Cycle counts for authenticated encryption

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

System	Cipher key bits	Cipher	MAC	Total key bits
abc-v3-poly1305	128	ABC v3	Poly1305	256
aes-128-poly1305	128	10-round AES	Poly1305	256
aes-256-poly1305	256	14-round AES	Poly1305	384
cryptmt-v3-poly1305	256	CryptMT 3	Poly1305	384
dicing-p2-poly1305	256	DICING P2	Poly1305	384
dragon-poly1305	256	Dragon	Poly1305	384
grain-128-poly1305	128	Grain-128	Poly1305	256
grain-v1-poly1305	80	Grainv1	Poly1305	208
hc-128-poly1305	128	HC-128	Poly1305	256
hc-256-poly1305	256	HC-256	Poly1305	384
lex-v1-poly1305	128	LEX v1	Poly1305	256
mickey-128-2-poly1305	128	MICKEY-128 2.0	Poly1305	256
nls-ae	128	NLS	built-in	128
nls-poly1305	128	NLS	Poly1305	256
phelix	256	Phelix	built-in	256
polarbear-2-poly1305	128	Polar Bear 2.0	Poly1305	256
py6-poly1305	256	Py6	Poly1305	384
py-poly1305	256	Py	Poly1305	384
pypy-poly1305	256	Pypy	Poly1305	384
rabbit-poly1305	128	Rabbit	Poly1305	256
rc4-poly1305	256	RC4	Poly1305	384
salsa20-8-poly1305	256	Salsa20/8	Poly1305	384
salsa20-12-poly1305	256	Salsa20/12	Poly1305	384
salsa20-poly1305	256	Salsa20	Poly1305	384
snow-2.0-poly1305	256	SNOW 2.0	Poly1305	384
sosemanuk-poly1305	256	SOSEMANUK	Poly1305	384
trivium-poly1305	80	TRIVIUM	Poly1305	208

* Date of this document: 2007.01.12. Permanent ID of this document: be6b4df07eb1ae67aba9338991b78388.

Abstract. How much time is needed to encrypt, authenticate, verify, and decrypt a message? The answer depends on the machine (most importantly, but not solely, the CPU), on the choice of authenticated-encryption function, on the message length, on the level of competition for the instruction cache, on the number of keys handled in parallel, et al. This paper reports, in graphical and tabular form, measurements of the speeds of a wide variety of authenticated-encryption functions on a wide variety of CPUs.

This paper reports speed measurements for the secret-key authenticated-encryption systems listed on the first page. I included all of the "hardware focus" ciphers in phase 2 of eSTREAM, the ECRYPT Stream Cipher Project: Grain, MICKEY, Phelix, and Trivium. I also included all of the "software" ciphers in phase 2 of eSTREAM: not just the "focus" ciphers DRAGON, HC, LEX, Phelix, Py, Salsa20, and SOSEMANUK, but also ABC, CryptMT, DICING, NLS, Polar Bear, and Rabbit.

I did not exclude ciphers for which there are claims of attacks: ABC, NLS, Py, and RC4. For LEX, I chose version 1 (for which there is a claim of an attack) rather than version 2 (for which there are no such claims) because I'm not aware of functioning software for version 2 of LEX; my impression is that the versions will have similar speeds, but speculation is no substitute for measurement.

I included counter-mode AES, the Advanced Encryption Standard, as a basis for comparison, along with SNOW 2.0 and RC4.

Non-authenticating stream ciphers

Most of the stream ciphers do not include message authentication. I converted each non-authenticating stream cipher into an authenticated-encryption system by combining it in a standard way with Poly1305, a state-of-the-art message-authentication code.

Here are the details: The key for the authenticated-encryption system is (r, k) where r a 16-byte Poly1305 key and k is a key for the non-authenticating stream cipher F. The authenticated encryption of a message m with nonce n is $(\text{Poly1305}_r(c, s), c)$ where $(s, c) = F_k(n) \oplus (0, m)$, both s and 0 having 16 bytes. Here $F_k(n)$ is the "keystream" produced by F using key k and nonce n; this keystream is implicitly truncated to same length as $(0, m)$.

Previous eSTREAM benchmarks did not include separate authenticators; they simply reported encryption timings for non-authenticating ciphers along with encryption timings for authenticating ciphers. The reality is that users need authenticated encryption, not just encryption, so they need to combine non-authenticating ciphers with message-authentication codes, slowing down those ciphers. How quickly do these combined systems handle legitimate packets, and how quickly do they reject forged packets? Are they faster than ciphers with built-in authentication? To compare the speeds of authenticating ciphers and non-authenticating ciphers from the user's perspective, benchmarks must take the extra authentication time into account.

"Isn't this a purely academic question?" one might ask. "Haven't all the authenticating ciphers been broken? Frogbit flunks a simple IV-diffusion test. Courtois broke SFINKS. Cho and Piperzyk broke both versions of NLS. Wu and Preneel broke Phelix. Okay, okay, VEST is untouched, but it's much too expensive for anyone to want to use." The simplest response is that, in fact, Phelix has not been broken. (The Wu-Preneel "attack" ignores both the concept of a nonce and the standard definition of cipher security; the "attack" assumes that senders repeat nonces. The same silly assumption easily "breaks" every eSTREAM submission.) Phelix remains one of the top eSTREAM submissions.

I'm planning future work to extend my database of timings to cover other authenticated-encryption systems. I plan to include more ciphers, for example; I plan to include other modes of use of Poly1305; and I plan to include UMAC, VMAC, CBC-MAC, and HMAC-SHA-1 as alternatives to Poly1305. I will also endeavor to incorporate improved implementations of systems already covered: for example, I'm planning a 64-bit implementation of Poly1305. But the existing data should already be useful in comparing eSTREAM candidates.

"Why is it necessary to time authenticated encryption?" one might ask. "If you want a table of authenticated-encryption timings, why not simply add a table of authentication timings to a table of encryption timings?" Response: The existing tables are deficient. This paper's timings are much more comprehensive than previous encryption timings. This paper systematically measures all packet lengths in a wide range, for example, and systematically measures multiple-key cache-miss costs. Furthermore, adding all the contributing times isn't as easy as it sounds; for example, if the authentication software uses more than half of the code cache, and the encryption software uses more than half of the code cache, authenticated encryption will need time for code-cache misses. Component benchmarks can be interesting and informative, but whole-function benchmarks are the simplest way to ensure that no components are forgotten.

API for authenticated-encryption systems

What does a secret-key authenticated-encryption system do for the user? It takes keys; it encrypts and authenticates each outgoing packet; it verifies and decrypts each incoming packet. So I specified an authenticated-encryption API with three functions: `makekey` to generate a key (and an "expanded key," the output of any desired precomputation); `encrypt` to authenticate and encrypt an outgoing packet; and `decrypt` to verify and decrypt an incoming packet.

The `encrypt` function includes an authenticator in its encrypted output packet. The `decrypt` function is given an encrypted packet allegedly produced by `encrypt`; it rejects the packet if the authenticator is wrong. Many systems can limit their decryption work for long messages when the authenticator is wrong. In particular, for the Poly1305 combination described above, an authenticator can be checked as soon as 16 bytes of keystream have been generated; if the authenticator is wrong then one can skip the work of generating the remaining bytes of keystream.

In contrast, in the official eSTREAM stream-cipher API, both `encrypt` and `decrypt` put an authenticator somewhere else. It is the responsibility of the `decrypt` user to verify authenticators. Having `decrypt` write an authenticator, rather than read it, means that rejection of forged packets is necessarily just as slow as decryption of legitimate packets. This doesn't seem to have been a problem for the authenticating stream ciphers submitted to eSTREAM, but it unnecessarily slows down other authenticated-encryption systems.

There are many other details of the API, but this paper can be read without regard to those details. Example: `encrypt` and `decrypt` receive lengths as 64-bit integers (`long long` in C). On many CPUs, using fewer bits for lengths would save a few cycles, marginally shifting the graphs in this paper.

Tools for benchmarking

Previous eSTREAM speed reports use the official eSTREAM benchmarking toolkit. The toolkit includes (1) software written by Christophe de Cannière to measure the speeds of stream-cipher implementations that follow the official eSTREAM stream-cipher API and (2) stream-cipher implementations collected from cipher authors.

To collect the timings reported in this paper I wrote a new benchmarking toolkit, `ciphercycles`, available from `http://cr.yp.to/streamciphers.html`. I wrote a separate tool to convert stream ciphers from the official eSTREAM stream-cipher API to my new API (and in particular to add authentication to the non-authenticating stream ciphers); the resulting implementations are included in the toolkit. Subsequent updates to the implementations in the official eSTREAM benchmarking toolkit will be easy to reflect in `ciphercycles`.

Many portions of `ciphercycles` are derived from BATMAN (Benchmarking of Asymmetric Tools on Multiple Architectures, Non-Interactively), a public-key benchmarking toolkit that I wrote for eBATS (ECRYPT Benchmarking of Asymmetric Systems). The new speed reports produced by `ciphercycles`, like the eBATS speed reports, are in a simple format designed for easy computer processing. I'm planning future work to integrate benchmarking projects.

The timings collected by `ciphercycles` include (authenticated) encryption, (verified) decryption of legitimately encrypted packets, and rejection of forged packets. Decryption times are usually almost identical to encryption times, but rejection times are often much smaller, for the reasons discussed above. The official eSTREAM timings include only encryption times.

The timings collected by `ciphercycles` systematically cover each packet length between 0 bytes and 8192 bytes. By superimposing graphs one can easily see the message-length cutoffs between different ciphers. The official eSTREAM timings include only a few selected lengths (40 bytes, 576 bytes, 1500 bytes, long), hiding block-size penalties and many other length-dependent effects.

The timings collected by `ciphercycles` include benchmarks for encryption of short packets bouncing between multiple keys: for example, when there are 1024 active keys, how many cycles are used for encryption of a 775-byte packet under a random choice of key, including the cache misses needed to access the

key? The official eSTREAM timings include one fuzzy "agility" number for each cipher but are otherwise dedicated to single-key benchmarks.

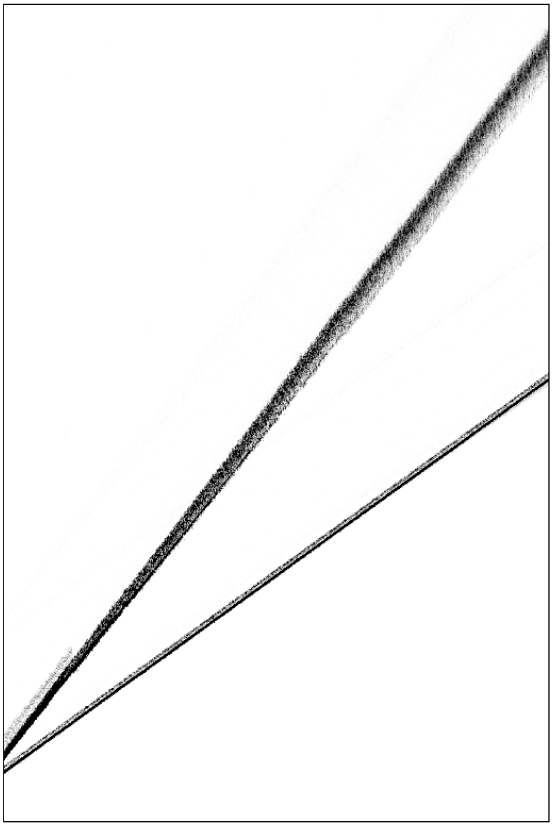
The timings collected by `ciphercycles` also include `makekey` timings, but those timings are not reported in this paper.

Graphs

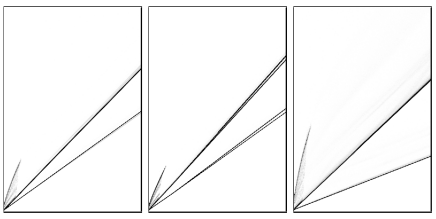
The sample graph on the left below shows timings for the `abc-v3-poly1305` system on a 900MHz AMD Athlon (622) computer named `thoth`.

The horizontal axis is packet length, between 0 bytes and 8192 bytes. The vertical axis is time, between 0 cycles and 98304 cycles. The diagonal from the lower left corner of the graph to the upper right corner is 12 cycles per byte.

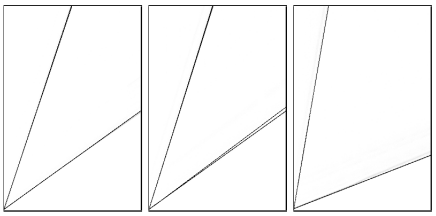
The two main lines visible on the graph are (1) roughly 7 cycles per byte for encryption and decryption and (2) roughly 3 cycles per byte for rejection. Faint lines are visible above the main lines; there are 15 timings for each packet length, and initial timings are slightly slower because of cache misses. There is also a short curve up the left side of the graph for encrypting packets of ≤ 1024 bytes using a random key from a pool of 1024 active keys. Also plotted, and faintly visible, are packet lengths of ≤ 960 bytes for 512 active keys, packet lengths of ≤ 896 bytes for 256 active keys, etc.



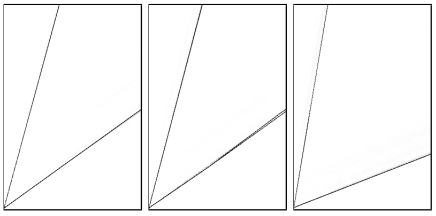
The sample graph on the right shows timings for the `pypy-poly1305` system on a 2137MHz Intel Core 2 Duo (6f6) computer named `katana`. The spreading line shows variance in Pypy's stream-generation time, perhaps from cache-timing effects. Note also the large cost of handling small packets.



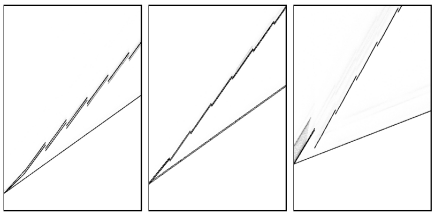
abc-v3-poly1305



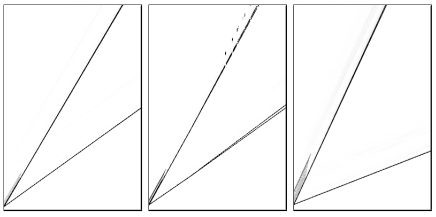
aes-128-poly1305



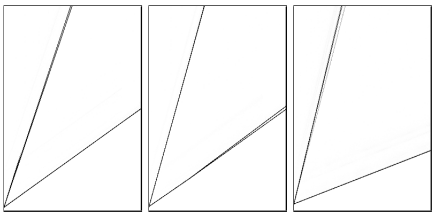
aes-256-poly1305



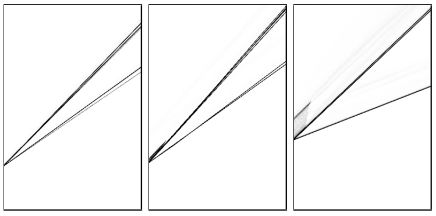
cryptmt-v3-poly1305



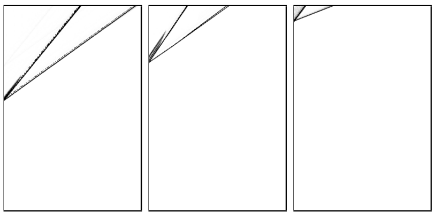
dicing-p2-poly1305



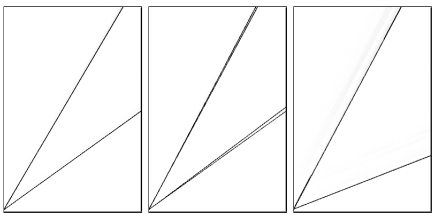
dragon-poly1305



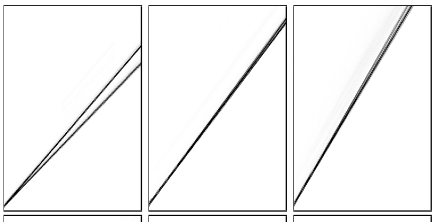
hc-128-poly1305



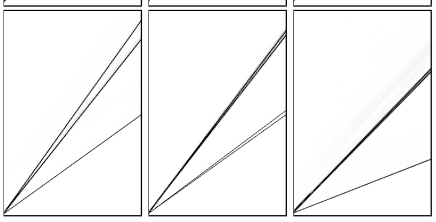
hc-256-poly1305



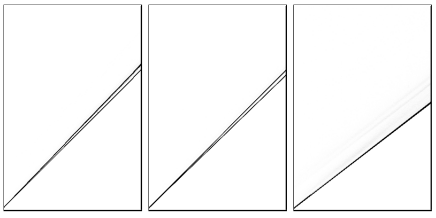
lex-v1-poly1305



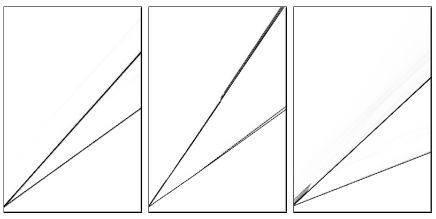
nls-ae



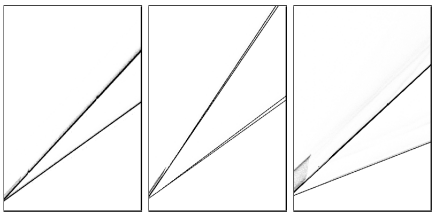
nls-poly1305



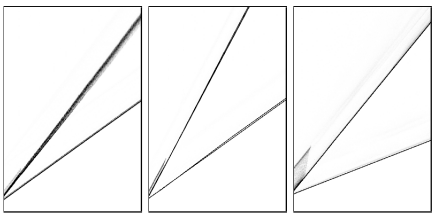
phelix-poly1305



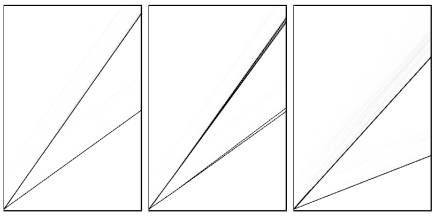
py6-poly1305



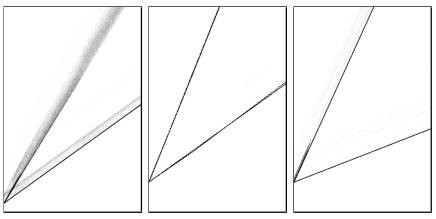
py-poly1305



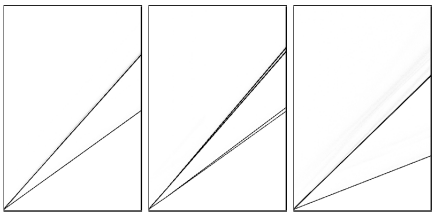
pypy-poly1305



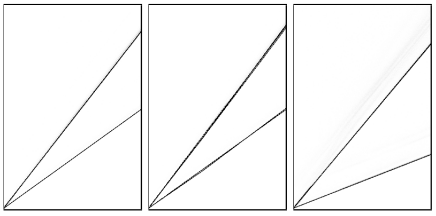
rabbit-poly1305



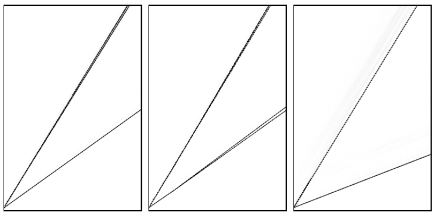
rc4-poly1305



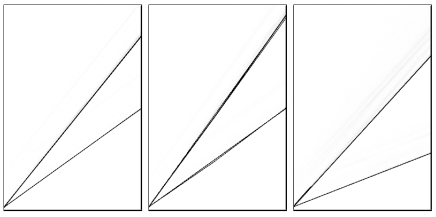
salsa20-8-poly1305



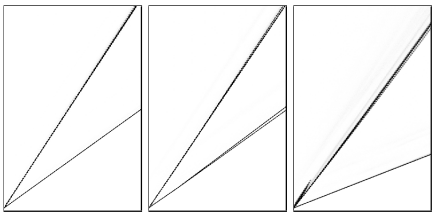
salsa20-12-poly1305



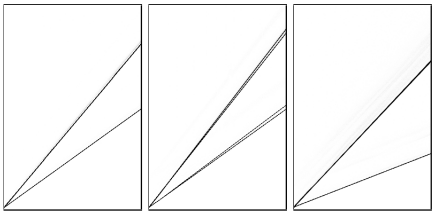
salsa20-poly1305



snow-2.0-poly1305



sosemanuk-poly1305



trivium-poly1305

Each line of graphs on the last three pages has

- a graph for a 2137MHz Intel Core 2 Duo (6f6) named `katana`;
- a graph for a 2000MHz AMD Athlon 64 X2 (15,75,2) named `mace`; and
- a graph for a 900MHz AMD Athlon (622) named `thoth`.

I omitted graphs for three very slow systems (grain-128, grain-v1, mickey-128-2). I also omitted timings for one system (polarbear-2) because I couldn't make the code work.

I'm planning to include graphs for an UltraSPARC and a Pentium M in this paper, and of course many more CPUs online.

Tables

Sorry, this is still a draft! Plans: one table of authenticated-encryption cycle counts for various CPUs, one table of authenticated-encryption cycle counts for various numbers of active keys, one table of decryption cycle counts for various message lengths, and one table of rejection cycle counts for various message lengths.