Digital Signatures

34842 characters in 6624 words on 924 lines

Florian Moser

August 17, 2020

1 introduction

digital equivalent to hand signatures
sender has (s_k, p_k); signs with s_k
receiver uses publicly known p_k to verify signature

motivation
want to receive message with guarantees
like hand signature (unique, receiver recognised)
integrity (no accidiential/malicious modification)
authenticity (sender is verified)

applications
signatures (like PKI)
payment (like EMV)
updates (like pacman, windows updates)
identity card (like passport)
for other cryto applications (like public key encryption)

definition
triple E = (Gen, Sign, Vfy)
Gen(1^k) produces (pk, sk) with security parameter k
Sign(sk, m) produces signature s
Vfy(pk, m, s) return 1 iff signature valid for m, pk else 0

properties
correctness ("it works")
if (pk, sk) ← Gen(1^k) and then s = Sign(sk, m) for some m
then Vfy(pk, m, s) must output 1
must hold for any m / (pk, sk) pairs (exceptions negligible)
soundness ("it is secure")
adversary with specific powers does not break the security guarantee
intended application motivates adversary powers

other properties
deterministic; ie wether same m/pk/sk lead to same signature
efficient; ie how many hashes + signature steps are done

2 foundations

shamir's trick
for J,S ∈ Z_n, e,f ∈ Z, ggt(e, f) = 1
can transform $J^f = S^e$ mod N ⇒ x^e = J mod N
calculate ggt(f, e) = a*f + b*e = 1
then x = $S^a * J^b$ mod N
works bc $J^f = S^e \Leftrightarrow J^{fa} = S^{ea} \Leftrightarrow$ J^(fa+be) = $S^{ae} * J^{be}$

hemming weight
#1 in bit string
010111 has hemming weight 4

random walks
each round, choose {-1, 0, 1}
track position on number line (simply add all together)
back at origin for length N with 1 / O(sqrt(N))

negligible
if negl(k) converges faster to 0 as inverse of polynom
negl(k) = o (1/poly(k))
for all c there is k_c such that for all k > k_c negl(k) ≤ $1/k^c$

probabilistic algorithms
las vegas if always correct but no guaranteed termination
monte carlo if always terminates but no guaranteed correctness
can turn algo with some success p in time t
to algo which always succeeds in expected time t/p

3 security definitions

combine adversary capabilities (what adversary is able to do)
with adversary goal (what adversary must be able to archive)
protocol secure if probability p of adversary win is ≤ negl(k)
stronger adversary with weaker target gives higher security guarantee

3.1 adversarial capabilities

what adversary is able to do
more capabilities lead to stronger security guarantee

no-message attacks (NMA)
adversary only receives pk as input
(no access to valid signatures)
Pr[A^C(pk, m∗) = (s∗): Vfy(pk, m∗, s∗) = 1] ≤ negl(k)

non-adaptive chosen-message attacks (naCMA)
adversary chooses (m_1, m_2, ...) messages
receives pk and signatures (s_1, s_2, ...)
Pr[A^C(pk) = (m∗, s∗): Vfy(pk, m∗, s∗) = 1] ≤ negl(k)
easier than CMA because can pick pk specifically for messages
for example to ensure pk can indeed sign all m

adaptive chosen-message attacks (CMA)
adversary receives pk as input
depending on pk and previously known messages/signatures
can request a signature for next message
Pr[A^C(pk) = (m∗, s∗): Vfy(pk, m∗, s∗) = 1] ≤ negl(k)
most relevant in practice

comparison
NMA < naCMA < CMA
hence NMA least powerful adversary

3.2 adversarial goal

what adversary must be able to archive
less restrictions lead to stronger security guarantee

universal unforgeable (UUF)
adversary produces signature for given randomly-chosen m∗
for CMA, prevent signer to produce signature for that m∗

selective unforgeable (SUF)
adversary decides at start of protocol which m∗ it wants to sign
for CMA, prevent signer to produce signature for that m∗

existential unforgeable (EUF)
adversary produces signature for self-chosen m∗
for CMA, adversary must choose m∗ != all signed m_i (m∗ ∉ {m_1, m_2, ...})
(so m_i must not be reused)

strong existential unforgeable (sEUF)
adversary produces signature for self-chosen m∗
for CMA, adversary must choose (m∗, s∗) != to all signed (m_i, s_i) ((m∗, s∗) ∉ {(m_1, s_1), (m_2, s_2), ...})
(so could reuse m_i if different s∗ can be generated)

comparison
UUF > EUF > sEUF
hence UUF hardest to archive for adversary

3.3 security experiment

to prove security of signature scheme under security definition
adversary A plays against challenger C
adversary wins when security definition is broken

UUF-NMA game
C generates (pk, sk) ← Gen(1^k)
C chooses randomly m∗ ← {0,1}^n
C sends pk, m∗ to A
A returns s∗
A wins if Vfy(pk, m∗, s∗) = 1
protocol secure if Pr[A^C(pk, m∗) = (s∗): Vfy(pk, m∗, s∗) = 1] ≤ negl(k)

EUF-CMA game
C generates (pk, sk) ← Gen(1^k)

C sends pk to A
A chooses m_i and sends to C
C sends $s_i \leftarrow$ Sign(sk, m_i) to A
A repeats process q times
A returns (m∗, s∗)
A wins if m∗ != m_i and Vfy(pk, m∗, s∗) = 1
protocol secure if Pr[A^C(pk) = (m∗, s∗): Vfy(pk, m∗, s∗) = 1 ^ m∗ $\notin \{m_1, m_2, ...\}] \leq$ negl(k)

sEUF-CMA game
C generates (pk, sk) with Gen(1^k)
C sends pk to A
A asks C for signature s_i of m_i one after the other
A returns (m∗, s∗)
A wins if (m∗, s∗) not equal to any previously seen pair and Vfy(pk, m∗, s∗) = 1
protocol secure if Pr[A^C(pk) = (m∗, s∗): Vfy(pk, m∗, s∗) = 1 ^ (m∗, s∗) $\notin \{m_1, s_1), ...\}] \leq$ negl(k)

3.4 adversary runtime

unrestricted adversary breaks any protocol (simply bruteforce)
hence restrict to probabilistic polynomial time (PPT) runtime

unlimited adversary breaks any scheme
for L max bit length of signature for any message m
construct UUF-NMA adversary with runtime 2^L and success p = 1
⇒ simply bruteforces signature

PPT adversary breaks any scheme with some probability
for L max bit length of signature for any message m
construct UUF-NMA adversary with runtime L and success p = 2^{-L}
⇒ simply guesses single signature

3.5 security reduction

usually have assumption A ⇒ security claim S
do proof by contradiction, hence show (not S) ⇒ (not A)

proof setup
show that for adversary A breaking security claim
that there exists an adversary B breaking security assumption
challenger C lets B break security assumption
B transforms C input
B acts as challenger to A, using input of C (simulation)
B responds to C, using output of A (extraction)

validity requirements
A must not able to differentiate B and real challenger
⇒ show that exchanged values have same distribution (simulation)
show relation of success probability of A e_A and B e_B
want tight reductions with constant loss ($e_B > e_A$ rather than $e_B > e_A$/N)
need at least PPT requirement
⇒ show how B gets solution from successful A (extraction)
show that time of t_A and t_B are comparable
⇒ argue that steps of B do not introduce big overhead

construct proof
draw challenger C, adversary B and adversary A
draw predefined communication between C, B and A
(b) show how B constructs values sent to A (simulation)
(a) show how B constructs solution out of successful A (extraction)

broken EUF-CMA ⇒ broken UUF-CMA
assume A can break EUF-CMA
construct B which breaks UUF-NMA using said A
let C be challenger for B
C sends pk to B
B chooses m∗
B sends pk, m∗ to A
A returns (m∗, s∗)
when A succeds, B outputs (m∗, s∗)

3.6 parameter choices

security given under security parameter 1^k
choose k according to real-world practical speed

supercomputer performance (2019)
2^{58} FLOP/s reached by summit (IBM)
2^{80} FLOPS in 2^{22} seconds (49 days)

adversary restrictions
t_A operations / steps
q signature queries
q_H signature computations (for ROM)

general number field sieve (GNFS) assumption
"no las vegas algorithm solves factorization more efficient than GNFS"
GNFS solves factorization with some probability p
helps to estimate RSA security parameters

RSA parameter estimation
$e_B \geq e_A$ / q_H as success probability of B (as seen in RSA-FDH)
hence 1/e_B ∗ $t_B \leq$ q/e_A ∗ t_A ressource consumption ("inverse quality")
choose n such that t_GNFS(n) > $q_H/e_A * t_A$ (to contradict GNFS assumption)
we want $e_A < 1/2^{80}$ (bc supercomputer) & allow 2^{30} hash queries
hence $e_B \geq 1/2^{110}$

3.7 random oracle model (ROM)

for problems with unknown / inefficient proofs in standard model
hash function replaced by oracle returning random bitstrings
output unique, independent, uniform for each m, programmable
attacker can only evaluate function (no further benefits whatsoever)
"random oracle heuristic" if used hash function assumed to be indistinguishable to ROM for attacker

construction as lookup table
if requested value m' in table, return value
else, choose random value, place in lookup table & return

random oracle interaction
attacker asks random oracle for hash of some m
oracle responds with uniform distributed y = H(m)
answers are consistent; for m' = m ⇒ H(m) = H(m')
all parties ask same oracle

reduction proof in ROM
program some value of H(m_i) to be the hardness assumption
then hope attacker signs said H(m_i) (hence breaking the hardness)
can assume attacker asks for hash (1) before signing query (2)
(1) else attacker has to guess H(m) output
but only success p = 1/N (bc uniform distribution)
(2) else execute hash query self, respond later with cached value

analysis
often more efficient than non-ROM schemes
may be first step towards proof in non-ROM model
but insecure constructions with ROM exist ((ab)using unlimited randomness)

3.8 programmable hash function (PHF)

similar tool as random oracle, but in standard model

group hash function (GHF)
for group G with generators g
Gen(g) outputs function description k
Eval(k, m) implements hash function H_k : {0,l}^l → G

(v,w,y)-programmable hash function (PHF)
based on group hash functions, adding a trap door
for group G with generators g, h
TrapGen(g, h) gets two generators of G
outputs function description k and trapdoor t
k_TrapGen and k_{Gen} must be identically distributed for all g, h
TrapEval(k, m) outputs (a,b) such that H_k(m) = $h^a * g^b$
Pr[zero for v m∗, non-zero for w m] ≥ y
a is "well-distributed" if "often-enough" used in hash leading to forgery

why PHF can replace ROM
ROM allows to use "uncontrolled" H(m_i) in proof
we can sign other messages, but not sign H(m_i)
but hope A uses it for forgery
PHF approximates behaviour with "uncontrolled" v and "controlled" w
we can sign w messages (which have h-component), but not v
we hope that A uses v for forgery
this happens with some probability y guaranteed by PHF

3.9 assumptions

hardness of protocols based on hardness assumption

P = NP ⇒ broken UUF-NMA
let L be language with all prefixes of signature s for (pk, m)
L ∈ NP because witness is s
when L ∈ P then TM T exists deciding language
assume P = NP, hence can use T in reduction

C sends pk, m to A
A uses T to find valid prefix bit-by-bit
signature length is polynomial, hence protocol has polynomial runtime

one-way functions (OWF)
assume non invertable functions exists
relatively weak assumption
for f : {0,1}∗ → {0,1}∗
given y for random x such that y = f(x)
find x' such that f(x') = y

discrete logarithm assumption (DL)
assume groups exist where discrete logarithm is hard
prime groups or elliptic courves used in practice
in group G with generator g
given random y (and g)
find x such that g^x = y

(weak) RSA assumption (RSA)
assume finding e'th root is hard
in group G_N for N = P∗Q (two random primes)
let ord(Z_N) = (P-1)∗(Q-1) = phi(N)
for some e such that e > 1, ggT(e, phi(N)) = 1
given random y (and N, e)
find x such that x^e mod N = y
at most as hard as factoring N, bc with known P,Q easily solvable
can use ggt(e, phi(N)) to get d = e^{-1} mod phi(N)
with d, can get x bc y^d = xˆ(e∗d) = x
factoring of N could serve as a trapdoor

strong RSA assumption (sRSA)
assume finding e'th root is hard
in group G_N for N = P∗Q (two random primes)
let ord(Z_N) = (P-1)∗(Q-1) = phi(N)
given random y (and N)
find x, e, e > 1 such that x^e mod N = y
"strong" because attacker has more possible ways to succeed

computational diffie-hellman problem assumption (CDH)
given (g, g^x, g^y)
find x such that x = g^{xy}

4 general constructs

4.1 one-way functions

f such that {0,1}∗ → {0,1}∗ and input x = {0,1}∗
y = f(x) efficient (runtime & |y| < some polynomial p(|x|))
x = fˆ-1(y) hard; no polytime algorithm exists

security experiment
f publicly known
C chooses x ← {0,1}ˆk randomly
C sends y = f(x) to A
A returns x' such that y = f(x')
Pr[A(1^k, y) = x' : f(x') = y] ≤ negl(k)

4.2 pseudo-random functions

PRF : {0,1}ˆk (seed) x {0,1}ˆn (value) → {0,1}ˆl
if seed is chosen randomly indistinguishable to randomly chosen F
Pr[AˆPRF(s,.) (1^k) = 1] - Pr[AˆF(.) (1^k)] ≤ negl(k)

4.3 hash functions

map {0,1}∗ → M_t for M_t some output message room

hash function
H = (Gen_H, $Eval_H$)
Gen_H(1ˆk) generates key t describing H_t: {0,1}∗ → M_t
Eval_H(1ˆk, t, m) calculates H_t(m)

collision resistant
for A poly-time algorithm
Pr[A(1^k, t) = (x, x') : H_t(x) = H_t(x')] ≤ negl(k)
requires a hardness assumption

in practice
crypto hash-functions like SHA-256 used
used heavily in practice, invertion is practically impossible
but no 1^k key, fixed output length
hence PPT adversary breaks scheme

4.4 prime-number hash functions

hash functions always mapping to prime

heuristic construction
let H be hash function of {0,1}∗ → {0,1}ˆl
construct h(m) = H(m || y) for y smallest number for result prime
if prime numbers distributed equally, expect log(2^l) = l tries

chamäleon construction
let ch be chamäleon hash function ch(m, r) → [0, N-1] for r ∈ N
choose random m', r' until ch(m', r') is prime number (log N tries)
calculate r such that ch(m, r) = ch(m', r') with TrapColl

4.5 merkle trees

able to compress public key cleverly

process
generate hash of all values
hash recursively; h_(j-1,i) = H(h_(j,2i-1) || h_(j,2i))

naming
root is h_(0,1)
siblings if hashed together as defined by recursive hashing
father if result of two sibilings
path includes all nodes on shortest connection
co-path includes all siblings of nodes on path

verify key
need hash of value to be verified & values on co-path
after recursive hashing, get hash of root

4.6 arbitrary size message rooms

want to sign messages of arbitrary length {0,1}∗
with signature scheme of limited message space (M_t)
use collisionresistant hash function for {0,1}∗ → M_t

construction ("hash-then-sign")
for E' be signature scheme with message room M
for H maps {0,1}∗ → M
Gen(1^k) = Gen(1^k)'
Sign(sk, m) = Sign'(sk, H(m))
Vfy(sk, m, s) = Vfy'(pk, H(m), s)

5 one-time signatures (OTS)

remain secure if one signature is known (but not more)
constructed from hardness assumption
building block of "repeated" signature schemes

5.1 security experiments

because one-time signatures only need to hold once
change CMA/naCMA parameter q(k) to q=1

EUF under onetime naCMA (EUF-1-naCMA)
A sends single message m to C
C responds with pk and signature s
A has to output valid (m∗, s∗)
Pr[AˆC(pk) = (m∗, s∗): Vfy(pk, m∗, s∗) = 1 ˆ m∗ != m] ≤ negl(k)

EUF under onetime CMA (EUF-1-CMA)
C sends pk to A
A sends single message m to C
C responds with signature s
A has to output valid (m∗, s∗)

UUF-NMA useless
construct provably secure scheme with signature as secret key
but useless, bc independent on m
Gen(1^k) chooses random sk, pk = f(sk)
Sign(sk, m) = sk
Vfy(pk, m, s) = f(s) = pk

5.2 lamport one-way function signature

EUF-1-naCMA using OWF

construction
for n length of message
choose n x_{i0} and n x_{i1} random values (2n values total)
calculate y_{ij} = f(x_{ij})
Gen(1^k) outputs pk = (f, y_{10}, y_{11}, ...), sk = (x_{10}, x_{11}, ...)
Sign(sk, m) = (x_{1a}, x_{2b}, ...) for a, b, ... respective bit in message

Vfy(pk, m, s) verifies if f(x_{1a}) = y_{1a}, ...

broken EUF-1-naCMA ⇒ broken one-way function
C chooses random x
C sends y = f(x) for random x and f to B
A sends m to B (due to 1-naCMA)
B chooses random y_{kj} such that j != bit of message m at place k
sets y_{kj} = y (challenge y)
generates other y_{ij} = f(x_{ij}) for 2n-1 random x_{ij}
generates signature s for m
sends (pk, s) to A
A sends (m∗, s∗) to B
when A succeeds, check m∗ bit k different (p = 1/n)
if yes, B can output s_k bc f(s_k) = y
hence success probability $e_B \geq$ 1/n ∗ e_A

5.3 discrete logarithm signature

EUF-1-naCMA using DL

contruction
for group Z_p with generator g
Gen(1^k) chooses random x, w ← Z_p
outputs pk =(g, h=g^x, c=g^w), sk = (x, w)
Sign(sk, m) → s = (w - m) / x
Vfy(pk, m, s) → c = g^m ∗ h^s
works bc c = g^w = gˆ(m + x((w-m)/x)) = g^m ∗ g^x∗s = g^m ∗ h^s

broken EUF-1-naCMA ⇒ broken discrete logarithm
C sends (g, h) to B
A sends m to B (due to 1-naCMA)
B chooses random s and calculates c = g^m ∗ h^s
sends pk=(g,h,c) and s to A
A sends (m∗, s∗) to B
when A succeeds, g^m∗ ∗ h^s∗ = g^m = h^s
B outputs x of m∗ + x ∗ s∗ = m + x ∗ s

broken with second signature
can extract x with two signatures of different messages
g^{m1} ∗ h^{s1} = g^{m2} ∗ h^{s2} for h = g^x

5.4 RSA signature

EUF-1-naCMA using RSA

construction
choose N = PQ for P, Q primes, e > 2^n with ggt(e, phi(N)) = 1
calculate d = e^{-1} mod phi(N)
choose random J,c
Gen(2^k) → pk = (N, e, J, c), sk = d
Sign(sk, m) → s = (c / J^m)ˆd mod N
Vfy(pk, m, s) → c = J^m ∗ s^e mod N
works bc c = (c/J^m)ˆ(d∗e) ∗ J^m = (c/J^m)ˆd

broken EUF-1-naCMA ⇒ broken RSA
C chooses random y
C sends (y, N, e) to B
A sends m to B (due to 1-naCMA)
B chooses random s, sets J = y and calculates c = J^m ∗ s^e mod N
sends pk=(N, e, J, c) and s to A
A sends (m∗, s∗) to B
when A suceeds, J^m∗ ∗ s∗ˆe = J^m ∗ s^m = c
B outputs x with shamirs trick on Jˆ(m-m∗) = (s∗ / s)ˆe
(with case distinction on m > m∗ or m < m∗)

broken with second signature
can extract x with two signatures of different messages
J^{m1} ∗ $s1^e$ = J^{m2} ∗ $s2^e$; use shamir's trick

6 constructions using one-time signatures

6.1 q-time signatures

convert one-time signatures into multiple-time signatures

naive
state consists of counter, initialized with 1
Gen(1^k) generates q (pk_i, sk_i)
pk = (pk_1, pk_2, ...), sk = (sk_1, sk_2, ...)
Sign(sk_i, m_i) = s_i & counter is increased
Vfy(pk_i, m_i, s_i) ?= 1
|pk| = |sk| = O(q), |s| = O(1)

hashed pk
set pk = (H, H(pk_1, pk_2, ...)) for H coll'resistant hash function
Sign appends all pk_i; hence s_i = (s_i, i, (pk_1, pk_2, ...)
Vfy(pk_i, s_i, m_i) ?= 1 and y = H(pk_1, pk_2, ...)
|pk| = O(1), |sk| = |s| = O(q)

merkle tree pk
pk is root of merkle tree
signature contains co-path values to do the merkle hashing
|pk| = O(1), |sk| = O(q), |s| = O(log q)

merkle tree with OST pk
start with public key of OTS as root
parent signs public key of children as soon as needed
improves runtime because tree can e built "lazily"

pseudo-random function sk
transform probabilistic Gen(1^k) to deterministic with random input
generate random input with pseudo-random function & random seed s
now only need to send seed instead of all public keys
|pk| = O(1), |sk| = O(1), |s| = O(log q)

stateless
avoid having to know which key has to be used next
instead choose random or one determined by message
cache calculated keys to avoid recomputation upon each signature

6.2 EUF-CMA

construct out of EUF-1-naCMA and EUF-naCMA

construction
let E_1 be EUF-1-naCMA signature, E' EUF-naCMA
Gen(1^k) = Gen'(1^k)
Sign(sk, m) generates (sk_1, pk_1) with Gen1(1^k)
computes s_1 = Sign1(sk_1, m)
computes s' = Sign'(sk, pk_1)
outputs s = (pk_1, s_1, s')
Vfy(pk, m, s) = Vfy'(pk, pk_1, s') && Vfy1(pk_1, m, s_1)

broken EUF-CMA ⇒ broken EUF-1-naCMA or broken EUF-naCMA
E_0 if (m∗, s∗) has reused pk_1 (broken EUF-1-naCMA)
E_1 else (broken EUF-naCMA)
either E_0 or E_1 happen, hence either Pr[E_0] or Pr[E_1] > e_A / 2
(to break EUF-1-naCMA, case E_0)
B generates (sk', pk') = (sk, pk)
B uses chooses random index v < q
B sends pk to A
A queries for m_i to be signed
B uses C at index v to get (pk_1, s_1) of m_v
else B works as honest challenger to A
when A succeeds with s∗ = (pk_1∗, s_1∗, s'∗)
B outputs (m∗, s_1∗) if same pk_1 reused than received from C
hence success rate $e_B \geq e_A$ / 2q (for 1/q same reused)
(to break EUF-naCMA, case E_1)
B generates q (sk_1, pk_1) one-time signature pairs
B uses C to get signatures s'_i for all $pk1_i$ and the pk
B sends pk to A
A queries for m_i to be signed
B answers ($pk1_i$, Sign1($sk1_i$, m_i), s'_i (of C))
when A succeeds with s∗ = (pk_1∗, s_1∗, s'∗)
B outputs (pk_1∗, s'∗)
hence success rate B is e_A / 2

7 RSA signatures

sign exponential number of messages

7.1 schoolbook ("naive")

UUF-NMA using RSA

construction
choose N = PQ for P, Q primes, e > 2^n with ggt(e, phi(N)) = 1
calculate d = e^{-1} mod phi(N)
Gen(2^k) → pk = (N, e), sk = d
Sign(sk, m) → s = m^d mod N
Vfy(pk, m, s) → m = s^e mod N
works bc m = mˆ(d∗e) = s^e

multiplicative homomorphic
if s_1, s_2 are valid for m_1, m_2
then s_3 = s_1∗s_2 is valid for m_1∗m_2

broken EUF-NMA
attacker simply chooses any signature s∗

calculates message m∗ = s^e

broken UUF-CMA
A reveives m∗ from C
A chooses random x != 1
calculates y = x^e mod N
calculates m_1 = m∗ ∗ y mod N
C signs m_1 to s_1
B calculates s∗ = s_1 ∗ x^{-1}
works bc s∗ ˆ e = $(s_1 * x^{-1})$ˆe = m∗ ∗ y ∗ y^{-1} = m∗
⇒ homomorphy allows this attack

broken UUF-NMA ⇒ broken RSA
C sends (N, e, y) of x^e mod N to B
B sends pk = (N, e) and m∗ = y to A
A answers with s∗ such that s∗ˆe = m∗
B outputs s∗ = x

7.2 RSA public key cryptography standard #1 (RSA PKCS #1)

like textbook RSA, but hashes messages
used in practice but unclear security (no attacks & no security proof)

hash function h
for H collision resistant
m = 0x00 | encoding | padding | boundary | H type | H(m)
like 0x00 | 0x01 | 0xFF ... 0xFF | 0x00 | 0x01 | H(m)

construction
choose N = PQ for P, Q primes, e > 2^n with ggt(e, phi(N)) = 1
calculate d = e^{-1} mod phi(N)
Gen(2^k) → pk = (N, e), sk = d
Sign(sk, m) → s = h(m)ˆd mod N
Vfy(pk, m, s) → h(m) = s^e mod N
works bc h(m) = h(m)ˆ(d∗e) = s^e

7.3 RSA full-domain hash (RSA-FDH)

EUF-CMA using ROM & RSA
like textbook RSA, but hashes messages (destroys homomorphism)

construction
choose N = PQ for P, Q primes, e > 2^n with ggt(e, phi(N)) = 1
choose H = hash function
calculate d = e^{-1} mod phi(N)
Gen(2^k) → pk = (N, e, H), sk = d
Sign(sk, m) → s = H(m)ˆd mod N
Vfy(pk, m, s) → H(m) = s^e mod N
works bc H(m) = H(m)ˆ(d∗e) = s^e

requirements H
must be collision resistant (else breaking trivial)
must destroy homomorphy (else equal to naive construction)
use hash function as given by random oracle model

broken EUF-CMA with q ROM ⇒ broken RSA
E_0 if attacker never asked RO, else E_1
either E_0 or E_1 happen, hence either Pr[E_0] or Pr[E_1] > e_A / 2
(case E_0)
hash value must be chosen at random
hence success p > 1/N
(case E_1)
C sends (N, e, y) of x^e mod N to B
B uses chooses random index v < q
B sends pk to A
A queries for m_i to be hashed
if i == v, then return y
else B chooses random x_i and retuns y_i = x_iˆe mod N
A queries for m_i to be signed
if i == v, then B has to abort bc can not create signature
else B returns x_i (works because x_iˆe = H(x_i))
A returns (m∗, s∗) such that m∗ = m_v
B outputs s∗ ⇒ s∗ˆe = y
only works if A does not ask for signature for m_v
hence success p > Pr[E_1]/q

7.4 RSA probabilistic signature scheme (RSA-PSS)

EUF-CMA using ROM, RSA
like textbook RSA, but preprocesses m
better security reduction than RSA-FDH
used in PKCS#1 version 2.1 (June 2002)

PSS-Encode
needs parameters k_0, k_1
needs hash function H {0,1}∗ → {0,1}ˆk_1
needs hash function G {0,1}ˆk_1 → {0,1}ˆ(k-k_1-1)
G separated in G_1 (first k_0 bits) and G_2 (rest)
choose random r ← {0,1}ˆk_0
w = H(m|r)
r∗ = G_1(w) XOR r
y = G_2(w)
outputs 0 || w || r∗ || y
gives many different encodings for simple message

PSS verify
ensure first bit of value is 0
split value into parts (w, r∗, y)
compute r = G_1(w) XOR r∗
ensure y = G_2(w) and w = H(m || r)

construction
choose N = PQ for P, Q primes, e > 2^n with ggt(e, phi(N)) = 1
choose H = hash function
calculate d = e^{-1} mod phi(N)
Gen(2^k) → pk = (N, e, H), sk = d
Sign(sk, m) → s = PSS-Encode(m)ˆd mod N
Vfy(pk, m, s) → y = s^e mod N && check y valid encoding of m
works bc PSS-Encode(m) = PSS-Encode(m)ˆ(d∗e) = s^e

proof scetch
similar to RSA-FDH; but can embedd RSA in every hash query
answer all signature queries by reencoding message with fresh r
hence B answers all signature queries, and A always solves RSA
(B knows only in reduction only single signature per message)
leads to tighter security reduction

vs RSA-FDH
tighter security reduction but 2 hash computations per signature
RSA-PSS in practice more efficient

7.5 gennaro-halevi-rabin (GHR)

EUF-naCMA using sRSA
proof in standard model, but needs stronger assumption

construction
choose N = PQ for P, Q primes
choose hash function h mapping bit strings to primes > N
(> N enforces gcd(h(m), phi(N)) = 1, hence inverse h(m) exists)
choose random r
Genk(1^k) → pk(N, r, h), sk(phi(N))
Sign(sk, m) → s = rˆ(1/h(m)) mod N
Vfy(pk, m, s) → sˆh(m) = r mod N
works bc sˆh(m) = rˆ(1/h(m) ∗ h(m)) = r

broken EUF-naCMA → broken coll'resistance h or broken strong RSA
E_0 if (m∗, s∗) with some h(m_i) = h(m∗) (broken coll'resistance h)
E_1 else (broken strong RSA)
either E_0 or E_1 happen, hence either Pr[E_0] or Pr[E_1] > e_A / 2
(to break coll'resistance h, case E_0)
C sends h to B
A sends m_1, m_2, ... to B
B generates pk and sk
B signs to s_1,s_2,...
B sends (s_1, s_2, ...) and pk to A
A responds with (m∗, s∗) such that h(m∗) = h(m_i)
B outputs (m∗, m_i) as collision of h
(to break strong RSA, case E_1)
C sends (N, y) to B such that x^e = y mod N
A sends m_1, m_2, ...
B calculates r = yˆ$\prod$(h(m_1), h(m_2), ...)
B calculates s_i = yˆ$\prod$(h(m_1), h(m_2), ...) without h(m_i)
B sends pk = (N, r, h) and (s_1, s_2, ...) to A
when A succeeds, returns (m∗, s∗) for h(m∗) not previously seen
it holds that s∗ˆh(m∗) = r = yˆ$\prod$(h(m_1), h(m_2), ...)
B uses shamirs trick with J = y, S=s∗ to get xˆh(m∗) = y mod N
B outputs (x, h(m∗)) such that xˆh(m∗) = y mod N

7.6 constructions

EUF-CMA using RSA assumption
use construction presented for one-time signatures
use GHR as E' EUF-naCMA under strong RSA assumption
use RSA one-time signatures as E_1 EUF-1-naCMA under RSA assumption

using (non-strong) RSA assumption
(only proof overview given)
GHR is SUF-naCMA under RSA assumption
transform SUF-naCMA to EUF-naCMA
leads to Hohenbergers-Waters signatures
transform EUF-naCMA to EUF-CMA
receive compact, but inefficient signatures

8 chamäleon hash functions

verify authenticity of message only to receiver

8.1 definition

Gen(1^k) generates (ch, pi)
for ch : M x R → N & pi is trapdoor
TrapColl(pi, m, r, m∗) → r∗ such that ch(m,r) = ch(m∗,r∗)
without pi, ch is collisionresistant
ch collision resistant when Pr[A(1^k, ch) = (m,r, m∗,r∗) : ch(m, r) = ch(m∗,r∗) ˆ (m,r) != (m∗,r∗)] ≤ negl(k)

8.2 discrete logarithm

construction
let G group with generator g of order prime p
ch : Z_p x Z_p → G
Gen(1^k) chooses x ∈ Z_p, calculates h = g^x
ch defined by (g,h) with trapdoor x; ch(m, r) = $g^m * h^r$
TrapColl(x, m, r, m∗) = r∗ = (m - m∗) / x + r mod p
works because ch(m, r) = g^m ∗ gˆ(x∗r) ⇒ m + x∗r

reduction
C sends g, y of g^x mod p = y
B sends g, h=y to A
A responds with (m, r, m∗, r∗)
B can calculate x because $g^m * y^r = g^m * g^{xr}$

8.3 RSA

construction
choose N = PQ for P, Q primes, e > 2^n with ggt(e, phi(N)) = 1
calculate d = e^{-1} mod phi(N)
choose random J
Gen(1^k) outputs (N, e, J)
ch defined by (J,e) with trapdoor d; ch(m, r) = $J^m * r^e$
TrapColl(x, m, r, m∗) = r∗ = (Jˆ(m - m∗) ∗ r^e)ˆd (bc r^e∗d = r^1)

reduction
C sends (e, y, N) of x^e mod N = y
B sends (N, e, J=y) to A
A responds with (m, r, m∗, r∗)
B can calculate x because $J^m * r^e = x^{em} * r^e$

9 signatures using chamäleon hash

when signing & verifying, use chamäleon function of receiver
others not convinced by signature, bc receiver knows trapdoor (repudiation)
signer should execute zero-knowledge proof of knowledge
so malicious receiver can not fake knowledge of trapdoor

9.1 security experiments

EUF-CMA game
C generates (sk, pk) and (ch, pi)
C sends pk, ch to A
A asks for q messages one after the other to be signed
C sends signature for each message
A responds with (m∗, s∗)
A wins if Vfy(pk, m∗, s∗, ch) = 1

EUF-CMA game analysis
adversary must use receiver delivered ch in signing
unrealistic, hence notion not strong enough
for dlog ch, assume C sends ch=(g,h) to A
constructs $ch_A = (g^a$, h)
then A lets C sign message m under ch_A to s
then can output (m∗a, s) as valid signature under ch

9.2 EUF-CMA

EUF-CMA with chamäleon

construction
for signature algorithm E'
Gen(1^k) = Gen'(1^k) to generate (pk, sk)
Sign(sk, m, ch) chooses seed r
then calculates m' = ch(m,r) and s' = Sign'(sk, m')
outputs s = (s', r)
Vfy(pk, m, s, ch) checks Vfy'(pk, ch(m, r), s')

broken EUF-CMA ⇒ broken coll'resistance ch or broken EUF-naCMA
E_0 if ch(m∗, r∗) is equal to some other ch already seen
E_1 else (hence EUF-naCMA directly broken)
either E_0 or E_1 happen, hence either Pr[E_0] or Pr[E_1] > e_A / 2
(to break coll'resistance, case E_0)
C sends ch to B
B generates (sk, pk)
B sends pk to A
A queries m_i to be signed
B signs
when A succeeds with s∗ = (s∗', r∗) and m∗
B outputs (m∗, r∗, m_i, r_i) for ch(m_i, r_i) = ch(m∗, r∗)
(to break EUF-naCMA, case E_1)
B generates (pi, ch) and q random values (m_{ib}, r_{ib})
B calculates y_{ib} = ch(m_{ib}, r_{ib})
B sends y_{ib} to C
C returns pk and signatures s_ib' for all y_{ib}
B now starts A with pk
A queries m_i to be signed
B uses TrapColl(pi, m_{ib}, r_{ib}, m_i) = r_i
B signs with s_i = (s_ib' (of C), r_i)
when A succeeds with s∗ = (s'∗, r∗) and m∗
B outputs (ch(m∗, r∗), s'∗)

9.3 EUF-1-naCMA

EUF-1-naCMA with chamäleon

construction
for E_{ch} chamäleon
Gen(1^k) generates (ch, pi)
chooses random (˜m, ˜r) ∈ M x R
calculates c = ch(˜m, ˜r)
outputs pk = (ch, c), sk = (pi, ˜m, ˜r)
Sign(sk, m) uses TrapColl(pi, ˜m, ˜r, m) to get r
Vfy(pk, m, s) checks ch(m, r) = c

broken EUF-1-naCMA ⇒ broken coll'resistance ch
C sends ch to B
A sends m to B
B chooses random r, calculates c = ch(m, r)
B sends (ch, c) to A
when A succeeds with (m∗, r∗) for m∗!= m and ch(m∗, r∗) = c
B outputs (m∗, r∗, m, r)

broken sEUF-1-naCMA ⇒ broken coll'resistance ch
same proof than before
A must answer with different r or m to win game
with that ch coll'resistance is already broken

9.4 sEUF-CMA

sEUF-CMA with chamäleon

construction
let E' be EUF-CMA and CH be chamäleon hash function
Gen(1^k) generates (pk', sk') ← Gen'(1^k)
two chamäleon signatures ch_F with pi_F and ch_H with pi_H
get pk = (pk', ch_F, ch_H), sk = (sk', pi_H)
Sign(sk, m) chooses random r_F, r_H, arbitrary m', s'
calculate h = ch_H(m' | s', r_H')
calculate ˜m = ch_F(h, r_F)
calculate ˜s = Sign'(sk', ˜m)
r_H ← TrapColl(pi_H, m' | s', r_H', m | ˜s)
s = (˜s, r_F, r_H)
Vfy(pk, m, s) = Vfy'(pk', ˜m, ˜s)
for h = ch_F(m | ˜s, r_H)
get ˜m = ch_f(h, r_F)

broken sEUF-CMA ⇒ broken coll'resistance ch or broken EUF-CMA
E_0 if forgery contains reused ˜m∗ (some ch coll'resistance broken)
E_1 else (hence EUF-CMA directly broken)
either E_0 or E_1 happen, hence either Pr[E_0] or Pr[E_1] > e_A / 2
same proof as already seen

for E_0, simply guess which ch is broken; use trapdoor for other

10 pairing-based signatures

10.1 pairing

for cyclic groups G_1, G_2, G_T of order p
a pairing is a mapping e : G_1 x $G_2 \to G_T$
G_1, G_2 called source groups
G_T usually subgroup of finite field; called target group

with properties
bilinearity (e(g_1 * g1', g_2) $\Rightarrow$ e(g_1, g_2) * e(g1', g_2) and vice versa)
non-degeneracy (e(g_1, g_2) != 1)
e is efficiently computable

types
type 1, symmetric ($G_1 = G_2$)
type 2, asymmetric but efficient homomorphism
type 3, asymmetric & no efficient homomorphism

usages
push problem from G_1 to G_T so it might be easier
generalize to multilinear maps for many more applications

self-bilinear map breaks CDH
e(g, g) = g^a; hence need to know g^1/a
then calc e(g^{xa}, g^(y*a^1)) = g^{xy}
to calculate g^1/a need to know group order
then use square-and-mult to find (g^a)^(p-3) = (g^a)^-2

10.2 joux's 3 party diffie hellmann

construction
for e : G x G $\to G_T$ symmetric pairing
for g generator of group G of order p
A chooses a, B chooses b, C chooses c
A sends g^a to others, B sends g^b to others, C ...
A calculates k = e(g^b, g^c)^a = e(g, g)^abc
hence all parties get same key

10.3 boneh-lynn-shacham (BLS) signatures

EUF-CMA using ROM, CDH

construction
Genk(1^k) chooses x and calculates g^x
pk = (g, g^x), sk = (x)
Sign(sk, m) $\to$ s = H(m)^x
Vfy(pk, m, s) = e(H(m), g^x) = e(s, g)
works because e(H(m), g^x) = e(H(m)^x, g)

broken EUF-CMA with q ROM $\Rightarrow$ broken CDH
E_0 if attacker never asked RO, else E_1
either E_0 or E_1 happen, hence either Pr[E_0] or Pr[E_1] $> e_A$ / 2
(case E_0)
hash value must be chosen at random
hence success p > 1/N
(case E_1)
B receives (g, g^x, g^y) from C
B chooses random index v < q
B sends pk = (g, g^x) to A
A queries for m_i to be hashed
if i == v, then return g^y
else B chooses random x_i and retuns y_i = g^x_i
A queries for m_i to be signed
if i == v, then B has to abort bc can not create signature
else B returns (g^x)^x_i (works because e(g, (g^x)^x_i) = e(g^x, g^x_i))
A returns (m*, s*) such that m* = m_v = g^y
B outputs s* $\Rightarrow$ s = (g^y)^x
only works if A does not ask for signature for m_v
hence success p > Pr[E_1]/q

aggregate
for U_1, U_2, ... senders with messages m_1, m_2, ... respectively
without aggregation, need also to transfer s_1, s_2, ... respectively
aggregator calculates s = $\prod s_i$
verifies checks that e(s, g) = $\prod$ e(H(m_i), g^x_i)
works bc s = $\prod$ H(m_i)^x_i
n+1 parings instead of 2n without aggregation
good bc saves bandwidth, computations

batch verification
for U sender with message/signature pairs (m_1, s_1), ...
without batching, need to verify each pair
batch verificator calculates h = $\prod$ H(m_i) and s = $\prod s_i$
and checks that e(g, s) = e(g^x, h)
works bc s = (H(m_1) * H(m_2) * ...)^x

10.4 waters (1,q,y)-PHF

using CDH

construction
Gen(1^k) chooses u_0, ..., u_l from G
Eval(k, m as bits m_1 ... ml) computes H_k(m) = u_0 * $\prod$ u_i^m_i
for q = q(k) polynomial y = 1 /O(q * sqrt(k))
TrapGen randomly chooses a_i = random walks, b_i such that u_i = h^a_i * g^b_i
hence k = (u_0, ..., u_l) and t = (a_0, ..., a_l, b_0, ..., b_l)
TrapEval(t, m as bits m_1 ... ml)
computes $a_m = a_0 + \sum m_i\ a_i$ and same for b_m
works bc h^a_m * g^b_m = h^a_0 * $\prod$ ... * g^h_0 * $\prod$ ... = u_0 * $\prod$ ... = H_k(m)

well-distributedness
distribution of k_TrapGen equal to k_{Gen} bc randomly chosen b_i
hence u_i is randomly distributed bc made out of g^b_i
a_i is composed of random walks of length q^2
a_i is a random walk of q^2 (all 0s) < length < k * g^2 (all 1s)
hence O(1/sqrt(k)*q) $\leq$ Pr[a_i is 0] $\leq$ O(1/q)
Pr[a_m != 0 | a_m* = 0] > 1 - 1/2q for m != m*
hence Pr[a_{mi} != 0 | a_{mi}* = 0] $\geq$ 1/2
hence Pr[a_{mi} != 0 ^ a_m* = 0] $\geq$ 1/O(q*sqrt(k))
hence a_i = 0 with probability 1/O(sqrt(k)*q))

10.5 waters signatures

EUF-CMA using CDH
less efficient than BLS (+1 group element) but proof in standard model

construction
for G and G_T prime order groups with order p and q respectively
for generator g of G, for (Gen, Eval) GHF to G
Gen(1^k) chooses random g^a, calculates e(g, g^a)
generate k $\leftarrow$ Gen_GHF(g) describing H
let pk = (g, k, e(g, g)^a), sk = (g^a)
Sign(sk, m) chooses random r
s = (s_1 = g^r, s_2 = g^a * H(m)^r)
Vfy(pk, m, s) = e(g, g)^a * e(s_1, H(m)) = e(g, s_2)
works bc e(g, g^a) * e(g, H(m))^r = e(g, g^a * H(m)^r)

broken EUF-CMA with (1,q,y)-PHF $\to$ broken CDH
C sends (g, g^x, g^y) to B
B generates (k, t) $\leftarrow$ TrapGen(g, g^x)
B sends pk = (g, k, e(g^x, g^y)) to A
(hence sk = g^{xy}, the value B needs to have)
A queries m_i to be signed
B calculates (a_i, b_i) $\leftarrow$ TrapEval(t, m_i)
if (a_i != 0) then can generate signature
B chooses random si
calculates s_{i1} = (g^y)^(-1/ai) * g^{si} (unformly distributed)
calculates s_{i2} = (g^x)^(a_i * s_i) * (g^y)^(-b_i / a_i) * g^(b_i * s_i)
A answers (m*, s*)
if (a* == 0) then can extract g^{xy}
bc H(m*) = (g^x)^0 * g^b* = g^b*
g^{xy} = s_2* * (s_1*)^-b* = g^{xy} * g^(b* * r*) * g^(r* * -b*)
hence B can return g^{xy}

rerandomize
can recompute signature without sk for r' = r + t
s' = (s1' = g^r * g^t, s_2 = g^a * H(m)^r * H(m)^t)

11 appendix

current research
leakage resilience (attacker sees parts of sk)
functional signatures (sk limited to certain m)
aggregatable signatures (many s to single one)
key infrastructures (secret-key, identity-based, certificateless)