Can cryptographic software be fixed?

D. J. Bernstein

Bob's laptop screen:

```
From: Alice


Thank you for your
submission. We received
many interesting papers,
and unfortunately your
```

Bob assumes this message is something Alice actually sent.

But today's "security" systems fail to guarantee this property. Attacker could have modified or forged the message.

Systems are too complex.

e.g. Firefox 60 (May 2018) code:

4582680 lines in `cpp` files,

3093398 lines in `h` files,

2623454 lines in `c` files, etc.

Systems are too complex.

e.g. Firefox 60 (May 2018) code:
4582680 lines in `cpp` files,
3093398 lines in `h` files,
2623454 lines in `c` files, etc.

Every line in this code has
full control over user messages.

Systems are too complex.
e.g. Firefox 60 (May 2018) code:
4582680 lines in `cpp` files,
3093398 lines in `h` files,
2623454 lines in `c` files, etc.

Every line in this code has full control over user messages.

Critical vulnerabilities fixed in 61: CVE-2018-12359, “Buffer overflow using computed size of canvas element”; CVE-2018-12360, “Use-after-free when using focus()”; CVE-2018-12361, “Integer overflow in SwizzleData”.

Trusted computing base (TCB)

TCB: portion of computer system that is responsible for enforcing the users' security policy.

Trusted computing base (TCB)

TCB: portion of computer system that is responsible for enforcing the users' security policy.

Security policy for this talk:
If message is displayed on
Bob's screen as "`From: Alice`"
then message is from Alice.

Trusted computing base (TCB)

TCB: portion of computer system that is responsible for enforcing the users' security policy.

Security policy for this talk:
If message is displayed on Bob's screen as "`From: Alice`" then message is from Alice.

If TCB works correctly, then message is guaranteed to be from Alice, no matter what the rest of the system does.

Examples of attack strategies:

1. Attacker uses buffer overflow in a device driver to control Linux kernel on Alice’s laptop.

Examples of attack strategies:

1. Attacker uses buffer overflow in a device driver to control Linux kernel on Alice's laptop.

2. Attacker uses buffer overflow in a web browser to control disk files on Bob's laptop.

Examples of attack strategies:

1. Attacker uses buffer overflow in a device driver to control Linux kernel on Alice's laptop.
2. Attacker uses buffer overflow in a web browser to control disk files on Bob's laptop.

Device driver is in the TCB.
Web browser is in the TCB.
CPU is in the TCB. Etc.

Examples of attack strategies:

1. Attacker uses buffer overflow in a device driver to control Linux kernel on Alice's laptop.
2. Attacker uses buffer overflow in a web browser to control disk files on Bob's laptop.

Device driver is in the TCB.
Web browser is in the TCB.
CPU is in the TCB. Etc.

Massive TCB has many bugs, including many security holes.
Any hope of fixing this?

Classic security strategy:

Rearchitect computer systems
to have a much smaller TCB.

Classic security strategy:

Rearchitect computer systems
to have a much smaller TCB.

Carefully audit the TCB.

Classic security strategy:

Rearchitect computer systems to have a much smaller TCB.

Carefully audit the TCB.

e.g. Bob runs many VMs:

VM A
Alice data

VM C
Charlie data

. . .

TCB stops each VM from touching data in other VMs.

Classic security strategy:

Rearchitect computer systems to have a much smaller TCB.

Carefully audit the TCB.

e.g. Bob runs many VMs:

VM A Alice data	VM C Charlie data	. . .

TCB stops each VM from touching data in other VMs.

Browser in VM C isn't in TCB. Can't touch data in VM A, if TCB works correctly.

Classic security strategy:

Rearchitect computer systems
to have a much smaller TCB.

Carefully audit the TCB.

e.g. Bob runs many VMs:

VM A
Alice data

VM C
Charlie data

. . .

TCB stops each VM from
touching data in other VMs.

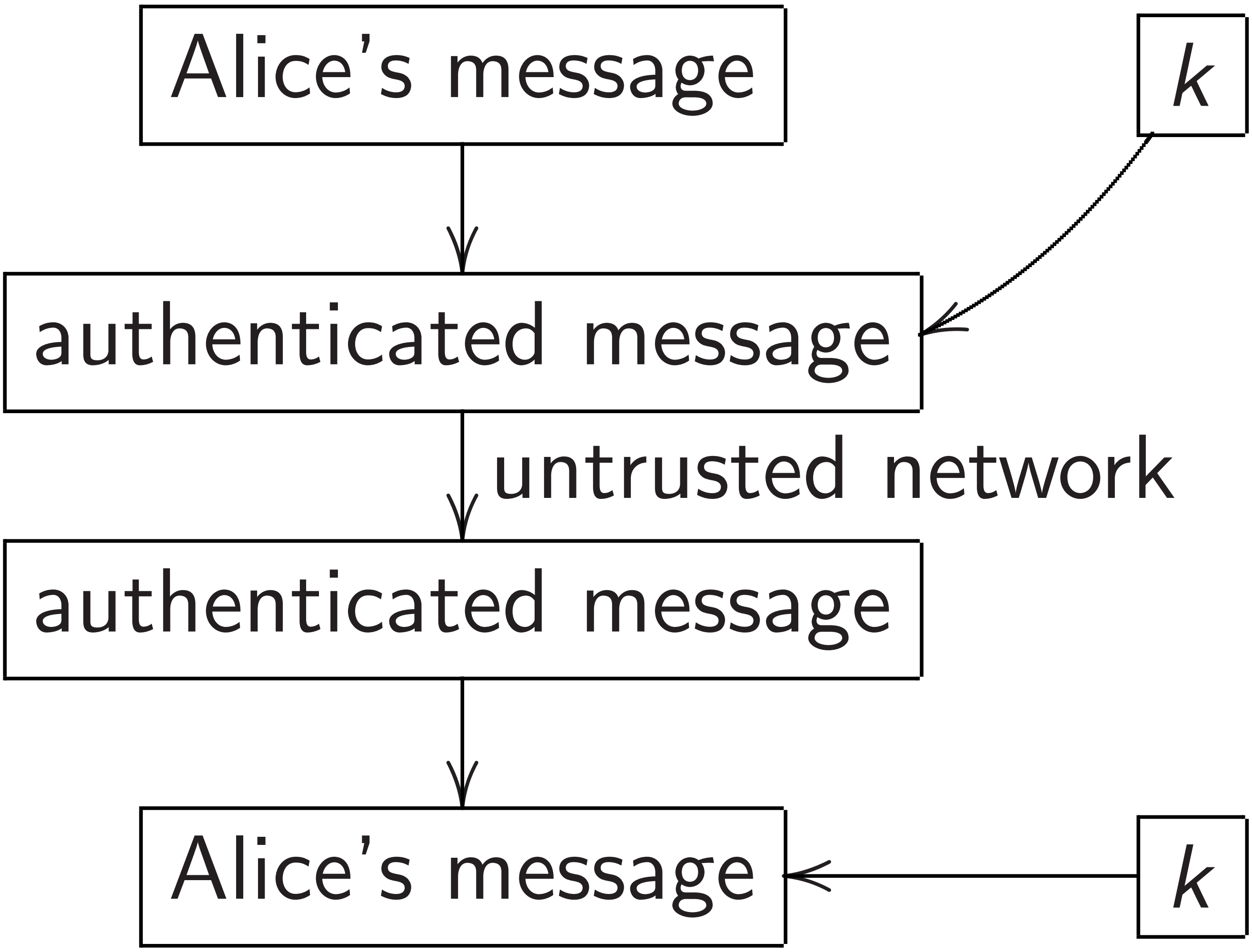
Browser in VM C isn't in TCB.
Can't touch data in VM A,
if TCB works correctly.

Alice also runs many VMs.

Focus of this talk: Cryptography

How does Bob's laptop know that incoming network data is from Alice's laptop?

Cryptographic solution: Message-authentication codes.



Alice's message

k

authenticated message

untrusted network

authenticated message

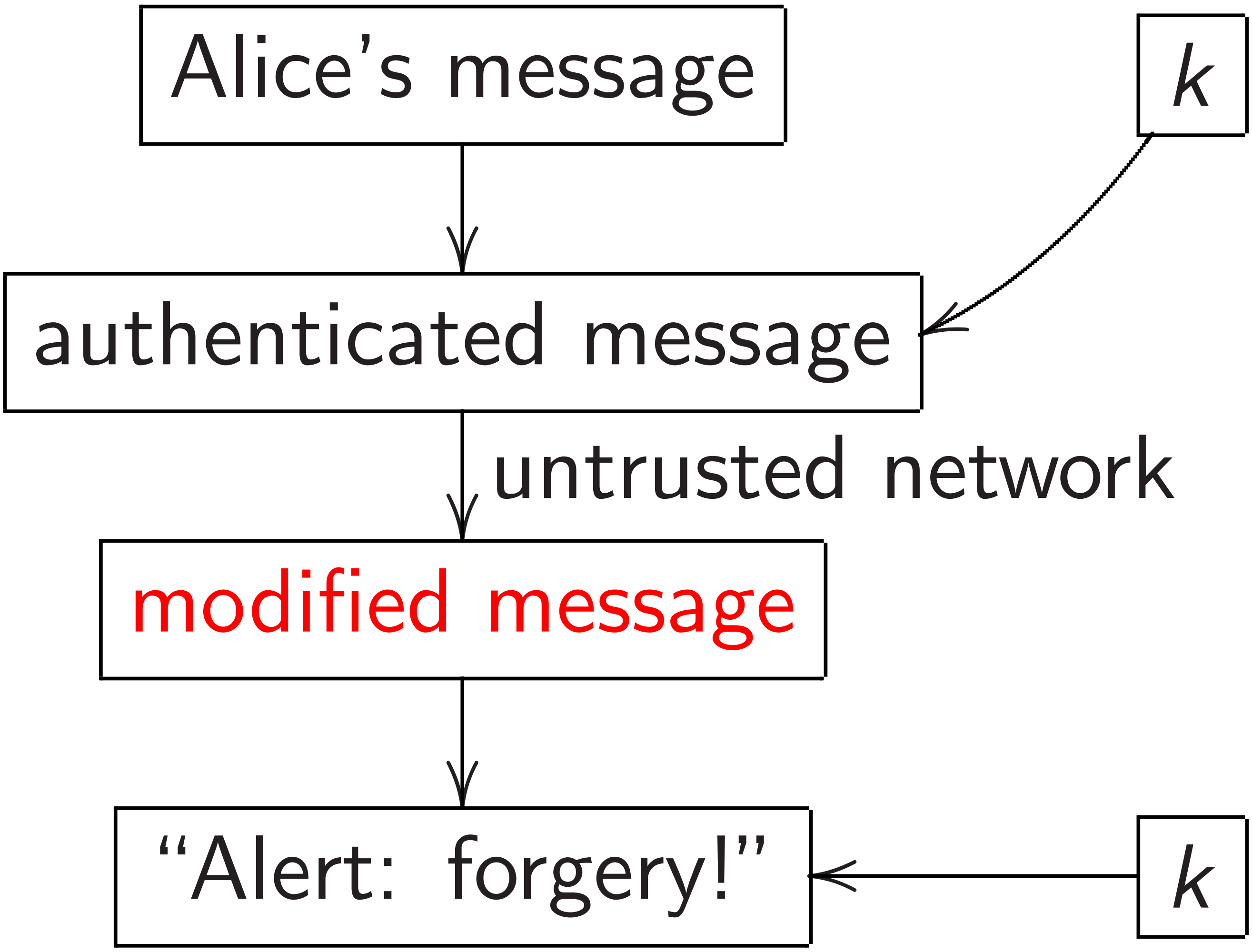
Alice's message

k

Focus of this talk: Cryptography

How does Bob's laptop know that incoming network data is from Alice's laptop?

Cryptographic solution: Message-authentication codes.



Alice's message

k

authenticated message

untrusted network

modified message

"Alert: forgery!"

k

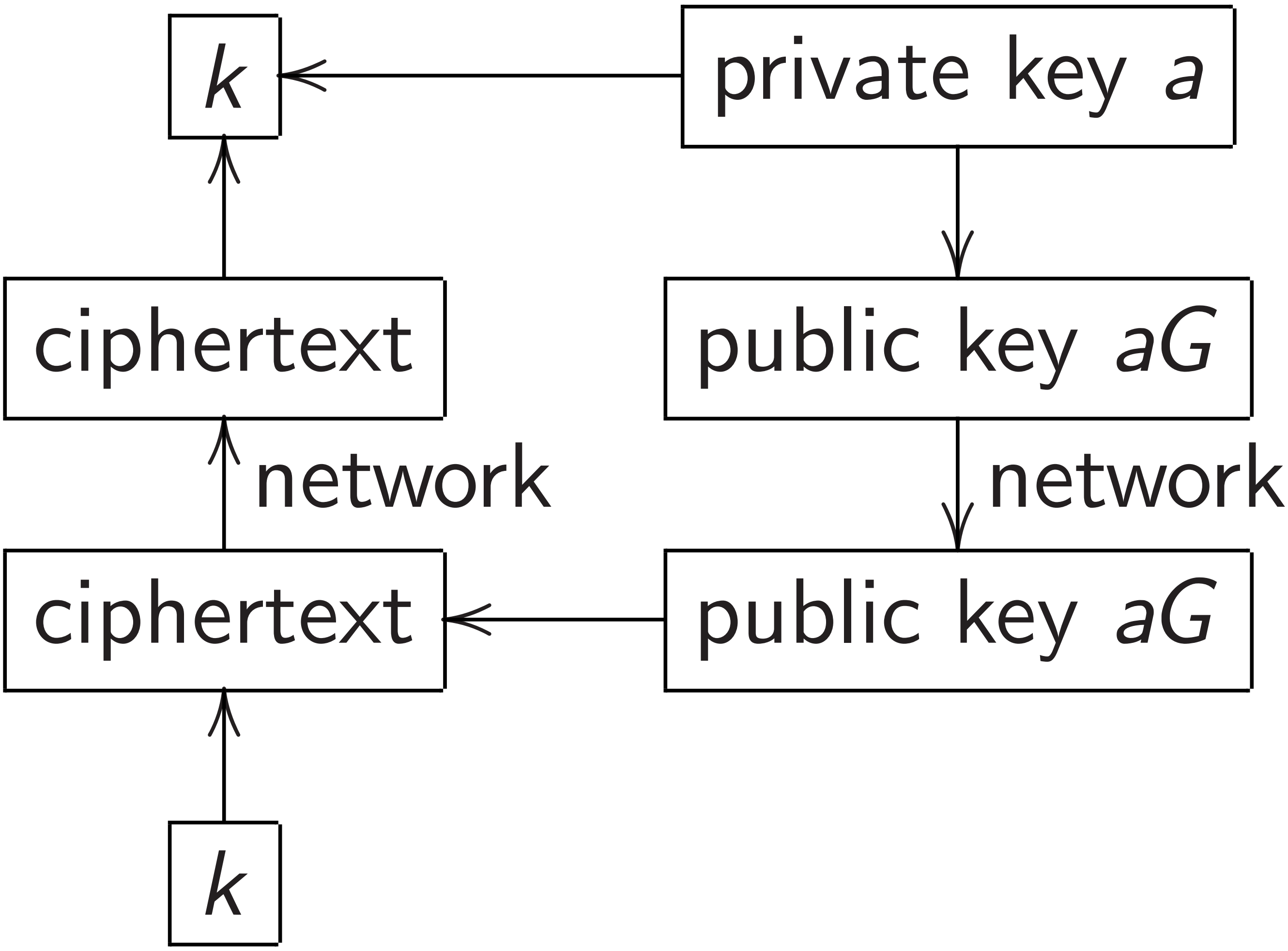
Important for Alice and Bob to share the same secret k.

What if attacker was spying on their communication of k?

Important for Alice and Bob to share the same secret k.

What if attacker was spying on their communication of k?

Solution 1:
Public-key encryption.



k

private key a

ciphertext

public key aG

network

network

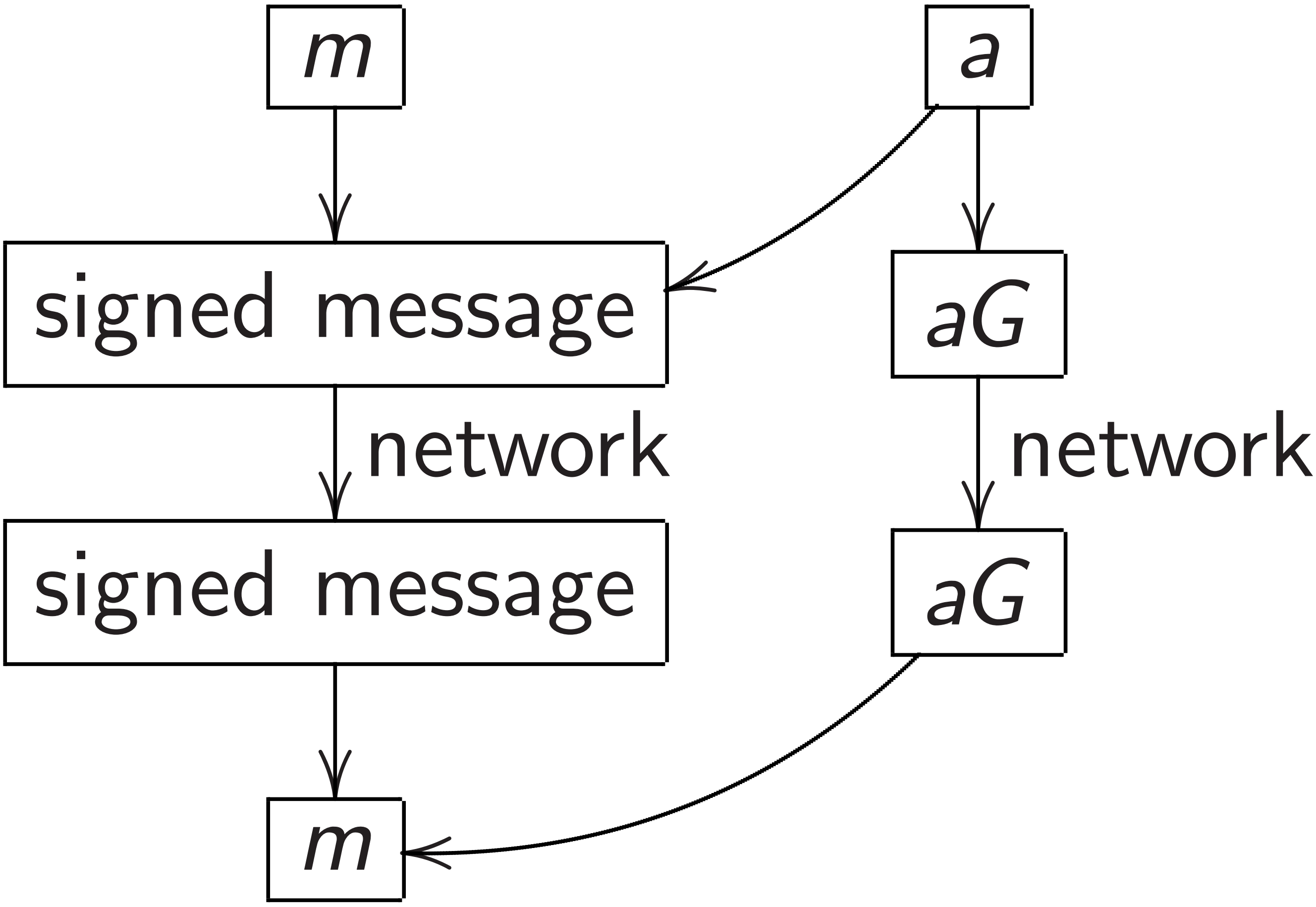
ciphertext

public key aG

k

Solution 2:

Public-key signatures.



m

a

signed message

aG

network

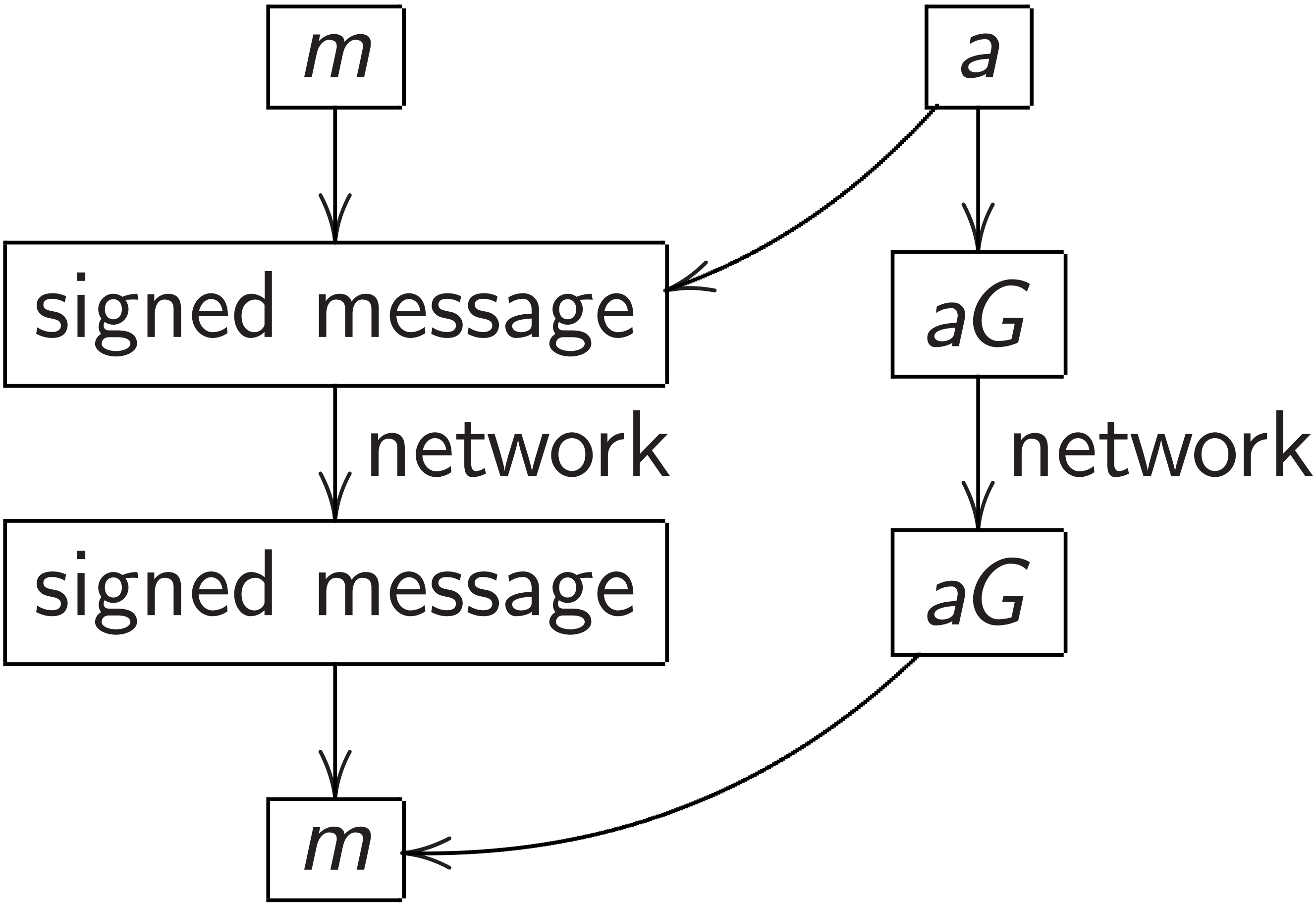
signed message

network

aG

m

Solution 2:
Public-key signatures.



Fantasy world: software for authentication/encryption/sigs is small and carefully audited ⇒ no cryptographic security failures.

Real world:

Cryptographic part of the TCB is huge. Many implementations of many cryptographic primitives.

Most complications are for speed.

Real world:

Cryptographic part of the TCB is huge. Many implementations of many cryptographic primitives.

Most complications are for speed.

e.g. February 2018: Google adds NSA's Speck cipher to Linux kernel using hand-written asm for ARM Cortex-A7 processors.

Real world:

Cryptographic part of the TCB is huge. Many implementations of many cryptographic primitives.

Most complications are for speed.

e.g. February 2018: Google adds NSA's Speck cipher to Linux kernel using hand-written asm for ARM Cortex-A7 processors.

August 2018: Google switches from Speck to ChaCha12, again using hand-written assembly. Why not ChaCha20? Speed.

Keccak (SHA-3) team maintains "Keccak Code Package" with $>$20 optimized implementations of Keccak: AVX2, NEON, etc. Includes "parallel Keccak": many further implementations.

Keccak (SHA-3) team maintains “Keccak Code Package” with $>$20 optimized implementations of Keccak: AVX2, NEON, etc. Includes “parallel Keccak”: many further implementations.

Why not portable C code using “optimizing” compiler? Slower.

Keccak (SHA-3) team maintains "Keccak Code Package" with $>$20 optimized implementations of Keccak: AVX2, NEON, etc. Includes "parallel Keccak": many further implementations.

Why not portable C code using "optimizing" compiler? Slower.

Another example: many different primitives in NIST competition for post-quantum public-key cryptography. (See next talk.)

Some overlap in implementations, but still huge volume of code.

Often people still complain about cryptographic performance.
e.g. NIST, May 2018: "we'd really like to see more platform-optimized implementations".
$\Rightarrow$ More and more software.

Often people still complain about cryptographic performance.
e.g. NIST, May 2018: “we’d really like to see more platform-optimized implementations”.
$\Rightarrow$ More and more software.

Many security failures from incorrect computations: e.g., CVE-2017-3732, CVE-2017-3736, CVE-2017-3738 in OpenSSL.

Often people still complain about cryptographic performance.
e.g. NIST, May 2018: “we’d really like to see more platform-optimized implementations”.
⇒ More and more software.

Many security failures from incorrect computations: e.g., CVE-2017-3732, CVE-2017-3736, CVE-2017-3738 in OpenSSL.

Many security failures from variable-time computations: e.g. CVE-2018-0495, CVE-2018-0737, CVE-2018-5407 in OpenSSL.

Timing attacks

Large portion of CPU hardware:
optimizations depending on
addresses of memory locations.

Consider data caching,
instruction caching,
parallel cache banks,
store-to-load forwarding,
branch prediction, etc.

Timing attacks

Large portion of CPU hardware: optimizations depending on addresses of memory locations.

Consider data caching,
instruction caching,
parallel cache banks,
store-to-load forwarding,
branch prediction, etc.

Many attacks (e.g. TLBleed from 2018 Gras–Razavi–Bos–Giuffrida) show that this portion of the CPU has trouble keeping secrets.

Typical literature on this topic:

Understand this portion of CPU.
But details are often proprietary,
not exposed to security review.

Try to push attacks further.
This becomes very complicated.

Tweak the attacked software
to try to stop the known attacks.

Typical literature on this topic:

Understand this portion of CPU.
But details are often proprietary,
not exposed to security review.

Try to push attacks further.
This becomes very complicated.

Tweak the attacked software
to try to stop the known attacks.

For researchers: This is great!

Typical literature on this topic:

Understand this portion of CPU.
But details are often proprietary,
not exposed to security review.

Try to push attacks further.
This becomes very complicated.

Tweak the attacked software
to try to stop the known attacks.

For researchers: This is great!

For auditors: This is a nightmare.
Many years of security failures.
No confidence in future security.

The “constant-time” solution:
Don’t give any secrets
to this portion of the CPU.
(1987 Goldreich, 1990 Ostrovsky:
Oblivious RAM; 2004 Bernstein:
domain-specific for better speed)

The “constant-time” solution:
Don’t give any secrets
to this portion of the CPU.
(1987 Goldreich, 1990 Ostrovsky:
Oblivious RAM; 2004 Bernstein:
domain-specific for better speed)

TCB analysis: Need this portion
of the CPU to be correct, but
don’t need it to keep secrets.
Makes auditing much easier.

The “constant-time” solution:
Don’t give any secrets
to this portion of the CPU.
(1987 Goldreich, 1990 Ostrovsky:
Oblivious RAM; 2004 Bernstein:
domain-specific for better speed)

TCB analysis: Need this portion
of the CPU to be correct, but
don’t need it to keep secrets.
Makes auditing much easier.

Good match for attitude and
experience of CPU designers: e.g.,
Intel issues errata for correctness
bugs, not for information leaks.

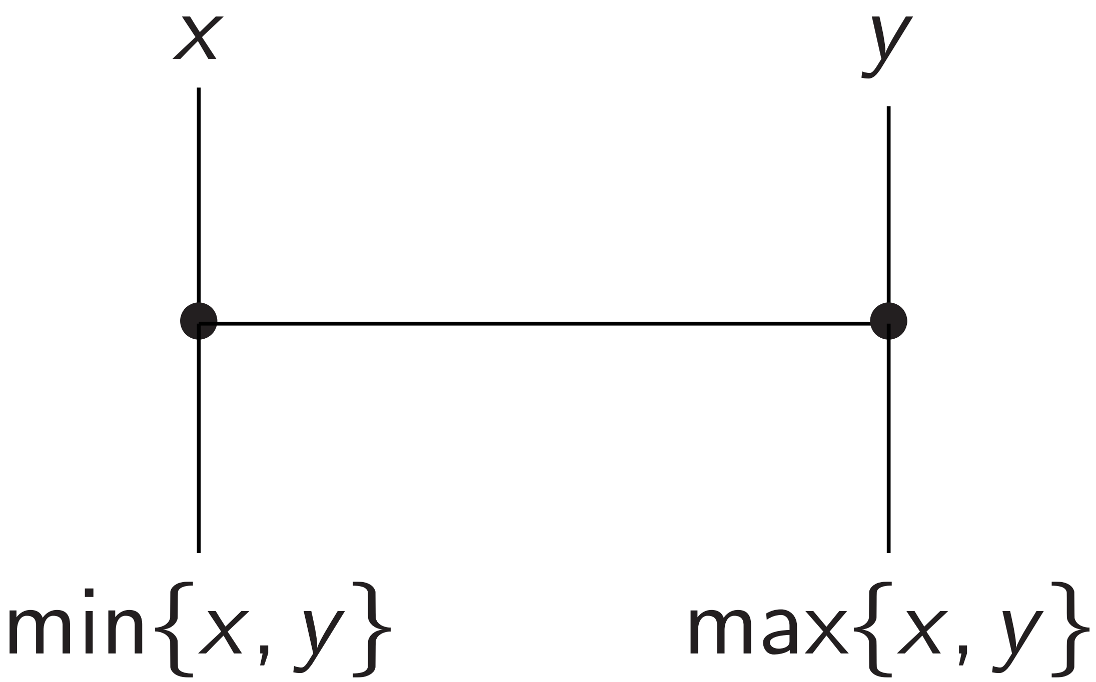
Case study: Constant-time sorting

Subroutine in (e.g.) BIG QUAKE, Classic McEliece, GeMSS, Gravity-SPHINCS, LEDAkem, LEDApkc, NTRU Prime, Round2: sort array of secret integers. e.g. sort 768 32-bit integers.

Typical sorting algorithms— merge sort, quicksort, etc.— choose load/store addresses based on secret data. Usually also branch based on secret data.

How to sort secret data without any secret addresses?

Foundation of solution:
a **comparator** sorting 2 integers.

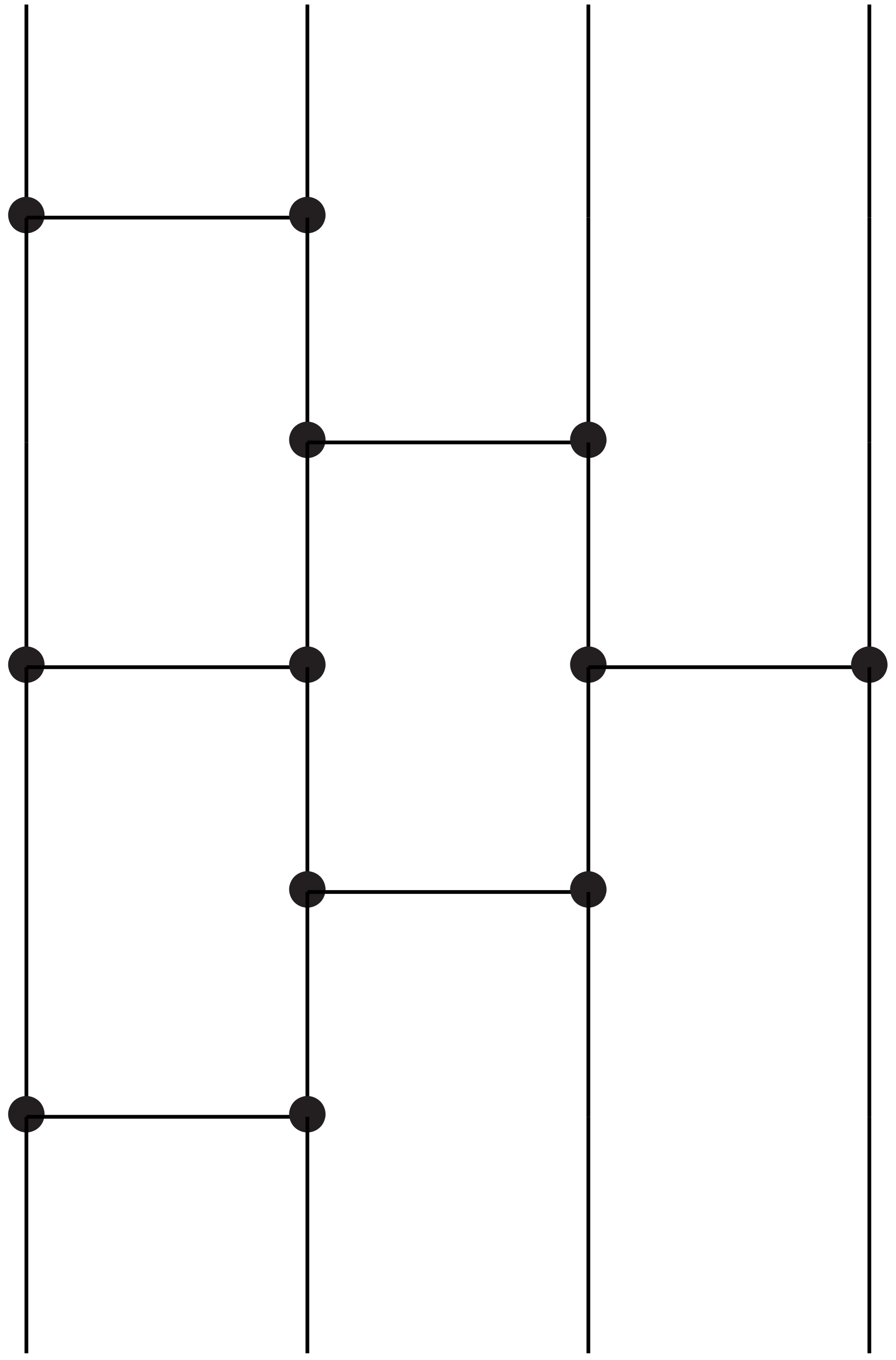


Easy constant-time exercise in C.
Warning: C standard allows
compiler to break the solution.

Even easier exercise in asm.

Combine comparators into a **sorting network** for more inputs.

Example of a sorting network:



Positions of comparators
in a sorting network are
independent of the input.
Naturally constant-time.

Positions of comparators
in a sorting network are
independent of the input.
Naturally constant-time.

But remember all the people
complaining about speed: e.g.,
"We would be happy to hear that
fixed weight sampling is efficient
on a variety of platforms . . .
We have not yet been convinced
that this is the case."

Positions of comparators in a sorting network are independent of the input. Naturally constant-time.

But remember all the people complaining about speed: e.g., "We would be happy to hear that fixed weight sampling is efficient on a variety of platforms . . . We have not yet been convinced that this is the case."

$(n^2 - n)/2$ comparators in bubble sort produce complaints about performance as n increases.

```
void int32_sort(int32 *x,int64 n)
{ int64 t,p,q,i;
  if (n < 2) return;
  t = 1;
  while (t < n - t) t += t;
  for (p = t;p > 0;p >>= 1) {
    for (i = 0;i < n - p;++i)
      if (!(i & p))
        minmax(x+i,x+i+p);
    for (q = t;q > p;q >>= 1)
      for (i = 0;i < n - q;++i)
        if (!(i & p))
          minmax(x+i+p,x+i+q);
  }
}
```

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

$\approx n(\log_2 n)^2/4$ comparators.
Much faster than bubble sort.

Warning: many other descriptions of Batcher’s sorting networks require n to be a power of 2.
Also, Wikipedia says “Sorting networks . . . are not capable of handling arbitrarily large inputs.”

This constant-time sorting code

vectorization
(for Haswell)

Constant-time sorting code
included in 2017
Bernstein–Chuengsatiansup–
Lange–van Vredendaal
“NTRU Prime” software release

revamped for
higher speed

New: “djbsort”
constant-time sorting code

The slowdown for constant time

Massive fast-sorting literature.
Includes several efforts to optimize sorting using AVX2 instructions on modern Intel CPUs: e.g. 2015 Gueron–Krasnov quicksort.

Haswell (`titan0`) cycles, $n = 768$:
25608 `stdsort`
21844 `herf`

15136 `krasnov`

The slowdown for constant time

Massive fast-sorting literature.
Includes several efforts to optimize sorting using AVX2 instructions on modern Intel CPUs: e.g. 2015 Gueron–Krasnov quicksort.

Haswell (`titan0`) cycles, $n = 768$:
25608 `stdsort`
21844 `herf`
18548 `oldavx2` (2017 BCLvV)
15136 `krasnov`

The slowdown for constant time

Massive fast-sorting literature. Includes several efforts to optimize sorting using AVX2 instructions on modern Intel CPUs: e.g. 2015 Gueron–Krasnov quicksort.

Haswell (`titan0`) cycles, $n = 768$:

25608 `stdsort`

21844 `herf`

18548 `oldavx2` (2017 BCLvV)

15136 `krasnov`

6596 `avx2` (2018 djbsort)

The slowdown for constant time

Massive fast-sorting literature.
Includes several efforts to optimize sorting using AVX2 instructions on modern Intel CPUs: e.g. 2015 Gueron–Krasnov quicksort.

Haswell (`titan0`) cycles, $n = 768$:
25608 `stdsort`
21844 `herf`
18548 `oldavx2` (2017 BCLvV)
15136 `krasnov`
6596 `avx2` (2018 djbsort)

No slowdown. New speed records!

How can an $n(\log n)^2$ algorithm beat standard $n \log n$ algorithms?

How can an $n(\log n)^2$ algorithm beat standard $n \log n$ algorithms?

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

How can an $n(\log n)^2$ algorithm beat standard $n \log n$ algorithms?

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

Every cycle, Haswell core can do 8 “min” ops on 32-bit integers + 8 “max” ops on 32-bit integers.

How can an $n(\log n)^2$ algorithm beat standard $n \log n$ algorithms?

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

Every cycle, Haswell core can do 8 "min" ops on 32-bit integers + 8 "max" ops on 32-bit integers.

Loading a 32-bit integer from a random address: much slower.

Conditional branch: much slower.

Verification

Sorting software is in the TCB.
Does it work correctly?

Test the sorting software on many random inputs, increasing inputs, decreasing inputs. Seems to work.

Verification

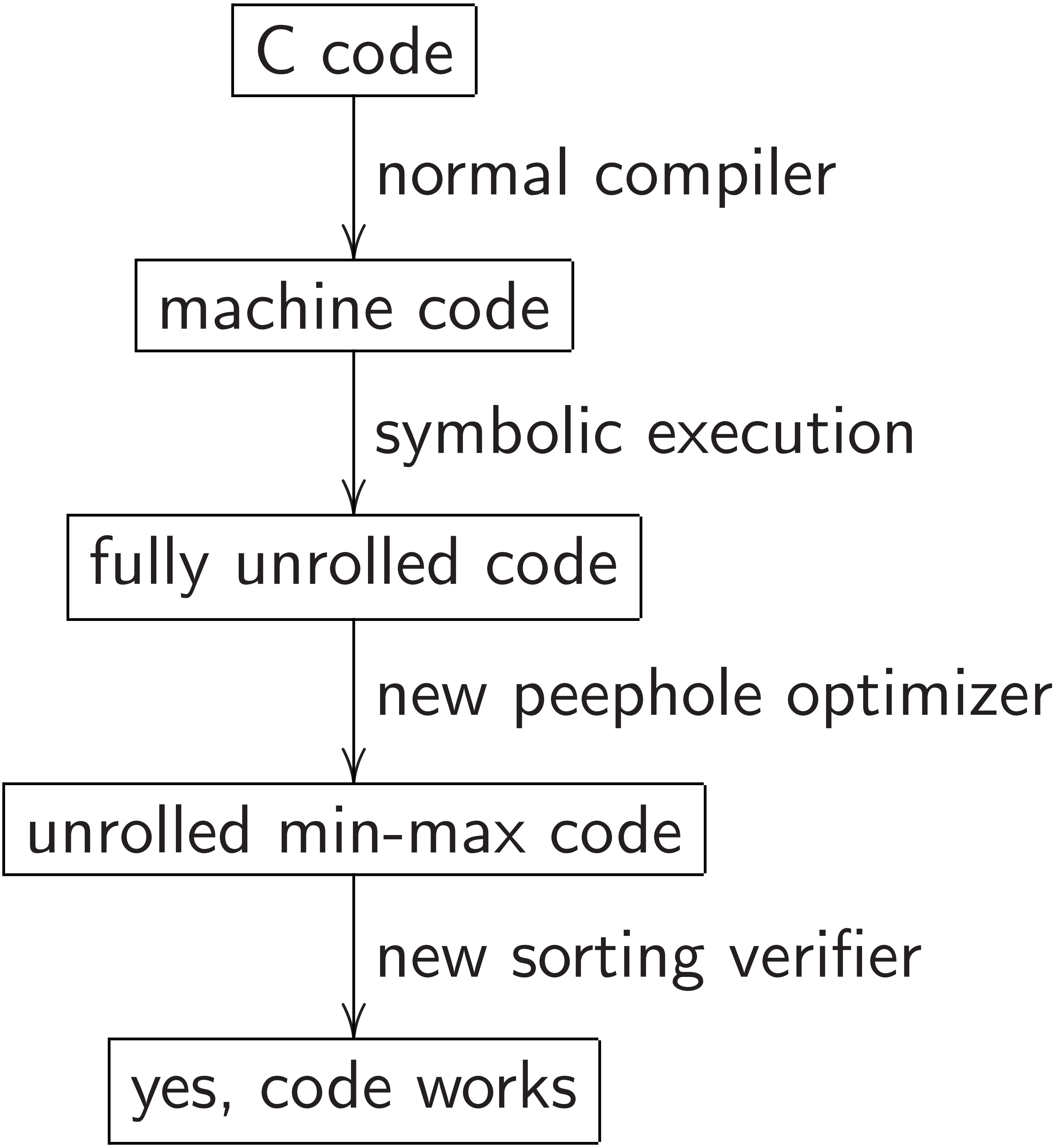
Sorting software is in the TCB.
Does it work correctly?

Test the sorting software on many random inputs, increasing inputs, decreasing inputs. Seems to work.

But are there *occasional* inputs where this sorting software fails to sort correctly?

History: Many security problems involve occasional inputs where TCB works incorrectly.

For each used n (e.g., 768):



C code

normal compiler

machine code

symbolic execution

fully unrolled code

new peephole optimizer

unrolled min-max code

new sorting verifier

yes, code works

Symbolic execution:
use existing “angr” library,
with tiny new patches for
eliminating byte splitting, adding
a few missing vector instructions.

Symbolic execution:
use existing “angr” library,
with tiny new patches for
eliminating byte splitting, adding
a few missing vector instructions.

Peephole optimizer:
recognize instruction patterns
equivalent to min, max.

Symbolic execution:
use existing “angr” library,
with tiny new patches for
eliminating byte splitting, adding
a few missing vector instructions.

Peephole optimizer:
recognize instruction patterns
equivalent to min, max.

Sorting verifier: decompose
DAG into merging networks.
Verify each merging network
using generalization of 2007
Even–Levi–Litman, correction of
1990 Chung–Ravikumar.

Current djbsort release,
verified AVX2 code and
verified portable code:

`https://sorting.cr.yp.to`

Includes the sorting code;
automatic build-time tests;
simple benchmarking program;
verification tools.

Web site shows how to
use the verification tools.

Next release planned:
verified ARM NEON code.

The future

I don't think there is a fundamental tension between
- crypto performance,
- stopping timing attacks,
- making sure software works.

See the sorting example.

Firefox has already deployed verified constant-time software for Curve25519+ChaCha20+Poly1305.

I'm working on easier verification, post-quantum code, faster code.