A state-of-the-art public-key signature system

D. J. Bernstein

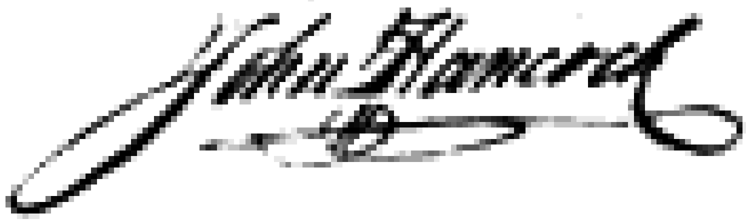
Thanks to:
University of Illinois at Chicago
NSF CCR–9983950
Alfred P. Sloan Foundation

Handwritten signatures

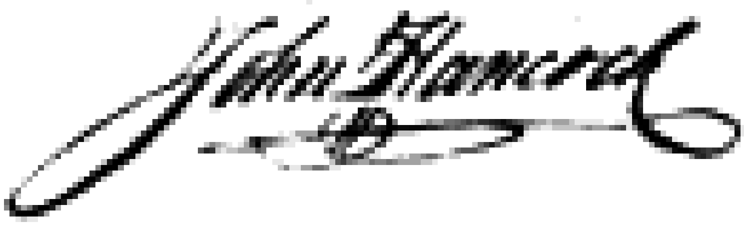
Want to transmit message:

"Pay $1000."

Sender attaches his signature:

"Pay $1000.  "

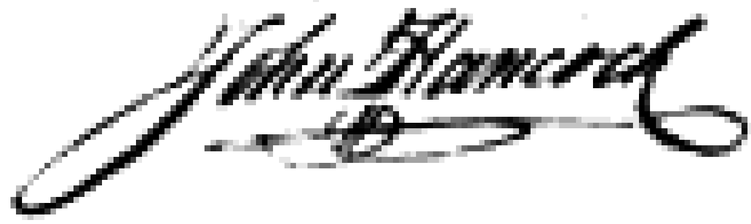
Recipient checks sender's signature:

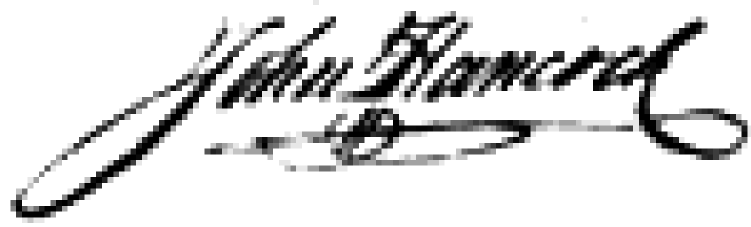
"  "

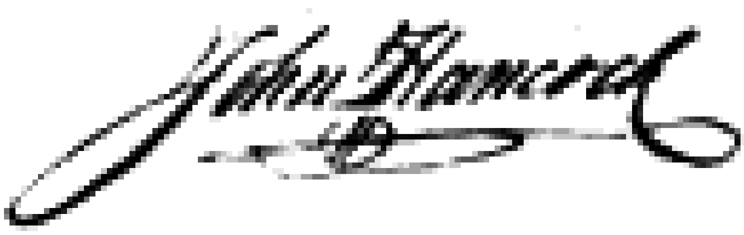
Recipient accepts message:

"Pay $1000."

Forging signed messages

Attacker intercepts
the signed message:
"Pay $1000.  "

Attacker modifies the message:
"Pay $3000.  "

Recipient checks sender's signature:
"  "

Recipient accepts message:
"Pay $3000."

How do we stop forgeries?

The signature has to depend on the message. Define set V of valid signed-message pairs (m, s).

Sender, given m, must be able to generate s such that $(m, s) \in V$.

Recipient must be able to check whether $(m, s) \in V$.

Attacker, given $(m, s) \in V$, must not be able to find $(m', s') \in V$ with $m' \neq m$.

Public-key signature systems

Sender has a **secret key**
and a **public key**.
Recipient knows the public key.

Sender uses secret key to
find $(m, s) \in V$, given m.

Recipient uses public key
to check whether $(m, s) \in V$.

Hopefully attacker can't
figure out secret key,
and can't figure out (m', s')
without secret key.

A state-of-the-art signature system

Sender's public key is an integer k with $2^{1536} < k < 2^{1537}$.
More restrictions on k, discussed later.

$(m, e, f, r, s) \in V$ iff $e \in \{-1, 1\}$, $f \in \{1, 2\}$, $r \in \{0, 1, \ldots, 15\}$, $s \in \left\{0, 1, \ldots, 2^{1536} - 1\right\}$, and $efs^2 - H_0(r, m) \in k\mathbf{Z}$.

$H_0 : \{\text{strings}\} \to \left\{1, 2, \ldots, 2^{1536}\right\}$ is a complicated public function, discussed later.

Given m, e, f, r, s, recipient computes $efs^2 - H_0(r, m)$, divides by k, checks that remainder is 0.

Attacker might select e', f', s', compute remainder $e'f'(s')^2 \bmod k$, hope to invert H_0 to find (r', m'); but we conjecture that inverting H_0 is difficult.

Attacker might select r', m', compute $H_0(r', m')$, hope to find square root modulo k; but we conjecture that finding square roots is difficult.

Square roots modulo *primes*
are easy to compute—
but k will never be prime.

Particularly easy
for primes $p \in 3 + 4\mathbf{Z}$:
Given $i^2 \bmod p$, compute
$i^4 \bmod p = (i^2 \bmod p)^2 \bmod p$,
$i^6 \bmod p = (i^4 \bmod p)(i^2 \bmod p) \bmod p$,
$i^{12} \bmod p = (i^6 \bmod p)^2 \bmod p$,
$\ldots$, $i^{(p+1)/2} \bmod p$.
By Fermat's little theorem, this is
a square root of i^2 modulo p.

About $\lg p$ multiplications.

Sender's secret key is (p, q, z)
where z is a 256-bit string,
p is prime, q is prime,
$p \in 3 + 8\mathbf{Z}$, $q \in 7 + 8\mathbf{Z}$,
$2^{767} < p < 2^{768} < q < 2^{769}$,
and $pq = k$.

Sender finds square roots mod k
using factorization of k.

Attacker isn't given factorization,
and conjecturally can't do this.

Given m, sender computes

- $r = H_1(z, m)$;
- $h = H_0(r, m)$;
- $e = 1$ if h is a square modulo q, otherwise $e = -1$;
- $f = 1$ if eh is a square modulo p, otherwise $f = 2$;
- $s =$ the unique square root of eh/f modulo pq with $s \in \{0, 1, \ldots, (pq-1)/2\}$ and with $\pm s$ a square modulo pq.

Signed message is (m, e, f, r, s).

H_1 is another public function.

The hash functions H_0, H_1

Start from this $\rightarrow$ function SHA. String input, 160-bit output.

Define $H_0(x) = 6 + 2^{96}\,\text{SHA}(1, x) + 2^{256}\,\text{SHA}(2, x) + \cdots + 2^{1376}\,\text{SHA}(9, x)$.

Define $H_1(x) = \text{SHA}(0, x) \bmod 16$.

```
static u_int32_t _K[] = { 0x5a827999, 0x6ed9eba1, 0x8f1bbcdc, 0xca62c1d6 };
#define K(t)    _K[(t) / 20]
#define F0(b, c, d)     (((b) & (c)) | ((~(b)) & (d)))
#define F1(b, c, d)     (((b) ^ (c)) ^ (d))
#define F2(b, c, d)     (((b) & (c)) | ((b) & (d)) | ((c) & (d)))
#define F3(b, c, d)     (((b) ^ (c)) ^ (d))
#define S(n, x)         (((x) << (n)) | ((x) >> (32 - n)))
#define H(n)    (ctxt->h.b32[(n)])
#define COUNT   (ctxt->count)
#define BCOUNT  (ctxt->c.b64[0] / 8)
#define W(n)    (ctxt->m.b32[(n)])
#define PUTBYTE(x)      { \
        ctxt->m.b8[(COUNT % 64)] = (x);         \
        COUNT++;                                \
        COUNT %= 64;                            \
        ctxt->c.b64[0] += 8;                    \
        if (COUNT % 64 == 0)                    \
                sha1_step(ctxt);                \
     }
#define PUTPAD(x)       { \
        ctxt->m.b8[(COUNT % 64)] = (x);         \
        COUNT++;                                \
        COUNT %= 64;                            \
        if (COUNT % 64 == 0)                    \
                sha1_step(ctxt);                \
     }

static void sha1_step __P((struct sha1_ctxt *));

static void
sha1_step(ctxt)
        struct sha1_ctxt *ctxt;
{
        u_int32_t       a, b, c, d, e;
        size_t t, s;
        u_int32_t       tmp;

        a = H(0); b = H(1); c = H(2); d = H(3); e = H(4);

        for (t = 0; t < 20; t++) {
                s = t & 0x0f;
                if (t >= 16) {
                        W(s) = S(1, W((s+13) & 0x0f) ^ W((s+8) & 0x0f) ^ W((s+2) & 0x0f) ^ W(s));
                }
                tmp = S(5, a) + F0(b, c, d) + e + W(s) + K(t);
                e = d; d = c; c = S(30, b); b = a; a = tmp;
        }
        for (t = 20; t < 40; t++) {
                s = t & 0x0f;
                W(s) = S(1, W((s+13) & 0x0f) ^ W((s+8) & 0x0f) ^ W((s+2) & 0x0f) ^ W(s));
                tmp = S(5, a) + F1(b, c, d) + e + W(s) + K(t);
                e = d; d = c; c = S(30, b); b = a; a = tmp;
        }
        for (t = 40; t < 60; t++) {
                s = t & 0x0f;
                W(s) = S(1, W((s+13) & 0x0f) ^ W((s+8) & 0x0f) ^ W((s+2) & 0x0f) ^ W(s));
                tmp = S(5, a) + F2(b, c, d) + e + W(s) + K(t);
                e = d; d = c; c = S(30, b); b = a; a = tmp;
        }
        for (t = 60; t < 80; t++) {
                s = t & 0x0f;
                W(s) = S(1, W((s+13) & 0x0f) ^ W((s+8) & 0x0f) ^ W((s+2) & 0x0f) ^ W(s));
                tmp = S(5, a) + F3(b, c, d) + e + W(s) + K(t);
                e = d; d = c; c = S(30, b); b = a; a = tmp;
        }

        H(0) = H(0) + a;
        H(1) = H(1) + b;
        H(2) = H(2) + c;
        H(3) = H(3) + d;
        H(4) = H(4) + e;

        bzero(&ctxt->m.b8[0], 64);
}

void
sha1_init(ctxt)
        struct sha1_ctxt *ctxt;
{
        bzero(ctxt, sizeof(struct sha1_ctxt));
        H(0) = 0x67452301;
        H(1) = 0xefcdab89;
        H(2) = 0x98badcfe;
        H(3) = 0x10325476;
        H(4) = 0xc3d2e1f0;
}

void
sha1_pad(ctxt)
        struct sha1_ctxt *ctxt;
{
        size_t padlen;          /*pad length in bytes*/
        size_t padstart;

        PUTPAD(0x80);

        padstart = COUNT % 64;
        padlen = 64 - padstart;
        if (padlen < 8) {
                bzero(&ctxt->m.b8[padstart], padlen);
                COUNT += padlen;
                COUNT %= 64;
                sha1_step(ctxt);
                padstart = COUNT % 64;  /* should be 0 */
                padlen = 64 - padstart; /* should be 64 */
        }
        bzero(&ctxt->m.b8[padstart], padlen - 8);
        COUNT += (padlen - 8);
        COUNT %= 64;
        PUTPAD(ctxt->c.b8[0]); PUTPAD(ctxt->c.b8[1]);
        PUTPAD(ctxt->c.b8[2]); PUTPAD(ctxt->c.b8[3]);
        PUTPAD(ctxt->c.b8[4]); PUTPAD(ctxt->c.b8[5]);
        PUTPAD(ctxt->c.b8[6]); PUTPAD(ctxt->c.b8[7]);
}

void
sha1_loop(ctxt, input, len)
        struct sha1_ctxt *ctxt;
        const u_int8_t *input;
        size_t len;
{
        size_t gaplen;
        size_t gapstart;
        size_t off;
        size_t copysiz;

        off = 0;

        while (off < len) {
                gapstart = COUNT % 64;
                gaplen = 64 - gapstart;

                copysiz = (gaplen < len - off) ? gaplen : len - off;
                bcopy(&input[off], &ctxt->m.b8[gapstart], copysiz);
                COUNT += copysiz;
                COUNT %= 64;
                ctxt->c.b64[0] += copysiz * 8;
                if (COUNT % 64 == 0)
                        sha1_step(ctxt);
                off += copysiz;
        }
}

void
sha1_result(ctxt, digest0)
        struct sha1_ctxt *ctxt;
        caddr_t digest0;
{
        u_int8_t *digest;

        digest = (u_int8_t *)digest0;
        sha1_pad(ctxt);
        bcopy(&ctxt->h.b8[0], digest, 20);
}
```

Many other possibilities.

General belief: Almost *every* reasonably-easy-to-compute function is safe.
Can choose a function randomly!

Some varieties of functions seem safe at higher speeds.

But nothing has been proven.

Wang et al. 2004 constructed collision in popular function MD5: m, m' with $\text{MD5}(m) = \text{MD5}(m')$.

Some credits

Concept of public key signatures:
1976 Diffie Hellman. No examples.

$s^{\text{something}} \bmod k = m$:
1977 Rivest Shamir Adleman;
independently Rabin, unpublished.
Bad system: allows trivial forgeries.

$s^{\text{something}} \bmod k = H_0(m)$:
1979 Rabin. Seems to be secure.

Small exponent: 1979 Rabin.
Saves verification time.

$s^2 \bmod k = H_0(m)$: 1979 Rabin.
Saves more time. Adds problem:
$H_0(m)$ has only 25% chance
of being a square modulo k.

$s^2 \bmod k = H_0(r, m)$, with r chosen
randomly by signer: 1979 Rabin.
Fixes the problem,
if r has enough bits.

Choosing r as secret function of m,
i.e., function of z and m: 1997
Barwood; independently Wigley.
Eliminates randomness from signing.

Extra factors

$e \in \{-1, 1\}$, $f \in \{1, 2\}$,
with $p \in 3 + 8\mathbf{Z}$, $q \in 7 + 8\mathbf{Z}$:
1980 Williams. Now efs^2 covers
all integers mod k, so no need
to try more than one r.

Can even omit r.
We'll see later why
state-of-the-art system
includes 4-bit r.

Security

An **attack** is an algorithm.
Algorithm receives public key k.
Algorithm selects message m_0,
receives signature s_0 of m_0.
Algorithm selects message m_1,
receives signature s_1 of m_1.
Et cetera.
Algorithm then prints (m', s').

Attack is **successful** if
$m' \notin \{m_0, m_1, \ldots\}$
and s' is a signature of m'.

Conjecture: Every fast attack has negligible chance of success against a random public key.

(Typical formalization: Every attack using $\leq 2^{60}$ steps on a 2-tape Turing machine has probability at most 2^{-30} of success.)

Of course, real signers restrict m_0, m_1, etc. Restricted conjecture: Every fast restricted attack has negligible chance of success against a random public key.

Best attack method we know:
Factor public key k
to discover p and q.
Then choose m' and compute
s' the same way sender does.

Best factorization method we know:
number-field sieve (NFS).
(1988 Pollard, et al.)

Some successful factorizations
of 512 bits and slightly beyond,
but nowhere near 1536 bits.

Conjecture: NFS costs $\approx 2^c$ to factor integers $\approx 2^b$, where $c/b^{1/3}(\lg b)^{2/3} \to$ constant as $b \to \infty$. (1993 Buhler Lenstra Pomerance, et al.)

Constant ≈ 1.976 for circuits. (2001 Bernstein)

Another algorithm has *proven* cost 2^c where $c/b^{1/2}(\lg b)^{1/2} \to$ constant. (1981 Dixon; better constants: 1987 Pomerance, 1991 Vallée, 1992 Lenstra Pomerance)

In factorization attack, H_0 and H_1 are **generic**: they can be oracles that compute arbitrary functions. Attack succeeds no matter what H_0 and H_1 are.

Given *any* generic attack that succeeds against all H_0, H_1, can build an algorithm that factors k at comparable speed.

Enough to assume that success probability, averaged over all H_0, H_1, is high.

Sketch of construction:

Factorization algorithm
chooses random integer c;
chooses random string z;
chooses random H_1 values;
chooses each $H_0(H_1(z, m), m)$ as
efs^2 for random e, f, s;
and chooses each $H_0(\text{other}, m)$ as
$ef(sc)^2$ for random e, f, s.

Can compute exactly
the right e, f, s distribution.

Factorization algorithm
can now simulate signer
with these functions H_0, H_1.

Factorization algorithm
runs the attack, obtaining
a forgery (m', e', f', r', s').

If $H_1(z, m') = r'$, give up;
chance $\leq 1/16 + \epsilon$.

Now $ef(sc)^2 \equiv e'f'(s')^2$.
Check $\gcd\{k, s'\}$, $\gcd\{k, s' - sc\}$;
chance $\leq 1/2$ of both in $\{1, k\}$.

So generic attacks can't be easier than factorization.

If r is omitted, this proof breaks down. Fix: can build a slower factorization algorithm; so generic attacks can't be *much* easier than factorization.

Conjecture: No attacks are easier than factorization. (Counterargument: MD5 collision.)

Conjecture: Factorization is hard. (Counterargument: NFS.)

More credits

Converting generic attacks
into factorization algorithms:
1987 Fiat Shamir, for a signature
system; 1993 Bellare Rogaway,
for some encryption systems.

Quantified conversions:
1996 Bellare Rogaway;
1998 Bernstein; 2000 Coron; et al.

Exploiting non-random r:
2003 Katz Wang; 2003 Bernstein.

Expanded signatures

(1997 Bernstein)

Expand e, f, r, s into e, f, r, s, t
where $t = (fs^2 - eH_0(r, m))/k$.
Verifier can check whether
$(f(s \bmod \ell)^2 - (k \bmod \ell)(t \bmod \ell)$
$- e(H_0(r, m) \bmod \ell)) \bmod \ell = 0$
for a random 128-bit prime ℓ.
This is very fast.

If input is valid, says yes.
Otherwise, chance $\leq 2^{-115}$
of saying yes.

Compressed signatures

(2003 Bleichenbacher)

Compress e, f, r, s to e, f, r, v where $v \in \{1, 2, \ldots, 2^{769} - 1\}$ and $efv^2 H_0(r, m) \bmod k$ is in $\{0^2, 1^2, 2^2, \ldots, (2^{768} - 1)^2\}$. 97 bytes instead of 193 bytes.

Easy to find v from continued fraction of fs/k.

Easy to uncompress, or to check e, f, r, v directly.

Compressed keys

(2003 Coppersmith)

Require $\lfloor k/2^{512} \rfloor =$ 179870286739608
1109087939864337792829527094371869801110276348868
0668010543030620350477208772441576518762853656940
3357866962021859070432575840490938673081114568020
8028015726391074333854880135338238893595433658057
3963972429710649524801380822741794895484671657643
1759705516797612912096782118234207449553394447817.

Transmit only k mod 2^{512}.
64 bytes instead of 192 bytes.

How to generate p, q with $2^{512}\alpha < pq < 2^{512}(\alpha + 1)$?

First generate random p_0.
Compute $q_0 \approx 2^{512}(\alpha + 1/2)/p_0$.

Find 256-bit integers x, y with $p_0 y + q_0 x$ close to $2^{512}(\alpha + 1/2) - p_0 q_0$.

Set $p = p_0 + x$ and $q = q_0 + y$.
Check that p, q are primes in the right range;
if not, try a new p_0.

Advertisement

MCS 590, High-Speed Cryptography, Spring 2005

Prerequisite: Computer algorithms.

Other necessary background from computer architecture, numerical analysis, commutative algebra, number theory, and cryptography will be introduced on the fly.