7 RSA

CRYPTO 101: Building Blocks

cryptography101.ca

V7 outline

- V7a: Basic RSA
- V7b: Integer factorization
- V7c: RSA encryption
- V7d: RSA signatures
- V7e: PKCS #1 v1.5 RSA signatures

V7a
Basic RSA

RSA

CRYPTO 101: Building Blocks

cryptography101.ca

RSA

- Invented by Ron Rivest, Adi Shamir and Len Adleman in 1977.
- RSA is used for public-key encryption and signatures.

Ron Rivest
CC BY-SA 4.0

Adi Shamir
Erik Tews
CC BY-SA 3.0

Len Adleman
CC BY-SA 3.0

A Method for Obtaining Digital Signatures and Public-Key Cryptosystems

R. L. Rivest, A. Shamir, and L. Adleman

RSA key generation

Each entity A does the following:

1. Randomly select two large, distinct primes p and q of the same bitlength.
2. Compute $n = pq$ and $\phi = \phi(n) = (p-1)(q-1)$.
   (n is called the RSA modulus)
3. Select arbitrary integer e, $1 < e < \phi$, with $\gcd(e, \phi) = 1$.
   (e is called the encryption exponent)
4. Compute the integer d, $1 < d < \phi$, with $ed \equiv 1 \pmod{\phi}$.
   ($d = e^{-1} \bmod \phi$ is called the decryption exponent)
5. A's public key is (n, e); her private key is d.

Basic RSA public-key encryption scheme

RSA encryption: To encrypt a message for A, B does the following:

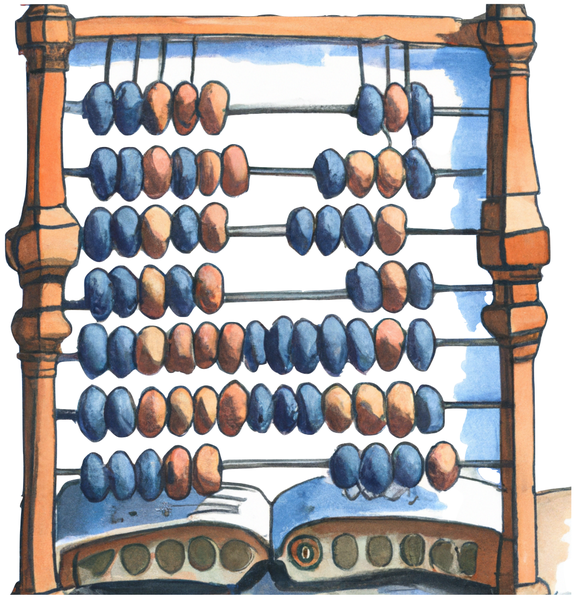
1. Obtain an authenticated copy of A's public key (n, e).
2. Represent the message as an integer $m \in [0, n-1]$.
3. Compute the ciphertext $c = m^e \bmod n$.
4. Send c to A.

RSA decryption: To decrypt c, A does the following:

1. Compute $m = c^d \bmod n$.

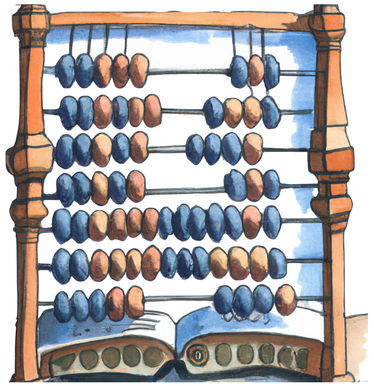
Toy example: RSA key generation

Alice does the following:



1. Selects primes $p = 23$ and $q = 37$.
2. Computes $n = pq = 851$
   and $\phi(n) = (p-1)(q-1) = 792$.
3. Selects $e = 631$ satisfying $\gcd(631,792) = 1$.
4. Solves $631d \equiv 1 \pmod{792}$ to get $d \equiv -305 \equiv 487 \pmod{792}$, and so sets $d = 487$.
5. Alice's public key is $(n = 851, e = 631)$; her private key is $d = 487$.

Toy example: RSA encryption



To encrypt a plaintext $m = 13$ for Alice, Bob does:

1. Obtains Alice's public key ($n = 851$, $e = 631$).
2. Computes $c = 13^{631} \bmod 851$ using repeated square-and-multiply:
   (a) Write $e = 631$ in binary: $e = 2^9 + 2^6 + 2^5 + 2^4 + 2^2 + 2^1 + 2^0$.
   (b) Compute successive squaring $(i, m^{2^i} \bmod n)$ of $m = 13$ modulo n:
   (0,13), (1,169), (2,478), (3,416), (4,303), (5,752), (6,440), (7,423), (8,219), (9,305).
   (c) Multiply together the squares m^{2^i} for which the ith bit of the binary representation of 631 is 1: $13^{631} \equiv 305 \cdot 440 \cdot 752 \cdot 303 \cdot 478 \cdot 169 \cdot 13 \equiv 616 \pmod{851}$.
3. Bob sends the ciphertext $c = 616$ to Alice.

To decrypt $c = 616$, Alice uses her private key $d = 487$ to compute $m = 616^{487} \bmod 851$. She gets $m = 13$.

RSA works

Theorem: For all $m \in [0, n-1]$, if $c = m^e \bmod n$, then $m = c^d \bmod n$.

Proof: We'll prove that $m^{ed} \equiv m \pmod{n}$ for all $m \in [0, n-1]$.

- Since $ed \equiv 1 \pmod{\phi}$, we can write $ed = 1 + k\phi = 1 + k(p-1)(q-1)$ for some $k \in \mathbb{Z}$. Since $ed > 1$ and $(p-1)(q-1) \geq 1$, we have $k \geq 1$.
- We'll now prove that $m^{ed} \equiv m \pmod{p}$.
- Suppose first that p divides m. Then $m \equiv 0 \pmod{p}$, so $m^{ed} \equiv 0^{ed} \equiv 0 \pmod{p}$. Thus, $m^{ed} \equiv m \pmod{p}$.
- Suppose now that p does not divide m. By Fermat's Little Theorem, we have $m^{p-1} \equiv 1 \pmod{p}$. Raising both sides to the power $k(q-1)$, and then multiplying by m, gives $m^{1+k(p-1)(q-1)} \equiv m \pmod{p}$. Thus, $m^{ed} \equiv m \pmod{p}$.
- So, we conclude that $m^{ed} \equiv m \pmod{p}$ for all $m \in [0, n-1]$.
- Similarly, $m^{ed} \equiv m \pmod{q}$. Since p and q both divide $m^{ed} - m$, and since p and q are distinct primes, we can conclude that pq divides $m^{ed} - m$. Thus, $m^{ed} \equiv m \pmod{n}$. □

Basic RSA signature scheme

RSA signature generation: To sign a message $m \in \{0,1\}^*$, A does the following:

1. Compute $M = H(m)$, where H is a hash function.
2. Compute the signature $s = M^d \bmod n$.
3. A's signed message is (m, s).

RSA signature verification: To verify (m, s), B does the following:

1. Obtain an authenticated copy of A's public key (n, e).
2. Compute $M = H(m)$.
3. Compute $M' = s^e \bmod n$.
4. Accept (m, s) if and only if $M = M'$.

V7b
Integer factorization

RSA

CRYPTO 101: Building Blocks

cryptography101.ca

Big-O and little-o notation

Let $f(n)$ and $g(n)$ be functions from the positive integers to the positive real numbers.

- Big-O notation: We write $f(n) = O(g(n))$ if there exists a positive constant c and a positive integer n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.
- **Example**: $3n^3 + 4n^2 + 79 = O(n^3)$.

- Little-o notation: We write $f(n) = o(g(n))$ if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$.
- **Example**: $\frac{1}{n} = o(1)$.

Measures of running time

Polynomial-time algorithm: One whose worst-case running time is of the form $O(n^c)$, where n is the *input size* and c is a constant.

Exponential-time algorithm: One whose worst-case running time is not of the form $O(n^c)$ for any constant c.

- In this course, fully exponential-time functions are of the form 2^{cn}, where c is a constant; example: $O(2^{n/2})$.
- Subexponential-time algorithm: One whose worst-case running time function is of the form $2^{o(n)}$, and not of the form $O(n^c)$ for any constant c; example: $O(2^{\sqrt{n}})$.

Roughly speaking, "polynomial-time = *efficient*", "fully exponential-time = *terribly inefficient*", and "subexponential-time = *inefficient, but not terribly so*".

Example: Trial division

- Consider the following algorithm (trial division) for factoring an RSA modulus n.
- Trial divide n by the primes $2,3,5,7,11,\ldots, \lfloor\sqrt{n}\rfloor$. If any of these, say ℓ, divides n, then stop and output the factor ℓ of n.
- The running time of this method is at most $\sqrt{n}$ trial divisions, which is $O(\sqrt{n})$.
- **Question**: Is this a polynomial-time algorithm for factoring RSA moduli?

Subexponential time

- Let A be an algorithm whose input is an integer n.
  The input size is $O(\log n)$.
- If the expected running time of A is of the form
  $L_n[\alpha, c] = O\left(\exp((c + o(1))(\log_e n)^{\alpha}(\log_e \log_e n)^{1-\alpha})\right)$,
  where c is a positive constant, and α is a constant satisfying $0 < \alpha < 1$,
  then A is a subexponential-time algorithm.
- Note: If $\alpha = 0$, then $L_n[0,c] = O((\log n)^{c+o(1)})$, which is polytime.
- Note: If $\alpha = 1$, then $L_n[1,c] = O(n^{c+o(1)})$, which is fully exponential time.

Special-purpose factoring algorithms

- **Examples**: Trial division, Pollard's $p-1$ algorithm, Pollard's ρ algorithm, elliptic curve factoring method, special number field sieve.
- These algorithms are only efficient if the number n being factored has a special form, e.g., n has a prime factor p that is relatively small, or $p-1$ has only small prime factors.
- To maximize resistance to these factoring attacks on RSA moduli, one should select the RSA primes p and q at random and of the same bitlength.

General-purpose factoring algorithms

- These are factoring algorithms whose running times do not depend on any properties of the number being factored (other than their size).
- There have been two major developments in the history of factoring:

1. (1982) Quadratic sieve factoring algorithm (QS)
   Running time: $L_n[1/2,1]$.

2. (1990) Number field sieve factoring algorithm (NFS)
   Running time: $L_n[1/3,1.923]$.

Recall: $L_n[\alpha, c] = O\left(\exp((c + o(1))(\log_e n)^{\alpha}(\log_e \log_e n)^{1-\alpha})\right)$.

History of factoring

Year	Number	Bitlength	Method	Notes
1903	$2^{67} - 1$	67	Naive	Francis Cole (3 years of Sundays)
1988	$\approx 10^{100}$	332	QS	100's of computers around the world
1994	RSA-129	425	QS	1600 computers around the world; 8 months
1999	RSA-155	512	NFS	300 workstations + Cray; 5 months
2005	RSA-200	663	NFS	
2009	RSA-768	768	NFS	2000 core years
2019	RSA-240	795	NFS	900 core years
2020	RSA-250	829	NFS	2700 core years

RSA Factoring Challenge : en.wikipedia.org/wiki/RSA_Factoring_Challenge

RSA-250

The largest "hard" number factored to date is RSA-250 (250 decimal digits, 829 bits), which was factored on February 28, 2020.

21403246502407449612644230728393335630086147151447550177977549208814180234471401366433455190958046796109928518724709145876873962619215573630474547705208051190564931066876915900197594056934574522305893259766974716817380693648946998715784949759374979 37

=

641352894770715802787901901705773890848250147429434472081168596320245323446302386235987526683477087766192558569463979885336 7

×

333720275949781565562260106053551142279407603447675546667845209870238417292100370802574486732968818775657189862580369320627 11

RSA-1024

The next interesting factoring challenge is RSA-1024 (1024 bits, 309 decimal digits):

135066410865995223349603216278805969938881475605667027524485143851
526510604859533833940287150571909441798207282164471551373680419703
964191743046496589274256239341020864383202110372958725762358509643
110564073501508187510676594629205563685529475213500852879416377328
533906109750544334999811150056977236890927563

Equivalent security levels

Security (in bits)	Block cipher	Hash function	RSA $\log_2 n$
80	SKIPJACK	(SHA-1)	1024
112	Triple-DES	SHA-224	2048
128	AES small	SHA-256	3072
192	AES medium	SHA-384	7680
256	AES large	SHA-512	15360

Recall that a cryptographic scheme has a security level of ℓ bits if the fastest attack known on the scheme takes approximately 2^{ℓ} operations.

Summary

- Factoring is believed to be a hard problem. However, we have no proof or theoretical evidence that factoring is indeed hard.
- In fact, factoring is known to be easy on a quantum computer.
  - Shor's algorithm (1994) can factor n in $O((\log n)^2)$ operations.
  - The largest number factored with Shor's algorithm is the number 21.
  - *The big open question is whether large-scale quantum computers can ever be built.*
- 512-bit RSA is considered insecure today.
- 1024-bit RSA is considered risky, but still deployed (in legacy applications).
- Most applications have moved to 2048-bit and 3072-bit RSA.

V7c
RSA encryption

RSA

CRYPTO 101: Building Blocks

cryptography101.ca

Security of RSA encryption

- Security of RSA key generation. If an adversary can factor n, she can compute d from (n, e). It has been proven that any efficient method for computing d from (n, e) is equivalent to factoring n.
- Security of Basic RSA encryption. A basic notion of security is that it should be computationally infeasible to compute m from c. This is known as the RSA problem.
- RSA Problem (RSAP): Given an RSA public key (n, e) and $c = m^e \bmod n$ (where $m \in_R [0, n-1]$), compute m.
- The only effective method known for solving RSAP is to factor n (and thereafter compute d and then m). Henceforth, we shall assume that *RSAP is intractable*.

Dictionary attack on Basic RSA encryption

- Dictionary attack. Suppose that the plaintext m is chosen from a relatively small (and known) set $\mathcal{M}$ of messages. Then, given a target ciphertext c, the adversary can encrypt each $m \in \mathcal{M}$ until c is obtained.
- Countermeasure: Append a randomly selected 128-bit string (called a salt) to m prior to encryption. Note that m is now encrypted to one of 2^{128} possible ciphertexts, so a dictionary attack is infeasible.

$m' =$ 

Chosen-ciphertext attack on Basic RSA encryption

Suppose that the adversary E has a target ciphertext c that was encrypted for A. Suppose also that E can induce A to decrypt *any* ciphertext for E, *except for c itself*. (We say that E has a decryption oracle.) Then E can decrypt c as follows:

1. Select arbitrary $x \in [2, n-1]$ with $\gcd(x, n) = 1$.
2. Compute $\hat{c} = cx^e \bmod n$, where (n, e) is A's public key.
   (Note that $\hat{c} \neq c$, unless $\gcd(c, n) \neq 1$.)
3. Obtain the decryption $\hat{m}$ of $\hat{c}$ from the decryption oracle.
   (Note that $\hat{m} \equiv \hat{c}^d \equiv (cx^e)^d \equiv c^d x^{ed} \equiv mx \pmod{n}$.)
4. Compute $m = \hat{m}x^{-1} \bmod n$.

Countermeasure to the chosen-ciphertext attack

Countermeasure: Add some prescribed formatting to m prior to encryption. After decrypting the ciphertext c, if the plaintext is not properly formatted, then A rejects c (and so the decryption oracle does not return a plaintext).

Summary: RSA encryption should incorporate salting and formatting.

Security definition

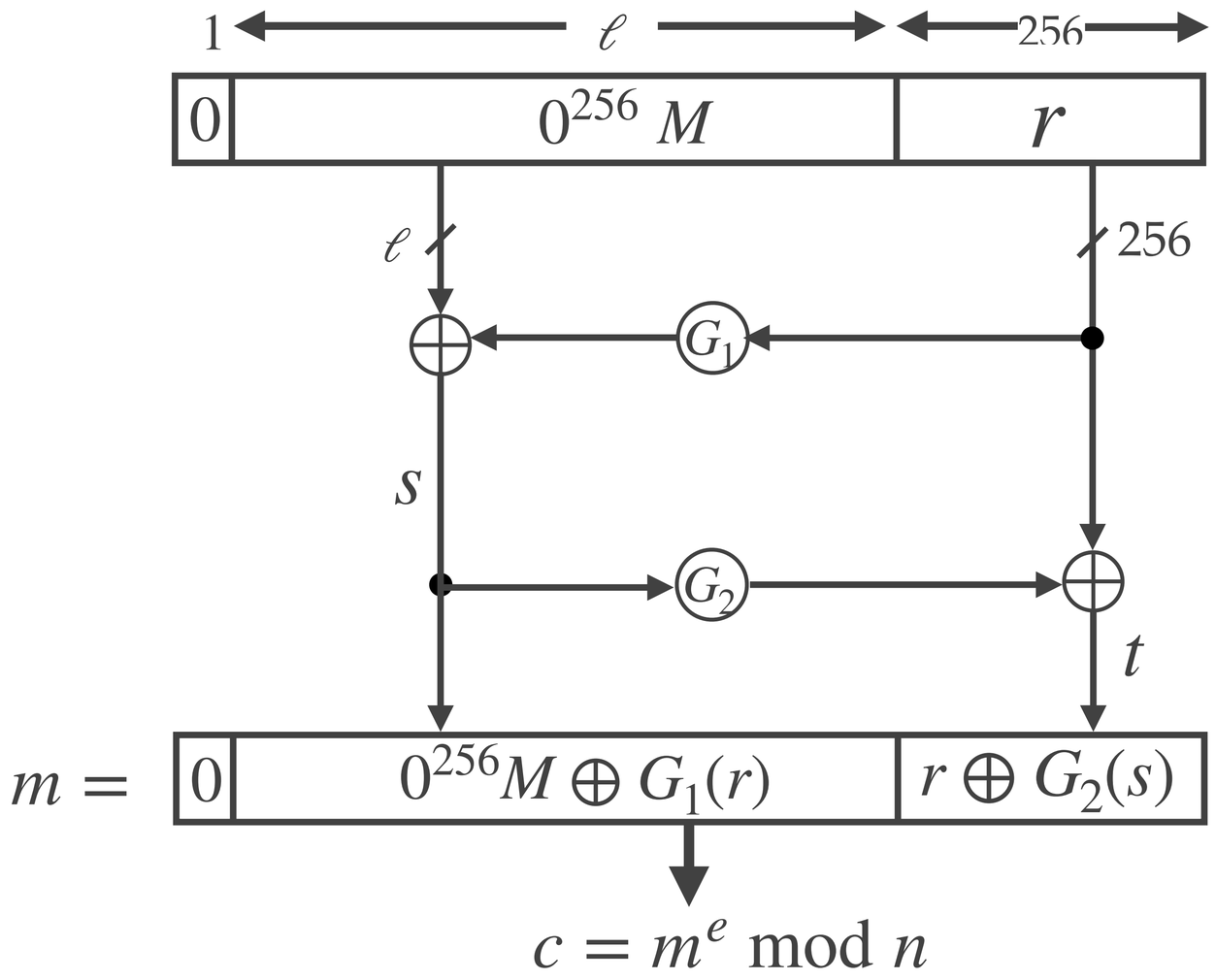
Definition: A public-key encryption scheme is secure if it is semantically secure against chosen-ciphertext attack by a computationally bounded adversary.

To break a public-key encryption scheme, the adversary E has to accomplish the following:

1. E is given the public key and a challenge ciphertext c.
2. E has a decryption oracle, to which she can present any ciphertexts for decryption *except for c itself*.
3. After a feasible amount of computation, E should learn *something* about the plaintext m that corresponds to c (other than its length).

RSA Optimal Asymmetric Encryption Padding (OAEP)

Encryption:



- k = bitlength of n
- $\ell = k - 256 - 1$
- $M \in \{0,1\}^{\ell-256}$ (plaintext)
- $r \in_R \{0,1\}^{256}$ (salt)
- $G_1 : \{0,1\}^{256} \longrightarrow \{0,1\}^{\ell}$
- $G_2 : \{0,1\}^{\ell} \longrightarrow \{0,1\}^{256}$
- G_1 and G_2 are masking functions built from H = SHA256,
  e.g., $G_1(r) = H(0,r) \,\|\, H(1,r) \,\|\, H(2,r) \,\|\, \cdots$

RSA-OAEP (cont'd)

Decryption. To decrypt c, do the following:

1. Compute $m = c^d \bmod n$.
2. Parse m: 

3. Compute $r = G_2(s) \oplus t$.
4. Compute $G_1(r) \oplus s =$ 

5. If $a = 0^{256}$, then output $M = b$;
   else reject c.

Theorem. (Bellare & Rogaway). Suppose that RSAP is intractable. Suppose that G_1 and G_2 are random functions. Then RSA-OAEP is a secure public-key encryption scheme

Key encapsulation mechanisms

- A key encapsulation mechanism (KEM) allows two parties to establish a shared secret key, called a session key.
- A KEM is comprised of three algorithms:
  - Key generation: Each user, say Alice, uses this algorithm to generate an encapsulation key ek (public key) and a decapsulation key dk (the private key).
  - Encapsulation: Bob uses Alice's encapsulation key ek to generate a secret key k and ciphertext c, and sends c to Alice.
  - Decapsulation: Alice uses her decapsulation key dk to recover k from the ciphertext c.

RSA-KEM

Key encapsulation: To select and transport a session key k for A, B does the following:

1. Obtain an authenticated copy of A's encapsulation key (n, e).
2. Select $r \in_R [0, n-1]$.
3. Compute $c = r^e \bmod n$ and $k = \text{KDF}(r)$.
4. Send c to A.

Key generation:

1. A's (public) encapsulation key is $ek = (n, e)$
2. A's (private) decapsulation key is $dk = d$.

Key decapsulation: A processes c as follows:

1. Compute $r = c^d \bmod n$ and $k = \text{KDF}(r)$
2. The session key is k.

V7d
RSA signatures

RSA

CRYPTO 101: Building Blocks

cryptography101.ca

Basic RSA signature scheme

Key generation: Each entity A does the following:

1. Randomly select two large distinct primes p, q of the same bitlength.
2. Compute $n = pq$ and $\phi = (p-1)(q-1)$.
3. Select arbitrary e, $1 < e < \phi$, such that $\gcd(e, \phi) = 1$.
4. Compute d, $1 < d < \phi$, such that $ed \equiv 1 \pmod{\phi}$.
5. A's public key is (n, e); A's private key is d.

Signature generation: To sign a message $m \in \{0,1\}^*$, A does the following:

1. Compute $M = H(m)$, where H is a hash function.
2. Compute $s = M^d \bmod n$ (so $s^e \equiv M^{ed} \equiv M \pmod{n}$).
3. A's signature on m is s.

Signature verification: To verify A's signed message (m, s), B does the following:

1. Obtain an authentic copy of A's public key (n, e).
2. Compute $M = H(m)$.
3. Compute $M' = s^e \bmod n$.
4. Accepts (m, s) if and only if $M = M'$.

Security of the basic RSA signature scheme

Hardness of RSAP: We require that RSAP be intractable, since otherwise E could forge A's signature as follows:

1. Select arbitrary m.
2. Compute $M = H(m)$.
3. Solve $s^e \equiv M \pmod{n}$ for s.
4. Then s is A's signature on m.

Security properties of the hash function

Preimage resistance: If H is not PR, and the range of H is $[0, n-1]$, then signatures can be forged as follows:

1. Select $s \in_R [0, n-1]$ and compute $M = s^e \bmod n$.
2. Find m such that $H(m) = M$.
3. Then s is A's signature on m.

2nd preimage resistance: If H is not 2PR, then signatures can be forged as follows:

1. Suppose that (m, s) is a valid signed message.
2. Find an m', $m' \neq m$, such that $H(m') = H(m)$.
3. Then s is A's signature on m'.

Collision resistance: If H is not CR, then signatures can be forged as follows:

1. Select m_1, m_2 with $m_1 \neq m_2$ and $H(m_1) = H(m_2)$.
2. Induce A to sign m_1: $s = H(m_1)^d \bmod n$.
3. Then s is A's signature on m_2.

The adversary

Goals of the adversary:

1. Total break: E recovers A's private key, or a method for systematically forging A's signatures.
2. Existential forgery: E forges A's signature for a single message of E's choosing; E might not have any control over the content or structure of this message.

Attack model:

1. Key-only attack: The only information E has is A's public key.
2. Known-message attack: E knows some message-signature pairs.
3. Chosen-message attack: E has access to a signing oracle which it can use to obtain A's signatures on some messages of its choosing.

Security definition

Definition: A signature scheme is secure if it is existentially unforgeable by a computationally bounded adversary who launches a chosen-message attack.

- Note: The adversary has access to a signing oracle. Her goal is to compute a single valid message-signature pair for any message (of the adversary's choosing) that was not previously given to the signing oracle.
- **Question**: Is the basic RSA signature scheme secure?
- **Answer**: *No*, if H is SHA-256 (details omitted); *Yes*, if H is "full domain".

Full Domain Hash RSA (RSA-FDH)

- Same as the basic RSA signature scheme, except that the hash function is $H : \{0,1\}^* \rightarrow [0, n-1]$ where n is the RSA modulus.
- In practice, one could define
  $H(m) = \text{SHA-256}(1,m) \,\|\, \text{SHA-256}(2,m) \,\|\, \cdots \,\|\, \text{trunc}(\text{SHA-256}(t,m))$.

Theorem (Bellare & Rogaway, 1996): If RSAP is intractable and H is a random function, then RSA-FDH is a secure signature scheme.

V7e
PKCS #1 v1.5 RSA signatures

RSA

CRYPTO 101: Building Blocks

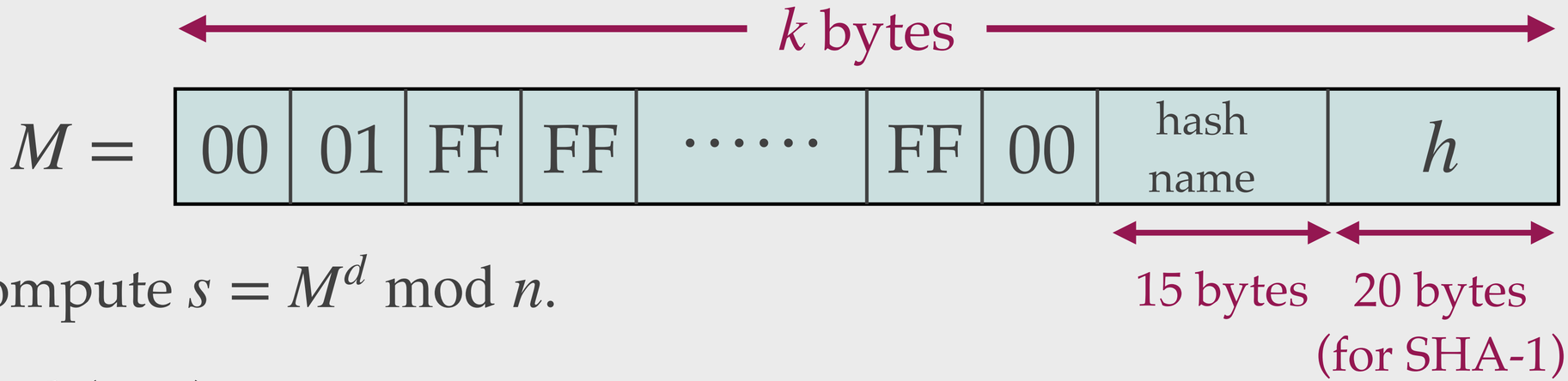
cryptography101.ca

PKCS #1 v1.5 RSA signatures (1993)

PKCS = Public Key Cryptographic Standards

Signature generation: To sign $m \in \{0,1\}^*$, Alice does:

1. Compute $h = H(m)$, where H is a hash function from an approved list.
2. Format h, where k = byte length of n (e.g. $k = 384$):

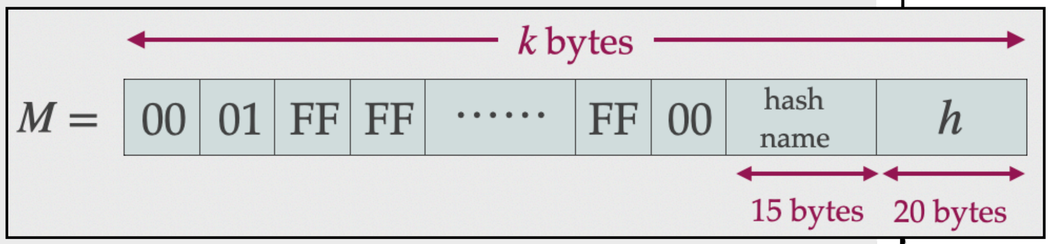


3. Compute $s = M^d \bmod n$.
4. Send (m, s).

PKCS #1 v1.5 RSA signature verification

Signature verification. Bob does:

1. Obtain an authentic copy of Alice's public key (n, e).
2. Compute $M = s^e \bmod n$, and write M as a byte string of length k.
3. Check the formatting:
   (a) First byte is 00.
   (b) Second byte is 01.
   (c) Consecutive FF bytes, followed by 00 byte.
4. From the next 15 bytes, get the hash name; say H = SHA-1.
5. Let h = next 20 bytes. Check that there are no bytes to the right of h.
6. Compute $h' = H(m)$.
7. Accept iff $h = h'$.



Bleichenbacher's attack: Breaking RSA "by hand" (1)

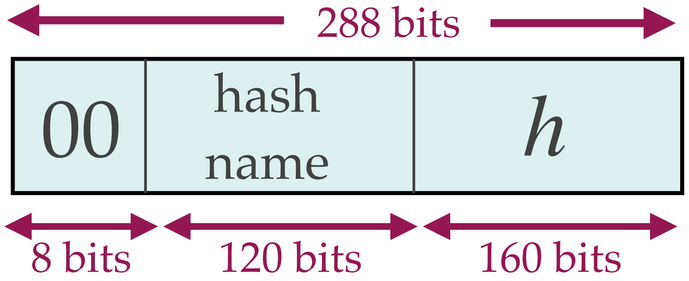
Assumptions:

1. The encryption exponent is $e = 3$: this is commonly used in practice.
2. The hash function is $H =$ SHA-1: this is without loss of generality.
3. The RSA modulus n has bitlength 3072 (384 bytes): this is without much loss of generality.
4. The verifier doesn't check that there are no leftover bytes to the right of h: it turned out many RSA implementation omitted this step, including OpenSSL, SUN's JAVA library, Adobe Acrobat, Firefox, ….

Bleichenbacher's attack (2)

Attack:

1. Select arbitrary $m \in \{0,1\}^*$.
2. Compute $h = H(m)$.
3. Let D be the following 288-bit integer:



4. Let $N = 2^{288} - D$.
5. Check that $3 \mid N$; if $3 \nmid N$, then modify m slightly and to to step 2.
6. Let $s = 2^{1019} - 2^{34}N/3$.
7. Output (m, s).

Bleichenbacher's attack (3)

Claim: The (faulty) verifier will accept (m, s).

Proof: The verifier computes:

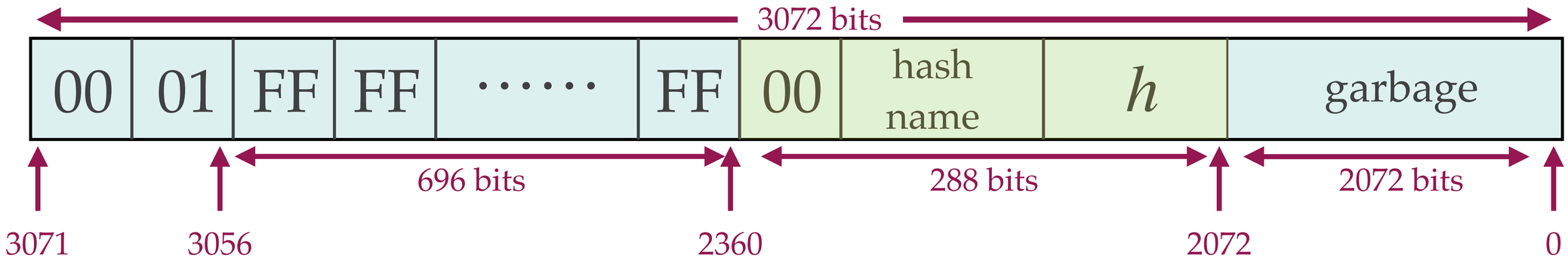
$$M = s^e \bmod n = (2^{1019} - 2^{34}N/3)^3 \bmod n$$
$$= 2^{3057} - 2^{2072}N + 2^{1087}N^2/3 - (2^{34}N/3)^3 \bmod n$$
$$= 2^{3057} - 2^{2072}(2^{288} - D) + \text{garbage} \bmod n$$
$$= 2^{2360}(2^{697} - 1) + 2^{2072}D + \text{garbage}$$

$$\text{garbage} = 2^{1087}N^2/3 - (2^{34}N/3)^3$$

garbage is ≥ 0 and $< 2^{2072}$

mod n is not needed since the integer on the right is less than 2^{3072}

$=$



So, the verifier extracts h, checks that $h = H(m)$, and accepts (m, s). □