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

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

* Date of this document: 2007.01.11. 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 (and in particular its conjectured security level), 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, although I marked them as “low” in the “security conjecture” column on the first page. 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.

Non-authenticating stream ciphers

Most of the selected stream ciphers do not include message authentication. I converted the non-authenticating stream ciphers into authenticated-encryption systems by combining them 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 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). I'm planning future work to integrate benchmarking projects. The new speed reports produced by `ciphercycles`, like the eBATS speed reports, are in a simple format designed for easy computer processing.

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.

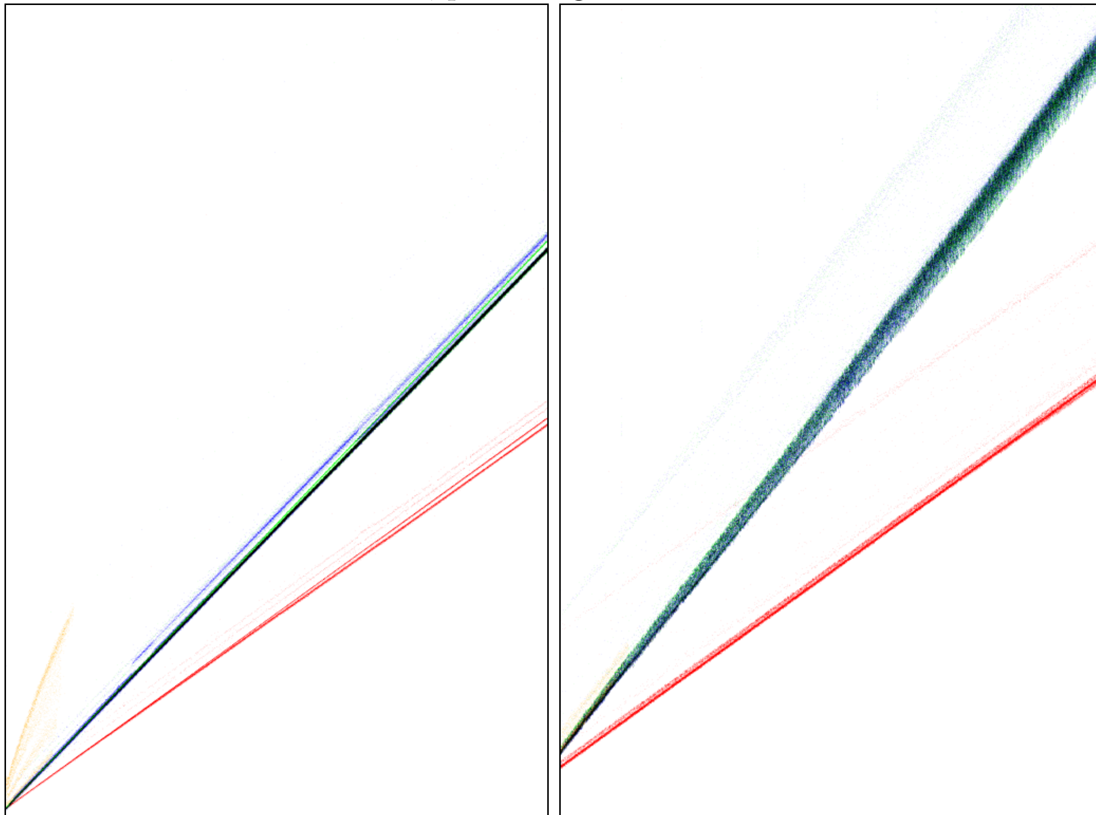
Sample graphs

The graph on the left below shows timings for the abc-v3-poly1305 system on a 2137MHz Intel Core 2 Duo computer named katana.

The horizontal axis is message 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.

Blue is encryption, roughly 8 cycles per byte. Green is decryption, essentially overlapping blue. Red is rejection, roughly 5 cycles per byte. Orange, a much higher slope faintly visible in the bottom left of the graph, is encryption bouncing between multiple keys: message lengths between 0 bytes and 1024 bytes for 1024 keys, message lengths between 0 bytes and 960 bytes for 512 keys, message lengths between 0 bytes and 896 bytes for 256 keys, etc.

There are 15 timings for each message length. Initial timings are slightly slower because of cache misses, producing faint lines above the main lines.



The graph on the right shows timings for the pypy-poly1305 system on the same computer. The wide spread of the dark line shows a considerable variance in Pypy's stream-generation time, perhaps because of cache-timing effects. Note also the relatively large cost of handling small packets.

More graphs

Sorry, this is still a draft! But I do have timings for all ciphers on a few machines, and I need only a little more scripting to quickly produce a bunch of graphs. Graphs will be small here, large online.

Tables

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