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
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: 2007.01.13. Permanent ID of this document: be6b4df07eb1ae67aba9338991b78388.

Abstract. How much time is needed to encrypt, authenticate, verify, and decrypt a packet? The answer depends on the machine (most importantly, but not solely, the CPU), on the choice of authenticated-encryption function, on the packet 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 "software focus" ciphers (Dragon, HC, LEX, Phelix, Py, Salsa20, SOSEMANUK) in phase 2 of eSTREAM, the ECRYPT Stream Cipher Project; all of the "hardware focus" ciphers (Grain, MICKEY, Phelix, Trivium); the remaining "software" ciphers, except for Polar Bear, which I couldn't make work; and the "benchmark" ciphers (AES, RC4, SNOW 2.0) for comparison.

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.

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, and $\oplus$ xors its inputs after truncating the longer input to the same length as the shorter input.

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 packets 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 new 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 packet-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.

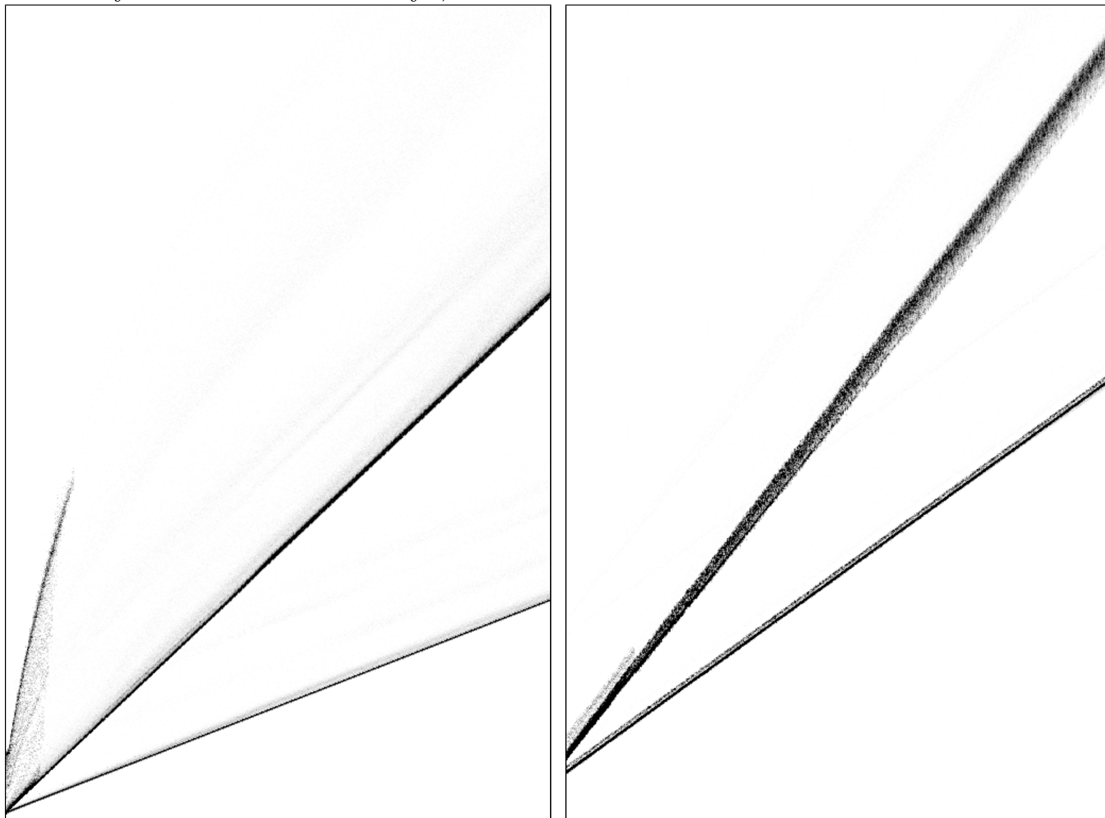
The timings collected by `ciphercycles` also include `makekey` timings, but those timings are not reported in this paper.

Graphs

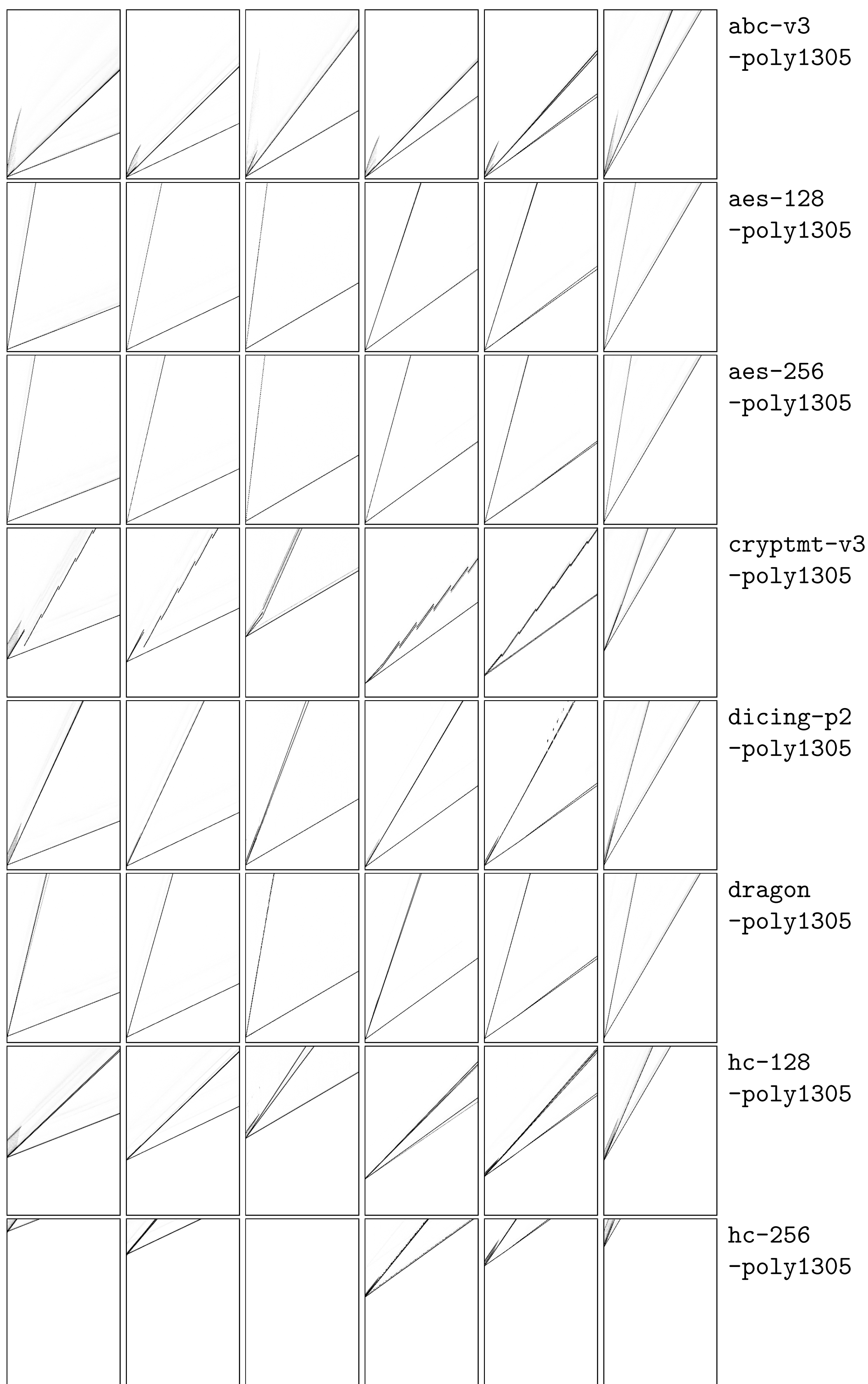
The sample graph on the left below shows timings for the `abc-v3-poly1305` system on a 900MHz AMD Athlon (622) computer named `thoth`.

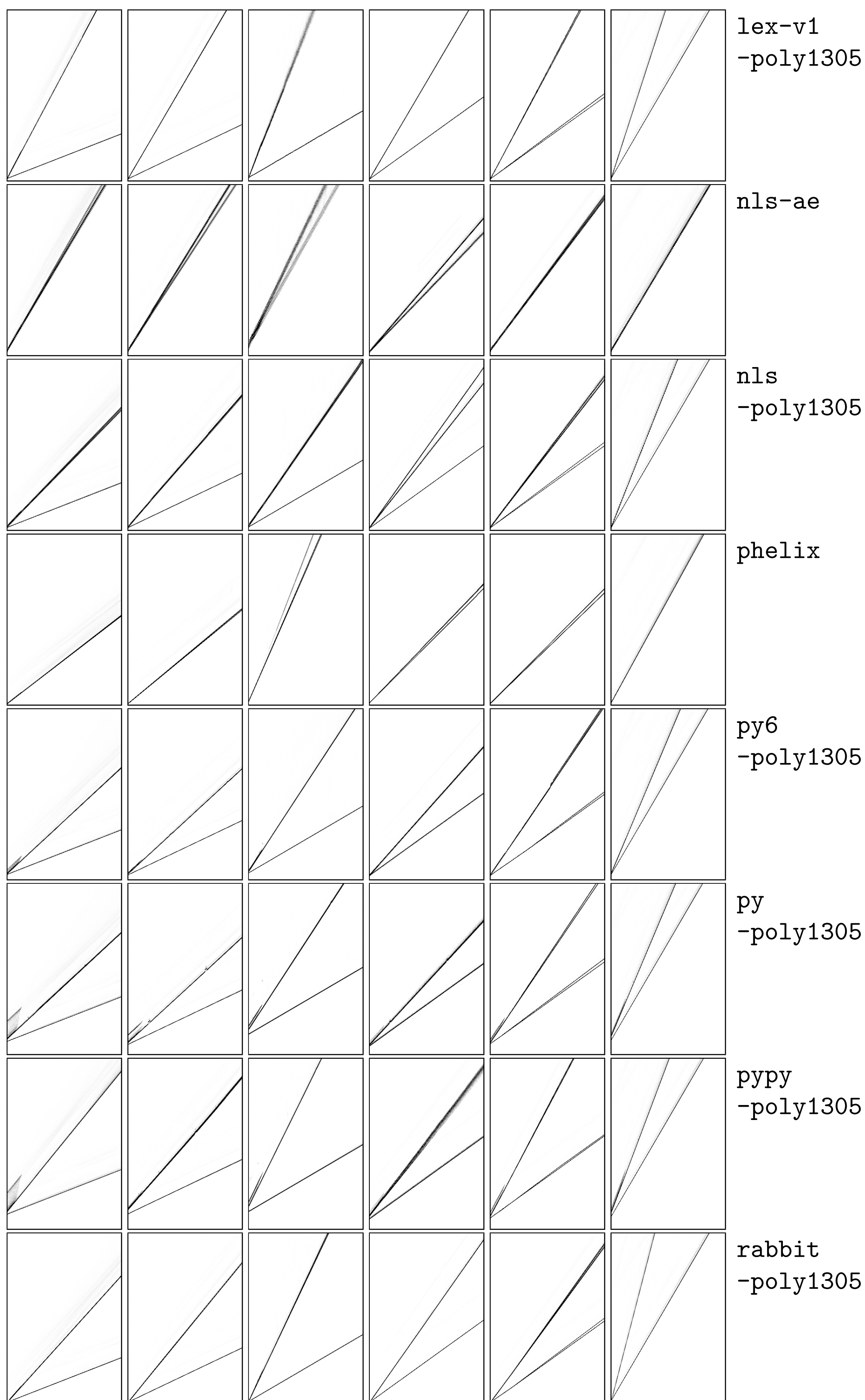
The horizontal axis is packet length, between 0 bytes and 8192 bytes. The vertical axis is time, between 0 cycles and 98304 cycles. The diagonal from the lower left corner of the graph to the upper right corner is 12 cycles per byte.

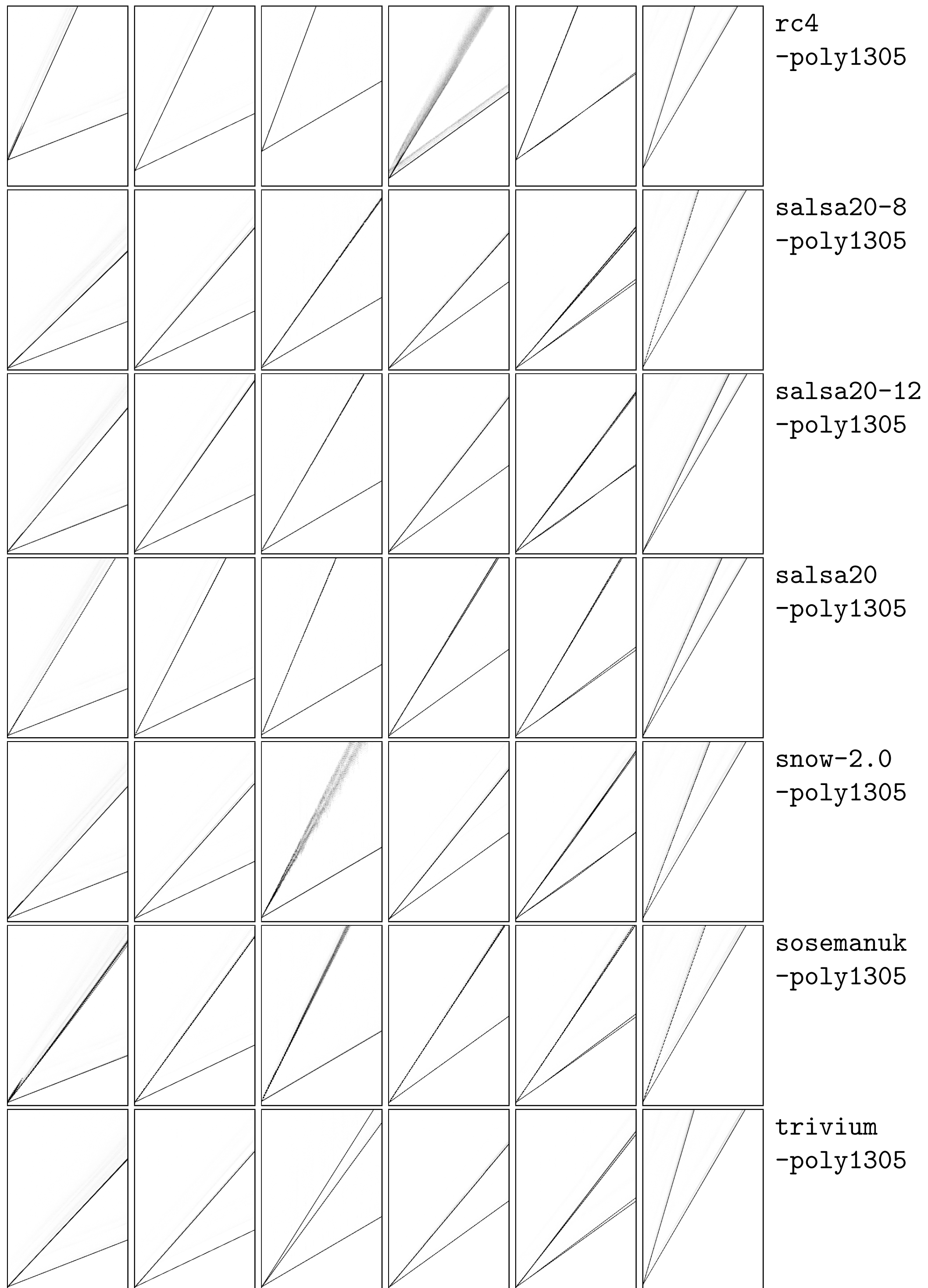
The two main lines visible on the graph are (1) roughly 7 cycles per byte for encryption and decryption and (2) roughly 3 cycles per byte for rejection. Faint lines are visible above the main lines; there are 15 timings for each packet length, and initial timings are slightly slower because of cache misses. There is also a short curve up the left side of the graph for encrypting packets of $\leq$ 1024 bytes using a random key from a pool of 1024 active keys. Also plotted, and faintly visible, are packet lengths of $\leq$ 960 bytes for 512 active keys, packet lengths of $\leq$ 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
py6
-poly1305
py
-poly1305
pypy
-poly1305
rabbit
-poly1305



Graphs for `grain-128-poly1305`, `grain-v1-poly1305`, `mickey-128-2-poly1305` are omitted. All three systems are very slow.

Here are the machines used (in order) for the above graphs:

- a 900MHz AMD Athlon (622) named `thoth`;
- an 800MHz Pentium M (6d8) named `atlas`;
- a 900MHz Sun UltraSPARC III named `wessel`;
- a 2137MHz Intel Core 2 Duo (6f6) named `katana`;
- a 2000MHz AMD Athlon 64 X2 (15,75,2) named `mace`; and
- a 533MHz Motorola PowerPC G4 7410 named `gggg`.

Tables

The following table shows median cycle counts for authenticated encryption as a function of cipher and packet length. All timings are from a 900MHz AMD Athlon (622) named `thoth`. All timings are for a single active key.

0	40	402	576	1500	8192	
946	1635	4341	5525	12654	64314	`abc-v3-poly1305`
957	3330	20623	28023	71473	393518	`aes-128-poly1305`
987	3442	21013	28627	73239	402759	`aes-256-poly1305`
21983	22678	27482	29679	33653	121893	`cryptmt-v3-poly1305`
2454	3446	9865	12581	28885	145413	`dicing-p2-poly1305`
3229	4747	17031	22634	53741	277644	`dragon-poly1305`
2280	4263	20739	28413	70373	385986	`grain-128-poly1305`
2399	5149	27917	38740	96765	526804	`grain-v1-poly1305`
33936	34635	37388	38666	45759	96712	`hc-128-poly1305`
90598	91359	95181	97045	106890	179098	`hc-256-poly1305`
1195	2269	7795	10270	24266	124971	`lex-v1-poly1305`
2300	3740	9410	11043	23220	118608	`nls-ae`
1374	2539	6001	6811	14226	70396	`nls-poly1305`
743	1025	3326	4356	10109	51716	`phelix`
2866	3749	6518	7724	14674	64486	`py6-poly1305`
7274	9181	11946	13187	20124	69881	`py-poly1305`
8561	10658	14231	15825	25065	91266	`pypy-poly1305`
697	1310	4552	5905	14302	73521	`rabbit-poly1305`
14085	14999	21367	24316	40761	158469	`rc4-poly1305`
662	1216	4227	5511	12716	64672	`salsa20-8-poly1305`
781	1342	5071	6708	15547	79788	`salsa20-12-poly1305`
1007	1563	6693	9016	21078	109477	`salsa20-poly1305`
1512	2197	5416	6887	15032	74002	`snow-2.0-poly1305`
1405	2037	5992	7554	16703	87773	`sosemanuk-poly1305`
1217	1755	4919	6258	14328	71471	`trivium-poly1305`

The packet lengths I selected are 40 bytes, 576 bytes, and 1500 bytes from the official eSTREAM timings; 0 bytes; 8192 bytes; and 402 bytes, an approximation to the average Internet packet length.

The following table shows median cycle counts for authenticated encryption as a function of cipher and the number of active keys. All timings are from a 900MHz AMD Athlon (622) named `thoth`. All timings are for 576-byte packets.

1	4	16	64	256	1024		bytes
5652	6437	20967	26962	29155	28836	`abc-v3-poly1305`	33872
28066	28040	28126	28129	28409	28559	`aes-128-poly1305`	276
28783	28636	28662	28788	29031	29426	`aes-256-poly1305`	276
29703	29703	31326	35937	37842	38501	`cryptmt-v3-poly1305`	4392
12538	12538	14951	14723	18080	18992	`dicing-p2-poly1305`	4412
22619	22582	21335	22441	22413	22920	`dragon-poly1305`	300
28563	28519	28629	29458	31558	32564	`grain-128-poly1305`	8328
38722	38766	38693	39587	41223	42613	`grain-v1-poly1305`	4184
38619	38681	40787	40823	46322	48922	`hc-128-poly1305`	4316
97370	97290	103092	108715	115247	119621	`hc-256-poly1305`	8412
10338	10326	10316	10298	10569	10926	`lex-v1-poly1305`	248
11030	11086	11033	11149	11139	11429	`nls-ae`	232
6859	6855	6857	6944	6919	7120	`nls-poly1305`	244
4357	4357	4412	4386	4514	4415	`phelix`	132
7731	7727	7787	8095	8372	10195	`py6-poly1305`	1140
13371	13380	14560	17433	20995	23702	`py-poly1305`	4212
15804	17163	16907	18252	25976	26964	`pypy-poly1305`	4260
5906	5921	5921	5985	6071	6217	`rabbit-poly1305`	152
24322	24280	24369	24792	26147	26456	`rc4-poly1305`	1084
5521	5530	5530	5583	5599	5622	`salsa20-8-poly1305`	80
6729	6708	6709	6708	6942	6973	`salsa20-12-poly1305`	80
9026	9019	9020	9066	9303	9311	`salsa20-poly1305`	80
6871	6884	7008	7104	7004	7355	`snow-2.0-poly1305`	124
7602	8230	8765	8791	9392	9706	`sosemanuk-poly1305`	468
6237	6242	6276	6273	6359	6466	`trivium-poly1305`	80

The "bytes" column in the above table indicates the number of bytes in an expanded key. The penalty for handling 1024 active keys, compared to just 1, is usually around 2 cycles for each expanded-key byte, presumably reflecting this machine's cache-load bandwidth. Grain shows a smaller penalty compared to its expanded-key size; presumably Grain does not access the entire expanded key for a 576-byte packet.

The following table shows median cycle counts for verified decryption as a function of cipher and machine. All timings are for 576-byte packets. All timings are for a single active key.

thoth	atlas	wessel	katana	mace	gggg	
5493	5453	7275	5656	6058	11920	abc-v3-poly1305
28114	22903	37371	14560	15388	25520	aes-128-poly1305
28594	20964	42039	17840	18526	29424	aes-256-poly1305
30090	29323	41915	13032	18157	39408	cryptmt-v3-poly1305
12546	11854	14582	9384	10745	19472	dicing-p2-poly1305
22372	19234	29724	15768	18838	25776	dragon-poly1305
28497	22713				48608	grain-128-poly1305
38660	35248				50912	grain-v1-poly1305
38302	36708	52004	26000	28001	43216	hc-128-poly1305
97446	83339	119769	58760	78089	94480	hc-256-poly1305
10348	9400	14095	9088	9929	16192	lex-v1-poly1305
10866	10683	15724	7688	9346	10960	nls-ae
6750	7453	8989	7936	7828	13584	nls-poly1305
4371	4773	12908	6152	5718	10032	phelix
7742	7688	11775	7920	9475	14800	py6-poly1305
13262	11051	21369	10608	13563	21472	py-poly1305
15882	16548	22560	13792	16305	22080	pypy-poly1305
6045	6623	11051	7456	7081	19696	rabbit-poly1305
24418	18112	31312	11912	25761	24384	rc4-poly1305
5570	6543	8001	6224	6304	16496	salsa20-8-poly1305
6734	7867	9708	7064	7236	10880	salsa20-12-poly1305
9045	10633	12984	8904	9019	11808	salsa20-poly1305
6892	7185	10773	7160	8098	14416	snow-2.0-poly1305
7379	8161	11537	8632	8357	15216	sosemanuk-poly1305
6333	6536	8531	6672	7034	18656	trivium-poly1305

The following table shows median cycle counts for rejection of a forged packet as a function of cipher and machine. All timings are for 576-byte packets. All timings are for a single active key.

thoth	atlas	wessel	katana	mace	gggg	
2854	3206	3873	4184	4212	9168	abc-v3-poly1305
2858	3109	4107	4024	4010	9088	aes-128-poly1305
2919	3068	4241	4104	4052	9216	aes-256-poly1305
23882	22687	37791	11232	15721	34800	cryptmt-v3-poly1305
4377	4223	5340	4848	5573	10640	dicing-p2-poly1305
5137	5017	5564	5008	5258	10336	dragon-poly1305
4196	4191				11712	grain-128-poly1305
4313	4463				11200	grain-v1-poly1305
35466	34312	47536	24472	26002	39984	hc-128-poly1305
92778	79852	113603	56240	74321	90176	hc-256-poly1305
3156	3353	4498	4352	4465	9488	lex-v1-poly1305
10876	10648	15650	7712	9347	10944	nls-ae
3287	3605	4444	4448	4766	9440	nls-poly1305
4368	4774	12914	6144	5717	10032	phelix
4778	4935	6539	5824	5694	11040	py6-poly1305
9194	7858	14172	8256	9295	16208	py-poly1305
10466	11023	12858	9256	9873	15424	pypy-poly1305
2650	3112	3987	4160	4024	9792	rabbit-poly1305
15984	10584	21629	11096	17582	17808	rc4-poly1305
2567	3057	3749	4048	4018	9536	salsa20-8-poly1305
2683	3205	3901	4168	4216	8880	salsa20-12-poly1305
2917	3488	4225	4312	4288	8944	salsa20-poly1305
3374	3781	4667	4552	4792	9888	snow-2.0-poly1305
3242	3785	4769	4664	4695	9824	sosemanuk-poly1305
3128	3447	4019	4376	4418	10688	trivium-poly1305

Appendix: Tunings

A cipher in the official eSTREAM benchmarking toolkit can have several tunings: several implementations in separate subdirectories of the cipher directory, and several "variants" of each implementation.

The new toolkit automatically tries encrypting several 1536-byte packets under each tuning. It then selects the tuning producing the smallest median cycle count, and uses that tuning for subsequent timings. The following table lists the selected tunings.

thoth	atlas	wessel	katana	mace	gggg	
1	1	1	1	1	1	abc-v3-poly1305
ref	ref	ref	ref	ref	ref	aes-128-poly1305
ref	ref	ref	ref	ref	ref	aes-256-poly1305
1	1	1	1	1	1	cryptmt-v3-poly1305
1	1	1	1	1	1	dicing-p2-poly1305
1	1	1	1	1	1	dragon-poly1305
opt	opt				opt	grain-128-poly1305
opt	opt				opt	grain-v1-poly1305
1	1	1	1	1	1	hc-128-poly1305
1	1	1	1	1	1	hc-256-poly1305
1	1	1	1	1	1	lex-v1-poly1305
2	2	2	2	2	2	nls-ae
2	2	2	2	2	2	nls-poly1305
i386	i386	ref	ref	ref	ref	phelix
1	1	1	1	1	1	py6-poly1305
1	1	1	1	1	1	py-poly1305
1	1	1	1	1	1	pypy-poly1305
opt/3	opt/4	opt/3	opt/3	opt/4	opt/3	rabbit-poly1305
2	2	1	2	2	1	rc4-poly1305
x86-athlon	x86-3	sparc	amd64-3	amd64-3	ref	salsa20-8-poly1305
x86-athlon	x86-pm	sparc	amd64-3	amd64-3	ppc-altivec	salsa20-12-poly1305
x86-athlon	x86-pm	sparc	amd64-3	amd64-3	ppc-altivec	salsa20-poly1305
1	1	1	1	1	1	snow-2.0-poly1305
1	1	1	1	1	1	sosemanuk-poly1305
1	1	1	1	1	1	trivium-poly1305

I'm surprised by some of the selections of "ref" in this table; I'm investigating.