Software benchmarking of SHA-3 candidates

http://bench.cr.yp.to

D. J. Bernstein
University of Illinois at Chicago

Tanja Lange
Technische Universiteit Eindhoven

Selecting cryptographic primitives

NIST's final AES report, 2001:

"Security was the most important factor in the evaluation . . . Rijndael appears to offer an *adequate* security margin. . . . Serpent appears to offer a *high* security margin."

(Emphasis added.)

So why didn't Serpent win?

Selecting cryptographic primitives

NIST's final AES report, 2001:

"Security was the most important factor in the evaluation . . . Rijndael appears to offer an *adequate* security margin. . . . Serpent appears to offer a *high* security margin."

(Emphasis added.)

So why didn't Serpent win?

Maybe hardware efficiency?
Or side-channel security?
Or something else?

Side channels: “The operations used by Serpent are among the easiest to defend against timing and power attacks.”

Side channels: “The operations used by Serpent are among the easiest to defend against timing and power attacks.”

Hardware speed: “Serpent is well suited to restricted-space environments . . . Fully pipelined implementations of Serpent offer the highest throughput of any of the finalists for non-feedback modes. . . . Efficiency is generally very good, and Serpent’s speed is independent of key size.”

Side channels: "The operations used by Serpent are among the easiest to defend against timing and power attacks."

Hardware speed: "Serpent is well suited to restricted-space environments . . . Fully pipelined implementations of Serpent offer the highest throughput of any of the finalists for non-feedback modes. . . . Efficiency is generally very good, and Serpent's speed is independent of key size."

Great! Why didn't Serpent win?

Aha: Software speed!

Aha: Software speed! “Serpent is generally the slowest of the finalists in software speed for encryption and decryption. . . . Serpent provides consistently low-end performance.”

Aha: Software speed! “Serpent is generally the slowest of the finalists in software speed for encryption and decryption. . . . Serpent provides consistently low-end performance.”

Conclusion: “NIST judged Rijndael to be the best overall algorithm for the AES. Rijndael appears to be consistently a very good performer in both hardware and software [and offers good key agility, low memory, easy defense, fast defense, flexibility, parallelism].”

2007 NIST SHA-3 call: “The security provided by an algorithm is the most important factor in the evaluation.”

2007 NIST SHA-3 call: "The security provided by an algorithm is the most important factor in the evaluation."

2011.02 NIST report:
"BLAKE . . . high security margin . . .
NIST feels that future results are less likely to dramatically narrow Grøstl's security margin than that of the other candidates. . . .
JH . . . solid security margin . . .
Keccak . . . high security margin . . .
Skein . . . high security margin"

Will this factor alone
decide the winner?

Will this factor alone
decide the winner?
Will further security analysis
kill 4 out of 5 SHA-3 candidates?

Will this factor alone
decide the winner?
Will further security analysis
kill 4 out of 5 SHA-3 candidates?
Perhaps, but probably not!

Presumably decision will depend
partially on speed in software,
speed in hardware, speed of
implementations with various
side-channel defenses, etc.

Will this factor alone
decide the winner?
Will further security analysis
kill 4 out of 5 SHA-3 candidates?
Perhaps, but probably not!

Presumably decision will depend
partially on speed in software,
speed in hardware, speed of
implementations with various
side-channel defenses, etc.

Remaining speed differences
seem larger than
remaining security differences.

Speed variability

Main question in this talk:
"How fast is hash software?"

Answer varies from
one hash function to another.

Perhaps this variability
is important to hash users.

Perhaps this variability will be
important in the SHA-3 selection.

Answer depends on hash-function parameters.

On a 3200MHz AMD Phenom II X6 1090T (100fa0), for the same input size, changing from 256-bit output to 512-bit output makes
BLAKE $\approx 1.55\times$ faster;
SHA-2 $\approx 1.31\times$ faster;
Skein $\approx 1.01\times$ faster;
JH neither faster nor slower;
Grøstl $\approx 1.48\times$ slower;
Keccak $\approx 1.86\times$ slower.

(2010.12 data, before tweaks.)

Answer depends on
#cores used for hashing.

2.4GHz Intel Core 2 Duo E4600 (6fd) has 2 CPU cores operating in parallel.

2.4GHz Intel Core 2 Quad Q6600 (6fb) has 4 CPU cores operating in parallel.
Hash twice as many messages per second!

Standard way to *reduce* this dependence: measure hash time on 1 core.

Warning: Single-core speed
is sometimes better than
speed of 4 cores
handling 4 messages in parallel.
Multiple active cores
can conflict in DRAM access etc.

Warning: Single-core speed$\times$4
is usually better than
speed of 4 cores cooperating
to handle 1 long message.

Warning: These issues
(and more issues coming up)
have different effects
on different hash functions.

Back to the main question:
How fast is hash software?

Answer depends on CPU.

In one second, single-core 533MHz PowerPC G4 (7410) computes SHA-256 hashes of 5985 4096-byte messages.

In one second, single core of 1800MHz PowerPC G5 (970) computes SHA-256 hashes of 20729 4096-byte messages.

Standard way to *reduce* this dependence: count cycles; i.e., divide #seconds by clock speed.

533MHz PowerPC G4 (7410): 86835 cycles to hash a 4096-byte message with SHA-256.

1800MHz PowerPC G5 (970): 89047 cycles to hash a 4096-byte message with SHA-256.

Note: Most CPUs have built-in cycle counters; “RDTSC” etc. Cycles are also a natural unit for serious programmers.

Warning: Different CPUs do different amounts of computation in a cycle.

Warning: Different CPUs with different speeds can have the same name.

Warning: Some CPU operations (e.g. DRAM access) do not scale linearly with clock speed.

Warning: A CPU in 64-bit mode is often faster (but sometimes slower!) than the same CPU in 32-bit mode.

4096-byte SHA-256 timings:

64421 cycles: amd64 architecture (64-bit), 2833MHz Intel Core 2 Quad Q9550 (10677).

64923 cycles: x86 architecture (32-bit), 2833MHz Intel Core 2 Quad Q9550 (10677).

88304 cycles: ppc32, 533MHz Motorola PowerPC G4 (7410).

94464 cycles: armeabi, 800MHz Freescale i.MX515 (Cortex A8).

197572 cycles: armeabi, 400MHz TI OMAP 2420.

4096-byte SHA-512 timings:

44200 cycles: amd64 architecture (64-bit), 2833MHz Intel Core 2 Quad Q9550 (10677).

77682 cycles: x86 architecture (32-bit), 2833MHz Intel Core 2 Quad Q9550 (10677).

228864 cycles: ppc32, 533MHz Motorola PowerPC G4 (7410).

390400 cycles: armeabi, 800MHz Freescale i.MX515 (Cortex A8).

500038 cycles: armeabi, 400MHz TI OMAP 2420.

How fast is hash software?

Answer depends on message length: hashing long message takes more time than hashing short message.

SHA-512 timings on 3200MHz AMD Phenom II X4 955 (100f42):

48166 cycles for 4096 bytes.

24917 cycles for 2048 bytes.

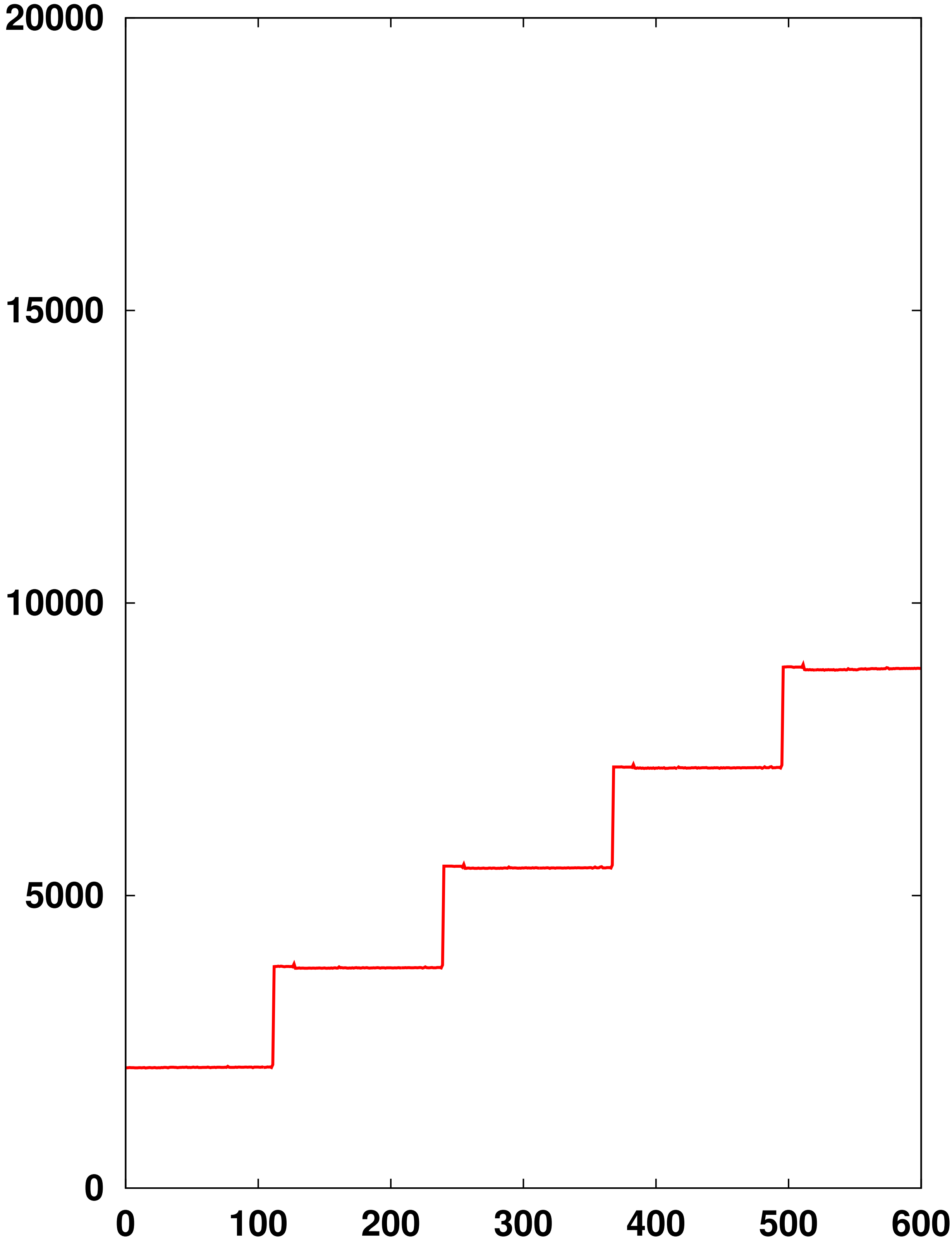
15584 cycles for 1024 bytes.

13304 cycles for 512 bytes.

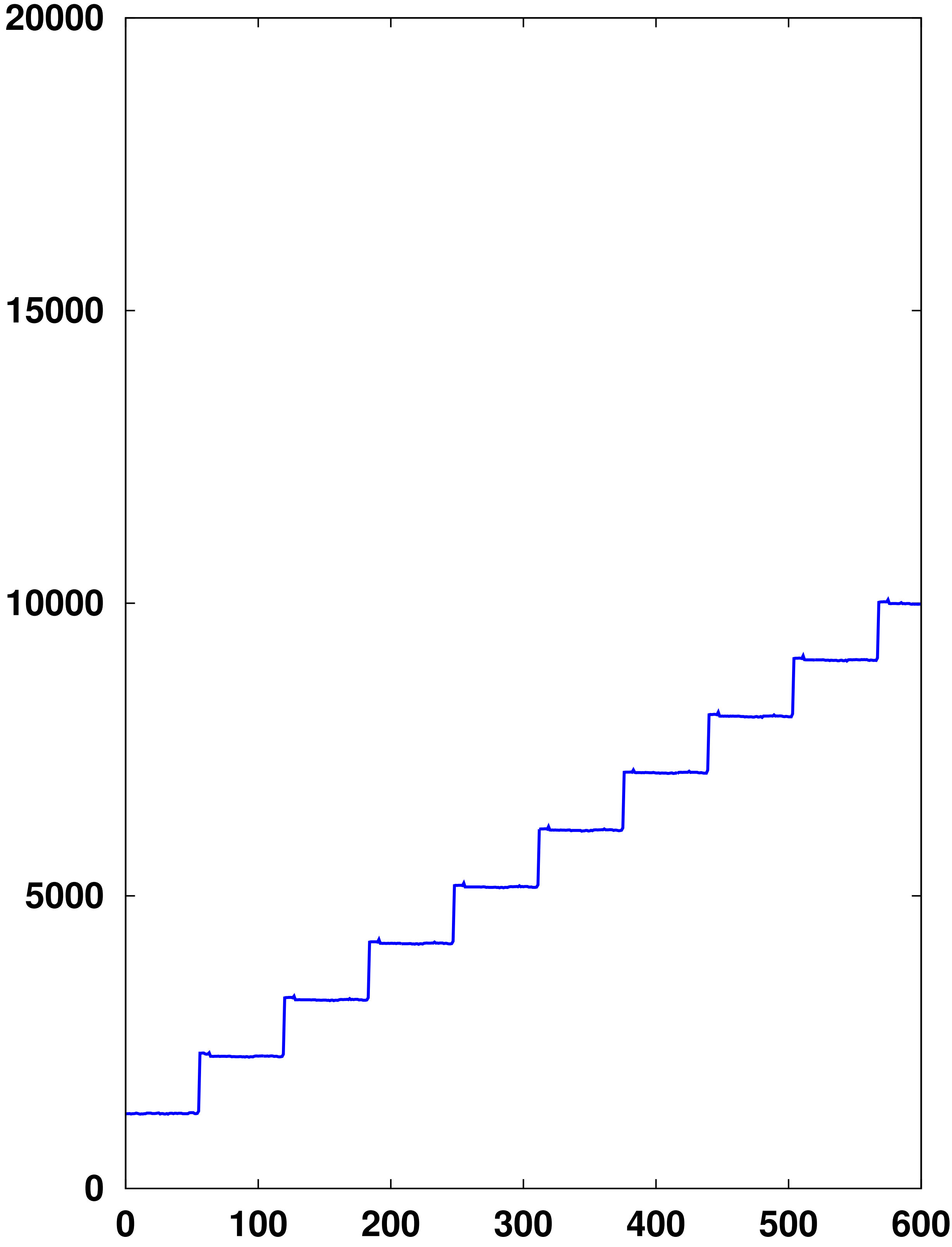
Standard way to
reduce this dependence:
divide cycles by message length.
Warning: Still have dependence.

SHA-512 on the same Phenom:
11.76 cycles/byte for 4096 bytes.
12.17 cycles/byte for 2048 bytes.
12.99 cycles/byte for 1024 bytes.
14.63 cycles/byte for 512 bytes.
17.86 cycles/byte for 256 bytes.
24.47 cycles/byte for 128 bytes.
28.03 cycles/byte for 112 bytes.
15.23 cycles/byte for 111 bytes.
25.81 cycles/byte for 64 bytes.

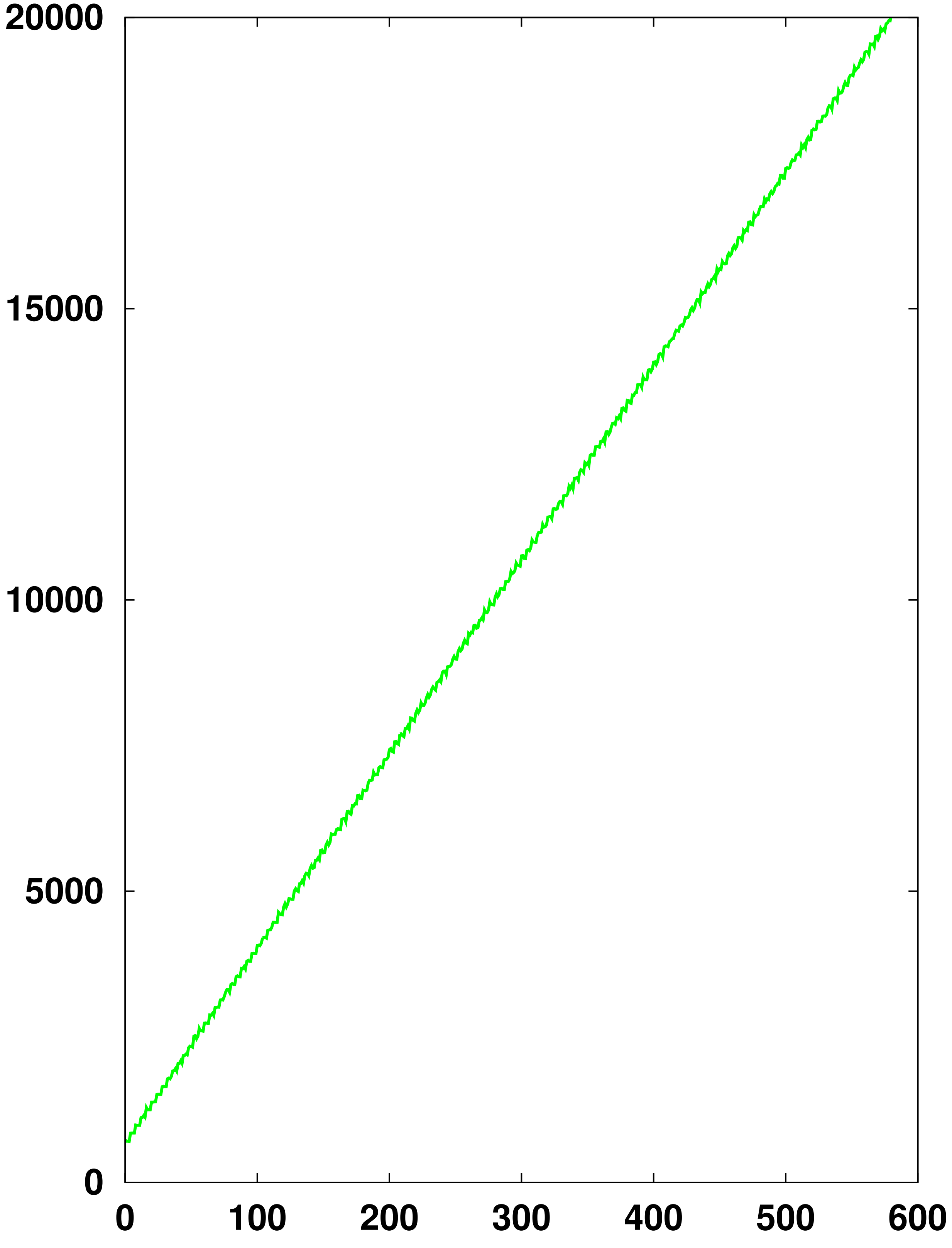
SHA-512 cycles vs. bytes:



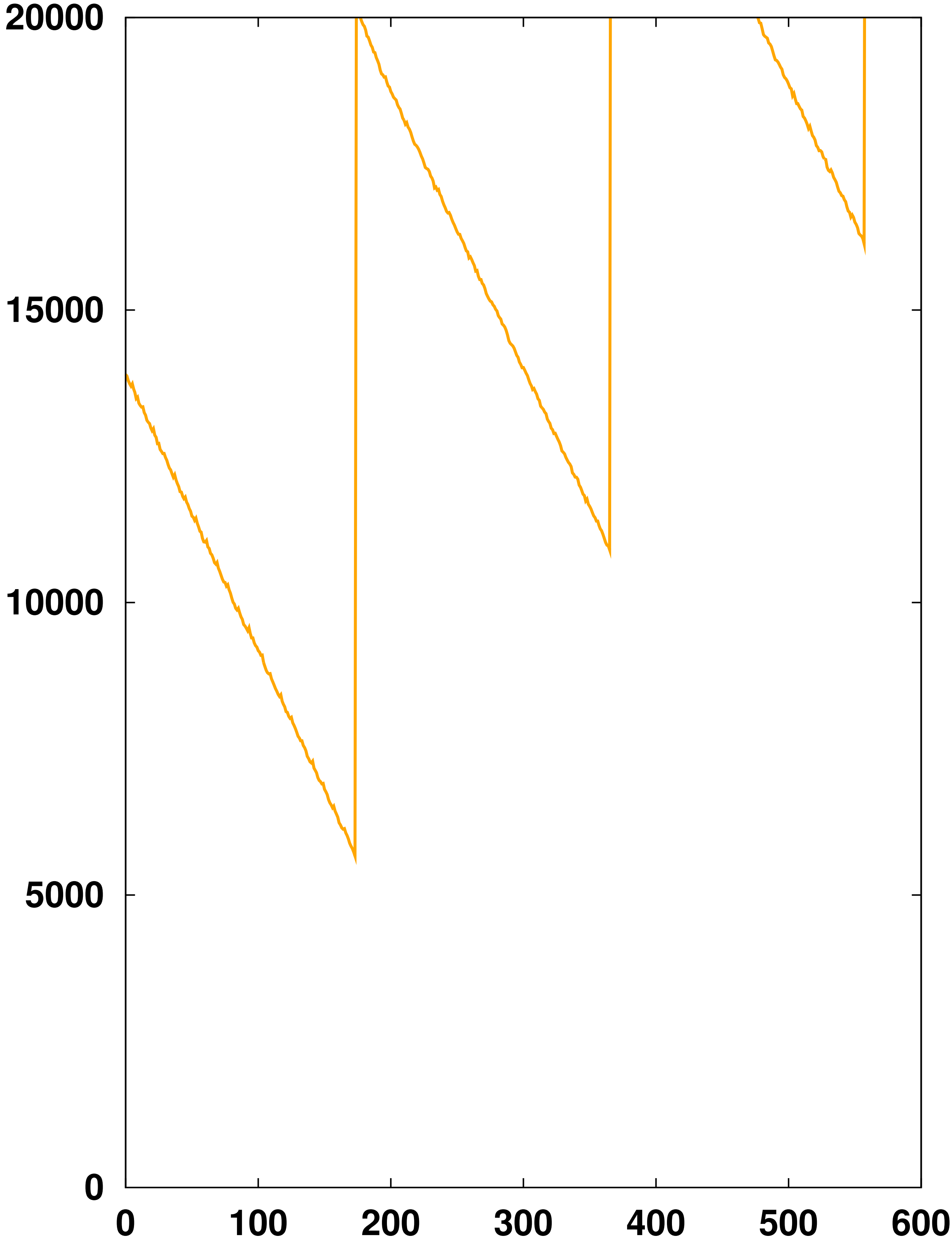
SHA-256 cycles vs. bytes:



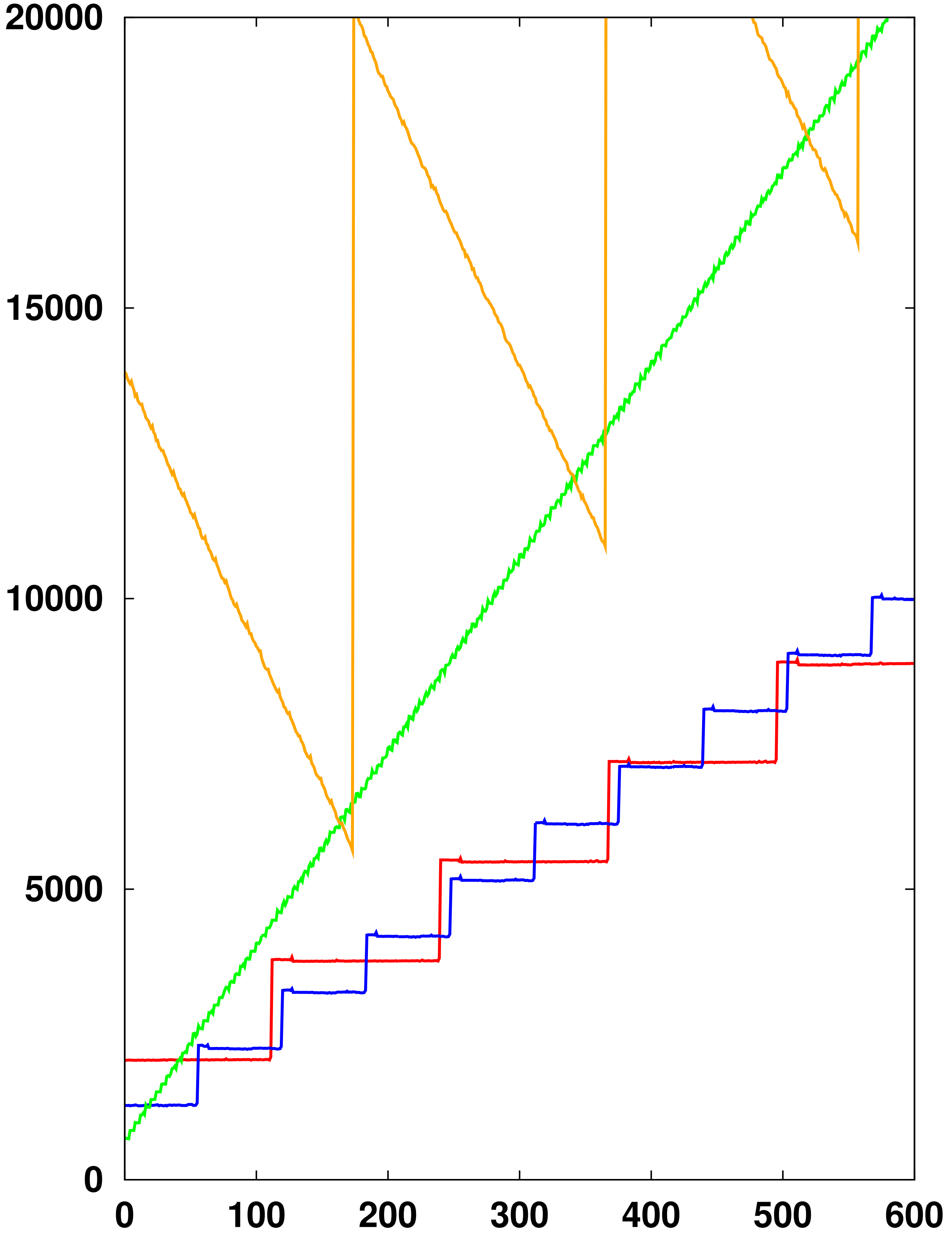
Hamsi cycles vs. bytes:



ECHO-256 cycles vs. bytes:



Cycles vs. bytes:



How fast is hash software?

Answer depends on implementation.

SHA-512: OpenSSL 0.9.8k is $1.31\times$ faster than a simple reference implementation on a typical Core 2 (for 1536 bytes).

Grøstl-256: The "core2duo" implementation is $3.75\times$ faster than the "opt32" implementation and $1.48\times$ faster than the "sphlib" implementation.

A user who cares about speed won't use a slow reference implementation. He'll use the fastest implementation available.

Slowness of unused software has no impact on user's final speed.

The ultimate goal of benchmark reports is to accurately predict the speed that the user will see.
$\Rightarrow$ Report speed of
the fastest implementation.

How fast is hash software?

Answer depends on compiler and on compiler options.

Skein-512, Atom N280, 1536 bytes, `-fomit-frame-pointer`:

177110 cycles: `opt` with `gcc -O2`

How fast is hash software?

Answer depends on compiler and on compiler options.

Skein-512, Atom N280, 1536 bytes, `-fomit-frame-pointer`:

177110 cycles: `opt` with `gcc -O2`

176290 cycles: `opt` with `gcc -O3`

How fast is hash software?

Answer depends on compiler and on compiler options.

Skein-512, Atom N280, 1536 bytes, `-fomit-frame-pointer`:

177110 cycles: `opt` with `gcc -O2`

176290 cycles: `opt` with `gcc -O3`

168580 cycles: `opt` with `gcc -funroll-loops -march=i386 -O3`

How fast is hash software?

Answer depends on compiler and on compiler options.

Skein-512, Atom N280, 1536 bytes, `-fomit-frame-pointer`:

177110 cycles: `opt` with `gcc -O2`

176290 cycles: `opt` with `gcc -O3`

168580 cycles: `opt` with `gcc -funroll-loops -march=i386 -O3`

156470 cycles: `opt` with `gcc -O`

How fast is hash software?

Answer depends on compiler and on compiler options.

Skein-512, Atom N280, 1536 bytes, `-fomit-frame-pointer`:

177110 cycles: `opt` with `gcc -O2`

176290 cycles: `opt` with `gcc -O3`

168580 cycles: `opt` with `gcc -funroll-loops -march=i386 -O3`

156470 cycles: `opt` with `gcc -O`

101460 cycles: `xmm`

Benchmarking in the dark ages

"I've finally finished
my SANDstorm implementation!
Hmmm, how fast is it?"

Benchmarking in the dark ages

“I’ve finally finished
my SANDstorm implementation!
Hmmm, how fast is it?”

Traditional answer:
“I’ll write a timing tool!
I’ll check the clock,
$10000\times$ hash 256 bytes,
check the clock again,
subtract, divide by 10000.”

Benchmarking in the dark ages

"I've finally finished
my SANDstorm implementation!
Hmmm, how fast is it?"

Traditional answer:
"I'll write a timing tool!
I'll check the clock,
10000$\times$ hash 256 bytes,
check the clock again,
subtract, divide by 10000."

Maybe more measurements:
"Oops, lots of overhead
in hashing 256 bytes.
I'll try 4096 bytes."

“Okay, 36.6 cycles/byte
for SANDstorm-256
on my 64-bit machine.
NIST says I have to beat SHA-2.
How fast is SHA-2?”

“Okay, 36.6 cycles/byte
for SANDstorm-256
on my 64-bit machine.
NIST says I have to beat SHA-2.
How fast is SHA-2?”

Traditional answer:
“I’ve written a SHA-256
implementation too.
Let’s see . . . 39.1 cycles/byte.
SANDstorm is faster!
This is a *fair comparison*, because
I wrote both implementations,
and put similar effort into both,
and measured both of them
with my own timing tool.”

Reality: This SHA-256 software is embarrassingly slow.
SHA-256 users actually see much better performance.

To the SANDstorm designer:
You think that SANDstorm can be made faster too? Prove it!
There's nothing "unfair" about comparing best available code.

If SANDstorm *can't* run quickly: comparing lazy implementations makes SANDstorm look better than it actually is. Do we want to reward slow functions? Stupid!

Every dark-ages implementor
builds his own timing tool.
Reports output as “Results”
in an implementation paper.

Summary:
Cryptographic implementor
is the benchmark implementor,
the benchmark operator, and
the competition’s misimplementor.

Every dark-ages implementor
builds his own timing tool.
Reports output as “Results”
in an implementation paper.

Summary:
Cryptographic implementor
is the benchmark implementor,
the benchmark operator, and
the competition’s misimplementor.

This pattern repeats for
every cryptographic implementor.
Hundreds (thousands?) of
separate ad-hoc timing tools
run on various hardware.

Moving out of the dark ages

European Union has funded NESSIE project (2000–2003), ECRYPT I network (2004–2008), ECRYPT II network (2008–2012).

NESSIE's performance evaluators tuned C implementations of 42 cryptographic systems, all supporting the same API; wrote a benchmarking toolkit; ran the toolkit on 25 computers.

Many specific performance results: e.g., 24 cycles/byte on P4 for 128-bit AES encryption.

ECRYPT I had five “virtual labs.”
STVL, symmetric-techniques lab, included four working groups.
STVL WG 1, stream-cipher group, ran eSTREAM (2004–2008).

De Cannière developed new API, wrote new benchmarking toolkit:

- Many more compiler options.
- Improvements in toolkit speed.
- *Published* toolkit $\Rightarrow$ implementation speedups; $>$60 benchmark machines.
- Support for C *and* assembly: e.g. 18 cycles/byte on P4 for third-party asm AES in toolkit.

2006: VAMPIRE, “Virtual Application and Implementation Lab,” started eBATS (“ECRYPT Benchmarking of Asymmetric Systems”), measuring efficiency of public-key encryption, signatures, DH.

2008: VAMPIRE started eBASC (“ECRYPT Benchmarking of Stream Ciphers”) for post-eSTREAM benchmarks.

VAMPIRE also started eBASH (“ECRYPT Benchmarking of All Submitted Hashes”).

New toolkit (Bernstein, Lange):

- New simplified API, co-developed with NaCl API. Reduced implementation cost; increased benefit.
- Improvements in robustness and comprehensiveness.
  e.g. many message lengths.
  e.g. medians *and* quartiles.
- More feedback to implementors.
  e.g. table showing impact of 1615 C compiler options, 945 C++ compiler options; reports show any test failures, compiler error messages, etc.

More operations

Secret-key operations measured in eBASH, eBASC:

- Hash functions.
- Stream ciphers.

Plan to measure more operations:

- Authenticators.
- One-time authenticators.
- Authenticated encryption.

Plan to extend precomputation.

More communication costs

Cryptographic software competes with other networking tools for instruction-cache space.

Current benchmarks don't see this.

Plan to systematically measure varying levels of cache contention.

Also plan to measure costs of many active keys etc.

Also plan to measure performance of batch operations.

More parallelism

Current benchmarks are limited to single-core computations.

Good for high-throughput servers that have many concurrent tasks and that keep all CPU cores busy with separate tasks.

But some applications need minimum *latency* for one task.

Multiple cores save time.
Plan to measure this.
(Multiple machines can save time too; lower priority.)

More security

"Stop using 160-bit hashes!"
. . . Users can easily find
speed of 256-bit hash software,
512-bit hash software, etc.

More security

“Stop using 160-bit hashes!” . . . Users can easily find speed of 256-bit hash software, 512-bit hash software, etc.

“Stop side-channel attacks!” . . . Can users find speed of constant-time hash software?

Plan to separately report speed of software declared to be constant time. (Maybe computer-verified?)

More automation

Implementor finishes software.
Easily sends in for benchmarking.

Software is *manually*
included in benchmark toolkit.
Toolkit is run *manually*.
Manual steps add latency:
often weeks or months.

Plan to have machines
automatically run new software
in resource-limited sandbox.

Much lower latency.
Fast feedback to implementor.

eBASH → public

eBASH has collected
574 implementations of
91 hash functions in 34 families.

http://bench.cr.yp.to
/results-hash.html shows
measurements on 93 machines;
138 machine-ABI combinations.
Even more: XBX for AVR etc.

Each implementation is
recompiled many times
with various compiler options
to identify best working option
for implementation, machine.

Online tables: medians, quartiles
of cycles/byte to hash
8-byte message,
64-byte message,
576-byte message,
1536-byte message,
4096-byte message,
(extrapolated) long message.

Actually have much more data.
e.g. Reports show best options.
e.g. Graphs show medians for
0-byte message, 1-byte message,
2-byte message, 3-byte message,
4-byte message, 5-byte message,
. . ., 2048-byte message.

Implementor → eBASH

Define output size in `api.h`:

```
#define CRYPTO_BYTES 64
```

Implementor → eBASH

Define output size in `api.h`:

```
#define CRYPTO_BYTES 64
```

Define hash function in `hash.c`, e.g. wrapping existing NIST API:

```
#include "crypto_hash.h"
#include "SHA3api_ref.h"
int crypto_hash(
  unsigned char *out,
  const unsigned char *in,
  unsigned long long inlen)
{ Hash(crypto_hash_BYTES*8
        ,in,inlen*8,out);
  return 0; }
```

Send to the mailing list
the URL of a `tar.gz`
with one directory
`crypto_hash/yourhash/ref`
containing `hash.c` etc.

Measurements magically appear!
Much easier than trying
to do your own benchmarks.

More details and options:
http://bench.cr.yp.to
/call-hash.html

Same API works for XBX:
http://xbx.das-labor.org