Introduction to Modern Cryptography[2]
Lecture 8

December 3, 2013

Instructor: Benny Chor

Teaching Assistant: Nir Bitansky

School of Computer Science
Tel-Aviv University

Fall Semester, 2013–14, 15:00–18:00
Dan David 201

Course site: http://tau-crypto-f13.wikidot.com/

[2]Some slides courtesy of Benny Appelbaum

Lecture 7: Reminder

- The Chinese remainder theorem.
- More properties of the RSA public key cryptosystem.
- Quadratic residues and the Jacobi symbol.
- Semantic security for public key encryption.
- Probabilistic encryption based on QR and QNR over Z_{pq}^*.

Lecture 8: Plan

- Random self reducibility of RSA and QR/QNR: Code.

- Digital Signatures.
- Diffie Hellman signatures paradigm and its shortcomings.
- The hash and decrypt signature paradigm.
- Elgamal probabilistic signature, based on discrete logarithm.
- Other signatures paradigm: Sketch.

- Integer factoring algorithms: Pollard's rho.

Reminder: Distinguished Lectures Day (Dec 10th, 2013)

A three-day celebration of the work of Shafi Goldwasser and Silvio Micali will be held at Weizmann in mid-December 2013, consisting of:
* Dec 10 (TUES): Weizmann Distinguished Lectures Day, featuring talks by Boaz Barak, Irit Dinur, Johan Hastad, and Salil Vadhan, and crowned by lectures of Shafi Goldwasser and Silvio Micali (serving as the annual Amir Pnueli Lectures).

All are welcome, but are asked to notify our administration (by email to distinguished.lectures@weizmann.ac.il) about their intentions to attend.

10-10:45 Boaz Barak (MSR, Cambridge, MA): Games, Proofs, Norms, and Algorithms.
11-11:45 Irit Dinur (Weizmann): Products of Games and Graphs.

(lunch)

13:30-14:15 Johan Hastad (KTH, Stockholm): Approximating Maximum Constraint Satisfaction Problems - A Survey.
14:30-15:15 Salil Vadhan (Harvard): The Complexity of Differential Privacy

(coffee break)

15:45 David Peleg: Introduction for Pnueli Lectures
Oded Goldreich: Introduction to the lecturers
16:00-17:00 Shafi Goldwasser: The Cryptographic Lens
17:15 Richard Karp: Brief remarks about Shafi and Silvio
17:25-18:10 Silvio Micali: Rational Proofs

(reception)

It is next Tuesday, and there will be no class on that day. You are all strongly encouraged to attend this very special and free WIS day (do register though).

Random Self Reducibility of RSA: Revisiting Using Code

Suppose `A(x,m,e)` is an efficient procedure that on inputs $m = pq, x \in Z_{pq}$, and public exponent $e \in Z^*(p-1)(q-1)$ outputs $x^d \bmod pq$, with probability $\varepsilon > 0$. The probability is over the choices of x and the internal coin tosses of `A` [3].

We want to construct a deciphering algorithm `D(x,m,e)` which will run in time polynomial in $n + \varepsilon^{-1}$ and on input as above outputs $x^d \bmod pq$, with probability $\geq 1 - 2^{-n}$.

[3] pq is n bits long, and $ed = 1 \bmod (p-1)(q-1)$.

Random Self Reducibility of RSA: Revisiting Using Code

We discussed this, at the end of lecture 6, on board. To complete the picture, we will now discuss a concrete implementation[4].

We generated two random, independent 500 bit primes p, q, satisfying $p = 2 \pmod 3, q = 2 \pmod 3$. Then 3 and $(p-1)(q-1)$ are relatively prime, so we can use $e = 3$ as the public encryption exponent modulo $m = pq$.

We implemented `A(x,m,epsilon=1/20, e=3)`. On input $x \in Z^*_{pq}$, it produces either $x^d \bmod m$ (a success) with probability ε, or `42` (in case of failure)[5]. To make life easier, $\varepsilon = 1/20$ and $e = 3$ are the default parameters.

[4] We are using Python, since Sage behaves bizarrely with modular arithmetic of very large numbers.

[5] 42 is an homage to Douglas Adams and The Hitchhiker's Guide to the Galaxy.

Random Self Reducibility of RSA: Revisiting Using Code

We start with a utility for inverting modulo m, using extended gcd.

```
def invert(s,m):
    """ s inverse modulo m """
    return(xgcd(s,m)[1] % m)  # since x < m
```

And a utility that calls `A(x,m,epsilon)` with a random argument, `x`, and counts the number of times till `A` succeeds.

```
import random
from oracle import *

def run_A(number=10,epsilon=1/20):
    for i in range(number):
        count=0
        while True:
            count+=1
            x=random.randint(1,m-1)
            y=A(x,m,epsilon)
            if y!= 42:
                print("number of attempts=", count)
                break
    return(x,y)  # returns the last x,y pair
```

Let's run this utility once or twice

Random Self Reducibility of RSA: Revisiting Using Code

Let's run this utility once or twice

```
>>> x,y=run_A()
number of attempts= 6
number of attempts= 16
number of attempts= 8
number of attempts= 34
number of attempts= 99
number of attempts= 9
number of attempts= 1
number of attempts= 4
number of attempts= 64
number of attempts= 4

>>> x,y=run_A()
number of attempts= 20
number of attempts= 9
number of attempts= 20
number of attempts= 7
number of attempts= 50
number of attempts= 2
number of attempts= 1
number of attempts= 6
number of attempts= 3
number of attempts= 38
```

Random Self Reducibility of RSA: Revisiting Using Code

And finally, the algorithm D, which employs A and inverts RSA everywhere (not only where A succeeds). D "spreads" its input randomly in Z_m^* until it succeeds. It also prints the number of randomizations performed till success.

```
def D(x,m,e=3):
    """ inverts RSA everywhere, using A """
    count=0
    t=42
    while t==42:
        count+=1
        r=random.randint(1,m-1)
        r_inverse=invert(r,m)  # r inverse modulo m
        z = (pow(r,e,m) * x) % m
        t=A(z,m)
        # if A fails, it returns 42, and we try again
    print("number of randomizations=", count)
    print
    sol=(t*r_inverse) % m
    return sol
```

Let's run this a few times.

Random Self Reducibility of RSA: Revisiting Using Code

We verify that the returned values are indeed correct.

```
>>> x= m-123456789
>>> sol=D(x,m)
number of randomizations= 26
>>> pow(sol,3,m)-x
0

>>> x= m//2+3
>>> sol=D(x,m)
number of randomizations= 39
>>> pow(sol,3,m)-x
0

>>> x=random.randint(1,m-1)
>>> sol=D(x,m)
number of randomizations= 31
>>> pow(sol,3,m)-x
0

>>> x= 123456789
>>> sol=D(x,m)
number of randomizations= 18
>>> pow(sol,3,m)-x
0
```

Random Self Reducibility of QR/QNR

Involves some simple sampling ideas.

Left as food for thought (or maybe as food for homework assignment).

And Now to Something Completely Different

The sun was shining on the sea,
Shining with all his might:
He did his very best to make
The billows smooth and bright –
And this was odd, because it was
The middle of the night.

The moon was shining sulkily,
Because she thought the sun
Had got no business to be there
After the day was done –
"It's very rude of him," she said,
"To come and spoil the fun!"

Through the Looking-Glass and What Alice Found There:
Lewis Carroll, 1871.

Signatures

Signatures

http://lacourphoto.net/uploaded_images/signatures1-770492.jpg

Hand Written Signatures

- Relate an individual, through a handwritten signature, to a document.
- Signature can be verified against a prior authenticated one, which was signed in person in a bank, in the presence of a public notary public, etc.
- Should be hard to forge.
- Are legally binding (convince a third party, e.g. a judge).

Digital Signature Schemes

- Relate an individual, through a digital string, to a document.
- Would like to achieve all features of hand written signatures, plus more.
- For example, should be able to base difficulty of forgery on some hard computational problem, not just on ineptitude of forger.
- Diffie and Hellman were first to propose such framework.
- An implementation was first given by RSA.

Signatures vs. MACs: Important Distinction

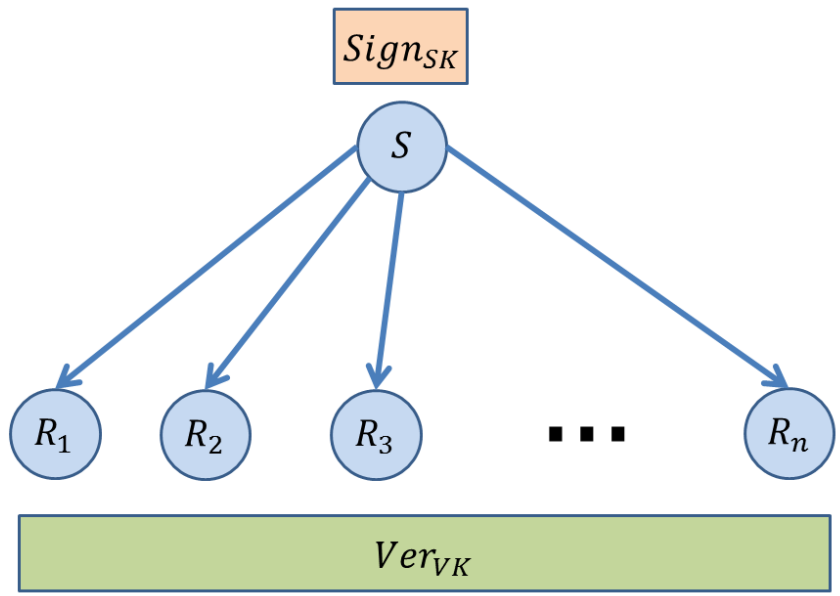
You surely remember message authentication codes (MACs).

- The obvious syntactic difference between signatures and MACs is that MACs require shared secret key, while the secret signature key is known to one user only.

- More importantly, semantically:
- Suppose parties A and B share the secret key K. Then $(M, MAC_K(M))$ convinces A that indeed M originated with B. But nobody else can verify this.
- By way of contrast, given a signature of A on M, it can be verified by anybody that the signature was created by A. Yet nobody else can create such signature.

MACs vs. Signatures (Artwork by an Anonymous Artist)

MACs vs. Signatures (Artwork by an Anonymous Artist)



MACs vs. Signatures: Deniability

Alice wants pay using checks. The Bank should have a mechanism to verify that checks were produced by Alice.

- MACs: Suppose parties $Alice$ and $Bank$ share the secret key K. Then $(M, MAC_K(M))$ convinces $Bank$ that indeed M originated with $Alice$. But in case of a dispute between them, $Alice$ can deny that she produced $(M, MAC_K(M))$, since $Bank$ could have generated it by itself.
- Signatures: Nobody but A can produce a signature of A on M. Thus A cannot deny it, and B can convince a judge the signature was created by A.

- This works well in an idealized world. In the real world, issues like public key infrastructure, legal issues, rational behavior, incriminations, etc., should be dealt with.
  We will not discuss any of these here.

Digital Signatures: Definition

Digital signature is the public-key "analog" of MAC schemes.

- Message space $\mathcal{M}$ (usually long binary strings, e.g., $\{0,1\}^*$)
- Key-Generation algorithm: generates pairs:
  Secret signing key, $\mathtt{sk}$, and public verification key $\mathtt{vk}$
- Signing algorithm: $S_{\mathtt{sk}}(m) \mapsto \sigma$
- Typically, $\sigma \in \{0,1\}^{\ell}$ where ℓ is relatively short
- Verification algorithm: $V_{\mathtt{vk}}(m,\sigma)$ outputs "accept"/"reject"

Security: Intuitively, should be hard to forge a valid tag even after seeing many legal tags

Security (Goldwasser, Micali, Rivest, 1984)[6]

Definition: (Existential Forgery under Chosen Plaintext Attack): A Signature is (t, ε)-secure if every polynomial time bounded adversary $\mathcal{A}$ which gets the public verification key $\mathtt{vk}$ and is allowed to ask for t legal pairs $(m_1, S_{\mathtt{sk}}(m_1)), \ldots, (m_t, S_{\mathtt{sk}}(m_t))$ outputs a new valid pair (m, σ) (such that $V_{\mathtt{vk}}(m, \sigma)$ accepts) with probability at most ε.

- The probability is taken over the choice of the random keys
- Adversary can choose the documents adaptively
- The adversary succeeds even if the document being forged is "meaningless". (As it is hard to predict what has and what does not have a meaning in an unknown context, and how will the receiver react to such successful forgery.)
- Want: large t and small ε (asymptotically, both are super-polynomial, or even exponential, in the key length.)

[6] A paradoxical solution to the signature problem.

Diffie and Hellman "New Directions in Cryptography" (76)

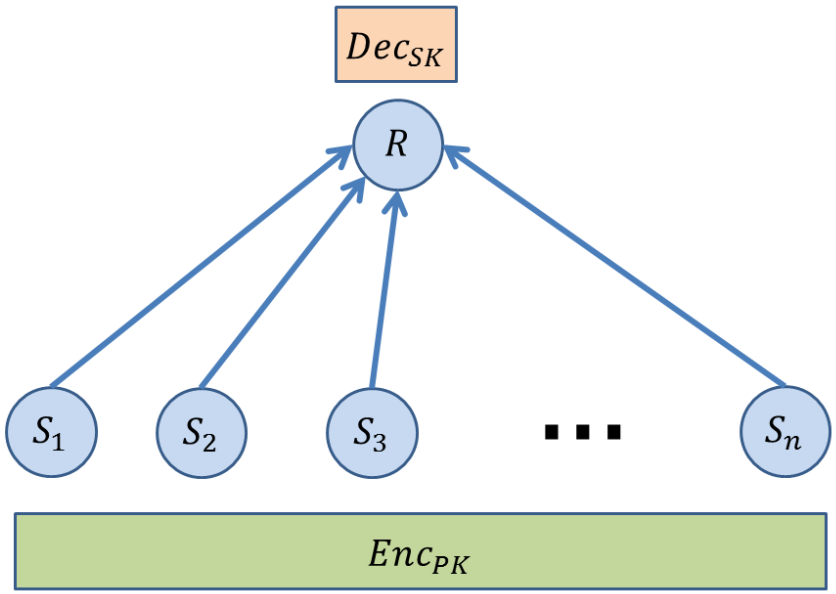
Diffie and Hellman proposed a "textbook framework", based on any deterministic public key cryptosystem.

We will describe this framework and implementation(s), and then discuss some specific shortcomings of it.

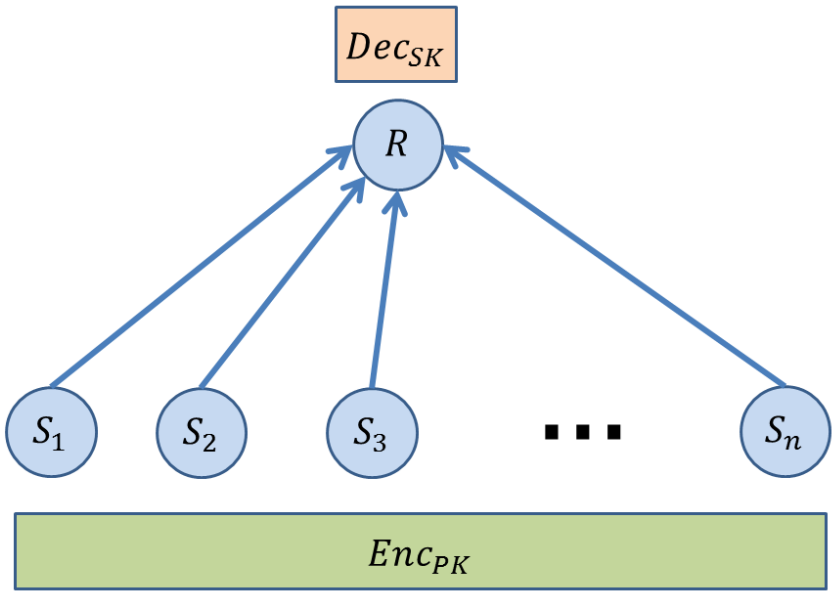
- Let E_A be Alice's public encryption key, and let D_A be Alice's private decryption key.
- To sign the message M, Alice computes the string $y = D_A(M)$ and sends (M, y) to Bob.
- To verify this is indeed Alice's signature, Bob computes the string $x = E_A(y)$ and checks that indeed $x = M$.

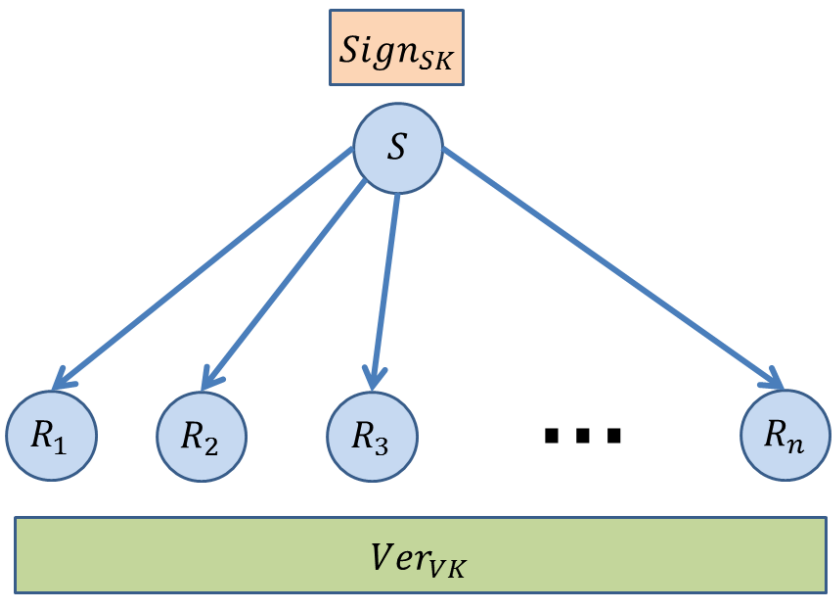
Remark: We tacitly assume here that E_A and D_A commute. This usually is guaranteed as both are permutations, and each is the inverse of the other.

Signatures vs. Public Key Encryption (Artwork by an Anonymous Artist)



Signatures vs. Public Key Encryption (Artwork by an Anonymous Artist)





Diffie and Hellman "New Directions in Cryptography" (76)

Diffie and Hellman proposed a "textbook framework", based on any deterministic public key cryptosystem.

We will describe this framework and some implementation(s), and then discuss some specific shortcomings of it.

Intuition: Only Alice can efficiently compute $y = D_A(M)$, thus forgery should be computationally infeasible.

Question: Do you buy this argument?

Problems with "Pure" Diffie-Hellman Paradigm

- Easy to forge signatures of random messages, even without holding D_A:
- Bob picks R arbitrarily, and computes $s = E_A(R)$ using Alice public key.
- Then the pair (s, R) is a valid signature of Alice on the "message" s.
- Therefore the scheme is subject to existential forgery.
- "So what" ?

Problems with "Pure" Diffie-Hellman Paradigm, cont.

- Easy to forge signatures of random messages, even without holding D_A:
- Bob picks R arbitrarily, and computes $s = E_A(R)$ using Alice public key.
- Then the pair (s, R) is a valid signature of Alice on the "message" s.
- Therefore the scheme is subject to existential forgery.

- Specific implementations may pose additional threats.
- For example, since RSA is multiplicative, we have $D_A(M_1 M_2) = D_A(M_1) D_A(M_2)$ (products are modulo pq).

Problems with RSA Implementation

- Consider specifically RSA. Being multiplicative, we have $D_A(M_1M_2) = D_A(M_1)D_A(M_2)$ (products are modulo pq).

- If M_1="`I OWE BOB $20`" and M_2 = "`100`" then under certain encodings of letters, we could get M_1M_2 ="`I OWE BOB $2000`" ...

Generic Solution: Hash First

- Let E_A be Alice's public encryption key, and let D_A be Alice's private decryption key.
- Let H be an ideal hash function: It hashes all strings into a smaller range, and "behaves like a random function".
- The hash function H is completely public, and in particular no secret key is employed.

Generic Solution: Hash First

- To sign the message M, Alice first hashes the message, namely computes the string $y = H(M)$.
- Then, Alice computes the string $z = D_A(y)$. She sends (M, z) to Bob.
- To verify this is indeed Alice's signature, Bob computes the string $y = E_A(z)$ and checks that indeed $y = H(M)$.

- Argue (intuitively) why this foils previous attacks.

Some Theory: Hash First

- Under the assumption that the hash function is ideal (aka the random oracle model) and the underlying public key encryption is a trapdoor permutation, it has been proven that the "hash first" paradigm is secure against chosen plaintext attacks.
- However, the random oracle assumption seems to require more than just the "standard assumptions".
- Important observation: A pseudo random function with known key is not random anymore!

- It is not known how to prove that the "hash first" paradigm is secure against chosen plaintext attacks under the assumption that the hash function is "just" collision resistant.

Signature Schemes Used in Practice

- RSA (with hashing first, *e.g.* SHA3).

- Elgamal Signature Scheme (1985), based on the intractability of discrete log in Z_p^*. Will be described shortly.
- The DSS/DSA (digital signature standard/algorithm), adopted by NIST in 1994. It is based on a modification of El-Gamal signature.
- Elgamal like scheme in the different groups of elliptic curves. This will probably not be described in this course.

Elgamal Probabilistic PKC, reminder (note resemblance to DH)

- Public information: A large prime p, where $p-1$ has a known factorization and a large prime factor. Recommended to take $p = 2q+1$, q is prime too, and p is 756 or 1024 bits long.
  - A multiplicative generator g of Z_p^*
  - Bob publishes p, g.
  - Bob picks $a \in [0..p-2]$ at random.
  - Bob computes and publishes $\beta = g^a \pmod p$.
- Bob's private information: a.
- Encryption: of the message m:
  - Alice picks $k \in [0..p-2]$ at random.
  - Alice computes $g^k \pmod p$, $m\beta^k \pmod p$.
  - Alice sends $E(m) = (g^k, m \cdot \beta^k)$ to Bob.
    (β^k "masks" m; k obviously is not made public).
- Decryption of $(g^k, m \cdot \beta^k) = (c_1, c_2)$:
  - Bob computes ${c_1}^a = (g^k)^a = (g^a)^k = \beta^k \pmod p$.
  - This enables Bob to compute the multiplicative inverse of $\beta^k \bmod p$, β^{-k} (even though he does not know k).
  - Bob now computes $\beta^{-k} \cdot c_2 = m$. ♠

Properties of Elgamal Public Key Cryptosystem

- Encryption is randomized: $m \to (g^k, m\beta^k)$.
- Alice should use a new, independent k for every encryption.
- Even if same m is sent twice, different k must be used.

- Encryption takes two modular exponentiations.
- Decryption takes one modular exponentiation.
- Ciphertext, $(g^k, m\beta^k)$, is twice as long as plaintext m.

Properties of Elgamal Public Key Cryptosystem (2)

- Cryptosystem is vulnerable to chosen ciphertext attacks.
- Given $E(m) = (c_1, c_2) = (g^k, m\beta^k)$,
- Attacker chooses a random s, computes $(c_1, s \cdot c_2) = (g^k, s \cdot m\beta^k)$
- Attacker asks for decryption of $(c_1, s \cdot c_2)$, which equals $s \cdot m$, from which m is easily recovered.

- Cryptosystem is multiplicative. Given $E(m) = (c_1, c_2) = (g^k, m\beta^k)$, $E(m') = (c_1', c_2') = (g^{k'}, m'\beta^{k'})$, can easily obtain $E(m \cdot m') = (c_1 c_1', c_2 c_2') = (g^{k+k'}, m \cdot m'\beta^{k+k'})$ (without knowing any secret information).

Elgamal Signature Scheme

- Key Generation:
  A large prime p with $p-1$ having a known factorization and a large prime. Recommended to take $p = 2q + 1$, where q is also a prime, and p is 1024 bits long.
  A standard, strongly collision resistant hash function, H.
  - Bob picks a multiplicative generator g of Z_p^*.
  - Bob picks $x \in [0..p-2]$ at random.
  - Bob computes and publishes $y = g^x \pmod p$.
- Bob's private key is x.
- Bob's public signature key is p, g, y.

So far it is similar to Elgamal encryption. This is not a coincidence (think of the hash and sign pardigm in the context of Elgamal's probabilistic encryption).

Elgamal Signature Scheme (2)

Signing the message M, employing randomization:

- Hash: Let $m = H(M)$:
- Bob picks at random $k \in [0..p-2]$ which is relatively prime to $p-1$ (there are $\varphi(p-1) \geq q-1$ such elements).
- Bob computes $r = g^k \pmod{p}$.
- Bob computes $s = (m - rx) \cdot k^{-1} \mod (p-1)$ (***)
- Bob outputs r, s.
- Together with M, this is the signature of M.

Elgamal Signature Scheme (3)

- Bob outputs r, s.
- Together with M, this is the signature of M.

- Verification of M's signature:
- Alice computes $m = H(M)$.
- If $0 < r < p$ and $y^r r^s = g^m \pmod{p}$, Alice accepts. Otherwise she rejects.

Elgamal Signature Scheme (3)

- Bob outputs r, s.
- Together with M, this is the signature of M.

- Verification of M's signature:
- Alice computes $m = H(M)$.
- If $0 < r < p$ and $y^r r^s = g^m \pmod{p}$, Alice accepts.
  Otherwise she rejects.

- What the &%$# is going on? (this is actually not black magic.)
- By (***), $s = (m - rx) \cdot k^{-1} \mod (p-1)$.
  So $sk + rx = m \mod (p-1)$.
- By construction $r = g^k$, so $r^s = g^{ks}$.
- Since $y = g^x$, we have $y^r = g^{rx}$,
  implying $y^r r^s = g^{rx} g^{ks} = g^{sk+rx} = g^m$.

♠

Elgamal Signature Scheme: Some Properties

- Since signing is randomized, same message could have multiple, different signatures. OK to sign same message many times.
- If $k \in [0..p-2]$ is repeated in different mssgs (rather being chosen at random), then Eve can extract the private key (try this!).
- Same applies to highly dependent choices of $k_1, k_2, \ldots$.
- Can be applied in other commutative groups where discrete log is believed to be hard. For example in Elliptic curves (more efficient operations).
- Forging amounts to finding r, s such that $y^r r^s = g^m \pmod p$, and seems to require discrete logs (but not proved equivalent!)
- Overhead:
  - Signature: One exponentiation, can be done offline (preprocessing). A constant number of modular multiplications/gcd (cheap).
  - Verification: Three exponentiations, plus a constant number of modular multiplication.

Signature Schemes: General Remarks

- It is important to realize that signature schemes that deviate from the Diffie-Helman plus hashing paradigm do exist.
- For example, Goldwasser-Micali-Rivest, 1988.

- Theorem: One way functions imply signature schemes secure against chosen plaintext attacks!
- General construction is impractical, though.
- Some of the underlying ideas will be described in the recitation.

And Now to Something Completely Different: Pollard's Rho Factoring Algorithm

'If seven maids with seven mops
Swept it for half a year,
Do you suppose,' the Walrus said,
'That they could get it clear?'
'I doubt it,' said the Carpenter,

And shed a bitter tear.
'O Oysters, come and walk with us!'
The Walrus did beseech.
'A pleasant walk, a pleasant talk,
Along the briny beach:
We cannot do with more than four,
To give a hand to each.'

Through the Looking-Glass and What Alice Found There: Lewis Carroll, 1871.

Factoring Algorithms

What is the running time (worst case) of factoring algorithms? Let m be an n bits composite. Hardest numbers to factor are the product of two distinct prime numbers $m = pq$, where both $p - 1$ and $q - 1$ have a large prime factor.
A (very partial) list of algorithms:

- Trial division: $O(2^{n/2})$.
- J.M. Pollard's rho method: $O(2^{n/4})$.
- Quadratic sieve algorithm: $O(e^{(n \log n)^{1/2}})$.
- General number sieve algorithm: $O(e^{(7n)^{1/3} \cdot \log^2 n})$.
- GNS was introduced by J.M. Pollard in 1988, and later refined by many well-known players of the computational number theory community.

Factoring Algorithms

- The general number sieve algorithm is considered the fastest of all published, "general purpose" factoring algorithms. It was employed to factor RSA-200, a 663-bit number (200 decimal digits), on May 2005. The algorithm was implemented on a cluster of 80 2.2 GHz Opterons. Execution took three months.
- RSA-200 =
  27997833911221327870829467638722601621070446786955428537560009929326128400107609345671052955360856061822351910951365788637105954482006576775098580557613579098734950144178863178946295187237869221823983
  Factors =
  35324619344027701212726049781984643686711974001976250236493034687761212536794232000585479565280883497925869954478333033347085841480059687737975857364219960734330341455767872818152135381409304740185467

Factoring Algorithms

- The general number sieve algorithm is considered the fastest of all published, "general purpose" factoring algorithms. It was employed to factor RSA-200, a 663-bit number (200 decimal digits), on May 2005. The algorithm was implemented on a cluster of 80 2.2 GHz Opterons. Execution took three months.
- RSA-200 =
  27997833911221327870829467638722601621070446786955428537560009929326128400107609345671052955360856061822351910951365788637105954482006576775098580557613579098734950144178863178946295187237869221823983
  Factors =
  3532461934402770121272604978198464368671197400197625023649303468776121253679423200058547956528088349
  and
  7925869954478333033347085841480059687737975857364219960734330341455767872818152135381409304740185467

- We will embark upon a much more modest task: Explain Pollard's rho method, implement it on a 2.2 GHz Core 2 Duo MacBook, using Python 3.1, and run it to factor a 100-bit number (in approximately two to six minutes).

Factoring Algorithms: Pollard's rho method

We will now embarked upon a much more modest task: Explain Pollard's rho method, implement it in Python, and run it to factor an 100-bit number (in approximately two to six minutes).

Let $m = p \cdot q$ be an n bits composite, the product of two primes. Denote by Z_m the ring of integers $\{0, 1, \ldots, m-1\}$, with addition and multiplication modulo m.

Let $f : Z_m \mapsto Z_m$ be a random function from Z_m to itself. Pick some $a_0 \in Z_m$ at random. Consider the sequence

$$a_0, a_1 = f(a_0), a_2 = f(a_1), \ldots, a_i = f(a_{i-1}) \ .$$

Finding Cycles: The Hare and the Tortoise

Let $f : Z_m \mapsto Z_m$ be a random function from Z_m to itself. Pick some $a_0 \in Z_m$ at random. Consider the sequence

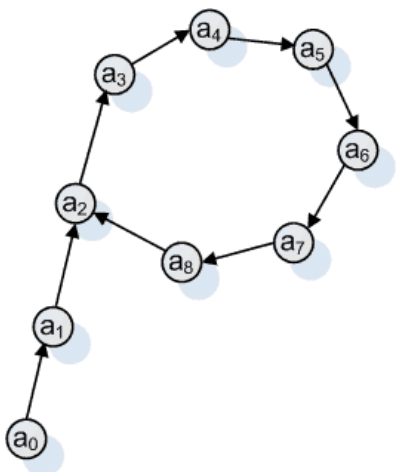
$$a_0, a_1 = f(a_0), a_2 = f(a_1), \ldots, a_i = f(a_{i-1}), \ldots$$

Finding Cycles: The Hare and the Tortoise

Let $f : Z_m \mapsto Z_m$ be a random function from Z_m to itself. Pick some $a_0 \in Z_m$ at random. Consider the sequence

$$a_0, a_1 = f(a_0), a_2 = f(a_1), \ldots, a_i = f(a_{i-1}), \ldots$$

Since Z_m is finite, this sequence is finite, and it must contain a cycle: Two indices $i \geq 0$ and $k \geq 1$ such that $a_i = a_{i+k}$. If we take $i \geq 0$ to be minimal and then $k \geq 1$ to be minimal, we get the following picture, where $i = 2$ and $k = 7$ (taken from Wikipedia site in Polish).



We will introduce the (fast) hare and the (slow) tortoise shortly.

Finding Cycles: The Hare and the Tortoise

The hare and the tortoise will start from the same random starting point, x_0.

The slow hare will proceed one step at a time:
$x_1 = f(x_0), x_2 = f(x_1), \ldots$.
The faster hare will proceed at double speed, two steps at a time:
$y_1 = f(f(x_0)), y_2 = f(f(x_1)), \ldots$.

It is not too hard to see that the hare and the tortoise will meet – namely there is a time step $j > 0$ where $x_j = y_j$. They will thus find a cycle (indices i, k) in $O(i + k)$ operations (invocations of f, and comparisons).

Length of Tails and Cycles

Let $f : Z_m \mapsto Z_m$ be a random function from Z_m to itself. Pick some $a_0 \in Z_m$ at random. Consider the sequence

$$a_0, a_1 = f(a_0), a_2 = f(a_1), \ldots, a_i = f(a_{i-1}), \ldots$$

By the birthday paradox, the expected length of the tail and cycle (combined), $i + k$, is $O(\sqrt{m})$.

Length of Tails and Cycles: Under the Hood

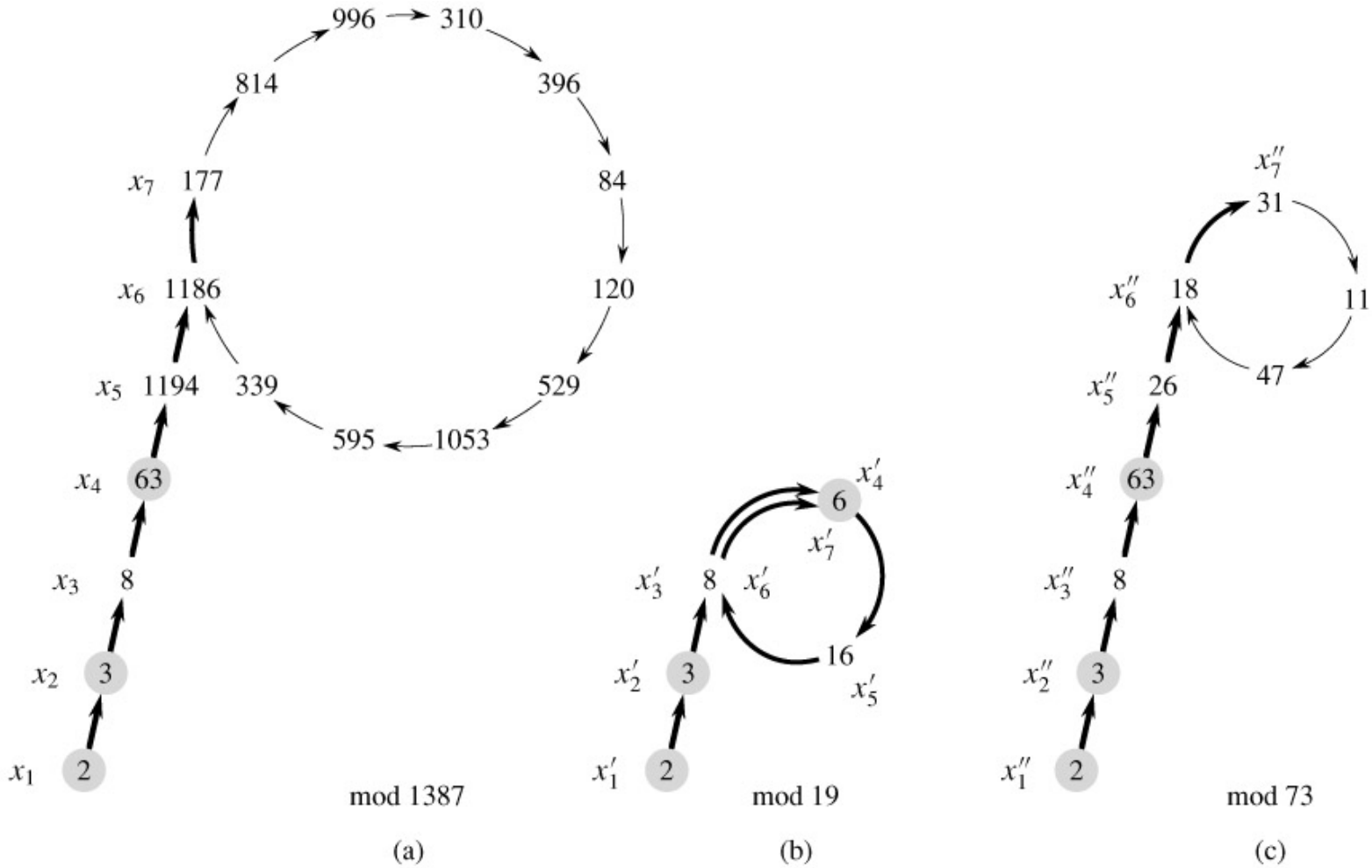
Let $f : Z_m \mapsto Z_m$ be a random function from Z_m to itself. Pick some $a_0 \in Z_m$ at random. Consider the sequence

$$\{a_0, a_1 = f(a_0), a_2 = f(a_1), \ldots, a_i = f(a_{i-1}), \ldots\} \subseteq Z_m \ .$$

Recall that $m = p \cdot q$. Under the hood, $f : Z_m \mapsto Z_m$ "splits" to $f_p : Z_p \mapsto Z_p$ and $f_q : Z_q \mapsto Z_q$.

The function f_p will close a cycle of expected length $O(\sqrt{p})$, while f_q will close a cycle of expected length $O(\sqrt{q})$. These two cycles are essentially independent and will typically not be of the same length.

Length of Tails and Cycles: Under the Hood



Figure, corresponding to m=1387, p=19, q=73, is taken from http://integrator-crimea.com/ddu0208.html

Length of Tails and Cycles: The Punchline

Recall that $m = p \cdot q$. Under the hood, $f : Z_m \mapsto Z_m$ "splits" to $f_p : Z_p \mapsto Z_p$ and $f_q : Z_q \mapsto Z_q$. The function f_p will close a cycle of expected length $O(\sqrt{p})$, while f_q will close a cycle of expected length $O(\sqrt{q})$. Note that we cannot "see" the functions f_p, f_q directly.

Length of Tails and Cycles: The Punchline

Recall that $m = p \cdot q$. Under the hood, $f : Z_m \mapsto Z_m$ "splits" to $f_p : Z_p \mapsto Z_p$ and $f_q : Z_q \mapsto Z_q$. The function f_p will close a cycle of expected length $O(\sqrt{p})$, while f_q will close a cycle of expected length $O(\sqrt{q})$. Note that we cannot "see" the functions f_p, f_q directly.

Suppose f_p closes a cycle, namely $f_p(a_i) = f_p(a_{i+k})$, but f_q does not close a cycle at the same place, namely $f_q(a_i) \neq f_q(a_{i+k})$.

The equality $f_p(a_i) = f_p(a_{i+k})$ means $f(a_i) = f(a_{i+k}) \pmod{p}$, while the inequality $f_q(a_i) \neq f_q(a_{i+k})$ means $f(a_i) \neq f(a_{i+k}) \pmod{q}$.

(we are almost there...)

Length of Tails and Cycles: The Punchline, cont.

The equality $f_p(a_i) = f_p(a_{i+k})$ means $f(a_i) - f(a_{i+k}) \pmod{p}$. Therefore, p divides $f(a_i) - f(a_{i+k})$. This implies that p divides the greatest common divisor of m and $f(a_i) - f(a_{i+k})$.

The inequality $f_q(a_i) \neq f_q(a_{i+k})$ implies that q does not divide the greatest common divisor of m and $f(a_i) - f(a_{i+k})$.

The two facts together imply that $\gcd(m, f(a_i) - f(a_{i+k})) = p$. So by closing a cycle in $O(\sqrt{p}) = O(m^{1/4}) = O(2^{n/4})$ steps and $O(1)$ memory, we find a prime factor of m.

Something is Missing

The memory required by the hare and tortoise is $O(1)$, regardless of the length of the tail plus the cycle. But the expected time analysis strongly relied on f being a random function.

Random functions are easy to generate, but to keep track of $\{a_0, a_1 = f(a_0), a_2 = f(a_1), \ldots, a_i = f(a_{i-1}), \ldots\} \subseteq Z_m$, we will have to store $O(\sqrt{p})$ elements, defying our saving of memory.

Idea: Instead of a truly random function, we will just employ a simple quadratic function $f : Z_m \mapsto Z_m$. For example, $f(x) = x^2 + 3 \pmod m$.

This way, we avoid the need to store all previous values in memory. But now there is no proof that the expected length of the cycle is indeed $O(\sqrt{p})$. However, it won't hurt to try this approach. If it successfully factors, the lack of proof of performance will not deter us.

Pollard ρ Algorithm: Python Code

```
def rho(m,f=lambda x: x*x+3):
    """ pollard rho factoring """
    a=random.randint(0,m-1)
    print("starting point=",a,"\n")
    t=h=a
    s=int(m**0.25)
    done=False
    count=1
    while not done:
        t=f(t) % m
        a=f(f(a) % m) % m
        g=gcd(m,a-t)
        if g>1 and g<m :
            done=True
        else:
            count=count+1
        if (count % s) == 0:
            print (count//s,"\n")
            # visualize length of run, in quanta of s
    print(g," divides ", m, "\n")
    return g
```

Pollard ρ Algorithm: Sample Runs

```
>>> m=next_prime(2**47)[0]*next_prime(2**50)[0]
>>> elapsed("rho(m)")
starting point= 131754030626751445081856576239

140737488355333  divides  158456325028542045248481657107
145.673607  # under 2.5 minutes

>>> m=next_prime(2**48)[0]*next_prime(2**50)[0]
>>> elapsed("rho(m)")
starting point= 144297373938793782605469763606

1125899906842679  divides  31691265005709647539593858368 3
292.098062 # under 5 minutes
>>> next_prime(2**48)[0]
281474976710677
>>> next_prime(2**50)[0]
1125899906842679

>>> m=next_prime(2**49+2**20)[0]*next_prime(2**50)[0]
>>> elapsed("rho(m)")
starting point= 485858716743943226937981458 73

562949954469907  divides  633825301294758675811488760853
24.16170699999998 # under 0.5 minutes
```

Quo vadis? – Eo Romam Iterum Crucifigi

Well, our planned future is not as bleak as the (claimed) source of this quote.

So far, we became familiar with a variety of cryptographic notions and primitives, relevant group theory and number theory, as well as specific implementations of them. These include:

- Indistinguishability; semantic security.
- Pseudo random generators, pseudo random functions and pseudo random permutations.
- Encryptions (stream cyphers, block cyphers, different modes of operation) and MACs (in the symmetric key setting).
- Cryptographic hashing.
- Public key framework; Key exchange (Diffie Hellman).
- Encryptions and signature schemes, including probabilistic encryptions (in the public key setting).

Quo vadis? – Eo Romam Iterum Crucifigi

Well, our planned future is not as bleak as the (claimed) source of this quote.
In the remaining weeks, we will introduce a number of additional cryptographic primitives, as well as cryptographic notions with proposed implementations. These include (tentative order):

- Multi party computations.
- Fully homeomorphic encryption and secure delegation of computation.
- Zero knowledge proofs.
- Secret sharing.
- Threshold cryptography.
- Coin flipping over the phone.
- TBA (so we retain some mystery).