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-v2-poly1305	256	CryptMT v2	Poly1305	384
dicing-v2-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	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: 2007.01.18. Permanent ID of this document: be6b4df07eb1ae67aba9338991b78388.

Abstract. Exactly 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 is 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 candidates.

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: `expandkey` to take a key and convert it into 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.

For the timings reported in this paper I wrote a new toolkit, `ciphercycles`, available from `http://cr.yp.to/streamciphers/timings.html`. I also wrote a 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 `ciphercycles`. Updates to the implementations in the official eSTREAM benchmarking toolkit will be easily reflected 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. 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.

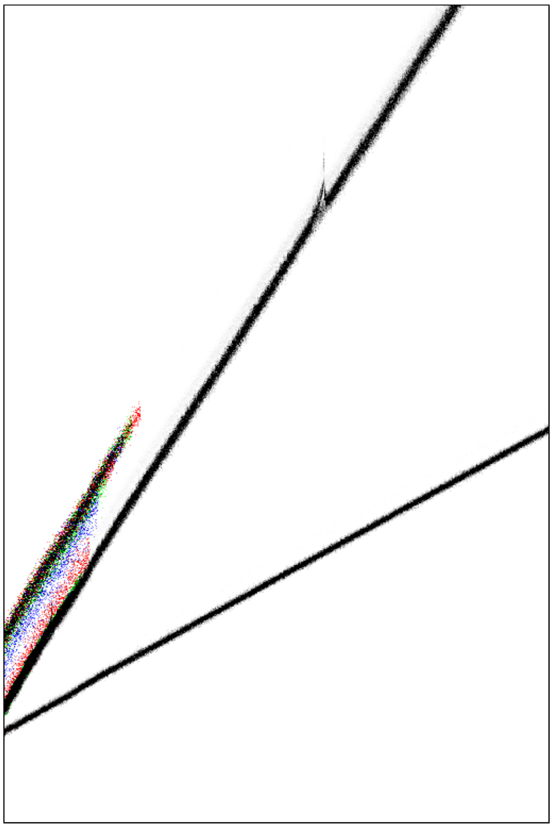
The timings collected by `ciphercycles` also include `expandkey` timings, but those timings are not reported in this paper.

Graphs

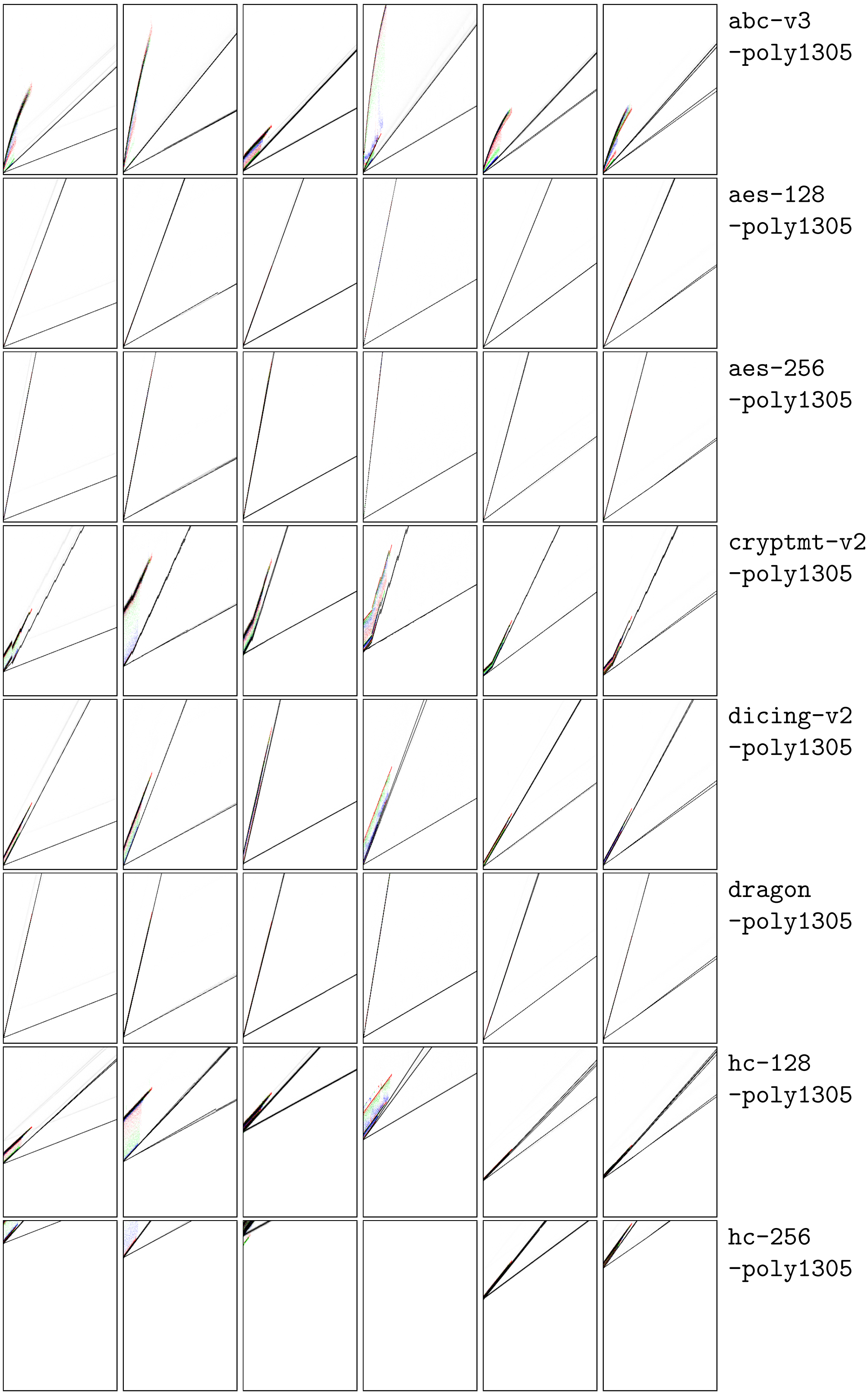
The sample graph on the left below shows timings for the `abc-v3-poly1305` system on a 2137MHz Intel Core 2 Duo (6f6) computer named `katana`.

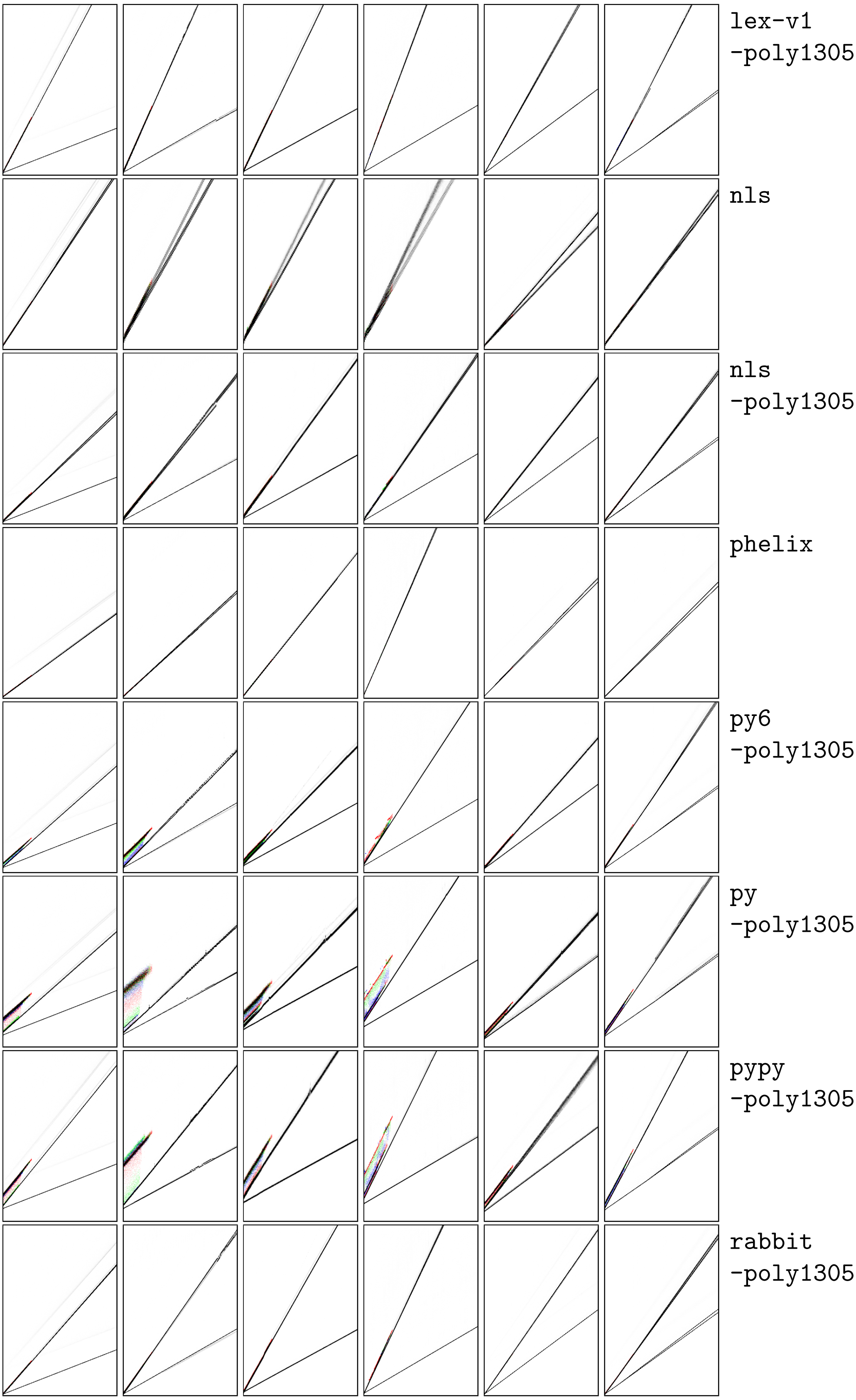
The horizontal axis is packet length, between 0 bytes and 8192 bytes. The vertical axis is time, between 0 cycles and 98304 cycles. The diagonal from the lower left corner of the graph to the upper right corner is 12 cycles per byte.

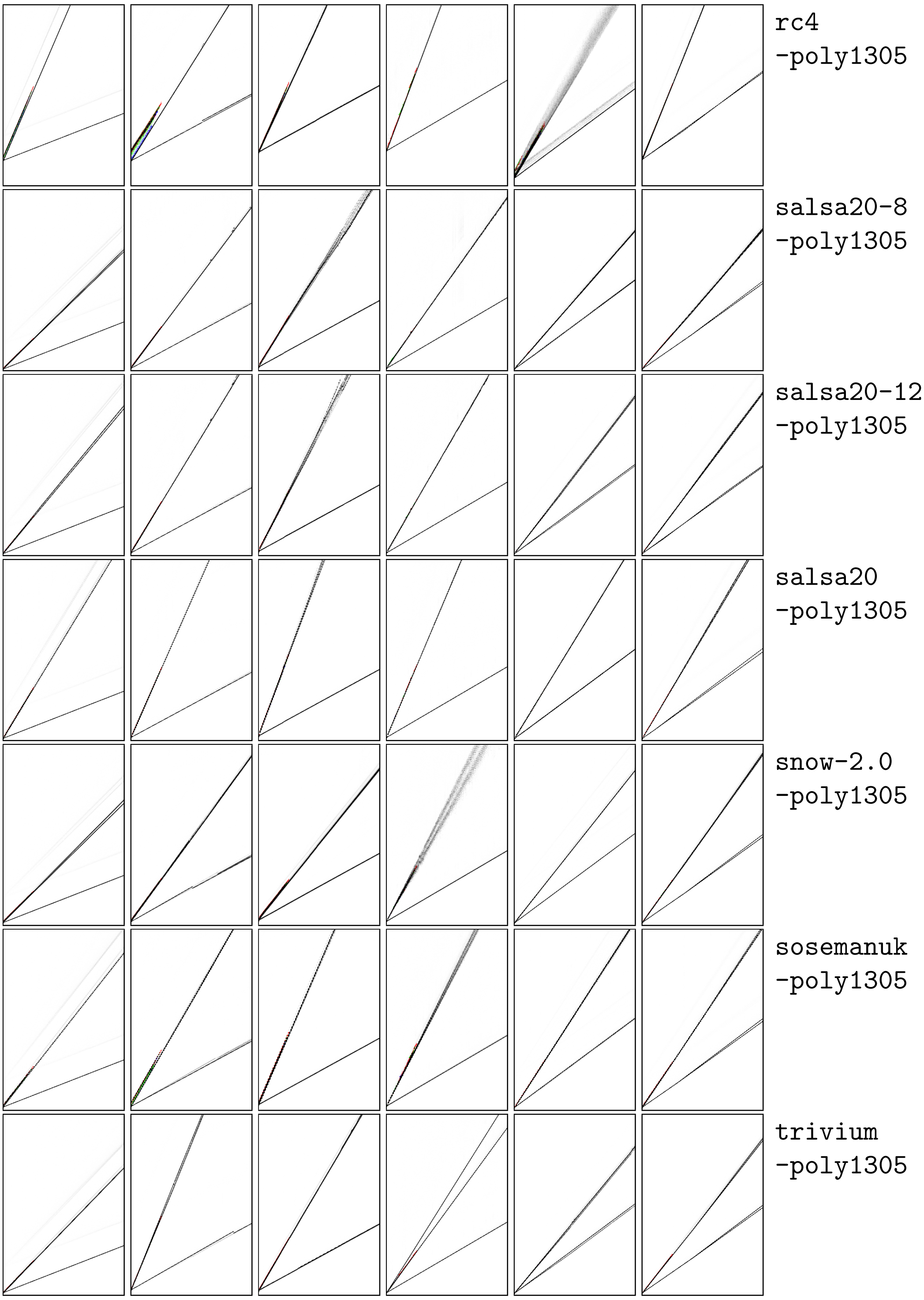
The two main lines visible on the graph are (1) roughly 8 cycles per byte for encryption and decryption and (2) roughly 6 cycles per byte for rejection. Faint lines are visible above the main lines; there are 15 timings for each packet length, and initial timings are slightly slower because of cache misses. There is also a short curve up the left side of the graph for encrypting packets of ≤ 2048 bytes using a random key from a pool of 8192 active keys. Also plotted (in various colors) are packet lengths of ≤ 1920 bytes for 4096 active keys, packet lengths of ≤ 1792 bytes for 2048 active keys, etc.



The sample graph on the right shows timings for the `pypy-poly1305` system on a 3400MHz Intel Pentium 4 (f29) named `shell`. 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-v2
-poly1305
dicing-v2
-poly1305
dragon
-poly1305
hc-128
-poly1305
hc-256
-poly1305


lex-v1
-poly1305
nls
nls
-poly1305
phelix
py6
-poly1305
py
-poly1305
pypy
-poly1305
rabbit
-poly1305



`grain-128-poly1305`, `grain-v1-poly1305`, `mickey-128-2-poly1305`: slow; graphs omitted.

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

- a 1343MHz AMD Athlon XP (662) x86 named `lpc36`;
- a 1000MHz Intel Pentium III (68a) x86 named `neumann`;
- a 3400MHz Intel Pentium 4 (f29) x86 named `shell`;
- a 900MHz Sun UltraSPARC III sparcv9 named `wessel`;
- a 2137MHz Intel Core 2 Duo (6f6) amd64 named `katana`; and
- a 2000MHz AMD Athlon 64 X2 (15,75,2) amd64 named `mace`.

Tables

The following table shows median cycle counts for authenticated encryption as a function of cipher and packet length. All timings are from a 3400MHz Intel Pentium 4 (f29) named `shell`. All timings are for a single active key.

0	40	402	576	1500	8192	
1508	2500	5596	6916	14972	71448	abc-v3-poly1305
988	2480	10592	14064	34804	183656	aes-128-poly1305
1312	3744	20040	26828	67936	361120	aes-256-poly1305
23584	24580	29784	32172	54860	223664	cryptmt-v2-poly1305
2780	4780	17892	23364	56828	295068	dicing-v2-poly1305
3084	4868	16424	21628	51784	263348	dragon-poly1305
2576	4756	20380	27672	69676	373356	grain-128-poly1305
2984	6424	31588	43500	107824	574584	grain-v1-poly1305
49816	50596	53832	55328	63600	121700	hc-128-poly1305
90872	91432	95624	96868	106804	172288	hc-256-poly1305
1648	3172	9124	11892	27712	139332	lex-v1-poly1305
3716	5768	12804	14152	27652	135676	nls
2640	4444	8916	10008	19864	94556	nls-poly1305
1292	1736	5392	7084	16364	83640	phelix
3556	5232	8364	9464	17564	72340	py6-poly1305
9576	11880	14644	15816	25012	79164	py-poly1305
10656	14256	18752	21120	33520	114860	pypy-poly1305
1616	2652	7672	9888	23884	118276	rabbit-poly1305
17820	19048	25324	28336	44520	160832	rc4-poly1305
1496	2276	7696	9828	22084	105644	salsa20-8-poly1305
1696	2624	9080	11896	26128	133060	salsa20-12-poly1305
2080	2852	11772	15556	35784	178940	salsa20-poly1305
2276	3372	7196	8844	18076	85112	snow-2.0-poly1305
2540	3720	10796	13736	30260	153888	sosemanuk-poly1305
2096	3136	8104	10256	23272	113908	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 3400MHz Intel Pentium 4 (f29) named `shell`. All timings are for 576-byte packets.

1	32	128	512	2048	8192		bytes
7136	7452	10772	14640	14620	15268	`abc-v3-poly1305`	4176
14024	14012	14112	14056	14116	14460	`aes-128-poly1305`	88
26628	27016	26700	27240	26948	27876	`aes-256-poly1305`	276
32916	33292	35736	42076	43280	41488	`cryptmt-v2-poly1305`	11812
23492	22984	23884	29532	29600	29840	`dicing-v2-poly1305`	4412
21668	21740	21872	21776	22108	22676	`dragon-poly1305`	300
25964	27956	27616	28716	29772	29668	`grain-128-poly1305`	8328
43092	43708	44100	47036	47016	48616	`grain-v1-poly1305`	4184
55196	55716	56172	57536	58096	59432	`hc-128-poly1305`	4316
90372	99280	101928	102392	102240	103376	`hc-256-poly1305`	8412
11960	11876	11928	11932	12296	13080	`lex-v1-poly1305`	248
14676	13880	13900	14308	14320	14716	`nls`	232
9968	9944	10008	9976	10840	10908	`nls-poly1305`	244
7104	7096	7120	7128	7152	7560	`phelix`	132
9740	9864	10344	10448	11340	12024	`py6-poly1305`	1140
15940	18464	18956	22604	23124	23736	`py-poly1305`	4212
21132	21624	24312	28920	30524	30068	`pypy-poly1305`	4260
10080	9928	9916	9948	10064	10732	`rabbit-poly1305`	152
27888	28292	28164	28624	28776	29332	`rc4-poly1305`	1084
9828	9892	9856	9900	9912	10324	`salsa20-8-poly1305`	80
11372	11608	11572	11528	11788	12180	`salsa20-12-poly1305`	80
15292	15420	15384	15396	15244	16008	`salsa20-poly1305`	80
8996	9004	8920	8836	9368	9792	`snow-2.0-poly1305`	124
13640	13524	13656	13556	14748	15100	`sosemanuk-poly1305`	468
10208	10204	10216	10204	10280	10716	`trivium-poly1305`	80

The "bytes" column in the above table indicates the number of bytes in an expanded key. The penalty for handling many active keys, compared to just 1, is usually around 2 cycles for each expanded-key byte, presumably reflecting this machine's cache-load bandwidth. Some systems (e.g., `grain-v1-poly1305`) show a smaller penalty compared to their expanded-key size; presumably these systems do 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.

lpc36	neumann	shell	wessel	katana	mace	
5399	7167	7340	7291	5744	6300	abc-v3-poly1305
13144	13755	14156	25047	12112	11715	aes-128-poly1305
24961	25299	26300	42257	18016	18675	aes-256-poly1305
19187	24740	32324	32768	16584	17469	cryptmt-v2-poly1305
11083	14319	23360	15066	9504	10757	dicing-v2-poly1305
22966	23421	21848	31997	15912	18765	dragon-poly1305
28355	26482	27364				grain-128-poly1305
38591	39402	37596				grain-v1-poly1305
35389	37434	54860	51987	26088	27672	hc-128-poly1305
90863	83860	96356	116925	59584	77855	hc-256-poly1305
10550	12518	12076	14139	9592	9872	lex-v1-poly1305
9204	14110	13000	15814	7664	9045	nls
5628	8431	10136	8961	7584	7660	nls-poly1305
4149	5513	7220	12880	6112	5647	phelix
7399	8059	9360	11767	7824	9429	py6-poly1305
12405	12546	15892	21375	9832	13449	py-poly1305
14938	14906	21116	22620	13128	16321	pypy-poly1305
5996	7626	10148	11040	7552	7081	rabbit-poly1305
24494	20957	28348	31295	11944	25881	rc4-poly1305
5630	7816	9964	8144	6224	6308	salsa20-8-poly1305
6941	9416	11852	9800	7152	7376	salsa20-12-poly1305
9100	12616	15552	13045	8896	9015	salsa20-poly1305
6402	8792	8860	11203	7328	8017	snow-2.0-poly1305
7827	10332	13752	11349	8472	8367	sosemanuk-poly1305
6161	14513	10616	8568	6744	7029	trivium-poly1305

Note the impressive performance of Phelix at verified decryption (and, as shown by the graphs, authenticated encryption). Phelix isn't always the fastest system, and it won't benefit from improvements in MAC speed, but the idea of unifying authentication and encryption in a single primitive is obviously worth further study.

The story for NLS is different. The authenticator built into NLS is slower than Poly1305 and should be scrapped.

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.

lpc36	neumann	shell	wessel	katana	mace	
2787	3821	4396	3871	4096	4400	abc-v3-poly1305
2443	3579	3796	4132	4112	3900	aes-128-poly1305
2801	3663	4104	4263	4184	4061	aes-256-poly1305
15677	19538	26356	28689	14816	15054	cryptmt-v2-poly1305
4053	4689	5680	5345	4992	5616	dicing-v2-poly1305
4770	5940	5776	5863	5104	5274	dragon-poly1305
4174	4770	5496				grain-128-poly1305
4346	5145	5716				grain-v1-poly1305
32588	34550	51256	47451	24456	25680	hc-128-poly1305
87086	79661	92176	111230	56856	74314	hc-256-poly1305
3154	4234	4496	4493	4528	4366	lex-v1-poly1305
9257	14107	13072	15803	7648	9037	nls
2822	4306	5456	4413	4592	4605	nls-poly1305
4145	5499	7236	12880	6112	5647	phelix
4601	5298	6304	6526	5712	5642	py6-poly1305
8495	9570	12264	14183	7552	9238	py-poly1305
9165	9808	13364	12850	9032	9805	pypy-poly1305
2527	3557	4376	3989	4288	4025	rabbit-poly1305
15518	16072	20968	21636	7464	17531	rc4-poly1305
2540	3657	4276	3802	4032	4022	salsa20-8-poly1305
2644	3817	4476	3959	4056	4214	salsa20-12-poly1305
2908	4137	4832	4287	4432	4284	salsa20-poly1305
3223	4562	5120	4747	4688	4680	snow-2.0-poly1305
3378	4508	5264	4771	4744	4696	sosemanuk-poly1305
3104	5499	5012	4007	4464	4411	trivium-poly1305

Phelix has to decrypt forged packets before it can reject them, and it can't decrypt as quickly as a separate MAC, as this table demonstrates.

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.

lpc36	neumann	shell	wessel	katana	mace	
v3/1	v3/1	v3/2	v3/1	v3/1	v3/1	abc-v3-poly1305
x86-mmx-1	x86-mmx-1	x86-mmx-1	big-1	amd64-2	amd64-1	aes-128-poly1305
gladman	gladman	gladman	gladman	gladman	gladman	aes-256-poly1305
v2	v2	v2	v2	v2	v2	cryptmt-v2-poly1305
v2	v2	v2	v2	sse2	v2	dicing-v2-poly1305
dragon	dragon	dragon	dragon	dragon	dragon	dragon-poly1305
opt	opt	opt				grain-128-poly1305
opt	opt	opt				grain-v1-poly1305
200701a	200701a	200701b	200606	200701a	200701a	hc-128-poly1305
200701	200701	200701	200701	200511	200701	hc-256-poly1305
v1	v1	v1	v1	v1	v1	lex-v1-poly1305
sync-ae/2	sync-ae/2	sync-ae/2	sync-ae/2	sync-ae/2	sync-ae/2	nls
sync/2	sync/2	sync/2	sync/2	sync/2	sync/2	nls-poly1305
i386	i386	i386	ref	ref	ref	phelix
py6	py6	py6	py6	py6	py6	py6-poly1305
py	py	py	py	py	py	py-poly1305
pypy	pypy	pypy	pypy	pypy	pypy	pypy-poly1305
opt/4	opt/3	opt/3	opt/3	opt/3	opt/3	rabbit-poly1305
rc4/2	rc4/2	rc4/1	rc4/1	rc4/2	rc4/2	rc4-poly1305
x86-athlon	x86-mmx	x86-athlon	sparc	amd64-3	amd64-3	salsa20-8-poly1305
x86-athlon	x86-mmx	x86-athlon	sparc	amd64-3	amd64-3	salsa20-12-poly1305
x86-athlon	x86-mmx	x86-3	sparc	amd64-3	amd64-3	salsa20-poly1305
snow-2.0	snow-2.0	snow-2.0	snow-2.0	snow-2.0	snow-2.0	snow-2.0-poly1305
sosemanuk	sosemanuk	sosemanuk	sosemanuk	sosemanuk	sosemanuk	sosemanuk-poly1305
trivium	trivium	trivium	trivium	trivium	trivium	trivium-poly1305

The underlying Poly1305 library selected the `athlon` implementation on `lpc36`, `neumann`, and `shell`; the `sparc` implementation on `wessel`; and the `53` implementation on `katana` and `mace`.