Applied Cryptography
Spring Semester 2023
Lectures 22, 23, 24, 25, 26

Kenny Paterson (@kennyog)

Applied Cryptography Group

https://appliedcrypto.ethz.ch/

Overview of lectures

- Introducing Public Key Encryption
- The KEM/DEM paradigm
- RSA Encryption
- Diffie-Hellman Key Exchange and ElGamal Encryption
- Appendix: Basic Number Theory

Introducing Public Key Encryption

Introducing Public Key Encryption

- So far, we have considered symmetric encryption schemes.
- Participants need to somehow share key *K* in order to encrypt/decrypt.
- In Public Key Encryption (PKE), we use different keys for encryption and decryption.
  - Bob generates a public/private key pair.
  - Alice uses Bob's public key to encrypt to Bob.
  - Bob uses his private key to decrypt.
- Also called Asymmetric Encryption.

Syntax for PKE

A PKE scheme $\mathcal{PKE}$ consists of a triple of algorithms: $\mathcal{PKE}$ = (KGen,Enc,Dec).

KGen: randomised key generation, generates a key pair (sk,pk) $\in \mathcal{SK} \times \mathcal{PK}$.
(sk is the **private** key, pk the **public** key).

Enc: usually randomised, takes as input public key pk, plaintext $m \in \mathcal{M} \subseteq \{0, 1\}^*$ and produces output $c \in \mathcal{C} \subseteq \{0, 1\}^*$.

Dec: takes as input private key sk, ciphertext $c \in \{0, 1\}^*$ and produces output $m \in \mathcal{M}$, or an error message denoted $\perp$.

Correctness: we require that for all key pairs (sk,pk) output by KGen, and for all plaintexts m,

$$\mathrm{Dec}_{sk}(\mathrm{Enc}_{pk}(m)) = m.$$

A concrete example: Textbook RSA

KGen: generates:

- random primes p, q of some bit-size $k/2$, with $N = pq$ (so bit-size of N is k);
- integers d, e such that $de = 1 \bmod (p-1)(q-1)$.

  Outputs key pair (sk, pk) where $sk = d$, $pk = (e, N)$.

Enc: inputs public key $pk = (e, N)$, plaintext $m \in [1, N-1]$; outputs $c = m^e \bmod N$.

Dec: inputs private key $sk = d$, ciphertext c; outputs $m = c^d \bmod N$.

Correctness: follows from property that if $de = 1 \bmod (p-1)(q-1)$ then $m^{de} = m \bmod N$ for all $m \in [1, N-1]$ (this demands a mathematical proof).

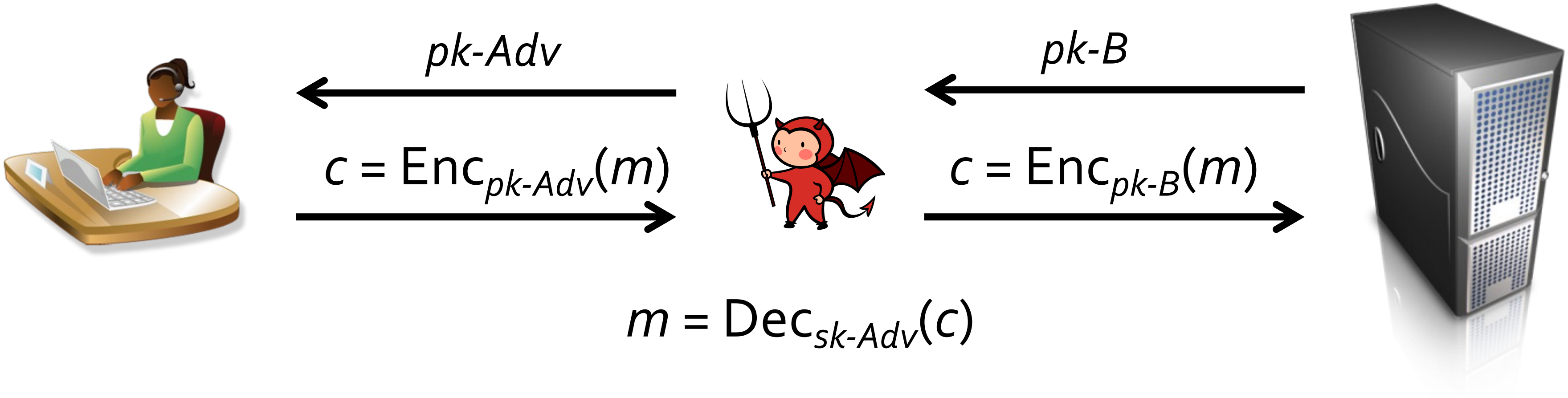
- Implementation issues: how do we generate random primes of a given bit-size? How do we generate d and e? How do we encode messages as integers in the interval $[1, N-1]$?
- Enc is not randomised here, which already hints at trouble.
- **This scheme is not secure. It must not be used as is in practice.**

What is PKE good for?

- PKE is typically much more expensive than symmetric encryption (in speed, key-size, etc)…
- Example: for textbook RSA, encryption involves an exponentiation mod *N*, where *N* typically has 2048 bits.
- PKE is often used in applications to transport symmetric keys which are then used to encrypt bulk data.
- This use of PKE is often referred to as hybrid encryption.
  - The symmetric keys are then used in a symmetric encryption scheme (e.g. nonce-based AEAD) to encrypt the actual data.
  - We will return to this in a few slides time.
- PKE is occasionally used to directly transport short, infrequent messages (e.g. PINs from card to payment terminal in EMV system).

What is PKE good for?

- Main problem with PKE: it requires distribution of **authentic** public keys.
- How does Alice know that a public key is genuinely Bob's?



$pk\text{-}Adv$

$c = \mathrm{Enc}_{pk\text{-}Adv}(m)$

$pk\text{-}B$

$c = \mathrm{Enc}_{pk\text{-}B}(m)$

$m = \mathrm{Dec}_{sk\text{-}Adv}(c)$

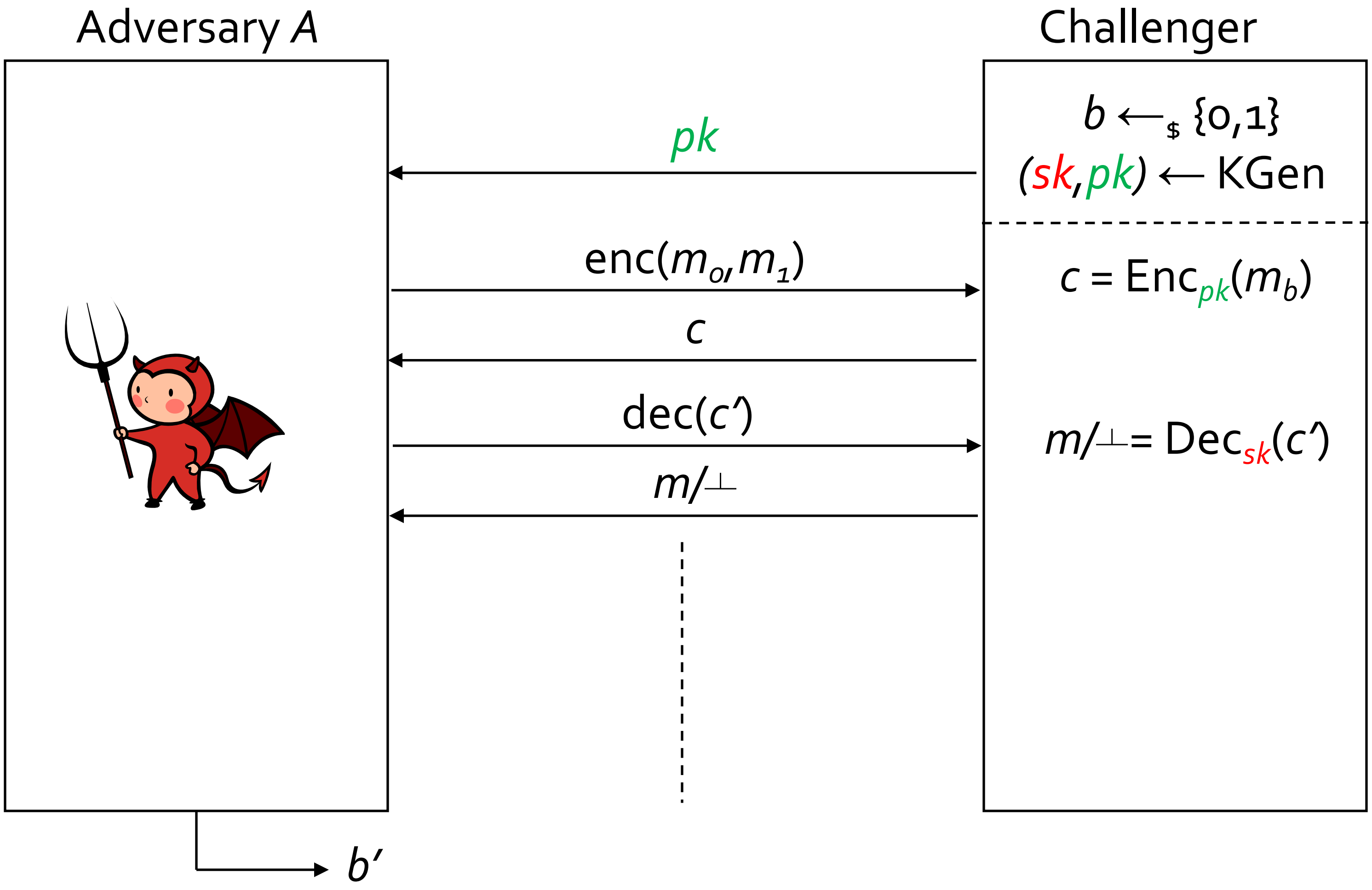
- At a high level, PKE translates the problem of symmetric key distribution into the problem of distribution of authentic public keys.
- **Question: Is that an easier problem?**

Formalising security for PKE

- As with symmetric encryption, it is not enough that the secret key should be hard to guess: we want confidentiality of messages, more precisely indistinguishability under chosen ciphertext attacks (IND-CCA).
- Captures the idea that nothing about the plaintext should leak to the adversary, even when the adversary can obtain decryptions of arbitrary messages.
  - Adversary has access to an *encryption* oracle and a *decryption* oracle for key pair (sk,pk).
  - Adversary can submit arbitrary pairs of messages (m_0,m_1) to the encryption oracle, and receives $\mathrm{Enc}_{pk}(m_b)$.
  - Adversary can submit arbitrary* bit-strings c' to the decryption oracle and receives $\mathrm{Dec}_{sk}(c')$.
  - Adversary has to recover the bit b.
  - Compare to definition for IND-CCA security of *symmetric* encryption.

*Adversary cannot submit an output from encryption oracle to decryption oracle, otherwise he can win trivially.

IND-CCA security for PKE in a picture



Adversary A

Challenger

$b \leftarrow_{\$} \{0,1\}$

$(sk, pk) \leftarrow \mathrm{KGen}$

pk

$\mathrm{enc}(m_0, m_1)$

$c = \mathrm{Enc}_{pk}(m_b)$

c

$\mathrm{dec}(c')$

$m/\perp = \mathrm{Dec}_{sk}(c')$

$m/\perp$

b'

$$\mathrm{Adv}^{IND\text{-}CCA}_{\mathcal{PKE}}(A) := 2|\Pr(b=b') - 1/2|.$$

IND-CCA security for PKE

- The adversary's *advantage* in the IND-CCA security game is defined to be:

$$\mathrm{Adv}_{\mathcal{PKE}}^{IND\text{-}CCA}(A) := 2|\Pr(b=b') - 1/2|.$$

- A PKE scheme $\mathcal{PKE} =$ (KGen, Enc, Dec) is said to be IND-CCA-secure if the advantage is "small" for *any* adversary using "reasonable" resources.
- More precisely, we say that a scheme $\mathcal{PKE}$ is $(q_e, q_d, t, \varepsilon)$-IND-CCA-secure if no adversary running in time t, making q_e enc(·) queries and q_d dec(·) queries has advantage more than ε.
- As with SE, we will provide concrete security reductions to underlying hard problems.
- One can also use complexity-theoretic notions such a poly-time, negligible, etc, after introducing a security parameter κ to dictate key sizes.
- Usually only a single enc(·) query is allowed; one can show equivalence to model with multiple such queries with a factor of q_e loss in security.
- IND-CPA security: as IND-CCA but remove the decryption oracle.

Implications of the IND-CCA definition

Q: Suppose an attacker can recover *sk* from *pk* by some means for a PKE scheme. Can the scheme be IND-CCA secure?

A: No. Attacker can just decrypt c = $\text{Enc}_{pk}(m_b)$.

Q. Suppose a scheme is IND-CCA secure. Is it necessarily IND-CPA secure?

A. Yes, because removing the decryption oracle only makes the adversary less powerful. (Can easily provide a reduction proof.)

Q. Suppose a PKE scheme has a deterministic encryption algorithm. Can it be IND-CPA secure?

A. No. Make one encryption query on (m_0, m_1) to receive c; then encrypt both m_0 and m_1 directly using *pk* and compare ciphertexts.

Q. Is textbook RSA IND-CCA secure? IND-CPA secure?

A. It's deterministic, so it cannot achieve either notion.

Hyrbid Encryption + KEM/DEM Paradigm

Key Encapsulation Mechanisms (KEMs)

A KEM $\mathcal{KEM}$ consists of a triple of algorithms: $\mathcal{KEM}$ = (KGen,Encap,Decap).

KGen: randomised key generation, generates a key pair (sk,pk) $\in \mathcal{SK} \times \mathcal{PK}$.
(sk is the **private** key, pk the **public** key).

Encap: usually randomised, takes as input public key pk, and produces output (c, K) $\in \mathcal{C} \times \mathcal{K}$.

Decap: takes as input private key sk, encapsulation $c \in \{0, 1\}^*$ and produces output $K \in \mathcal{K}$, or an error message denoted $\perp$.

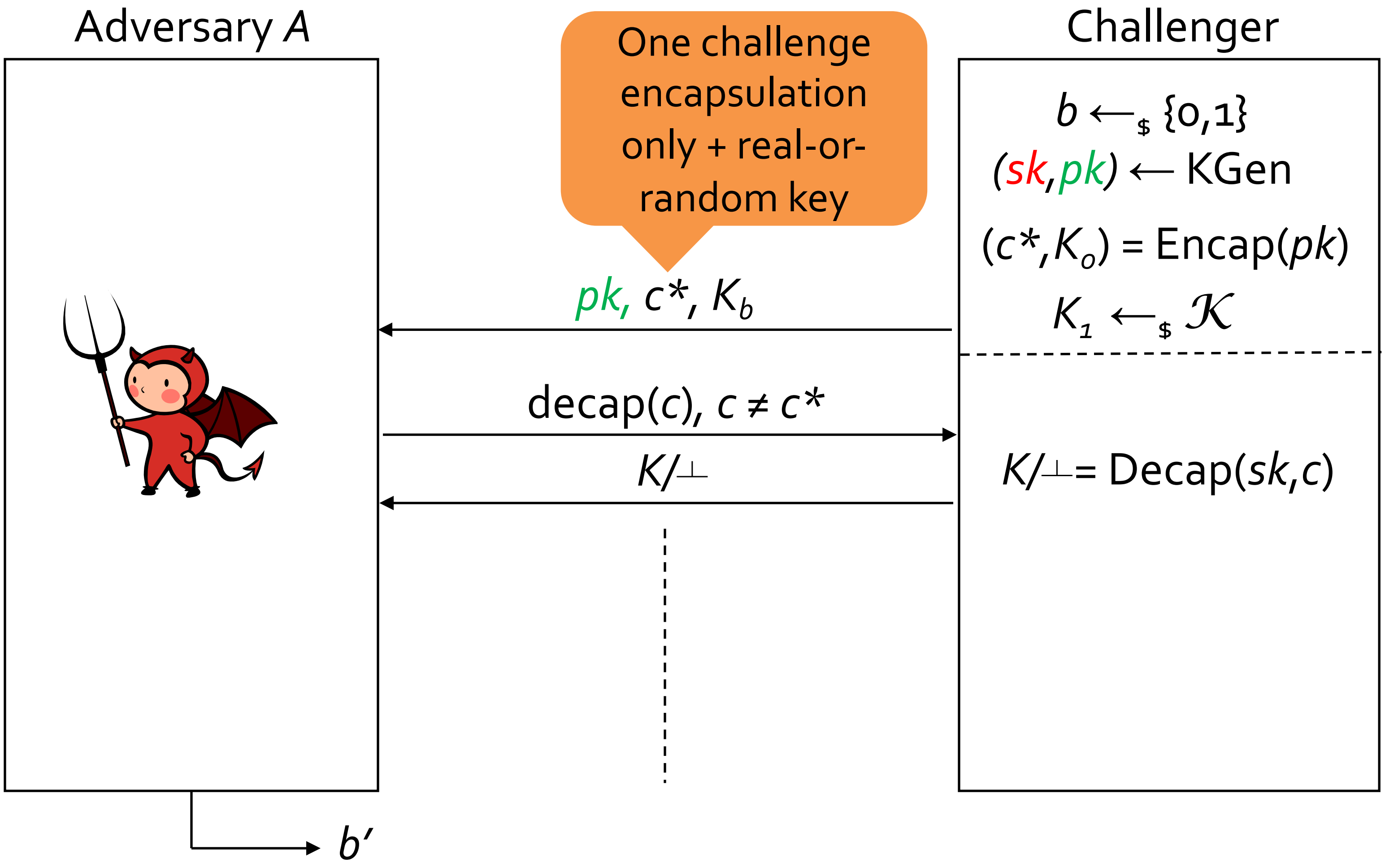
Correctness: for all key pairs (sk,pk) output by KGen,

if $(c, K) \leftarrow \text{Encap}(pk)$,

then $K \leftarrow \text{Decap}(sk,c)$.

Comparison to PKE: Encap has no message input; Encap internally generates a key K and its encapsulation c.

IND-CCA security for KEMs in a picture



Adversary A

One challenge encapsulation only + real-or-random key

Challenger

$b \leftarrow_{\$} \{0,1\}$

$(sk, pk) \leftarrow \mathsf{KGen}$

$(c^*, K_0) = \mathsf{Encap}(pk)$

$K_1 \leftarrow_{\$} \mathcal{K}$

pk, c^*, K_b

decap(c), $c \neq c^*$

$K/\perp$

$K/\perp = \mathsf{Decap}(sk, c)$

b'

$$\mathrm{Adv}_{\mathcal{KEM}}^{IND\text{-}CCA}(A) := 2|\Pr(b=b') - 1/2|.$$

The KEM/DEM Composition Paradigm

A Data Encapsulation Mechanism (DEM) is nothing other than a symmetric encryption scheme $\mathcal{DEM}$, with algorithms (DEM.KGen,DEM.Enc,DEM.Dec).

KEM/DEM Composition:

Let $\mathcal{KEM}$ = (KGen,Encap,Decap) be a KEM and $\mathcal{DEM}$ = (DEM.KGen,DEM.Enc,DEM.Dec) be a DEM such that $\mathcal{KEM}.\mathcal{K} = \mathcal{DEM}.\mathcal{K}$.

Then we build a PKE scheme $\mathcal{PKE}$ = (PKE.KGen,PKE.Enc,PKE.Dec) from $\mathcal{KEM}$ and $\mathcal{DEM}$ as follows:

```
PKE.KGen:              $(sk,pk) \leftarrow$ KEM.KGen; return $(sk,pk)$.
PKE.Enc($pk,m$):       1. $(c_0,K) \leftarrow$ KEM.Encap($pk$);
                       2. $c_1 \leftarrow$ DEM.Enc($K,m$);
                       3. return $(c_0,c_1)$.
PKE.Dec($sk, (c_0,c_1)$):  1. $m \leftarrow \perp$;
                       2. $K \leftarrow$ KEM.Decap($sk,c_0$);
                       3. If $K \neq \perp$ then $m \leftarrow$ DEM.Dec($K,c_1$);
                       4. return $m$.
```

NB: message space for $\mathcal{PKE}$ is $\mathcal{DEM}.\mathcal{M}$; ciphertext space for $\mathcal{PKE}$ is $\mathcal{KEM}.\mathcal{C}$ x $\mathcal{DEM}.\mathcal{C}$.

The KEM/DEM Composition Paradigm

Theorem:

Suppose $\mathcal{PKE}$ is built from $\mathcal{KEM}$ and $\mathcal{DEM}$ as above. If $\mathcal{KEM}$ is IND-CCA secure and $\mathcal{DEM}$ is IND-CCA secure, then $\mathcal{PKE}$ is IND-CCA secure.

We give a proof for this theorem for a PKE adversary A making exactly one encryption oracle query ($q_e = 1$).

More formally, we show for any IND-CCA adversary A against PKE with q_e=1, there exist adversaries B and C such that:

$$\mathrm{Adv}^{\mathit{IND\text{-}CCA}}_{\mathcal{PKE}}(A) \leq 2 \cdot \mathrm{Adv}^{\mathit{IND\text{-}CCA}}_{\mathcal{KEM}}(B) + \mathrm{Adv}^{\mathit{IND\text{-}CCA}}_{\mathcal{DEM}}(C).$$

Moreover, B and C run in the same time as A. If A makes q_d decryption oracle queries, then B makes at most q_d decapsulation queries and C makes at most q_d decryption queries.

The KEM/DEM Composition Paradigm

Proof (sketch):

The proof involves a sequence of games G_0, G_1, G_2, G_3.

G_0: A plays the normal IND-CCA game for PKE, with q_e=1; let $c = (c_0,c_1)$ denote the response to A's single encryption oracle query on (m_0,m_1); let b be the hidden bit chosen by A's challenger. By construction:

$$(c_0, K_0) \leftarrow \text{KEM.Encap}(pk) \text{ and } c_1 \leftarrow \text{DEM.Enc}(K_0, m_b).$$

PKE dec(·) oracle: as per the PKE scheme, using private key *sk*.

G_1: A plays the normal IND-CCA game for PKE, with q_e=1; the response (c_0,c_1) to A's encryption oracle query on (m_0,m_1) is constructed using "real" output from an IND-CCA challenger in the KEM security game. By construction:

$$(c_0, K_0) \leftarrow \text{KEM.Encap}(pk) \text{ and } c_1 \leftarrow \text{DEM.Enc}(K_0, m_b).$$

PKE dec(·) oracle: on input (c'_0,c'_1): if $c'_0 \neq c_0$ then we use a KEM decap(·) oracle from KEM security game to get a key K (or $\perp$) and then output DEM.Dec(K,c'_1); otherwise, (when $c'_0 = c_0$), we decrypt c'_1 using K_0.

The KEM/DEM Composition Paradigm

Proof (sketch):

G_2: As G_1 but the response (c_0, c_1) to A's encryption oracle query on (m_0, m_1) is constructed using "random" output K_1 from an IND-CCA challenger in the KEM security game. By construction:

$$(c_0, K_0) \leftarrow \text{KEM.Encap}(pk) \text{ and } c_1 \leftarrow \text{DEM.Enc}(\boldsymbol{K_1}, m_b) \textbf{ where } \boldsymbol{K_1} \leftarrow_{\$} \mathcal{K}.$$

PKE dec(·) oracle: on input (c'_0, c'_1): if $c'_0 \neq c_0$ then we use the KEM decap(·) oracle on c'_0 to get a key K (or $\perp$) and then output DEM.Dec(K,c'_1); otherwise (when $c'_0 = c_0$), we output DEM.Dec($\boldsymbol{K_1}$,c'_1).

G_3: As G_2 but we now run KEM.KGen to get (sk, pk), obtain c_0 by running KEM.Encap(pk), and obtain c_1 from a call to the enc(·) oracle in IND-CCA security game for the DEM (with hidden bit b). By construction:

$$(c_0, K_0) \leftarrow \text{KEM.Encap}(pk) \text{ and } c_1 \leftarrow \text{DEM.Enc}(\boldsymbol{K_1}, m_b) \textbf{ where } \boldsymbol{K_1} \leftarrow_{\$} \mathcal{K}.$$

PKE dec(·) oracle: on input (c'_0, c'_1): if $c'_0 \neq c_0$ then we use sk; otherwise (when $c'_0 = c_0$), we use a call to the dec(·) oracle in DEM security game.

The KEM/DEM Composition Paradigm

Proof (sketch):

The proof involves a sequence of games G_0, G_1, G_2 , G_3.

From G_0 to G_1: this is mostly a syntactic change, where we change how decryption is done (introducing a KEM challenger and its oracles).

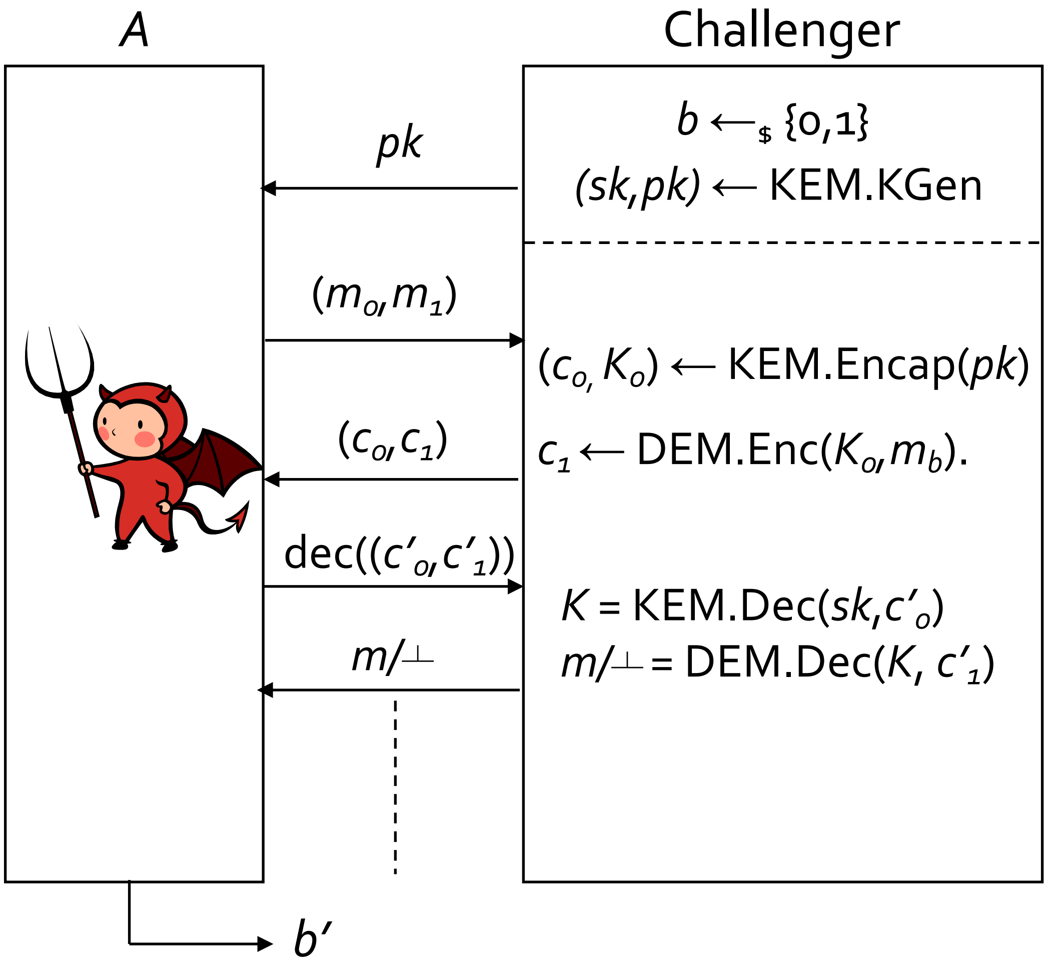
From G_1 to G_2: we can construct an IND-CCA adversary *B* for the KEM that “bridges” between these games (G_1 corresponds to *b*=0 case, G_2 to *b*=1 case in KEM game). Doing this allows us to use K_0 in the KEM but **independent, random** K_1 in the DEM.

From G_2 to G_3: this is again mostly a syntactic change, where we modify how encryption and decryption are done (using DEM oracles instead of KEM oracles).

In G_3: an adversary *A* who wins in this game can be used to construct an IND-CCA adversary *C* against the DEM.

A standard computation involving probabilities of events W_i (the event that *A* outputs $b' = b$ in Game G_i) completes the proof.

G_0



A

Challenger

$b \leftarrow_{\$} \{0,1\}$

$(sk,pk) \leftarrow$ KEM.KGen

pk

(m_0,m_1)

$(c_0, K_0) \leftarrow$ KEM.Encap(pk)

$c_1 \leftarrow$ DEM.Enc(K_0,m_b).

(c_0,c_1)

dec((c'_0,c'_1))

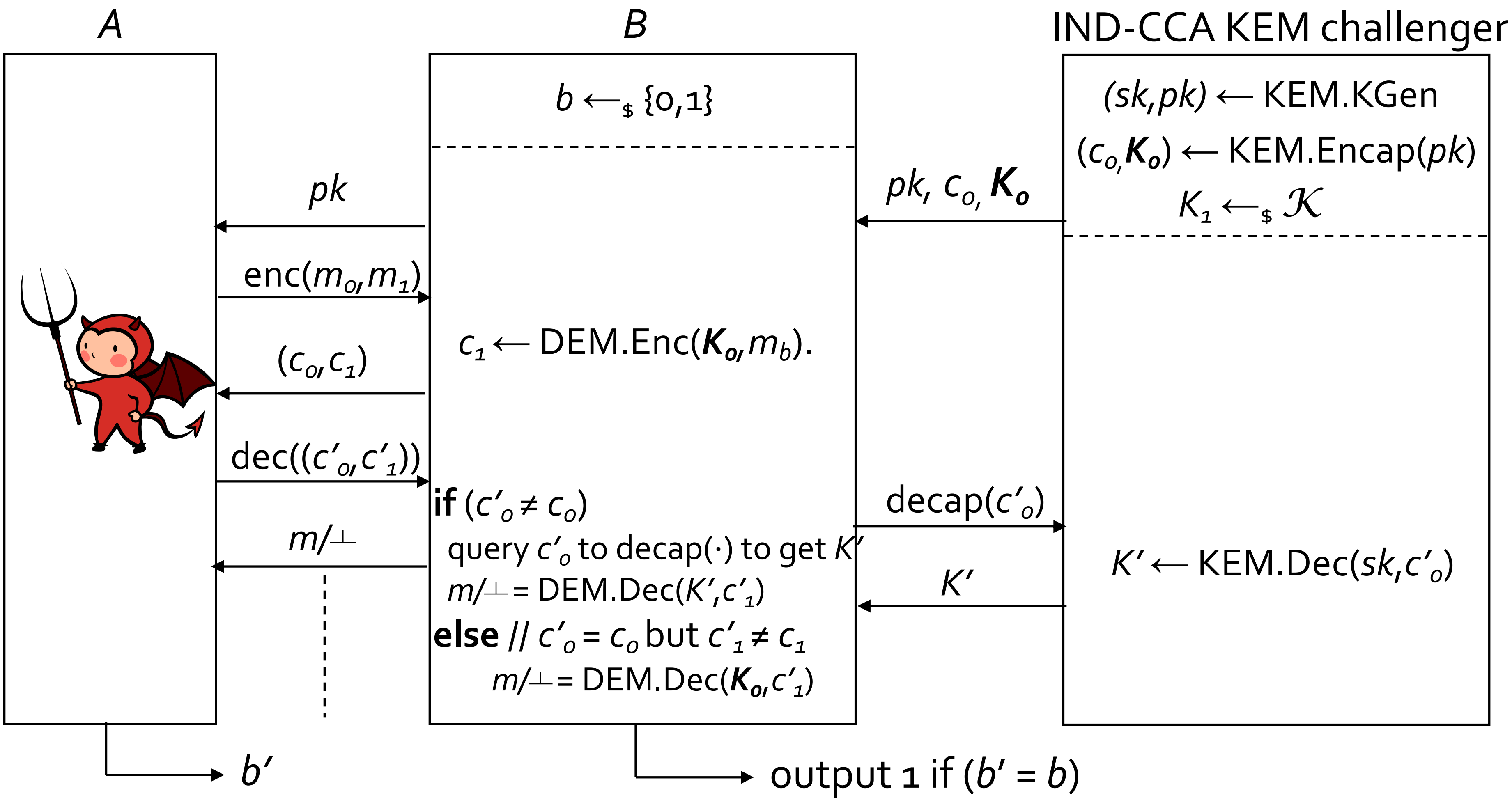
K = KEM.Dec(sk,c'_0)

$m/\perp$ = DEM.Dec(K, c'_1)

$m/\perp$

b'

G_1



A

B

IND-CCA KEM challenger

$(sk,pk) \leftarrow$ KEM.KGen

$(c_0, \boldsymbol{K_0}) \leftarrow$ KEM.Encap(pk)

$K_1 \leftarrow_\$ \mathcal{K}$

$b \leftarrow_\$ \{0,1\}$

$pk, c_0, \boldsymbol{K_0}$

pk

enc(m_0,m_1)

$c_1 \leftarrow$ DEM.Enc($\boldsymbol{K_0}, m_b$).

(c_0,c_1)

dec((c'_0,c'_1))

if ($c'_0 \neq c_0$)

  query c'_0 to decap(·) to get K'

  $m/\perp$ = DEM.Dec(K',c'_1)

else // $c'_0 = c_0$ but $c'_1 \neq c_1$

  $m/\perp$ = DEM.Dec($\boldsymbol{K_0}, c'_1$)

decap(c'_0)

$K' \leftarrow$ KEM.Dec(sk,c'_0)

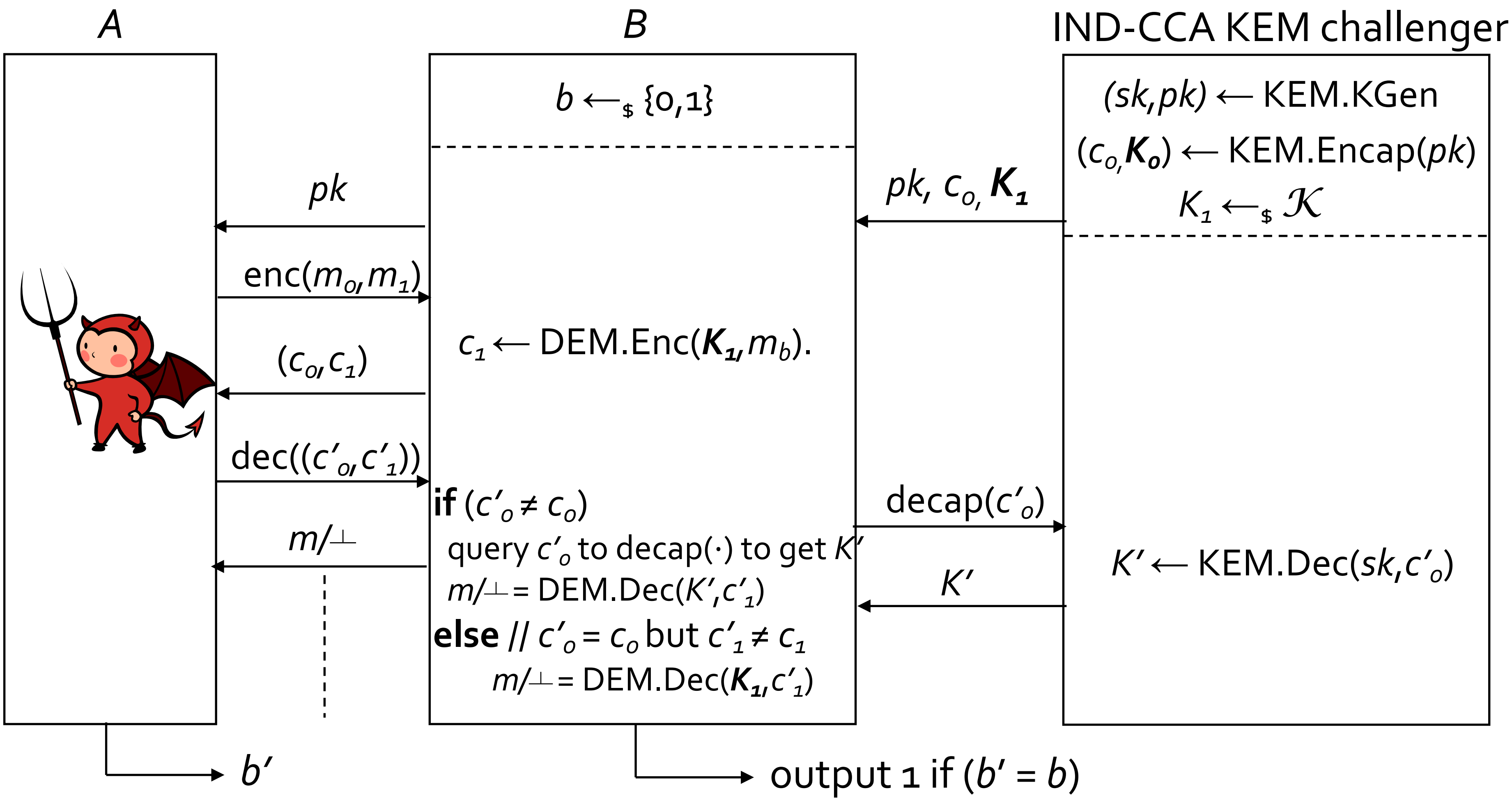
K'

$m/\perp$

b'

output 1 if ($b' = b$)

G_2



A

B

IND-CCA KEM challenger

$b \leftarrow_{\$} \{0,1\}$

$(sk,pk) \leftarrow$ KEM.KGen

$(c_0, \boldsymbol{K_0}) \leftarrow$ KEM.Encap(pk)

$K_1 \leftarrow_{\$} \mathcal{K}$

$pk, c_0, \boldsymbol{K_1}$

pk

enc(m_0, m_1)

$c_1 \leftarrow$ DEM.Enc($\boldsymbol{K_1}, m_b$).

(c_0, c_1)

dec((c'_0, c'_1))

if ($c'_0 \neq c_0$)
  query c'_0 to decap(·) to get K'
  $m/\perp$ = DEM.Dec(K', c'_1)
else // $c'_0 = c_0$ but $c'_1 \neq c_1$
  $m/\perp$ = DEM.Dec($\boldsymbol{K_1}, c'_1$)

decap(c'_0)

K'

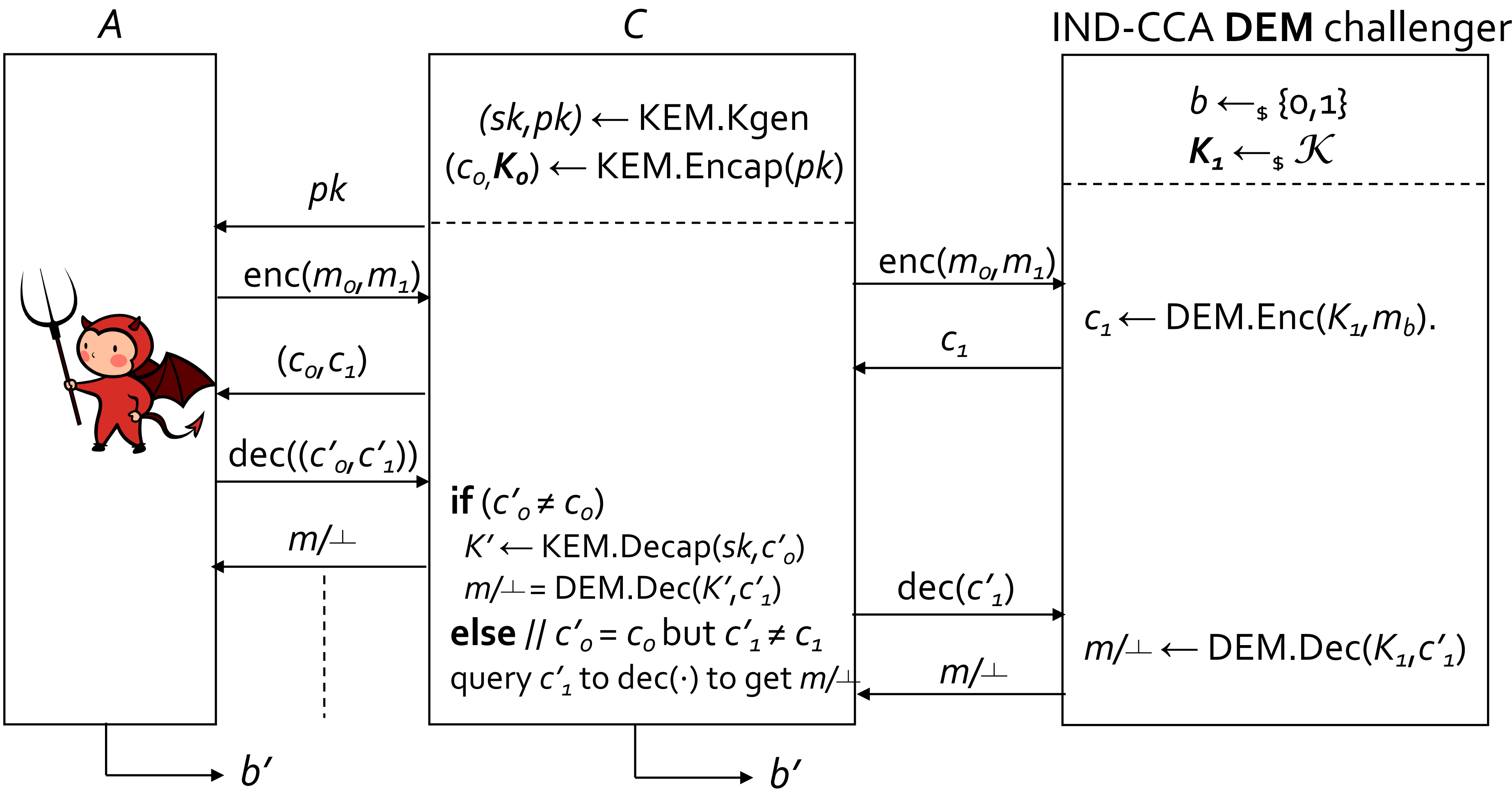
$K' \leftarrow$ KEM.Dec(sk, c'_0)

$m/\perp$

b'

output 1 if ($b' = b$)

G_3



A

C

IND-CCA **DEM** challenger

$(sk,pk) \leftarrow$ KEM.Kgen

$(c_0, \boldsymbol{K_0}) \leftarrow$ KEM.Encap(pk)

$b \leftarrow_\$ \{0,1\}$

$\boldsymbol{K_1} \leftarrow_\$ \mathcal{K}$

pk

enc(m_0,m_1)

(c_0,c_1)

dec((c'_0,c'_1))

$m/\perp$

enc(m_0,m_1)

$c_1 \leftarrow$ DEM.Enc(K_1,m_b).

c_1

if ($c'_0 \neq c_0$)

$K' \leftarrow$ KEM.Decap(sk,c'_0)

$m/\perp$ = DEM.Dec(K',c'_1)

else // $c'_0 = c_0$ but $c'_1 \neq c_1$

query c'_1 to dec(·) to get $m/\perp$

dec(c'_1)

$m/\perp \leftarrow$ DEM.Dec(K_1,c'_1)

$m/\perp$

b'

b'

The KEM/DEM Composition Paradigm

Notes:

- In game G_3, *C* is an IND-CCA adversary against the DEM that only makes one encryption query (but many decryption queries).
  - This implies that only ”one-time” security of the DEM is required.
- The proof is only for an IND-CCA adversary *A* against the PKE scheme that makes a single encryption query.
  - It can be extended to an adversary making q_e queries, at the cost of some complexity in the proof and some factors q_e in the bounds.
  - But one-time security of the DEM still suffices.
- We don’t have an integrity definition for PKE/KEMs; the public-key setting makes creating ciphertext/encapsulation forgeries trivial.

RSA Encryption

Recap: Textbook RSA

KGen: generates a pair of random primes p, q of some bit-size $k/2$, and integers d, e such that $de = 1 \bmod (p-1)(q-1)$. Set $N = pq$. Output key pair (sk,pk) where $sk = d$, $pk = (e,N)$.

Enc: inputs public key $pk = (e,N)$, plaintext $m \in [1,N-1]$; output $c = m^e \bmod N$.

Dec: inputs private key $sk = d$, ciphertext c; output $m = c^d \bmod N$.

Recall:

- **This scheme is not secure and must not be used in practice.**
- **It's not even randomised, so has no chance of even being IND-CPA secure.**

Generating keys for RSA

KGen: generates a pair of random primes p, q of some bit-size $k/2$, and integers d, e such that $de = 1 \bmod (p-1)(q-1)$. Set $N = pq$. Output key pair (sk,pk) where $sk = d$, $pk = (e,N)$.

Generating large, random primes with k/2 bits:

- Needs a good source of randomness and an efficient primality test with low error rate.

Many things can go wrong, part 1:

“Mining your p’s and q’s”:

- $N_1 = \mathbf{p_1}q_1$, $N_2 = \mathbf{p_1}q_2 \Rightarrow p_1 = \gcd(N_{1.},N_2) \Rightarrow$ easy recovery of common prime p_1 and break of both keys.
- If M distinct RSA moduli are available, we can compute $O(M^2)$ pairwise gcds using an $O(M \log M)$ algorithm due to Bernstein.
- In 2012, this attack broke 0.50% of server public RSA keys on the Internet.
- Root cause: insufficient entropy in RSA key generation process.
- https://factorable.net/weakkeys12.extended.pdf.

Generating keys for RSA

Many things can go wrong, part 2:

ROCA attack on an over-optimised prime generation algorithm.

- Using *p* and *q* of special form to speed-up prime generation on smart-cards led to a major vulnerability in millions of smart-cards, including Estonian ID card system.
- https://crocs.fi.muni.cz/_media/public/papers/nemec_roca_ccs17_preprint.pdf

Take-aways:

- Implementations of even a 40-year-old algorithm can (and do) still get things wrong.
- Use a well-vetted library and standardised approaches to parameter selection.

Generating keys for RSA

KGen: generates a pair of random primes p, q of some bit-size $k/2$, and integers d, e such that $de = 1 \bmod (p-1)(q-1)$. Set $N = pq$. Output key pair (sk,pk) where $sk = d$, $pk = (e, N)$.

Solving $de = 1 \bmod (p-1)(q-1)$:

- Suppose e is coprime to $(p-1)(q-1)$.
- Running the extended Euclid algorithm yields integers s, t such that

$$e \cdot s + (p-1)(q-1) \cdot t = 1.$$

- Reduce modulo $(p-1)(q-1)$ to get:

$$e \cdot s = 1 \mod (p-1)(q-1).$$

- Then take $d = s$.
- $e = 2^{16} + 1$ is often used in practice.
  - Faster encryption, no known weaknesses, e is prime and highly likely coprime to $(p-1)(q-1)$.
  - Many other tempting optimization lead to vulnerabilities (e.g. small d).

Using Chinese Remainder Theorem in RSA

Dec: inputs private key $sk = d$, ciphertext c; output $m = c^d \mod N$.

Textbook RSA decryption can be made faster by working mod p and mod q, and then combining the results using the Chinese Remainder Theorem (CRT).

Typically then include p, q and $u := q^{-1} \bmod p$ as part of the private key.

To decrypt:

- Set $f = d \bmod p\text{-}1$; $g = d \bmod q\text{-}1$
- Compute $m_p = c^f \bmod p$
- Compute $m_q = c^g \bmod q$
- Set $h = u(m_p - m_q) \bmod p$
- Output $m_q + hq \bmod N$

It is not too hard to show that this correctly implements RSA decryption.

Working with exponents f and g (mod p and mod q, respectively) makes the arithmetic operations much faster than direct implementation of $m = c^d \mod N$.

Many things can go wrong, Part 3:

- In MEGA cloud storage system, (d, p, q, $u=q^{-1} \bmod p$) are stored on server encrypted under a symmetric key K known only to the client **using ECB mode**!
- On login, client fetches ECB-encrypted (d, p, q, u) from server, decrypts to obtain (d, p, q, u), recomputes $u'=q^{-1} \bmod p$ and tests if $u'=u$ to sanity check the key.
- But an adversarial cloud server can **overwrite** blocks of private key with chosen values, because the system exposes an encryption oracle for key K (due to a key reuse vulnerability).
- So the client tries to compute $(q')^{-1} \bmod p'$ where now p' and q' are controlled by the adversary.
  - This process succeeds if and only if q' is coprime to p'.
  - And if it fails an error message is returned to the server.
  - Based on presence/absence of the error message, information about coprimality of p', q' leaks to the adversary.
  - **Key trick**: make q' depend on selected blocks of d by exploiting malleability of ECB mode when doing private key overwriting.
  - Allows complete recovery of d using around $2^{11} – 2^{12}$ login attempts.
  - Full details at: https://mega-caveat.github.io/

Keysize requirements

- One way to break RSA is to recover the private key d from the public key (e,N) by first factorising N to find p, q and then solving $de = 1 \bmod (p-1)(q-1)$.
- Other attacks may be possible, but we have to *at least* make sure that factorising N is hard.
- The Integer Factorisation Problem (IFP) has been studied for thousands of years, and intensively since the 1970s (because of its importance in cryptography).
- The current best algorithm for solving IFP on a classical computer is called the Number Field Sieve (NFS).
- This algorithm was invented in the early 1990s, and there has been no significant **algorithmic** improvement in IFP since then.
- But Moore's law + code improvements have had a significant effect.
- Running time + space required by NFS:

$$\exp\left[(c+o(1))\,(\ln N)^{1/3}\,(\ln \ln N)^{2/3}\right] \quad \text{with} \quad c = (64/9)^{1/3}.$$

- Super-polynomial but sub-exponential in the bit-size of N.

Keysize requirements

- In 2015, a **512-bit RSA modulus** could be factored using NFS on Amazon EC2 in about 4 hours at a cost of USD75.
    - See https://eprint.iacr.org/2015/1000.
- Between 2009 and 2019, the biggest modulus publicly factored was a **768-bit** RSA modulus, one of the RSA challenges.
    - Using the equivalent of about 2,000 core years on 2.2 GHz CPUs.
    - See https://eprint.iacr.org/2010/006.pdf.
- Nov. 2019: RSA-240 (**795** bits), Boudot et al.: 900 core years – significantly less computation than the 2009 record!
- Feb. 2020: RSA-250 (**829** bits), Boudot et al.: 2700 core years on Intel Xeon Gold 6130 CPUs as a reference (2.1GHz), using CadoNFS software.
- See https://lists.gforge.inria.fr/pipermail/cado-nfs-discuss/2020-February/001166.html and https://arxiv.org/abs/2006.06197

RSA-250

2140324650240744961264423072839333563008614715144755017797
7549208814180234471401366433455190958046796109928518724709
1458768739626192155736304745477052080511905649310668769159
0019759405693457452230589325976697471681738069364894699871
578494975937497937

=

6413528947707158027879019017057738908482501474294344720811
6859632024532344630238623598752668347708737661925585694639
798853367

*

3337202759497815655622601060535511422794076034476755466678
4520987023841729210037080257448673296881877565718986258036
932062711

Keysize requirements

- It is conjectured that a **1024-bit modulus** would require about 2^{80} machine operations, hence providing 80 bits of security.
  - This is well beyond the capabilities of academic groups but probably within reach for a national security agency with a large budget.

- To achieve 128-bit security, the modulus *N* needs to be somewhere between **2048** and **3072** bits.
  - Estimates vary – BSI vs NIST vs ECRYPT.
  - See https://www.keylength.com.
  - Most websites still using RSA keys now use keys of 2048 bits, a few use 3072.
  - Question: Why not just "go large" on keysize?

Other attacks on RSA: Malleability

- We already know that textbook RSA is not IND-CPA secure, and we are going to need some kind of randomisation.
- But this is not all: textbook RSA also has a dangerous **malleability** property:
  - Suppose $c = m^e \bmod N$.
  - Choose s arbitrarily from $[1, N-1]$.
  - Consider $s^e \cdot c \bmod N$:

$$s^e \cdot c = s^e \cdot m^e = (s \cdot m)^e \bmod N.$$

  - So $s^e \cdot c \bmod N$ is an encryption of $s \cdot m \bmod N$ in textbook RSA!
  - This gives an active attacker the ability to modify an RSA plaintext in a controlled fashion, by carefully modifying ciphertexts.
  - This leads to various attacks against deployed systems.

Other attacks on RSA: Small e

- Using small exponents, such as $e = 3$, can speed up encryption.
- Suppose Alice has public key ($e=3$, N) and suppose Bob is only sending a very small message $m < N^{1/3}$.
- Then $c = m^3$ over the integers, i.e., no modular reduction has taken place.
- If Eve knows that m will be very small then she can recover the message m from the ciphertext c simply by taking cube roots over the integers (trivial, using Newton's method).
- Example:

  N has 1024 bits, m is an AES key with 128 bits, $e = 3$.

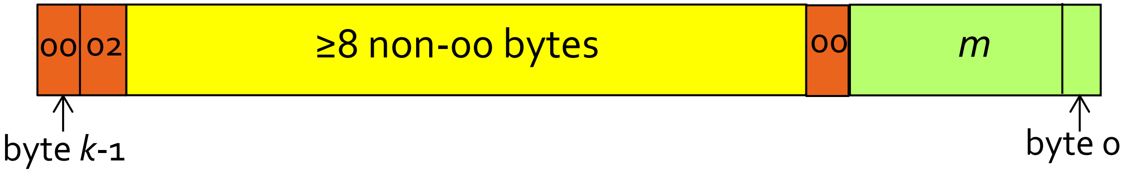
  Then m^e has only 3 x 128 = 384 bits, much less than 1024 bits.
- To mitigate such attacks, **we need some kind of message padding**.
- Small d is also insecure even up to $d \approx N^{1/4}$ (Wiener's attack).

Padding for RSA

- In response to attacks like those we've discussed, special padding schemes for the RSA setting have been introduced.
- The three goals of such padding schemes are to:
  1. Introduce randomness into the message.
  2. Expand short messages to full size.
  3. Destroy algebraic relationships among messages to remove the malleability property.

- The ultimate aim should be to achieve IND-CCA security for RSA encryption.
- This has proved tricky to do, and even today the most widely-deployed RSA padding scheme, called PKCS#1 v1.5, does not achieve IND-CCA security.
- We do have padding schemes that enable us to achieve IND-CCA security, but they are not so widely deployed, e.g. RSA-OAEP.
- Why?

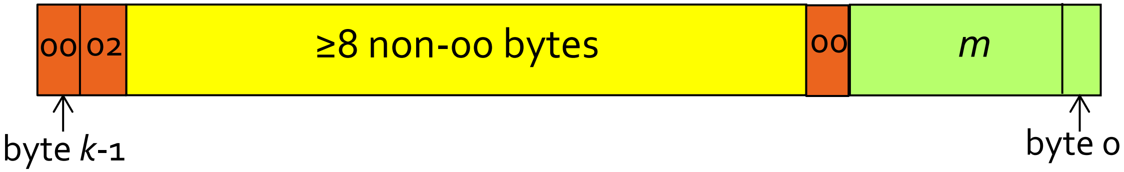
PKCS#1 v1.5 padding



- Byte-oriented encoding scheme for RSA.
- Assume *N* has *k* bytes; maximum message size is then *k* - 11 bytes.
- Message *m* placed in least significant bytes.
- Set first two most significant bytes to "00 02".
- Follow with at least 8 non-00 random bytes, then a 00 byte.
- Then apply the RSA operation to this encoded version of *m:*

$$c = pad(m)^e \bmod N.$$

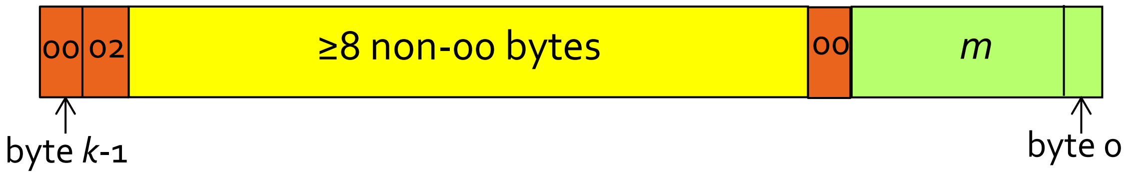
PKCS#1 v1.5 padding



Dec$_d$(c):

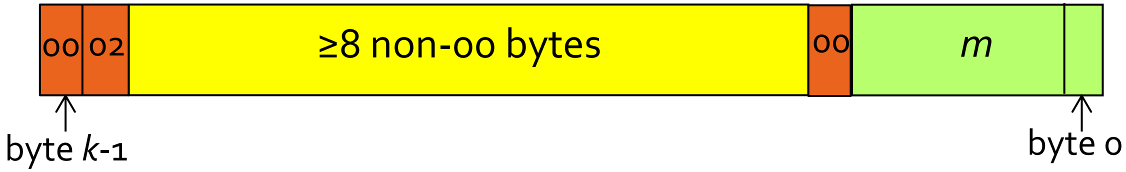
- Compute $m' = c^d \bmod N$.
- Check that m' begins with 00 02, reject if not.
- Check that m' then has at least 8 non-00 bytes, reject if not.
- Check that m' then has a 00 byte; reject if least significant byte is reached without finding one.
- Return as m all the bytes to the right of the 00 byte.
- Security?

Bleichenbacher’s attack on PKCS#1 v1.5



- If an attacker sends a random string of bytes for decryption, then it has probability about 2^{-16} of having a valid padding format:
  - the first two bytes must be “00 02” – prob. 2^{-16}.
  - there are then at least 8 non-00 bytes – prob. $(255/256)^8 = 0.97$.
  - there is at least one “00” byte before the LSB is reached – prob. depends on key-length, higher for longer keys.
- Suppose now that the attacker has an **oracle** telling it whether the decryption algorithm succeeds or fails on an input *c′*.
- As with padding oracles for CBC mode, such an oracle is common in implementations.

Bleichenbacher’s attack on PKCS#1 v1.5

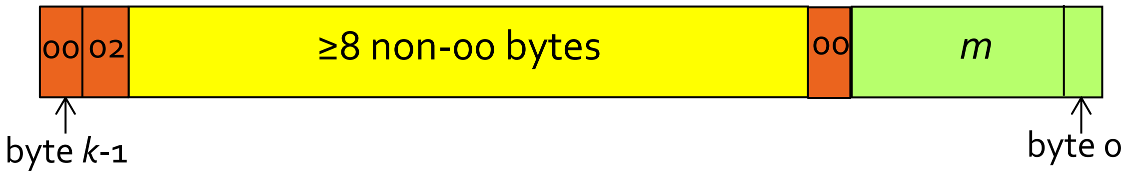


- Attacker wants to decrypt $c = (00\ 02 \| r \| 00 \| m)^e \bmod N$.
- Let’s write $pad(m) = (00\ 02 \| r \| 00 \| m)$ to denote the **padded** message.
- Attacker asks for decryption of $s^e \cdot c = (s \cdot pad(m))^e \bmod N$, for some choice of s.
- Most choices of s will lead to a decryption failure.
- With probability roughly 2^{-16}, decryption **succeeds**, and the adversary learns that $s \cdot pad(m) \bmod N$ begins with bytes “00 02”.
- This gives a range for $s \cdot pad(m) \bmod N$ over the integers:

$$s \cdot pad(m) \bmod N \in [2B, 3B\text{-}1] \text{ where } B = 2^{8(k-2)}$$

- **By gathering many such inequalities for different and carefully chosen values of *s* in an *adaptive* attack, attacker can recover *pad*(*m*), and hence *m*.**

Bleichenbacher's attack on PKCS#1 v1.5



- Bleichenbacher's attack requires many decryption attempts, around 2^{20} in its original version.
- The attack has been improved a lot; many variants have been discovered.
- Typically 5-10k queries are needed for a 1024-bit modulus N in modern versions.
- Many real-world systems, including SSL/TLS, have been found to be vulnerable to the style of attack.
- For example, DROWN and ROBOT attacks on SSL/TLS.
  - https://drownattack.com/
  - https://robotattack.org/
- RSA-PKCS#1 v1.5 should be avoided but is still widely used in practice.
- A similar attack shows that RSA-PKCS#1 v1.5 is not IND-CCA secure.

RSA-OAEP

- OAEP is an alternative encoding scheme for RSA due to Bellare and Rogaway. OAEP stands for "Optimal Asymmetric Encryption Padding".
- A version of it is widely standardized in PKCS#1 v2.1.

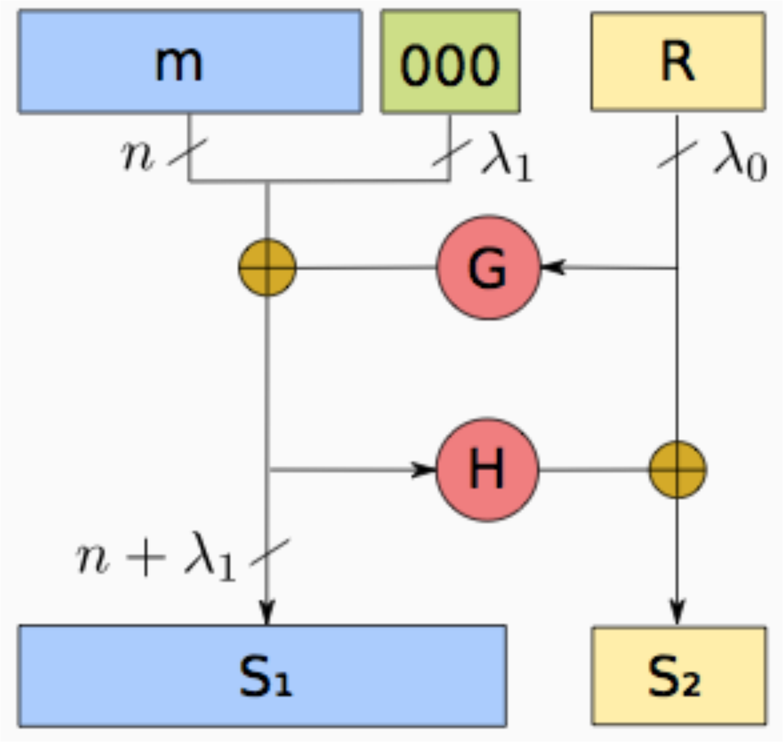
- **Setup of OAEP.** Suppose we are using λ-bit RSA moduli (e.g. $\lambda = 2048$).
- Let λ_0 and λ_1 be chosen so that no adversary can perform 2^λ operations for $\lambda = \lambda_0$ or $\lambda = \lambda_1$, e.g. $\lambda_0 = \lambda_1 = 128$.
- Set $n = \lambda - \lambda_0 - \lambda_1$.
- Messages in RSA-OAEP are assumed to be n-bit strings.
- Let G be a hash function mapping λ_0 bit strings to $n + \lambda_1$ bit strings.
- Let H be a hash function from $n + \lambda_1$ bit strings to λ_0 bit strings.
- These hash functions should be collision resistant.

RSA-OAEP

- Let *m* be an *n*-bit string.
- Choose a random λ_0-bit string *R*.
- Set $S_1 = (m \,||\, 0^{\lambda_1}) \oplus G(R)$.
- Set $S_2 = R \oplus H(S_1)$.
- Form the bit-string $S = S_1 \,||\, S_2$.
- Interpret *S* as a λ-bit integer and compute

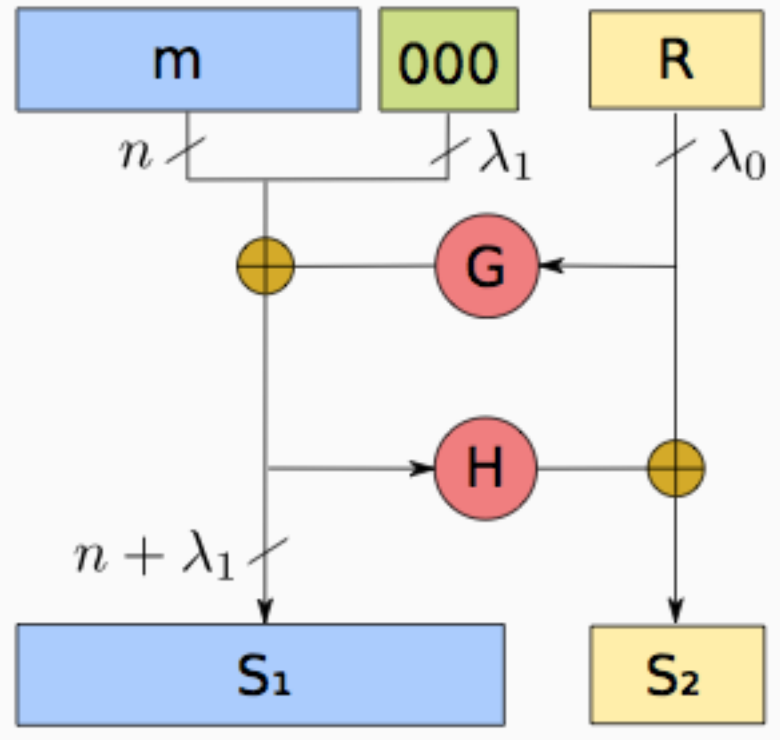
$$c = S^e \bmod N.$$

- Decryption reverses these steps and checks that the result has λ_1 0-bits in the right position.
- Decryption fails if the 0-bits are not present; outputs *m* if they are.



Intuition for RSA-OAEP design

- RSA messages are randomised and have full length.
- Since S_1 and S_2 are both outputs of hash functions, the bit-string S looks random and algebraic relationships among the message components are broken up.
- Suppose you are given a decryption oracle. Because of the way the hash functions are combined, it's hard to come up with a ciphertext that decrypts to produce λ_1 zero bits in the correct position, without having done an encryption in the first place; so the decryption oracle essentially becomes useless.



- RSA-OAEP can be proven to be IND-CCA secure, though with strong assumptions on G and H, and under a strong number theoretic assumption (much stronger than assuming that factoring is hard). It's not perfect, but:

Use RSA-OAEP!

RSA.KEM: Building a simple KEM from RSA

RSA.KGen: generates a pair of random primes p, q of some bit-size $k/2$, and integers d, e such that $de = 1 \bmod (p-1)(q-1)$. Set $N = pq$. Output key pair (sk,pk) where $sk = d$, $pk = (e,N)$. Let $H: \{0,\ldots,N-1\} \longrightarrow \{0,1\}^k$ be a hash function.

RSA.Encap: inputs public key $pk = (e,N)$; $s \leftarrow_\$ \{0,\ldots,N-1\}$; return (c,K) where

$$c = s^e \bmod N\,;\; K = H(s)$$

RSA.Decap: inputs private key $sk = d$ and encapsulation c.

compute $s = c^d \bmod N$;

output $K = H(s)$.

Theorem:

RSA.KEM is IND-CCA secure in the **Random Oracle Model** provided the **RSA inversion problem** is hard.

The RSA Inversion Problem

1. Challenger C runs RSA KeyGen to produce pair (sk,pk) with

$$sk = d,\ pk = (e,N).$$

2. C selects $x \leftarrow_{\$} \{0,\dots,N\text{-}1\}$; sets $y = x^e \bmod N$.

3. A is given input (N,e,y) by C; A runs and outputs some value x'.

A wins if $x' = x$.

- Strictly speaking, this version of the RSA Inversion problem is relative to the choice of KeyGen that we use.
- If A can factor N then it can solve the RSA inversion problem.
- The reverse implication is open, but no algorithm faster than factoring N is known for solving RSA inversion in general.
- For analysis of RSA.KEM, see extra slides.
- RSA.KEM is a much better scheme than RSA-OAEP, but is not widely used in practice.

Introducing the Discrete Logarithm Setting

The discrete logarithm setting

- So far, all of our public key schemes have been in the RSA setting:
  - Public/verification keys involve integers N that are the product of two primes, p,q, along with an integer e.
  - Private/signing keys are integers d such that $de = 1 \bmod (p-1)(q-1)$.
  - Security depends on hardness of factoring and related problems (RSA inversion).
- The discrete logarithm setting provides another platform for carrying out public key cryptography.
  - After introducing this setting, we'll look at Diffie-Hellman key exchange, and finally public key encryption.

The discrete logarithm setting

- Assume p and q are large primes and q divides p-1.
- So we can write $p = kq + 1$ for some integer k; so $k = (p-1)/q$.
- (So p and q do **not** have the same flavour as in RSA!).
- Often, but not always, $k=2$.

Toy example: $p = 29$, $q = 7$, $k = 4$.

- In reality, for 128-bit security, p will need to have 3072 bits and q will need to have (at least) 256 bits.
- Typical real-world deployments: p has 1024 or 2048 bits, q has (at least) 160 bits.

The discrete logarithm setting

- Now we pick h, a random integer mod p, and compute $g = h^{(p-1)/q}$ mod p (= h^k mod p).
  - If $g = 1$ mod p, we try again (we will succeed with very high probability)
- Fact 1: if $g \neq 1$ mod p, then the q powers of g, namely $G_q = \{g, g^2, g^3, \ldots g^q\}$ are all distinct mod p.
- Fact 2: $g^q = 1$ mod p.
- Fact 3: if we multiply together two elements in the set G_q, we obtain a third element that is also in the set G_q.
- In combination, these facts mean that the set G_q forms a group of order q; the group operation is multiplication mod p.
  - G_q is a cyclic group (everything is a power of g).
  - We say that g is a generator of G_q.
  - The number of elements in G_q is q, a prime.

The discrete logarithm setting

Toy example (ctd):

- $p = 29$, $q = 7$, $k = 4$ ($27 = 4.7+1$).
- $h = 2$; $g = h^{(p-1)/q} = 2^4 = 16 \bmod 29$.
- $G_q = \{g, g^2, g^3, \ldots g^q\}$:
  - So $g^2 = 16^2 = 256 = 24 \bmod 29$; and:
  - $g^3 = 16^3 = 16^2 \text{ x } 16 = 24 \text{ x } 16 = 384 = 7 \bmod 29$;
  - $g^4 = 16^4 = 16^3 \text{ x } 16 = 7 \text{ x } 16 = 112 = 25 \bmod 29$;
  - $g^5 = 16^5 = 16^4 \text{ x } 16 = 25 \text{ x } 16 = 400 = 23 \bmod 29$;
  - $g^6 = 16^6 = 16^5 \text{ x } 16 = 23 \text{ x } 16 = 368 = 20 \bmod 29$;
  - $g^7 = 16^7 = 16^6 \text{ x } 16 = 20 \text{ x } 16 = 320 = 1 \bmod 29$;
- Hence $G_q = \{16, 24, 7, 25, 23, 20, 1\}$.
- And, for example, $24 \text{ x } 7 = g^2 \text{ x } g^3 = g^5 = 23 \bmod 29$.

The discrete logarithm problem (DLP)

- The set $G_q = \{g, g^2, g^3, \ldots g^q\}$ of powers of g mod p is a cyclic group of prime order q with generator g.
- It forms a subgroup of the integers mod p under multiplication.
- We can write $g^q = 1 = g^0$ and so take $G_q = \{1 = g^0, g^1, g^2, g^3, \ldots g^{q-1}\}$.

The discrete logarithm problem in G_q:

Let (p, q, g) be as above. Set $y = g^x$ mod p, where x is a uniformly random value in $\{0,1,\ldots,q-1\}$. **Find x.**

(Think of value x as being the logarithm of $y = g^x$ mod p when the base of logarithms is g.)

Solving the DLP

- The DLP has received intense analysis from mathematicians and computer scientists for nearly 40 years.
  - Several different algorithms exist, but they are beyond the scope of this course.
  - This is an active area of research, with recent breakthroughs in related settings.
- The Function Field Sieve has complexity of the form*:*

$$\exp[(1+o(1)).(32/9)^{1/3} (\log p)^{1/3}(\log\log p)^{2/3}]$$

which is sub-exponential (but super-polynomial) in log p.

- The Pollard-ρ algorithm and its variants has complexity of the form: $O(q^{1/2})$, which is exponential in log q.
- To achieve 80-bits of security against both algorithms, we need q to have 160 bits and p to have 1024 bits.
- To achieve 128-bits of security against both algorithms, we need q to have 256 bits and p to have about 3072 bits.

Cryptography in the discrete logarithm setting

- Basic message so far: we can choose p and q big enough so as to make the DLP in G_q sufficiently hard.
- But we have to keep an eye on developments in algorithms for solving the DLP.
- Similar to RSA setting: what if someone comes up with a better factoring algorithm? Or a large quantum computer?

- Question now is: can we use the DLP to build cryptosystems (in particular, encryption schemes)?
- We will look at Diffie-Hellman, then ElGamal.

Diffie-Hellman Key Exchange

Diffie-Hellman Key Exchange (DHKE)

- DHKE is a public key method for agreeing on a shared secret (which can be used as a session key, for example).
- Introduced in famous 1976 paper by Diffie and Hellman that launched public key cryptography.
    - https://www-ee.stanford.edu/~hellman/publications/24.pdf

- We will use the Discrete Logarithm setting.
- Recall: $p = k.q + 1$, and g generates G_q, a cyclic group of prime order q in the set of integers modulo p:

$$G_q = \{g^0 = 1, g^1, g^2, \ldots, g^{q-1}\}.$$

Diffie-Hellman Key Exchange

- In Diffie and Hellman's original presentation, we have a collection of n users.
- All users make use of the same set of public parameters (p,q,g).
- User U_i picks x_i uniformly at random from {0,1,…,q-1}.
- User U_i's private key is x_i; its public key is $Y\_i = g^{x\_i} \bmod p$.
- All the public keys Y_i,…., Y_n are assumed to be available in a public, authentic directory.
    - In the 1970s, telephone directories were common!

Diffie-Hellman Key Exchange

To compute a shared value between user U_i and user U_j:

- User U_i computes $K\_ij = (Y\_j)^{x\_i} = g^{x\_j \cdot x\_i} = g^{x\_i \cdot x\_j} \bmod p;$
- User U_j computes $K\_ji = (Y\_i)^{x\_j} = g^{x\_i \cdot x\_j} \bmod p.$
- Key point: the values K_ij and K_ji are the same.
- We should not use K_ij directly as a key since it is large integer mod p.
- Instead, we should derive key(s) from it using a Key Derivation Function (KDF).
- For now, think of KDF as just being a hash function.

Diffie-Hellman Key Exchange

- Notice that no interaction is needed between any pair of users in order to establish a key.
- Instead, the users are assumed to have access to the public, authentic database of public keys $Y_1, \ldots, Y_n$.
- We have **non-interactive key exchange (NIKE).**

- **Question 1:** How is NIKE different from PKE?
- **Question 2:** Can we get from NIKE to PKE?

Security of Diffie-Hellman Key Exchange

- An attacker can see all public keys Y_1,..., Y_n and would like to compute some (or all) of the shared values K_ij (or the keys derived from them).
- The underlying algorithmic problem can be stated as follows:

> Given (p,q,g), and the values g^a mod p, g^b mod p, for uniformly random a, b, find g^{ab} mod p.

- This is the Computational Diffie-Hellman Problem (CDHP).

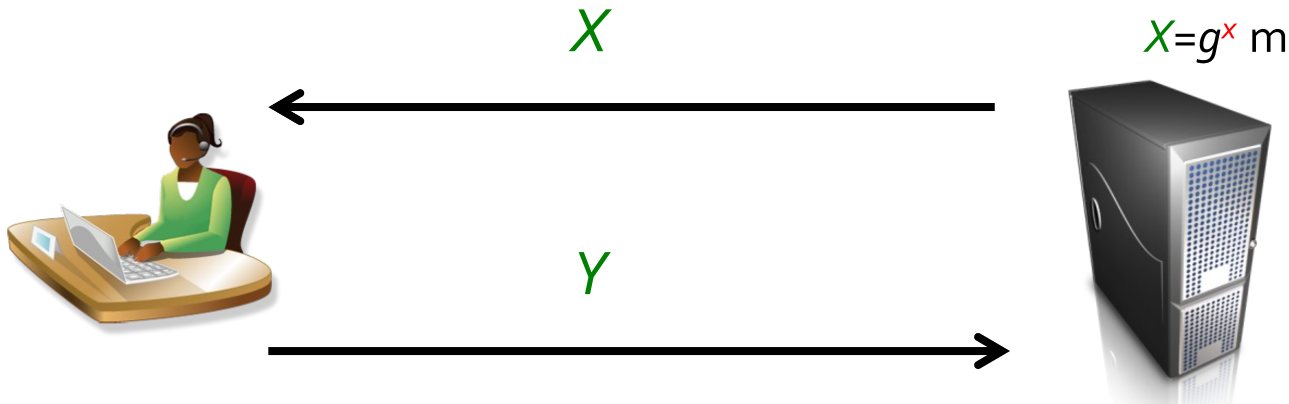
Security of Diffie-Hellman Key Exchange

- The Computational Diffie-Hellman Problem (CDHP).

  Given (p,q,g), and the values g^a mod p, g^b mod p, for uniformly random a, b, find g^{ab} mod p.

- If we can efficiently solve the DLP, then we can efficiently solve CDHP: find a from g^a mod p by solving DLP, then compute $(g^b)^a$ mod p.
- So CDHP is not harder than DLP, and could be easier.
- We don't know in general whether CDHP is equivalent to DLP, but this is widely believed to be so, and known to be so in special cases (den Boer 1988, Maurer-Wolf 1999).
- This helps us choose secure parameters for Diffie-Hellman: just assume CDHP is as hard as DLP.

A modern view of Diffie-Hellman Key Exchange



Pick random y;
$Y = g^{y} \bmod p$

Pick random x;
$X = g^{x} \bmod p$

$K = (X)^{y} \bmod p$

$K = (Y)^{x} \bmod p$

A modern view of Diffie-Hellman Key Exchange

- We simplify to just two parties, Alice and Bob.
- Alice and Bob first agree on parameters g, p and q, select fresh random private values x, y, then exchange the corresponding values X, Y over a public communications channel.
- We regard the private values x, y and the public values X, Y as being *ephemeral.*
  - That is, they are generated once and used only once, setting up a fresh shared value $K = g^{xy} \bmod p$ each time.
- It now seems that any two parties can securely agree upon a key without having had any previous association!

MITM Attack on Diffie-Hellman Key Exchange

But an active man-in-the-middle attacker can modify the public values in transit.

- For example, the attacker can change X to a value g^z mod p for which he knows z.
- The attacker can then compute the key that Alice would compute on receipt of g^z mod p, namely, $(g^z)^{y\_} = (g^y)^z = (Y)^z$ mod p.
- And similarly in the other direction.
- So an active attacker can completely compromise the security of “modern” Diffie-Hellman key exchange.
- Was that the case for the NIKE version?

Preventing MITM Attack on Diffie-Hellman Key Exchange

- So the modern version requires the authenticity of the public values X , Y to be protected in transit.
- How can this be done?
- Two possible solutions: use a MAC or use digital signatures.

Q: If we had a key for a MAC, why do we need to agree a new key via Diffie-Hellman?

A: Forward security: even if the MAC key was to be revealed at some point in time, the keys from Diffie-Hellman exchanges completed before that time would still be secure (more in later lectures).

Q: What additional requirements does the use of signatures bring?

A: We now need an assurance of the authenticity of the signature verification keys. This is typically done using certificates, CAs, and PKI.

From Diffie-Hellman to ElGamal Encryption

From Diffie-Hellman Key Exchange to Public Key Encryption

- Diffie-Hellman paper appeared in 1976.
- Introduced the *concept* of public key encryption, but contained no concrete example of a public key encryption scheme.
- Included Diffie-Hellman key exchange (with a public database).
- So can we get from DHKE to PKE?
- Yes we can!
- We next define the ElGamal public key encryption scheme, dating from 1985.

ElGamal Public Encryption Scheme

Public parameters (p,q,g) as usual (can be shared amongst many users).

KeyGen: pick x uniformly at random from $\{0,1,\ldots,q\text{-}1\}$. Set public key to be $X = g^x \bmod p$ and private key to be x.

Enc: given public key X and message M (assumed to be encoded as an element of G_q):

1. Pick r uniformly at random from $\{0,1,\ldots,q\text{-}1\}$. Set $Y = g^r \bmod p$.
2. Compute $Z = X^r \bmod p$.
3. Output the ciphertext $C = (Y, M \cdot Z \bmod p)$.

ElGamal Public Encryption Scheme

Dec: given private key x and ciphertext $C = (Y, C')$:

1. Check that Y is in G_q, return "fail" if not.
2. Compute $Z' = Y^x \bmod p$.
3. Output $M = C' \cdot (Z')^{-1} \bmod p$.

Correctness of the scheme:

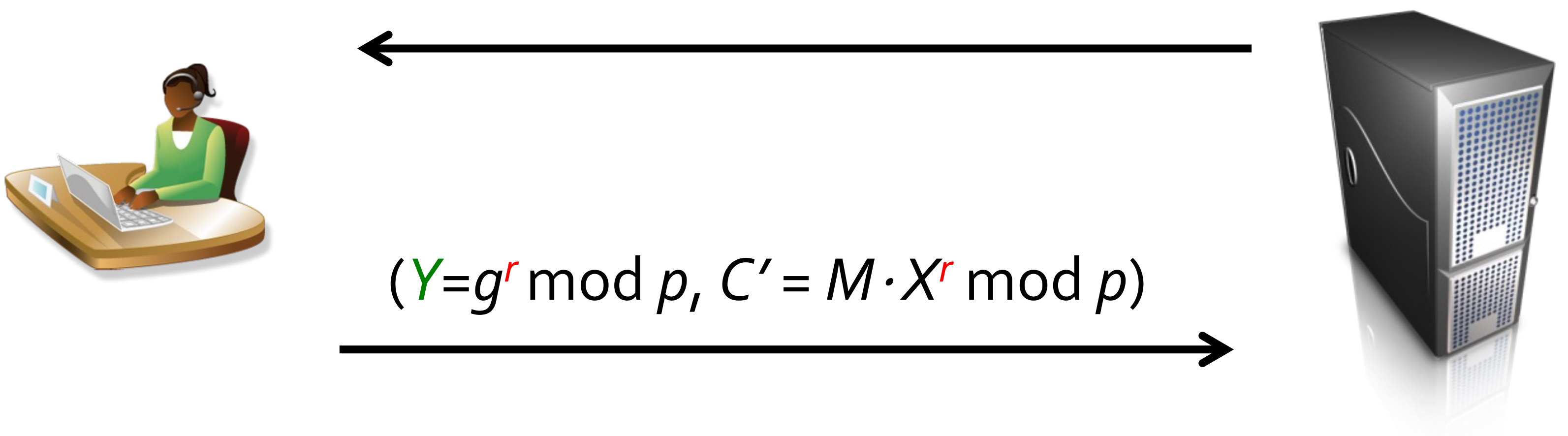
$$C' \cdot (Z')^{-1} = M \cdot Z \cdot (Z')^{-1} = M \cdot X^r \cdot (Y^x)^{-1} \bmod p$$

$$= M \cdot (g^x)^r \cdot ((g^r)^x)^{-1} \bmod p$$

$$= M \cdot g^{xr} \cdot g^{-xr} \bmod p$$

$$= M$$

Relationship Between Diffie-Hellman and ElGamal



Public key: $(p,q,g,X=g^x)$

$(Y=g^r \bmod p,\ C' = M \cdot X^r \bmod p)$

- We effectively have a Diffie-Hellman key exchange, involving long-term key pair (x,X) and one-time key pair (r,Y), with the sender choosing (r,Y) and including the public value Y as part of the ciphertext.
- We use the shared Diffie-Hellman value $X^r = g^{xr} \bmod p$ as an encryption mask.
- The encrypting party need not have a "registered" public key to use the scheme (as should be the case for PKE!).

Security of ElGamal Public Encryption Scheme

IND-CPA security for ElGamal encryption can be proven based on the hardness of a variant of the CDH Problem, called the Decisional Diffie-Hellman Problem:

> Given (p,q,g), and uniformly random values a,b,c from $\{0,1,\ldots,q\text{-}1\}$, distinguish the triple (g^a, g^b, g^{ab}) from the triple (g^a, g^b, g^c).

Informally:

- Ciphertext includes $M \cdot Z \bmod p$ where $Z = g^{xr} \bmod p$.
- If (g^x, g^r, g^{xr}) is indistinguishable from (g^x, g^r, g^c), then we can replace $Z = g^{xr}$ by $Z = g^c$ in the construction of ciphertext.
- But, since g^c is uniformly random in G_q, it follows that $M \cdot Z \bmod p$ is uniformly random in G_q as well.
- Hence M is perfectly hidden!

Formally: do a game hopping proof with G_0 the normal IND-CPA game and G_1 the game in which the challenge ciphertext uses g^c in place of g^{xr}.

DHIES

DHIES

ElGamal PKE has two significant disadvantages:

- It is not IND-CCA secure (find an attack exploiting the homomorphic nature of ElGamal encryption!)
- It is inconvenient to work with messages that have to be encoded as elements of G_q.
- DHIES (Diffie-Hellman Integrated Encryption Scheme) addresses both of these problems in a PKE scheme.
- One can also define a KEM based on ElGamal and use the KEM/DEM paradigm.

DHIES

Public parameters (p,q,g) as usual, H a hash function with suitable output domain.

KeyGen: pick x uniformly at random from $\{0,1,\ldots,q\text{-}1\}$. Set public key to be $X = g^x \bmod p$ and private key to be x.

Enc: given public key X and message M (a bit-string):

1. Pick r uniformly at random from $\{0,1,\ldots,q\text{-}1\}$. Set $Y = g^r \bmod p$.
2. Compute $Z = X^r \bmod p$.
3. Set $K = H(Z,X,Y)$.
4. Split K into K_e and K_m.
5. Compute $C' =$ EtM(M) using keys K_e and K_m for encryption and MAC, respectively.
6. Output the ciphertext $C = (Y, C')$.

DHIES

Dec: given private key x and ciphertext C = (Y, C'):

1. Check that Y is in G_q, return "fail" if not.
2. Compute $Z = Y^x \bmod p$.
3. Set $K = H(Z,X,Y)$.
4. Split K into K_e and K_m.
5. Decrypt C' using keys K_e and K_m for encryption and MAC, respectively.
6. Output "fail" if step 5 fails, otherwise output the message returned in step 5.

Correctness again relies on: $X^r = Z = Y^x \bmod p$, that is, that we implicitly have a Diffie-Hellman key exchange.

DHIES

- Checking that Y is in G_q is essential for security!
- This is done by checking if $Y^q = 1 \bmod p$: a somewhat expensive check to do.
- But failure to check can lead to *small subgroup attacks.*
- Idea of the attack is that an adversary who can force the use of g of small order t (instead of large order q) may be able to learn x mod t by seeing if decryption fails or not.
- Attack requires t trials on special ciphertexts (in the worst case) to recover x mod t.
- Attack requires p-1 to be divisible by t.
- If adversary can do this for enough different, small, coprime values t then it can even reconstruct x (using the CRT).

DHIES

- DHIES is IND-CCA secure in the Random Oracle Model.
- The security proof relies on the hardness of a variant of the Computational Diffie-Hellman Problem (assuming we use an IND-CPA symmetric encryption scheme and a SUF-CMA MAC scheme).
- It’s really an instance of the KEM/DEM paradigm: we are sending information Y that allows a shared key K to be computed, and then using that key K in a symmetric encryption scheme.
- We could replace “EtM” with any AE scheme.
- Can go nonce-based, use AEAD, etc.

From Diffie-Hellman Key Exchange to Public Key Encryption

- An elliptic-curve variant of DHIES also exists, called ECIES.
  - It has smaller ciphertext sizes, because of the more compact representation of group elements (g^x) in the elliptic curve setting.
  - Security relies on the hardness of the Elliptic Curve Diffie-Hellman (ECDH) Problem.
- These schemes are standardised in ANSI X9.63, IEEE P1363a, …
- ECIES will be used in the next generation of EMV "chip and PIN" card standards.
- We'll return to ECIES when we look at elliptic curve cryptography.

Homework

- Read Chapters 10-12 of Boneh-Shoup for many more details, constructions and proofs.
- Work on exercise sheets 8, 9.
- Prepare for this week's lab.

Appendix: Some Basic Number Theory

Basic Number Theory: Remainders, factors,…

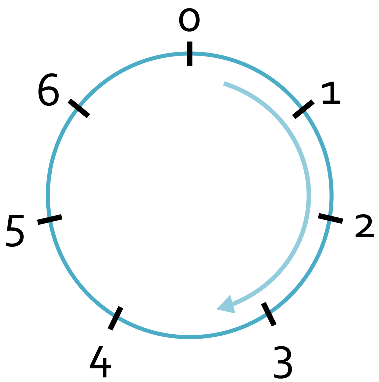
- We use the notation **N** to denote the non-negative integers:

  N = {0,1,2,…}.

- We write a mod b for the "remainder of a on division by b", i.e. the integer r in $\{0,\ldots,b-1\}$ such that $a = q \cdot b + r$ for some q.
- For example,
  - 9 mod 7 = 2,because $9 = 1 \cdot 7 + 2$;
  - 21 mod 7 = 0, because $21 = 3 \cdot 7 + 0$.
- We say "a is a factor of b" if a divides b with remainder 0.
- For example,
  - 3 is a factor of 6, but
  - 4 is not a factor of 6.

Basic Number Theory: Modular arithmetic

- Modular arithmetic refers to arithmetic done with the numbers in $\{0,\ldots,b-1\}$ (for some b), where we always reduce our results to remainders on division by b.



- For example, with $b = 7$, we have:
  - $6 + 4 = 10 = 3$ mod 7.
  - $6 \cdot 4 = 24 = 7{\cdot}3 + 3 = 3$ mod 7.
  - $6^4 = 6 \cdot 6 \cdot 6 \cdot 6 = (6^2)^2 = 36^2 = 1^2 = 1$ mod 7.

- Important principle: we never need to work with numbers much bigger than b if we are doing arithmetic mod b: we can just reduce mod b as we go along.

- Note that if b is composite, then we can multiply two non-zero numbers mod b and get zero!

  - For example, with $b = 12$ we have $6{\cdot}4 = 24 = 2{\cdot}12 + 0 = 0$ mod 12.

- This is different from normal arithmetic….

Basic Number Theory: Primes and gcds

- A number greater than 1 with only itself and 1 as factors is called a prime number. For example, 13 is prime, but 21 is composite (= not prime).
- The greatest common divisor (gcd) of two numbers is the largest number that is a factor of both numbers. For example, gcd(4,6) = 2.
- Two numbers are said to be relatively prime if their gcd is 1.

Basic Number Theory: Modular inverses

- Over the rational numbers we define the (multiplicative) inverse of a, denoted a^{-1} as the number such that $a \cdot a^{-1} = 1.$
- Modulo some number p, we can also define (modular) inverses.
  - For example, $5 \cdot 5$ mod 6= 25 mod 6 = 1 mod 6.
  - Hence 5^{-1} = 5 mod 6.
- All numbers $0 < a < p$ such that gcd(a,p) = 1 have an inverse modulo p.
- But if $0 < a < p$ is such that gcd(a,p) > 1, then a will not have an inverse modulo p.
- If p is a prime, then gcd(a,p) = 1 for all $0 < a < p$ (why?), and so every number a in {1,…,p-1} will have an inverse mod p.
- The numbers modulo a prime p form a field, which means that addition, multiplication and division are defined and work for all inputs "as expected".

Basic Number Theory

- As an example, let's make the addition and multiplication tables mod 5.

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

·	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

- Notice the pattern along rows and columns of addition table.
- Notice how every element has a multiplicative inverse except 0.
- 1 and 4 are self-inverse.
- As an exercise, make the same tables for modulus 6.

Basic Number Theory: The Euclidean algorithm

- Given a, $b \in \mathbf{N}$ we want to compute $\gcd(a,b)$.
- Euclid's algorithm is based on recursive application of the formulae:

$$\gcd(a,b) = \gcd(b, a \bmod b) \qquad (1)$$

$$\gcd(a,0) = a \qquad (2)$$

- Notice that $0 \le a \bmod b < b$, so the numbers decrease in size at every step.
- Example:

$$\begin{aligned} & \gcd(21, 14) \\ &= \gcd(14, 21 \bmod 14) && \text{(using (1))} \\ &= \gcd(14, 7) \\ &= \gcd(7, 14 \bmod 7) && \text{(using (1))} \\ &= \gcd(7,0) \\ &= 7. && \text{(using (2))} \end{aligned}$$

Basic Number Theory: The extended Euclidean algorithm

- The extended Euclidean algorithm outputs integers r, s, t where $r = \gcd(a,b)$ and $s \cdot a + t \cdot b = r$.
- In other words, it gives an expression for $\gcd(a,b)$ as a linear combination of a and b over the integers.
- It works like the Euclidean algorithm but keeps track of additional information as we go along.
- It computes a series of coefficients r_i, s_i, t_i such that:

$$r_i = s_i \cdot a + t_i \cdot b.$$

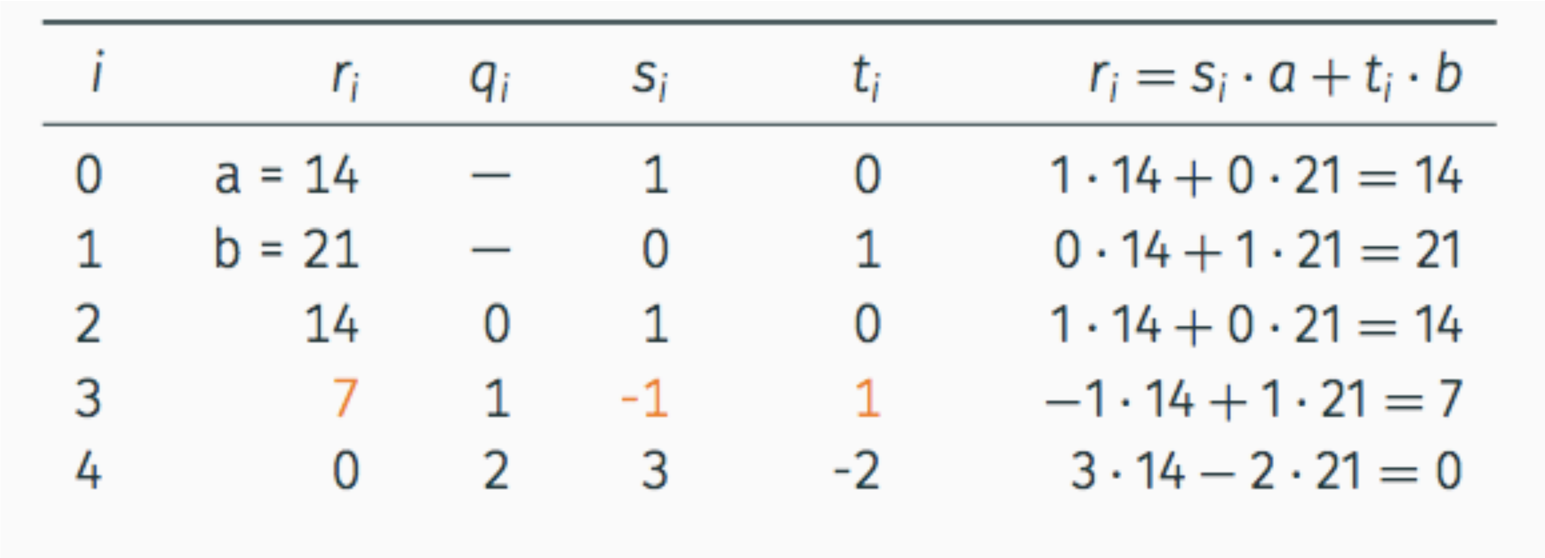
- In each step we compute

$$r_{i+1} = r_{i-1} - q_i \cdot r_i \qquad \text{where} \quad 0 \le r_{i+1} < |r_i|$$
$$s_{i+1} = s_{i-1} - q_i \cdot s_i$$
$$t_{i+1} = t_{i-1} - q_i \cdot t_i$$

- We initialise the algorithm with $r_0 = a$, $r_1 = b$ and $s_0 = 1$, $s_1 = 0$, $t_0 = 0$, $t_1 = 1$.
- Eventually $r_k = \gcd(a,b)$ and $\gcd(a,b) = s_k \cdot a + t_k \cdot b$.

Basic Number Theory: The extended Euclidean algorithm

- The extended Euclidean algorithm outputs integers r, s, t where $r = \gcd(a,b)$ and $s \cdot a + t \cdot b = r$.
- Example:



i	r_i	q_i	s_i	t_i	$r_i = s_i \cdot a + t_i \cdot b$
0	a = 14	—	1	0	$1 \cdot 14 + 0 \cdot 21 = 14$
1	b = 21	—	0	1	$0 \cdot 14 + 1 \cdot 21 = 21$
2	14	0	1	0	$1 \cdot 14 + 0 \cdot 21 = 14$
3	7	1	-1	1	$-1 \cdot 14 + 1 \cdot 21 = 7$
4	0	2	3	-2	$3 \cdot 14 - 2 \cdot 21 = 0$

Using the extended Euclidean algorithm

- The numbers s and t are often useful for applications.
- For example, running the extended Euclid's algorithm on a with $0 < a < p$ and a prime p gives s and t such that:

$$a \cdot s + p \cdot t = 1.$$

- Reducing this equation mod p, we get:

$$a \cdot s = 1 \text{ mod } p.$$

- So Euclid's algorithm allows us to find the inverse of a mod p.
- Recall that in RSA key generation, we need to a pair (e,d) such that:

$$e \cdot d = 1 \text{ mod } (p\text{-}1)(q\text{-}1).$$

- This can also be done using the extended Euclidean algorithm:
    - Choose e at random in the range $[1, (p\text{-}1)(q\text{-}1)]$.
    - With high probability, $\gcd(e, (p\text{-}1)(q\text{-}1)) = 1$ and then running Euclid's algorithm gives us the required d.

Correctness of textbook RSA

Let $N = pq$ and suppose $ed = 1 \bmod (p-1)(q-1)$.

Hence $(p-1)(q-1) \mid ed - 1$.

So write $ed = 1 + k(p-1)(q-1)$.

Let m be coprime to N. Then m is coprime to p and q.

By Fermat's little theorem, $m^{p-1} = 1 \bmod p$.

Then $m^{ed} = m^{1 + k(p-1)(q-1)} = m \cdot (m^{p-1})^{k(q-1)} = m \cdot 1^{k(q-1)} = m \bmod p$.

Hence p divides $m^{ed} - m$.

By symmetry q also divides $m^{ed} - m$. Then $N=pq$ also divides $m^{ed} - m$.

Hence $m^{ed} = m \bmod N$.

This completes the correctness proof.

(What about the case where m is not coprime to N?)

Appendix: Building a Simple KEM from RSA

Building a simple KEM from RSA

RSA.KGen: generates a pair of random primes p, q of some bit-size $k/2$, and integers d, e such that $de = 1 \bmod (p-1)(q-1)$. Set $N = pq$. Output key pair (sk,pk) where $sk = d$, $pk = (e,N)$. Let $H: \{0,\ldots,N-1\} \longrightarrow \{0,1\}^k$ be a hash function.

RSA.Encap: inputs public key $pk = (e,N)$; $s \leftarrow_{\$} \{0,\ldots,N-1\}$; return (c,K) where

$c = s^e \bmod N$; $K = H(s)$

RSA.Decap: inputs private key $sk = d$ and encapsulation c.

compute $s = c^d \bmod N$;

output $K = H(s)$.

Theorem:

The above scheme RSA.KEM is IND-CCA secure in the **Random Oracle Model** provided the **RSA inversion problem** is hard.

The RSA Inversion Problem

1. Challenger C runs RSA KeyGen to produce pair (sk,pk) with

$$sk = d,\ pk = (e,N).$$

1. C selects $x \leftarrow_{\$} \{0,\ldots,N\text{-}1\}$; sets $y = x^e \bmod N$.
2. A is given input (N,e,y) by C; A runs and outputs some value x'.
3. A wins if $x' = x$.

- NB $x = y^d = y^{1/e} \bmod N$, so x is the e-th root of y mod N.
- Strictly speaking, this version of the RSA Inversion problem is relative to the choice of KeyGen that we use.
- If A can factor N then it can solve the RSA inversion problem.
- The reverse implication is open, but no algorithm faster than factoring N is known for solving RSA inversion in general.

The Random Oracle Model (ROM)

- The ROM is a strong abstraction of hash functions.
- Formalized by Bellare and Rogaway (1993).
- In short, in the ROM, we model a hash function *H* as a *random* function (from a given domain to a given range).
- An adversary *A* cannot then compute *H* for itself but must instead "outsource" all its computations of *H* to its challenger in the form of oracle queries.
- This gives a security reduction in a proof the power to "inspect" queries made by *A*, possibly extracting useful information from them.
- It also allows the reduction to "program" specific values into its *H*-oracle replies.
- We can also argue that any output *H*(*x*) is uniformly random unless and until *A* queries *H* on input *x.*
- We will see this in action in our analysis of RSA.KEM ahead.

The Random Oracle Model (ROM)

- Of course, real hash functions like SHA-256 are fixed functions and not random oracles.
- So we must **instantiate** the random oracle somehow in order to implement any scheme whose analysis use the ROM.
- **Heuristic step**: replace the random oracle with a fixed hash function.
- There are many arguments for and against using the ROM in security proofs.
- Pro: it's a useful tool, especially in the public key setting.
- Con: it's not formally sound.
- Pro: none of the examples showing this are very natural.
- Pro: it enables more efficient schemes (in comparison to the *standard* model).
- A lot of research effort has gone into closing the gap between standard-model-secure schemes and ROM-secure schemes.

Building a simple KEM from RSA (recap)

RSA.KGen: generates a pair of random primes p, q of some bit-size $k/2$, and integers d, e such that $de = 1 \bmod (p-1)(q-1)$. Set $N = pq$. Output key pair (sk,pk) where $sk = d$, $pk = (e,N)$. Let $H\colon \{0,\dots,N-1\} \longrightarrow \{0,1\}^k$ be a hash function.

RSA.Encap: inputs public key $pk = (e,N)$; $s \leftarrow_{\$} \{0,\dots,N-1\}$; return (c,K) where

$$c = s^e \bmod N\,;\ K = H(s)$$

RSA.Decap: inputs private key $sk = d$ and encapsulation c.

compute $s = c^d \bmod N$;

output $K = H(s)$.

An RSA inversion instance!

Uniformly random unless adversary queries H on s.

Proof Intuition for simple KEM from RSA – 1

Claim:

RSA.KEM is IND-CCA secure in the ROM assuming the hardness of RSA inversion.

Proof – intuition for IND-**CPA** security

We show that from any IND-**CPA** adversary *A* against the KEM, we can build an adversary *B* solving the RSA inversion problem.

Given an RSA inversion challenge (N,e,y), *B* sets $pk = (N,e)$ and uses *y* to construct challenge encapsulation $c*$ in the KEM security game for *A*.

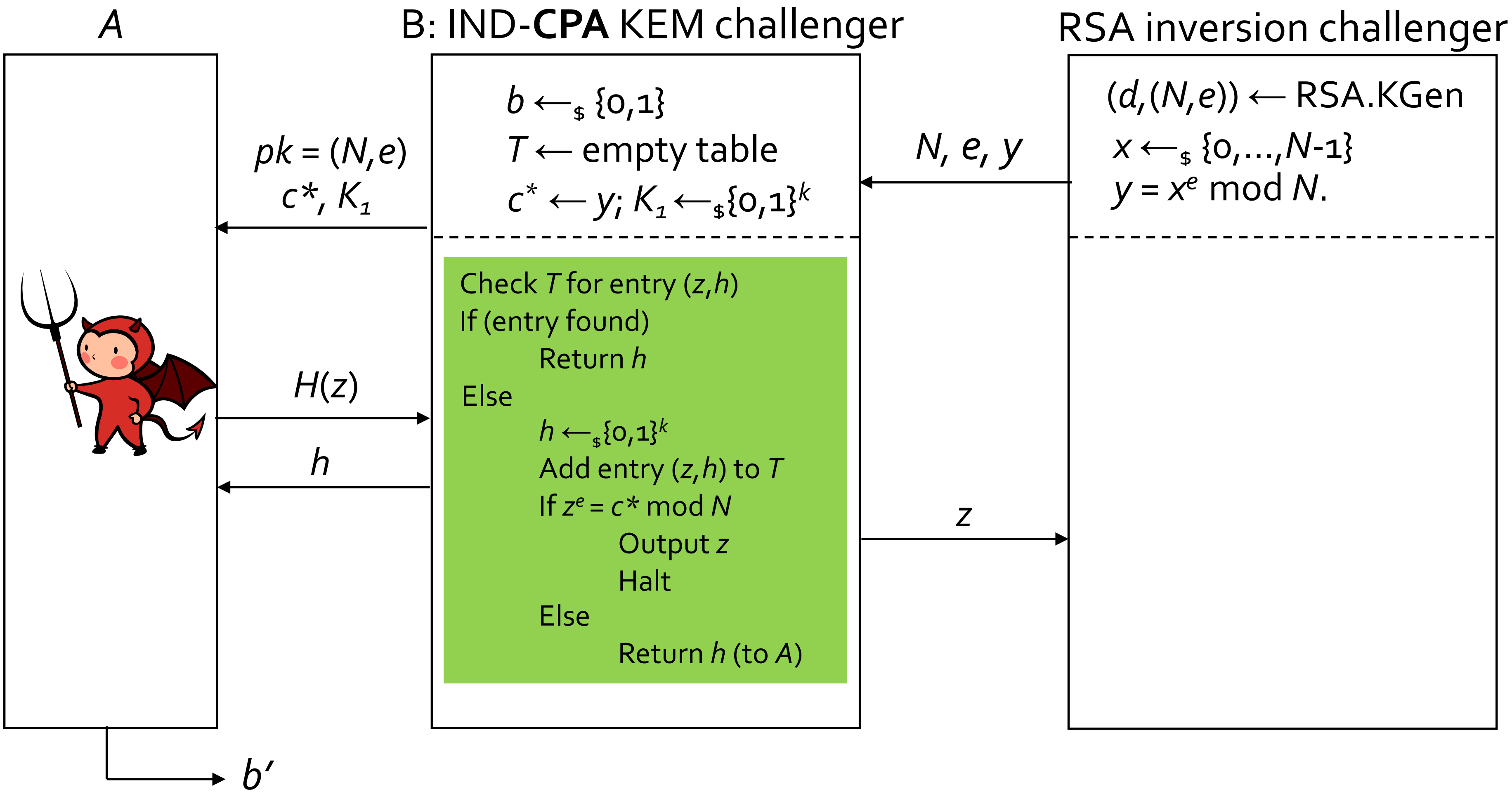
- Key K_0 then equals $H((c*)^d \bmod N) = H(y^d \bmod N) = H(y^{1/e} \bmod N) = H(x)$.
- Key K_1 is set to random by *B*.

Recall that *A*'s challenger (played by *B*) gives $(pk, c*, K_b)$ to *A*; to win in the KEM security game, *A* must be able to distinguish K_0 from K_1, i.e. distinguish $H(x)$ from random.

A can't do this unless it queries *H* on *x* at some point, since otherwise $H(x)$ is uniformly random and has an identical distribution to K_1.

If *A* does query *H* on *x*, then *B* can find *x* amongst the queries made by *A*; in fact, *A* can check each *H*-query *z* made by *A* and test if $z^e = y \bmod N$.

Sketch of reduction for IND-CPA security



A

B: IND-**CPA** KEM challenger

RSA inversion challenger

$(d,(N,e)) \leftarrow$ RSA.KGen
$x \leftarrow_\$ \{0,\ldots,N\text{-}1\}$
$y = x^e \bmod N.$

N, e, y

$b \leftarrow_\$ \{0,1\}$
$T \leftarrow$ empty table
$c^* \leftarrow y;\ K_1 \leftarrow_\$ \{0,1\}^k$

$pk = (N,e)$
c^*, K_1

$H(z)$

h

```
Check $T$ for entry $(z,h)$
If (entry found)
        Return $h$
Else
        $h \leftarrow_\$ \{0,1\}^k$
        Add entry $(z,h)$ to $T$
        If $z^e = c^* \bmod N$
                Output $z$
                Halt
        Else
                Return $h$ (to $A$)
```

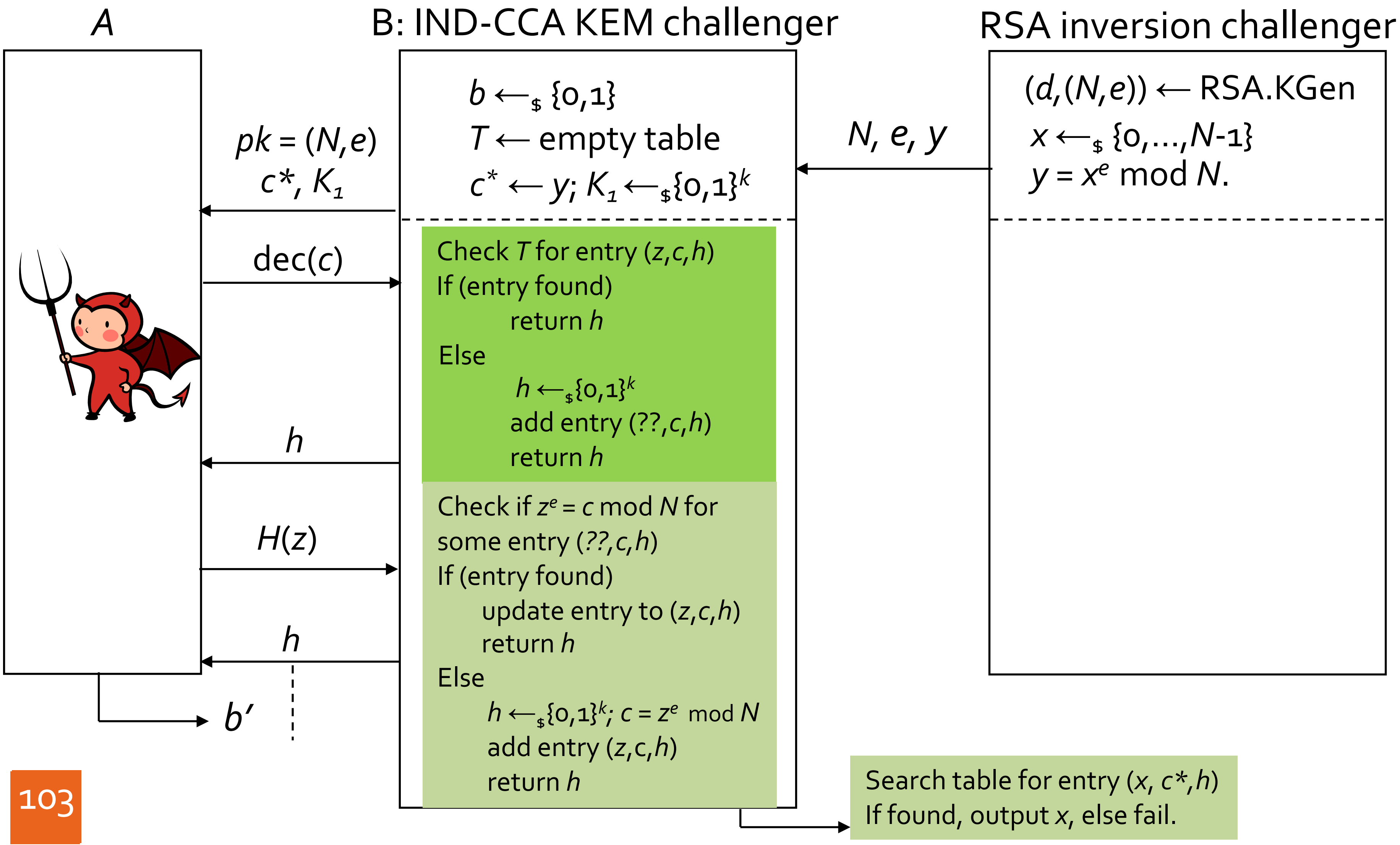
z

b'

Proof Intuition for simple KEM from RSA – 2

- What about IND-CCA security?
- The crucial point is that *B* must provide a consistent simulation of dec(·) and *H*(·) queries for *A*, at least until *x* is queried (after that, we don't care – the simulation can go wrong).
- *B* "patches the random oracle" using a table of triples of one of two forms: (*z*, *c*, *h*) or (??, c, *h*).
- **On dec(*c*) query:**
  - Check for a table entry (*z*,*c*,*h*); if one exists, respond with *h.*
  - Otherwise, choose random *h* and add entry (*??*,*c*,*h*) to the table; this implicitly defines *H*(*z*) = *h* where $z = c^{1/e}$ mod *N* is unknown (hence "??" in place of *z* in the table).
- **On *H* query with input *z*:**
  - Check if there is already an entry (*z*,*c*,*h*) in the table; if so, return *h* (basic consistency).
  - Check if $z^e = c$ mod *N* for some entry (*??*,*c*,*h*) in the table.
    - If yes, then complete table entry to (*z*,*c*,*h*); return *h* (maintain consistency with dec queries).
    - If not, select a random *h*, set $c = z^e$ mod *N* and add a new entry (*z*,*c*,*h*); return *h.*
- Working out all the details is a bit messy, so we omit them.
- *B*'s advantage is tightly related to that of *A* but its running time is higher (due to cost of maintaining table consistency).

Incomplete sketch of reduction



A

B: IND-CCA KEM challenger

RSA inversion challenger

```
$(d,(N,e)) \leftarrow$ RSA.KGen
$x \leftarrow_\$ \{0,\dots,N\text{-}1\}$
$y = x^e \bmod N.$
```

N, e, y

```
$b \leftarrow_\$ \{0,1\}$
$T \leftarrow$ empty table
$c^* \leftarrow y$; $K_1 \leftarrow_\$ \{0,1\}^k$
```

$pk = (N,e)$

c^*, K_1

dec(c)

```
Check $T$ for entry $(z,c,h)$
If (entry found)
        return $h$
Else
        $h \leftarrow_\$ \{0,1\}^k$
        add entry $(??,c,h)$
        return $h$
```

h

$H(z)$

```
Check if $z^e = c \bmod N$ for
some entry $(??,c,h)$
If (entry found)
     update entry to $(z,c,h)$
     return $h$
Else
     $h \leftarrow_\$ \{0,1\}^k$; $c = z^e \bmod N$
     add entry $(z,c,h)$
     return $h$
```

h

b'

Search table for entry (x, c^*, h)

If found, output x, else fail.