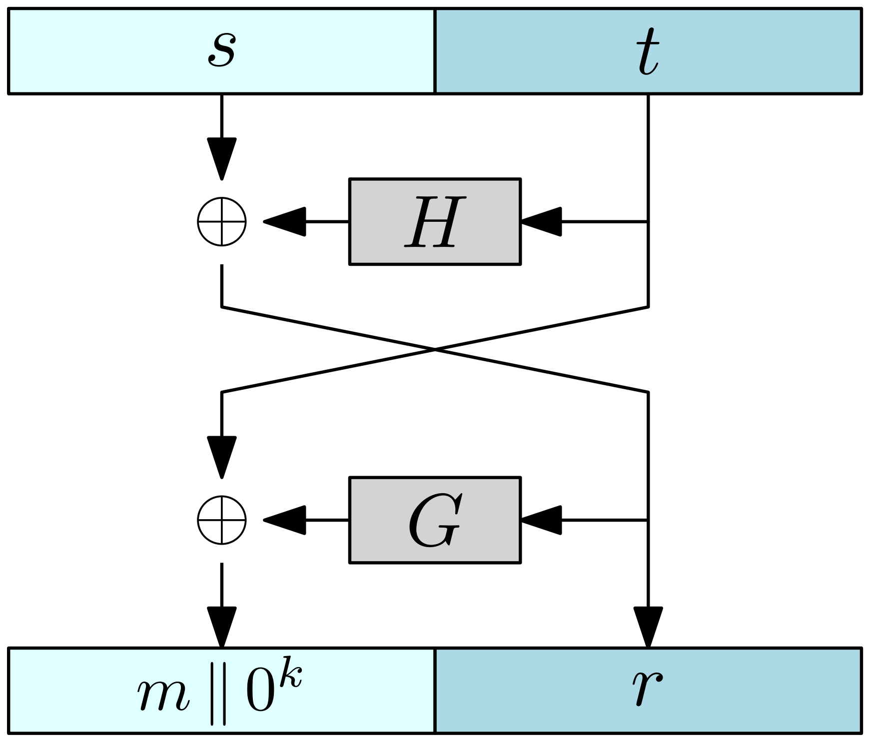
Can be proven CCA-Secure if (i) the RSA assumption holds, and (ii) G and H are modeled as independent random oracles

Warning: there are 2 sources of errors. If the returned errors differ, an attacker can recover the plaintext!

RSA-OAEP (Decryption)

To decrypt c, the recipient can compute $\hat{m} = c^d \pmod N$ and then invert the Feistel network to recover $(m \,\|\, 0^k) \,\|\, r$

- $\hat{m} = c^d \pmod N$
- If $\hat{m}$ has more than $\ell(n) + 2k(n)$ bits: return $\perp$
- Interpret $\hat{m}$ as $s \,\|\, t$, where s has $k(n)$ bits and t has $\ell(n) + k(n)$ bits
- $r = s \oplus H(t)$
- $m' = t \oplus G(r)$
- If m' is of the form $m \,\|\, 0^k$: return m
- Otherwise return $\perp$



Can be proven CCA-Secure if (i) the RSA assumption holds, and (ii) G and H are modeled as independent random oracles

Warning: there are 2 sources of errors. If the returned errors differ, an attacker can recover the plaintext!

Side channel attacks: Even if the error is the same, an attacker might be able to distinguish the two cases by observing the time elapsed before an error is returned.

“Fixing” RSA

How do we make RSA secure?

Option 1: Use randomized encoding

- Make sure that the message is “random enough”
- Choose some **randomized encoding** between messages and group elements
- To encrypt: encode the message, then encrypt the group element
- To decrypt: decrypt the ciphertext to recover the group element, then decode the group element

Option 2: Use RSA as a KEM **with a key derivation function**

RSA as a KEM

We can design a KEM based on RSA:

- We will encrypt **a random group element**
- We will use a key derivation function $H : \mathbb{Z}^* \to \{0,1\}^n$, modeled as a random oracle

Gen(1^n):

- $(N, e, d) \leftarrow \mathsf{GenRSA}(1^n)$
- Return (pk, sk) where $pk = \langle N, e\rangle$ and $sk = \langle N, d\rangle$

RSA as a KEM

We can design a KEM based on RSA:

- We will encrypt **a random group element**
- We will use a key derivation function $H : \mathbb{Z}^* \to \{0,1\}^n$, modeled as a random oracle

Gen(1^n):

- $(N, e, d) \leftarrow$ GenRSA(1^n)
- Return (pk, sk) where $pk = \langle N, e \rangle$ and $sk = \langle N, d \rangle$

Encaps$_{pk}$(1^n):

- Here $pk = \langle N, e \rangle$
- Choose a uniform r in $\mathbb{Z}_N^*$
- Output the ciphertext $c = r^e \pmod{N}$ and the key $H(r)$

RSA as a KEM

We can design a KEM based on RSA:

- We will encrypt **a random group element**
- We will use a key derivation function $H : \mathbb{Z}^* \to \{0,1\}^n$, modeled as a random oracle

Gen(1^n):

- $(N, e, d) \leftarrow$ GenRSA(1^n)
- Return (pk, sk) where $pk = \langle N, e \rangle$ and $sk = \langle N, d \rangle$

Encaps$_{pk}(1^n)$:

- Here $pk = \langle N, e \rangle$
- Choose a uniform r in $\mathbb{Z}_N^*$
- Output the ciphertext $c = r^e \pmod N$ and the key $H(r)$

Decaps$_{sk}(c)$:

- Here $sk = \langle N, d \rangle$
- Compute $r = c^d \pmod N$
- Return the key $H(r)$

RSA as a KEM: Security

Theorem: If the RSA problem is hard relative to GenRSA and H is modeled as a random oracle, then the RSA-based KEM described before is CCA-secure.

RSA as a KEM: Security

Theorem: If the RSA problem is hard relative to GenRSA and H is modeled as a random oracle, then the RSA-based KEM described before is CCA-secure.

Idea:

- We are encrypting a random group element, therefore we can rely on the hardness of the RSA problem

RSA as a KEM: Security

Theorem: If the RSA problem is hard relative to GenRSA and H is modeled as a random oracle, then the RSA-based KEM described before is CCA-secure.

Idea:

- We are encrypting a random group element, therefore we can rely on the hardness of the RSA problem
- Even if an attacker can compute a ciphertext c' related to c, the decapsulation oracle returns $H(r')$, where $r' = (c')^d \pmod N$

RSA as a KEM: Security

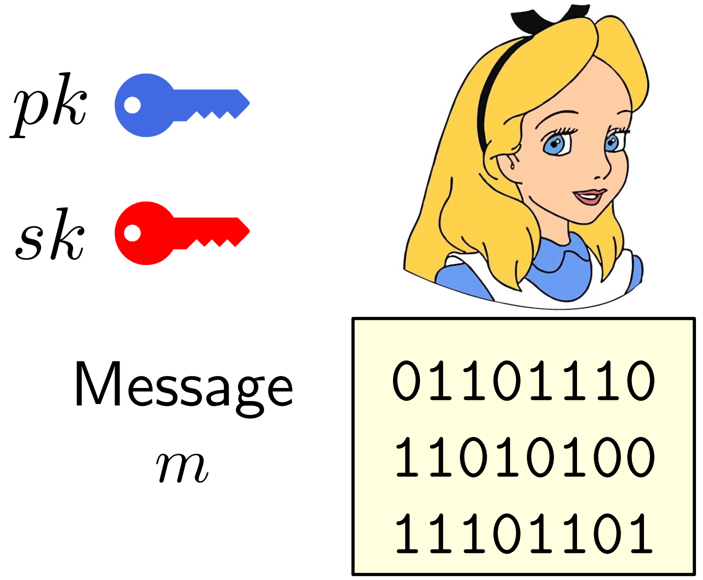
Theorem: If the RSA problem is hard relative to GenRSA and H is modeled as a random oracle, then the RSA-based KEM described before is CCA-secure.

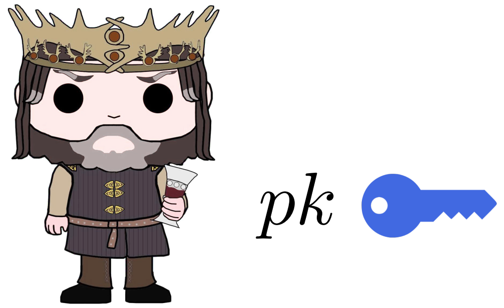
Idea:

- We are encrypting a random group element, therefore we can rely on the hardness of the RSA problem
- Even if an attacker can compute a ciphertext c' related to c, the decapsulation oracle returns $H(r')$, where $r' = (c')^d \pmod N$
- Since $r' \neq r$ and H is modeled as a random oracle, $H(r')$ is a random binary string independent of $H(r)$

Digital Signatures

We can guarantee authentication and integrity in the public-key setting by using **digital signatures**



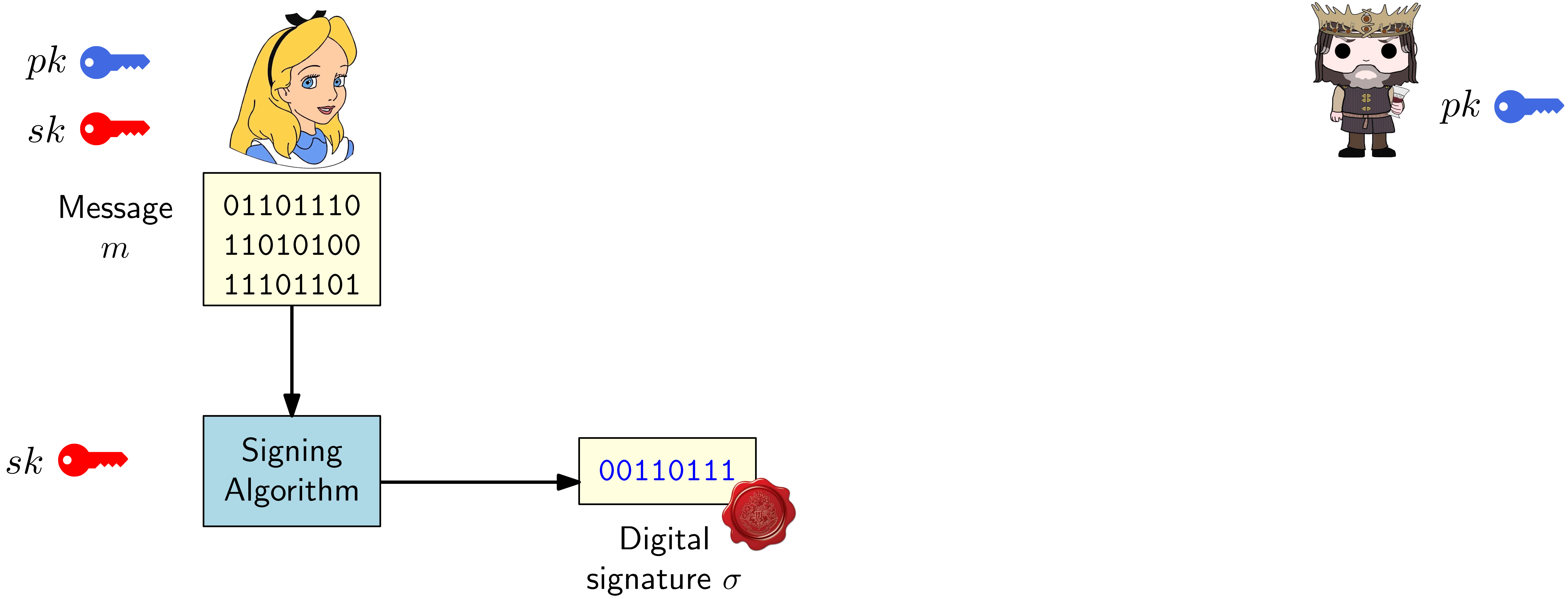


Digital Signatures

We can guarantee authentication and integrity in the public-key setting by using **digital signatures**

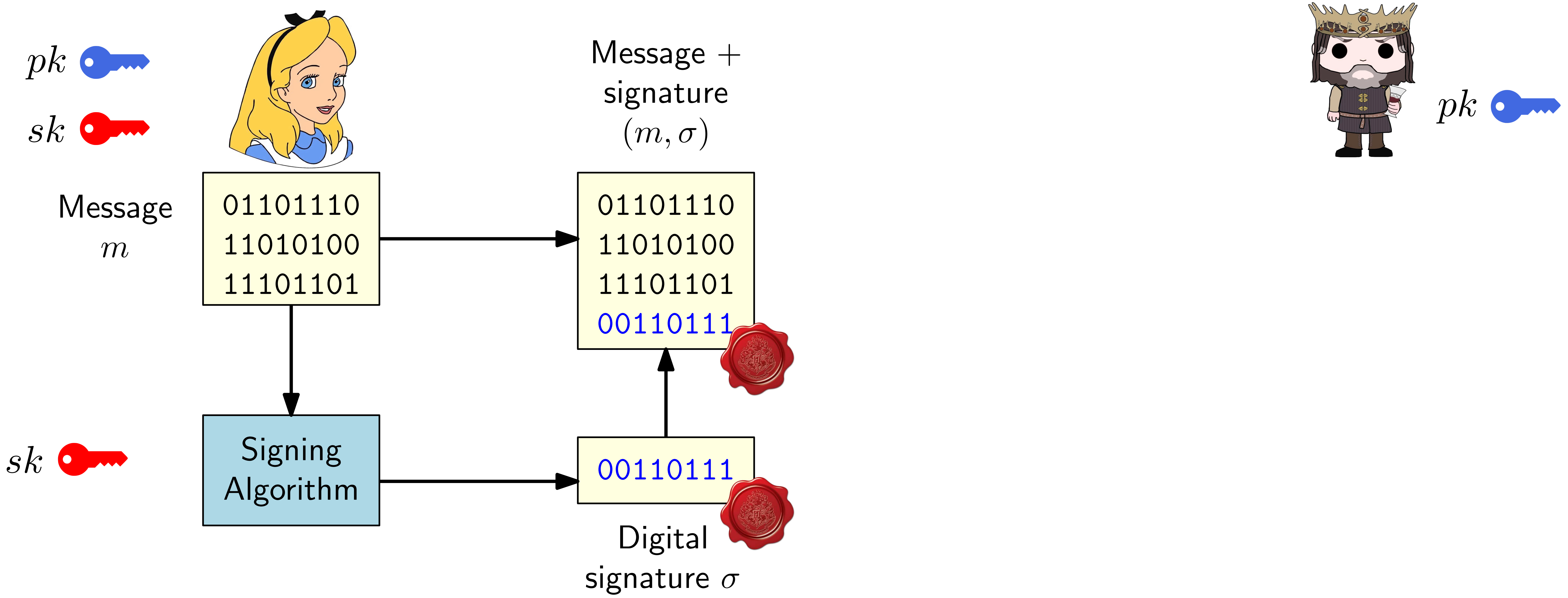
Digital Signatures

We can guarantee authentication and integrity in the public-key setting by using **digital signatures**



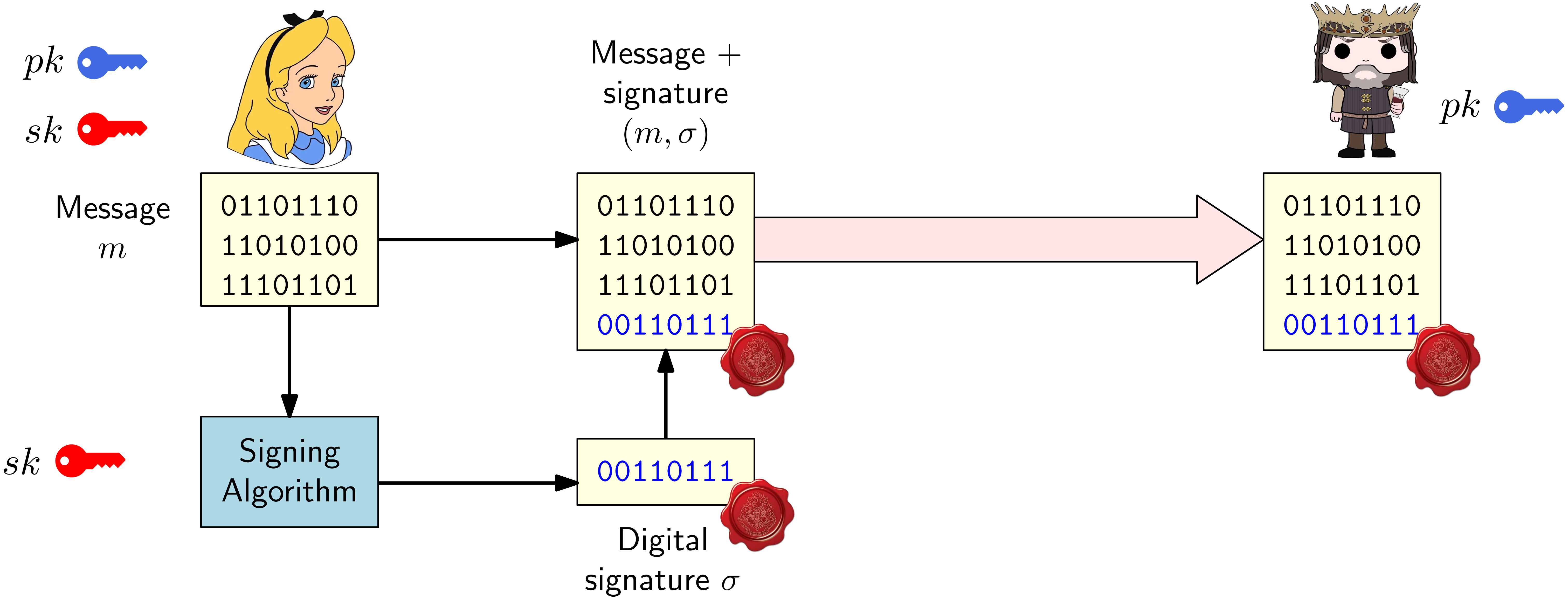
Digital Signatures

We can guarantee authentication and integrity in the public-key setting by using **digital signatures**



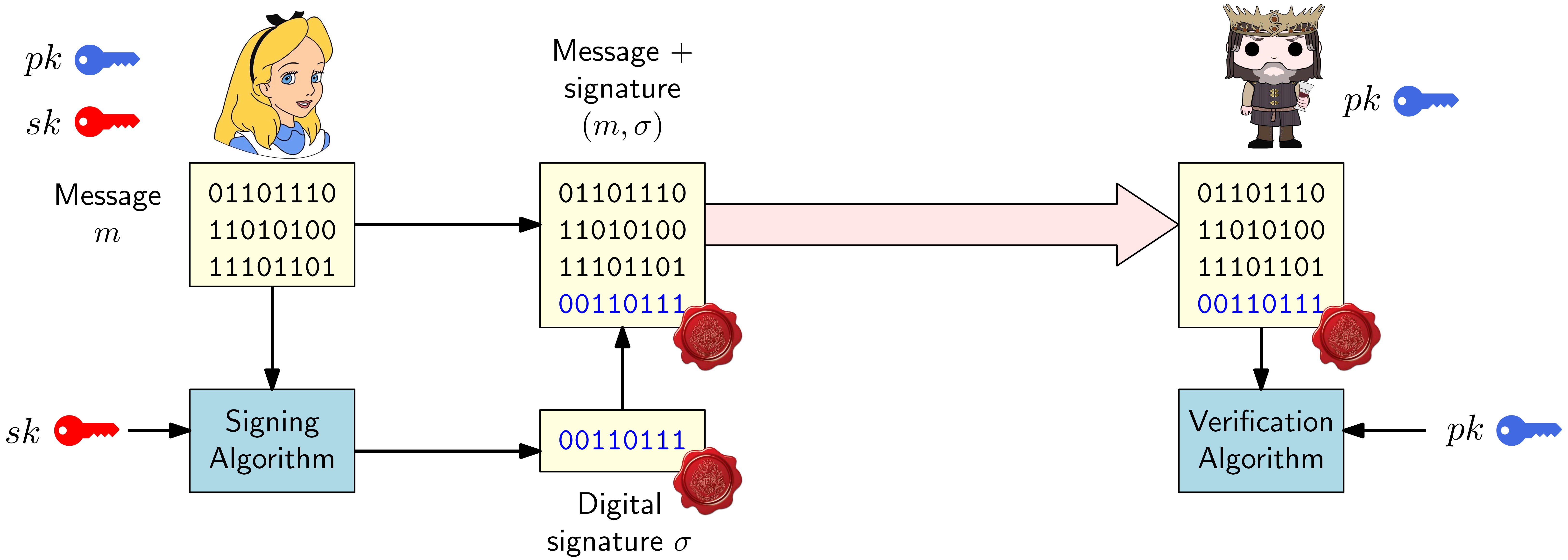
Digital Signatures

We can guarantee authentication and integrity in the public-key setting by using **digital signatures**



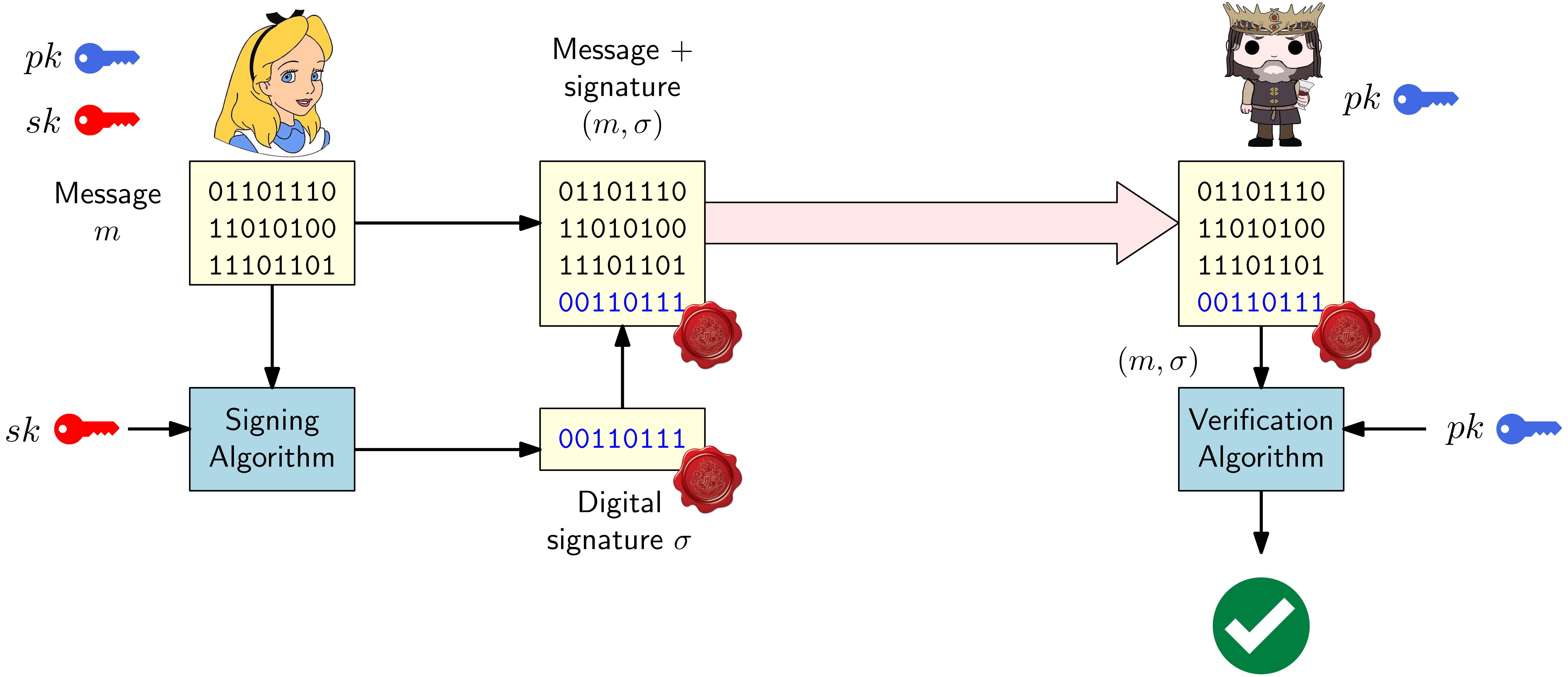
Digital Signatures

We can guarantee authentication and integrity in the public-key setting by using **digital signatures**



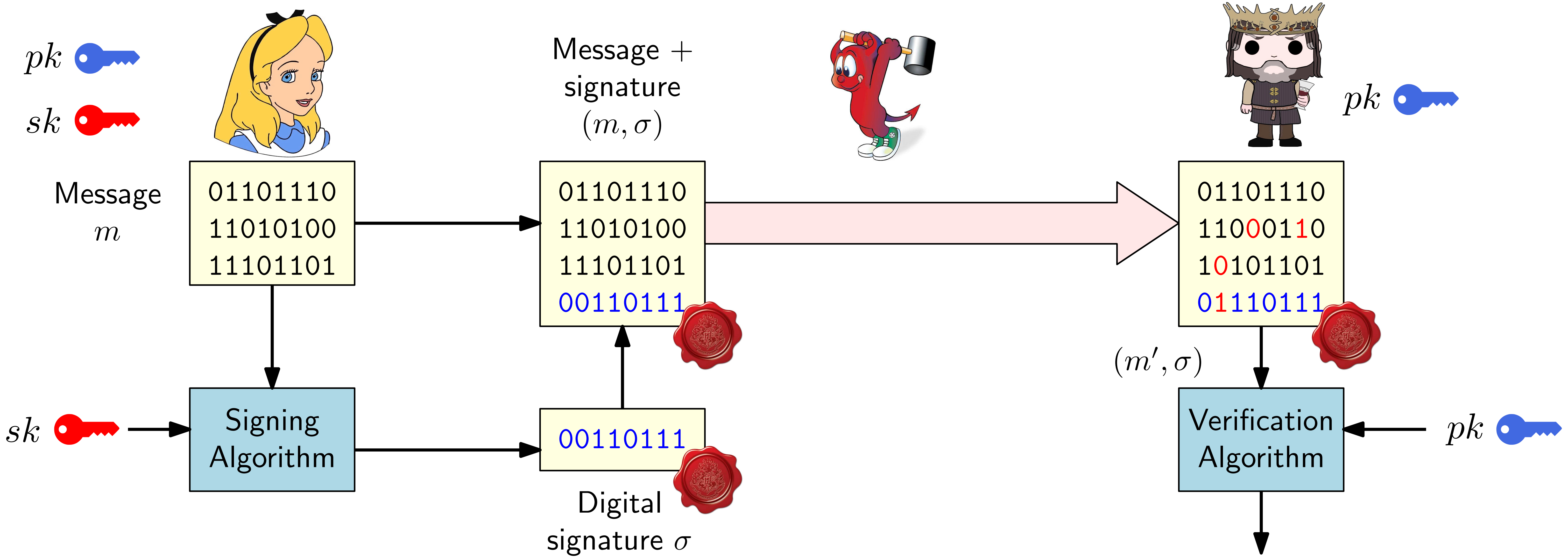
Digital Signatures

We can guarantee authentication and integrity in the public-key setting by using **digital signatures**



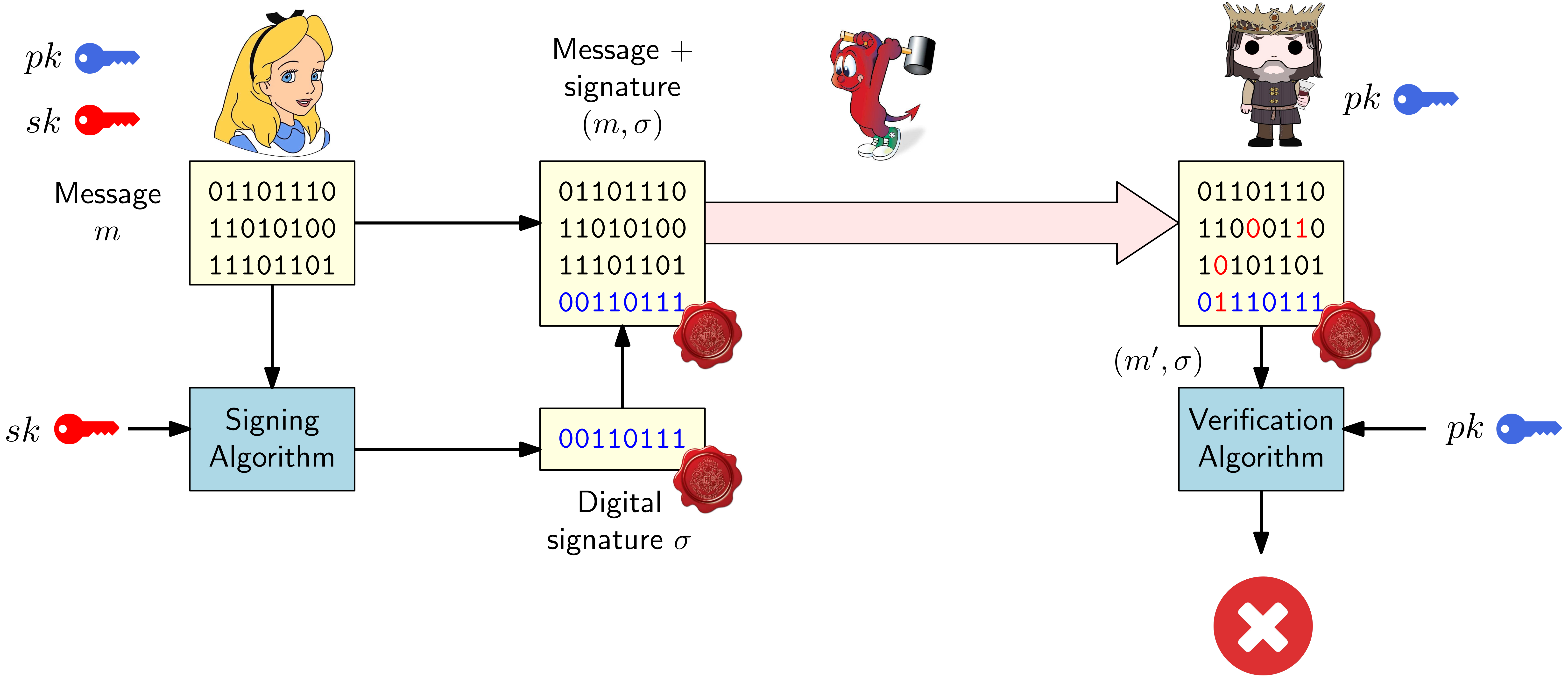
Digital Signatures

We can guarantee authentication and integrity in the public-key setting by using **digital signatures**



Digital Signatures

We can guarantee authentication and integrity in the public-key setting by using **digital signatures**



Digital Signatures vs MACs

Digital Signatures:

- **One key per sender**: A sender only needs a single secret key to be able to sign messages intended for any number of recipients.

Digital Signatures vs MACs

Digital Signatures:

- **One key per sender**: A sender only needs a single secret key to be able to sign messages intended for any number of recipients.
- **Publicly verifiable:** if a recipient verifies that a signature on a given message is legitimate, then all other parties who receive this signed message will also verify it as legitimate.

Digital Signatures vs MACs

Digital Signatures:

- **One key per sender**: A sender only needs a single secret key to be able to sign messages intended for any number of recipients.
- **Publicly verifiable:** if a recipient verifies that a signature on a given message is legitimate, then all other parties who receive this signed message will also verify it as legitimate.
- **Transferable:** A digital signature can be exhibited to a third party to convince them that the original sender of the message authenticated it

Digital Signatures vs MACs

Digital Signatures:

- **One key per sender**: A sender only needs a single secret key to be able to sign messages intended for any number of recipients.
- **Publicly verifiable:** if a recipient verifies that a signature on a given message is legitimate, then all other parties who receive this signed message will also verify it as legitimate.
- **Transferable:** A digital signature can be exhibited to a third party to convince them that the original sender of the message authenticated it
- **Non-repudiable:** Once the sender signs a message, he cannot later deny having done so. The signature acts a *proof* that the sender authenticad the message

Digital Signatures vs MACs

Digital Signatures:

- **One key per sender**: A sender only needs a single secret key to be able to sign messages intended for any number of recipients.
- **Publicly verifiable:** if a recipient verifies that a signature on a given message is legitimate, then all other parties who receive this signed message will also verify it as legitimate.
- **Transferable:** A digital signature can be exhibited to a third party to convince them that the original sender of the message authenticated it
- **Non-repudiable:** Once the sender signs a message, he cannot later deny having done so. The signature acts a *proof* that the sender authenticad the message

MACs:

- Require a different symmetric key for each recipient

Digital Signatures vs MACs

Digital Signatures:

- **One key per sender**: A sender only needs a single secret key to be able to sign messages intended for any number of recipients.
- **Publicly verifiable:** if a recipient verifies that a signature on a given message is legitimate, then all other parties who receive this signed message will also verify it as legitimate.
- **Transferable:** A digital signature can be exhibited to a third party to convince them that the original sender of the message authenticated it
- **Non-repudiable:** Once the sender signs a message, he cannot later deny having done so. The signature acts a *proof* that the sender authenticad the message

MACs:

- Require a different symmetric key for each recipient
- Consequently, that tag of each recipient differs (even if the message is the same)

Digital Signatures vs MACs

Digital Signatures:

- **One key per sender**: A sender only needs a single secret key to be able to sign messages intended for any number of recipients.
- **Publicly verifiable:** if a recipient verifies that a signature on a given message is legitimate, then all other parties who receive this signed message will also verify it as legitimate.
- **Transferable:** A digital signature can be exhibited to a third party to convince them that the original sender of the message authenticated it
- **Non-repudiable:** Once the sender signs a message, he cannot later deny having done so. The signature acts a *proof* that the sender authenticad the message

MACs:

- Require a different symmetric key for each recipient
- Consequently, that tag of each recipient differs (even if the message is the same)
- Not transferable. A third part cannot check that a tag is valid without knowing the symmetric key.

Digital Signatures vs MACs

Digital Signatures:

- **One key per sender**: A sender only needs a single secret key to be able to sign messages intended for any number of recipients.
- **Publicly verifiable:** if a recipient verifies that a signature on a given message is legitimate, then all other parties who receive this signed message will also verify it as legitimate.
- **Transferable:** A digital signature can be exhibited to a third party to convince them that the original sender of the message authenticated it
- **Non-repudiable:** Once the sender signs a message, he cannot later deny having done so. The signature acts a *proof* that the sender authenticad the message

MACs:

- Require a different symmetric key for each recipient
- Consequently, that tag of each recipient differs (even if the message is the same)
- Not transferable. A third part cannot check that a tag is valid without knowing the symmetric key.
- Repudiable: Even if the recipient reveals the symmetric key, the sender can claim that the recipient generated the tag himself!

Digital Signatures vs MACs

Digital Signatures:

Slower, Longer signatures

- **One key per sender**: A sender only needs a single secret key to be able to sign messages intended for any number of recipients.
- **Publicly verifiable:** if a recipient verifies that a signature on a given message is legitimate, then all other parties who receive this signed message will also verify it as legitimate.
- **Transferable:** A digital signature can be exhibited to a third party to convince them that the original sender of the message authenticated it
- **Non-repudiable:** Once the sender signs a message, he cannot later deny having done so. The signature acts a *proof* that the sender authenticad the message

MACs:

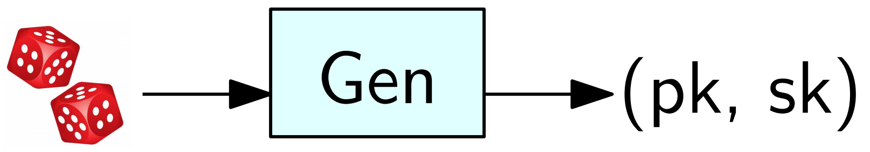
Faster (by 2,3 order of magnitude), shorter tags

- Require a different symmetric key for each recipient
- Consequently, that tag of each recipient differs (even if the message is the same)
- Not transferable. A third part cannot check that a tag is valid without knowing the symmetric key.
- Repudiable: Even if the recipient reveals the symmetric key, the sender can claim that the recipient generated the tag himself!

Digital Signature Schemes: Formal Definition

A **Digital Signature Scheme** is a triple of algorithms (Gen, Sign, Vrfy)

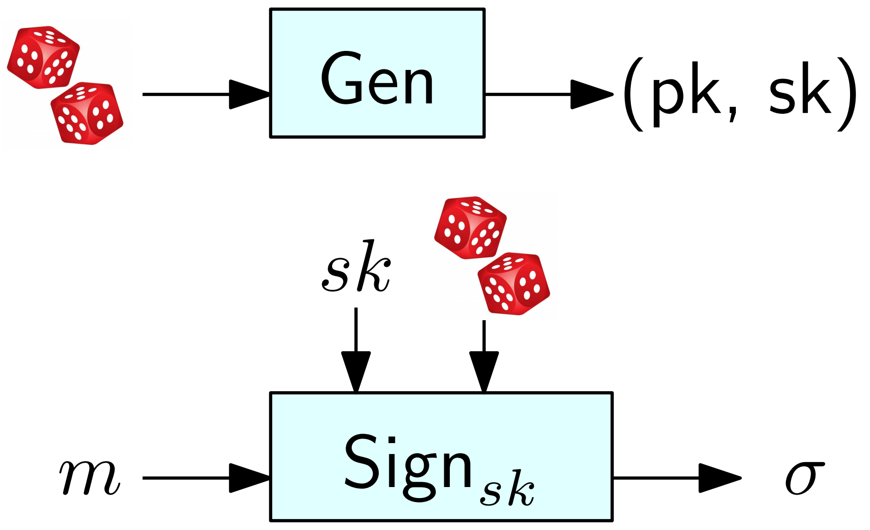
- Gen is a probabilistic polynomial-time **key-generation** algorithm that takes 1^n as input and outputs a pair (pk, sk) where pk is a public key, and sk is the secret key. This implicitly defines a message space.



Digital Signature Schemes: Formal Definition

A **Digital Signature Scheme** is a triple of algorithms (Gen, Sign, Vrfy)

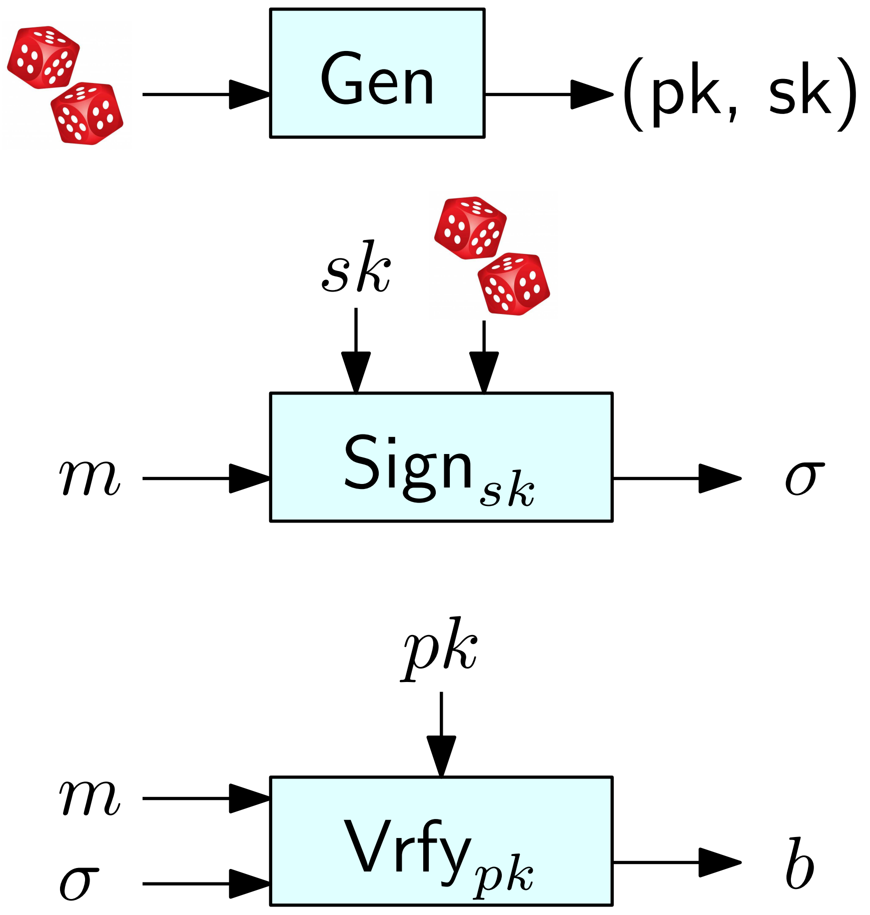
- Gen is a probabilistic polynomial-time **key-generation** algorithm that takes 1^n as input and outputs a pair (pk, sk) where pk is a public key, and sk is the secret key. This implicitly defines a message space.
- Sign is a probabilistic polynomial-time **signing** algorithm that takes as input a secret key sk and a message m (in the message space) and outputs a signature σ.



Digital Signature Schemes: Formal Definition

A **Digital Signature Scheme** is a triple of algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$

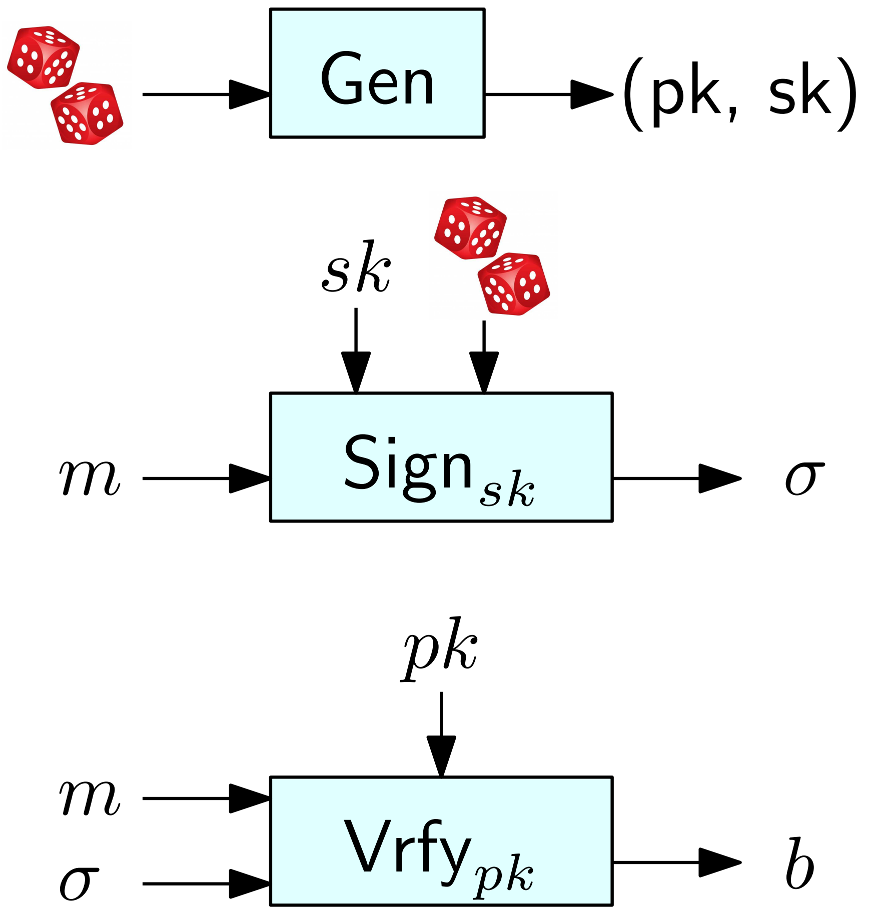
- Gen is a probabilistic polynomial-time **key-generation** algorithm that takes 1^n as input and outputs a pair (pk, sk) where pk is a public key, and sk is the secret key. This implicitly defines a message space.
- Sign is a probabilistic polynomial-time **signing** algorithm that takes as input a secret key sk and a message m (in the message space) and outputs a signature σ.
- Vrfy is a deterministic polynomial-time **verification** algorithm that takes as input a public key pk, a message m, and a signature σ, and outputs a single bit b. If $b = 1$ then the signature is valid (for pk and m), otherwise $(b = 0)$ the signature is **invalid**.



Digital Signature Schemes: Formal Definition

A **Digital Signature Scheme** is a triple of algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$

- Gen is a probabilistic polynomial-time **key-generation** algorithm that takes 1^n as input and outputs a pair (pk, sk) where pk is a public key, and sk is the secret key. This implicitly defines a message space.
- Sign is a probabilistic polynomial-time **signing** algorithm that takes as input a secret key sk and a message m (in the message space) and outputs a signature σ.
- Vrfy is a deterministic polynomial-time **verification** algorithm that takes as input a public key pk, a message m, and a signature σ, and outputs a single bit b. If $b = 1$ then the signature is valid (for pk and m), otherwise $(b = 0)$ the signature is **invalid**.

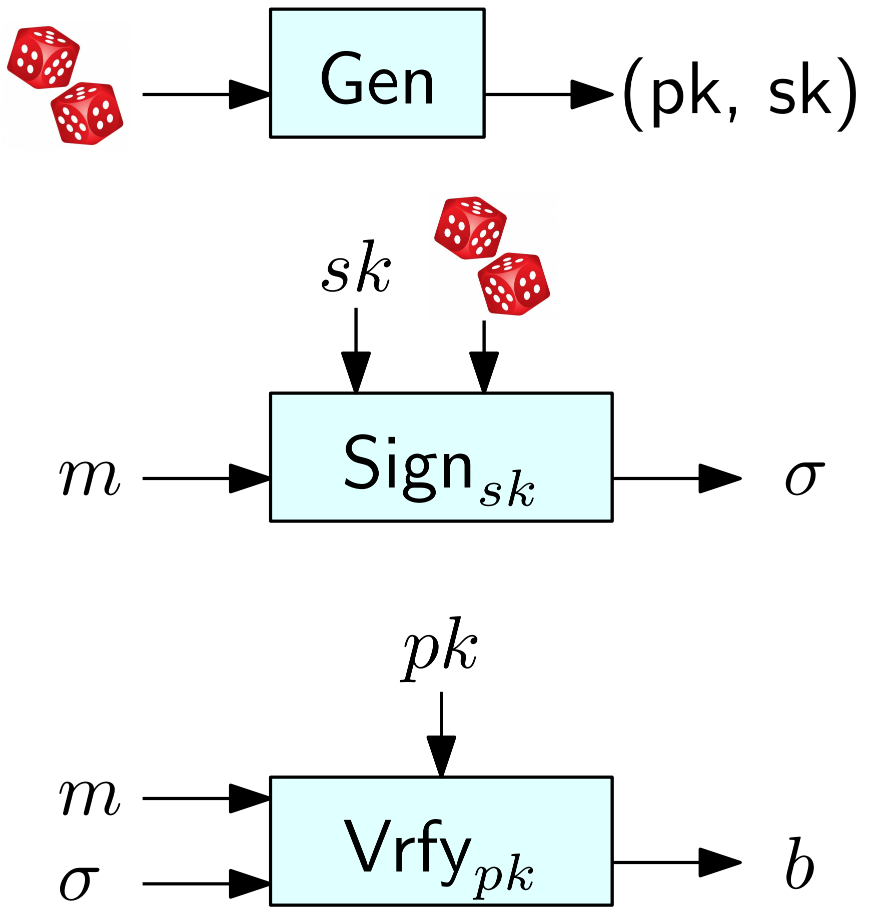


Correctness: We require that $\Pr[\mathsf{Vrfy}_{pk}(m, \mathsf{Sign}_{sk}(m)) = 1] = 1$ for every message m (here the key-pair (pk, sk) is output by $\mathsf{Gen}(1^n)$).

Digital Signature Schemes: Formal Definition

A **Digital Signature Scheme** is a triple of algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$

- Gen is a probabilistic polynomial-time **key-generation** algorithm that takes 1^n as input and outputs a pair (pk, sk) where pk is a public key, and sk is the secret key. This implicitly defines a message space.
- Sign is a probabilistic polynomial-time **signing** algorithm that takes as input a secret key sk and a message m (in the message space) and outputs a signature σ.
- Vrfy is a deterministic polynomial-time **verification** algorithm that takes as input a public key pk, a message m, and a signature σ, and outputs a single bit b. If $b = 1$ then the signature is valid (for pk and m), otherwise $(b = 0)$ the signature is **invalid**.



Correctness: We require that $\Pr[\mathsf{Vrfy}_{pk}(m, \mathsf{Sign}_{sk}(m)) = 1] = 1$ for every message m (here the key-pair (pk, sk) is output by $\mathsf{Gen}(1^n)$).

If the message space is $\{0,1\}^{\ell(n)}$, we call $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ a signature scheme for messages of length $\ell(n)$.

Digital Signature Schemes: Security Definition

Let $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ be a digital signature scheme. We name the following experiment $\mathsf{Sig\text{-}forge}_{\mathcal{A},\Pi}(n)$:

- A key-pair (pk, sk) is generated using $\mathsf{Gen}(1^n)$
- The public key pk is sent to the adversary
- The adversary can interact with an oracle that can be queried with a message m' and outputs a signature σ' obtained by running $\mathsf{Sign}_{sk}(m')$
- The adversary outputs a pair (m, σ) such that (*) no query with the message m has been performed
- The outcome of the experiment is 1 if (*) holds and $\mathsf{Vrfy}_k(m, \sigma) = 1$. Otherwise the outcome is 0.

Digital Signature Schemes: Security Definition

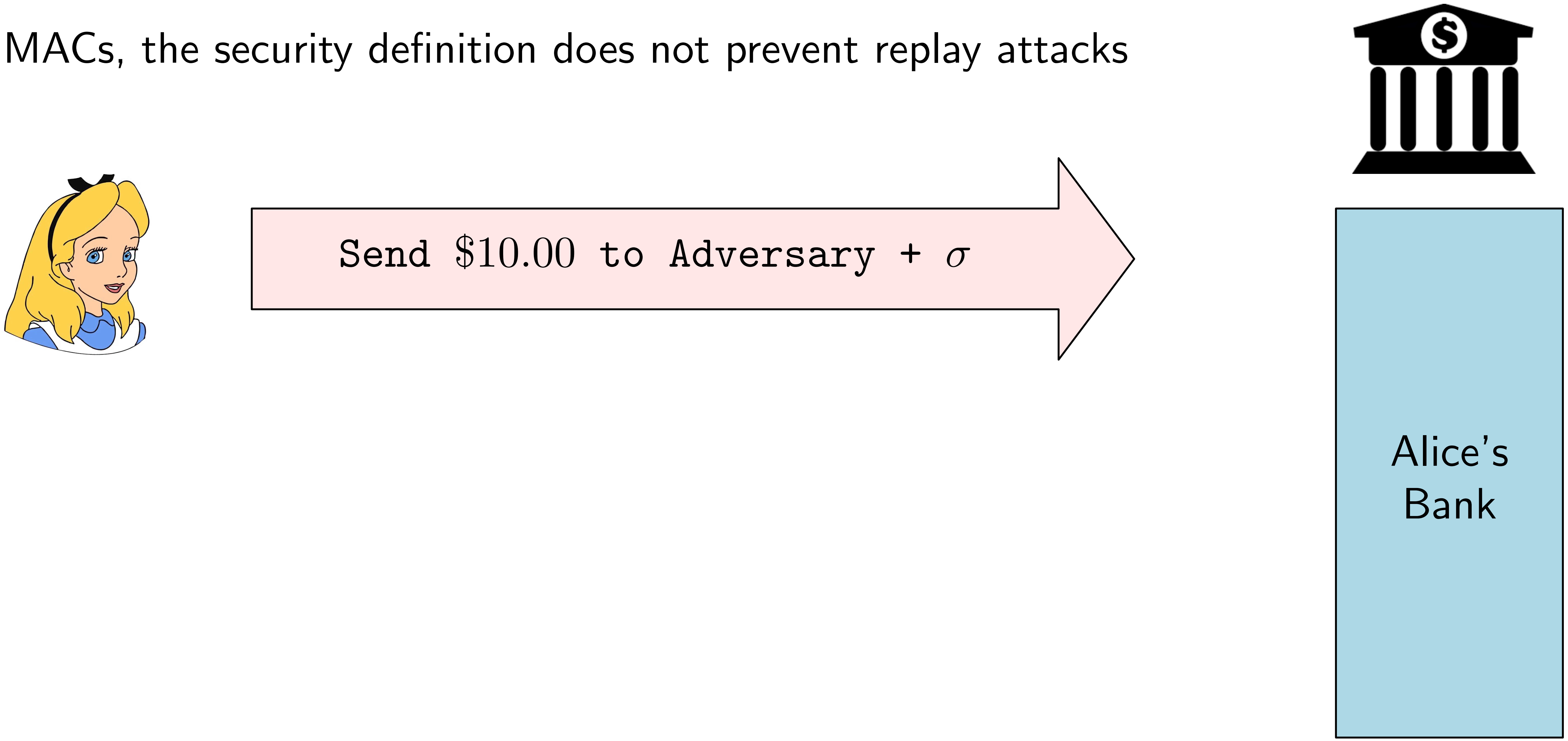
Let $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ be a digital signature scheme. We name the following experiment Sig-forge$_{\mathcal{A},\Pi}(n)$:

- A key-pair (pk, sk) is generated using $\mathsf{Gen}(1^n)$
- The public key pk is sent to the adversary
- The adversary can interact with an oracle that can be queried with a message m' and outputs a signature σ' obtained by running $\mathsf{Sign}_{sk}(m')$
- The adversary outputs a pair (m, σ) such that (*) no query with the message m has been performed
- The outcome of the experiment is 1 if (*) holds and $\mathsf{Vrfy}_k(m, \sigma) = 1$. Otherwise the outcome is 0.

> **Definition**: *A digital signature scheme Π is existentially unforgeable under an adaptive chosen-message attack (is* **secure***) if, for every probabilistic polynomial-time adversary $\mathcal{A}$, there is a negligible function ε such that:*
>
> $$\Pr[\textit{Sig-forge}_{\mathcal{A},\Pi}(n) = 1] \leq \varepsilon(n)$$

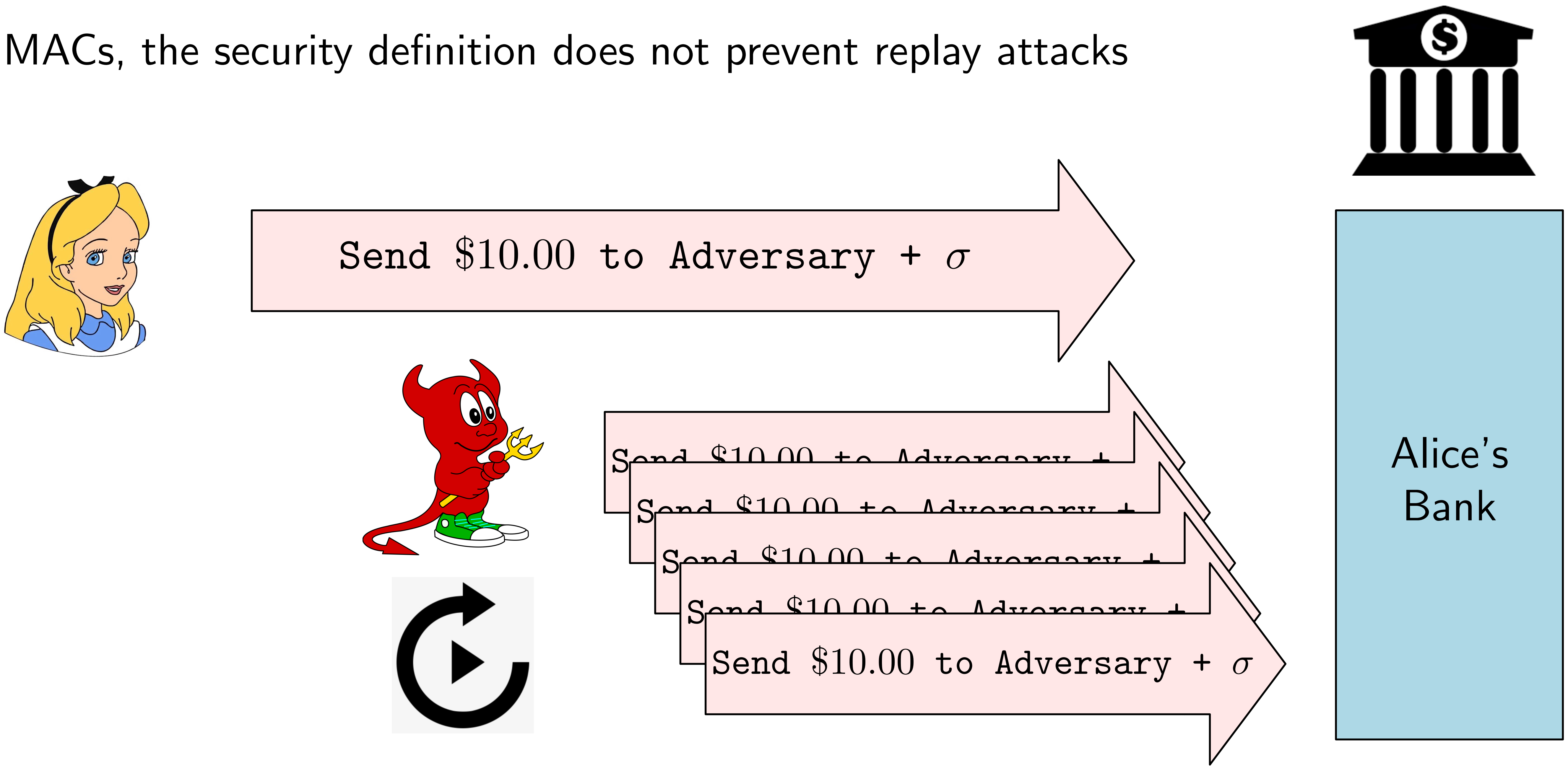
Some Remarks on the Security Definition

Just like for MACs, the security definition does not prevent replay attacks



Some Remarks on the Security Definition

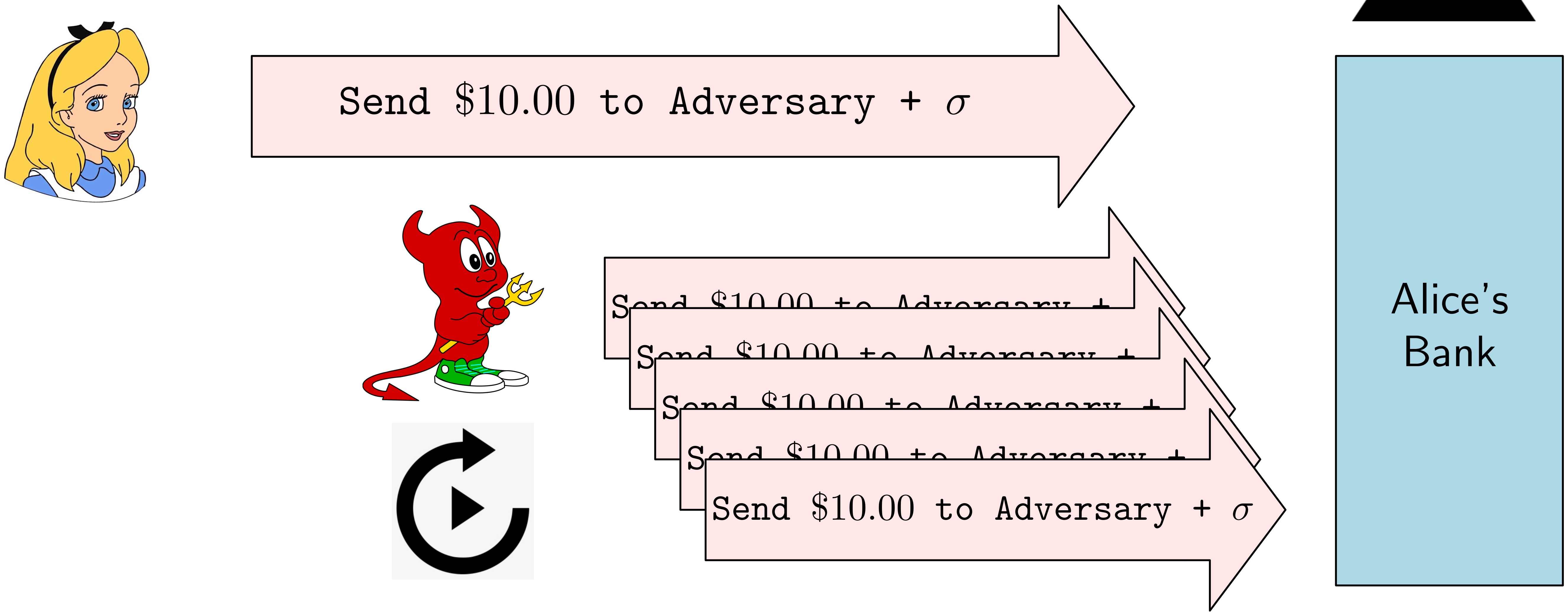
Just like for MACs, the security definition does not prevent replay attacks



Some Remarks on the Security Definition

Just like for MACs, the security definition does not prevent replay attacks

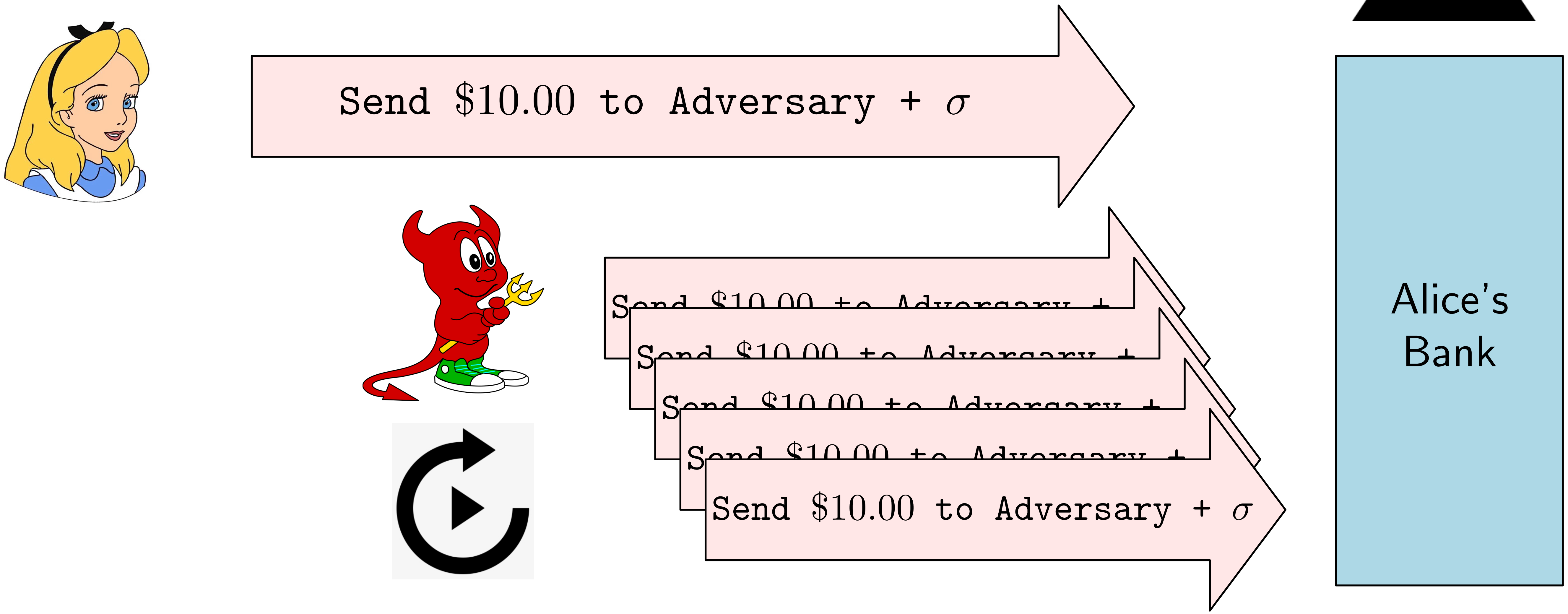
- Similar mitigations as for MACs can be used



Some Remarks on the Security Definition

Just like for MACs, the security definition does not prevent replay attacks

- Similar mitigations as for MACs can be used

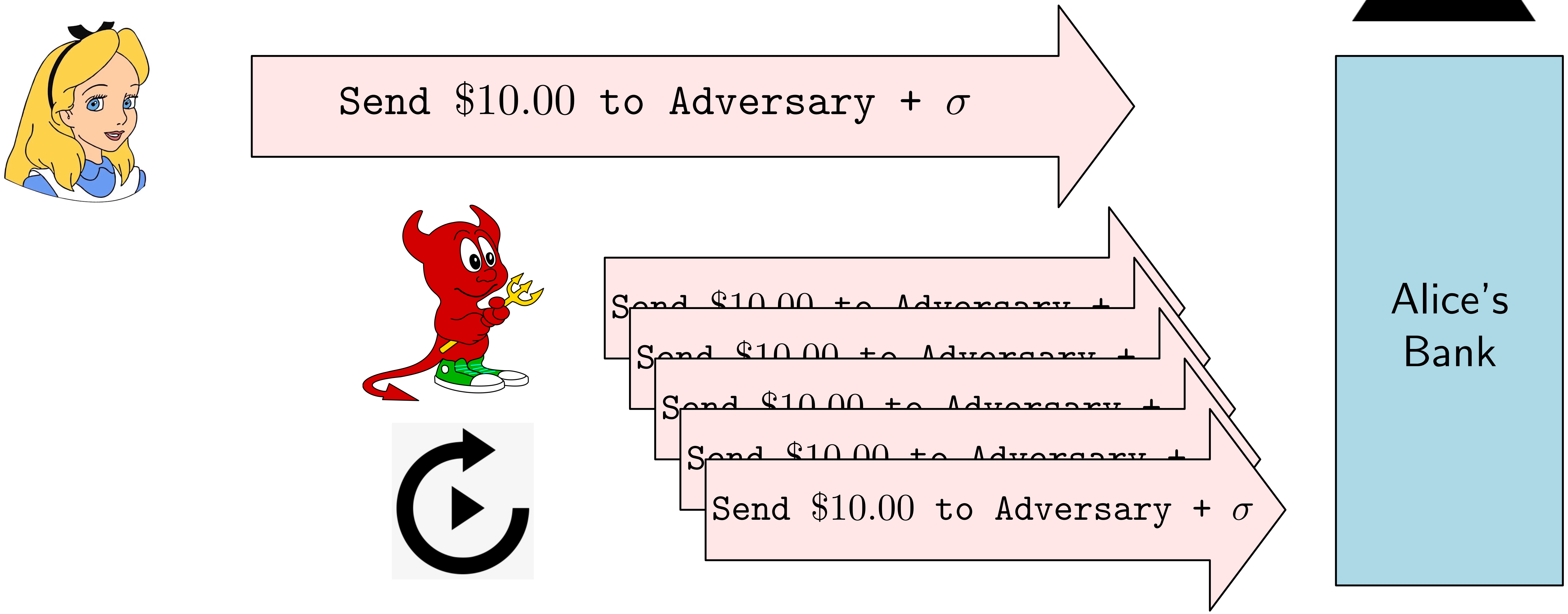


The security definition does not prevent the adversary from outputting a new valid signature for a message that was previously signed

Some Remarks on the Security Definition

Just like for MACs, the security definition does not prevent replay attacks

- Similar mitigations as for MACs can be used



The security definition does not prevent the adversary from outputting a new valid signature for a message that was previously signed

Strongly Secure Digital Signature schemes can be defined to account for this (similarly to the modified experiment for MACs)

The Hash-and-Sign Paradigm

Say that we have a digital signature scheme $\Pi' = (\mathsf{Gen}', \mathsf{Sign}', \mathsf{Vrfy})'$ for messages of length ℓ, but we would like to sign arbitrary length messages

The Hash-and-Sign Paradigm

Say that we have a digital signature scheme $\Pi' = (\mathsf{Gen}', \mathsf{Sign}', \mathsf{Vrfy})'$ for messages of length ℓ, but we would like to sign arbitrary length messages

Idea:

- Combine Π' with a Hash function $\mathcal{H} = (\mathsf{Gen}_H, H)$ where $H^s : \{0,1\}^* \to \{0,1\}^{\ell(n)}$
- Instead of signing m, sign a the hash $H^s(m)$ of m
- Analogous to Hash-and-MAC !

The Hash-and-Sign Paradigm

Say that we have a digital signature scheme $\Pi' = (\mathsf{Gen}', \mathsf{Sign}', \mathsf{Vrfy})'$ for messages of length ℓ, but we would like to sign arbitrary length messages

Idea:

- Combine Π' with a Hash function $\mathcal{H} = (\mathsf{Gen}_H, H)$ where $H^s : \{0,1\}^* \to \{0,1\}^{\ell(n)}$
- Instead of signing m, sign a the hash $H^s(m)$ of m
- Analogous to Hash-and-MAC !

Gen(1^n):

- $(pk, sk) \leftarrow \mathsf{Gen}'(1^n)$
- $s \leftarrow \mathsf{Gen}_H(1^n)$
- Return the public key $\langle pk, s\rangle$ and the private-key $\langle sk, s\rangle$

The Hash-and-Sign Paradigm

Say that we have a digital signature scheme $\Pi' = (\mathsf{Gen}', \mathsf{Sign}', \mathsf{Vrfy})'$ for messages of length ℓ, but we would like to sign arbitrary length messages

Idea:

- Combine Π' with a Hash function $\mathcal{H} = (\mathsf{Gen}_H, H)$ where $H^s : \{0,1\}^* \to \{0,1\}^{\ell(n)}$
- Instead of signing m, sign a the hash $H^s(m)$ of m
- Analogous to Hash-and-MAC !

Gen(1^n):

- $(pk, sk) \leftarrow \mathsf{Gen}'(1^n)$
- $s \leftarrow \mathsf{Gen}_H(1^n)$
- Return the public key $\langle pk, s\rangle$ and the private-key $\langle sk, s\rangle$

Sign$_{\langle pk,s\rangle}(m)$:

- Return $\mathsf{Sign}'_{sk}(H^s(m))$

The Hash-and-Sign Paradigm

Say that we have a digital signature scheme $\Pi' = (\mathsf{Gen}', \mathsf{Sign}', \mathsf{Vrfy})'$ for messages of length ℓ, but we would like to sign arbitrary length messages

Idea:

- Combine Π' with a Hash function $\mathcal{H} = (\mathsf{Gen}_H, H)$ where $H^s : \{0,1\}^* \to \{0,1\}^{\ell(n)}$
- Instead of signing m, sign a the hash $H^s(m)$ of m
- Analogous to Hash-and-MAC !

Gen(1^n):

- $(pk, sk) \leftarrow \mathsf{Gen}'(1^n)$
- $s \leftarrow \mathsf{Gen}_H(1^n)$
- Return the public key $\langle pk, s\rangle$ and the private-key $\langle sk, s\rangle$

Sign$_{\langle pk,s\rangle}(m)$:

- Return $\mathsf{Sign}'_{sk}(H^s(m))$

Vrfy$_{\langle pk,s\rangle}(m, \sigma)$:

- Return $\mathsf{Vrfy}'_{pk}(H^s(m), \sigma)$

The Hash-and-Sign Paradigm

Say that we have a digital signature scheme $\Pi' = (\mathsf{Gen}', \mathsf{Sign}', \mathsf{Vrfy})'$ for messages of length ℓ, but we would like to sign arbitrary length messages

Idea:

- Combine Π' with a Hash function $\mathcal{H} = (\mathsf{Gen}_H, H)$ where $H^s : \{0,1\}^* \to \{0,1\}^{\ell(n)}$
- Instead of signing m, sign a the hash $H^s(m)$ of m
- Analogous to Hash-and-MAC !

Gen(1^n):

- $(pk, sk) \leftarrow \mathsf{Gen}'(1^n)$
- $s \leftarrow \mathsf{Gen}_H(1^n)$
- Return the public key $\langle pk, s\rangle$ and the private-key $\langle sk, s\rangle$

Sign$_{\langle pk,s\rangle}(m)$:

- Return $\mathsf{Sign}'_{sk}(H^s(m))$

Vrfy$_{\langle pk,s\rangle}(m, \sigma)$:

- Return $\mathsf{Vrfy}'_{pk}(H^s(m), \sigma)$

Theorem: if Π' is a secure digital signature scheme for messages of length ℓ and $\mathcal{H}$ is collision resistant, then the above construction is a secure digital signature scheme

The Hash-and-Sign Paradigm

Say that we have a digital signature scheme $\Pi' = (\mathsf{Gen}', \mathsf{Sign}', \mathsf{Vrfy})'$ for messages of length ℓ, but we would like to sign arbitrary length messages

Idea:

- Combine Π' with a Hash function $\mathcal{H} = (\mathsf{Gen}_H, H)$ where $H^s : \{0,1\}^* \to \{0,1\}^{\ell(n)}$
- Instead of signing m, sign a the hash $H^s(m)$ of m
- Analogous to Hash-and-MAC !

Gen(1^n):

- $(pk, sk) \leftarrow \mathsf{Gen}'(1^n)$
- $s \leftarrow \mathsf{Gen}_H(1^n)$
- Return the public key $\langle pk, s \rangle$ and the private-key $\langle sk, s \rangle$

Sign$_{\langle pk,s \rangle}(m)$:

- Return $\mathsf{Sign}'_{sk}(H^s(m))$

Vrfy$_{\langle pk,s \rangle}(m, \sigma)$:

- Return $\mathsf{Vrfy}'_{pk}(H^s(m), \sigma)$

Theorem: if Π' is a secure digital signature scheme for messages of length ℓ and $\mathcal{H}$ is collision resistant, then the above construction is a secure digital signature scheme

We can focus on designing fixed-length digital signature schemes.

Signature schemes in practice

Signature schemes based on RSA

Can be proven secure under the RSA assumption in the random oracle model

Longer signatures, slower

Signature schemes in practice

Signature schemes based on RSA

Can be proven secure under the RSA assumption in the random oracle model

Longer signatures, slower

Signature schemes based on Discrete Logarithms

Shorter signatures, faster

- Schnorr:

  Can be proven secure under the discrete logarithm assumption in the random oracle model

Signature schemes in practice

Signature schemes based on RSA

Can be proven secure under the RSA assumption in the random oracle model

Longer signatures, slower

Signature schemes based on Discrete Logarithms

Shorter signatures, faster

- Schnorr:

  Can be proven secure under the discrete logarithm assumption in the random oracle model

- DSA/ECDSA:

  Widely used

  No proof of security if implemented according to the Digital Signature Standard of NIST

Signature schemes in practice

Signature schemes based on RSA

Can be proven secure under the RSA assumption in the random oracle model

Longer signatures, slower

Signature schemes based on Discrete Logarithms

Shorter signatures, faster

- Schnorr:

  Can be proven secure under the discrete logarithm assumption in the random oracle model

- DSA/ECDSA:

  Widely used

  No proof of security if implemented according to the Digital Signature Standard of NIST

Plain RSA Signatures

Idea: Reverse the roles of sender and receiver, and of the public and private-keys

Plain RSA Signatures

Idea: Reverse the roles of sender and receiver, and of the public and private-keys

- The sender "encrypts" m using its private key . The resulting "ciphertext" is the digital signature

Plain RSA Signatures

Idea: Reverse the roles of sender and receiver, and of the public and private-keys

- The sender "encrypts" m using its private key . The resulting "ciphertext" is the digital signature

- The receiver "decrypts" the digital signature with the public key of the sender and checks that recovered "plaintext" matches the message

Plain RSA Signatures

Idea: Reverse the roles of sender and receiver, and of the public and private-keys

- The sender "encrypts" m using its private key . The resulting "ciphertext" is the digital signature

- The receiver "decrypts" the digital signature with the public key of the sender and checks that recovered "plaintext" matches the message

Gen(1^n):

- $(N, e, d) \leftarrow \text{GenRSA}(1^n)$
- Return (pk, sk) where $pk = \langle N, e \rangle$ and $sk = \langle N, d \rangle$

Same as RSA Encryption

Plain RSA Signatures

Idea: Reverse the roles of sender and receiver, and of the public and private-keys

- The sender "encrypts" m using its private key . The resulting "ciphertext" is the digital signature
- The receiver "decrypts" the digital signature with the public key of the sender and checks that recovered "plaintext" matches the message

Gen(1^n):

- $(N, e, d) \leftarrow$ GenRSA(1^n)
- Return (pk, sk) where $pk = \langle N, e \rangle$ and $sk = \langle N, d \rangle$

Same as RSA Encryption

Sign$_{sk}(m)$:

- Here $sk = \langle N, d \rangle$ and $m \in Z_N^*$
- $\sigma \leftarrow m^d \pmod N$
- Return σ

Plain RSA Signatures

Idea: Reverse the roles of sender and receiver, and of the public and private-keys

- The sender "encrypts" m using its private key . The resulting "ciphertext" is the digital signature
- The receiver "decrypts" the digital signature with the public key of the sender and checks that recovered "plaintext" matches the message

Gen(1^n):

- $(N, e, d) \leftarrow \text{GenRSA}(1^n)$
- Return (pk, sk) where $pk = \langle N, e \rangle$ and $sk = \langle N, d \rangle$

Same as RSA Encryption

Sign$_{sk}(m)$:

- Here $sk = \langle N, d \rangle$ and $m \in Z_N^*$
- $\sigma \leftarrow m^d \pmod N$
- Return σ

Vrfy$_{pk}(m, \sigma)$:

- Here $pk = \langle N, e \rangle$ and $\sigma \in Z_N^*$
- $\widetilde{m} \leftarrow \sigma^e \pmod N$
- Return 1 if $\widetilde{m} = m$ and 0 otherwise

Security of Plain RSA Signatures

Are plain RSA signatures secure (under the RSA assumption)?

Security of Plain RSA Signatures

Are plain RSA signatures secure (under the RSA assumption)? **No!**

Security of Plain RSA Signatures

Are plain RSA signatures secure (under the RSA assumption)? **No!**

- The RSA assumption only implies hardness of computing a signature (the e-th root m^d) of a uniform message m
- It says nothing about specific messages m that are **chosen by the adversary**

Security of Plain RSA Signatures

Are plain RSA signatures secure (under the RSA assumption)? **No!**

- The RSA assumption only implies hardness of computing a signature (the e-th root m^d) of a uniform message m
- It says nothing about specific messages m that are **chosen by the adversary**

Some messages are easy to sign:

- $1^d = 1 \pmod N$, therefore $\sigma = 1$ is a valid signature for $m = 1$

Security of Plain RSA Signatures

Are plain RSA signatures secure (under the RSA assumption)? **No!**

- The RSA assumption only implies hardness of computing a signature (the e-th root m^d) of a uniform message m
- It says nothing about specific messages m that are **chosen by the adversary**

Some messages are easy to sign:

- $1^d = 1 \pmod N$, therefore $\sigma = 1$ is a valid signature for $m = 1$
- In general, if $m = x^e < N$ then $m^d = x^{ed} = x$ therefore $\sigma = x$ is a valid signature for m

Security of Plain RSA Signatures

Are plain RSA signatures secure (under the RSA assumption)? **No!**

- The RSA assumption only implies hardness of computing a signature (the e-th root m^d) of a uniform message m
- It says nothing about specific messages m that are **chosen by the adversary**

Some messages are easy to sign:

- $1^d = 1 \pmod N$, therefore $\sigma = 1$ is a valid signature for $m = 1$
- In general, if $m = x^e < N$ then $m^d = x^{ed} = x$ therefore $\sigma = x$ is a valid signature for m

Easy to forge a valid signature of some **arbitrary message** (not chosen by the adversary):

- Pick $\sigma \in \mathbb{Z}_N^*$ and compute $m = \sigma^e \pmod N$

Security of Plain RSA Signatures

Are plain RSA signatures secure (under the RSA assumption)? **No!**

- The RSA assumption only implies hardness of computing a signature (the e-th root m^d) of a uniform message m
- It says nothing about specific messages m that are **chosen by the adversary**

Some messages are easy to sign:

- $1^d = 1 \pmod N$, therefore $\sigma = 1$ is a valid signature for $m = 1$
- In general, if $m = x^e < N$ then $m^d = x^{ed} = x$ therefore $\sigma = x$ is a valid signature for m

Easy to forge a valid signature of some **arbitrary message** (not chosen by the adversary):

- Pick $\sigma \in \mathbb{Z}_N^*$ and compute $m = \sigma^e \pmod N$
- σ is a valid signature for m. Indeed: $m^d = \sigma^{ed} = \sigma \pmod N$

Security of Plain RSA Signatures

Are plain RSA signatures secure (under the RSA assumption)? **No!**

- The RSA assumption only implies hardness of computing a signature (the e-th root m^d) of a uniform message m
- It says nothing about specific messages m that are **chosen by the adversary**

Some messages are easy to sign:

- $1^d = 1 \pmod N$, therefore $\sigma = 1$ is a valid signature for $m = 1$
- In general, if $m = x^e < N$ then $m^d = x^{ed} = x$ therefore $\sigma = x$ is a valid signature for m

Easy to forge a valid signature of some **arbitrary message** (not chosen by the adversary):

- Pick $\sigma \in \mathbb{Z}_N^*$ and compute $m = \sigma^e \pmod N$
- σ is a valid signature for m. Indeed: $m^d = \sigma^{ed} = \sigma \pmod N$

Each of the above approaches suffices to win the Sig-forge$_{\mathcal{A},\Pi}(n)$ experiment!

Attacks on Plain RSA Signatures: Combining Signatures

Signatures are easy to **combine**:

- Suppose that the adversary knows two valid signatures σ_1, σ_2 for two distinct messages m_1, m_2, respectively

Attacks on Plain RSA Signatures: Combining Signatures

Signatures are easy to **combine**:

- Suppose that the adversary knows two valid signatures σ_1, σ_2 for two distinct messages m_1, m_2, respectively
- The adversary can compute $\sigma = \sigma_1 \cdot \sigma_2 \pmod N$

Attacks on Plain RSA Signatures: Combining Signatures

Signatures are easy to **combine**:

- Suppose that the adversary knows two valid signatures σ_1, σ_2 for two distinct messages m_1, m_2, respectively
- The adversary can compute $\sigma = \sigma_1 \cdot \sigma_2 \pmod N$
- σ is a valid signature for $m = m_1 \cdot m_2 \pmod N$

$$(\sigma_1 \cdot \sigma_2)^e = \sigma_1^e \cdot \sigma_2^e = m_1^{ed} \cdot m_2^{ed} = m_1 \cdot m_2 = m \pmod N$$

Attacks on Plain RSA Signatures: Combining Signatures

Signatures are easy to **combine**:

- Suppose that the adversary knows two valid signatures σ_1, σ_2 for two distinct messages m_1, m_2, respectively
- The adversary can compute $\sigma = \sigma_1 \cdot \sigma_2 \pmod N$
- σ is a valid signature for $m = m_1 \cdot m_2 \pmod N$

$$(\sigma_1 \cdot \sigma_2)^e = \sigma_1^e \cdot \sigma_2^e = m_1^{ed} \cdot m_2^{ed} = m_1 \cdot m_2 = m \pmod N$$

How can an attacker use this?

- Pick a malicious message m and factor it into $m_1 \cdot m_2$ (recall that factoring is easy on average)
- Convince an honest party to sign the two "innocuous"-looking messages m_1 and m_2 separately
- Forge a signature σ for m
- Use (m, σ) to convince a third party (e.g., a judge) that the honest party signed m (e.g., a contract)

RSA full-domain hash (RSA-FDH)

Idea: instead of "signing" m, we sign some function $H(m) : \{0,1\}^* \to \mathbb{Z}_N^*$

RSA full-domain hash (RSA-FDH)

Idea: instead of "signing" m, we sign some function $H(m) : \{0,1\}^* \to \mathbb{Z}_N^*$

Gen(1^n):

- $(N, e, d) \leftarrow \mathsf{GenRSA}(1^n)$
- Return (pk, sk) where $pk = \langle N, e\rangle$ and $sk = \langle N, d\rangle$

RSA full-domain hash (RSA-FDH)

Idea: instead of "signing" m, we sign some function $H(m) : \{0,1\}^* \to \mathbb{Z}_N^*$

$\mathsf{Gen}(1^n)$:

- $(N, e, d) \leftarrow \mathsf{GenRSA}(1^n)$
- Return (pk, sk) where $pk = \langle N, e \rangle$ and $sk = \langle N, d \rangle$

$\mathsf{Sign}_{sk}(m)$:

- Here $sk = \langle N, d \rangle$ and $m \in Z_N^*$
- $\sigma \leftarrow H(m)^d \pmod N$
- Return σ

RSA full-domain hash (RSA-FDH)

Idea: instead of "signing" m, we sign some function $H(m) : \{0,1\}^* \to \mathbb{Z}_N^*$

Gen(1^n):

- $(N, e, d) \leftarrow$ GenRSA(1^n)
- Return (pk, sk) where $pk = \langle N, e \rangle$ and $sk = \langle N, d \rangle$

Sign$_{sk}(m)$:

- Here $sk = \langle N, d \rangle$ and $m \in Z_N^*$
- $\sigma \leftarrow H(m)^d \pmod N$
- Return σ

It is important that the range of the function is $\mathbb{Z}_N^*$ (or at least large enough).

We cannot simply use an off-the-shelf hash function (whose output length can be too short compared to the number of bits of N).

RSA full-domain hash (RSA-FDH)

Idea: instead of "signing" m, we sign some function $H(m) : \{0,1\}^* \to \mathbb{Z}_N^*$

$\mathsf{Gen}(1^n)$:

- $(N, e, d) \leftarrow \mathsf{GenRSA}(1^n)$
- Return (pk, sk) where $pk = \langle N, e \rangle$ and $sk = \langle N, d \rangle$

$\mathsf{Sign}_{sk}(m)$:

- Here $sk = \langle N, d \rangle$ and $m \in Z_N^*$
- $\sigma \leftarrow H(m)^d \pmod{N}$
- Return σ

$\mathsf{Vrfy}_{pk}(m, \sigma)$:

- Here $pk = \langle N, e \rangle$ and $\sigma \in Z_N^*$
- $\widetilde{m} \leftarrow \sigma^e \pmod{N}$
- Return 1 if $\widetilde{m} = H(m)$ and 0 otherwise

It is important that the range of the function is $\mathbb{Z}_N^*$ (or at least large enough).

We cannot simply use an off-the-shelf hash function (whose output length can be too short compared to the number of bits of N).

RSA full-domain hash (RSA-FDH)

Idea: instead of "signing" m, we sign some function $H(m) : \{0,1\}^* \to \mathbb{Z}_N^*$

Gen(1^n):

- $(N, e, d) \leftarrow$ GenRSA(1^n)
- Return (pk, sk) where $pk = \langle N, e \rangle$ and $sk = \langle N, d \rangle$

Sign$_{sk}(m)$:

- Here $sk = \langle N, d \rangle$ and $m \in Z_N^*$
- $\sigma \leftarrow H(m)^d \pmod N$
- Return σ

Vrfy$_{pk}(m, \sigma)$:

- Here $pk = \langle N, e \rangle$ and $\sigma \in Z_N^*$
- $\widetilde{m} \leftarrow \sigma^e \pmod N$
- Return 1 if $\widetilde{m} = H(m)$ and 0 otherwise

It is important that the range of the function is $\mathbb{Z}_N^*$ (or at least large enough).

We cannot simply use an off-the-shelf hash function (whose output length can be too short compared to the number of bits of N).

RSA-FDH natively handles long messages without using the Hash-and-Sign paradigm

RSA full-domain hash (RSA-FDH)

Idea: instead of "signing" m, we sign some function $H(m) : \{0,1\}^* \to \mathbb{Z}_N^*$

How do we choose H?

RSA full-domain hash (RSA-FDH)

Idea: instead of "signing" m, we sign some function $H(m) : \{0,1\}^* \to \mathbb{Z}_N^*$

How do we choose H?

- It should be infeasible for an attacker to start with σ and compute some $\widetilde{m} = \sigma^e \pmod{N}$ and then find a message m such that $H(m) = \widetilde{m}$

  H should be "hard to invert"

  This prevents an attacker from computing valid signatures of some arbitrary message

RSA full-domain hash (RSA-FDH)

Idea: instead of "signing" m, we sign some function $H(m) : \{0,1\}^* \to \mathbb{Z}_N^*$

How do we choose H?

- It should be infeasible for an attacker to start with σ and compute some $\widetilde{m} = \sigma^e \pmod N$ and then find a message m such that $H(m) = \widetilde{m}$

  H should be "hard to invert"

  This prevents an attacker from computing valid signatures of some arbitrary message

- It should be hard to find three messages m, m_1, m_2 such that $H(m) = H(m_1) \cdot H(m_2)$

  This prevents an attacker from combining signatures

RSA full-domain hash (RSA-FDH)

Idea: instead of "signing" m, we sign some function $H(m) : \{0,1\}^* \to \mathbb{Z}_N^*$

How do we choose H?

- It should be infeasible for an attacker to start with σ and compute some $\widetilde{m} = \sigma^e \pmod{N}$ and then find a message m such that $H(m) = \widetilde{m}$

  H should be "hard to invert"

  This prevents an attacker from computing valid signatures of some arbitrary message

- It should be hard to find three messages m, m_1, m_2 such that $H(m) = H(m_1) \cdot H(m_2)$

  This prevents an attacker from combining signatures

Security?

- There are known ways to choose H so that the resulting scheme can be proven secure

> If the RSA problem is hard relative to GenRSA and H is modeled as a random oracle, then RSA full-domain hash is secure.

RSA full-domain hash (RSA-FDH)

Idea: instead of "signing" m, we sign some function $H(m) : \{0,1\}^* \to \mathbb{Z}_N^*$

How do we choose H?

- It should be infeasible for an attacker to start with σ and compute some $\widetilde{m} = \sigma^e \pmod N$ and then find a message m such that $H(m) = \widetilde{m}$

  H should be "hard to invert"

  This prevents an attacker from computing valid signatures of some arbitrary message

- It should be hard to find three messages m, m_1, m_2 such that $H(m) = H(m_1) \cdot H(m_2)$

  This prevents an attacker from combining signatures

Security?

- There are known ways to choose H so that the resulting scheme can be proven secure

> If the RSA problem is hard relative to GenRSA and H is modeled as a random oracle, then RSA full-domain hash is secure.

The RSA PKCS #1 v2.1 standard includes a randomized variant of RSA-FDH

Signcryption

Signcryption

Sometimes it is desirable to achieve **both secrecy and integrity** in the public-key setting

Essentially, we want the public-key equivalent of authenticated encryption

Signcryption

Sometimes it is desirable to achieve **both secrecy and integrity** in the public-key setting

Essentially, we want the public-key equivalent of authenticated encryption

We assume that Alice is sending a message m to Bob

- Alice has a key-pair used for signing (vk, sk) (verification key, signing key)
- Bob has a key-pair used for encryption (ek, dk) (a public encryption key, and a secret decryption key)

Signcryption

Sometimes it is desirable to achieve **both secrecy and integrity** in the public-key setting

Essentially, we want the public-key equivalent of authenticated encryption

We assume that Alice is sending a message m to Bob

- Alice has a key-pair used for signing (vk, sk) (verification key, signing key)
- Bob has a key-pair used for encryption (ek, dk) (a public encryption key, and a secret decryption key)

Two natural approaches:

- **Naive encrypt-then-authenticate:** Alice computes $c \leftarrow \mathsf{Enc}_{ek}(m)$ and sends $(c, \mathsf{Sign}_{sk}(c))$

Signcryption

Sometimes it is desirable to achieve **both secrecy and integrity** in the public-key setting

Essentially, we want the public-key equivalent of authenticated encryption

We assume that Alice is sending a message m to Bob

- Alice has a key-pair used for signing (vk, sk) (verification key, signing key)
- Bob has a key-pair used for encryption (ek, dk) (a public encryption key, and a secret decryption key)

Two natural approaches:

- **Naive encrypt-then-authenticate:** Alice computes $c \leftarrow \mathsf{Enc}_{ek}(m)$ and sends $(c, \mathsf{Sign}_{sk}(c))$
- **Naive authenticate-then-encrypt:** Alice computes $\sigma \leftarrow \mathsf{Sign}_{sk}(m)$ and sends $c \leftarrow \mathsf{Enc}_{ek}(m \| \sigma)$

Signcryption

Sometimes it is desirable to achieve **both secrecy and integrity** in the public-key setting

Essentially, we want the public-key equivalent of authenticated encryption

We assume that Alice is sending a message m to Bob

- Alice has a key-pair used for signing (vk, sk) (verification key, signing key)
- Bob has a key-pair used for encryption (ek, dk) (a public encryption key, and a secret decryption key)

Two natural approaches:

- **Naive encrypt-then-authenticate:** Alice computes $c \leftarrow \mathsf{Enc}_{ek}(m)$ and sends $(c, \mathsf{Sign}_{sk}(c))$
- **Naive authenticate-then-encrypt:** Alice computes $\sigma \leftarrow \mathsf{Sign}_{sk}(m)$ and sends $c \leftarrow \mathsf{Enc}_{ek}(m \| \sigma)$

Both approaches have problems if used in this naive way

Problems with the natural approaches

Naive encrypt-then-authenticate: Alice computes $c \leftarrow \mathsf{Enc}_{ek}(m)$ and sends $(c, \mathsf{Sign}_{sk}(c))$

- An attacker that intercepts $(c, \mathsf{Sign}_{sk}(c))$ can strip Alice's signature and replace it with its own

Problems with the natural approaches

Naive encrypt-then-authenticate: Alice computes $c \leftarrow \mathsf{Enc}_{ek}(m)$ and sends $(c, \mathsf{Sign}_{sk}(c))$

- An attacker that intercepts $(c, \mathsf{Sign}_{sk}(c))$ can strip Alice's signature and replace it with its own
- The attacker sends $(c, \mathsf{Sign}_{sk'}(c))$ to Bob, where sk' is the attacker's secret signature key
- Bob does not detect the tampering, he thinks that the message came from the attacker and might reply leaking information about m

Problems with the natural approaches

Naive encrypt-then-authenticate: Alice computes $c \leftarrow \text{Enc}_{ek}(m)$ and sends $(c, \text{Sign}_{sk}(c))$

- An attacker that intercepts $(c, \text{Sign}_{sk}(c))$ can strip Alice's signature and replace it with its own
- The attacker sends $(c, \text{Sign}_{sk'}(c))$ to Bob, where sk' is the attacker's secret signature key
- Bob does not detect the tampering, he thinks that the message came from the attacker and might reply leaking information about m
- Moreover, the scheme no longer provides non-repudiation. To convince a third party that Alice signed m, Bob needs to reveal his decryption key dk

Problems with the natural approaches

Naive encrypt-then-authenticate: Alice computes $c \leftarrow \mathsf{Enc}_{ek}(m)$ and sends $(c, \mathsf{Sign}_{sk}(c))$

- An attacker that intercepts $(c, \mathsf{Sign}_{sk}(c))$ can strip Alice's signature and replace it with its own
- The attacker sends $(c, \mathsf{Sign}_{sk'}(c))$ to Bob, where sk' is the attacker's secret signature key
- Bob does not detect the tampering, he thinks that the message came from the attacker and might reply leaking information about m
- Moreover, the scheme no longer provides non-repudiation. To convince a third party that Alice signed m, Bob needs to reveal his decryption key dk

Naive authenticate-then-encrypt: Alice computes $\sigma \leftarrow \mathsf{Sign}_{sk}(m)$ and sends $c \leftarrow \mathsf{Enc}_{ek}(m \| \sigma)$

- Bob can decrypt c to obtain $m \| \sigma$, then re-encrypt it with the public encryption key ek' of another recipient
- Bob sends $\mathsf{Enc}_{ek'}(m \| \sigma)$ to the other recipient

Problems with the natural approaches

Naive encrypt-then-authenticate: Alice computes $c \leftarrow \mathsf{Enc}_{ek}(m)$ and sends $(c, \mathsf{Sign}_{sk}(c))$

- An attacker that intercepts $(c, \mathsf{Sign}_{sk}(c))$ can strip Alice's signature and replace it with its own
- The attacker sends $(c, \mathsf{Sign}_{sk'}(c))$ to Bob, where sk' is the attacker's secret signature key
- Bob does not detect the tampering, he thinks that the message came from the attacker and might reply leaking information about m
- Moreover, the scheme no longer provides non-repudiation. To convince a third party that Alice signed m, Bob needs to reveal his decryption key dk

Naive authenticate-then-encrypt: Alice computes $\sigma \leftarrow \mathsf{Sign}_{sk}(m)$ and sends $c \leftarrow \mathsf{Enc}_{ek}(m \| \sigma)$

- Bob can decrypt c to obtain $m \| \sigma$, then re-encrypt it with the public encryption key ek' of another recipient
- Bob sends $\mathsf{Enc}_{ek'}(m \| \sigma)$ to the other recipient
- The other recipient thinks that Alice sent the (signed!) message m to him.
  E.g., $m =$ "I owe you 100$".

Problems with the natural approaches

Naive encrypt-then-authenticate: Alice computes $c \leftarrow \mathsf{Enc}_{ek}(m)$ and sends $(c, \mathsf{Sign}_{sk}(c))$

- An attacker that intercepts $(c, \mathsf{Sign}_{sk}(c))$ can strip Alice's signature and replace it with its own
- The attacker sends $(c, \mathsf{Sign}_{sk'}(c))$ to Bob, where sk' is the attacker's secret signature key
- Bob does not detect the tampering, he thinks that the message came from the attacker and might reply leaking information about m
- Moreover, the scheme no longer provides non-repudiation. To convince a third party that Alice signed m, Bob needs to reveal his decryption key dk

Naive authenticate-then-encrypt: Alice computes $\sigma \leftarrow \mathsf{Sign}_{sk}(m)$ and sends $c \leftarrow \mathsf{Enc}_{ek}(m \| \sigma)$

- Bob can decrypt c to obtain $m \| \sigma$, then re-encrypt it with the public encryption key ek' of another recipient
- Bob sends $\mathsf{Enc}_{ek'}(m \| \sigma)$ to the other recipient
- The other recipient thinks that Alice sent the (signed!) message m to him.
  E.g., $m =$ "I owe you 100\$".
- This scheme provides non-repudiation: Bob can reveal m and σ

Fixing the natural approaches

To prevent these attacks:

- The sender includes its identity along with the plaintext when encrypting
- The sender includes the recipient's identity along with the message when signing

Fixing the natural approaches

To prevent these attacks:

- The sender includes its identity along with the plaintext when encrypting
- The sender includes the recipient's identity along with the message when signing
- The recipient checks that the purported sender matches the decrypted sender's identity
- The recipient checks that the signature incorporates its own identity

Fixing the natural approaches

To prevent these attacks:

- The sender includes its identity along with the plaintext when encrypting
- The sender includes the recipient's identity along with the message when signing
- The recipient checks that the purported sender matches the decrypted sender's identity
- The recipient checks that the signature incorporates its own identity

Encrypt-then-authenticate:

- Alice computes $c \leftarrow \mathsf{Enc}_{ek}(\text{“Alice”} \,\|\, m)$
- She sends: $(c, \mathsf{Sign}_{sk}(c \,\|\, \text{“Bob”}))$

Fixing the natural approaches

To prevent these attacks:

- The sender includes its identity along with the plaintext when encrypting
- The sender includes the recipient's identity along with the message when signing
- The recipient checks that the purported sender matches the decrypted sender's identity
- The recipient checks that the signature incorporates its own identity

Encrypt-then-authenticate:

- Alice computes $c \leftarrow \mathsf{Enc}_{ek}(\text{“Alice”} \,\|\, m)$
- She sends: $(c, \mathsf{Sign}_{sk}(c \,\|\, \text{“Bob”}))$

Authenticate-then-encrypt:

- Alice computes $\sigma \leftarrow \mathsf{Sign}_{sk}(m \,\|\, \text{“Bob”})$
- Alice sends $\mathsf{Enc}_{ek}(\text{“Alice”} \,\|\, m \| \sigma)$

Fixing the natural approaches

To prevent these attacks:

- The sender includes its identity along with the plaintext when encrypting
- The sender includes the recipient's identity along with the message when signing
- The recipient checks that the purported sender matches the decrypted sender's identity
- The recipient checks that the signature incorporates its own identity

Encrypt-then-authenticate:

- Alice computes $c \leftarrow \mathsf{Enc}_{ek}(\text{“Alice”} \,\|\, m)$
- She sends: $(c, \mathsf{Sign}_{sk}(c \,\|\, \text{“Bob”}))$

Authenticate-then-encrypt:

- Alice computes $\sigma \leftarrow \mathsf{Sign}_{sk}(m \,\|\, \text{“Bob”})$
- Alice sends $\mathsf{Enc}_{ek}(\text{“Alice”} \,\|\, m \| \sigma)$

Both authenticate-then-encrypt and encrypt-then-authenticate are secure if a CCA-secure encryption scheme and a strongly secure signature scheme are used.

Fixing the natural approaches

To prevent these attacks:

- The sender includes its identity along with the plaintext when encrypting
- The sender includes the recipient's identity along with the message when signing
- The recipient checks that the purported sender matches the decrypted sender's identity
- The recipient checks that the signature incorporates its own identity

Encrypt-then-authenticate:

- Alice computes $c \leftarrow \mathsf{Enc}_{ek}(\text{“Alice”} \,\|\, m)$
- She sends: $(c, \mathsf{Sign}_{sk}(c \,\|\, \text{“Bob”}))$

Authenticate-then-encrypt: ← Also provides non-repudiation

- Alice computes $\sigma \leftarrow \mathsf{Sign}_{sk}(m \,\|\, \text{“Bob”})$
- Alice sends $\mathsf{Enc}_{ek}(\text{“Alice”} \,\|\, m \| \sigma)$

Both authenticate-then-encrypt and encrypt-then-authenticate are secure if a CCA-secure encryption scheme and a strongly secure signature scheme are used.

Digital Certificates

Digital Certificates & Public-Key Infrastructure

Alice wants to communicate with Bob but has never met him before, and they do not share any secret information

Alice = web browser Bob = bank-website.com

How can Alice be sure to be talking to the “right” Bob and not to an impersonator?

Digital Certificates & Public-Key Infrastructure

Alice wants to communicate with Bob but has never met him before, and they do not share any secret information

Alice = web browser   Bob = bank-website.com

How can Alice be sure to be talking to the "right" Bob and not to an impersonator?

- This problem can't be solved without introducing a trusted third-party
- Trusted third parties are called **certification authorities (CAs)**

Digital Certificates & Public-Key Infrastructure

Alice wants to communicate with Bob but has never met him before, and they do not share any secret information

Alice = web browser   Bob = bank-website.com

How can Alice be sure to be talking to the "right" Bob and not to an impersonator?

- This problem can't be solved without introducing a trusted third-party
- Trusted third parties are called **certification authorities (CAs)**

Let's call the trusted third party Charlie:

- Charlie has generated a key-pair (pk_C, sk_C) for a secure digital signature scheme
- Bob has generated a key-pair (pk_B, sk_B) for either a public-key encryption scheme or a digital signature scheme

Digital Certificates & Public-Key Infrastructure

Alice wants to communicate with Bob but has never met him before, and they do not share any secret information

Alice = web browser Bob = bank-website.com

How can Alice be sure to be talking to the "right" Bob and not to an impersonator?

- This problem can't be solved without introducing a trusted third-party
- Trusted third parties are called **certification authorities (CAs)**

Let's call the trusted third party Charlie:

- Charlie has generated a key-pair (pk_C, sk_C) for a secure digital signature scheme
- Bob has generated a key-pair (pk_B, sk_B) for either a public-key encryption scheme or a digital signature scheme
- Charlie can sign (the digital equivalent of) the following message: "the public key of Bob is pk_B"...

$$\text{cert}_{C\to B} \leftarrow \text{Sign}_{sk_C}(\text{“the public key of Bob is } pk_B\text{”})$$

Digital Certificates & Public-Key Infrastructure

Alice wants to communicate with Bob but has never met him before, and they do not share any secret information

Alice = web browser Bob = bank-website.com

How can Alice be sure to be talking to the "right" Bob and not to an impersonator?

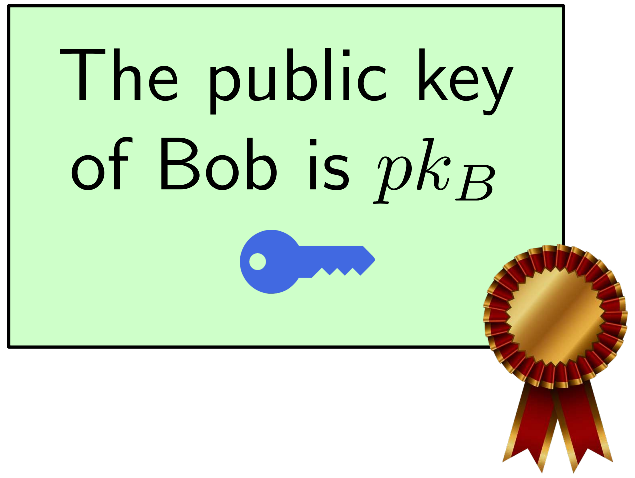
- This problem can't be solved without introducing a trusted third-party
- Trusted third parties are called **certification authorities (CAs)**

Let's call the trusted third party Charlie:

- Charlie has generated a key-pair (pk_C, sk_C) for a secure digital signature scheme
- Bob has generated a key-pair (pk_B, sk_B) for either a public-key encryption scheme or a digital signature scheme
- Charlie can sign (the digital equivalent of) the following message: "the public key of Bob is pk_B"...

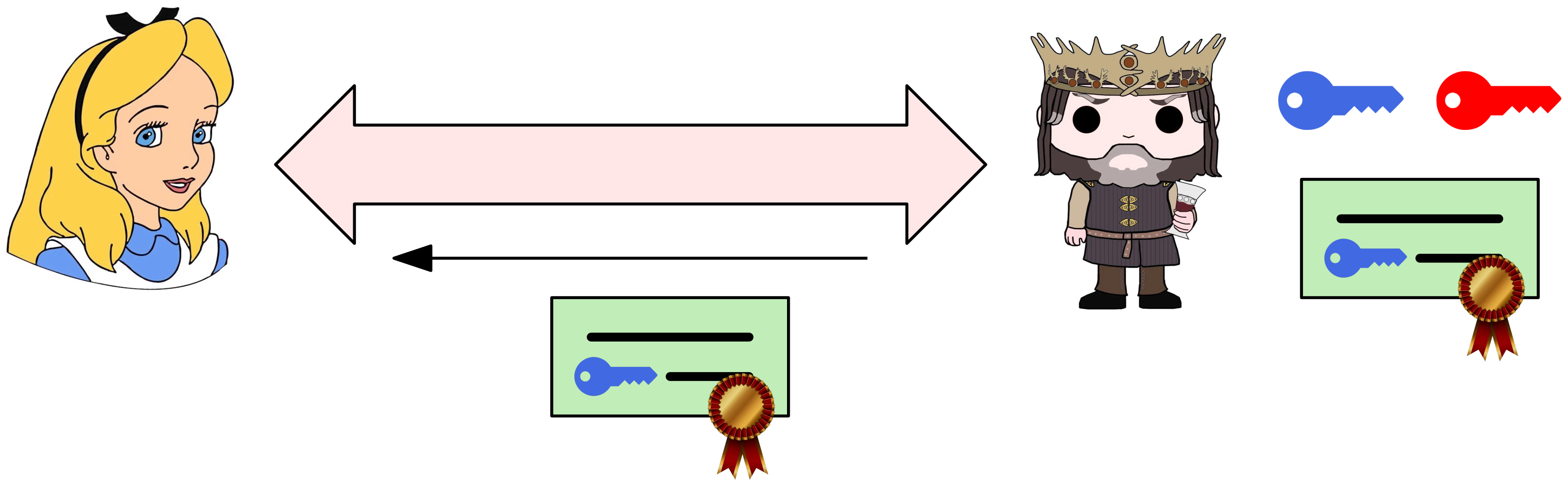
$$\text{cert}_{C \to B} \leftarrow \text{Sign}_{sk_C}(\text{“the public key of Bob is } pk_B\text{”})$$

...and send the resulting signature, called a **digital certificate**, to Bob



Public-Key Infrastructure & Digital Certificates

A communication session between Alice and Bob starts as follows:



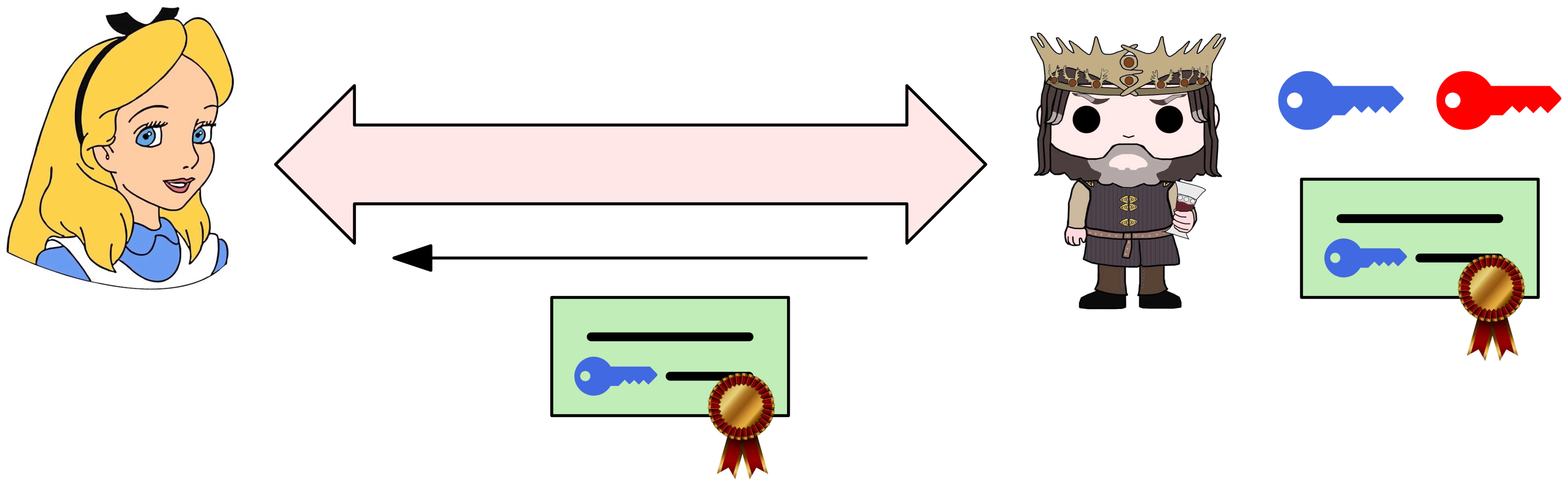
- Alice contacts Bob
- Bob replies with the message $(pk_B, \mathsf{cert}_{C\to B})$
- Alice checks that cert is valid

$$\mathsf{Vrfy}_{pk_C}(\text{“the public key of bob is } pk_B\text{”})$$

- If the verification succeeds, Alice accepts Bob's certificate and the session continues. Alice can now use pk_B for encryption (of messages sent to Bob) / authentication (of Bob's messages)

Public-Key Infrastructure & Digital Certificates

A communication session between Alice and Bob starts as follows:



- Alice contacts Bob
- Bob replies with the message $(pk_B, \mathsf{cert}_{C\to B})$
- Alice checks that cert is valid

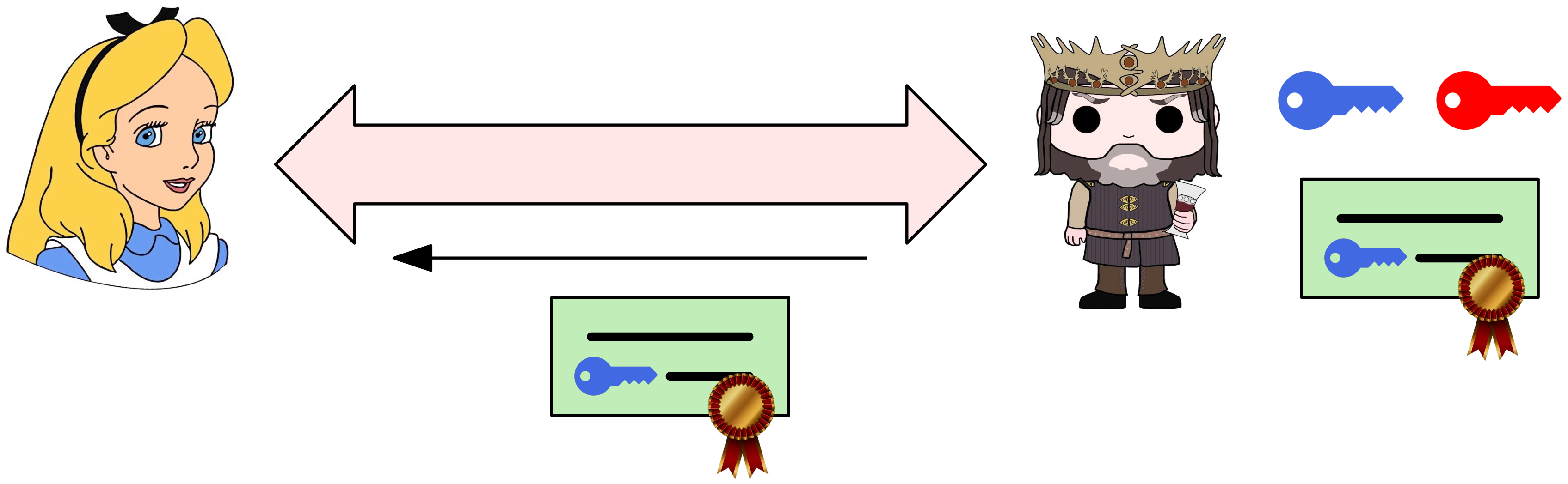
$$\mathsf{Vrfy}_{pk_C}(\text{“the public key of bob is } pk_B\text{”})$$

- If the verification succeeds, Alice accepts Bob's certificate and the session continues. Alice can now use pk_B for encryption (of messages sent to Bob) / authentication (of Bob's messages)

How does Alice learn pk_C?

Public-Key Infrastructure & Digital Certificates

A communication session between Alice and Bob starts as follows:



- Alice contacts Bob
- Bob replies with the message $(pk_B, \mathsf{cert}_{C \to B})$
- Alice checks that cert is valid

$$\mathsf{Vrfy}_{pk_C}(\text{“the public key of bob is } pk_B\text{”})$$

- If the verification succeeds, Alice accepts Bob's certificate and the session continues. Alice can now use pk_B for encryption (of messages sent to Bob) / authentication (of Bob's messages)

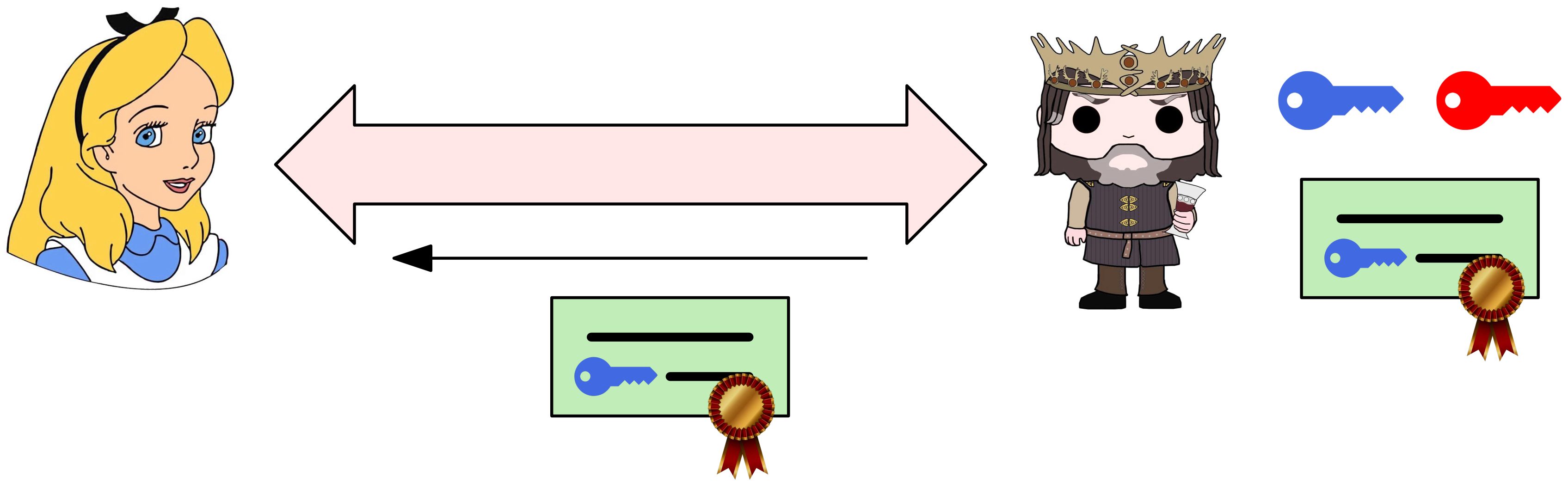
How does Alice learn pk_C?

- Alice needs to obtain Charlie's public key "out of band"

E.g., in person, where Alice is given pk_C by Charlie

Public-Key Infrastructure & Digital Certificates

A communication session between Alice and Bob starts as follows:



- Alice contacts Bob
- Bob replies with the message $(pk_B, \mathsf{cert}_{C\to B})$
- Alice checks that cert is valid

$$\mathsf{Vrfy}_{pk_C}(\text{"the public key of bob is } pk_B\text{"})$$

- If the verification succeeds, Alice accepts Bob's certificate and the session continues. Alice can now use pk_B for encryption (of messages sent to Bob) / authentication (of Bob's messages)

How does Alice learn pk_C?

- Alice needs to obtain Charlie's public key "out of band"

  E.g., in person, where Alice is given pk_C by Charlie

- In practice pk_C is bundled together with some piece of software

  E.g., Web browsers have a default trusted list of CAs and are shipped with their public keys

Certificate Chains

A CA (Charlie) can also **delegate** its ability to issue certificates to an **intermediate** CA

Charlie is the **Root CA** and is trusted by Alice

Certificate Chains

A CA (Charlie) can also **delegate** its ability to issue certificates to an **intermediate** CA

Charlie is the **Root CA** and is trusted by Alice

Say Daisy also has a key pair (pk_D, sk_D) and Charlie wants to allow Daisy to issue certificates on his behalf

- Charlie can sign (the digital equivalent of) the message "The public key of Daisy is pk_D and she is allowed to issue other certificates"

The public key of Daisy is pk_D and she is allowed to issue other certificates

$$\text{cert}_{C \to D} \leftarrow \text{Sign}_{sk_C}(\text{"the public key of Daisy is } pk_D \text{ and she is allowed to issue other certificates"})$$

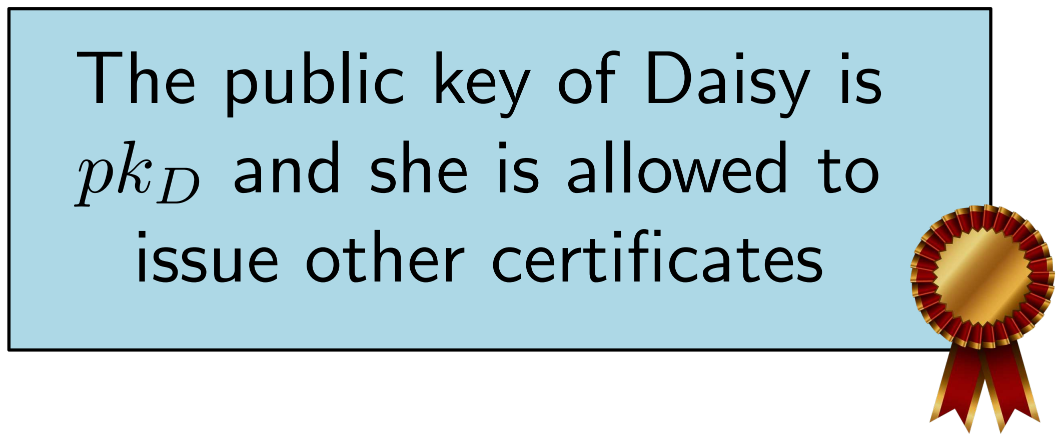
Certificate Chains

A CA (Charlie) can also **delegate** its ability to issue certificates to an **intermediate** CA

Charlie is the **Root CA** and is trusted by Alice

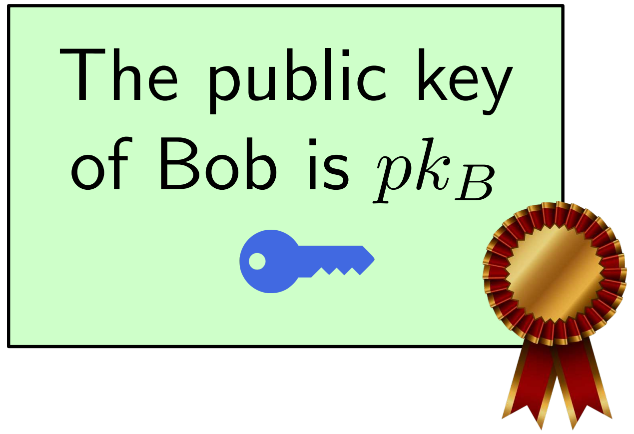
Say Daisy also has a key pair (pk_D, sk_D) and Charlie wants to allow Daisy to issue certificates on his behalf

- Charlie can sign (the digital equivalent of) the message "The public key of Daisy is pk_D and she is allowed to issue other certificates"



$$\text{cert}_{C \to D} \leftarrow \text{Sign}_{sk_C}(\text{"the public key of Daisy is } pk_D \text{ and she is allowed to issue other certificates"})$$

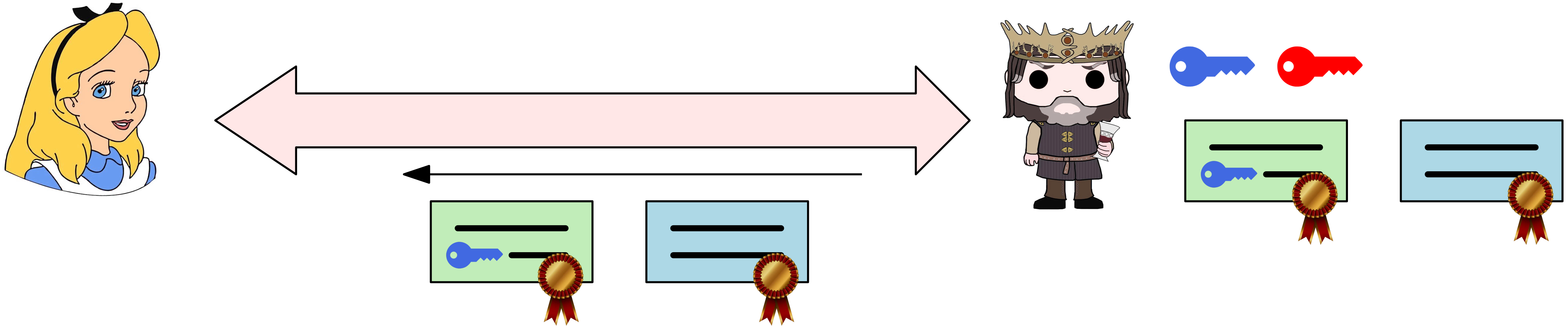
Daisy can then issue a certificate to Bob



$$\text{cert}_{D \to B} \leftarrow \text{Sign}_{sk_C}(\text{"the public key of Bob is } pk_B\text{"})$$

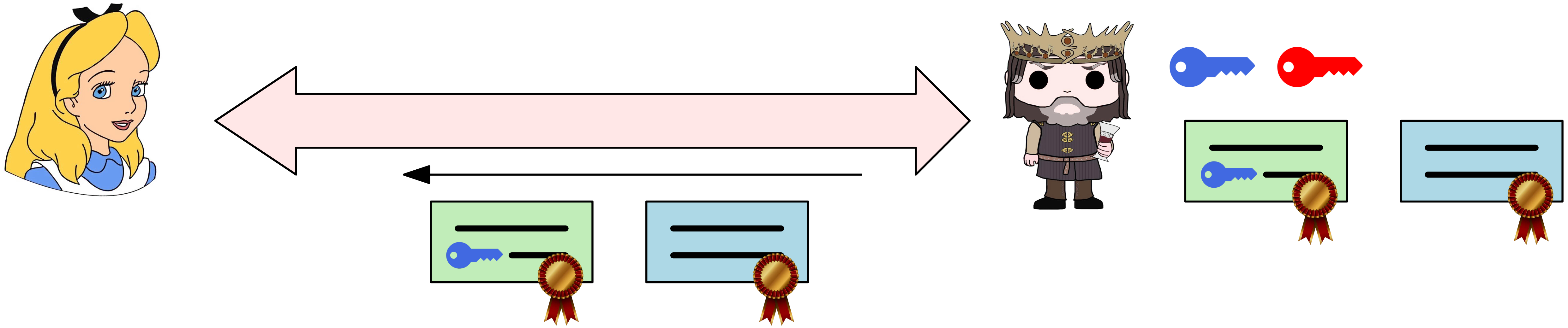
Certificate Chains

When Alice contacts Bob, Bob now needs to provide both his public key and certificate $(pk_B, \text{cert}_{D \to B})$ and the intermediate CA (Daisys's) public key and certificate $(pk_D, \text{cert}_{C \to D})$



Certificate Chains

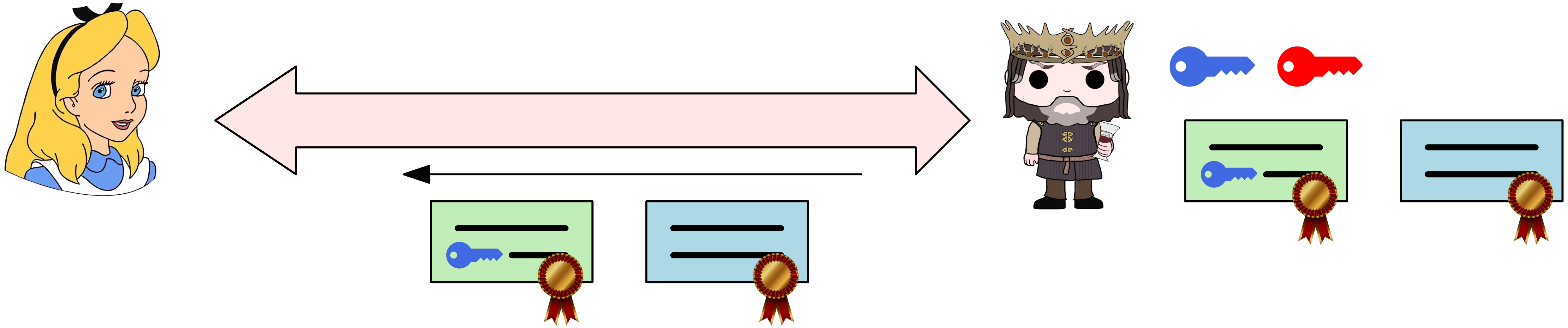
When Alice contacts Bob, Bob now needs to provide both his public key and certificate $(pk_B, \text{cert}_{D \to B})$ and the intermediate CA (Daisys's) public key and certificate $(pk_D, \text{cert}_{C \to D})$



- Alice uses pk_C to check that $\text{cert}_{C \to D}$ is a valid signature from Charlie and learns that:
  - Daisy is authorized to issue certificates on behalf of Charlie
  - The public key of Daisy is pk_D

Certificate Chains

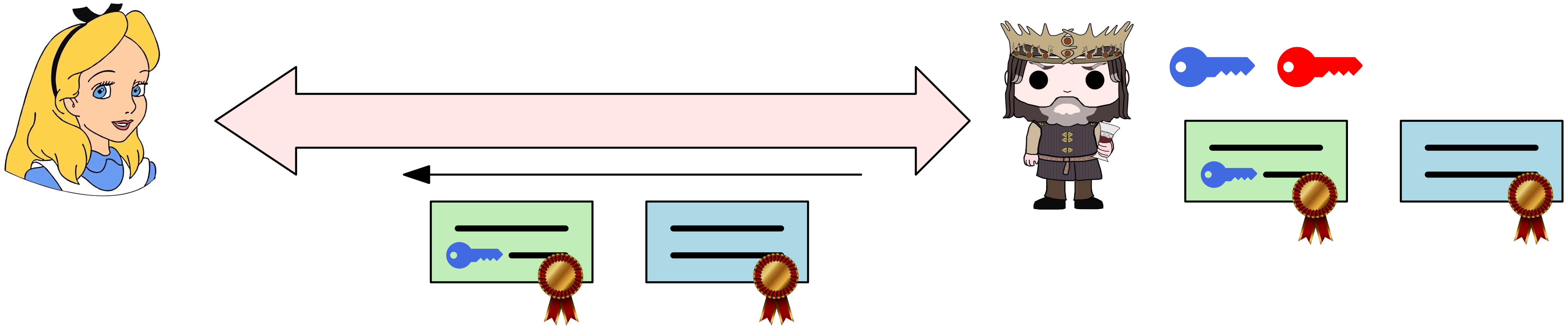
When Alice contacts Bob, Bob now needs to provide both his public key and certificate $(pk_B, \text{cert}_{D \to B})$ and the intermediate CA (Daisys's) public key and certificate $(pk_D, \text{cert}_{C \to D})$



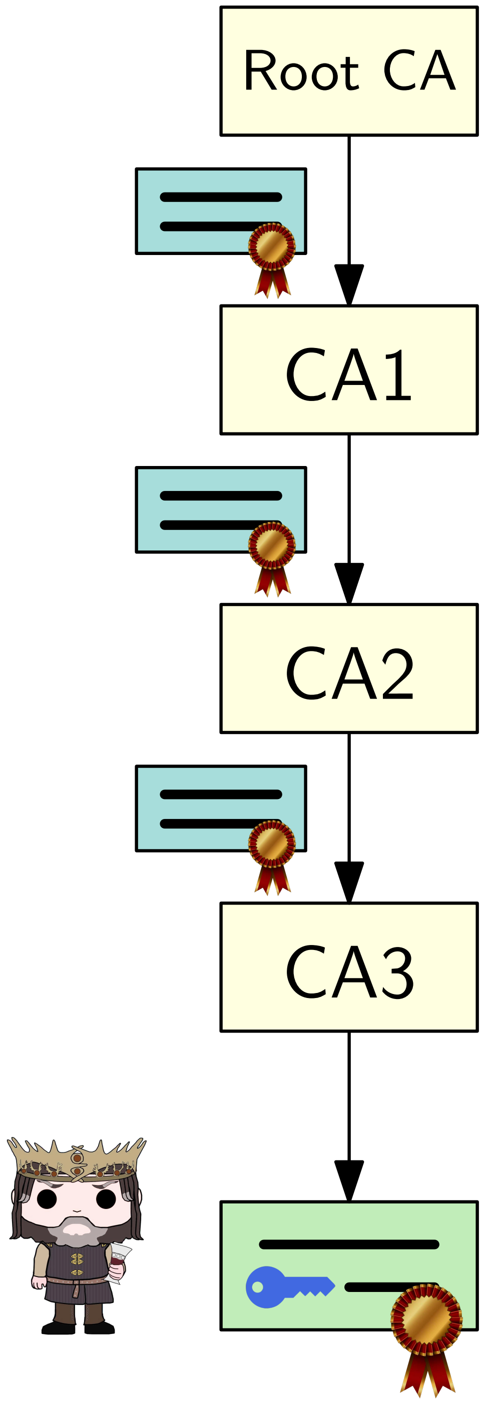
- Alice uses pk_C to check that $\text{cert}_{C \to D}$ is a valid signature from Charlie and learns that:
  — Daisy is authorized to issue certificates on behalf of Charlie
  — The public key of Daisy is pk_D
- Alice uses pk_D to check that $\text{cert}_{D \to B}$ is a valid signature from Daisy
  — Alice learns that pk_B is the public key of Bob

Certificate Chains

When Alice contacts Bob, Bob now needs to provide both his public key and certificate $(pk_B, \mathsf{cert}_{D \to B})$ and the intermediate CA (Daisys's) public key and certificate $(pk_D, \mathsf{cert}_{C \to D})$

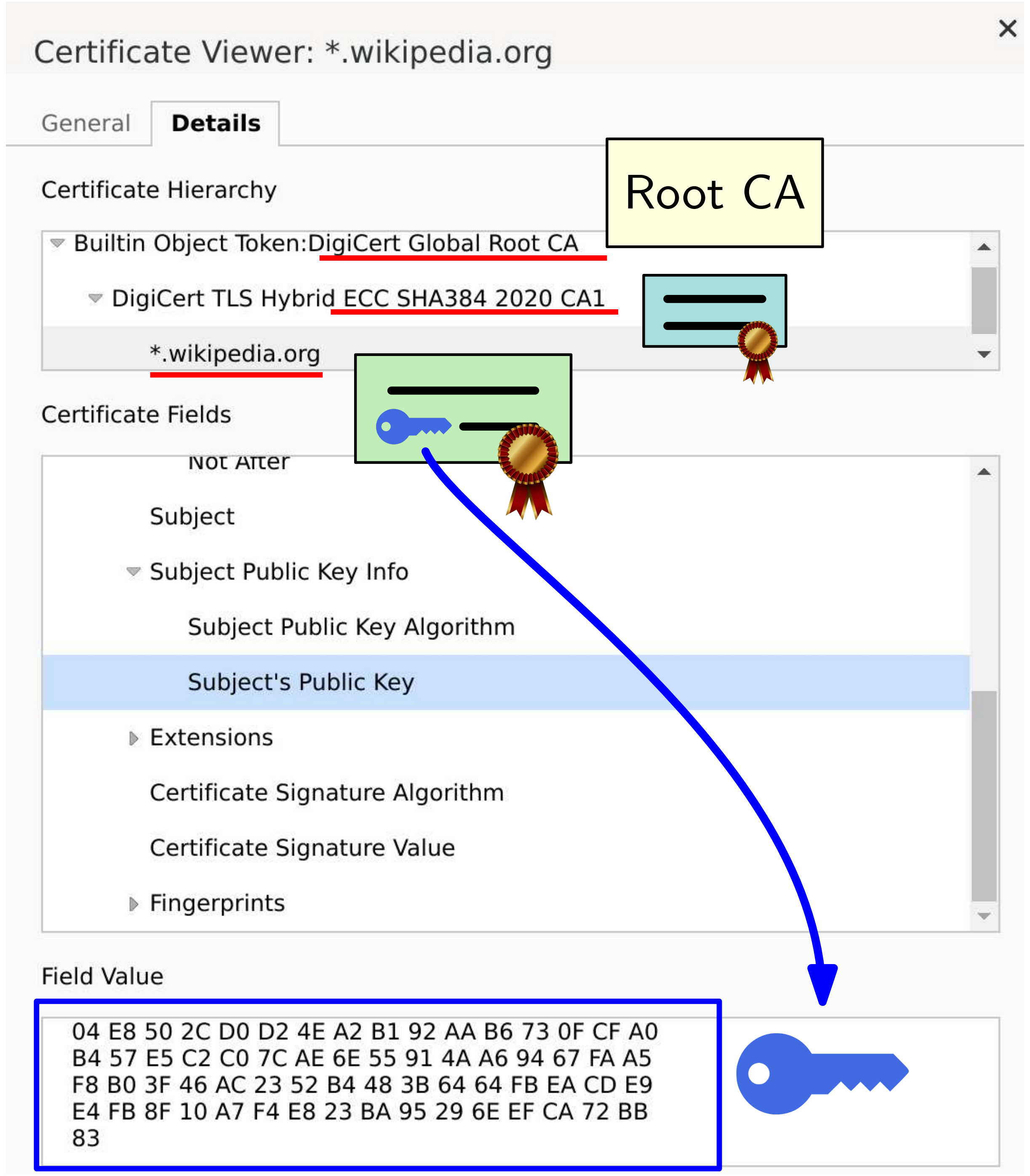


- Alice uses pk_C to check that $\mathsf{cert}_{C \to D}$ is a valid signature from Charlie and learns that:
  - Daisy is authorized to issue certificates on behalf of Charlie
  - The public key of Daisy is pk_D
- Alice uses pk_D to check that $\mathsf{cert}_{D \to B}$ is a valid signature from Daisy
  - Alice learns that pk_B is the public key of Bob



The same idea generalizes to any number of intermediate CAs

Certificate Chains



Invalidating Certificates

Certificates should generally not be valid indefinitely.

E.g., if (when) a private key gets stolen, the corresponding certificate should be considered invalid

Invalidating Certificates

Certificates should generally not be valid indefinitely.

E.g., if (when) a private key gets stolen, the corresponding certificate should be considered invalid

- **Expiration**

  — The CA signs the message (“The public key of Bob is pk_b, date), where date is a point of time in the future after which the certificate will no longer be valid

  — When Alice checks Bob's certificate, she also checks its expiration date against the current date

  — Disadvantage: if the private key is stolen, the certificate will remain valid until the expiration date

Invalidating Certificates

Certificates should generally not be valid indefinitely.

E.g., if (when) a private key gets stolen, the corresponding certificate should be considered invalid

- **Expiration**

  — The CA signs the message ("The public key of Bob is pk_b, date), where date is a point of time in the future after which the certificate will no longer be valid

  — When Alice checks Bob's certificate, she also checks its expiration date against the current date

  — Disadvantage: if the private key is stolen, the certificate will remain valid until the expiration date

- **Revocation**

  — The CA signs the message ("The public key of Bob is pk_b, serial_no), where serial_no is a unique serial number

  — The CA periodically (e.g., daily) creates a **revocation list** of all the serial numbers of the certificates it issued but need to be invalided, signs it, and publishes the signed list

  — When Alice checks Bob's certificate, she also checks that the serial number is not in the revocation list

  — Disadvantage: Alice needs to keep an updated copy of the revocation list

Transport Layer Security (TLS)

Used to establish secure communication session over the internet

Standardized in 1999 and updated multiple times. The current version is TLS 1.3

The following is just a high-level description!

Transport Layer Security (TLS)

Used to establish secure communication session over the internet

Standardized in 1999 and updated multiple times. The current version is TLS 1.3

The following is just a high-level description!

The server owns a certificate (or a certificate chain) cert issued by some CA and the corresponding key-pair (pk_S, sk_S) for a digital signature scheme

Transport Layer Security (TLS)

Used to establish secure communication session over the internet

Standardized in 1999 and updated multiple times. The current version is TLS 1.3

The following is just a high-level description!

The server owns a certificate (or a certificate chain) cert issued by some CA and the corresponding key-pair (pk_S, sk_S) for a digital signature scheme

Two phases:

- **Handshake protocol:** performs authenticated key exchange to establish two shared *symmetric* private keys k_S and k_C

Transport Layer Security (TLS)

Used to establish secure communication session over the internet

Standardized in 1999 and updated multiple times. The current version is TLS 1.3

The following is just a high-level description!

The server owns a certificate (or a certificate chain) cert issued by some CA and the corresponding key-pair (pk_S, sk_S) for a digital signature scheme

Two phases:

- **Handshake protocol:** performs authenticated key exchange to establish two shared *symmetric* private keys k_S and k_C
- **Record-layer protocol:** uses the shared keys to encrypt and authenticate the communication

— Messages from client to server and vice-versa are encrypted using an **authenticated encryption scheme**

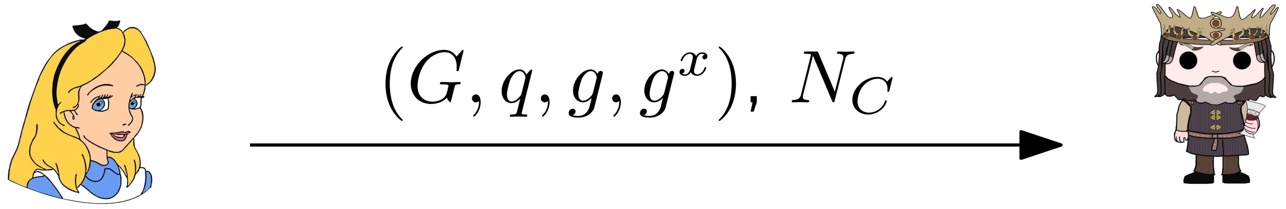
— The client encrypts with key k_C (and decrypts with key k_S)

— The server encrypts with key k_S (and decrypts with key k_C)

— Sequence numbers are used to prevent replay attacks

TLS: Handshake protocol

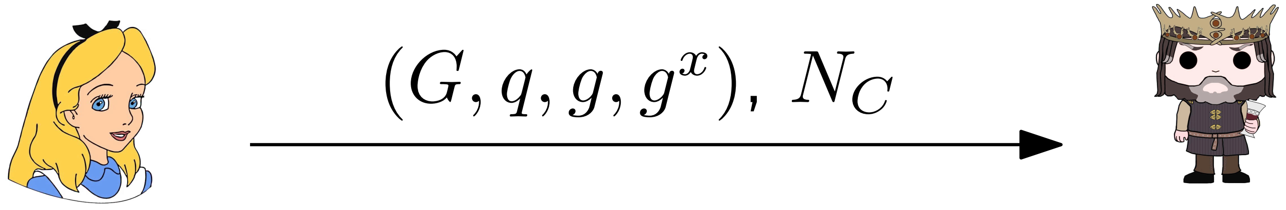
- The client initiates the handshake by sending the initial message (G, q, g, g^x) of the Diffie-Hellman key-exchange protocol and a nonce N_C chosen u.a.r. from $\{0,1\}^n$



TLS: Handshake protocol

- The client initiates the handshake by sending the initial message (G, q, g, g^x) of the Diffie-Hellman key-exchange protocol and a nonce N_C chosen u.a.r. from $\{0,1\}^n$

  The underlying group is selected by the client from a set of standardized options, and can be either a prime-order subgroup of Z_p^* for some prime p or an elliptic-curve group

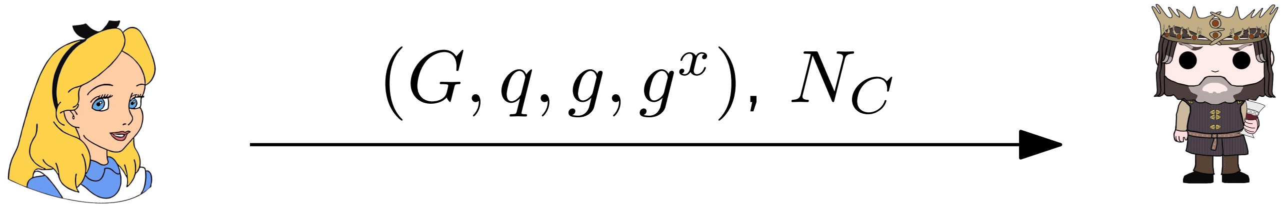


TLS: Handshake protocol

- The client initiates the handshake by sending the initial message (G, q, g, g^x) of the Diffie-Hellman key-exchange protocol and a nonce N_C chosen u.a.r. from $\{0,1\}^n$

  The underlying group is selected by the client from a set of standardized options, and can be either a prime-order subgroup of Z_p^* for some prime p or an elliptic-curve group

  The client also sends to the server a list of all supported cryptographic algorithms

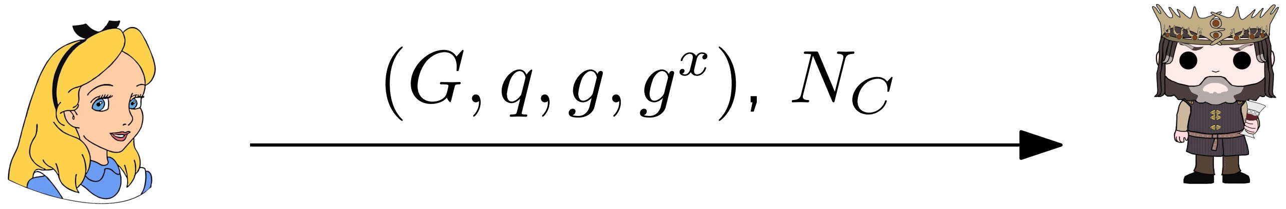


TLS: Handshake protocol

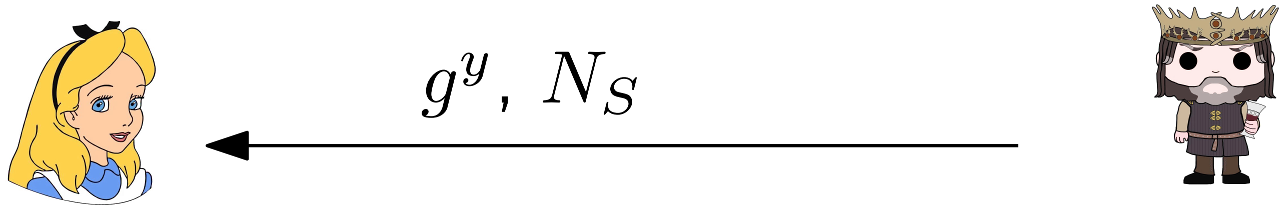
- The client initiates the handshake by sending the initial message (G, q, g, g^x) of the Diffie-Hellman key-exchange protocol and a nonce N_C chosen u.a.r. from $\{0,1\}^n$

  The underlying group is selected by the client from a set of standardized options, and can be either a prime-order subgroup of Z_p^* for some prime p or an elliptic-curve group

  The client also sends to the server a list of all supported cryptographic algorithms



- The server completes the Diffie-Hellman key-exchange by sending g^y to the client. The server also sends a nonce N_S chosen u.a.r. from $\{0,1\}^n$

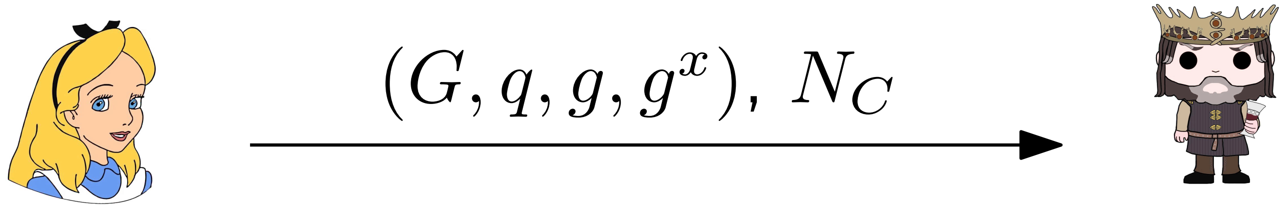


TLS: Handshake protocol

- The client initiates the handshake by sending the initial message (G, q, g, g^x) of the Diffie-Hellman key-exchange protocol and a nonce N_C chosen u.a.r. from $\{0,1\}^n$

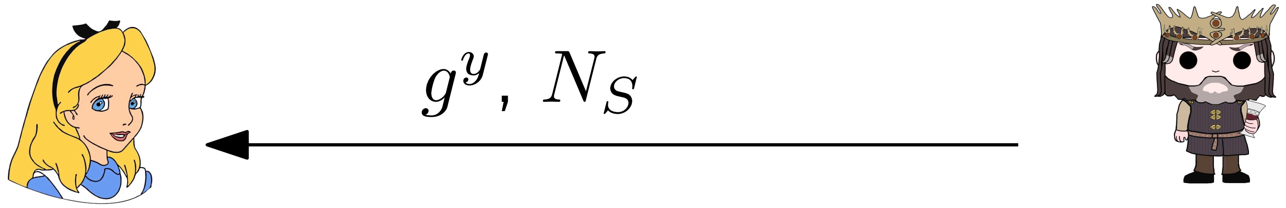
  The underlying group is selected by the client from a set of standardized options, and can be either a prime-order subgroup of Z_p^* for some prime p or an elliptic-curve group

  The client also sends to the server a list of all supported cryptographic algorithms



- The server completes the Diffie-Hellman key-exchange by sending g^y to the client. The server also sends a nonce N_S chosen u.a.r. from $\{0,1\}^n$

  The client and the server now share a secret group element $K = g^{xy}$. They apply a key-derivation function to K to obtain *four* keys k'_S, k'_C, k_S, and k_C for an authenticated encryption scheme.

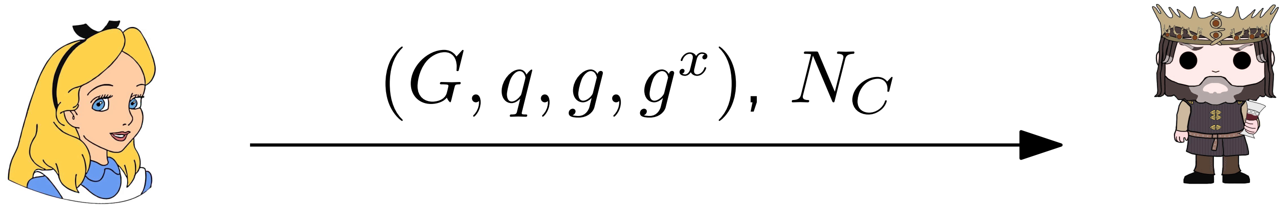


TLS: Handshake protocol

- The client initiates the handshake by sending the initial message (G, q, g, g^x) of the Diffie-Hellman key-exchange protocol and a nonce N_C chosen u.a.r. from $\{0,1\}^n$

  The underlying group is selected by the client from a set of standardized options, and can be either a prime-order subgroup of Z_p^* for some prime p or an elliptic-curve group

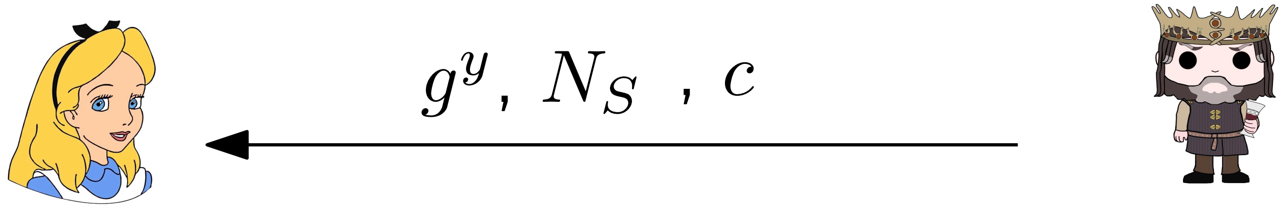
  The client also sends to the server a list of all supported cryptographic algorithms



- The server completes the Diffie-Hellman key-exchange by sending g^y to the client. The server also sends a nonce N_S chosen u.a.r. from $\{0,1\}^n$

  The client and the server now share a secret group element $K = g^{xy}$. They apply a key-derivation function to K to obtain *four* keys k'_S, k'_C, k_S, and k_C for an authenticated encryption scheme.

  The server uses pk_S to compute a digital signature σ of all the handshake messages exchanged so far. The server encrypts pk_S, cert, and σ with the symmetric key k'_S and sends the resulting ciphertext c to the client.



TLS: Handshake protocol (cont.)

- The client decrypts the ciphertext to recover pk_S, cert, and σ. Then, it checks whether some trusted CA issued cert and that cert is a valid certificate for pk_S (and is not expired or revoked).

TLS: Handshake protocol (cont.)

- The client decrypts the ciphertext to recover pk_S, cert, and σ. Then, it checks whether some trusted CA issued cert and that cert is a valid certificate for pk_S (and is not expired or revoked).

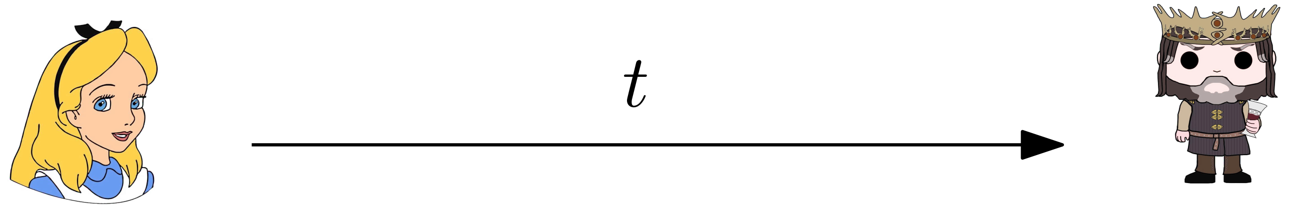
  The client uses pk_S to verify the signature σ on the handshake messages. If any of these checks fails, the client aborts the protocol

TLS: Handshake protocol (cont.)

- The client decrypts the ciphertext to recover pk_S, cert, and σ. Then, it checks whether some trusted CA issued cert and that cert is a valid certificate for pk_S (and is not expired or revoked).

  The client uses pk_S to verify the signature σ on the handshake messages. If any of these checks fails, the client aborts the protocol

  Finally, the client uses a MAC with key k'_C to compute a tag t on the handshake messages exchanged so far, and sends t to the server



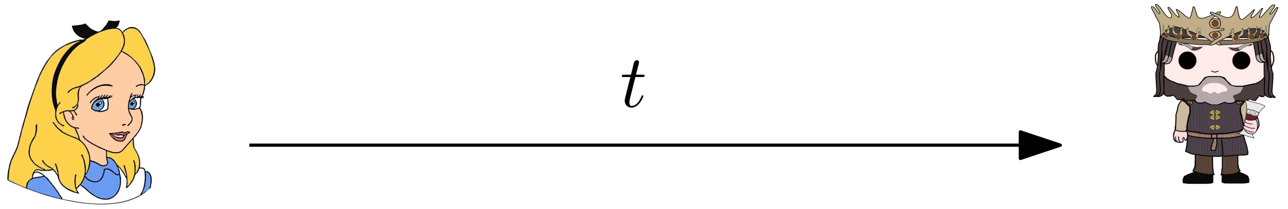
TLS: Handshake protocol (cont.)

- The client decrypts the ciphertext to recover pk_S, cert, and σ. Then, it checks whether some trusted CA issued cert and that cert is a valid certificate for pk_S (and is not expired or revoked).

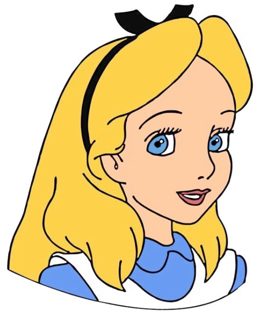
  The client uses pk_S to verify the signature σ on the handshake messages. If any of these checks fails, the client aborts the protocol

  Finally, the client uses a MAC with key k'_C to compute a tag t on the handshake messages exchanged so far, and sends t to the server

  The server checks t. If verification fails, the server aborts the protocol. Otherwise the actual communication starts using the record-layer protocol with keys k_S and k_C
  (the keys k'_S and k'_C are destroyed: they are needed during the handshake phase)

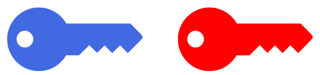


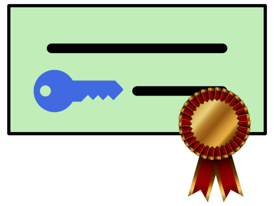
TLS: Handshake protocol: Security (informal)



$(G, q, g, g^x),\ N_C$ →

pk_S sk_S





cert

Derive k'_S, k'_C, k_S, k_C from $K = g^{xy}$

$\sigma \leftarrow \mathsf{Sign}_{sk_S}((G, q, g, g^x, N_C, g^y, N_S))$

$c \leftarrow \mathsf{Enc}_{k'_S}(pk, \mathsf{cert}, \sigma)$

← $g^y,\ N_S,\ c$

Derive k'_S, k'_C, k_S, k_C from $K = g^{xy}$

$(pk, \mathsf{cert}, \sigma) \leftarrow \mathsf{Dec}_{k'_S}(c)$

Validate cert

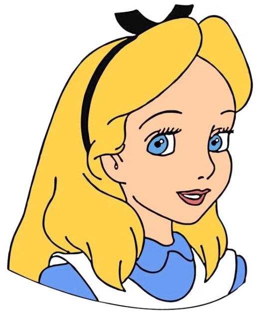
$\mathsf{Vrfy}_{pk_S}((G, q, g, g^x, N_C, g^y, N_S), \sigma)$

$t \leftarrow \mathsf{Mac}_{k'_C}(G, q, g, g^x, N_C, g^y, N_S, c)$

t →

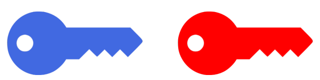
$\mathsf{Vrfy}_{k'_C}((G, q, g, g^x, N_C, g^y, N_S, c), t)$

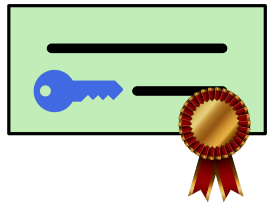
TLS: Handshake protocol: Security (informal)



$(G, q, g, g^x),\ N_C$ $\longrightarrow$

pk_S sk_S





cert

Derive k'_S, k'_C, k_S, k_C from $K = g^{xy}$

$\sigma \leftarrow \mathsf{Sign}_{sk_S}((G, q, g, g^x, N_C, g^y, N_S))$

$c \leftarrow \mathsf{Enc}_{k'_S}(pk, \mathsf{cert}, \sigma)$

$\longleftarrow$ $g^y,\ N_S,\ c$

Derive k'_S, k'_C, k_S, k_C from $K = g^{xy}$

$(pk, \mathsf{cert}, \sigma) \leftarrow \mathsf{Dec}_{k'_S}(c)$

Validate cert

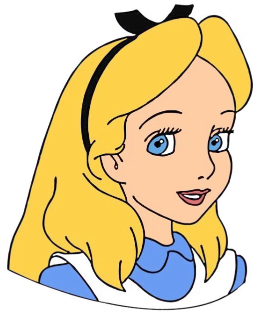
$\mathsf{Vrfy}_{pk_S}((G, q, g, g^x, N_C, g^y, N_S), \sigma)$

$t \leftarrow \mathsf{Mac}_{k'_C}(G, q, g, g^x, N_C, g^y, N_S, c)$ $\xrightarrow{\ t\ }$

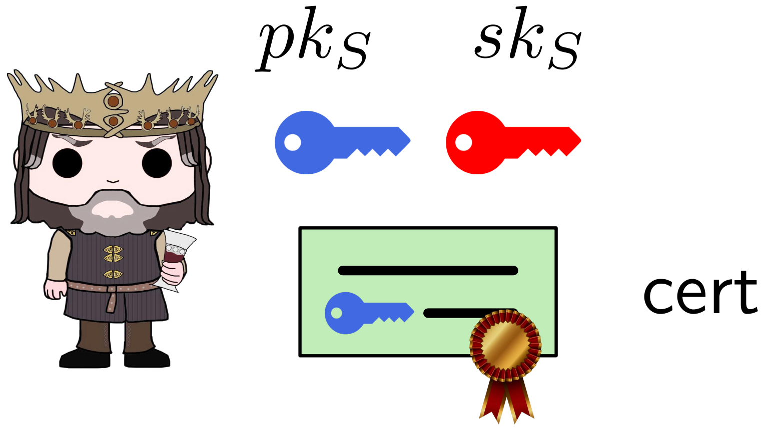
$\mathsf{Vrfy}_{k'_C}((G, q, g, g^x, N_C, g^y, N_S, c), t)$

The client knows that pk_S is the correct public key

TLS: Handshake protocol: Security (informal)



$(G, q, g, g^x),\ N_C$ $\longrightarrow$



Derive k'_S, k'_C, k_S, k_C from $K = g^{xy}$

$\sigma \leftarrow \mathsf{Sign}_{sk_S}((G, q, g, g^x, N_C, g^y, N_S))$

$c \leftarrow \mathsf{Enc}_{k'_S}(pk, \mathsf{cert}, \sigma)$

$\longleftarrow$ $g^y,\ N_S,\ c$

Derive k'_S, k'_C, k_S, k_C from $K = g^{xy}$

$(pk, \mathsf{cert}, \sigma) \leftarrow \mathsf{Dec}_{k'_S}(c)$

Validate cert

$\mathsf{Vrfy}_{pk_S}((G, q, g, g^x, N_C, g^y, N_S), \sigma)$

$t \leftarrow \mathsf{Mac}_{k'_C}(G, q, g, g^x, N_C, g^y, N_S, c)$

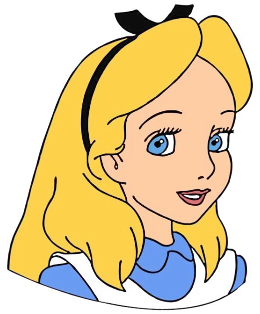
t $\longrightarrow$

$\mathsf{Vrfy}_{k'_C}((G, q, g, g^x, N_C, g^y, N_S, c), t)$

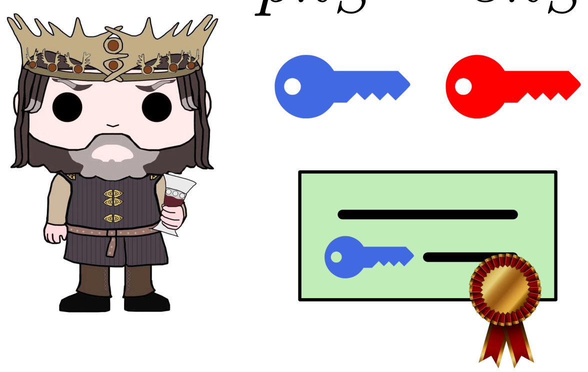
The client knows that pk_S is the correct public key

The client knows that it is talking to the right server

TLS: Handshake protocol: Security (informal)



(G, q, g, g^x), N_C $\longrightarrow$



pk_S sk_S

cert

Derive k'_S, k'_C, k_S, k_C from $K = g^{xy}$

$\sigma \leftarrow \mathsf{Sign}_{sk_S}((G, q, g, g^x, N_C, g^y, N_S))$

$c \leftarrow \mathsf{Enc}_{k'_S}(pk, \mathsf{cert}, \sigma)$

$\longleftarrow$ g^y, N_S, c

Derive k'_S, k'_C, k_S, k_C from $K = g^{xy}$

$(pk, \mathsf{cert}, \sigma) \leftarrow \mathsf{Dec}_{k'_S}(c)$

Validate cert

$\mathsf{Vrfy}_{pk_S}((G, q, g, g^x, N_C, g^y, N_S), \sigma)$

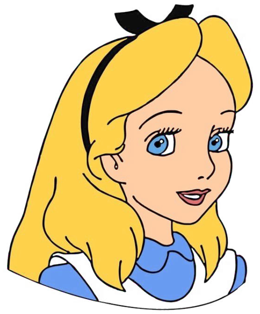
$t \leftarrow \mathsf{Mac}_{k'_C}(G, q, g, g^x, N_C, g^y, N_S, c)$ $\xrightarrow{t}$ $\mathsf{Vrfy}_{k'_C}((G, q, g, g^x, N_C, g^y, N_S, c), t)$

The client knows that pk_S is the correct public key

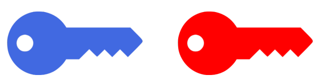
The client knows that it is talking to the right server

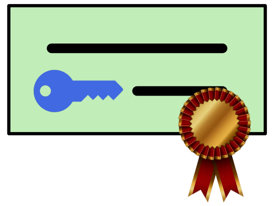
The signed messages have high-entropy: no replay attacks

TLS: Handshake protocol: Security (informal)



pk_S sk_S





cert

(G, q, g, g^x), N_C $\longrightarrow$

Derive k'_S, k'_C, k_S, k_C from $K = g^{xy}$

$\sigma \leftarrow \mathsf{Sign}_{sk_S}((G, q, g, g^x, N_C, g^y, N_S))$

$c \leftarrow \mathsf{Enc}_{k'_S}(pk, \mathsf{cert}, \sigma)$

$\longleftarrow$ g^y, N_S, c

Derive k'_S, k'_C, k_S, k_C from $K = g^{xy}$

$(pk, \mathsf{cert}, \sigma) \leftarrow \mathsf{Dec}_{k'_S}(c)$

Validate cert

$\mathsf{Vrfy}_{pk_S}((G, q, g, g^x, N_C, g^y, N_S), \sigma)$

$t \leftarrow \mathsf{Mac}_{k'_C}(G, q, g, g^x, N_C, g^y, N_S, c)$

t $\longrightarrow$

$\mathsf{Vrfy}_{k'_C}((G, q, g, g^x, N_C, g^y, N_S, c), t)$

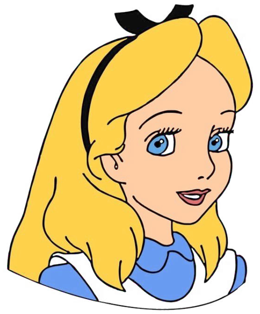
The client knows that pk_S is the correct public key

The client knows that it is talking to the right server

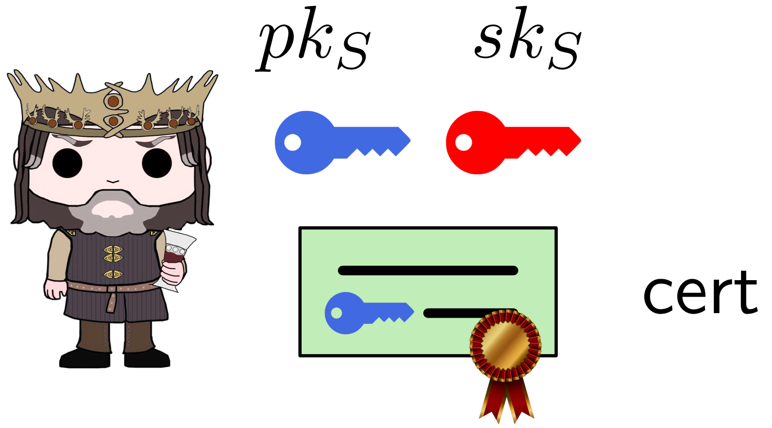
The signed messages have high-entropy: no replay attacks

All Diffie-Hellman messages are authenticated: no man in the middle attack

TLS: Handshake protocol: Security (informal)



(G, q, g, g^x), N_C →



Derive k'_S, k'_C, k_S, k_C from $K = g^{xy}$

$\sigma \leftarrow \mathsf{Sign}_{sk_S}((G, q, g, g^x, N_C, g^y, N_S))$

$c \leftarrow \mathsf{Enc}_{k'_S}(pk, \mathsf{cert}, \sigma)$

← g^y, N_S, c

Derive k'_S, k'_C, k_S, k_C from $K = g^{xy}$

$(pk, \mathsf{cert}, \sigma) \leftarrow \mathsf{Dec}_{k'_S}(c)$

Validate cert

$\mathsf{Vrfy}_{pk_S}((G, q, g, g^x, N_C, g^y, N_S), \sigma)$

$t \leftarrow \mathsf{Mac}_{k'_C}(G, q, g, g^x, N_C, g^y, N_S, c)$

t →

$\mathsf{Vrfy}_{k'_C}((G, q, g, g^x, N_C, g^y, N_S, c), t)$

The client knows that pk_S is the correct public key

The client knows that it is talking to the right server

The signed messages have high-entropy: no replay attacks

All Diffie-Hellman messages are authenticated: no man in the middle attack

Diffie-Hellman protocol: the adversary learns nothing about K (and k'_S, k'_C, k_S, k_C)

TLS: Handshake protocol: Security (informal)

Why do we use Diffie-Hellman?

Consider the following alternative protocol:

- The client verifies the server certificate
- The client picks a random secret key K
- The client encrypts K with pk_S ands sends the ciphertext c to the server
- The server decrypts c with sk_S, and replies using an authenticated encryption scheme Π with key K
- Communication continues using Π

TLS: Handshake protocol: Security (informal)

Why do we use Diffie-Hellman?

Consider the following alternative protocol:

- The client verifies the server certificate
- The client picks a random secret key K
- The client encrypts K with pk_S ands sends the ciphertext c to the server
- The server decrypts c with sk_S, and replies using an authenticated encryption scheme Π with key K
- Communication continues using Π

Does it work?

TLS: Handshake protocol: Security (informal)

Why do we use Diffie-Hellman?

Consider the following alternative protocol:

- The client verifies the server certificate
- The client picks a random secret key K
- The client encrypts K with pk_S ands sends the ciphertext c to the server
- The server decrypts c with sk_S, and replies using an authenticated encryption scheme Π with key K
- Communication continues using Π

Does it work? Yes, this was allowed in TLS 1.2. But...

TLS: Handshake protocol: Security (informal)

Why do we use Diffie-Hellman?

Consider the following alternative protocol:

- The client verifies the server certificate
- The client picks a random secret key K
- The client encrypts K with pk_S ands sends the ciphertext c to the server
- The server decrypts c with sk_S, and replies using an authenticated encryption scheme Π with key K
- Communication continues using Π

Does it work? Yes, this was allowed in TLS 1.2. But...

No forward secrecy!

TLS: Handshake protocol: Security (informal)

Why do we use Diffie-Hellman?

Consider the following alternative protocol:

- The client verifies the server certificate
- The client picks a random secret key K
- The client encrypts K with pk_S ands sends the ciphertext c to the server
- The server decrypts c with sk_S, and replies using an authenticated encryption scheme Π with key K
- Communication continues using Π

Does it work? Yes, this was allowed in TLS 1.2. But...

No forward secrecy!

- If sk_S is leaked or stolen at a later point in time, the secrecy of the past communication between client and server is compromised

TLS: Handshake protocol: Security (informal)

Why do we use Diffie-Hellman?

Consider the following alternative protocol:

- The client verifies the server certificate
- The client picks a random secret key K
- The client encrypts K with pk_S ands sends the ciphertext c to the server
- The server decrypts c with sk_S, and replies using an authenticated encryption scheme Π with key K
- Communication continues using Π

Does it work? Yes, this was allowed in TLS 1.2. But...

No forward secrecy!

- If sk_S is leaked or stolen at a later point in time, the secrecy of the past communication between client and server is compromised
- Using the Diffie-Hellman key-exchange, the symmetric keys k'_S, k'_C, k_S, k_C are **ephemeral** and can be erased at the end of the handshake/session.