# On the design of message-authentication codes

D. J. Bernstein
University of Illinois at Chicago

When we design
hash functions, stream ciphers,
and other secret-key primitives,
should we use
integer multiplication?

AES uses $32, 32 \to 32$ xor;
$32 \to 8$ byte extraction;
and $8 \to 32$ inversion box.

IDEA uses $16, 16 \to 16$ xor;
$16, 16 \to 16$ addition; and
$16, 16 \to 16$ multiplication.

Rabbit uses $32 \to 32$ rotation;
$32, 32 \to 32$ addition;
$32, 32 \to 32$ xor; and
$32, 32 \to 32, 32$ multiplication.

RC6 uses $32, 8 \to 32$ rotation;
$32, 32 \to 32$ addition;
$32, 32 \to 32$ xor; and
$32, 32 \to 32$ multiplication.

Salsa20 uses $32 \to 32$ rotation;
$32, 32 \to 32$ addition; and
$32, 32 \to 32$ xor.

"Multiplication is slow!"
$> 10\times$ as many bit operations as addition.

Counterargument:
"Multiplication is surprisingly fast!"
Has many applications, so CPU designers include big multiplication circuits.
Typical CPUs can start a new multiplication every cycle.

“Multiplication
scrambles its output
as thoroughly as
several simple operations!”

“No, it doesn’t!
Look at these scary attacks.
Need many multiplications
to achieve confidence.”

What if we can *prove*
that multiplication provides
the security we need?

# An authentication system

Let's use multiplication
to authenticate messages.

Standardize a prime $p = 1000003$.

Sender rolls 10-sided die
to generate independent
uniform random secrets
$r \in \{0, 1, \ldots, 999999\}$,
$s_1 \in \{0, 1, \ldots, 999999\}$,
$s_2 \in \{0, 1, \ldots, 999999\}$,
$\ldots$,
$s_{100} \in \{0, 1, \ldots, 999999\}$.

Sender meets receiver in private and tells receiver the same secrets $r, s_1, s_2, \ldots, s_{100}$.

Later: Sender wants to send 100 messages $m_1, \ldots, m_{100}$, each having 5 components $m_n[1], m_n[2], m_n[3], m_n[4], m_n[5]$ with $m_n[i] \in \{0, 1, \ldots, 999999\}$.

Sender transmits 30-digit $m_n[1], m_n[2], m_n[3], m_n[4], m_n[5]$ together with an **authenticator**

$$(m_n[1]r + \cdots + m_n[5]r^5 \bmod p)$$
$$\quad + s_n \bmod 1000000$$

and the message number $n$.

e.g. $r = 314159$, $s_{10} = 265358$,

$m_{10} =$ 000006 000007 000000 000000 000000:

Sender computes authenticator

$(6r + 7r^2 \bmod p)$

$\quad + s_{10} \bmod 1000000 =$

$(6 \cdot 314159 + 7 \cdot 314159^2$

$\quad \bmod 1000003)$

$\quad + 265358 \bmod 1000000 =$

$953311 + 265358 \bmod 1000000 =$

$218669$.

Sender transmits

authenticated message

10 000006 000007 000000 000000 000000 218669.

# Speed analysis

Notation: $m_n(x) = \sum m_n[i]x^i$.

To compute $m_n(r)$ mod $p$:
multiply $m_n[5]$ by $r$,
add $m_n[4]$, multiply by $r$,
add $m_n[3]$, multiply by $r$,
add $m_n[2]$, multiply by $r$,
add $m_n[1]$, multiply by $r$.
Reduce mod $p$ after each mult.

Slightly more time to
compute authenticator $a_n =$
$(m_n(r) \bmod p) + s_n \bmod 1000000$.

Reducing mod 1000003 is easy:
e.g., $240881099091 =$
$240881 \cdot 1000000 + 99091 \equiv$
$240881(-3) + 99091 =$
$-722643 + 99091 =$
$-623552$.

Easily adjust to range
$\{0, 1, \ldots, p - 1\}$
by adding/subtracting a few $p$'s.
(Beware timing attacks!)

Speedup: Delay the adjustment;
extra $p$'s won't damage
subsequent field operations.

Main work is multiplication.
For each 6-digit message chunk, have to do one multiplication of the 6-digit secret $r$ into an accumulator mod $p$.

Scaled up for serious security: "Poly1305" uses $p = 2^{130} - 5$.
For each 128-bit message chunk, have to do one multiplication of a 128-bit secret $r$ into an accumulator mod $2^{130} - 5$.
$\approx$ 5 cycles per message byte, depending on the CPU.

# Security analysis

Attacker's goal:
Find $n', m', a'$ such that
$m' \neq m_{n'}$ but $a' =$
$(m'(r) \bmod p) + s_{n'} \bmod 1000000$.
Here $m'(x) = \sum_i m'[i]x^i$.

Obvious attack:
Choose any $m' \neq m_1$.
Choose uniform random $a'$.
Success chance 1/1000000.

Can repeat attack.
Each forgery has chance
1/1000000 of being accepted.

More subtle attack:

Choose $m' \neq m_1$ so that the polynomial $m'(x) - m_1(x)$ has 5 distinct roots $x \in \{0, 1, \ldots, 999999\}$ modulo $p$. Choose $a' = a$.

e.g. $m_1 = (100, 0, 0, 0, 0)$, $m' = (125, 1, 0, 0, 1)$: $m'(x) - m_1(x) = x^5 + x^2 + 25x$ which has five roots mod $p$: $0, 299012, 334447, 631403, 735144$.

Success chance $5/1000000$.

Actually, success chance can be above 5/1000000.

Example: If $m_1(334885)$ mod $p$ $\in \{1000000, 1000001, 1000002\}$ then a forgery $(1, m', a_1)$ with $m'(x) = m_1(x) + x^5 + x^2 + 25x$ also succeeds for $r = 334885$; success chance 6/1000000.
Reason: 334885 is a root of $m'(x) - m_1(x) + 1000000$.

Can have as many as 15 roots of $(m'(x) - m_1(x)) \cdot$ $(m'(x) - m_1(x) + 1000000) \cdot$ $(m'(x) - m_1(x) - 1000000)$.

Do better by varying $a'$?

No. Easy to prove: Every choice of $(n', m', a')$ with $m' \neq m_{n'}$ has chance $\leq 15/1000000$ of being accepted by receiver.

Underlying fact: $\leq 15$ roots of $(m'(x) - m_1(x) - a' + a_1) \cdot$
$(m'(x) - m_1(x) - a' + a_1 + 10^6) \cdot$
$(m'(x) - m_1(x) - a' + a_1 - 10^6)$.

Warning: very easy to break the oversimplified authenticator
$(m_n[1] + \cdots + m_n[5]r^4 \bmod p)$
$\quad + s_n \bmod 1000000$:
solve $m'(x) - m_1(x) = a' - a_1$.

Scaled up for serious security:

Poly1305 uses 128-bit $r$'s, with 22 bits cleared for speed. Adds $s_n$ mod $2^{128}$.

Assuming $\leq L$-byte messages: Each forgery succeeds for $\leq 8 \lceil L/16 \rceil$ choices of $r$. Probability $\leq 8 \lceil L/16 \rceil / 2^{106}$.

$D$ forgeries are all rejected with probability $\geq 1 - 8D \lceil L/16 \rceil / 2^{106}$.

e.g. $2^{64}$ forgeries, $L = 1536$: Pr[all rejected] $\geq 0.9999999998$.

Authenticator is still secure
for variable-length messages,
if different messages are
different polynomials mod $p$.

Split string into 16-byte chunks,
maybe with smaller final chunk;
append 1 to each chunk;
view as little-endian integers
in $\{1, 2, 3, \ldots, 2^{129}\}$.
Multiply first chunk by $r$,
add next chunk, multiply by $r$,
etc., last chunk, multiply by $r$,
mod $2^{130} - 5$, add $s_n$ mod $2^{128}$.

# Reducing the key length

Like the one-time pad,
this authentication system
has a security guarantee.

One-time pad needs
$L$ shared secret bytes
to encrypt $L$ message bytes.

Authentication system needs
16 shared secret bytes
to authenticate $L$ message bytes.

Each new message needs
new shared secret bytes,
used only once.
How to handle many messages?

Authenticator is $m_n(r) \bmod p$
encrypted with one-time pad $s_n$.

Can replace one-time pad
with stream-cipher output.

Typical stream cipher:
AES in counter mode.
Sender, receiver share $(r, k)$
where $k$ is 16-byte AES key;
compute $s_n = \text{AES}_k(n)$.

Security proof breaks down
since $s_n$'s are dependent,
but can still prove that
attack on authenticator
implies attack on AES.

```
unsigned int j;
mpz_class rbar = 0;
for (j = 0;j < 16;++j)
  rbar += ((mpz_class) r[j]) << (8 * j);
mpz_class h = 0;
mpz_class p = (((mpz_class) 1) << 130) - 5;
while (mlen > 0) {
  mpz_class c = 0;
  for (j = 0;(j < 16) && (j < mlen);++j)
    c += ((mpz_class) m[j]) << (8 * j);
  c += ((mpz_class) 1) << (8 * j);
  m += j; mlen -= j;
  h = ((h + c) * rbar) % p;
}
unsigned char aeskn[16];
aes(aeskn,k,n);
for (j = 0;j < 16;++j)
  h += ((mpz_class) aeskn[j]) << (8 * j);
for (j = 0;j < 16;++j) {
  mpz_class c = h % 256;
  h >>= 8;
  out[j] = c.get_ui();
}
```

Another stream cipher:
$F_k(n) = \text{MD5}(k, n)$.
Somewhat slower than AES.

"Hasn't MD5 been broken?"
Distinct $(k, n), (k', n')$ are known
with $\text{MD5}(k, n) = \text{MD5}(k', n')$.
(2004 Wang)
Still not obvious how to predict
$n \mapsto \text{MD5}(k, n)$ for secret $k$.
We know AES collisions too!

Many other stream ciphers
are unbroken, faster than AES.

# Alternatives to $+$

Use $\cdots \oplus \mathrm{AES}_k(n)$
instead of $\cdots + \mathrm{AES}_k(n)$?
No! Destroys security analysis;
might allow successful forgeries
even if AES is secure.

Use $\mathrm{AES}_k(\cdots)$, omitting $n$?
No! Broken by known attacks
using $< 2^{64}$ authenticators.
But ok for small # messages.

Use $\mathrm{Salsa20}(k, n, \cdots)$?
Seems to be massive overkill.

# Alternatives to Poly1305

Notation: $\text{Poly1305}_r(m) =$
$(m(r) \bmod 2^{130} - 5) \bmod 2^{128}$.

For all distinct messages $m, m'$:
$\Pr[\text{Poly1305}_r(m) =$
$\quad \text{Poly1305}_r(m')]$ is very small.
"Small collision probabilities."

For all distinct messages $m, m'$
and all 16-byte sequences $\Delta$:
$\Pr[\text{Poly1305}_r(m) =$
$\quad \text{Poly1305}_r(m') + \Delta \bmod 2^{128}]$
is very small.
"Small differential probabilities."

Easy to build other functions
that satisfy these properties.

Embed messages and outputs into
polynomial ring $\mathbf{Z}[x_1, x_2, x_3, \ldots]$.

Use $m \mapsto m \bmod r$ where
$r$ is a random prime ideal.

Small differential probability
means that $m - m' - \Delta$
is divisible by very few $r$'s
when $m \neq m'$.

(Addition of $\Delta$ is
mod $2^{128}$; be careful.)

Example: (1981 Karp Rabin)

View messages $m$ as integers,
specifically multiples of $2^{128}$.
Outputs: $\{0, 1, \ldots, 2^{128} - 1\}$.

Reduce $m$ modulo a uniform
random prime number $r$
between $2^{120}$ and $2^{128}$.
(Problem: generating $r$ is slow.)

Low differential probability:
if $m \neq m'$ then $m - m' - \Delta \neq 0$
so $m - m' - \Delta$ is divisible
by very few prime numbers.

Variant that works with $\oplus$:

View messages $m$ as polynomials $m_{128}x^{128} + m_{129}x^{129} + \cdots$ with each $m_i$ in $\{0, 1\}$.

Outputs: $o_0 + o_1 x + \cdots + o_{127}x^{127}$ with each $o_i$ in $\{0, 1\}$.

Reduce $m$ modulo $2, r$ where $r$ is a uniform random irreducible degree-128 polynomial over $\mathbf{Z}/2$. (Problem: division by $r$ is slow; typical CPU has no big circuit for polynomial multiplication.)

Example: (1974 Gilbert MacWilliams Sloane)

Choose prime number $p \approx 2^{128}$.
View messages $m$ as linear polys $m_1 x_1 + m_2 x_2 + m_3 x_3$ with $m_1, m_2, m_3 \in \{0, \ldots, p-1\}$.
Outputs: $\{0, \ldots, p-1\}$.

Reduce $m$ modulo $p, x_1 - r_1, x_2 - r_2, x_3 - r_3$ to $m_1 r_1 + m_2 r_2 + m_3 r_3 \bmod p$.
(Problem: long $m$ needs long $r$.)

Example: (1993 den Boer; independently 1994 Taylor; independently 1994 Bierbrauer Johansson Kabatianskii Smeets)

Choose prime number $p \approx 2^{128}$. View messages $m$ as polynomials $m_1 x + m_2 x^2 + m_3 x^3 + \cdots$ with $m_1, m_2, \ldots \in \{0, 1, \ldots, p-1\}$. Outputs: $\{0, 1, \ldots, p-1\}$.

Reduce $m$ modulo $p, x - r$ where $r$ is a uniform random element of $\{0, 1, \ldots, p-1\}$; i.e., compute $m_1 r + m_2 r^2 + \cdots \bmod p$.

“hash127”: 32-bit $m_i$’s,
$p = 2^{127} - 1$. (1999 Bernstein)

“PolyR”: 64-bit $m_i$’s,
$p = 2^{64} - 59$; re-encode $m_i$’s between $p$ and $2^{64} - 1$; run twice to achieve reasonable security.
(2000 Krovetz Rogaway)

“Poly1305”: 128-bit $m_i$’s,
$p = 2^{130} - 5$. (2002 Bernstein, fully developed in 2004–2005)

“CWC”: 96-bit $m_i$’s, $p = 2^{127} - 1$.
(2003 Kohno Viega Whiting)

There are other ways to build functions with small proven or conjectured differential probabilities.

Example:
("CBC": "cipher block chaining")
Conjecturally $m_1, m_2, m_3 \mapsto \text{AES}_r(\text{AES}_r(\text{AES}_r(m_1) \oplus m_2) \oplus m_3)$ has small differential probabilities.
True if AES is secure.

(Much slower than Poly1305.)

Example: (1970 Zobrist, adapted)
Conjecturally $m_1, m_2, m_3 \mapsto$
$\text{AES}_r(1, m_1) \oplus$
$\text{AES}_r(2, m_2) \oplus$
$\text{AES}_r(3, m_3)$
has small differential probabilities.

(Even slower.)

Example: $m \mapsto \text{MD5}(r, m)$
is conjectured to have
small collision probabilities.

(Faster than AES,
but not as fast as Poly1305,
and "small" is debatable.)

# How to build your own MAC

1. Choose a combination method:
$h(m) + f(n)$ or $h(m) \oplus f(n)$
or $f(h(m))$—worse security—
or $f(n, h(m))$—bigger $f$ input.

2. Choose a random function $h$
where the appropriate probability
($+$-differential or $\oplus$-differential
or collision or collision) is small:
e.g., Poly1305$_r$.

3. Choose a random function $f$
that seems indistinguishable
from uniform: e.g., AES$_k$.

4. Optional complication:
Generate $k, r$ from a shorter key;
e.g., $k = \text{AES}_s(0)$, $r = \text{AES}_s(1)$;
or $k = \text{MD5}(s)$, $r = \text{MD5}(s \oplus 1)$;
many more possibilities.

5. Choose a Googleable name
for your MAC.

6. Put it all together.

7. Publish!

Example:

1. Combination: $f(h(m))$.
2. Low collision probability: $\text{AES}_r(\text{AES}_r(m_1) \oplus m_2)$.
3. Unpredictable: $\text{AES}_k$.
4. Optional complication: No.
5. Name: "EMAC."
6. $\text{EMAC}_{k,r}(m_1, m_2) = \text{AES}_k(\text{AES}_r(\text{AES}_r(m_1) \oplus m_2))$.
7. (2000 Petrank Rackoff)

Example: “NMAC-MD5” is MD5$(k, \text{MD5}(r, m))$.

“HMAC-MD5” is NMAC-MD5 plus the optional complication.

(1996 Bellare Canetti Krawczyk, claiming “the first rigorous treatment of the subject”)

Stronger: MD5$(k, n, \text{MD5}(r, m))$.
Stronger and faster: MD5$(k, n, \text{Poly1305}_r(m))$.
Wow, I’ve just invented two new MACs! Time to publish!

# State-of-the-art MACs

Cycles per byte to authenticate 1024-byte packet:

| | Poly 1305 -AES | UMAC -128 |
|---|---|---|
| Athlon | 3.75 | 7.38 |
| Pentium M | 4.50 | 8.48 |
| Pentium 4 | 5.33 | 3.12 |
| SPARC III | 5.47 | 51.06 |
| PPC G4 | 8.27 | 21.72 |
| bytes/key | 32 | 1600 |

UMAC really likes the P4.

Similar: VMAC likes Athlon 64.

Some important speed issues:

1. Implementor flexibility.
Poly1305 uses 128-bit integers,
split into whatever sizes are
convenient for the CPU.
UMAC uses P4-size integers
and suffers on other CPUs.

2. Key agility.
Poly1305 can fit thousands
of simultaneous keys into cache,
and remains fast even when
keys are out of cache.
UMAC needs big expanded keys.

# 3. Number of multiplications.

den Boer et al.; Poly1305:
$(m_1 r + m_2) r + \cdots$.
Each chunk: mult, add.

Gilbert-MacWilliams-Sloane:
$m_1 r_1 + m_2 r_2 + \cdots$.
Each chunk: mult, add.

Winograd; UMAC; VMAC:
$(m_1 + r_1)(m_2 + r_2) + \cdots$.
Each chunk: 0.5 mults, 1.5 adds.

Does small key $r$ allow
0.5 mults per message chunk?

Yes!

Another old trick of Winograd:

$$\begin{aligned}&(((m_1 + r)(m_2 + r^2) +\\ &\quad (m_3 + r))(m_4 + r^4) +\\ &\ ((m_5 + r)(m_6 + r^2) +\\ &\quad (m_7 + r)))(m_8 + r^8) + \cdots\end{aligned}$$

times a final nonzero $m_n$
times $r$.

"MAC1071," coming soon.