

Getting a subroutine right: a case study

Daniel J. Bernstein

Sorting arrays of secret integers

Typical applications inside cryptographic primitives:

- Insert bottom bits $1, 1, \dots, 1, 1, 0, 0, \dots, 0, 0$ into a random array. Sort the array to generate a random vector with exactly that many 1s.

Sorting arrays of secret integers

Typical applications inside cryptographic primitives:

- Insert bottom bits $1, 1, \dots, 1, 1, 0, 0, \dots, 0, 0$ into a random array. Sort the array to generate a random vector with exactly that many 1s.
- Sort randomness on top of $0, 1, 2, 3, \dots$ to generate a random permutation.

Sorting arrays of secret integers

Typical applications inside cryptographic primitives:

- Insert bottom bits $1, 1, \dots, 1, 1, 0, 0, \dots, 0, 0$ into a random array. Sort the array to generate a random vector with exactly that many 1s.
- Sort randomness on top of $0, 1, 2, 3, \dots$ to generate a random permutation.
- Sort a permutation on top of some data to apply the permutation to the data.

Sorting arrays of secret integers

Typical applications inside cryptographic primitives:

- Insert bottom bits $1, 1, \dots, 1, 1, 0, 0, \dots, 0, 0$ into a random array. Sort the array to generate a random vector with exactly that many 1s.
- Sort randomness on top of $0, 1, 2, 3, \dots$ to generate a random permutation.
- Sort a permutation on top of some data to apply the permutation to the data.

Also many higher-level security protocols:
shuffling votes, shuffling network packets, etc.

Is sorting secrets safe?

Most sorting algorithms have data-dependent branches: quicksort, heapsort, mergesort, ...

Is sorting secrets safe?

Most sorting algorithms have data-dependent branches: quicksort, heapsort, mergesort, ...

Round2 KEM **says**: “Radix sort is used as **constant-time sorting algorithm**.”

No, radix sort has data-dependent array indices.

Is sorting secrets safe?

Most sorting algorithms have data-dependent branches: quicksort, heapsort, mergesort, ...

Round2 KEM **says**: “Radix sort is used as **constant-time sorting algorithm**.”

No, radix sort has data-dependent array indices.

Can attackers deduce the secret data from branch timings or cache timings, as in many other timing attacks? Maybe! Unnecessary attack surface.

Bubblesort to the rescue

```
void int32_sort(int32_t *x, long long n)
{ for (long long j = n; j > 1; --j)
    for (long long i = 1; i < j; ++i)
        crypto_int32_minmax(&x[i-1], &x[i]);
}
```

`crypto_int32_minmax(&u, &v)` from [cryptoint](#)
sets `u` and `v` to min and max of original values.
`cryptoint` is designed to run in constant time.

Bubblesort to the rescue

```
void int32_sort(int32_t *x, long long n)
{ for (long long j = n; j > 1; --j)
    for (long long i = 1; i < j; ++i)
        crypto_int32_minmax(&x[i-1], &x[i]);
}
```

`crypto_int32_minmax(&u, &v)` from [cryptoint](#) sets `u` and `v` to min and max of original values. `cryptoint` is designed to run in constant time.

Bubblesort is an example of a **sorting network**: a sequence of `minmax` at predictable positions.

Speed

A faster sorting network

```
void int32_sort(int32_t *x, long long n)
{ if (n < 2) return;
  long long t = 1; while (t < n - t) t += t;
  for (long long p = t; p > 0; p >>= 1) {
    for (long long i = 0; i < n - p; ++i)
      if (!(i & p))
        crypto_int32_minmax(x+i, x+i+p);
    for (long long q = t; q > p; q >>= 1)
      for (long long i = 0; i < n - q; ++i)
        if (!(i & p))
          crypto_int32_minmax(x+i+p, x+i+q);
  }
}
```

What is this algorithm?

Previous slide: C translation of 1973 Knuth “merge exchange”, which is a simplified version of 1968 Batcher “odd-even merge” sorting network.

This uses about $n(\log_2 n)^2/4$ comparisons.
Bubblesort used about $n^2/2$ comparisons.

What is this algorithm?

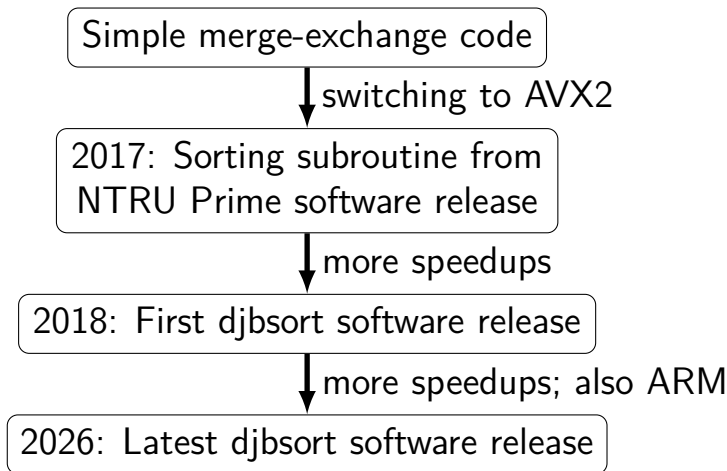
Previous slide: C translation of 1973 Knuth “merge exchange”, which is a simplified version of 1968 Batcher “odd-even merge” sorting network.

This uses about $n(\log_2 n)^2/4$ comparisons.

Bubblesort used about $n^2/2$ comparisons.

Beware that many sorting-network descriptions are more limited: e.g., requiring n to be a power of 2. Also, Wikipedia claims that “Sorting networks . . . are **not capable of handling arbitrarily large inputs**”.

Even faster constant-time sorting



<https://sorting.cr.yp.to>

How big is the constant-time penalty?

How big is the constant-time penalty?

Next slide compares the following libraries:

- stdsort: built-in C++ `std::sort`.

How big is the constant-time penalty?

Next slide compares the following libraries:

- `stdsort`: built-in C++ `std::sort`.
- [herf](#): 2001 radixsort code from Michael Herf.

How big is the constant-time penalty?

Next slide compares the following libraries:

- `std::sort`: built-in C++ `std::sort`.
- [herf](#): 2001 radixsort code from Michael Herf.
- [aspas](#): from a 2018 paper.

How big is the constant-time penalty?

Next slide compares the following libraries:

- `std::sort`: built-in C++ `std::sort`.
- [herf](#): 2001 radixsort code from Michael Herf.
- [aspas](#): from a 2018 paper.
- [vqsort/highway](#): introduced by Google in 2022.

How big is the constant-time penalty?

Next slide compares the following libraries:

- `stdsort`: built-in C++ `std::sort`.
- [herf](#): 2001 radixsort code from Michael Herf.
- [aspas](#): from a 2018 paper.
- [vqsort/highway](#): introduced by Google in 2022.
- [vxsort](#): used by Microsoft starting in 2020.

How big is the constant-time penalty?

Next slide compares the following libraries:

- `stdsort`: built-in C++ `std::sort`.
- [herf](#): 2001 radixsort code from Michael Herf.
- [aspas](#): from a 2018 paper.
- [vqsort/highway](#): introduced by Google in 2022.
- [vxsort](#): used by Microsoft starting in 2020.
- [x86simdsort](#): introduced by Intel in 2022.

How big is the constant-time penalty?

Next slide compares the following libraries:

- `stdsort`: built-in C++ `std::sort`.
- `herf`: 2001 radixsort code from Michael Herf.
- `aspas`: from a 2018 paper.
- `vqsort/highway`: introduced by Google in 2022.
- `vxsort`: used by Microsoft starting in 2020.
- `x86simdsort`: introduced by Intel in 2022.
- `far`: the 2020 parent of `vqsort`.

How big is the constant-time penalty?

Next slide compares the following libraries:

- `stdsort`: built-in C++ `std::sort`.
- `herf`: 2001 radixsort code from Michael Herf.
- `aspas`: from a 2018 paper.
- `vqsort/highway`: introduced by Google in 2022.
- `vxsort`: used by Microsoft starting in 2020.
- `x86simdsort`: introduced by Intel in 2022.
- `far`: the 2020 parent of `vqsort`.
- `djb`sort.

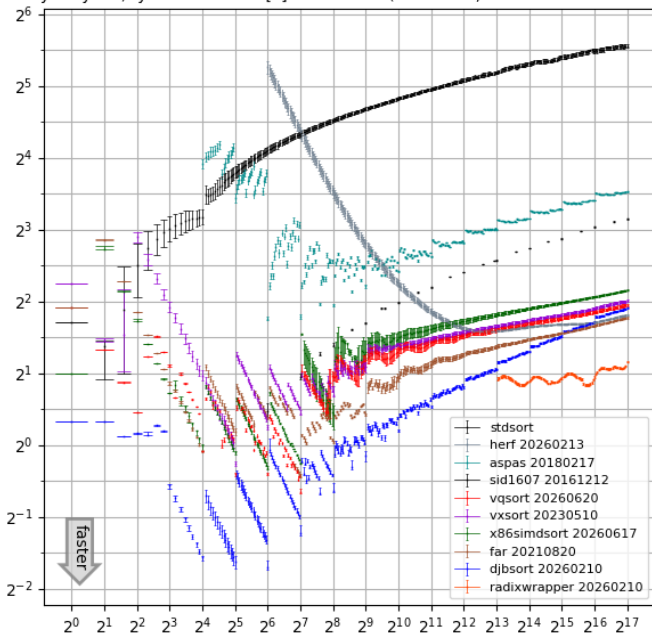
How big is the constant-time penalty?

Next slide compares the following libraries:

- `stdsort`: built-in C++ `std::sort`.
- `herf`: 2001 radixsort code from Michael Herf.
- `aspas`: from a 2018 paper.
- `vqsort/highway`: introduced by Google in 2022.
- `vxsort`: used by Microsoft starting in 2020.
- `x86simdsort`: introduced by Intel in 2022.
- `far`: the 2020 parent of `vqsort`.
- `djbsort`.

Comparison platform: one core of an AMD Ryzen 5 PRO 5650G (2021 CPU launch; Zen 3 microarch).

$y = \text{cycles/byte to sort int32}[x] \text{ on cezanne (20260621; sortbench-20260621)}$



How can $n(\log_2 n)^2/4$ beat $n \log_2 n$?

How can $n(\log_2 n)^2/4$ beat $n \log_2 n$?

Answer: well-known trends in CPU design, reflecting fundamental hardware costs of various operations.

How can $n(\log_2 n)^2/4$ beat $n \log_2 n$?

Answer: well-known trends in CPU design, reflecting fundamental hardware costs of various operations.

Every cycle, typical Intel/AMD core can do
8 “min” ops on 32-bit integers +
8 “max” ops on 32-bit integers.

How can $n(\log_2 n)^2/4$ beat $n \log_2 n$?

Answer: well-known trends in CPU design, reflecting fundamental hardware costs of various operations.

Every cycle, typical Intel/AMD core can do
8 “min” ops on 32-bit integers +
8 “max” ops on 32-bit integers.

That's less hardware than a conditional branch or loading a 32-bit integer from a random address.

How can $n(\log_2 n)^2/4$ beat $n \log_2 n$?

Answer: well-known trends in CPU design, reflecting fundamental hardware costs of various operations.

Every cycle, typical Intel/AMD core can do
8 “min” ops on 32-bit integers +
8 “max” ops on 32-bit integers.

That’s less hardware than a conditional branch or loading a 32-bit integer from a random address.

far, vxsort, vqsort, x86simdsort use sorting networks for small n ; semi-vectorized quicksort for larger n .

`std::sort` uses insertion sort for small n ;

quicksort for large n . (Heapsort if quicksort stalls.)

Security

Security benefits of sorting networks

Automatic protection against timing attacks:
no data flow from secrets into instruction timing.

Security benefits of sorting networks

Automatic protection against timing attacks:
no data flow from secrets into instruction timing.

Automatic protection against denial of service.
(Unlike the quicksort situation of performing very badly for some inputs. How badly? [Depends on library](#). See, e.g., [2021 clang patch](#) to `std::sort`.)

Security benefits of sorting networks

Automatic protection against timing attacks:
no data flow from secrets into instruction timing.

Automatic protection against denial of service.
(Unlike the quicksort situation of performing very badly for some inputs. How badly? [Depends on library](#). See, e.g., [2021 clang patch](#) to `std::sort`.)

If a memory-safety test passes for one size- n array, and the code doesn't inspect array alignment etc., then the code is memory-safe for all size- n arrays.
(Unlike `vxsort`, which [crashes](#) on *some* inputs. Does `vxsort` allow data leaks? Stack smashing? Maybe.)

Is djbsort really constant-time?

`cryptoint` takes responsibility for constant-time `minmax`, including defenses against current compiler screwups. But are those defenses successful?

Is djbsort really constant-time?

`cryptoint` takes responsibility for constant-time `minmax`, including defenses against current compiler screwups. But are those defenses successful?

— Shouldn't we fix the compiler to not screw up?

Is djbsort really constant-time?

cryptoint takes responsibility for constant-time minmax, including defenses against current compiler screwups. But are those defenses successful?

— Shouldn't we fix the compiler to not screw up?

— I have a [clang patch](#) that removes many of the branch-introducing screwups in that compiler. Patch already adopted by [Fil-C](#). But universal rollout will take time; “many” is unlikely to be all; and clang will probably keep adding new screwups.

Another compiler modification

A [2018 paper](#) complained that OpenSSL had “37 different functions” for constant-time computations. Patched clang for `__builtin_ct_choose`.

In 2025, Trail of Bits provided and [upstreamed](#) a patch for a similar `__builtin_ct_select`.

Another compiler modification

A [2018 paper](#) complained that OpenSSL had “37 different functions” for constant-time computations. Patched clang for `__builtin_ct_choose`.

In 2025, Trail of Bits provided and [upstreamed](#) a patch for a similar `__builtin_ct_select`.

So `__builtin_ct_select(a<b,a,b)` is safe?

Another compiler modification

A [2018 paper](#) complained that OpenSSL had “37 different functions” for constant-time computations. Patched clang for `__builtin_ct_choose`.

In 2025, Trail of Bits provided and [upstreamed](#) a patch for a similar `__builtin_ct_select`.

So `__builtin_ct_select(a<b,a,b)` is safe?

No: `a<b` can already generate a branch!

This is why [NaCl's coding rules](#) say “always assume that a comparison in C is compiled into a branch”.

What people actually need is safe comparison operations, as in the `cryptoint` API.

Does OpenSSL protect against compilers?

Example in OpenSSL's `constant_time.h`:

```
lt = constant_time_lt(a, b);  
c = constant_time_select(lt, a, b);
```

Does OpenSSL protect against compilers?

Example in OpenSSL's `constant_time.h`:

```
lt = constant_time_lt(a, b);  
c = constant_time_select(lt, a, b);
```

Part of the `constant_time_lt` computation:

```
return 0 - (a >> (sizeof(a) * 8 - 1));
```

Does OpenSSL protect against compilers?

Example in OpenSSL's `constant_time.h`:

```
lt = constant_time_lt(a, b);  
c = constant_time_select(lt, a, b);
```

Part of the `constant_time_lt` computation:

```
return 0 - (a >> (sizeof(a) * 8 - 1));
```

Compiler might see the 1-bit data path and turn it into a branch, despite subsequent “value barriers”.

Analyze the binaries

C programmers+compilers screw up in many ways.
Important to run `valgrind` to test whether there's
any data flow to branches or array indices
starting from an uninitialized input array.

Analyze the binaries

C programmers+compilers screw up in many ways. Important to run `valgrind` to test whether there's any data flow to branches or array indices starting from an uninitialized input array.

This is part of the automatic tests in SUPERCOP, in the new `djbsort` release, in `libmceliece`, in `libntruprime`, in `lib25519`, etc.

Analyze the binaries

C programmers+compilers screw up in many ways. Important to run `valgrind` to test whether there's any data flow to branches or array indices starting from an uninitialized input array.

This is part of the automatic tests in SUPERCOP, in the new `djbsort` release, in `libmceliece`, in `libntruprime`, in `lib25519`, etc.

This achieves path coverage for `djbsort`'s code once the test runs cover enough array sizes.

The test suites in `libmceliece` etc. cover the specific array sizes needed for those cryptosystems.

Take the compiler out of the loop?

The djbsort code is generated by a Python script.
Could modify this to generate assembly directly.

Maybe preferable: generate code in the [Jasmin](#) language, which has various security features including secret data types.

Verification

The correctness question

Does djbsort actually sort correctly?

What if some comparisons are missing for size- n arrays so the code isn't actually a sorting network?

The correctness question

Does `djb`sort actually sort correctly?

What if some comparisons are missing for size- n arrays so the code isn't actually a sorting network?

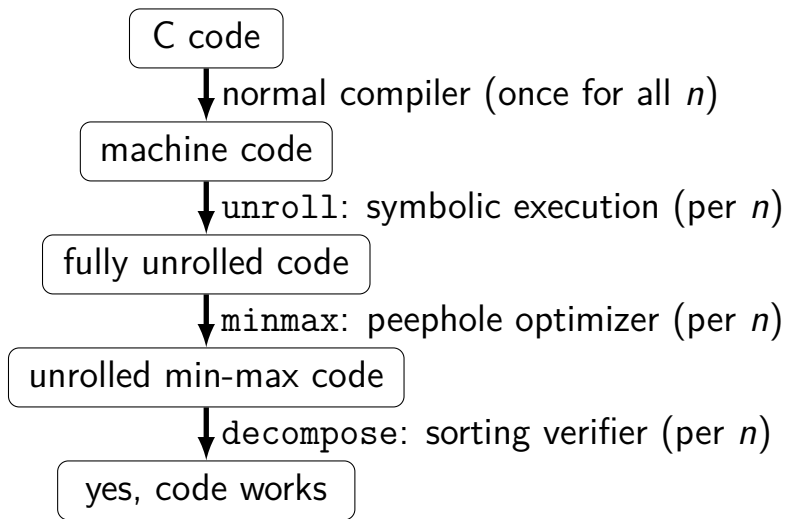
Random-input correctness tests under `valgrind` make clear that it's *close* to a sorting network:

- The code is memory-safe for all size- n inputs.
- The output is correct for *most* size- n inputs.

But could occasional inputs be mis-sorted?

Maybe this is a problem for the application!

Verification pipeline for djb sort



What's going on inside verification?

The `unroll` tool uses [angr](#). See also [saferewrite](#) for a more general unrolling tool using `angr`.

What's going on inside verification?

The `unroll` tool uses [angr](#). See also [saferewrite](#) for a more general unrolling tool using `angr`.

The `minmax` tool rewrites the instruction stream from `angr` to recognize the original min-max operations. In the latest release of `djbsort`, the rewrite rules are verified by an SMT solver.

What's going on inside verification?

The `unroll` tool uses [angr](#). See also [saferewrite](#) for a more general unrolling tool using `angr`.

The `minmax` tool rewrites the instruction stream from `angr` to recognize the original min-max operations. In the latest release of `djbsort`, the rewrite rules are verified by an SMT solver.

The `decompose` tool recognizes the structure of sorting networks built from merging networks, and applies a theorem on merge verification.

What's going on inside verification?

The `unroll` tool uses [angr](#). See also [saferewrite](#) for a more general unrolling tool using `angr`.

The `minmax` tool rewrites the instruction stream from `angr` to recognize the original min-max operations. In the latest release of `djbsort`, the rewrite rules are verified by an SMT solver.

The `decompose` tool recognizes the structure of sorting networks built from merging networks, and applies a theorem on merge verification.

Do these tools have bugs? Does `angr` have bugs? Switching to HOL Light would reduce risks.

Testing vs. verification

```
void int32_sort(int32_t *x, long long n)
{ for (long long j = n; j > 1; --j)
    for (long long i = 1; i < j; ++i)
        if (i != 1 || j != 20 || n != 40)
            crypto_int32_minmax(&x[i-1], &x[i]);
}
```

For $n = 40$: This passes billions of random tests.

Testing vs. verification

```
void int32_sort(int32_t *x, long long n)
{ for (long long j = n; j > 1; --j)
    for (long long i = 1; i < j; ++i)
        if (i != 1 || j != 20 || n != 40)
            crypto_int32_minmax(&x[i-1], &x[i]);
}
```

For $n = 40$: This passes billions of random tests.
But this is rapidly rejected by djb'sort's verification.

Testing vs. verification

```
void int32_sort(int32_t *x, long long n)
{ for (long long j = n; j > 1; --j)
    for (long long i = 1; i < j; ++i)
        if (i != 1 || j != 20 || n != 40)
            crypto_int32_minmax(&x[i-1], &x[i]);
}
```

For $n = 40$: This passes billions of random tests.
But this is rapidly rejected by djb'sort's verification.
This code mis-sorts with probability about 2^{-37} .

Testing vs. verification

```
void int32_sort(int32_t *x, long long n)
{ for (long long j = n; j > 1; --j)
    for (long long i = 1; i < j; ++i)
        if (i != 1 || j != 20 || n != 40)
            crypto_int32_minmax(&x[i-1], &x[i]);
}
```

For $n = 40$: This passes billions of random tests.
But this is rapidly rejected by djb'sort's verification.
This code mis-sorts with probability about 2^{-37} .
Can write tests to catch this; verification is easier!

Coda

This isn't just about sorting

Broader benefits of restricting data flow from inputs to branch conditions and array indices:

- Simplifies verification.
- Makes testing more effective.
- Often outperforms alternatives.
- Helps protect against denial of service.
- Avoids leaking secret data through timings.
(But make sure to also **disable overclocking**,
avoid **division instructions**, etc.)

Use TIMECOP etc. to test for data-flow violations.
Switch to algorithms using straight-line arithmetic.