Benchmarking benchmarking, and optimizing optimization

Daniel J. Bernstein

University of Illinois at Chicago &
Technische Universiteit Eindhoven

Bit operations per bit of plaintext (assuming precomputed subkeys), as listed in recent Skinny paper:

key	ops/bit	cipher
128	88	Simon: 60 ops broken
128	100	NOEKEON
128	117	Skinny
256	144	Simon: 106 ops broken
128	147.2	PRESENT
256	156	Skinny
128	162.75	Piccolo
128	202.5	AES
256	283.5	AES

Bit operations per bit of plaintext (assuming precomputed subkeys), not entirely listed in Skinny paper:

key	ops/bit	cipher
256	54	Salsa20/8
256	78	Salsa20/12
128	88	Simon: 60 ops broken
128	100	NOEKEON
128	117	Skinny
256	126	Salsa20
256	144	Simon: 106 ops broken
128	147.2	PRESENT
256	156	Skinny
128	162.75	Piccolo
128	202.5	AES
256	283.5	AES

Operation counts are a poor model of hardware cost, worse model of software cost.

Pick a cipher: e.g., Salsa20. How fast is Salsa20 software?

First step in analysis: Write simple software.

e.g. Bernstein–van Gastel–Janssen–Lange–Schwabe–Smetsers "TweetNaCl" includes essentially the following implementation of Salsa20:

```
int crypto_core_salsa20(u8 *out,
const u8 *in,const u8 *k,const u8 *c)
{
  u32 w[16],x[16],y[16],t[4];
  int i,j,m;

  FOR(i,4) {
    x[5*i] = ld32(c+4*i);
    x[1+i] = ld32(k+4*i);
    x[6+i] = ld32(in+4*i);
    x[11+i] = ld32(k+16+4*i);
  }

  FOR(i,16) y[i] = x[i];
```

```
  FOR(i,20) {
    FOR(j,4) {
      FOR(m,4) t[m] = x[(5*j+4*m)%16];
      t[1] ^= L32(t[0]+t[3], 7);
      t[2] ^= L32(t[1]+t[0], 9);
      t[3] ^= L32(t[2]+t[1],13);
      t[0] ^= L32(t[3]+t[2],18);
      FOR(m,4) w[4*j+(j+m)%4] = t[m];
    }
    FOR(m,16) x[m] = w[m];
  }

  FOR(i,16) st32(out + 4 * i,x[i] + y[i]);
  return 0;
}
```

```
static const u8 sigma[16]
= "expand 32-byte k";


int crypto_stream_salsa20_xor(u8 *c,
const u8 *m,u64 b,const u8 *n,const u8 *k)
{
  u8 z[16],x[64];
  u32 u,i;
  if (!b) return 0;
  FOR(i,16) z[i] = 0;
  FOR(i,8) z[i] = n[i];
  while (b >= 64) {
    crypto_core_salsa20(x,z,k,sigma);
    FOR(i,64) c[i] = (m?m[i]:0) ^ x[i];
    u = 1;
```

```
    for (i = 8;i < 16;++i) {
      u += (u32) z[i];
      z[i] = u;
      u >>= 8;
    }
    b -= 64;
    c += 64;
    if (m) m += 64;
  }
  if (b) {
    crypto_core_salsa20(x,z,k,sigma);
    FOR(i,b) c[i] = (m?m[i]:0) ^ x[i];
  }
  return 0;
}
```

Next step in analysis:
For each target CPU,
compile the simple code,
and see how fast it is.

Next step in analysis:
For each target CPU,
compile the simple code,
and see how fast it is.

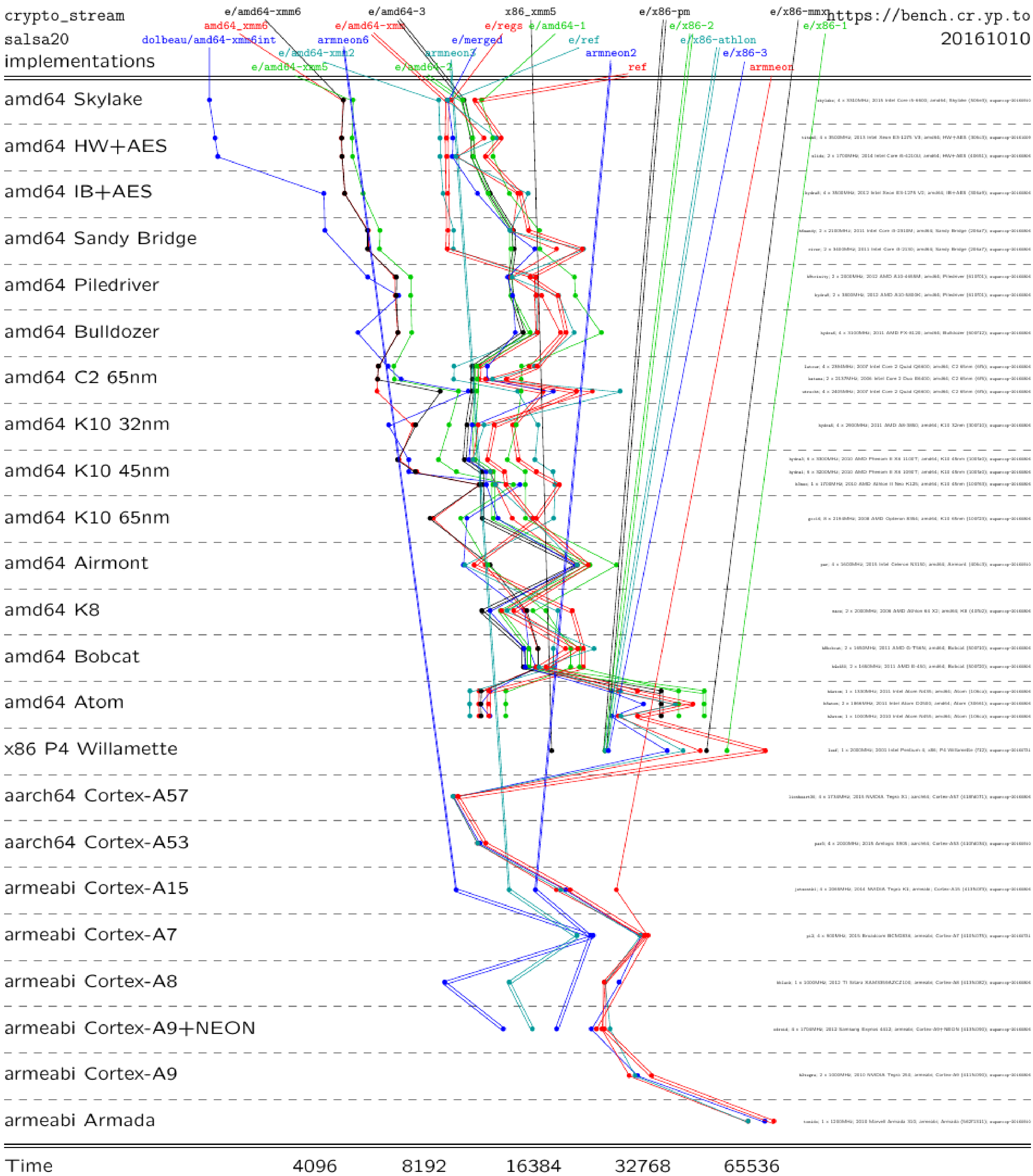
In compiler writer's fantasy world,
the analysis now ends.

Next step in analysis:
For each target CPU,
compile the simple code,
and see how fast it is.

In compiler writer's fantasy world,
the analysis now ends.

"We come so close to optimal on most architectures that we can't do much more without using NP complete algorithms instead of heuristics. We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

Reality is more complicated:



SUPERCOP benchmarking toolkit includes 2064 implementations of 563 cryptographic primitives. >20 implementations of Salsa20.

Haswell: Reasonably simple `ref` implementation compiled with `gcc -O3 -fomit-frame-pointer` is 6.15$\times$ slower than fastest Salsa20 implementation.

`merged` implementation with "machine-independent" optimizations and best of 121 compiler options: 4.52$\times$ slower.

Many more implementations were developed on the way to the (currently) fastest implementation for this CPU.

Many more implementations were developed on the way to the (currently) fastest implementation for this CPU.

This is a common pattern.
Very fast development cycle:
modify the implementation,
check that it still works,
evaluate its performance.

Many more implementations were developed on the way to the (currently) fastest implementation for this CPU.

This is a common pattern. Very fast development cycle: modify the implementation, check that it still works, evaluate its performance.

Results of each evaluation guide subsequent modifications.

Many more implementations were developed on the way to the (currently) fastest implementation for this CPU.

This is a common pattern. Very fast development cycle: modify the implementation, check that it still works, evaluate its performance.

Results of each evaluation guide subsequent modifications.

The software engineer needs fast evaluation of performance.

The unfortunate reality:

Slow evaluation of performance
is often a huge obstacle
to this optimization process.

The unfortunate reality:

Slow evaluation of performance is often a huge obstacle to this optimization process.

When performance evaluation is too slow, the software engineer has to switch context, and then switching back to optimization produces severe cache misses inside software engineer's brain. ("I'm out of the zone.")

The unfortunate reality:

Slow evaluation of performance is often a huge obstacle to this optimization process.

When performance evaluation is too slow, the software engineer has to switch context, and then switching back to optimization produces severe cache misses inside software engineer's brain. ("I'm out of the zone.")

Often optimization is aborted. ("I'll try some other time.")

Goal of this talk:
Speed up the optimization process
by speeding up benchmarking.

“Optimize benchmarking to
help optimize optimization.”

Goal of this talk:
Speed up the optimization process
by speeding up benchmarking.

"Optimize benchmarking to
help optimize optimization."

What are the bottlenecks
that really need speedups?
Measure the benchmarking
process to gain understanding.

"Benchmark benchmarking to
help optimize benchmarking."

Accessing different CPUs

The software engineer writes code on his laptop, but cares about performance on many more CPUs.

Accessing different CPUs

The software engineer writes code on his laptop, but cares about performance on many more CPUs.

Or at least *should* care!

Surprisingly common failure:
A paper with “faster algorithms” actually has slower algorithms running on faster processors.

Accessing different CPUs

The software engineer writes code on his laptop, but cares about performance on many more CPUs.

Or at least *should* care!

Surprisingly common failure:
A paper with “faster algorithms” actually has slower algorithms running on faster processors.

Systematic fix: Optimize each algorithm, new or old, for older and newer processors.

For each target CPU:
Find a machine with that CPU, copy code to that machine (assuming it's on the Internet), collect measurements there.

For each target CPU:
Find a machine with that CPU,
copy code to that machine
(assuming it's on the Internet),
collect measurements there.

But, for security reasons,
most machines on the Internet
disallow access by default,
except access by the owner.

For each target CPU:
Find a machine with that CPU, copy code to that machine (assuming it's on the Internet), collect measurements there.

But, for security reasons, most machines on the Internet disallow access by default, except access by the owner.

Solution #1: Each software engineer buys each CPU.
This is expensive at high end, time-consuming at low end.

Solution #2: Amazon.

Poor coverage of CPUs.

Solution #2: Amazon.
Poor coverage of CPUs.

Solution #3: Compile farms,
such as GCC Compile Farm.
Coverage of CPUs is better
but not good enough for crypto.
Usual goals are OS coverage
and architecture coverage.

Solution #2: Amazon.
Poor coverage of CPUs.

Solution #3: Compile farms,
such as GCC Compile Farm.
Coverage of CPUs is better
but not good enough for crypto.
Usual goals are OS coverage
and architecture coverage.

Solution #4: Figure out who
has the right machines. (How?)
Send email saying "Are you
willing to run this code?"
Slow; unreliable; scales badly.

Solution #5: Send email saying "Can I have an account?"
Saves time but less reliable.

Solution #5: Send email saying "Can I have an account?" Saves time but less reliable.

Solution #6: eBACS.

Good: One-time centralized effort to find machines.

Solution #5: Send email saying "Can I have an account?" Saves time but less reliable.

Solution #6: eBACS.

Good: One-time centralized effort to find machines.

Good: For each code submission, one-time centralized audit.

Solution #5: Send email saying "Can I have an account?" Saves time but less reliable.

Solution #6: eBACS.

Good: One-time centralized effort to find machines.

Good: For each code submission, one-time centralized audit.

Good: High reliability, high coverage, built-in tests.

Solution #5: Send email saying "Can I have an account?" Saves time but less reliable.

Solution #6: eBACS.

Good: One-time centralized effort to find machines.

Good: For each code submission, one-time centralized audit.

Good: High reliability, high coverage, built-in tests.

Bad: Much too slow.

The eBACS data flow

Software engineer has impl: something to benchmark.

Software engineer submits impl: sends package by email or (with centralized account) `git push`.

eBACS manager audits impl, integrates into SUPERCOP.

eBACS manager builds new SUPERCOP package: currently 26-megabyte `xz`.

eBACS manager uploads
and announces package.

Each machine operator
waits until the machine
is sufficiently idle.

Each machine operator
downloads SUPERCOP, runs it.

SUPERCOP scans data
stored on disk from previous runs.
On a typical high-end CPU:
millions of files, several GB.

For each new impl-compiler pair, SUPERCOP compiles+tests impl.

SUPERCOP measures each working compiled impl, saves results on disk.

Typically at least an hour.

SUPERCOP collects all data from this machine, typically 700-megabyte `data.gz`.

Machine operator uploads `data.gz`, announces it.

eBACS manager copies `data.gz` into central database.

Database currently uses 500GB: 53% current uncompressed data, 47% archives of superseded data.

For each new `data.gz` (or for cross-cutting updates): scripts process all results. Typically an hour per machine.

Web pages are regenerated. Under an hour.

In progress: SUPERCOP 2

New database stored centrally:

All impls ever submitted.

Some metadata not affecting measurements. But turning on "publish results" for an impl *does* force new measurements.

All compiled impls.

All checksums of outputs.

All measurements.

All tables, graphs, etc.

When new impl is submitted:

Impl is pushed to compile servers.

Each compiled impl is pushed to checksum machines.

Each working compiled impl is pushed to benchmark machines (when they are sufficiently idle).

Each measurement is available immediately to submitter.

If impl says "publish results": Measurements are put online after comparisons are done.

Wait, what about security?

No more central auditing:
there's no time for it.

Critical integrity concerns:
Can a rogue code submitter
take over the machine?
Or corrupt benchmarks
from other submitters?

Wait, what about security?

No more central auditing:
there's no time for it.

Critical integrity concerns:
Can a rogue code submitter
take over the machine?
Or corrupt benchmarks
from other submitters?

Concerns start before code is
tested and measured: compilers
have bugs, sometimes serious.

Wait, what about security?

No more central auditing: there's no time for it.

Critical integrity concerns: Can a rogue code submitter take over the machine? Or corrupt benchmarks from other submitters?

Concerns start before code is tested and measured: compilers have bugs, sometimes serious.

Smaller availability concerns: e.g., Bitcoin mining.

SUPERCOP 1 sets some OS-level resource limits: impl cannot open any files, cannot fork any processes.

SUPERCOP 2 manages pool of uids and chroot jails on each compile server, checksum machine, benchmark machine.

Enforces reasonable policy for files legitimately used in compiling an impl.

More difficult to enforce: integrity policy for, e.g., tables comparing impls.