The Poly1305-AES message-authentication code

Daniel J. Bernstein ⋆

Department of Mathematics, Statistics, and Computer Science (M/C 249)
The University of Illinois at Chicago
Chicago, IL 60607–7045
djb@cr.yp.to

Abstract. Poly1305-AES is a state-of-the-art message-authentication code suitable for a wide variety of applications. Poly1305-AES computes a 16-byte authenticator of a variable-length message, using a 16-byte AES key, a 16-byte additional key, and a 16-byte nonce. The security of Poly1305-AES is very close to the security of AES; the security gap is at most $14D\lceil L/16\rceil/2^{106}$ if messages have at most L bytes, the attacker sees at most 2^{64} authenticated messages, and the attacker attempts D forgeries. Poly1305-AES can be computed at extremely high speed: for example, fewer than $3.1\ell + 780$ Athlon cycles for an ℓ-byte message. This speed is achieved *without* precomputation; consequently, 1000 keys can be handled simultaneously without cache misses. Special-purpose hardware can compute Poly1305-AES at even higher speed. Poly1305-AES is parallelizable, incremental, and not subject to any intellectual-property claims.

1 Introduction

This paper introduces and analyzes Poly1305-AES, a state-of-the-art secret-key message-authentication code suitable for a wide variety of applications.

Poly1305-AES computes a 16-byte authenticator $\text{Poly1305}_r(m, \text{AES}_k(n))$ of a variable-length message m, using a 16-byte AES key k, a 16-byte additional key r, and a 16-byte nonce n. Section 2 of this paper presents the complete definition of Poly1305-AES.

Poly1305-AES has several useful features:

- **Guaranteed security if AES is secure.** The security gap is small, even for long-term keys; the only way for an attacker to break Poly1305-AES is to break AES. Assume, for example, that messages are packets up to 1024 bytes; that the attacker sees 2^{64} messages authenticated under a Poly1305-AES key; that the attacker attempts a whopping 2^{75} forgeries; and that the attacker cannot break AES with probability above δ. Then, with probability at least $0.999999 - \delta$, *all* of the 2^{75} forgeries are rejected.

⋆ The author was supported by the National Science Foundation under grant CCR–9983950, and by the Alfred P. Sloan Foundation. Date of this document: 2005.03.29. Permanent ID of this document: 0018d9551b5546d97c340e0dd8cb5750. This version is final and may be freely cited.

- **Cipher replaceability.** If anything does go wrong with AES, users can switch from Poly1305-AES to Poly1305-AnotherFunction, with an identical security guarantee. All the effort invested in the non-AES part of Poly1305-AES can be reused; the non-AES part of Poly1305-AES cannot be broken.
- **Extremely high speed.** My published Poly1305-AES software takes just 3843 Athlon cycles, 5361 Pentium III cycles, 5464 Pentium 4 cycles, 4611 Pentium M cycles, 8464 PowerPC 7410 cycles, 5905 PowerPC RS64 IV cycles, 5118 UltraSPARC II cycles, or 5601 UltraSPARC III cycles to verify an authenticator on a 1024-byte message. Poly1305-AES offers *consistent* high speed, not just high speed for one CPU.
- **Low per-message overhead.** The same software takes just 1232 Pentium 4 cycles, 1264 PowerPC 7410 cycles, or 1077 UltraSPARC III cycles to verify an authenticator on a 64-byte message. Poly1305-AES offers *consistent* high speed, not just high speed for long messages. Most competing functions have much larger overhead for each message; they are designed for long messages, without regard to short-packet performance.
- **Key agility.** Poly1305-AES offers *consistent* high speed, not just high speed for single-key benchmarks. The timings in this paper do *not* rely on any pre-expansion of the 32-byte Poly1305-AES key (k, r); Poly1305-AES can fit thousands of simultaneous keys into cache, and remains fast even when keys are out of cache. This was my primary design goal for Poly1305-AES. Almost all competing functions use a large table for each key; as the number of keys grows, those functions miss the cache and slow down dramatically.
- **Parallelizability and incrementality.** The circuit depth of Poly1305-AES is quite small, even for long messages. Consequently, Poly1305-AES can take advantage of additional hardware to reduce the latency for long messages. For essentially the same reason, Poly1305-AES can be recomputed at low cost for a small modification of a long message.
- **No intellectual-property claims.** I am not aware of any patents or patent applications relevant to Poly1305-AES.

Section 3 of this paper analyzes the security of Poly1305-AES. Section 4 discusses the software achieving the speeds stated above. Section 5 discusses the speed of Poly1305-AES in other contexts.

Genealogy

Gilbert, MacWilliams, and Sloane in [15] introduced the idea of provably secure authentication. The Gilbert-MacWilliams-Sloane system is fast, but it requires keys longer than L bytes to handle L-byte messages, and it requires a completely new key for each message.

Wegman and Carter in [32] pointed out that the key length could be merely $64 \lg L$ for the first message plus 16 bytes for each additional message. At about the same time, in a slightly different context, Karp and Rabin achieved a key length of 32 bytes for the first message; see [19] and [26]. The system in [19] is fast once keys are generated, but key generation is slow.

The idea of using a cipher such as AES to expand a short key into a long key is now considered obvious. Brassard in [12] published the idea in the Wegman-Carter context; I don't know whether the idea was considered obvious back then.

Polynomial-evaluation MACs—MACs that treat each message as a univariate polynomial over a finite field and then evaluate that polynomial at the key—were introduced in three papers independently: [14] by den Boer; [31, Section 3] by Taylor; [9, Section 4] by Bierbrauer, Johansson, Kabatianskii, and Smeets. Polynomial-evaluation MACs combine several attractive features: short keys, fast key generation, and fast message authentication. Several subsequent papers reported implementations of polynomial-evaluation MACs over binary fields: [28] by Shoup; [4] by Afanassiev, Gehrmann, and Smeets, reinventing Kaminski's division algorithm in [18]; [22] by Nevelsteen and Preneel.

Polynomial-evaluation MACs over prime fields can exploit the multipliers built into many current CPUs, achieving substantially better performance than polynomial-evaluation MACs over binary fields. This idea was first published in my paper [5] in April 1999, and explained in detail in [7]. Another MAC, avoiding binary fields for the same reason, was published independently by Black, Halevi, Krawczyk, Krovetz, and Rogaway in [11] in August 1999.

I used 32-bit polynomial coefficients modulo $2^{127} - 1$ ("hash127") in [5] and [7]. The short coefficients don't allow great performance (for short messages) without precomputation, so I casually precomputed a few kilobytes of data for each key; this is a disaster for applications handling many keys simultaneously, but I didn't think beyond a single key. Similarly, [11] ("UMAC") uses large keys.

Krovetz and Rogaway in [21] suggested 64-bit coefficients modulo $2^{64} - 59$, with an escape mechanism for coefficients between $2^{64} - 59$ and $2^{64} - 1$. They did not claim competitive performance: their software, run twice to achieve a reasonable 100-bit security level, was more than three times slower than hash127 (and more than six times slower for messages with all bits set). Krovetz and Rogaway did point out, however, that their software did not require large tables.

In `http://cr.yp.to/talks.html#2002.06.15`, posted July 2002, I pointed out that 128-bit coefficients over the slightly *larger* prime field $\mathbf{Z}/(2^{130} - 5)$ allow excellent performance without precomputation. This paper explains Poly1305-AES in much more detail.

Kohno, Viega, and Whiting subsequently suggested 96-bit coefficients modulo $2^{127} - 1$ ("CWC HASH"). They published some non-competitive timings for CWC HASH and then gave up on the idea. A careful implementation of CWC HASH without precomputation would be quite fast, although still not as fast as Poly1305-AES.

2 Specification

This section defines the Poly1305-AES function. The Poly1305-AES formula is a straightforward polynomial evaluation modulo $2^{130} - 5$; most of the detail is in key format and message padding.

Messages

Poly1305-AES authenticates messages. A **message** is any sequence of bytes $m[0], m[1], \ldots, m[\ell-1]$; a **byte** is any element of $\{0, 1, \ldots, 255\}$. The length ℓ can be any nonnegative integer, and can vary from one message to another.

Keys

Poly1305-AES authenticates messages using a 32-byte secret key shared by the message sender and the message receiver. The key has two parts: first, a 16-byte AES key k; second, a 16-byte string $r[0], r[1], \ldots, r[15]$. The second part of the key represents a 128-bit integer r in unsigned little-endian form: i.e., $r = r[0] + 2^8 r[1] + \ldots + 2^{120} r[15]$.

Certain bits of r are required to be 0: $r[3], r[7], r[11], r[15]$ are required to have their top four bits clear (i.e., to be in $\{0, 1, \ldots, 15\}$), and $r[4], r[8], r[12]$ are required to have their bottom two bits clear (i.e., to be in $\{0, 4, 8, \ldots, 252\}$). Thus there are 2^{106} possibilities for r. In other words, r is required to have the form $r_0 + r_1 + r_2 + r_3$ where $r_0 \in \{0, 1, 2, 3, \ldots, 2^{28} - 1\}$, $r_1/2^{32} \in \{0, 4, 8, 12, \ldots, 2^{28} - 4\}$, $r_2/2^{64} \in \{0, 4, 8, 12, \ldots, 2^{28} - 4\}$, and $r_3/2^{96} \in \{0, 4, 8, 12, \ldots, 2^{28} - 4\}$.

Nonces

Poly1305-AES requires each message to be accompanied by a 16-byte **nonce**, i.e., a unique message number. Poly1305-AES feeds each nonce n through AES_k to obtain the 16-byte string $\mathrm{AES}_k(n)$.

There is nothing special about AES here. One can replace AES with an arbitrary keyed function from an arbitrary set of nonces to 16-byte strings. This paper focuses on AES for concreteness.

Conversion and padding

Let $m[0], m[1], \ldots, m[\ell-1]$ be a message. Write $q = \lceil \ell/16 \rceil$. Define integers $c_1, c_2, \ldots, c_q \in \{1, 2, 3, \ldots, 2^{129}\}$ as follows: if $1 \le i \le \lfloor \ell/16 \rfloor$ then

$$c_i = m[16i - 16] + 2^8 m[16i - 15] + 2^{16} m[16i - 14] + \cdots + 2^{120} m[16i - 1] + 2^{128};$$

if ℓ is not a multiple of 16 then

$$c_q = m[16q - 16] + 2^8 m[16q - 15] + \cdots + 2^{8(\ell \bmod 16) - 8} m[\ell - 1] + 2^{8(\ell \bmod 16)}.$$

In other words: Pad each 16-byte chunk of a message to 17 bytes by appending a 1. If the message has a final chunk between 1 and 15 bytes, append 1 to the chunk, and then zero-pad the chunk to 17 bytes. Either way, treat the resulting 17-byte chunk as an unsigned little-endian integer.

Authenticators

$\text{Poly1305}_r(m, \text{AES}_k(n))$, the Poly1305-AES authenticator of a message m with nonce n under secret key (k, r), is defined as the 16-byte unsigned little-endian representation of

$$(((c_1 r^q + c_2 r^{q-1} + \cdots + c_q r^1) \bmod 2^{130} - 5) + \text{AES}_k(n)) \bmod 2^{128}.$$

Here the 16-byte string $\text{AES}_k(n)$ is treated as an unsigned little-endian integer, and $c_1, c_2, \ldots, c_q$ are the integers defined above. See Appendix B for examples.

Sample code

The following C++ code reads k from `k[0]`, `k[1]`, ..., `k[15]`, reads r from `r[0]`, `r[1]`, ..., `r[15]`, reads $\text{AES}_k(n)$ from `s[0]`, `s[1]`, ..., `s[15]`, reads m from `m[0]`, `m[1]`, ..., `m[l-1]`, and places $\text{Poly1305}_r(m, \text{AES}_k(n))$ into `out[0]`, `out[1]`, ..., `out[15]`:

```
#include <gmpxx.h>

void poly1305_gmpxx(unsigned char *out,
  const unsigned char *r,
  const unsigned char *s,
  const unsigned char *m,unsigned int l)
{
  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 (l > 0) {
    mpz_class c = 0;
    for (j = 0;(j < 16) && (j < l);++j)
      c += ((mpz_class) m[j]) << (8 * j);
    c += ((mpz_class) 1) << (8 * j);
    m += j; l -= j;
    h = ((h + c) * rbar) % p;
  }
  for (j = 0;j < 16;++j)
    h += ((mpz_class) s[j]) << (8 * j);
  for (j = 0;j < 16;++j) {
    mpz_class c = h % 256;
    h >>= 8;
    out[j] = c.get_ui();
  }
}
```

See [16] for the underlying integer-arithmetic library, `gmpxx`.

This code is not meant as a high-speed implementation; it does not have even the simplest speedups; it should be expected to provide intolerable performance. It is simply a secondary statement of the definition of Poly1305-AES.

Design decisions

I considered various primes above 2^{128}. I chose $2^{130} - 5$ because its sparse form makes divisions particularly easy in both software and hardware. My encoding of messages as polynomials takes advantage of the gap between 2^{128} and $2^{130} - 5$.

There are several reasons that Poly1305-AES uses nonces. First, comparable protocols without nonces have security bounds that look like $C(C + D)L/2^{106}$ rather than $DL/2^{106}$—here C is the number of messages authenticated by the sender, D is the number of forgery attempts, and L is the maximum message length—and thus cannot be used with confidence for large C. Second, nonces allow the invocation of AES to be carried out in parallel with most of the other operations in Poly1305-AES, reducing latency in many contexts. Third, most protocols have nonces anyway, for a variety of reasons: nonces are required for secure encryption, for example, and nonces allow trivial rejection of replayed messages.

I constrained r to simplify and accelerate implementations of Poly1305-AES in various contexts. A wider range of r—e.g., all 128-bit integers—would allow a quantitatively better security bound, but the current bound $DL/2^{106}$ will be perfectly satisfactory for the foreseeable future, whereas slower authenticator computations would not be perfectly satisfactory.

I chose little-endian instead of big-endian to improve overall performance. Little-endian saves time on the most popular CPUs (the Pentium and Athlon) while making no difference on most other CPUs (the PowerPC, for example, and the UltraSPARC).

The definition of Poly1305-AES could easily be extended from byte strings to bit strings, but there is no apparent benefit of doing so.

3 Security

This section discusses the security of Poly1305-AES.

Responsibilities of the user

Any protocol that uses Poly1305-AES must ensure unpredictability of the secret key (k, r). This section assumes that secret keys are chosen from the uniform distribution: i.e., probability 2^{-234} for each of the 2^{234} possible pairs (k, r).

Any protocol that uses Poly1305-AES must ensure that the secret key is, in fact, kept secret. This section assumes that all operations are independent of (k, r), except for the computation of authenticators by the sender and receiver.

(There are safe ways to reuse k for encryption, but those ways are not analyzed in this paper.)

The sender must *never* use the same nonce for two different messages. The simplest way to achieve this is for the sender to use an increasing sequence of nonces in, e.g., reverse-lexicographic order of 16-byte strings. (Problem: If a key is stored on disk, while increasing nonce values are stored in memory, what happens when the power goes out? Solution: Store a safe nonce value—a new nonce larger than any nonce used—on disk alongside the key.) Any protocol that uses Poly1305-AES must specify a mechanism of nonce generation and maintenance that prevents duplicates.

Security guarantee

Poly1305-AES guarantees that the only way for the attacker to find an (n, m, a) such that $a = \text{Poly1305}_r(m, \text{AES}_k(n))$, other than the authenticated messages (n, m, a) sent by the sender, is to break AES. If the attacker cannot break AES, and the receiver discards all (n, m, a) such that $a \neq \text{Poly1305}_r(m, \text{AES}_k(n))$, then the receiver will see only messages authenticated by the sender.

This guarantee is not limited to "meaningful" messages m. It is true even if the attacker can see all the authenticated messages sent by the sender. It is true even if the attacker can see whether the receiver accepts a forgery. It is true even if the attacker can influence the sender's choice of messages and unique nonces. (But it is not true if the nonce-uniqueness rule is violated.)

Here is a quantitative form of the guarantee. Assume that the attacker sees at most C authenticated messages and attempts at most D forgeries. Assume that the attacker has probability at most δ of distinguishing AES_k from a uniform random permutation after $C + D$ queries. Assume that all messages have length at most L. Then, with probability at least

$$1 - \delta - \frac{(1 - C/2^{128})^{-(C+1)/2} 8D\lceil L/16\rceil}{2^{106}},$$

all of the attacker's forgeries are discarded. In particular, if $C \leq 2^{64}$, then the attacker's chance of success is at most $\delta + 1.649 \cdot 8D\lceil L/16\rceil/2^{106} < \delta + 14D\lceil L/16\rceil/2^{106}$.

The most important design goal of AES was for δ to be small. There is, however, no hope of *proving* that δ is small. Perhaps AES will be broken someday. If that happens, users should switch to Poly1305-AnotherFunction. Poly1305-AnotherFunction provides the same security guarantee relative to the security of AnotherFunction.

Proof of the security guarantee

For each message m, write $\underline{m}$ for the polynomial $c_1 x^q + c_2 x^{q-1} + \cdots + c_q x^1$, where $q, c_1, c_2, \ldots, c_q$ are defined as in Section 2. Define $H_r(m)$ as the 16-byte unsigned little-endian representation of $(\underline{m}(r) \bmod 2^{130} - 5) \bmod 2^{128}$; note that H_r and

k are independent. Define a group operation $+$ on 16-byte strings as addition modulo 2^{128}, where each 16-byte string is viewed as the unsigned little-endian representation of an integer in $\{0, 1, 2, \ldots, 2^{128} - 1\}$. Then the authenticator $\text{Poly1305}_r(m, \text{AES}_k(n))$ is equal to $H_r(m) + \text{AES}_k(n)$.

The crucial property of H_r is that it has small differential probabilities: if g is a 16-byte string, and m, m' are distinct messages of length at most L, then $H_r(m) = H_r(m') + g$ with probability at most $8\lceil L/16 \rceil / 2^{106}$. See below.

Theorem 5.4 of [8] now guarantees that $H_r(m) + \text{AES}_k(n)$ is secure if AES is secure: specifically, that the attacker's success chance against $H_r(m) + \text{AES}_k(n)$ is at most $\delta + D(1 - C/2^{128})^{-(C+1)/2} 8\lceil L/16 \rceil / 2^{106}$.

The rest of this section is devoted to proving that H_r has small differential probabilities.

Theorem 3.1. $2^{130} - 5$ *is prime.*

Proof. Define $p_1 = (2^{130} - 6)/1517314646$ and $p_2 = (p_1 - 1)/222890620702$. Observe that 37003 and 221101 are prime divisors of $p_2 - 1$; $(37003 \cdot 221101)^2 > p_2$; $2^{p_2 - 1} - 1$ is divisible by p_2; $2^{(p_2-1)/37003} - 1$ and $2^{(p_2-1)/221101} - 1$ are coprime to p_2; $p_2^2 > p_1$; $2^{p_1 - 1} - 1$ is divisible by p_1; $2^{(p_1-1)/p_2} - 1$ is coprime to p_1; $p_1^2 > 2^{130} - 5$; $2^{2^{130}-6} - 1$ is divisible by $2^{130} - 5$; and $2^{(2^{130}-6)/p_1} - 1$ is coprime to $2^{130} - 5$. Hence p_2, p_1, and $2^{130} - 5$ are prime by Pocklington's theorem. $\square$

Theorem 3.2. *Let m and m' be messages. Let u be an integer. If the polynomial $\underline{m'} - \underline{m} - u$ is zero modulo $2^{130} - 5$ then $m = m'$.*

Proof. Define $c_1, c_2, \ldots, c_q$ as above, and define $c'_1, c'_2, \ldots, c'_{q'}$ for m' similarly.

If $q > q'$ then the coefficient of x^q in $\underline{m'} - \underline{m}$ is $0 - c_1$. By construction c_1 is in $\{1, 2, 3, \ldots, 2^{129}\}$, so it is nonzero modulo $2^{130} - 5$; contradiction. Thus $q \le q'$. Similarly $q \ge q'$. Hence $q = q'$.

If $i \in \{1, 2, \ldots, q\}$ then $c_i - c'_i$ is the coefficient of x^{q+1-i} in $\underline{m'} - \underline{m} - u$, which by hypothesis is divisible by $2^{130} - 5$. But $c_i - c'_i$ is between -2^{129} and 2^{129} by construction. Hence $c_i = c'_i$. In particular, $c_q = c'_q$.

Define ℓ as the number of bytes in m. Recall that $q = \lceil \ell/16 \rceil$; thus ℓ is between $16q - 15$ and $16q$. The exact value of ℓ is determined by q and c_q: it is $16q$ if $2^{128} \le c_q$, $16q - 1$ if $2^{120} \le c_q < 2^{121}$, $16q - 2$ if $2^{112} \le c_q < 2^{113}$, $\ldots$, $16q - 15$ if $2^8 \le c_q < 2^9$. Hence m' also has ℓ bytes.

Now consider any $j \in \{0, 1, \ldots, \ell - 1\}$. Write $i = \lfloor j/16 \rfloor + 1$; then $16i - 16 \le j \le 16i - 1$, and $1 \le i \le \lceil \ell/16 \rceil = q$, so $m[j] = \lfloor c_i / 2^{8(j-16i+16)} \rfloor \bmod 256 = \lfloor c'_i / 2^{8(j-16i+16)} \rfloor \bmod 256 = m'[j]$. Hence $m = m'$. $\square$

Theorem 3.3. *Let m, m' be distinct messages, each having at most L bytes. Let g be a 16-byte string. Let R be a subset of $\{0, 1, \ldots, 2^{130} - 6\}$. Then there are at most $8\lceil L/16 \rceil$ integers $r \in R$ such that $H_r(m) = H_r(m') + g$.*

Consequently, if $\#R = 2^{106}$, and if r is a uniform random element of R, then $H_r(m) = H_r(m') + g$ with probability at most $8\lceil L/16 \rceil / 2^{106}$.

Proof. Define U as the set of integers in $[-2^{130}+6, 2^{130}-6]$ congruent to g modulo 2^{128}. Note that $\#U \le 8$.

If $H_r(m) = H_r(m') + g$ then $(\underline{m'}(r) \bmod 2^{130}-5) - (\underline{m}(r) \bmod 2^{130}-5) \equiv g \pmod{2^{128}}$ so $(\underline{m'}(r) \bmod 2^{130}-5) - (\underline{m}(r) \bmod 2^{130}-5) = u$ for some $u \in U$. Hence r is a root of the polynomial $\underline{m'} - \underline{m} - u$ modulo the prime $2^{130}-5$. This polynomial is nonzero by Theorem 3.2, and has degree at most $\lceil L/16 \rceil$, so it has at most $\lceil L/16 \rceil$ roots modulo $2^{130}-5$. Sum over all $u \in U$: there are most $8\lceil L/16 \rceil$ possibilities for r. □

4 A floating-point implementation

This section explains how to compute $\text{Poly1305}_r(m, \text{AES}_k(n))$, given (k, r, n, m), at very high speeds on common general-purpose CPUs.

These techniques are used by my `poly1305aes` software library to achieve the Poly1305-AES speeds reported in Section 1. See Appendix A for further speed information. The software itself is available from `http://cr.yp.to/mac.html`.

The current version of `poly1305aes` includes separate implementations of Poly1305-AES for the Athlon, the Pentium, the PowerPC, and the UltraSPARC; it also includes a backup C implementation to handle other CPUs. This section focuses on the Athlon for concreteness.

Outline

The overall strategy to compute $\text{Poly1305}_r(m, \text{AES}_k(n))$ is as follows. Start by setting an accumulator h to 0. For each chunk c of the message m, first set $h \leftarrow h + c$, and then set $h \leftarrow rh$. Periodically reduce h modulo $2^{130}-5$, not necessarily to the smallest remainder but to something small enough to continue the computation. After all input chunks c are processed, fully reduce h modulo $2^{130}-5$, and add $\text{AES}_k(n)$.

Large-integer arithmetic in floating-point registers

Represent each of h, c, r as a sum of floating-point numbers, as in [7]. Specifically:

- As in Section 2, write r as $r_0 + r_1 + r_2 + r_3$ where $r_0 \in \{0, 1, 2, \ldots, 2^{28}-1\}$, $r_1/2^{32} \in \{0, 4, 8, \ldots, 2^{28}-4\}$, $r_2/2^{64} \in \{0, 4, 8, \ldots, 2^{28}-4\}$, and $r_3/2^{96} \in \{0, 4, 8, \ldots, 2^{28}-4\}$. Store each of r_0, r_1, r_2, r_3, $5 \cdot 2^{-130} r_1$, $5 \cdot 2^{-130} r_2$, $5 \cdot 2^{-130} r_3$ in memory in 8-byte floating-point format.
- Write each message chunk c as $d_0 + d_1 + d_2 + d_3$ where $d_0, d_1/2^{32}, d_2/2^{64} \in \{0, 1, 2, 3, \ldots, 2^{32}-1\}$ and $d_3/2^{96} \in \{0, 1, 2, 3, \ldots, 2^{34}-1\}$.
- Write h as $h_0 + h_1 + h_2 + h_3$ where h_i is a multiple of 2^{32i} in the range specified below. Store each h_i in one of the Athlon's floating-point registers.

Warning: The FreeBSD operating system starts each program by instructing the CPU to round all floating-point mantissas to 53 bits, rather than using the

CPU's natural 64-bit precision. Make sure to disable this instruction. Under `gcc`, for example, the code `asm volatile("fldcw %0"::"m"(0x137f))` specifies full 64-bit mantissas.

To set $h \leftarrow h+c$, set $h_0 \leftarrow h_0+d_0$, $h_1 \leftarrow h_1+d_1$, $h_2 \leftarrow h_2+d_2$, $h_3 \leftarrow h_3+d_3$. Before these additions, $h_0, h_1/2^{32}, h_2/2^{64}, h_3/2^{96}$ are required to be integers in $[-(63/128)\cdot 2^{64}, (63/128)\cdot 2^{64}]$. After these additions, $h_0, h_1/2^{32}, h_2/2^{64}, h_3/2^{96}$ are integers in $[-(127/256)\cdot 2^{64}, (127/256)\cdot 2^{64}]$.

Before multiplying h by r, reduce the range of each h_i by performing four parallel carries as follows:

- Define $\alpha_0 = 2^{95}+2^{94}$, $\alpha_1 = 2^{127}+2^{126}$, $\alpha_2 = 2^{159}+2^{158}$, and $\alpha_3 = 2^{193}+2^{192}$.
- Compute $y_i = \mathrm{fp}_{64}(h_i+\alpha_i)-\alpha_i$ and $x_i = h_i - y_i$. Here $\mathrm{fp}_{64}(h_i+\alpha_i)$ means the 64-bit-mantissa floating-point number closest to $h_i+\alpha_i$, with ties broken in the usual way; see [3]. Then $y_0/2^{32}, y_1/2^{64}, y_2/2^{96}, y_3/2^{130}$ are integers.
- Set $h_0 \leftarrow x_0 + 5\cdot 2^{-130}y_3$, $h_1 \leftarrow x_1+y_0$, $h_2 \leftarrow x_2+y_1$, and $h_3 \leftarrow x_3+y_2$.

This substitution changes h by $(2^{130}-5)2^{-130}y_3$, so it does not change h mod $2^{130}-5$. There are 17 floating-point operations here: 8 additions, 8 subtractions, and 1 multiplication by the constant $5\cdot 2^{-130}$.

Ranges: x_0, $x_1/2^{32}$, and $x_2/2^{64}$ are in $[-(1/2)\cdot 2^{32}, (1/2)\cdot 2^{32}]$; $x_3/2^{96}$ is in $[-2\cdot 2^{32}, 2\cdot 2^{32}]$; $y_0/2^{32}$, $y_1/2^{64}$, $y_2/2^{96}$, and $y_3/2^{128}$ are in $[-(127/256)\cdot 2^{32}, (127/256)\cdot 2^{32}]$; h_0 is in $[-(1147/1024)\cdot 2^{32}, (1147/1024)\cdot 2^{32}]$; $h_1/2^{32}$ is in $[-(255/256)\cdot 2^{32}, (255/256)\cdot 2^{32}]$; $h_2/2^{64}$ is in $[-(255/256)\cdot 2^{32}, (255/256)\cdot 2^{32}]$; $h_3/2^{96}$ is in $[-(639/256)\cdot 2^{32}, (639/256)\cdot 2^{32}]$.

To multiply h by r modulo $2^{130}-5$, replace (h_0, h_1, h_2, h_3) with

$$\begin{aligned}
(r_0h_0 &+ 5\cdot 2^{-130}r_1h_3 + 5\cdot 2^{-130}r_2h_2 + 5\cdot 2^{-130}r_3h_1,\\
r_0h_1 &+ r_1h_0 + 5\cdot 2^{-130}r_2h_3 + 5\cdot 2^{-130}r_3h_2,\\
r_0h_2 &+ r_1h_1 + r_2h_0 + 5\cdot 2^{-130}r_3h_3,\\
r_0h_3 &+ r_1h_2 + r_2h_1 + r_3h_0).
\end{aligned}$$

Recall that $2^{-34}r_1$, $2^{-66}r_2$, and $2^{-98}r_3$ are integers, so $2^{-130}r_1h_3$, $2^{-130}r_2h_2$, and $2^{-130}r_3h_1$ are integers; similarly, $2^{-130}r_2h_3$ and $2^{-130}r_3h_2$ are multiples of 2^{32}, and $2^{-130}r_3h_3$ is a multiple of 2^{64}. There are 28 floating-point operations here: 16 multiplications and 12 additions.

Ranges: $h_0, h_1/2^{32}, h_2/2^{64}, h_3/2^{96}$ are now integers of absolute value at most $2^{28}(1147/1024 + 2\cdot(5/4)255/256 + (5/4)639/256)2^{32} < (63/128)2^{64}$, ready for the next iteration of the inner loop.

Note that the carries can be omitted on the first loop: d_0 is an integer in $[0, 2^{32}]$; $d_1/2^{32}$ is an integer in $[0, 2^{32}]$; $d_2/2^{64}$ is an integer in $[0, 2^{32}]$; $d_3/2^{96}$ is an integer in $[0, 3\cdot 2^{32}]$; and $2^{28}(1+(5/4)+(5/4)+(5/4)3)2^{32} < (63/128)2^{64}$.

Output conversion

After the last message chunk is processed, carry one last time, to put h_0, h_1, h_2, h_3 into the small ranges listed above.

Add $2^{130} - 2^{97}$ to h_3; add $2^{97} - 2^{65}$ to h_2; add $2^{65} - 2^{33}$ to h_1; and add $2^{33} - 5$ to h_0. This makes each h_i positive, and puts $h = h_0 + h_1 + h_2 + h_3$ into the range $\{0, 1, \ldots, 2(2^{130} - 5) - 1\}$.

Perform a few integer add-with-carry operations to convert the accumulator into a series of 32-bit words in the usual form. Subtract $2^{130} - 5$, and keep the result if it is nonnegative, being careful to use constant-time operations so that no information is leaked through timing.

Finally, add $\mathrm{AES}_k(n)$. There are two reasons to pay close attention to the AES computation:

- It is extremely difficult to write high-speed *constant-time* AES software. Typical AES software leaks key bytes to the simplest conceivable timing attack. See [6]. My new AES implementations go to extensive effort to reduce the AES timing variability.
- The time to compute $\mathrm{AES}_k(n)$ from (k, n) is more than half of the time to compute $\mathrm{Poly1305}_r(m, \mathrm{AES}_k(n))$ for short messages, and remains quite noticeable for longer messages. My new AES implementations are, as far as I know, the fastest available software for computing $\mathrm{AES}_k(n)$ from (k, n). Of course, *if* there is room in cache, then one can save some time by instead computing $\mathrm{AES}_k(n)$ from (K, n), where K is a pre-expanded version of k.

Details of the AES computation are not discussed in this paper but are discussed in the `poly1305aes` documentation.

Instruction selection and scheduling

Consider an integer (such as d_0) between 0 and $2^{32} - 1$, stored in the usual way as four bytes. How does one load the integer into a floating-point register, when the Athlon does not have a load-four-byte-unsigned-integer instruction? Here are three possibilities:

- Concatenate the four bytes with $(0, 0, 0, 0)$, and use the Athlon's load-eight-byte-signed-integer instruction. Unfortunately, the four-byte store forces the eight-byte load to wait for dozens of cycles.
- Concatenate the bytes with $(0, 0, 56, 67)$, producing an eight-byte floating-point number. Load that number, and subtract $2^{52}+2^{51}$ to obtain the desired integer. This well-known trick has the virtue of also allowing the integer to be scaled by (e.g.) 2^{32}: replace 67 with 69 and $2^{52} + 2^{51}$ with $2^{84} + 2^{83}$. Unfortunately, as above, the four-byte store forces the eight-byte load to wait for dozens of cycles.
- Subtract 2^{31} from the integer, use the Athlon's load-four-byte-signed-integer instruction, and add 2^{31} to the result. This has smaller latency, but puts more pressure on the floating-point unit.

Top performance requires making the right choice.

(A variant of Poly1305-AES using signed 32-bit integers would save time on the Athlon. On the other hand, it would lose time on typical 64-bit CPUs.)

This is merely one example of several low-level issues that can drastically affect speed: instruction selection, instruction scheduling, register assignment, instruction fetching, etc. A "fast" implementation of Poly1305-AES, with just a few typical low-level mistakes, will use twice as many cycles per byte as the software described here.

Other modern CPUs

The same floating-point operations also run at high speed on the Pentium 1, Pentium MMX, Pentium Pro, Pentium II, Pentium III, Pentium 4, Pentium M, Celeron, Duron, et al.

The UltraSPARC, PowerPC, et al. support fast arithmetic on floating-point numbers with 53-bit, rather than 64-bit, mantissas. The simplest way to achieve good performance on these chips is to break a 32-bit number into two 16-bit pieces before multiplying it by another 32-bit number.

As in the case of the Athlon, careful attention to low-level CPU details is necessary for top performance.

5 Other implementation strategies

Some people, upon hearing that there is a tricky way to use the Athlon's floating-point unit to compute a function quickly, leap to the unjustified conclusion that the same function cannot be computed quickly except on an Athlon. Consider, for example, the incorrect statement "hash-127 needs good hardware support for a fast implementation" in [17, footnote 3].

This section outlines three non-floating-point methods to compute Poly1305-AES, and indicates contexts where the methods are useful.

Integer registers

The 130-bit accumulator in Poly1305-AES can be spread among several integer registers rather than several floating-point registers.

This is good for low-end CPUs that do not support floating-point operations but that still have reasonably fast integer multipliers. It is also good for some high-end CPUs, such as the Athlon 64, that offer faster multiplication through integer registers than through floating-point registers.

Tables

One can make a table of the integers $r, 2r, 4r, 8r, \ldots, 2^{129}r$ modulo $2^{130} - 5$, and then multiply any 130-bit integer by r by adding, on average, about 65 elements of the table.

One can reduce the amount of work by using both additions and subtractions, by increasing the table size, and by choosing table entries more carefully. For example, one can include $3r, 24r, 192r, \ldots$ in the table, and then multiply any

130-bit integer by r by adding and subtracting, on average, about 38 elements of the table. This is a special case of an algorithm often credited to Brickell, Gordon, McCurley, Wilson, Lim, and Lee, but actually introduced much earlier by Pippenger in [23].

One can also balance the table size against the effort in reduction modulo $2^{130} - 5$. Consider, for example, the table $r, 2r, 3r, 4r, \ldots, 255r$.

Table lookups are often the best approach for tiny CPUs that do not have any fast multiplication operations. Of course, their key agility is poor, and they are susceptible to timing attacks if they are not implemented very carefully.

Special-purpose circuits

An 1800MHz AMD Duron, costing under $50, can feed 4 gigabits per second of 1500-byte messages through Poly1305-AES with the software discussed in Section 4. Hardware implementations of Poly1305-AES can strip away a great deal of unnecessary cost: the multiplier is only part of the cost of the Duron; furthermore, some of the multiplications are by sparse constants; furthermore, only about 20% of the multiplier area is doing any useful work, since each input is much smaller than 64 bits; furthermore, almost all carries can be deferred until the end of the Poly1305-AES computation, rather than being performed after each multiplication; furthermore, hardware implementations need not, and should not, imitate traditional software structures—one can directly build a fast multiplier modulo $2^{130}-5$, taking advantage of more sophisticated multiplication algorithms than those used in the Duron. Evidently Poly1305-AES can handle next-generation Ethernet speeds at reasonable cost.

References

1. —, *17th annual symposium on foundations of computer science*, IEEE Computer Society, Long Beach, California, 1976. MR 56:1766.
2. —, *20th annual symposium on foundations of computer science*, IEEE Computer Society, New York, 1979. MR 82a:68004.
3. —, *IEEE standard for binary floating-point arithmetic*, Standard 754–1985, Institute of Electrical and Electronics Engineers, New York, 1985.
4. Valentine Afanassiev, Christian Gehrmann, Ben Smeets, *Fast message authentication using efficient polynomial evaluation*, in [10] (1997), 190–204. URL: `http://cr.yp.to/bib/entries.html#1997/afanassiev`.
5. Daniel J. Bernstein, *Guaranteed message authentication faster than MD5 (abstract)* (1999). URL: `http://cr.yp.to/papers.html#hash127-abs`.
6. Daniel J. Bernstein, *Cache-timing attacks on AES* (2004). URL: `http://cr.yp.to/papers.html#cachetiming`. ID `cd9faae9bd5308c440df50fc26a517b4`.
7. Daniel J. Bernstein, *Floating-point arithmetic and message authentication* (2004). URL: `http://cr.yp.to/papers.html#hash127`. ID `dabadd3095644704c5cbe9690ea3738e`.
8. Daniel J. Bernstein, *Stronger security bounds for Wegman-Carter-Shoup authenticators*, Proceedings of Eurocrypt 2005, to appear (2005). URL: `http://cr.yp.to/papers.html#securitywcs`. ID `2d603727f69542f30f7da2832240c1ad`.

9. Jürgen Bierbrauer, Thomas Johansson, Gregory Kabatianskii, Ben Smeets, *On families of hash functions via geometric codes and concatenation*, in [30] (1994), 331–342. URL: `http://cr.yp.to/bib/entries.html#1994/bierbrauer`.
10. Eli Biham (editor), *Fast Software Encryption '97*, Lecture Notes in Computer Science, 1267, Springer-Verlag, Berlin, 1997. ISBN 3–540–63247–6.
11. John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, Phillip Rogaway, *UMAC: fast and secure message authentication*, in [34] (1999), 216–233. URL: `http://www.cs.ucdavis.edu/~rogaway/umac/`.
12. Gilles Brassard, *On computationally secure authentication tags requiring short secret shared keys*, in [13] (1983), 79–86. URL: `http://cr.yp.to/bib/entries.html#1983/brassard`.
13. David Chaum, Ronald L. Rivest, Alan T. Sherman (editors), *Advances in cryptology: proceedings of Crypto 82*, Plenum Press, New York, 1983. ISBN 0–306–41366–3. MR 84j:94004.
14. Bert den Boer, *A simple and key-economical unconditional authentication scheme*, Journal of Computer Security **2** (1993), 65–71. ISSN 0926–227X. URL: `http://cr.yp.to/bib/entries.html#1993/denboer`.
15. Edgar N. Gilbert, F. Jessie MacWilliams, Neil J. A. Sloane, *Codes which detect deception*, Bell System Technical Journal **53** (1974), 405–424. ISSN 0005–8580. MR 55:5306. URL: `http://cr.yp.to/bib/entries.html#1974/gilbert`.
16. Torbjorn Granlund (editor), *GMP 4.1.2: GNU multiple precision arithmetic library* (2004). URL: `http://www.swox.com/gmp/`.
17. Shai Halevi, Phil Rogaway, *A tweakable enciphering mode* (2003). URL: `http://www.research.ibm.com/people/s/shaih/pubs/hr03.html`.
18. Michael Kaminski, *A linear time algorithm for residue computation and a fast algorithm for division with a sparse divisor*, Journal of the ACM **34** (1987), 968–984. ISSN 0004–5411. MR 89f:68033.
19. Richard M. Karp, Michael O. Rabin, *Efficient randomized pattern-matching algorithms*, IBM Journal of Research and Development **31** (1987), 249–260. ISSN 0018–8646. URL: `http://cr.yp.to/bib/entries.html#1987/karp`.
20. Neal Koblitz (editor), *Advances in cryptology—CRYPTO '96*, Lecture Notes in Computer Science, 1109, Springer-Verlag, Berlin, 1996.
21. Ted Krovetz, Phillip Rogaway, *Fast universal hashing with small keys and no preprocessing: the PolyR construction* (2000). URL: `http://www.cs.ucdavis.edu/~rogaway/papers/poly.htm`.
22. Wim Nevelsteen, Bart Preneel, *Software performance of universal hash functions*, in [29] (1999), 24–41.
23. Nicholas Pippenger, *On the evaluation of powers and related problems (preliminary version)*, in [1] (1976), 258–263; newer version split into [24] and [25]. MR 58:3682. URL: `http://cr.yp.to/bib/entries.html#1976/pippenger`.
24. Nicholas Pippenger, *The minimum number of edges in graphs with prescribed paths*, Mathematical Systems Theory **12** (1979), 325–346; see also older version [23]. ISSN 0025–5661. MR 81e:05079. URL: `http://cr.yp.to/bib/entries.html#1979/pippenger`.
25. Nicholas Pippenger, *On the evaluation of powers and monomials*, SIAM Journal on Computing **9** (1980), 230–250; see also older version [23]. ISSN 0097–5397. MR 82c:10064. URL: `http://cr.yp.to/bib/entries.html#1980/pippenger`.
26. Michael O. Rabin, *Fingerprinting by random polynomials*, Harvard Aiken Computational Laboratory TR-15-81 (1981). URL: `http://cr.yp.to/bib/entries.html#1981/rabin`.

27. Victor Shoup, *On fast and provably secure message authentication based on universal hashing*, in [20] (1996), 313–328; see also newer version [28].
28. Victor Shoup, *On fast and provably secure message authentication based on universal hashing* (1996); see also older version [27]. URL: `http://www.shoup.net/papers`.
29. Jacques Stern (editor), *Advances in cryptology: EUROCRYPT '99*, Lecture Notes in Computer Science, 1592, Springer-Verlag, Berlin, 1999. ISBN 3–540–65889–0. MR 2000i:94001.
30. Douglas R. Stinson (editor), *Advances in cryptology—CRYPTO '93: 13th annual international cryptology conference, Santa Barbara, California, USA, August 22–26, 1993, proceedings*, Lecture Notes in Computer Science, 773, Springer-Verlag, Berlin, 1994. ISBN 3–540–57766–1, 0–387–57766–1. MR 95b:94002.
31. Richard Taylor, *An integrity check value algorithm for stream ciphers*, in [30] (1994), 40–48. URL: `http://cr.yp.to/bib/entries.html#1994/taylor`.
32. Mark N. Wegman, J. Lawrence Carter, *New classes and applications of hash functions*, in [2] (1979), 175–182; see also newer version [33]. URL: `http://cr.yp.to/bib/entries.html#1979/wegman`.
33. Mark N. Wegman, J. Lawrence Carter, *New hash functions and their use in authentication and set equality*, Journal of Computer and System Sciences **22** (1981), 265–279; see also older version [32]. ISSN 0022–0000. MR 82i:68017. URL: `http://cr.yp.to/bib/entries.html#1981/wegman`.
34. Michael Wiener (editor), *Advances in cryptology—CRYPTO '99*, Lecture Notes in Computer Science, 1666, Springer-Verlag, Berlin, 1999. ISBN 3–5540–66347–9. MR 2000h:94003.

A Appendix: Speed graphs

These graphs show the time to verify an authenticator in various situations. The horizontal axis on the graphs is message length, from 0 bytes to 4096 bytes. The vertical axis on the graphs is time, from 0 CPU cycles to 24576 CPU cycles; time includes function-call overhead, timing overhead, etc. The bottom-left-to-top-right diagonal is 6 CPU cycles per byte. Color scheme:

- Non-reddish (black, green, dark blue, light blue): Keys are in cache.
- Reddish (red, yellow, purple, gray): Keys are not in cache. Loading the keys from DRAM takes extra time.
- Non-greenish (black, red, dark blue, purple): Messages, authenticators, and nonces are in cache.
- Greenish (green, yellow, light blue, gray): Messages, authenticators, and nonces are not in cache. Loading the data from DRAM takes extra time, typically growing with the message length.
- Non-blueish (black, red, green, yellow): Keys, message, authenticators, and nonces are aligned.
- Blueish (dark blue, purple, light blue, gray): Keys, message, authenticators, and nonces are unaligned. This hurts some CPUs.

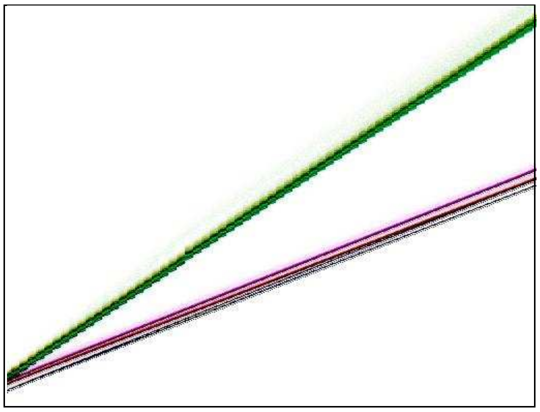
The graphs include code in cache and code out of cache, with no color change. The out-of-cache case costs between 10000 and 30000 cycles, depending on the CPU; it is often faintly visible as a cloud above the in-cache case.

Lengths divisible by 16 are slightly faster than lengths not divisible by 16. The best case in (almost) every graph is length divisible by 16, everything in cache, everything aligned; this case is visible as 256 black dots at the bottom of the graph.

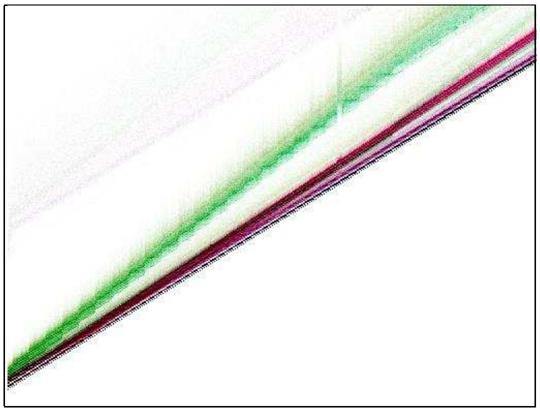
In black-and-white printouts, the keys-not-in-cache case is a slightly higher line at the same slope; the data-not-in-cache case is a line at a considerably higher slope; the unaligned case is a line at a slightly higher slope.

See `http://cr.yp.to/mac/speed.html` for much more speed information.

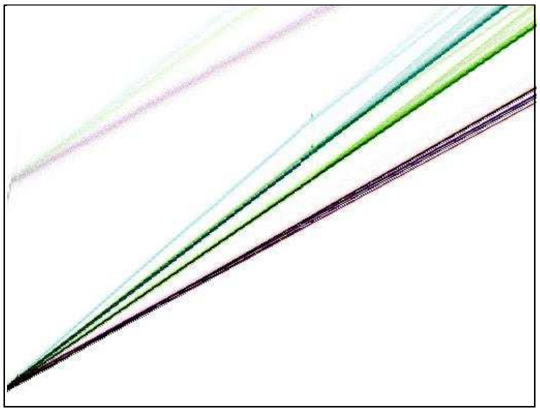
AMD Athlon, 900MHz:



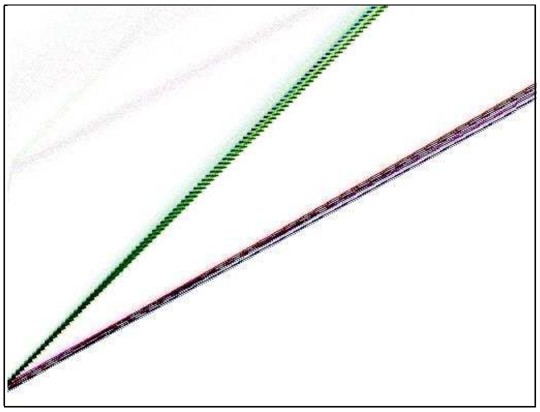
IBM PowerPC RS64 IV, 668MHz:



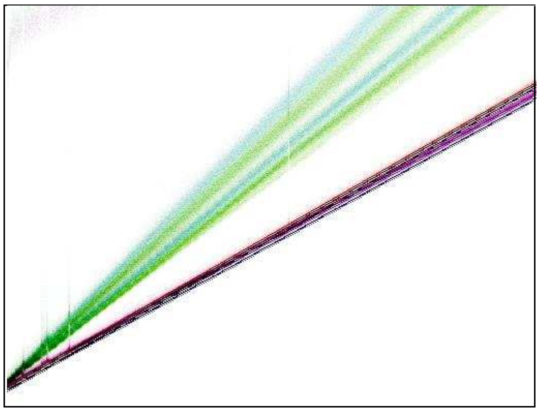
Intel Pentium III, 500MHz:



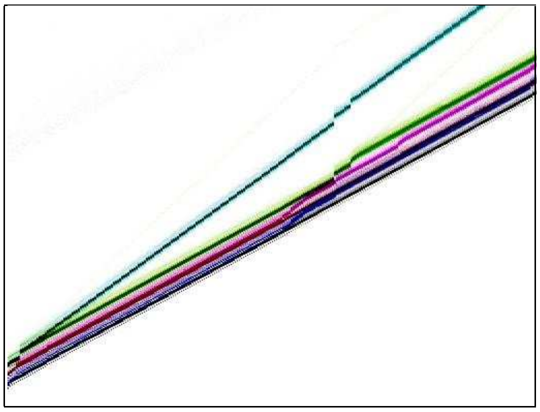
Intel Pentium III, 850MHz:



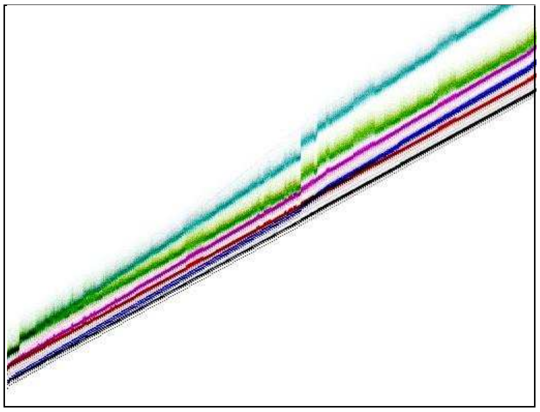
Intel Pentium III, 1000MHz:



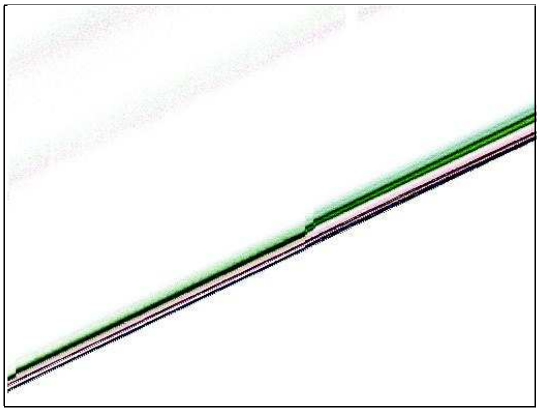
Intel Pentium 4, 1900MHz:



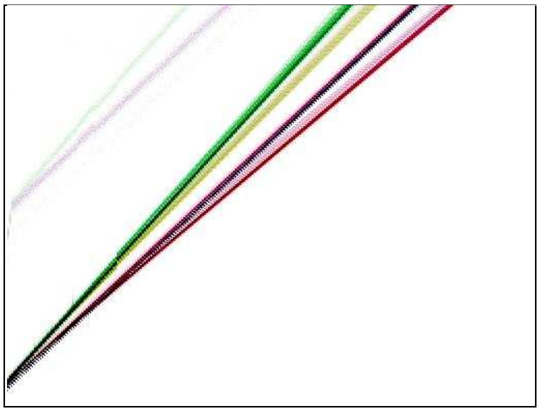
Intel Pentium 4, 3400MHz:



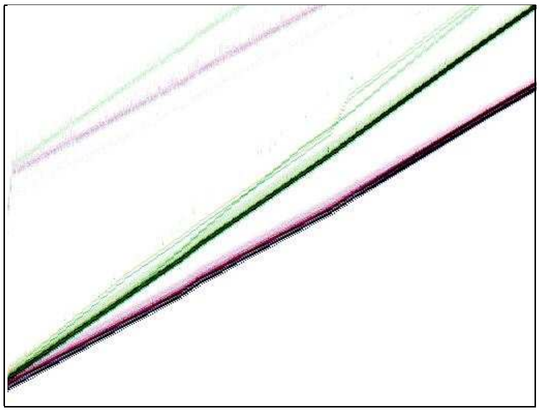
Intel Pentium M, 1300MHz:



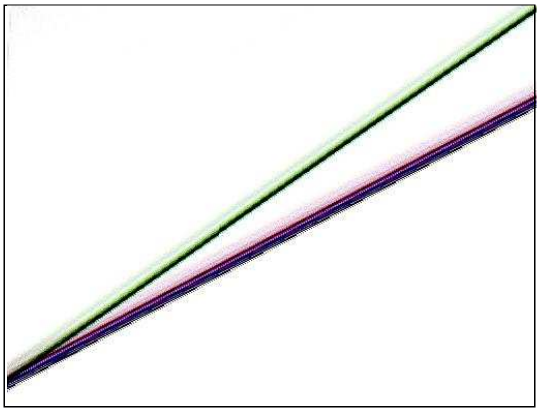
Motorola PowerPC 7410, 533MHz:



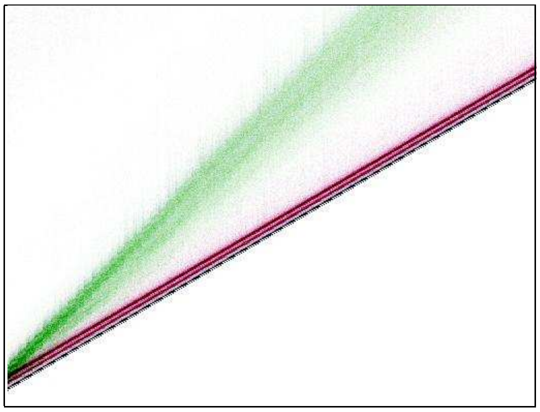
Sun UltraSPARC II, 296MHz:



Sun UltraSPARC IIi, 360MHz:



Sun UltraSPARC III, 900MHz:



Two notes: 1. The load-keys-from-DRAM penalty (red) is quite small, thanks to Poly1305-AES's key agility. On the PowerPC 7410, keys in cache are *slower* than keys out of cache, presumably because of a cache-associativity accident that slightly more sophisticated code will be able to avoid.

2. The load-data-from-DRAM penalty (green) is generally quite noticeable. I have not yet experimented with prefetch instructions. But the penalty is small on the Pentium 4 and almost invisible on the Pentium M; the Pentium M does a good job of figuring out for itself which data to prefetch.

B Appendix: Examples

The following table, with all integers on the right displayed in hexadecimal, illustrates authenticator computations for strings of length 2, 0, 32, and 63. The notation $\underline{m}(r)$ means $c_1 r^q + c_2 r^{q-1} + \cdots + c_q r^1$. A much more extensive test suite appears in `http://cr.yp.to/mac/test.html`.

m	`f3 f6`
c_1	`00000000000000000000000000001f6f3`
r	`85 1f c4 0c 34 67 ac 0b e0 5c c2 04 04 f3 f7 00`
$\underline{m}(r) \bmod 2^{130}-5$	`321e58e25a69d7f8f27060770b3f8bb9c`
k	`ec 07 4c 83 55 80 74 17 01 42 5b 62 32 35 ad d6`
n	`fb 44 73 50 c4 e8 68 c5 2a c3 27 5c f9 d4 32 7e`
$\mathrm{AES}_k(n)$	`58 0b 3b 0f 94 47 bb 1e 69 d0 95 b5 92 8b 6d bc`
$\mathrm{Poly1305}_r(m, \mathrm{AES}_k(n))$	`f4 c6 33 c3 04 4f c1 45 f8 4f 33 5c b8 19 53 de`
m	
r	`a0 f3 08 00 00 f4 64 00 d0 c7 e9 07 6c 83 44 03`
$\underline{m}(r) \bmod 2^{130}-5$	`000000000000000000000000000000000`
k	`75 de aa 25 c0 9f 20 8e 1d c4 ce 6b 5c ad 3f bf`
n	`61 ee 09 21 8d 29 b0 aa ed 7e 15 4a 2c 55 09 cc`
$\mathrm{AES}_k(n)$	`dd 3f ab 22 51 f1 1a c7 59 f0 88 71 29 cc 2e e7`
$\mathrm{Poly1305}_r(m, \mathrm{AES}_k(n))$	`dd 3f ab 22 51 f1 1a c7 59 f0 88 71 29 cc 2e e7`
m	`66 3c ea 19 0f fb 83 d8 95 93 f3 f4 76 b6 bc 24`
	`d7 e6 79 10 7e a2 6a db 8c af 66 52 d0 65 61 36`
c_1	`124bcb676f4f39395d883fb0f19ea3c66`
c_2	`1366165d05266af8cdb6aa27e1079e6d7`
r	`48 44 3d 0b b0 d2 11 09 c8 9a 10 0b 5c e2 c2 08`
$\underline{m}(r) \bmod 2^{130}-5$	`1cfb6f98add6a0ea7c631de020225cc8b`
k	`6a cb 5f 61 a7 17 6d d3 20 c5 c1 eb 2e dc dc 74`
n	`ae 21 2a 55 39 97 29 59 5d ea 45 8b c6 21 ff 0e`
$\mathrm{AES}_k(n)$	`83 14 9c 69 b5 61 dd 88 29 8a 17 98 b1 07 16 ef`
$\mathrm{Poly1305}_r(m, \mathrm{AES}_k(n))$	`0e e1 c1 6b b7 3f 0f 4f d1 98 81 75 3c 01 cd be`
m	`ab 08 12 72 4a 7f 1e 34 27 42 cb ed 37 4d 94 d1`
	`36 c6 b8 79 5d 45 b3 81 98 30 f2 c0 44 91 fa f0`
	`99 0c 62 e4 8b 80 18 b2 c3 e4 a0 fa 31 34 cb 67`
	`fa 83 e1 58 c9 94 d9 61 c4 cb 21 09 5c 1b f9`
c_1	`1d1944d37edcb4227341e7f4a721208ab`
c_2	`1f0fa9144c0f2309881b3455d79b8c636`
c_3	`167cb3431faa0e4c3b218808be4620c99`
c_4	`001f91b5c0921cbc461d994c958e183fa`
r	`12 97 6a 08 c4 42 6d 0c e8 a8 24 07 c4 f4 82 07`
$\underline{m}(r) \bmod 2^{130}-5$	`0c3c4f37c464bbd44306c9f8502ea5bd1`
k	`e1 a5 66 8a 4d 5b 66 a5 f6 8c c5 42 4e d5 98 2d`
n	`9a e8 31 e7 43 97 8d 3a 23 52 7c 71 28 14 9e 3a`
$\mathrm{AES}_k(n)$	`80 f8 c2 0a a7 12 02 d1 e2 91 79 cb cb 55 5a 57`
$\mathrm{Poly1305}_r(m, \mathrm{AES}_k(n))$	`51 54 ad 0d 2c b2 6e 01 27 4f c5 11 48 49 1f 1b`