Sorting: an update

Daniel J. Bernstein

Sorting arrays of secret integers

Cryptosystems that use a sorting subroutine:

- Classic McEliece: various deployments
- NTRU-HPS: another candidate for ISO
- NTRU Prime: now in 13 SSH implementations
- LESS: candidate for NIST
- PERK: candidate for NIST
- CTIDH: non-interactive key exchange
- More that I've noticed: Round2, LEDA, BIG QUAKE, GeMSS, PKP-DSS, WAVE

Why do cryptosystems sort?

Typical applications inside cryptographic primitives:

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

Why do cryptosystems sort?

Typical applications inside cryptographic primitives:

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

Why do cryptosystems sort?

Typical applications inside cryptographic primitives:

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

Why do cryptosystems sort?

Typical applications inside cryptographic primitives:

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

So cryptosystem designers shouldn't sort?

Some lattice KEMs avoid sorting by, e.g., flipping coins independently for each position of a vector (and don't have other reasons to use permutations).

So cryptosystem designers shouldn't sort?

Some lattice KEMs avoid sorting by, e.g., flipping coins independently for each position of a vector (and don't have other reasons to use permutations).

But then some vectors have fewer 1s
and are more vulnerable to close-vector attacks.

So cryptosystem designers shouldn't sort?

Some lattice KEMs avoid sorting by, e.g., flipping coins independently for each position of a vector (and don't have other reasons to use permutations).

But then some vectors have fewer 1s
and are more vulnerable to close-vector attacks.

Meanwhile other vectors have more 1s
and are more likely to trigger decryption failures that leak secret keys. Some attack strategies rely on amplifying the probability of decryption failures.

So cryptosystem designers shouldn't sort?

Some lattice KEMs avoid sorting by, e.g., flipping coins independently for each position of a vector (and don't have other reasons to use permutations).

But then some vectors have fewer 1s
and are more vulnerable to close-vector attacks.

Meanwhile other vectors have more 1s
and are more likely to trigger decryption failures that leak secret keys. Some attack strategies rely on amplifying the probability of decryption failures.

Can these attacks be pushed further? Maybe!

Constant-time sorting

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.

Constant-time sorting

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.

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.

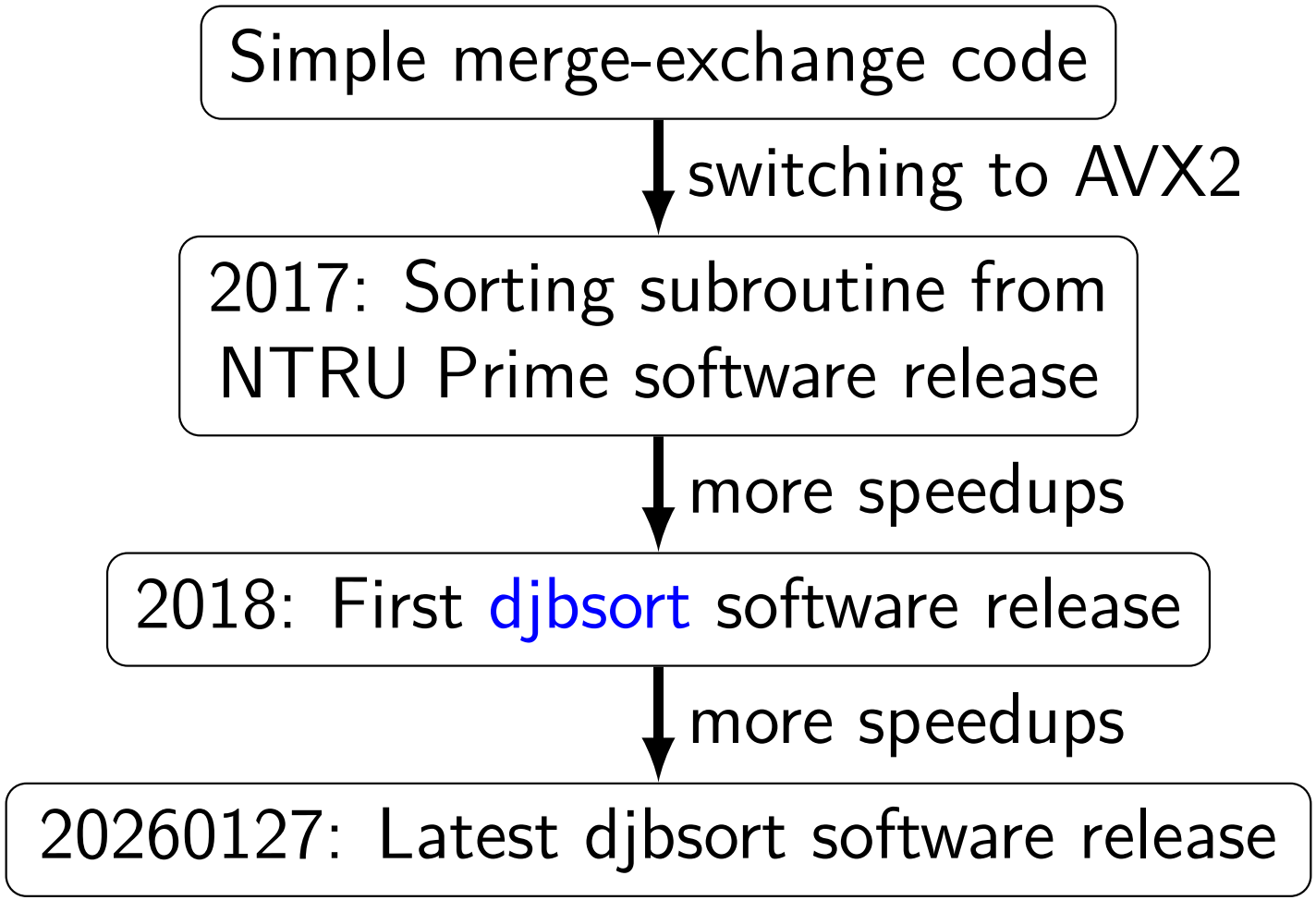
What is this algorithm?

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.

Many sorting-network descriptions are more limited: e.g., requiring n to be a power of 2.
Also, Wikipedia says "Sorting networks ...
are not capable of handling arbitrarily large inputs".

Even faster constant-time sorting



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:

- stdsort: 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:

- 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.

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:

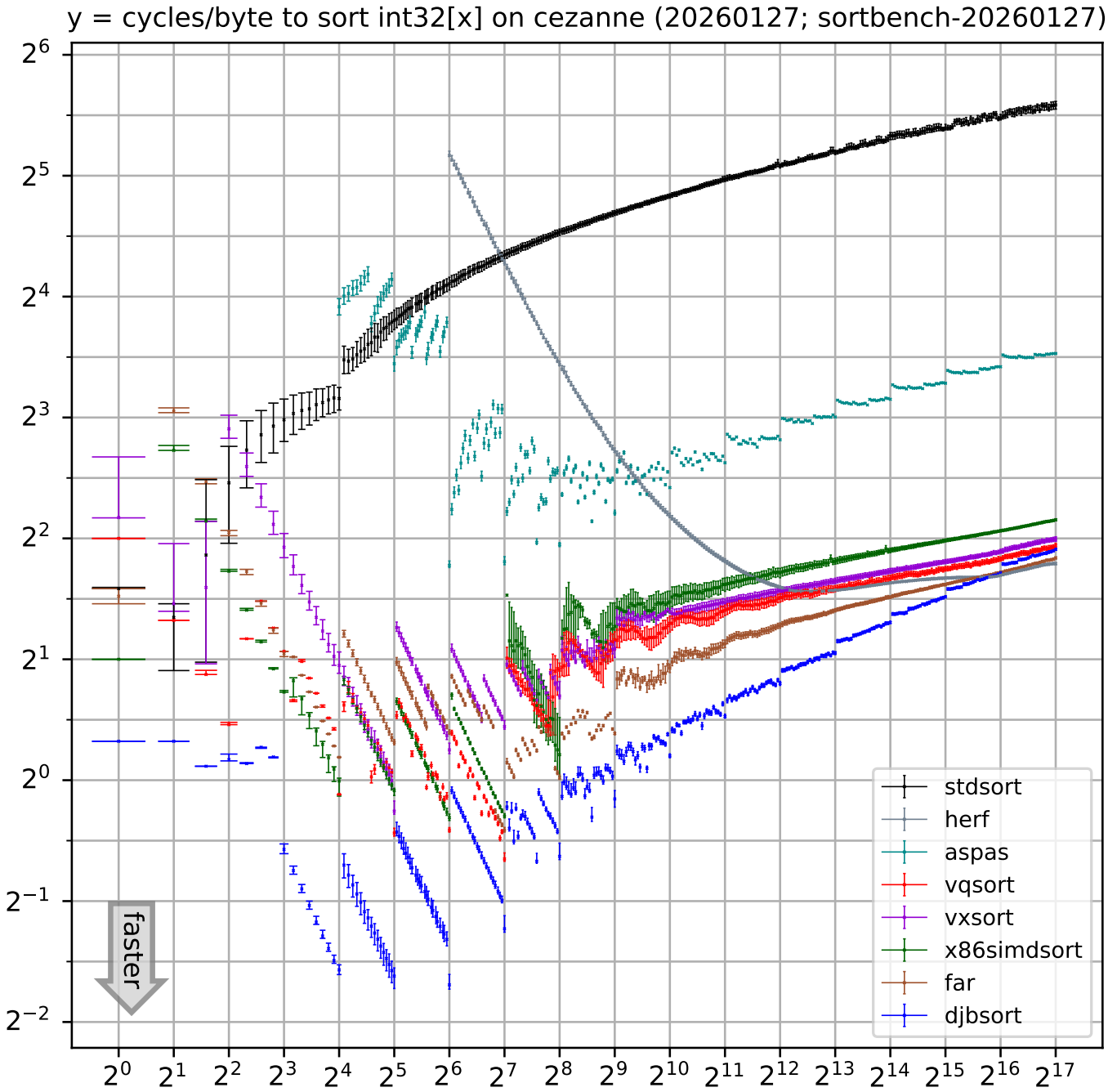
- stdsort: built-in C++ `std::sort`.
- herf: 2001 radixsort code from Michael Herf.
- aspas: from a 2018 paper.
- vqsort/highway: introduced by Google in 2022.
- vxsort: used by Microsoft starting in 2020.
- x86simdsort: introduced by Intel in 2022.
- far: the 2020 parent of vqsort.
- djbsort.

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 = cycles/byte to sort int32[x] on cezanne (20260127; sortbench-20260127)
2^6
2^5
2^4
2^3
2^2
2^1
2^0
2^{-1}
2^{-2}
faster
2^0 2^1 2^2 2^3 2^4 2^5 2^6 2^7 2^8 2^9 2^{10} 2^{11} 2^{12} 2^{13} 2^{14} 2^{15} 2^{16} 2^{17}
stdsort
herf
aspas
vqsort
vxsort
x86simdsort
far
djbsort

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.)

Asymptotics vs. reality

C++03 required `std::sort` to use "approximately $N \log(N)$" comparisons on average. C++11...23 require "$O(N \log(N))$" comparisons for all inputs.

Asymptotics vs. reality

C++03 required `std::sort` to use "approximately $N \log(N)$" comparisons on average. C++11...23 require "$O(N \log(N))$" comparisons for all inputs.

Impact of this rule on `int32` sorting speed on AVX2:

switching `std::sort` to	makes it	and is
bubblesort	slower	prohibited

Asymptotics vs. reality

C++03 required `std::sort` to use "approximately $N \log(N)$" comparisons on average. C++11...23 require "$O(N \log(N))$" comparisons for all inputs.

Impact of this rule on `int32` sorting speed on AVX2:

switching `std::sort` to	makes it	and is
bubblesort	slower	prohibited
djbsort	faster	prohibited

Asymptotics vs. reality

C++03 required `std::sort` to use "approximately $N \log(N)$" comparisons on average. C++11...23 require "$O(N \log(N))$" comparisons for all inputs.

Impact of this rule on `int32` sorting speed on AVX2:

switching `std::sort` to	makes it	and is
bubblesort	slower	prohibited
djbsort	faster	prohibited
insertionsort if $N \leq 2^4$	unchanged	allowed

Asymptotics vs. reality

C++03 required `std::sort` to use "approximately $N \log(N)$" comparisons on average. C++11...23 require "$O(N \log(N))$" comparisons for all inputs.

Impact of this rule on `int32` sorting speed on AVX2:

switching `std::sort` to	makes it	and is
bubblesort	slower	prohibited
djbsort	faster	prohibited
insertionsort if $N \leq 2^4$	unchanged	allowed
bubblesort if $N \leq 2^{60}$	slower	allowed

Asymptotics vs. reality

C++03 required `std::sort` to use "approximately $N \log(N)$" comparisons on average. C++11...23 require "$O(N \log(N))$" comparisons for all inputs.

Impact of this rule on `int32` sorting speed on AVX2:

switching `std::sort` to	makes it	and is
bubblesort	slower	prohibited
djbsort	faster	prohibited
insertionsort if $N \leq 2^4$	unchanged	allowed
bubblesort if $N \leq 2^{60}$	slower	allowed
djbsort if $N \leq 2^{60}$	faster	allowed

Security

Security benefits of sorting networks

Automatic protection against timing attacks.

Security benefits of sorting networks

Automatic protection against timing attacks.

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.

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.

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, 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, 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

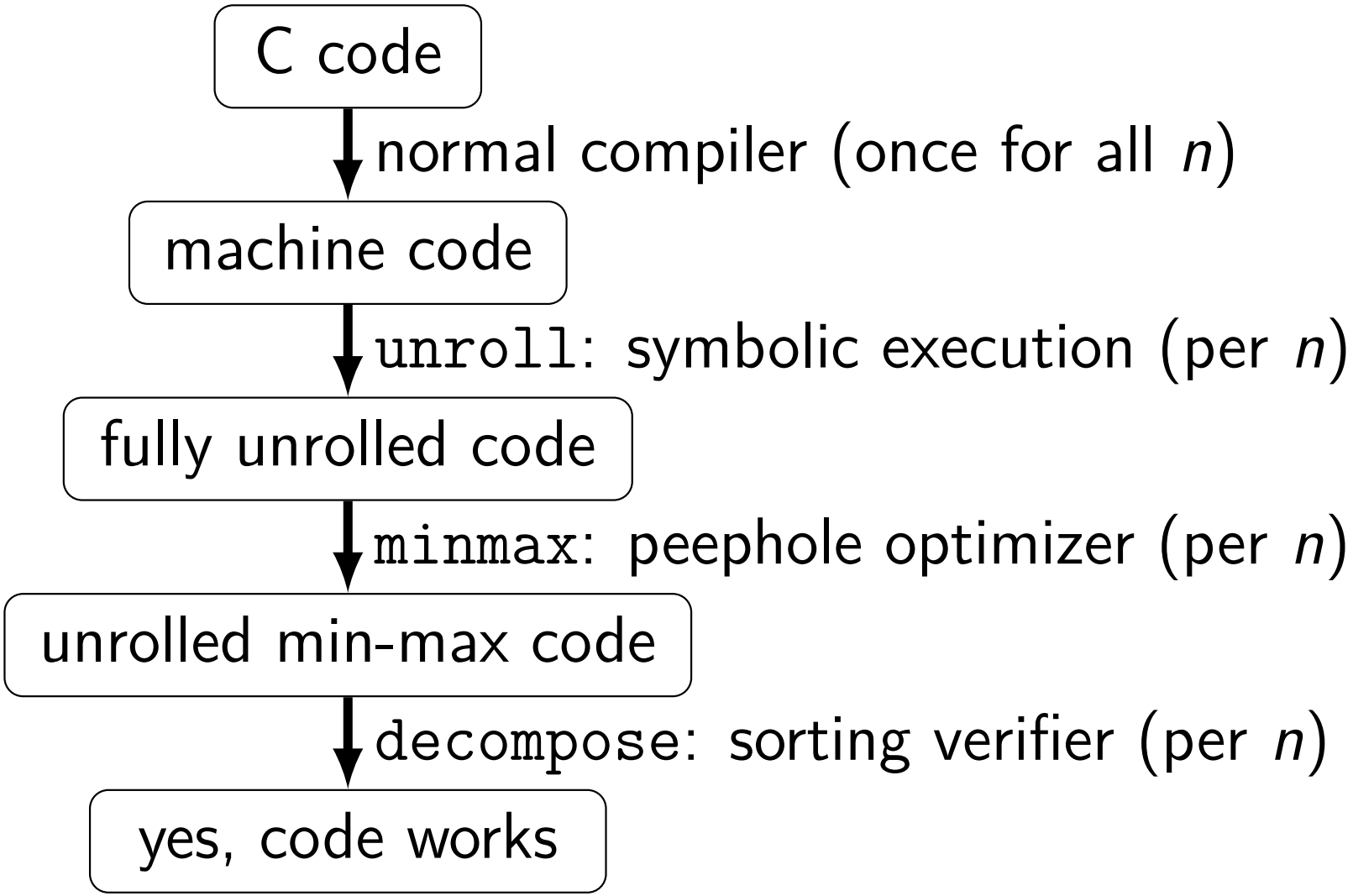
Does djbsort actually sort correctly?
What if some comparisons are missing for size-n arrays so the code isn't actually a sorting network?

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 djbsort



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 djbsort'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 djbsort'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 djbsort's verification.
This code mis-sorts with probability about 2^{-37}.
Can write tests to catch this; verification is easier!