The Poly1305-AES message-authentication code

D. J. Bernstein

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

The AES function

(“Rijndael” 1998 Daemen Rijmen; 2001 standardized as “AES”)

Given 16-byte sequence n and 16-byte sequence k, AES produces 16-byte sequence $\text{AES}_k(n)$.

Uses table lookup and $\oplus$ (xor):

$\texttt{e0} = \texttt{tab[k[13]]}\oplus\texttt{1}$

$\texttt{e1} = \texttt{tab[k[0]}\oplus\texttt{n[0]]}\oplus\texttt{k[0]}\oplus\texttt{e0}$

etc.

$\text{AES}_k(n) = (\texttt{e784}, \ldots, \texttt{e799})$.

Unpredictability

Consider two oracles.

One oracle knows a uniform random 16-byte sequence k. Given a 16-byte sequence n, this oracle returns $\text{AES}_k(n)$.

The other oracle knows a uniform random permutation f of the set of 16-byte sequences. Given n, this oracle returns $f(n)$.

Design goal of AES: These oracles are indistinguishable.

Define δ as attacker's chance of distinguishing AES_k from uniform random permutation: i.e., distance between Pr[attacker says yes given f] and Pr[attacker says yes given AES_k].

We *believe* that $\delta \le 2^{-40}$ even for an attacker using 100 years of CPU time on all the world's computers.

Can't prove it, but many experts have failed to disprove it.

The Poly1305-AES function

Given byte sequence m,
16-byte sequence n,
16-byte sequence k,
16-byte sequence r
with certain bits cleared,
Poly1305-AES produces
16-byte sequence
$\text{Poly1305}_r(m, \text{AES}_k(n))$.

Uses polynomial evaluation
modulo the prime $2^{130} - 5$.

```
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();
}
```

Poly1305-AES authenticators

Sender, receiver share secret uniform random k, r.

Sender attaches authenticator $a = \text{Poly1305}_r(m, \text{AES}_k(n))$ to message m with nonce n.

(The usual nonce requirement: never use the same nonce for two different messages.)

Receiver rejects n', m', a' if $a' \neq \text{Poly1305}_r(m', \text{AES}_k(n'))$.

Poly1305-AES security guarantee

Attacker adaptively
chooses $C \le 2^{64}$ messages,
sees their authenticators,
attempts D forgeries;
all messages $\le L$ bytes.

Then Pr[all forgeries rejected]
$\ge 1 - \delta - 14D \lceil L/16 \rceil / 2^{106}$.

Example: Say $\delta \le 2^{-40}$; $L = 1536$;
see 2^{64} authenticators;
attempt 2^{64} forgeries. Then
Pr[all rejected] ≥ 0.999999999998.

Alternatives to AES

Can replace AES_k with any F_k that is conjecturally unpredictable.

Example: $F_k(n) = \text{MD5}(k, n)$. 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!

Alternatives to $+$

$\text{Poly1305}_r(m, \text{AES}_k(n))$ equals $\text{Poly1305}_r(m, 0) + \text{AES}_k(n)$ where $+$ is addition modulo 2^{128}.

Use $\text{Poly1305}_r(m, 0) \oplus \text{AES}_k(n)$? No! Eliminates security guarantee.

Use $\text{AES}_k(\text{Poly1305}_r(m, 0))$? Has a guarantee, but bad for large C: roughly $8C(C + D)\lceil L/16 \rceil / 2^{106}$.

Use $\text{MD5}(k, n, \text{Poly1305}_r(m, 0))$? That's fine if MD5 is ok.

Alternatives to Poly1305

The crucial property of Poly1305_r:
If m, m' are distinct messages
and Δ is a 16-byte sequence then

$$\Pr[\text{Poly1305}_r(m, 0) = \text{Poly1305}_r(m', 0) + \Delta]$$

is very small: $\leq 8\lceil L/16\rceil/2^{106}$.
"Small differential probabilities."

In particular, for $\Delta = 0$:
If m, m' are distinct messages then

$$\Pr[\text{Poly1305}_r(m, 0) = \text{Poly1305}_r(m', 0)]$$ is very small.

"Small collision probabilities."

Easy to build 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 Δ is actually 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; no polynomial-multiplication circuit in a typical computer.)

Example: (1974 Gilbert MacWilliams Sloane)

Choose prime number $p \approx 2^{128}$.
View messages m as linear polynomials $m_1x_1 + m_2x_2 + m_3x_3$ 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 $m_1r_1 + m_2r_2 + m_3r_3 \bmod p$.
(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 $m_1, m_2, m_3, \ldots \in \{0, 1, \ldots, p-1\}$. 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)

Often people use functions where the differential probabilities are merely *conjectured* to be small.

Example: ("cipher block chaining") *If* AES_r is unpredictable then $m_1, m_2, m_3 \mapsto$ $\text{AES}_r(\text{AES}_r(\text{AES}_r(m_1) \oplus m_2) \oplus m_3)$ has small differential probabilities.

(Much slower than Poly1305.)

Example: (1970 Zobrist, adapted)
If AES_r is unpredictable
then $m_1, m_2, m_3 \mapsto$
$\text{AES}_r(1, m_1) \oplus \text{AES}_r(2, m_2) \oplus$
$\text{AES}_r(3, m_3)$
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.)

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 unpredictable:
e.g., AES_k.

4. Optional complication:
Generate k, r from a shorter key;
e.g., $k = \text{AES}_s(0)$, $r = \text{AES}_s(1)$;
e.g., $k = \text{MD5}(s)$, $r = \text{MD5}(s \oplus 1)$;
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: AES_k.
4. Optional complication: No.
5. Name: “EMAC.” (Whoops.)
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, MD5(r, m)).

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

(1996 Bellare Canetti Krawczyk, claiming novelty of the entire structure)

Stronger: MD5(k, n, MD5(r, m)).
Stronger and faster:
MD5(k, n, $\text{Poly1305}_r(m, 0)$).
Wow, I’ve just invented two new MACs! Time to publish!

Speed

"MMH: software message authentication in the Gbit/second rates" (1997 Halevi Krawczyk)

Gilbert-MacWilliams-Sloane (incorrectly credited to Carter and Wegman), slightly tweaked.

1.5 Pentium Pro cycles/byte
. . . for a 4-byte authenticator.
6 Pentium Pro cycles/byte
for reasonable security.
Not as fast as MD5.

Polynomial evaluation mod $2^{127} - 1$
faster than MD5 on
Pentium, UltraSPARC, etc.
(1999 Bernstein)

... using a big precomputed
table of powers of r.
MMH also uses large table.

Problem: What happens in
applications that handle
many keys simultaneously?
Tables don't fit into cache,
and take a long time to load!

Independently: “UMAC-MMX-60, 0.98 Pentium II cycles/byte” (1999 Black Halevi Krawczyk Krovetz Rogaway, using a Winograd trick without credit)

. . . for an 8-byte authenticator.

. . . plus many cycles per message.

. . . and much slower on PowerPC etc. (Newest UMAC benchmark page: “All speeds were measured on a Pentium 4.”)

. . . and again using large tables.

Poly1305: *consistent* high speed.

Fast on a wide variety of CPUs.

No precomputation. Still fast
when handling many keys.
("High key agility.")

No constraints on message length,
message alignment, etc.

Fast public-domain software now
available: `cr.yp.to/mac.html`.

CPU cycles for ℓ-byte message with all data aligned in L1 cache:

ℓ	16	128	1024
Athlon	634	979	3767
Pentium III	746	1247	5361
Pentium M	726	1161	4611
PowerPC 7410	896	1728	8464
PowerPC Sstar	910	1459	5905
UltraSPARC II	816	1288	5118
UltraSPARC III	854	1383	5601

Comprehensive speed tables:

`cr.yp.to/mac/speed.html`

Some important speed tips:

- Represent large integers as sums of floating-point numbers (1968 Veltkamp, 1971 Dekker) in pre-specified ranges (1999 Bernstein).
- Schedule instructions manually. C compiler can't figure out, e.g., which additions associate.
- Allocate registers manually. C compiler spills values for all sorts of silly reasons.

200× faster than easy code.