eBACS: ECRYPT Benchmarking of Cryptographic Systems

Daniel J. Bernstein

How fast is cryptographic software?

The impact of the CPU: two examples

computer name	`berry2`	`rome0`
computer size	Raspberry Pi 2	big server
CPUs	1	2
CPU name	Broadcom BCM2836	AMD EPYC 7742
cores per CPU	4	64
core cycles/second	$0.9 \cdot 10^9$	$2.245 \cdot 10^9$
instruction set	32-bit ARM	64-bit AMD
microarchitecture	ARM Cortex-A7	AMD Zen 2

Instruction set says which instructions the CPU provides.
Microarchitecture says how many cycles each instruction uses,
how many instructions a core runs at the same time, etc.

Reducing the impact of the CPU on benchmarks

We measure cycles for a computation on one core.

Cycle counts are unaffected by other cores.
Normally 128 cores are running 128 independent computations.

Reducing the impact of the CPU on benchmarks

We measure cycles for a computation on one core.

Cycle counts are unaffected by other cores.
Normally 128 cores are running 128 independent computations.

Faster cores normally have the same cycle counts
if they have the same instruction set and microarchitecture.

Reducing the impact of the CPU on benchmarks

We measure cycles for a computation on one core.

Cycle counts are unaffected by other cores.
Normally 128 cores are running 128 independent computations.

Faster cores normally have the same cycle counts
if they have the same instruction set and microarchitecture.

Exception: if a computation is using RAM outside a core,
then computation time depends on RAM speed
and depends on what other cores are doing.

Reducing the impact of the CPU on benchmarks

We measure cycles for a computation on one core.

Cycle counts are unaffected by other cores.
Normally 128 cores are running 128 independent computations.

Faster cores normally have the same cycle counts
if they have the same instruction set and microarchitecture.

Exception: if a computation is using RAM outside a core,
then computation time depends on RAM speed
and depends on what other cores are doing.

Another complication: Overclocking (Turbo Boost, Turbo Core) can runs more cycles/second on a core, although (1) it has much less effect when multiple cores are active and (2) it is a security problem. We normally disable overclocking.

Variations when a computation is repeated

Many reasons that measurements are not stable:

- First few runs of a computation are generally slower:
  CPU is loading code into cache, predicting branches, . . .
- Computation time can depend on randomness:
  e.g., “generate random p, repeat if p is not prime”.
- Often computation time depends on secret data.
  Often this is a security problem.

We run many measurements of each computation,
reporting medians and quartiles to see variability.
We use high-precision cycle counters to reduce noise.

Variations across different software

$$\begin{tikzcd}
\boxed{\text{Family}} & \boxed{\text{Primitive}} & \boxed{\text{Implementation}} & \boxed{\text{Compiler}} \\
 & & \boxed{\texttt{mem\_opt}} & \\
 & & \boxed{\texttt{aarch64}} & \\
 & \boxed{\texttt{aimer128f}} \arrow[uur, bend left] \arrow[ur] \arrow[dr] \arrow[dddr, bend left] & & \boxed{\texttt{clang -O2}} \\
\boxed{\text{AIMer}} \arrow[ur, bend left] \arrow[dr, bend right] & & \boxed{\texttt{avx2}} \arrow[ur] \arrow[dr] & \\
 & \boxed{\texttt{aimer256s}} & & \boxed{\texttt{gcc -O3}} \\
 & & \boxed{\texttt{ref}} &
\end{tikzcd}$$

Handling multiple implementations and compilers

Normally users who care about speed will take the fastest software. The point of benchmarking: predict the results.

On each CPU, for each primitive,
we try all combinations of implementation+compiler
and then **report measurements of the fastest**.

Handling multiple implementations and compilers

Normally users who care about speed will take the fastest software. The point of benchmarking: predict the results.

On each CPU, for each primitive,
we try all combinations of implementation+compiler
and then **report measurements of the fastest**.

If there's optimized code then measurements ignore slow reference code. Reference code can focus on simplicity.

Managing benchmark updates

Anyone can submit new primitives and new implementations to our SUPERCOP benchmarking framework.

Currently 4541 implementations from hundreds of people. 1407 different primitives.

Each SUPERCOP release is run on many CPUs, including old CPUs for direct comparisons to old papers. Results are tabulated and graphed on https://bench.cr.yp.to.

People who have the same CPUs can double-check the results by downloading and running SUPERCOP.

Important warnings

Important warning 1: Often the speed for future users will be better because a future implementation is faster.

Important warnings

Important warning 1: Often the speed for future users will be better because a future implementation is faster.

Important warning 2: Compiler writers often claim that compilers produce near-optimal code; this is not true.

Important warnings

Important warning 1: Often the speed for future users will be better because a future implementation is faster.

Important warning 2: Compiler writers often claim that compilers produce near-optimal code; this is not true.

Important warning 3: Total cost of application includes cryptographic computation + cryptographic communication + other costs. Often a faster computation isn't better for application: e.g., outweighed by communication costs.

Does the software work correctly?

Automated testing in SUPERCOP

SUPERCOP has added more and more tests over the years. Beyond the measurement web pages, SUPERCOP has separate web pages reporting test results for implementors.

One important test: SUPERCOP uses a deterministic RNG to generate known inputs and known randomness, and then hashes the outputs to obtain a “checksum”. (“Test vectors”/“known-answer tests” are the same idea but need more storage.)

SUPERCOP actually measures the fastest implementation *that produces the expected checksum*. Failed checksums are reported.

Automated testing in SUPERCOP, continued

Some other important aspects of SUPERCOP's testing:

- SUPERCOP tries invalid inputs (e.g., modified ciphertexts), included in checksums.
- SUPERCOP includes tests for software reading outside input buffers or writing outside output buffers.
- SUPERCOP includes TIMECOP 2, which watches test runs to see if there is data flow from secrets (e.g., randomness) to array indices and branch conditions, for implementations marked as constant-time. "**T:**" means an implementation is not marked as constant-time.
- Two of the SUPERCOP computers are using an experimental TIMECOP modification to watch for data flow from secrets to divisions.

The SUPERCOP API

The importance of a shared API

All submitted implementations have to follow the same API so that they can all be tested+measured by unified central code in SUPERCOP.

SUPERCOP API (`crypto_sign` etc.) was introduced by 2008 Bernstein–Lange to unify the API for a broad range of cryptosystems. Goes far beyond previous efforts such as NESSIE, eSTREAM, our BATMAN, and various library APIs.

SUPERCOP API has been reused by various libraries, by various competitions, and by various further tests.