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.

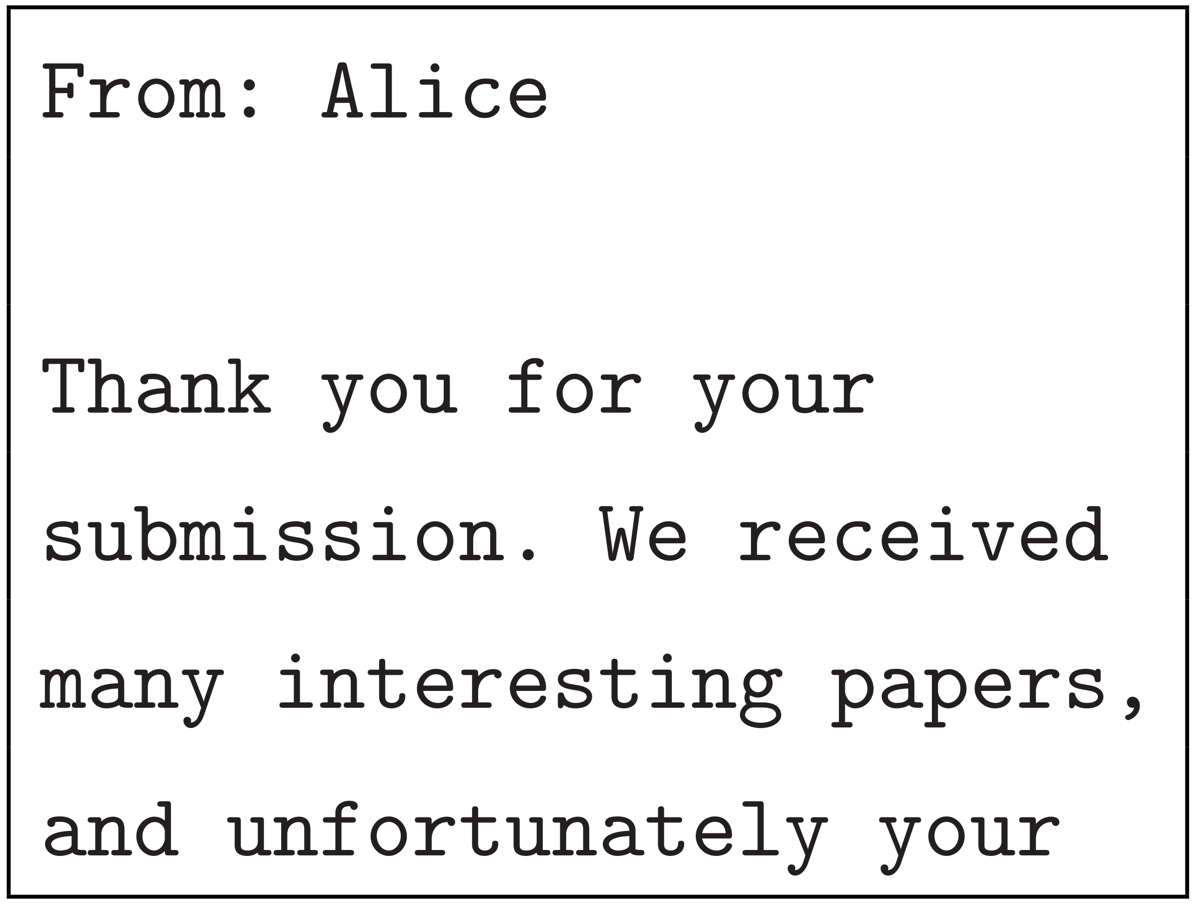
But today's "security" systems fail to guarantee this property. Attacker could have modified or forged the message.

ptographic software

?

ernstein

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

e.g. Fire

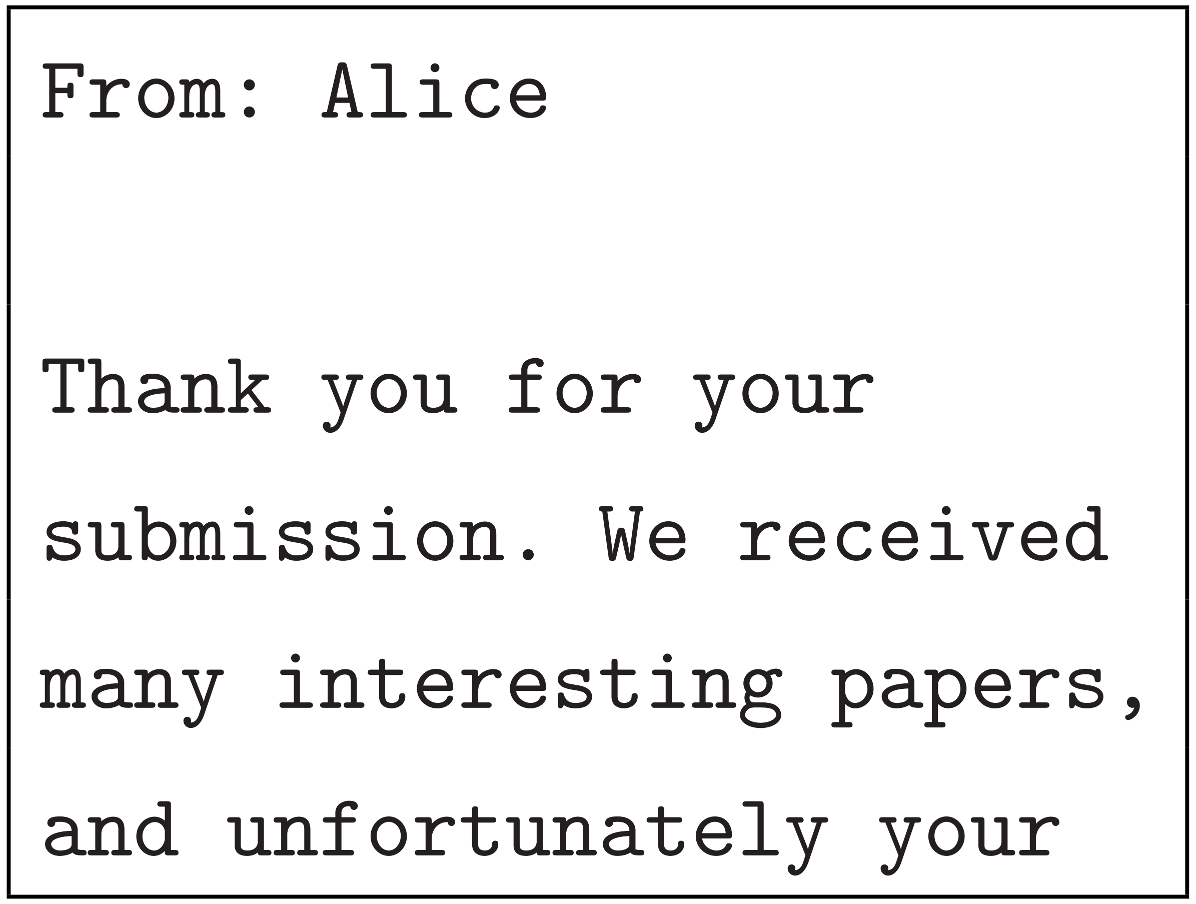
4582680

3093398

2623454

: software

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 c

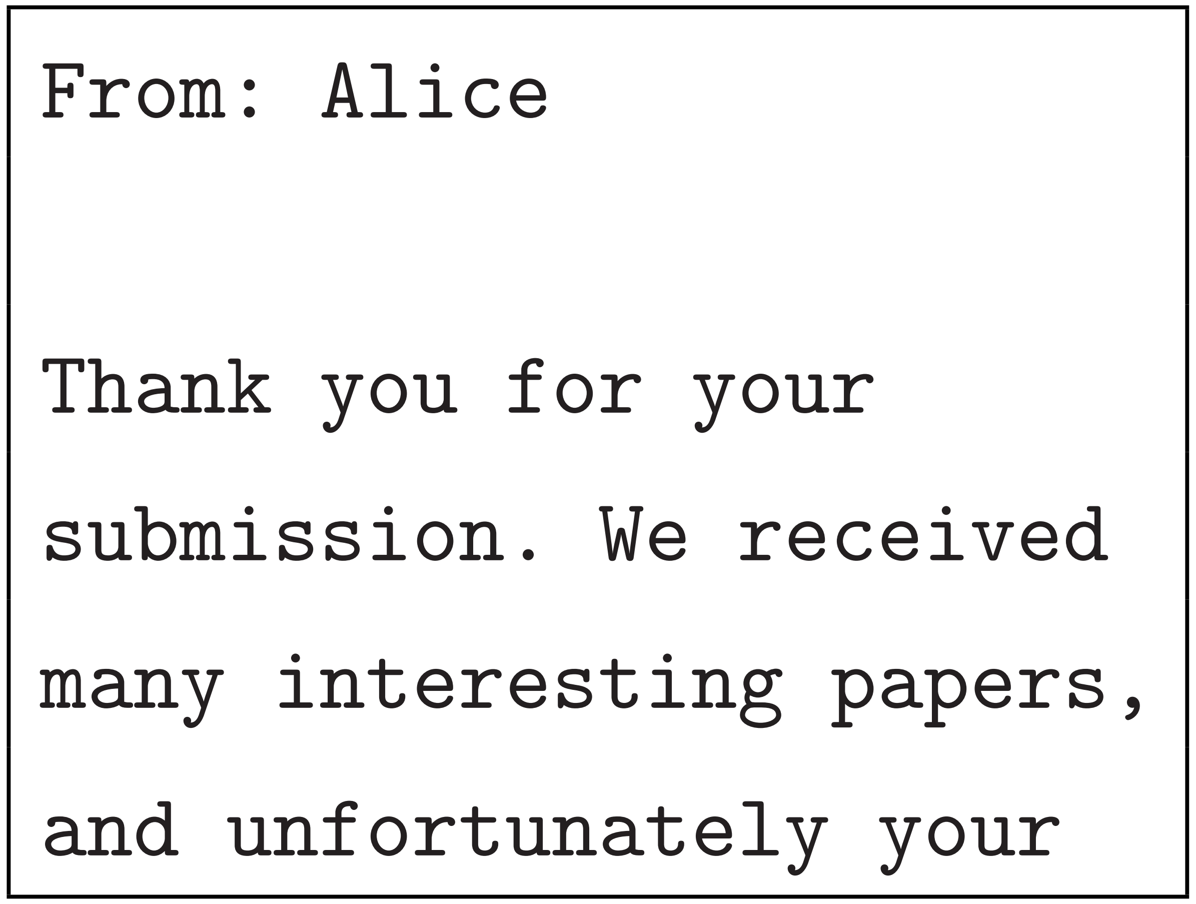
e.g. Firefox 60 (M

4582680 lines in c

3093398 lines in h

2623454 lines in c

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.

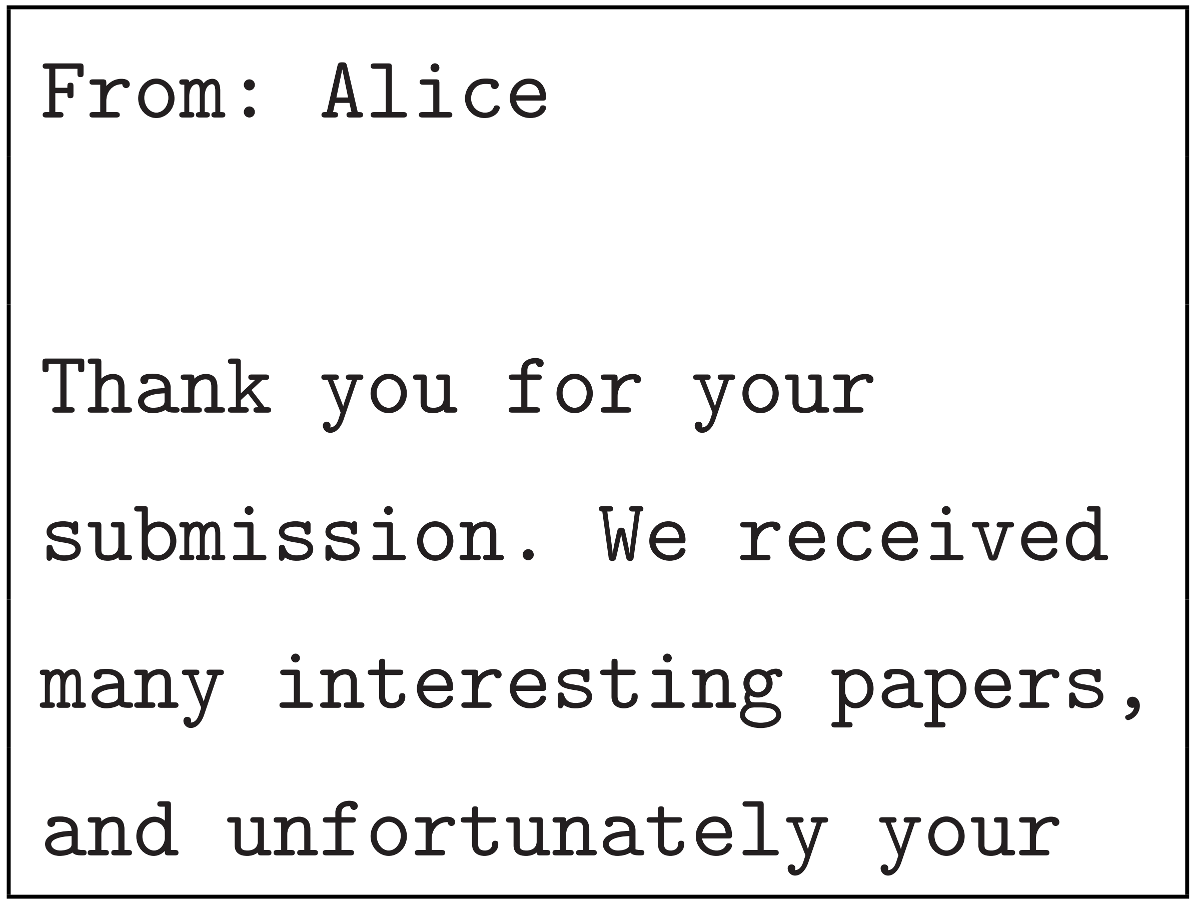
e.g. Firefox 60 (May 2018) c

4582680 lines in cpp files,

3093398 lines in `h` files,

2623454 lines in c files, etc.

Bob's laptop screen:



```
From: Alice


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

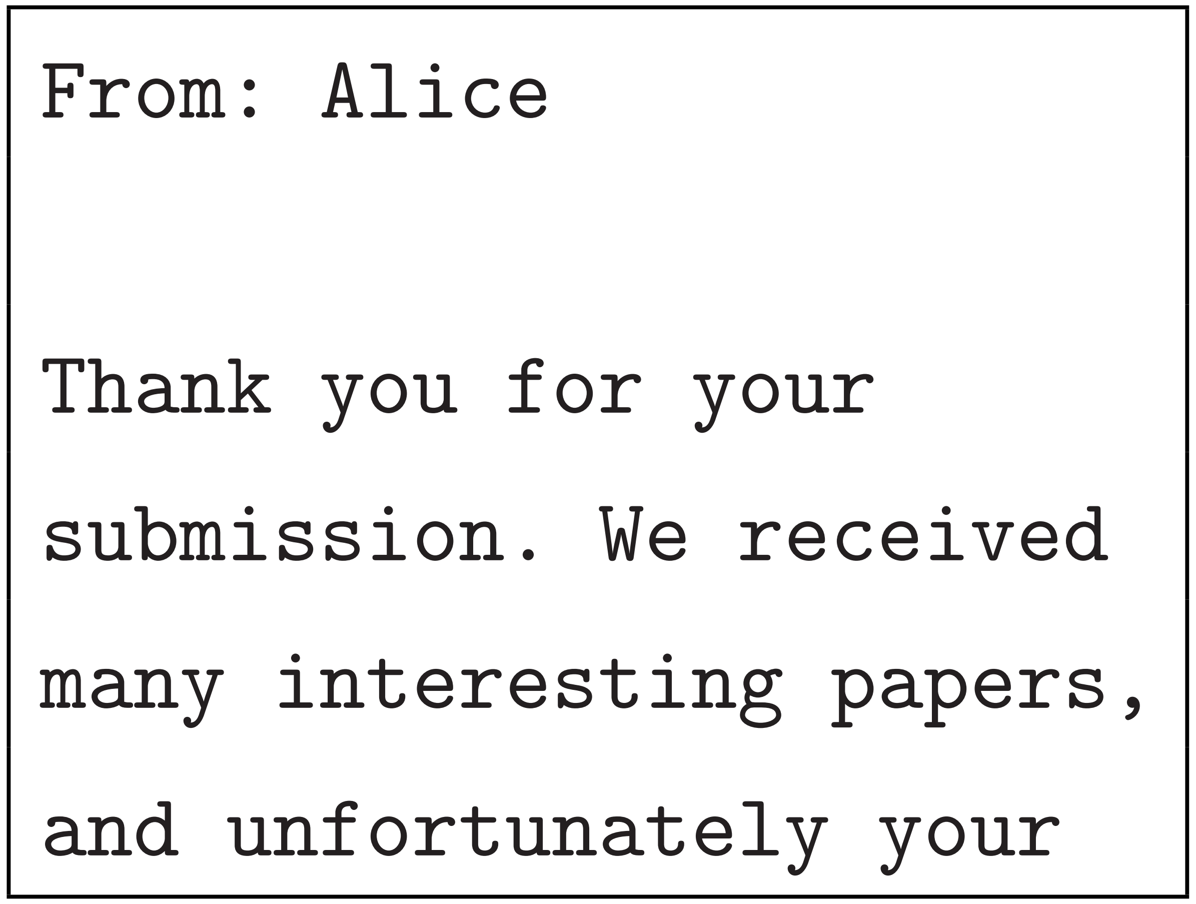
Bob assumes this message is something Alice actually sent.

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.

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.

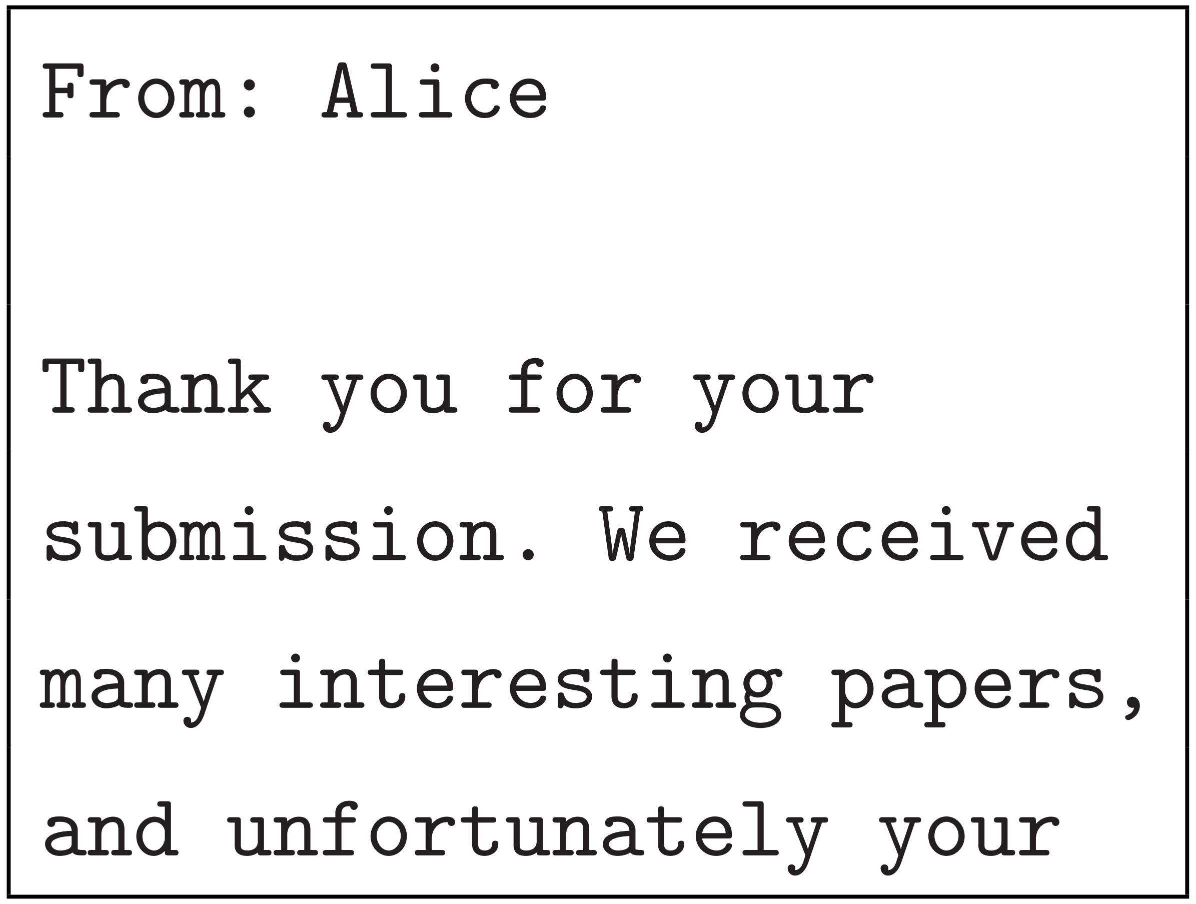
But today’s “security” systems fail to guarantee this property. Attacker could have modified or forged the message.

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.

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

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.

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".

ptop screen:

```
: Alice

k you for your
ission. We received
 interesting papers,
unfortunately your
```

umes this message is
ng Alice actually sent.

ay's “security” systems
uarantee this property.
could have modified
d 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.

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

TCB: po
that is r
the users

en:

```
r your
e received
ing papers,
tely your
```

message is
ctually sent.

rity" systems
his property.
ve modified
sage.

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 computin

TCB: portion of c
that is responsible
the users' security

ed

rs,

r

s

t.

ms

ty.

d

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 (TC

TCB: portion of computer s
that is responsible for enforc
the users' security policy.

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.

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.

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

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.

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.

are too complex.
fox 60 (May 2018) code:
lines in cpp files,
lines in h files,
lines in c files, etc.

ne in this code has
rol over user messages.

vulnerabilities fixed in 61:
18-12359, "Buffer
using computed size
s element"; CVE-2018-
'Use-after-free when
cus()"; CVE-2018-12361,
overflow in SwizzleData".

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.

Example

1. Attac
in a c
Linux

omplex.
ay 2018) code:
pp files,
. files,
 files, etc.

code has
ser messages.

ties fixed in 61:
 "Buffer
nputed size
"; CVE-2018-
-free when
VE-2018-12361,
in SwizzleData".

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 attac

1. Attacker uses b
   in a device driv
   Linux kernel on

code:

ges.

in 61:

e

018-

2361,

Data".

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 strategie

1. Attacker uses buffer over
   in a device driver to cont
   Linux kernel on Alice's la

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.

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

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.
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.

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.

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?

computing base (TCB)

rtion of computer system
esponsible for enforcing
s' security policy.

policy for this talk:
ge is displayed on
reen as "`From: Alice`"
ssage is from Alice.

works correctly,
ssage is guaranteed
om Alice, no matter what
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.
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 s

Rearchit
to have

g base (TCB)

omputer system
for enforcing
policy.

this talk:
ayed on
From: Alice”
om Alice.

ectly,
uaranteed
no matter what
tem does.

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 str

Rearchitect compu
to have a much sr

CB)

ystem
cing

ce”

what

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 systen
to have a much smaller TCE

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.

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.

Carefully audit the TCB.

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.

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.

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.

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.

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.

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.

es of attack strategies:

cker uses buffer overflow
device driver to control
kernel on Alice's laptop.

cker uses buffer overflow
web browser to control
files on Bob's laptop.

driver is in the TCB.
wser is in the TCB.
n the TCB. Etc.

TCB has many bugs,
g many security holes.
pe of fixing this?

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

How doe
that inco
is from A

Cryptogr
Message

Alic

authent

authent

Alic

k strategies:

buffer overflow
er to control
Alice's laptop.

buffer overflow
er to control
b's laptop.

the TCB.
the TCB.
B. Etc.

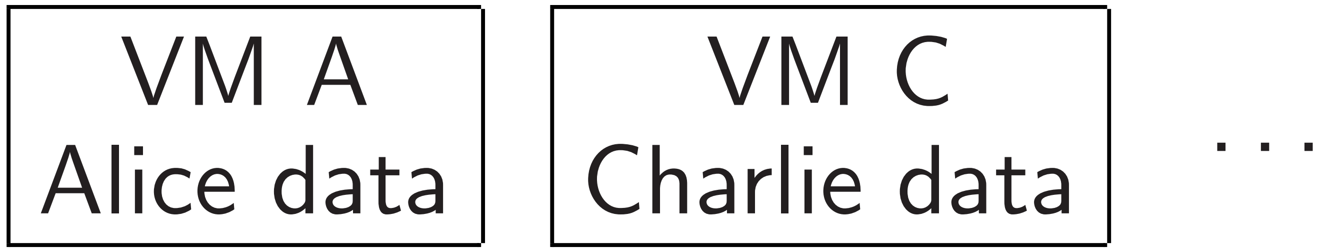
many bugs,
curity holes.
this?

Classic security strategy:

Rearchitect computer systems
to have a much smaller TCB.

Carefully audit the TCB.

e.g. Bob runs many VMs:



TCB stops each VM from
touching data in other VMs.

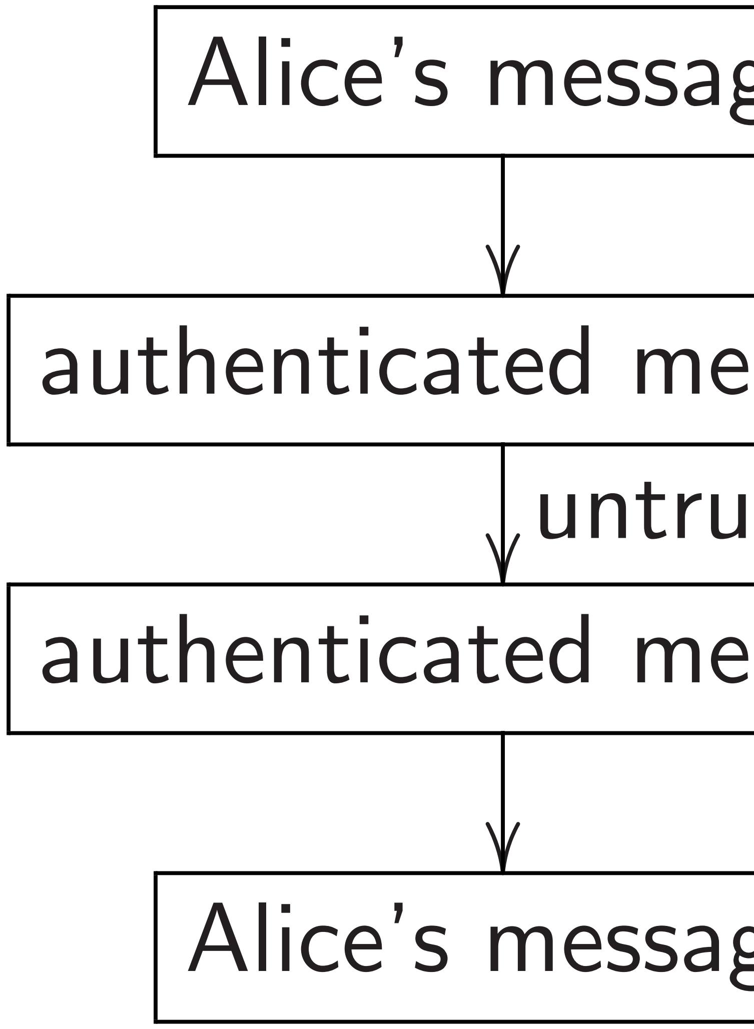
Browser in VM C isn't in TCB.
Can't touch data in VM A,
if TCB works correctly.

Alice also runs many VMs.

Focus of this talk:

How does Bob's la
that incoming netw
is from Alice's lapt

Cryptographic solu
Message-authentic



es:

flow
rol
ptop.

flow
rol
.

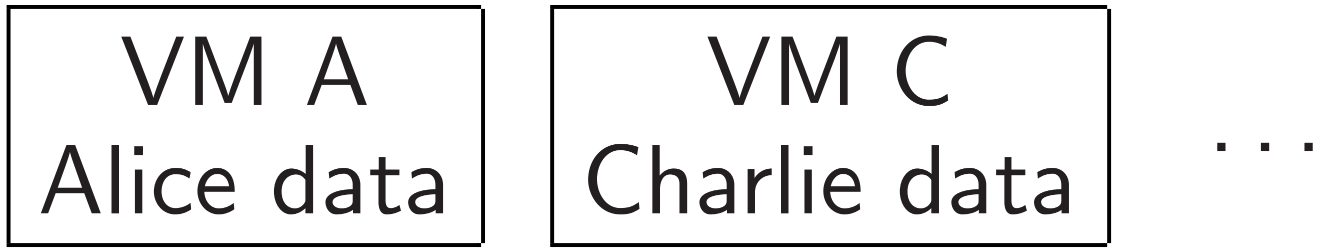
s,
es.

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.

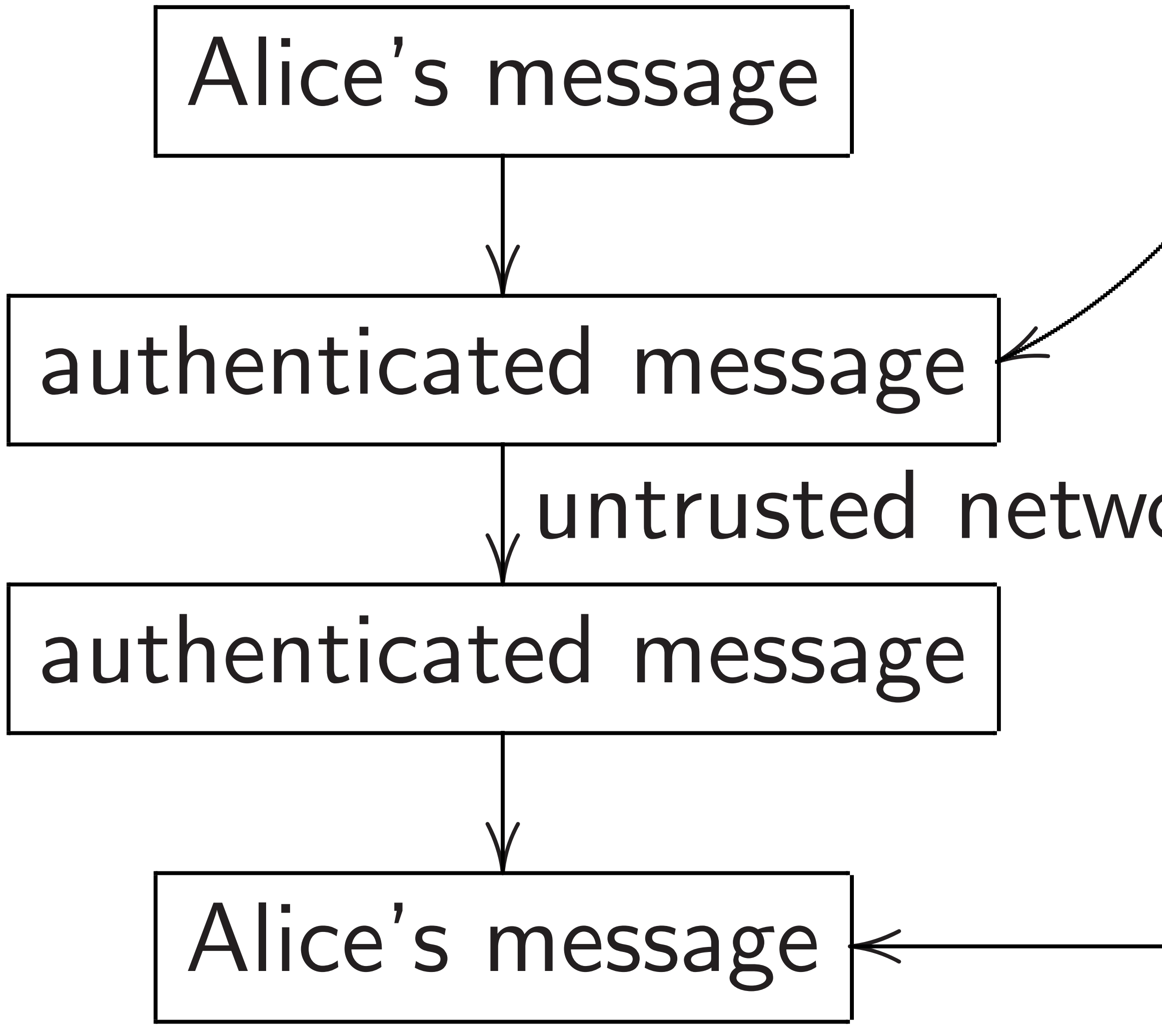
Browser in VM C isn't in TCB. Can't touch data in VM A, if TCB works correctly.

Alice also runs many VMs.

Focus of this talk: Cryptogr

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

Cryptographic solution: Message-authentication code


Alice's message
authenticated message
untrusted netwo
authenticated message
Alice's message

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.

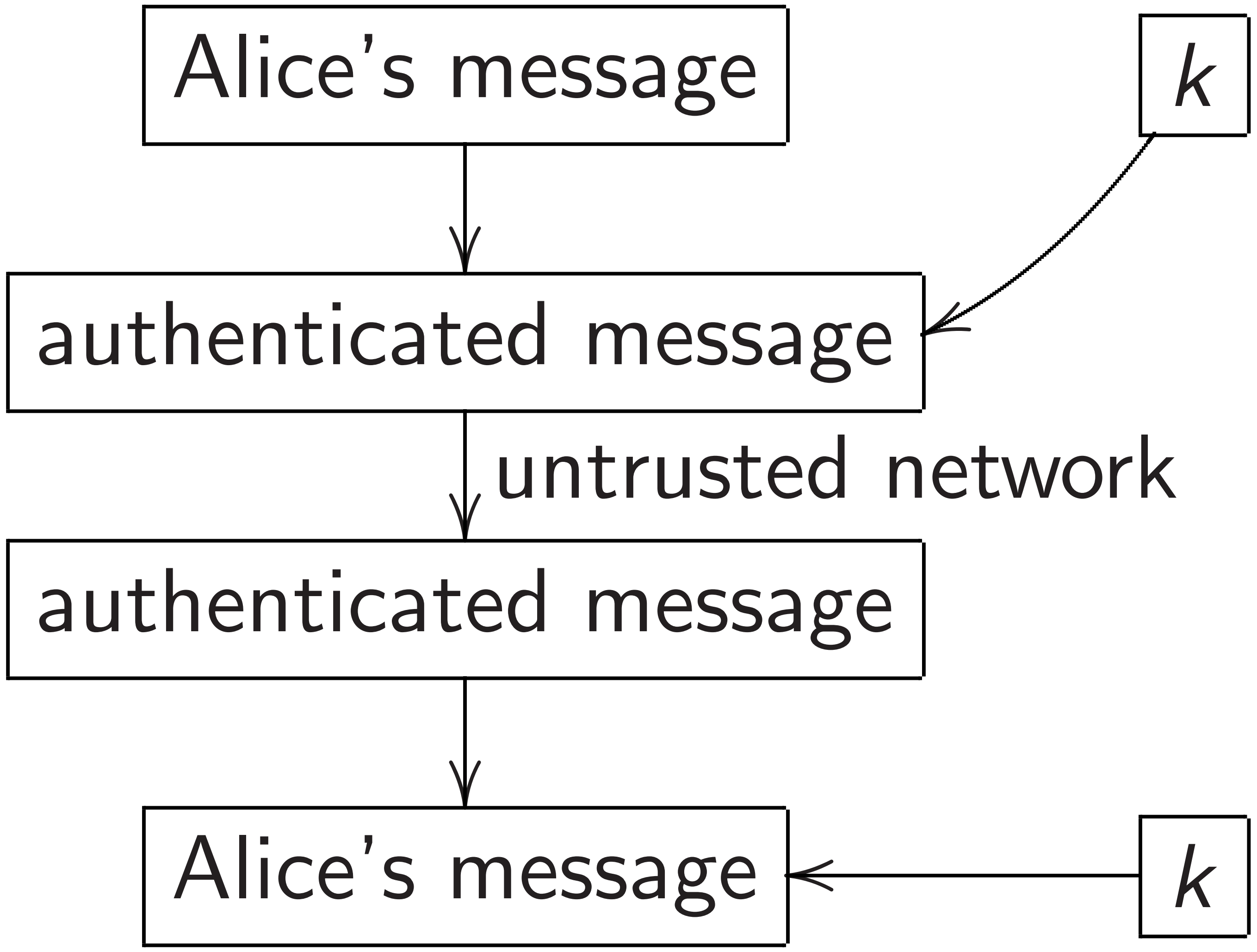
Browser in VM C isn't in TCB. Can't touch data in VM A, if TCB works correctly.

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

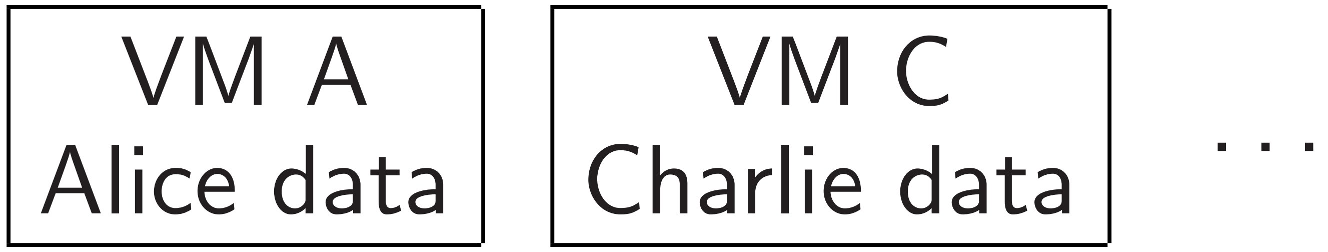
k

Classic security strategy:

Rearchitect computer systems to have a much smaller TCB.

Carefully audit the TCB.

e.g. Bob runs many VMs:



TCB stops each VM from touching data in other VMs.

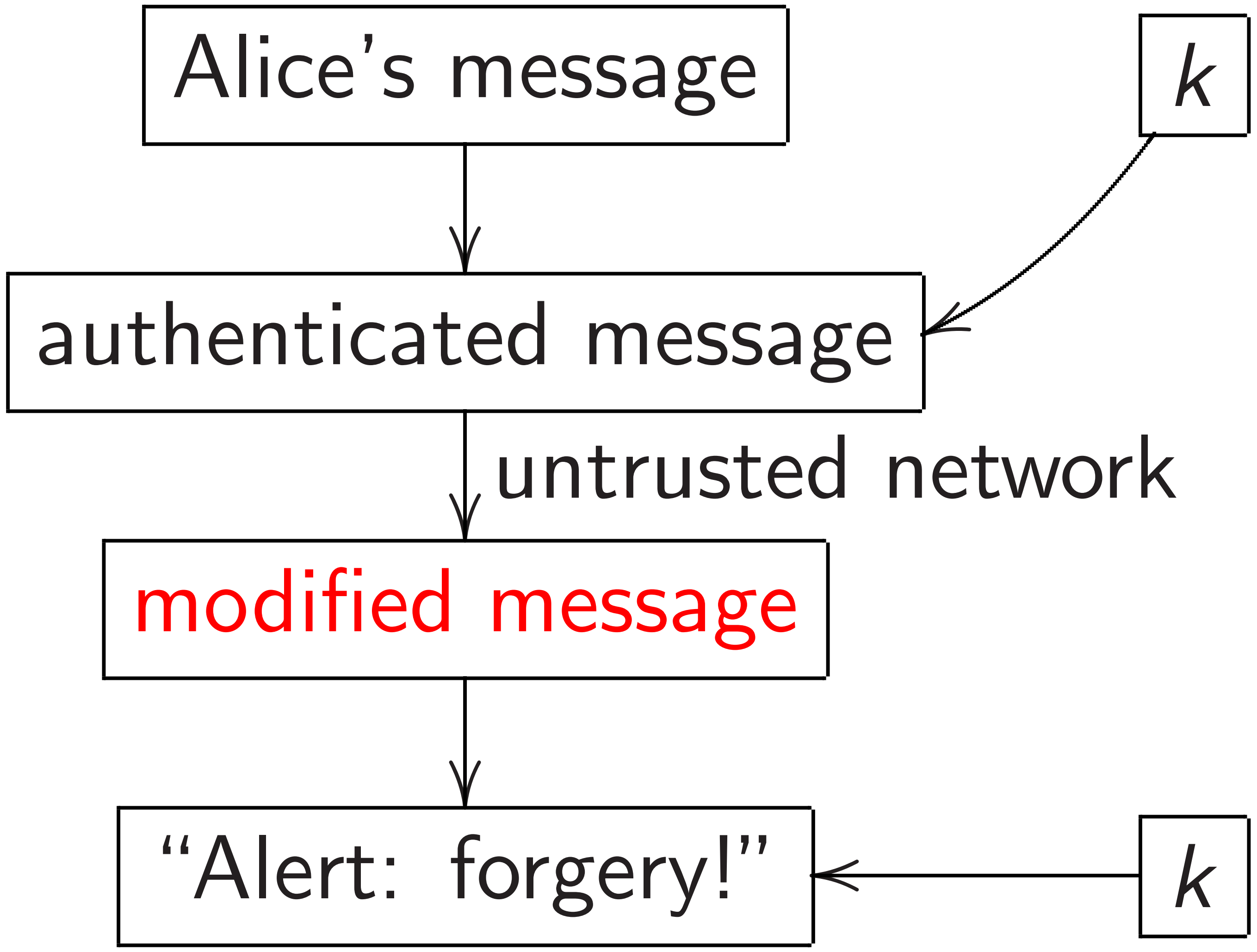
Browser in VM C isn't in TCB. Can't touch data in VM A, if TCB works correctly.

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.



security strategy:

ect computer systems
a much smaller TCB.

audit the TCB.

runs many VMs:

A
ata
VM C
Charlie data
. . .

ps each VM from
data in other VMs.

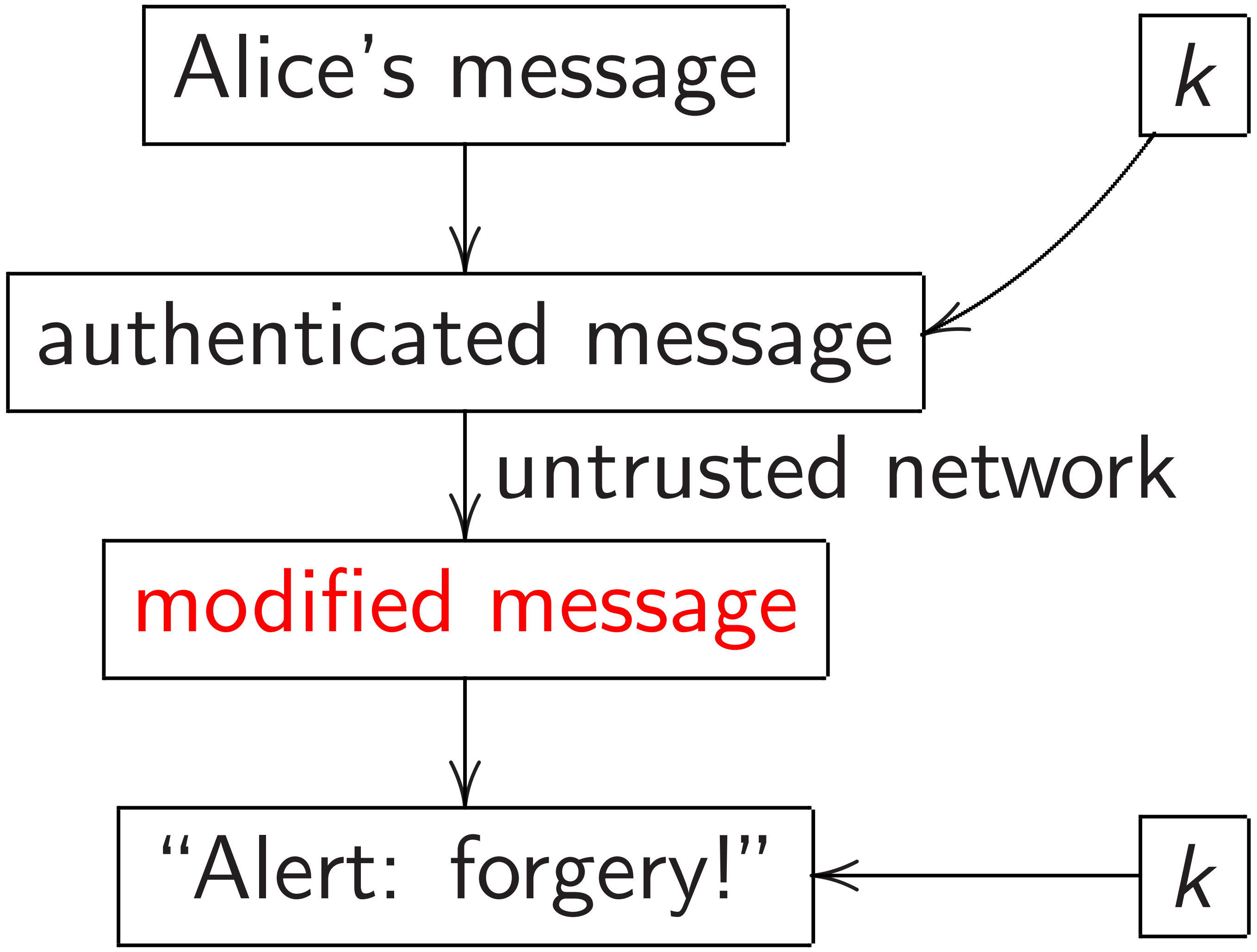
in VM C isn't in TCB.
uch data in VM A,
works correctly.

o 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.



Importa
to share

What if
on their

rategy:

uter systems
maller TCB.

e TCB.

ny VMs:

VM C
rlie data

. . .

VM from
other VMs.

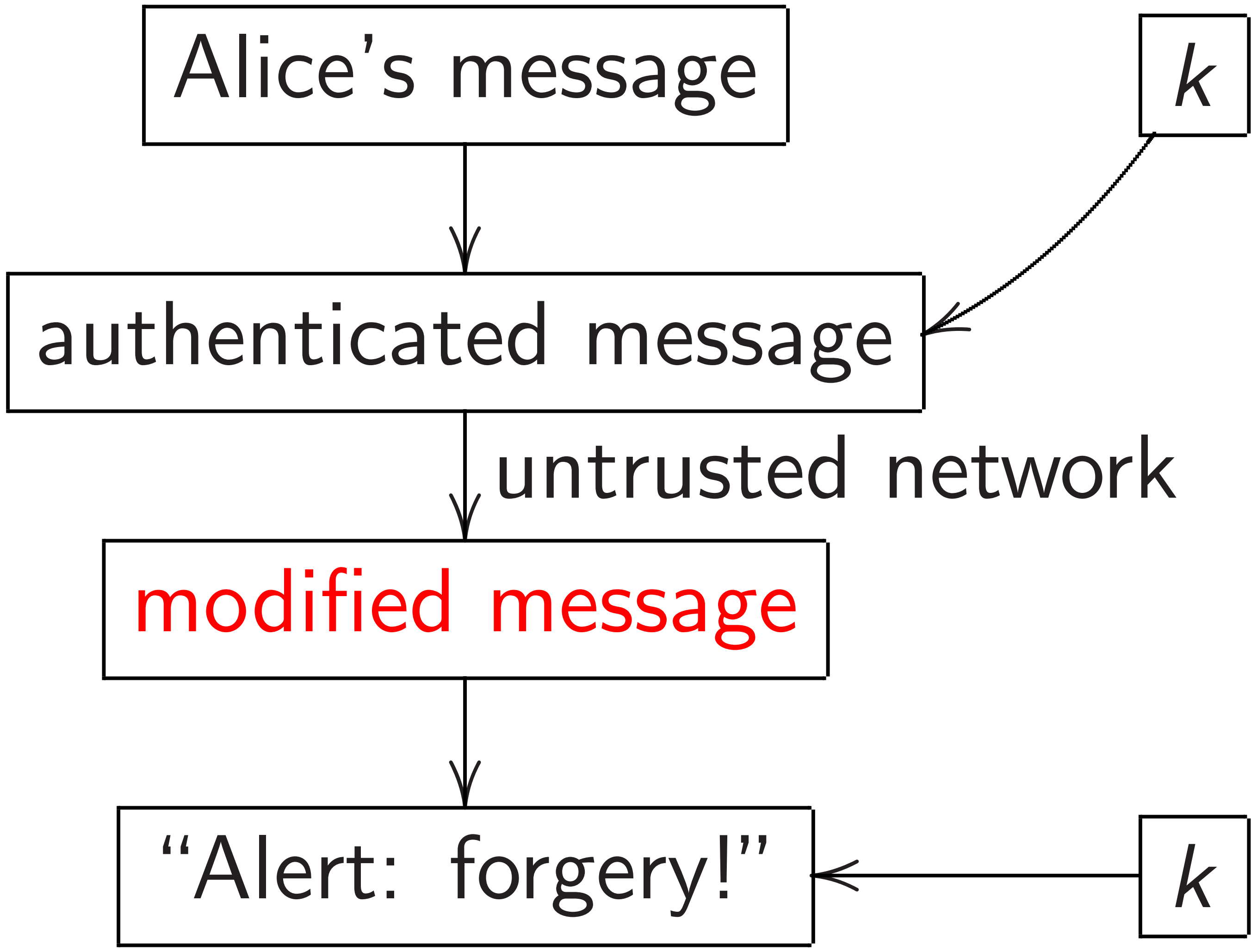
isn't in TCB.
in VM A,
ectly.

ny 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.



Important for Alic
to share the same

What if attacker w
on their communic

ns

B.

. . .

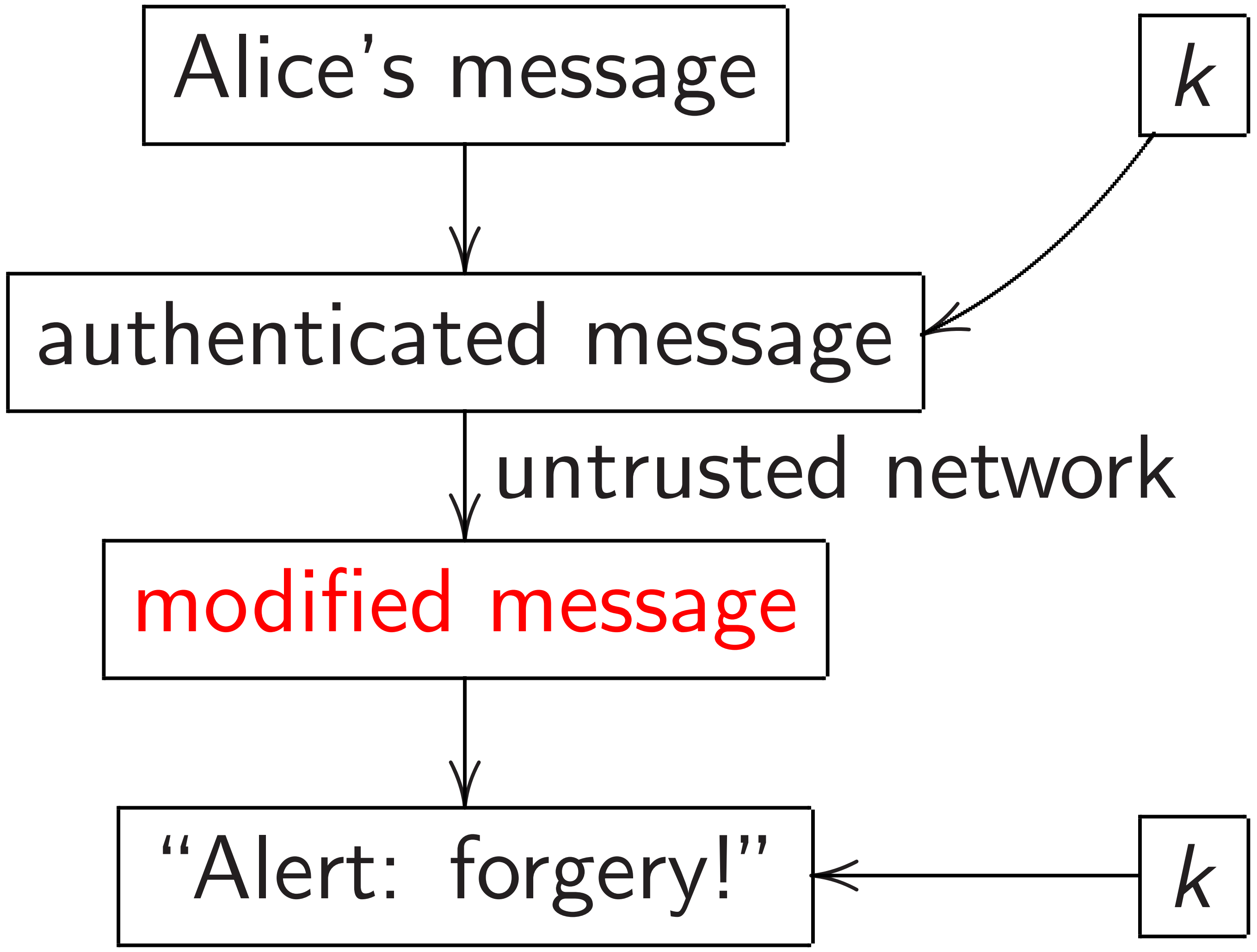
.

CB.

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.



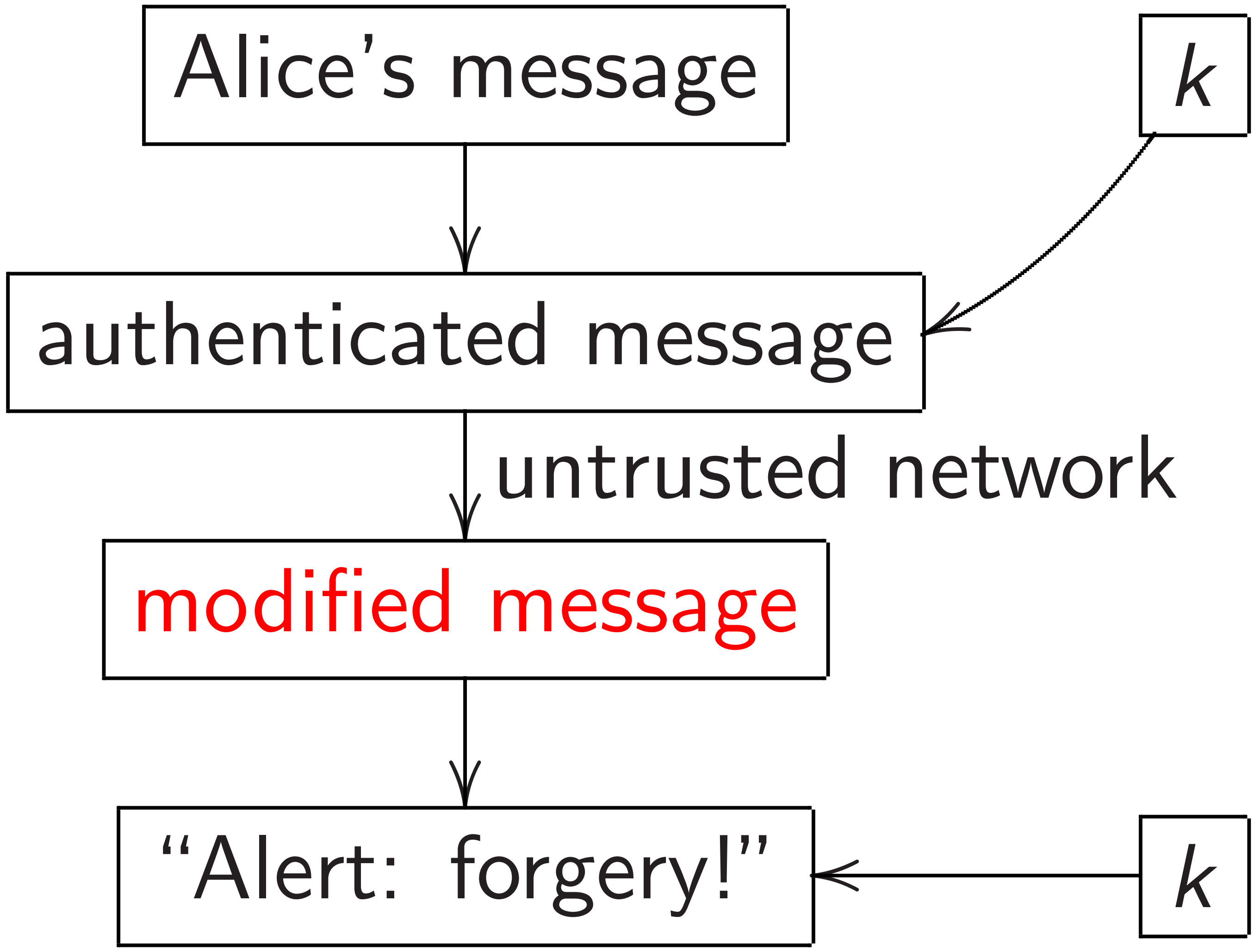
Important for Alice and Bob to share the same secret k.

What if attacker was spying on their communication of 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.



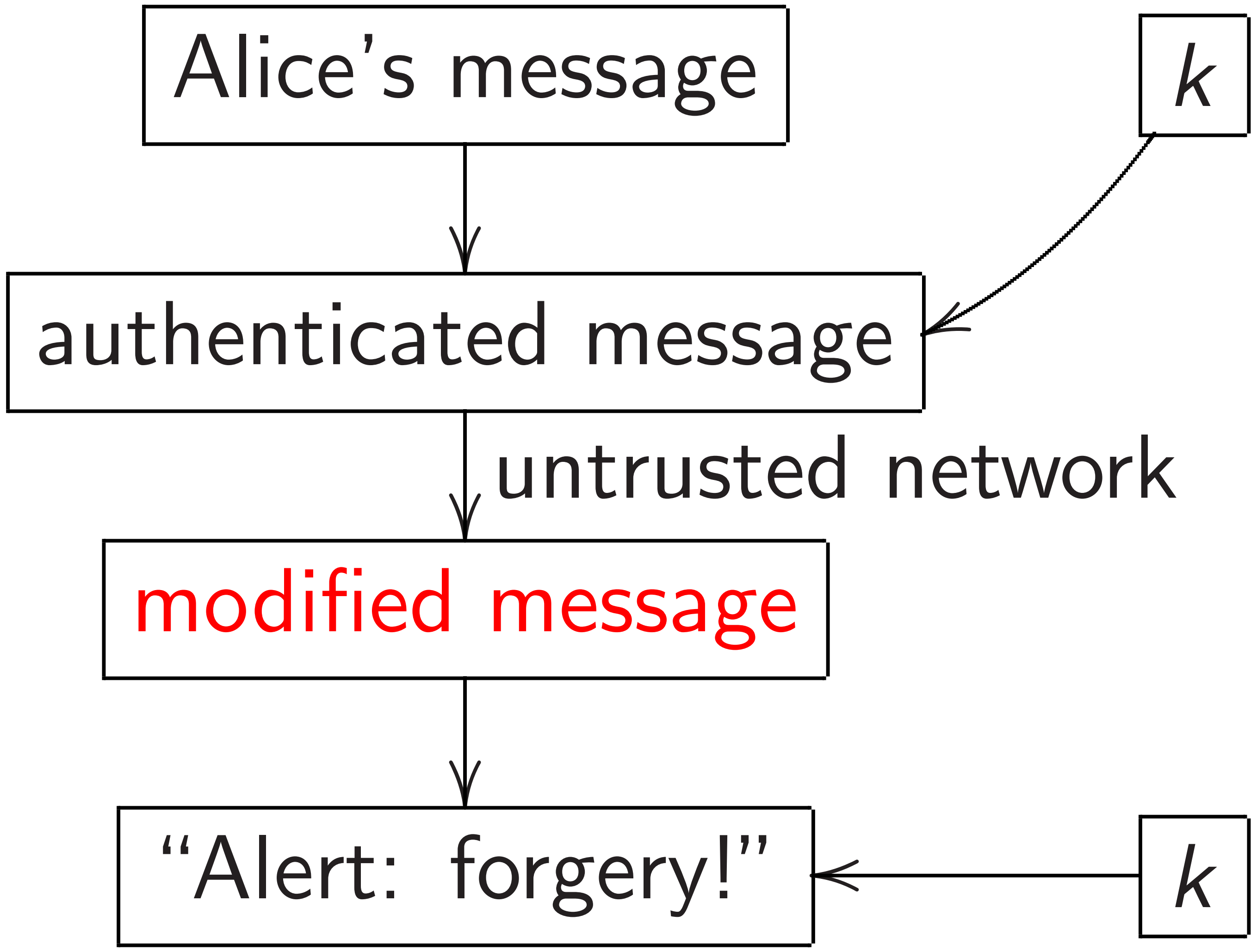
Important for Alice and Bob to share the same secret k.

What if attacker was spying on their communication of k?

Focus of this talk: Cryptography

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

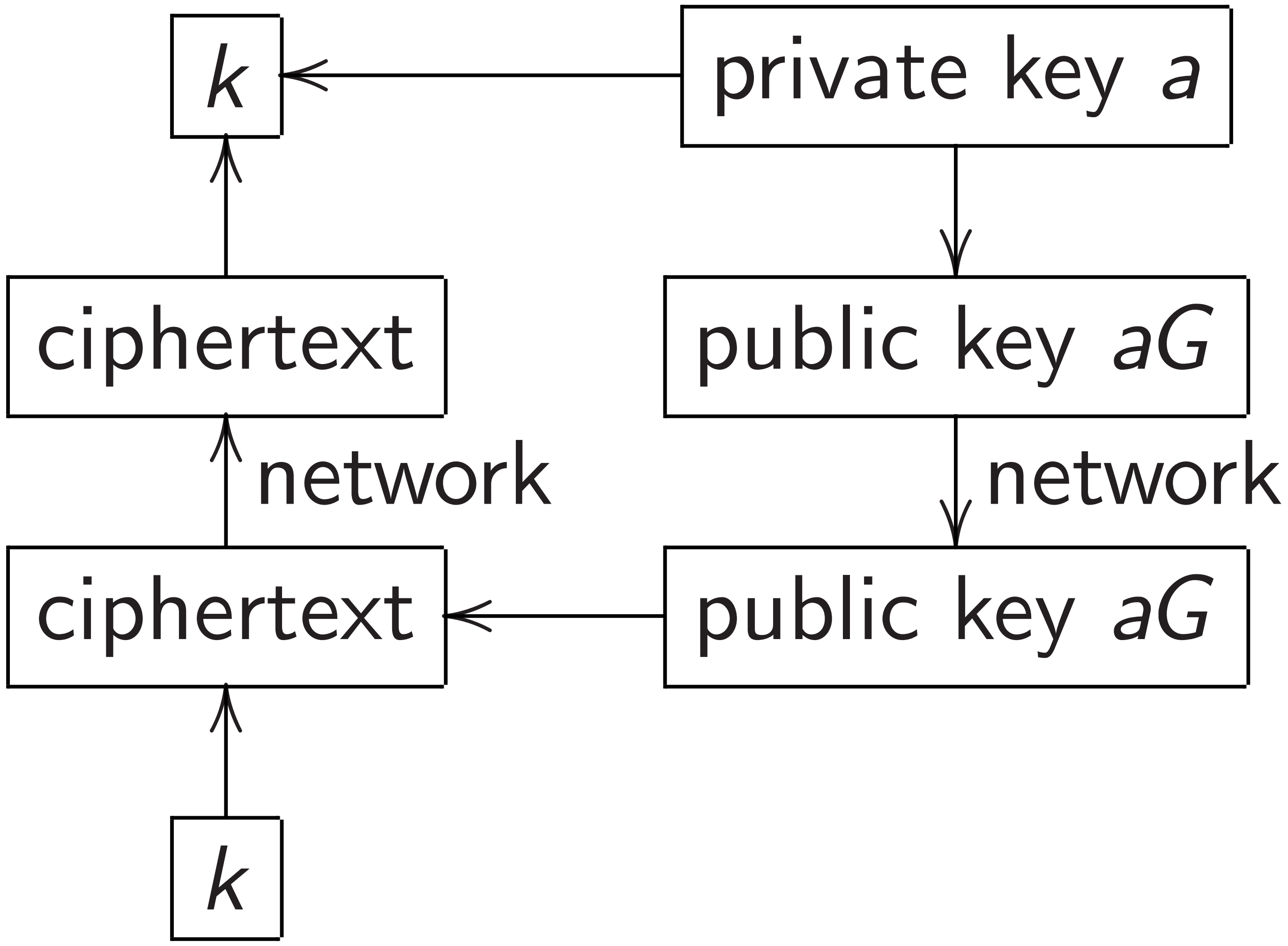
Cryptographic solution: Message-authentication codes.



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

What if attacker was spying on their communication of k?

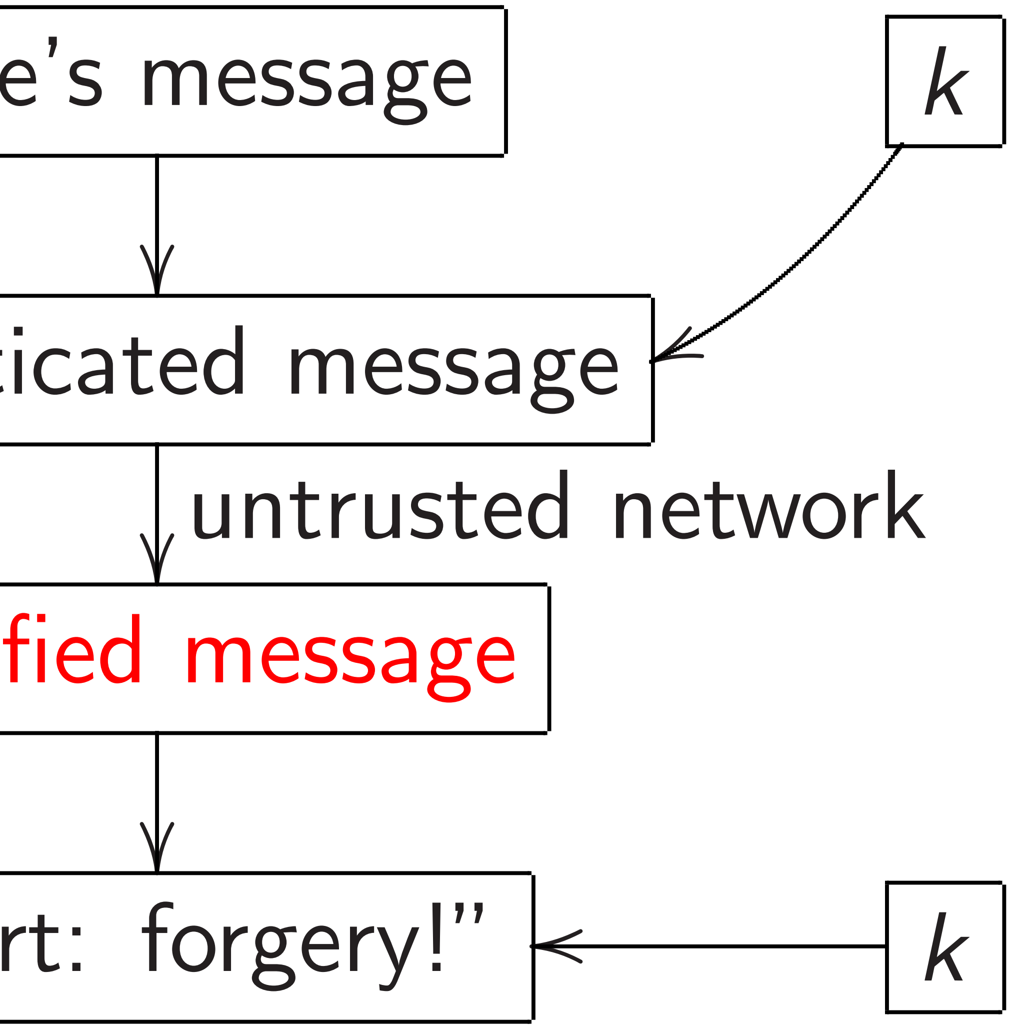
Solution 1: Public-key encryption.



f this talk: Cryptography

es Bob's laptop know
oming network data
Alice's laptop?

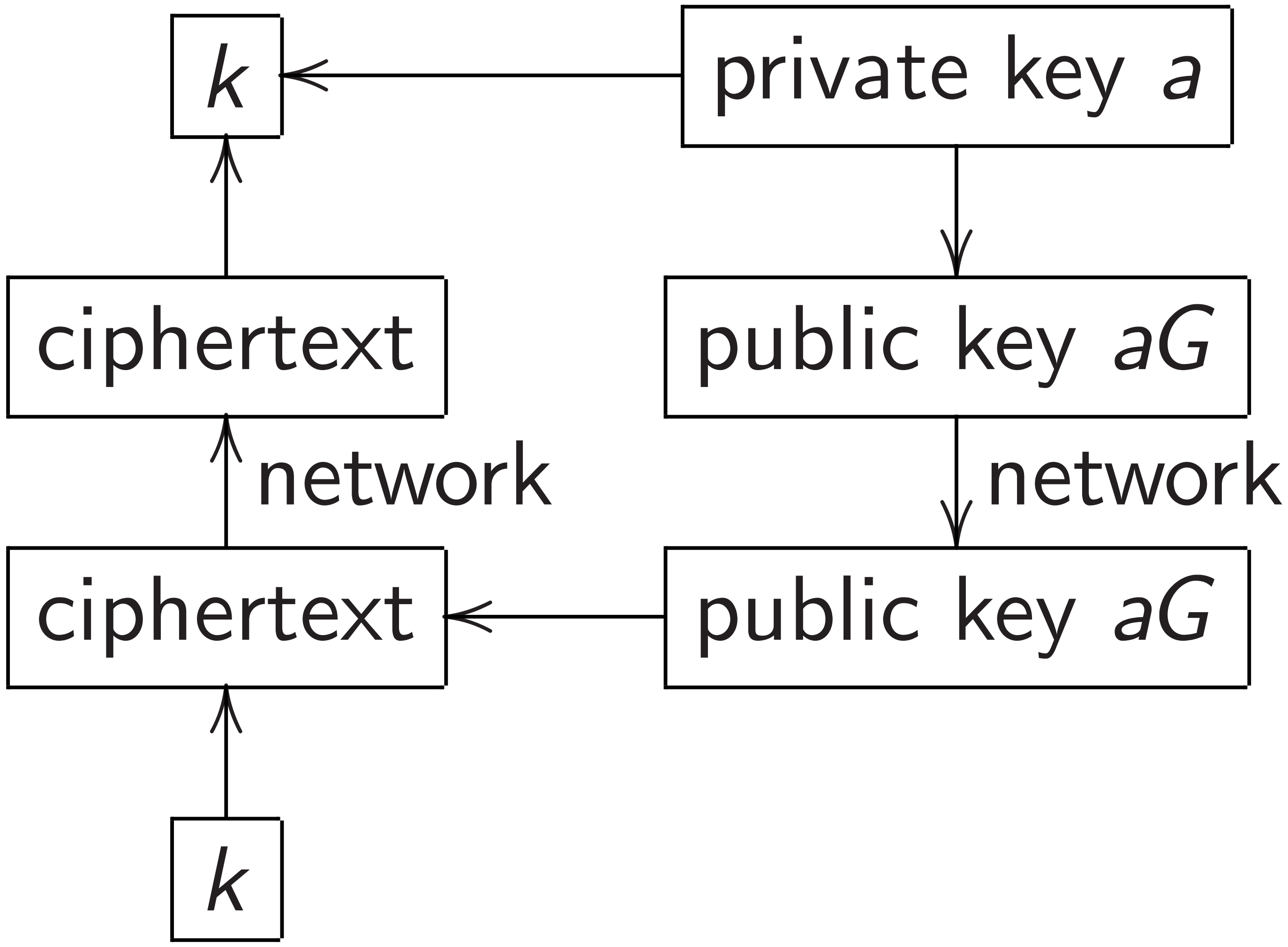
raphic solution:
e-authentication codes.



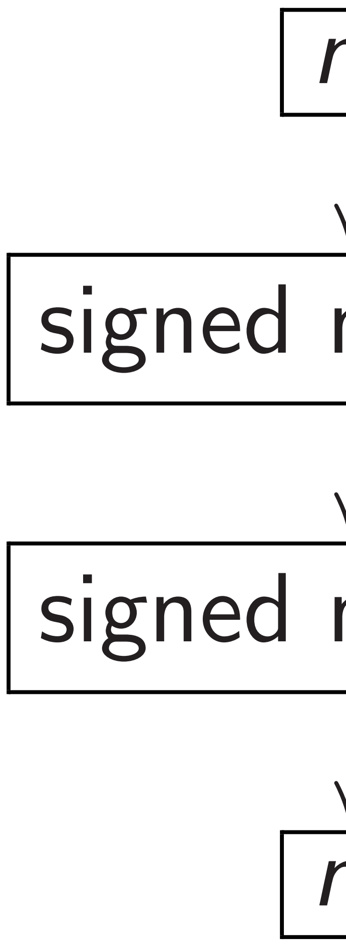
Important for Alice and Bob
to share the same secret k.

What if attacker was spying
on their communication of k?

Solution 1:
Public-key encryption.



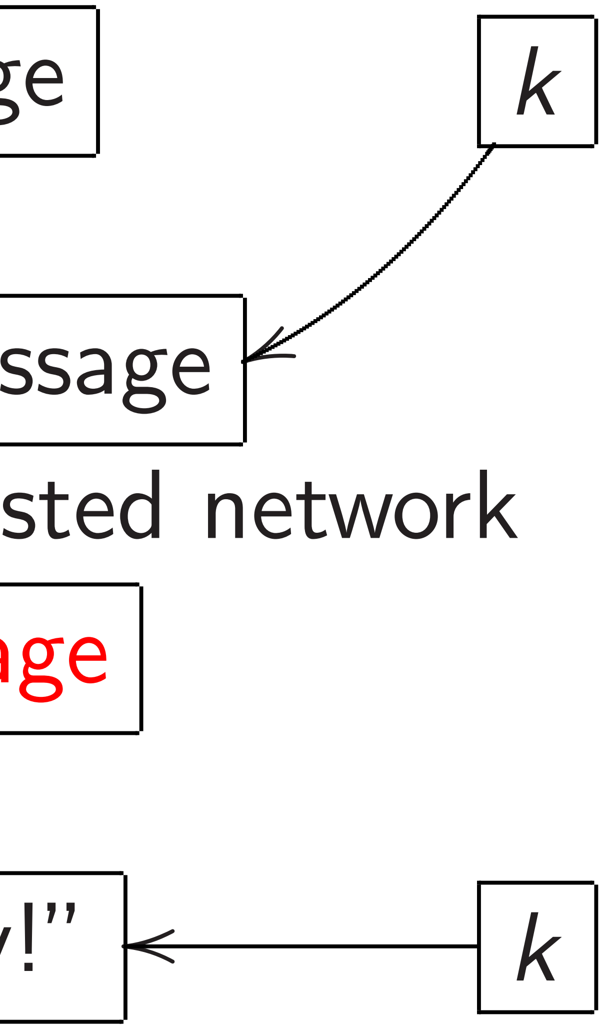
Solution
Public-k



Cryptography

aptop know
work data
top?

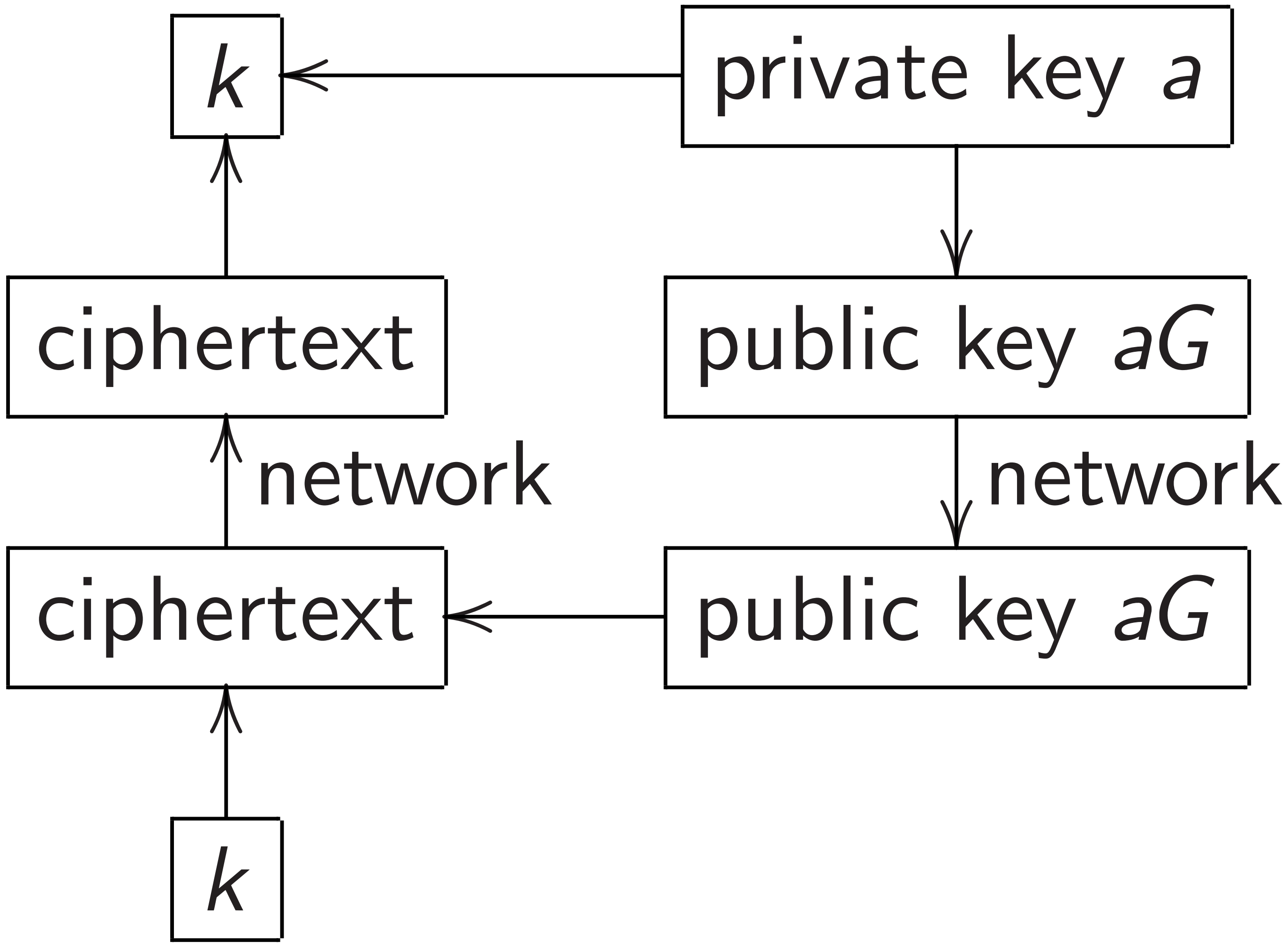
ution:
cation codes.



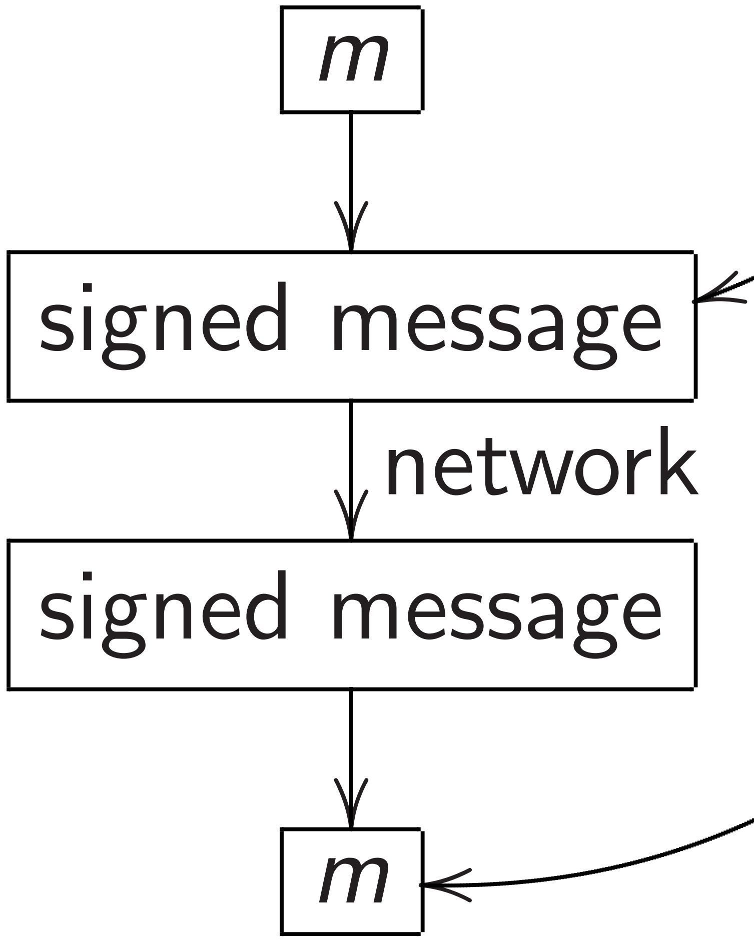
Important for Alice and Bob to share the same secret k.

What if attacker was spying on their communication of k?

Solution 1:
Public-key encryption.



Solution 2:
Public-key signatu



aphy

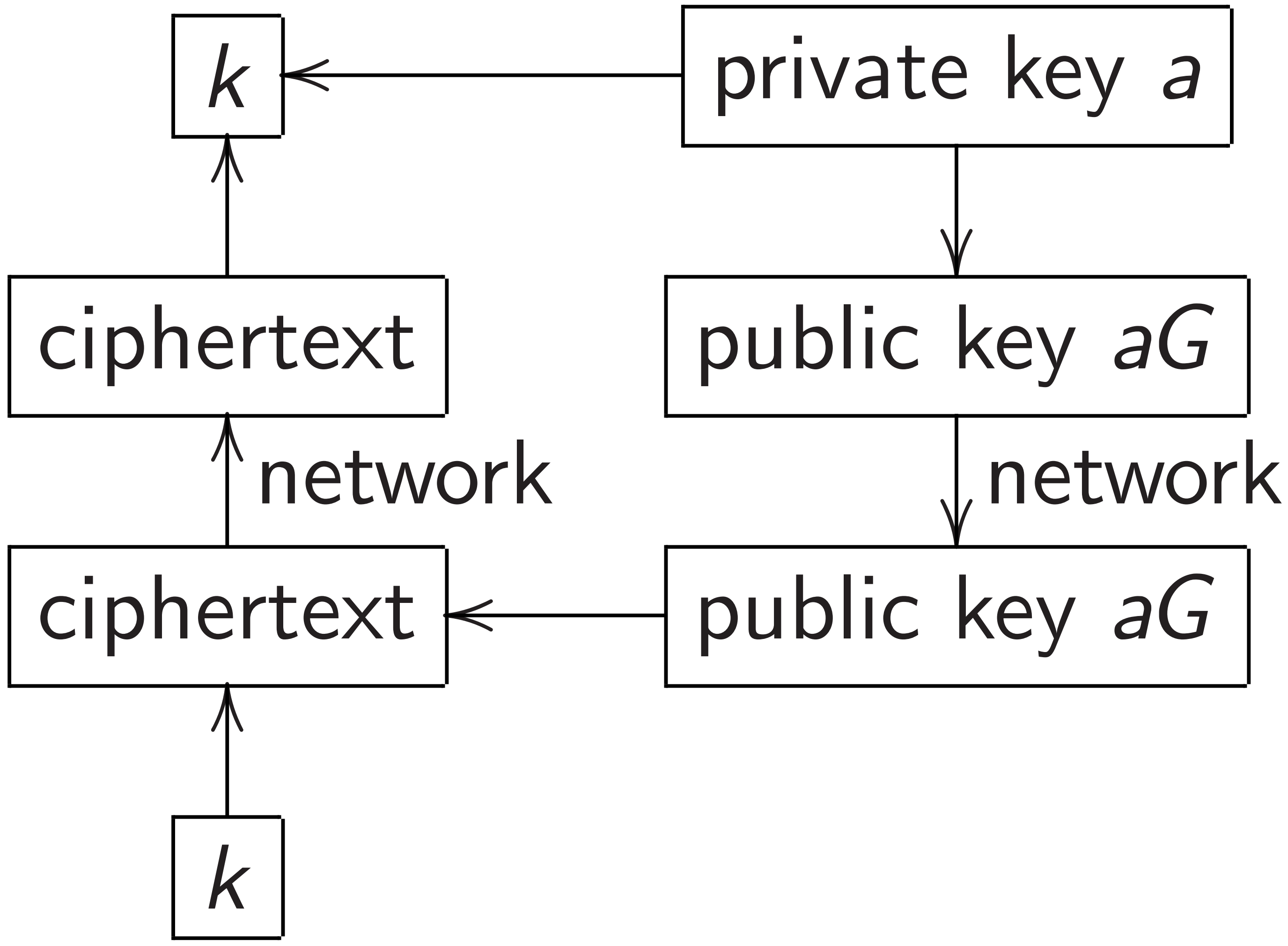
w

es.

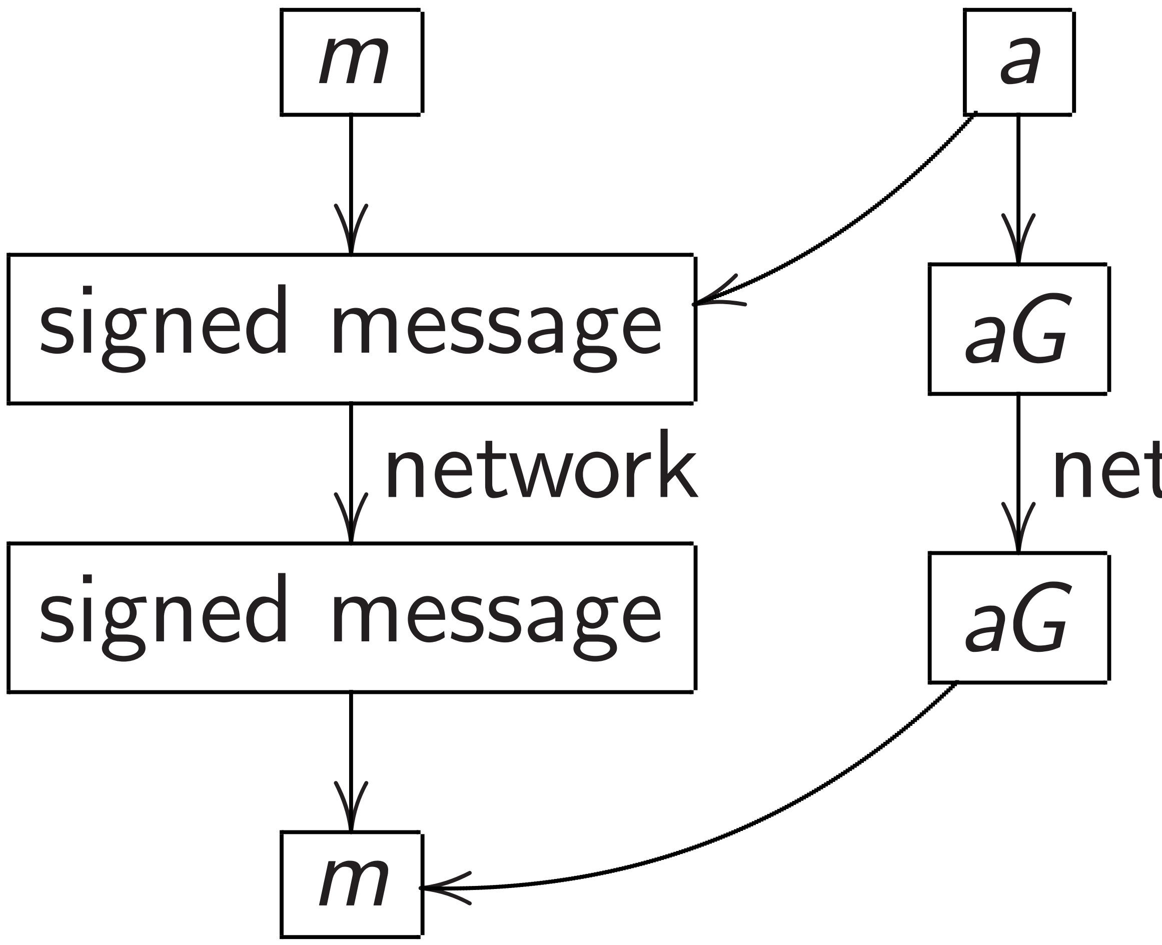
Important for Alice and Bob to share the same secret k.

What if attacker was spying on their communication of k?

Solution 1:
Public-key encryption.



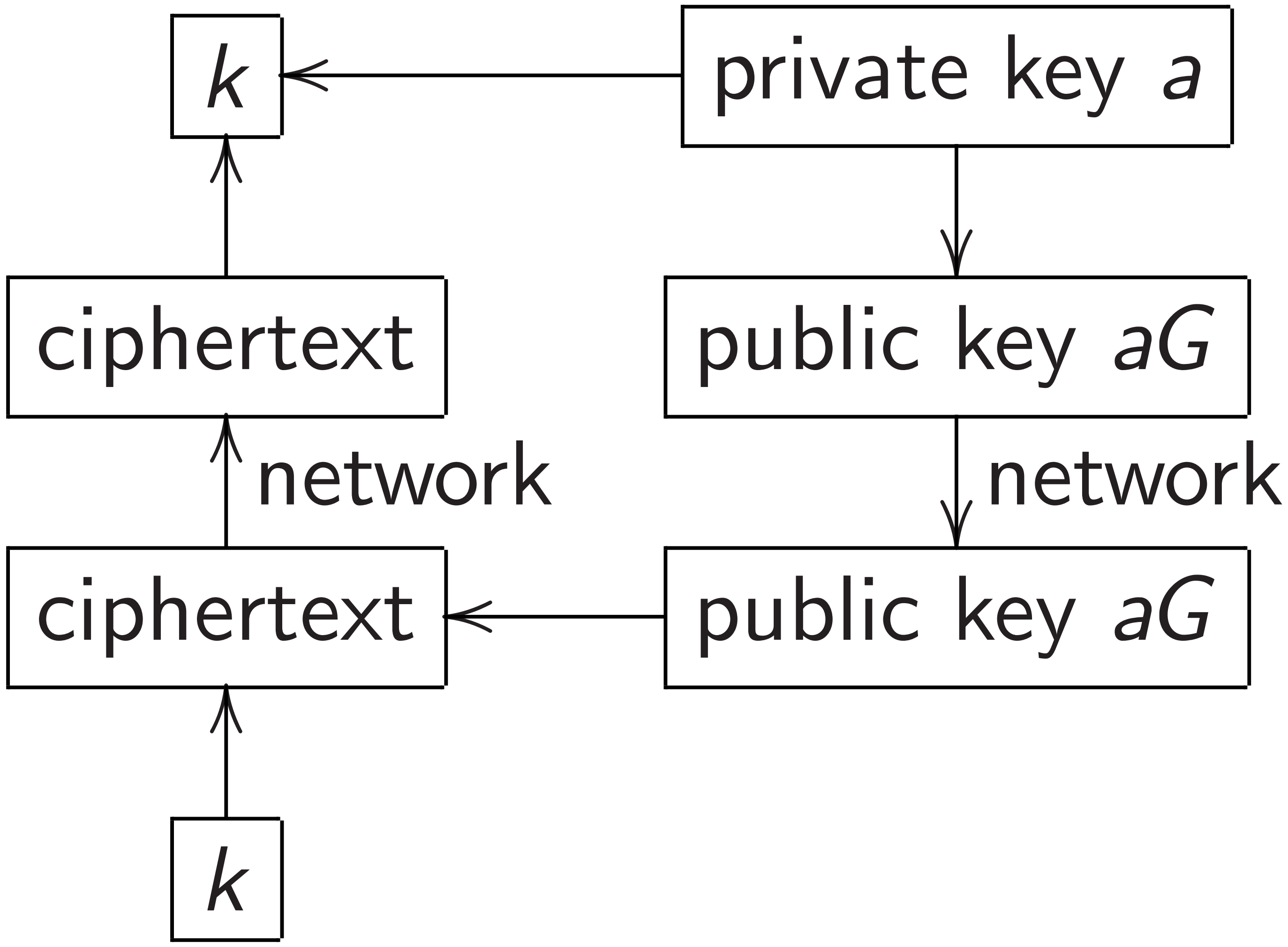
Solution 2:
Public-key signatures.



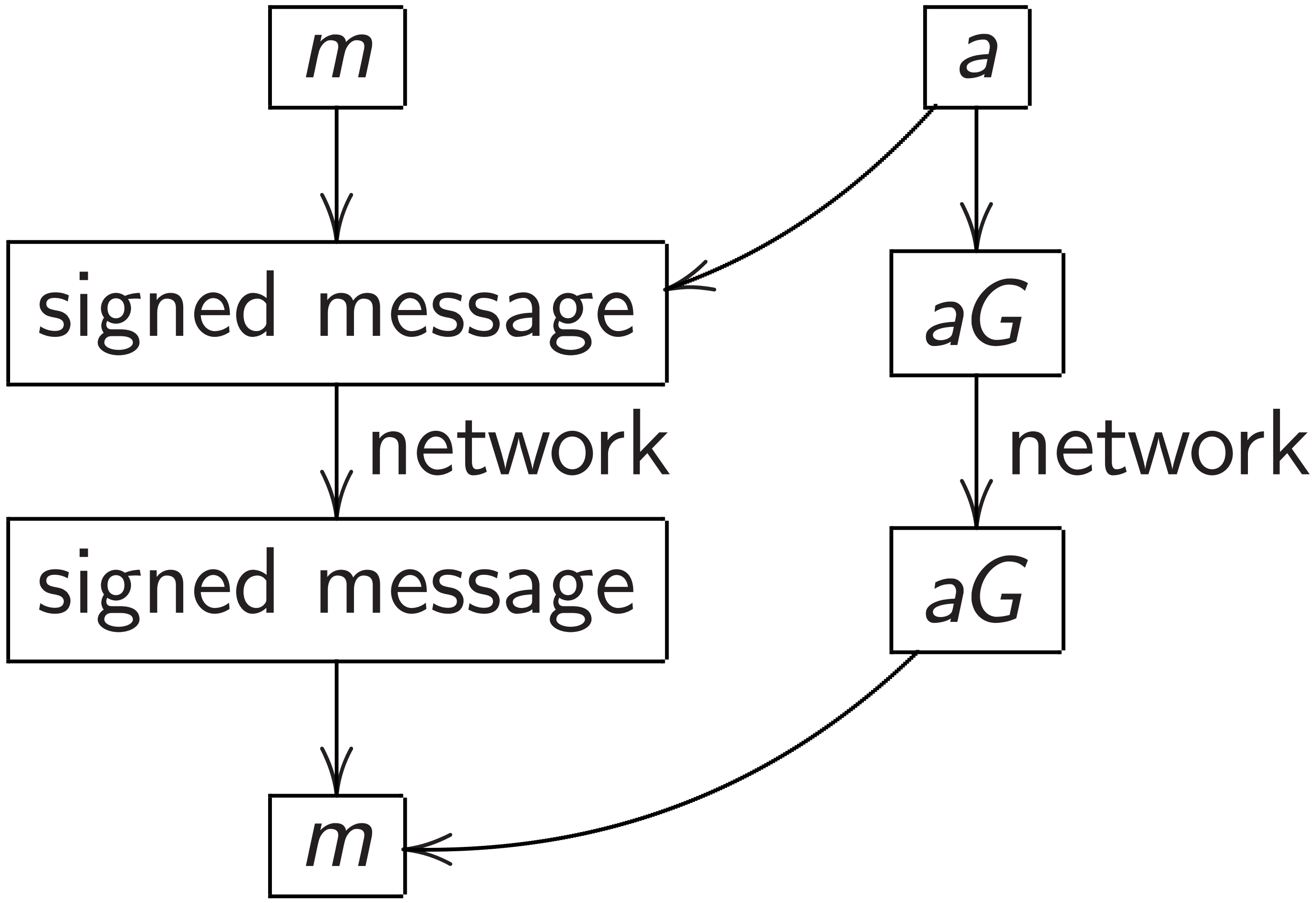
Important for Alice and Bob to share the same secret k.

What if attacker was spying on their communication of k?

Solution 1:
Public-key encryption.



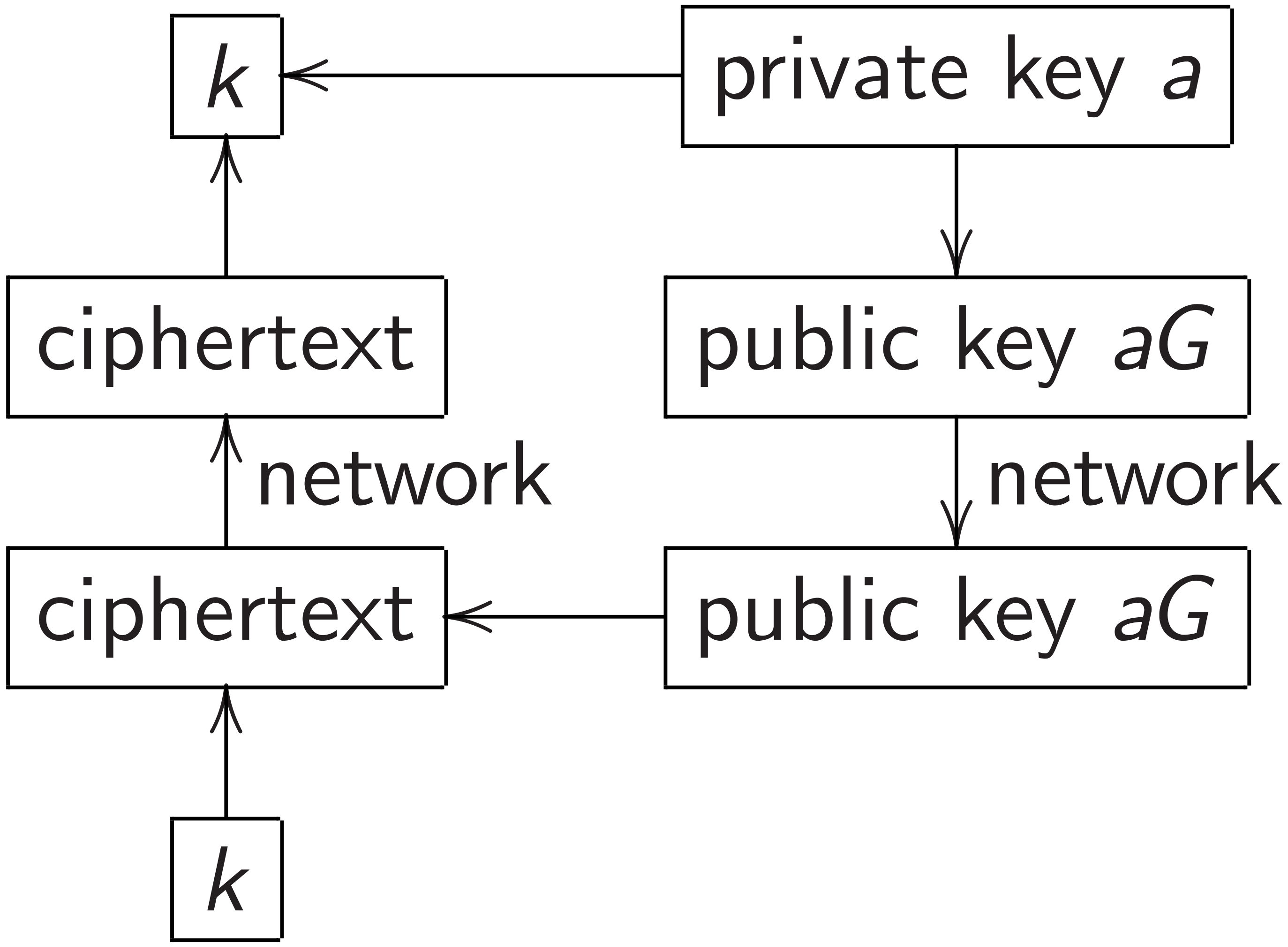
Solution 2:
Public-key signatures.



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

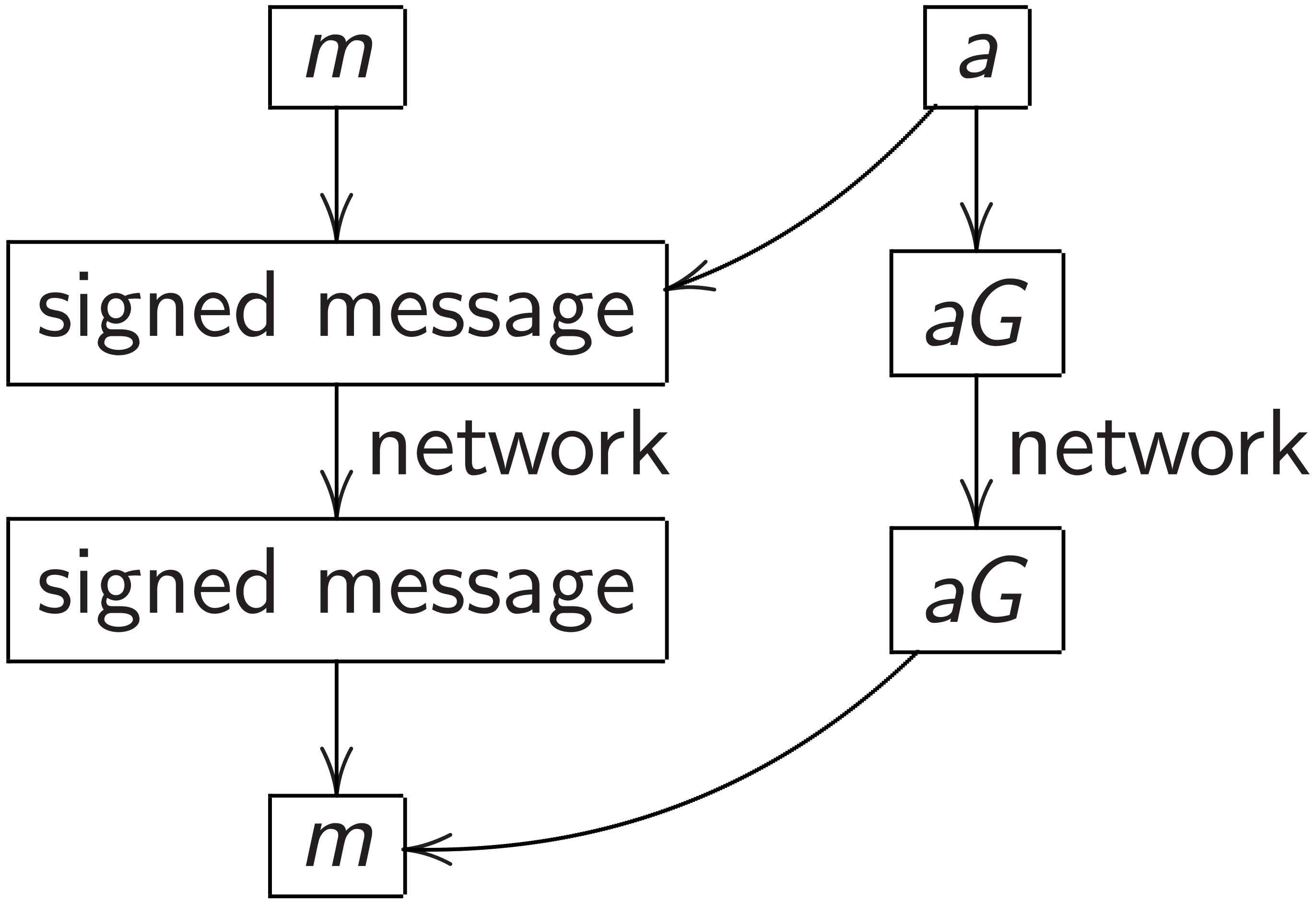
network

ciphertext

public key aG

k

Solution 2:
Public-key signatures.



m

a

signed message

aG

network

network

signed message

aG

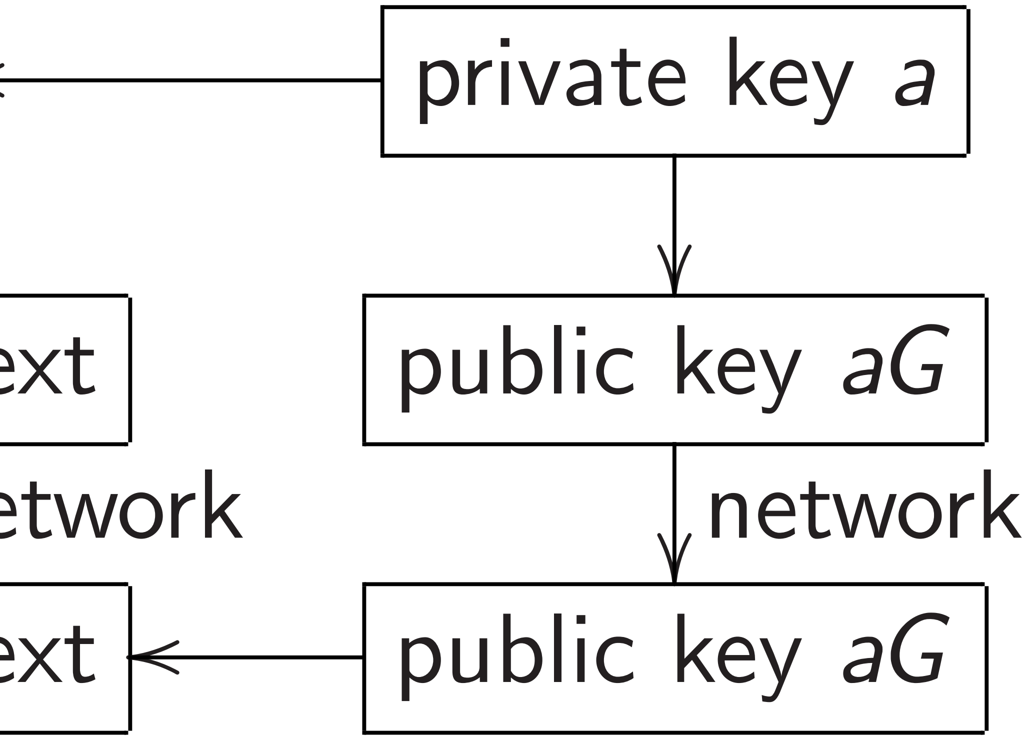
m

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

nt for Alice and Bob
the same secret k.

attacker was spying
communication of k?

1:
ey encryption.


private key a

ext

public key aG

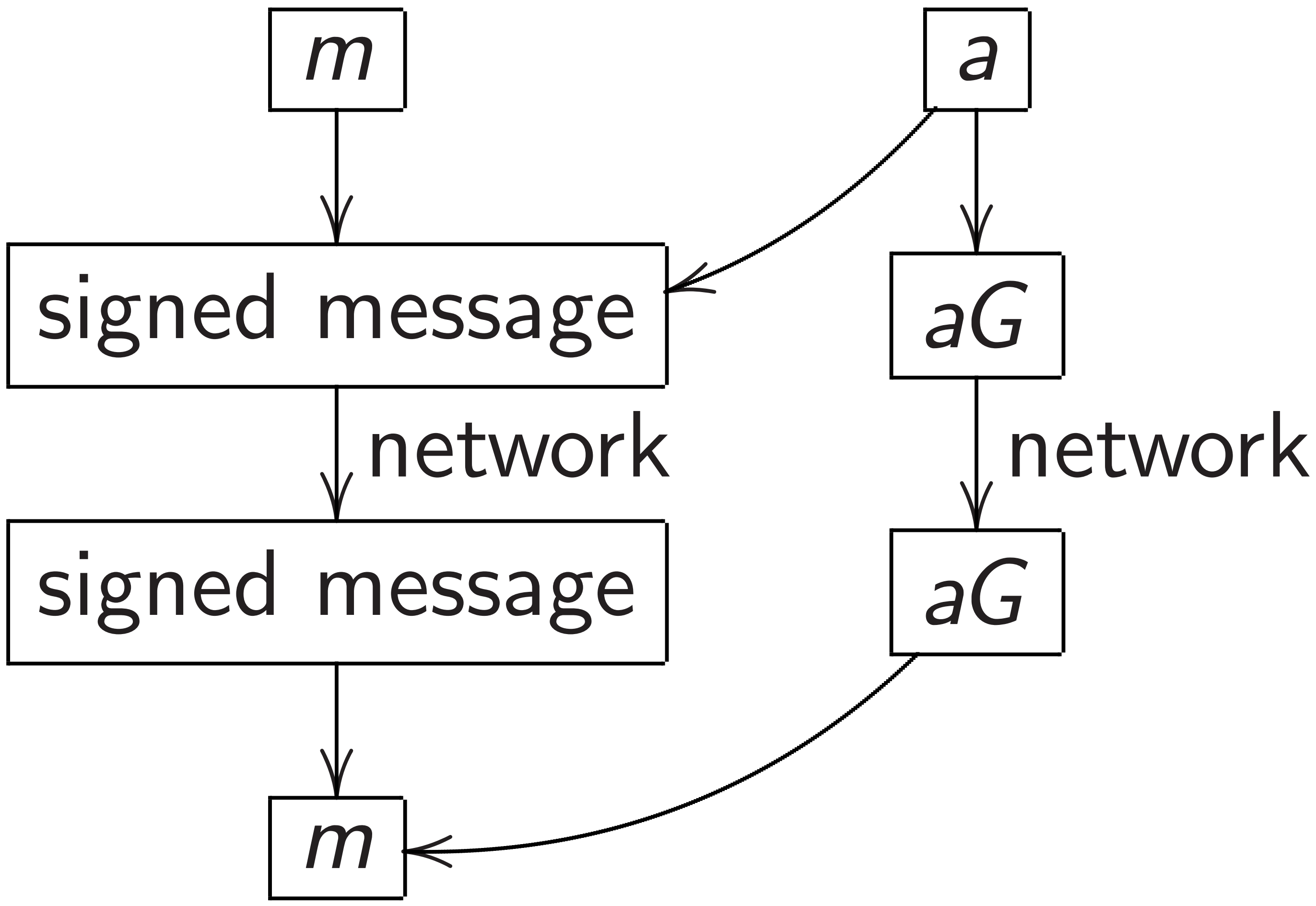
etwork

network

ext

public key aG

Solution 2:
Public-key signatures.


m

a

signed message

aG

network

network

signed message

aG

m

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

Real wor

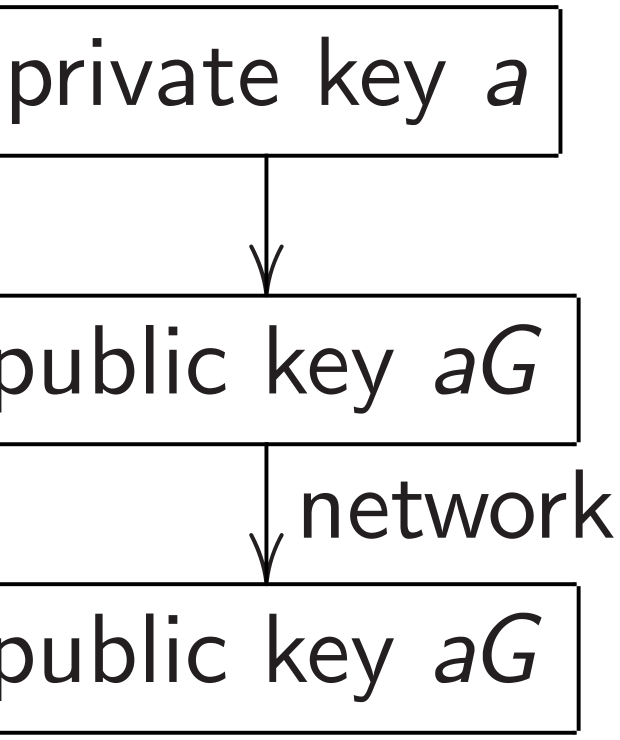
Cryptog
is huge.
of many

Most co

e and Bob
secret k.

was spying
cation of k?

tion.



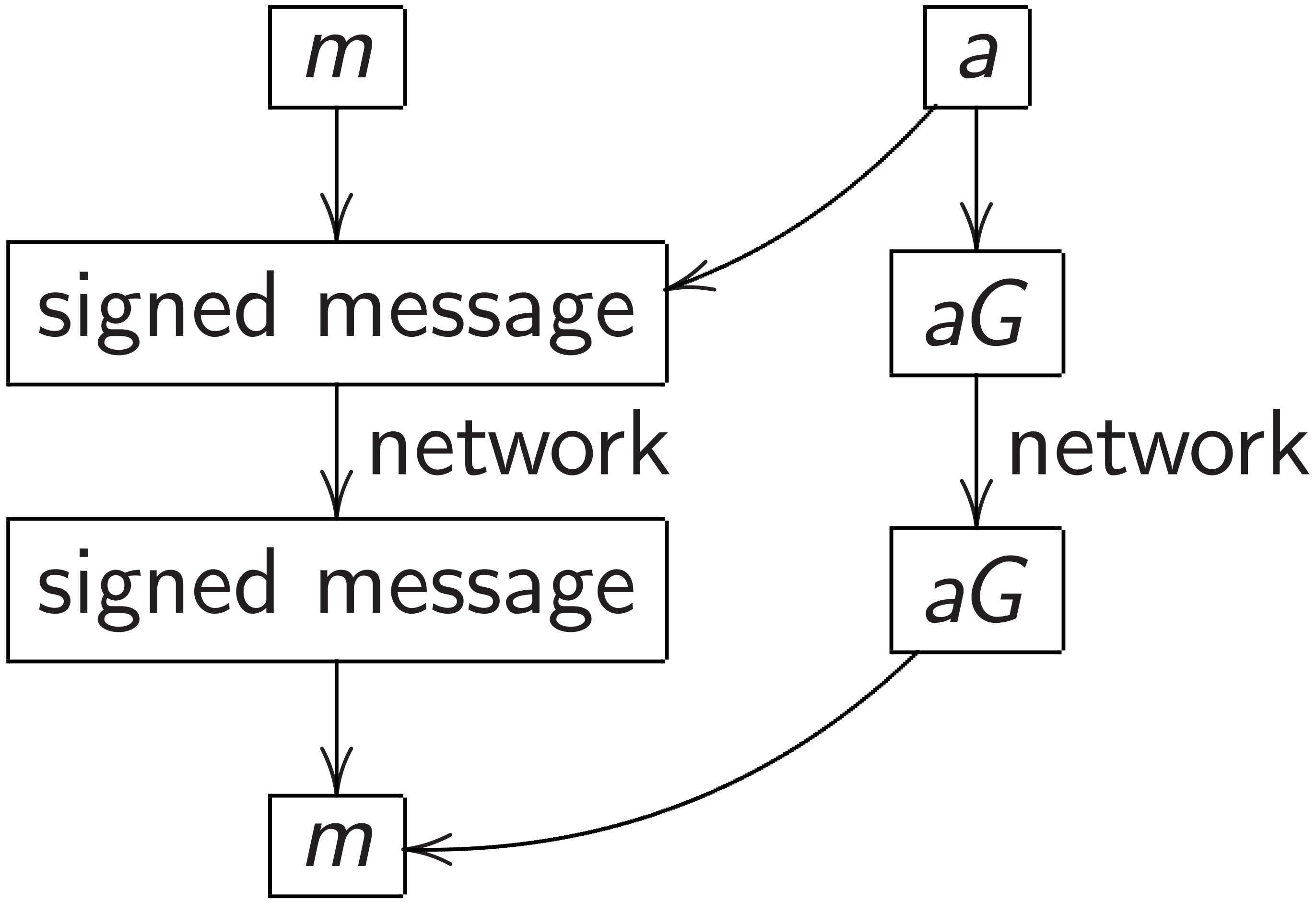
private key a

public key aG

network

public key aG

Solution 2:
Public-key signatures.



m

a

signed message

aG

network

network

signed message

aG

m

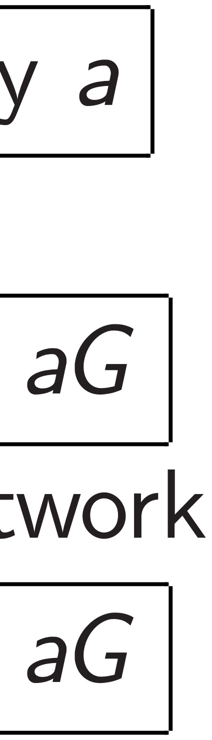
Fantasy world: software for authentication/encryption/sigs is small and carefully audited ⇒ no cryptographic security failures.

Real world:

Cryptographic par
is huge. Many im
of many cryptogra

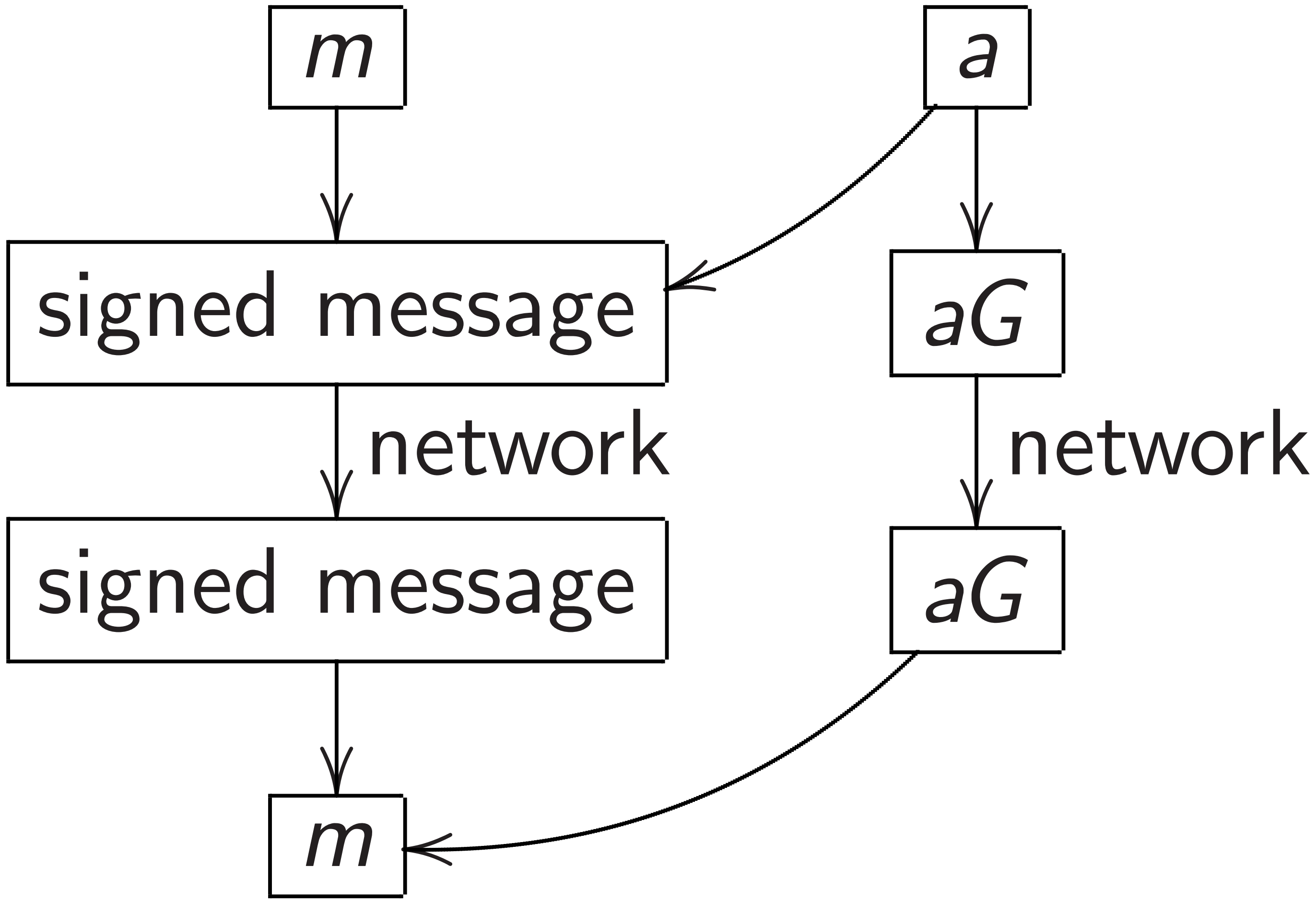
Most complicatio

k?



Solution 2:

Public-key signatures.



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

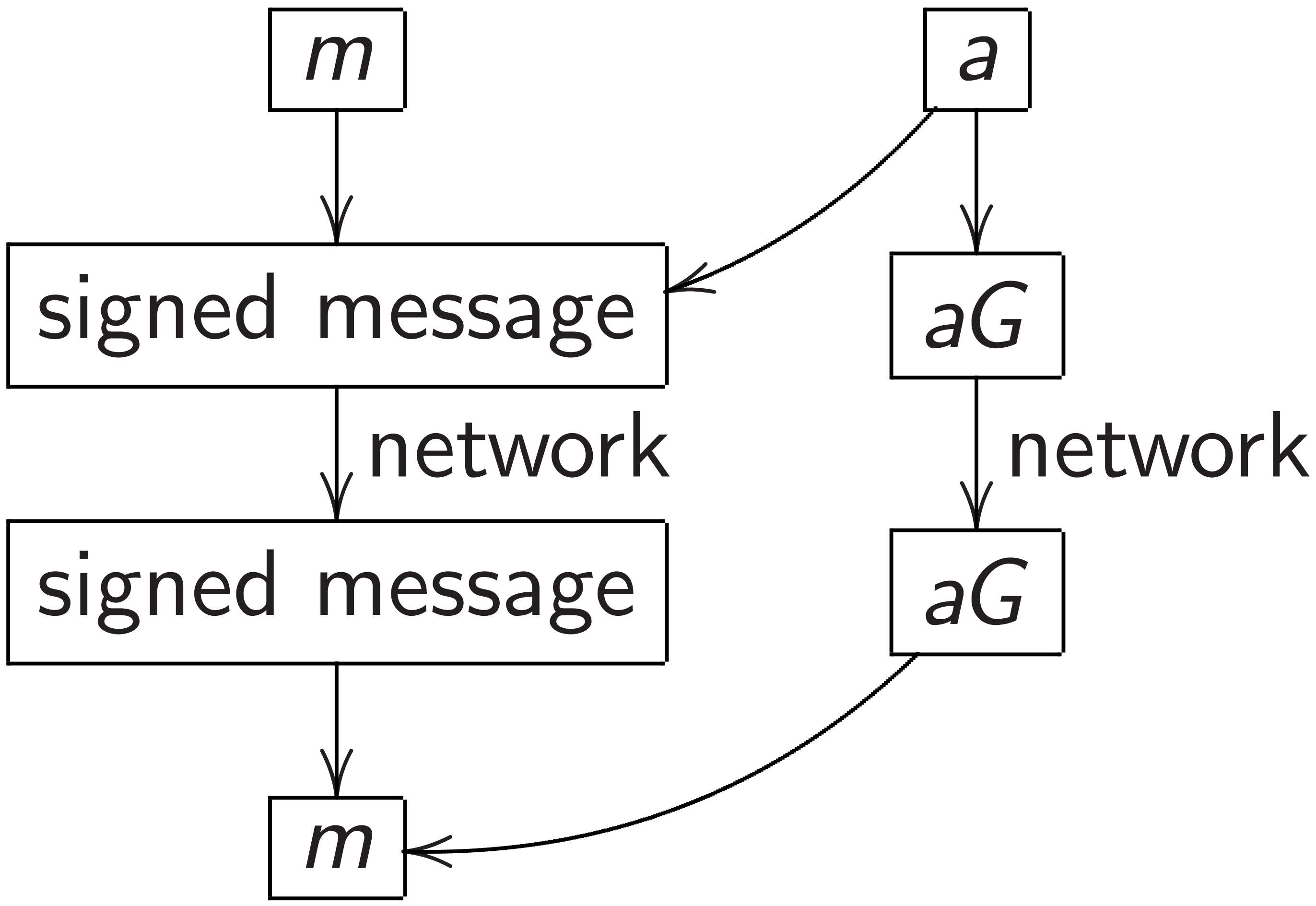
Real world:

Cryptographic part of the T
is huge. Many implementati
of many cryptographic primi

Most complications are for s

Solution 2:

Public-key signatures.



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

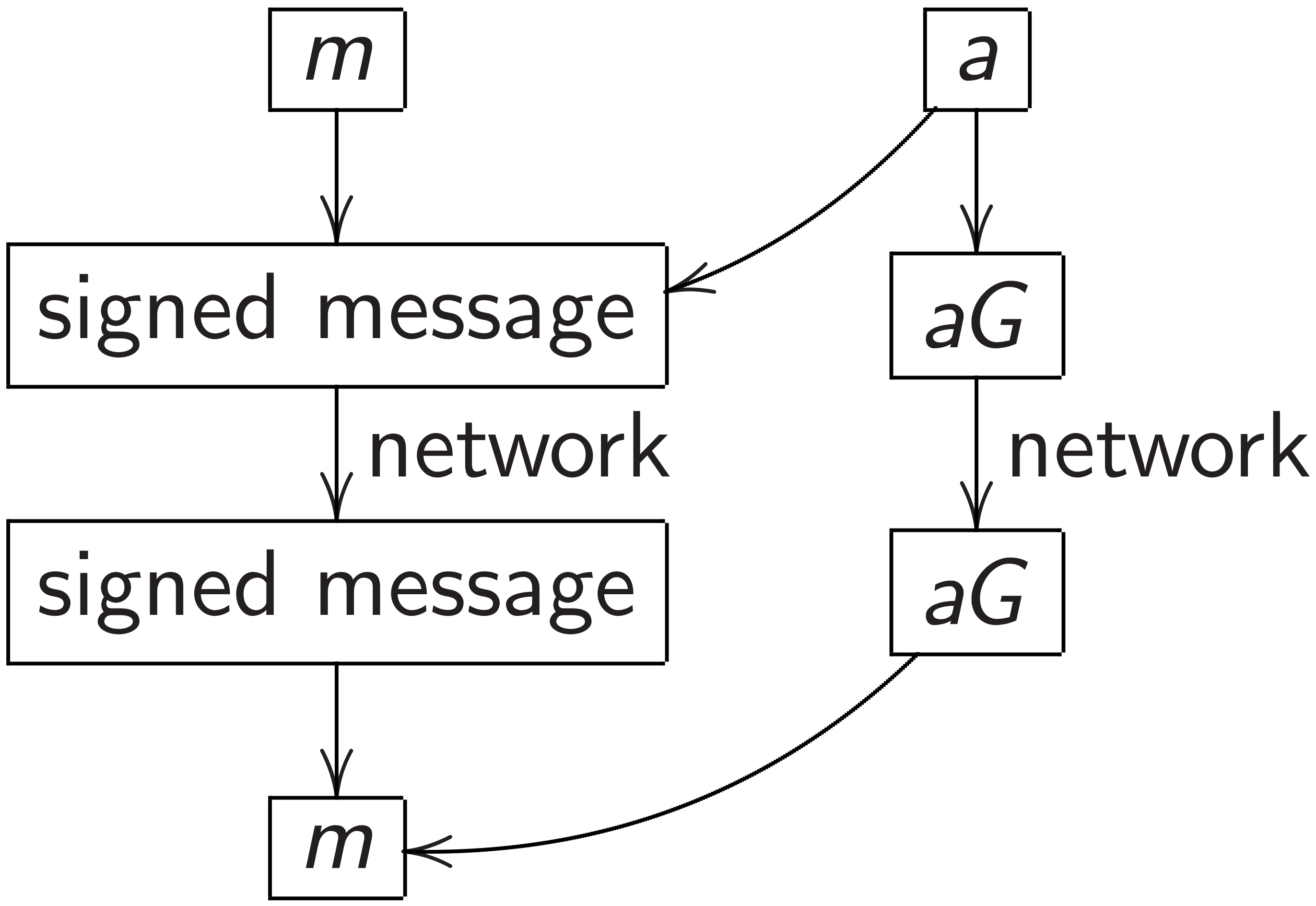
Real world:

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

Most complications are for speed.

Solution 2:

Public-key signatures.



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

Real world:

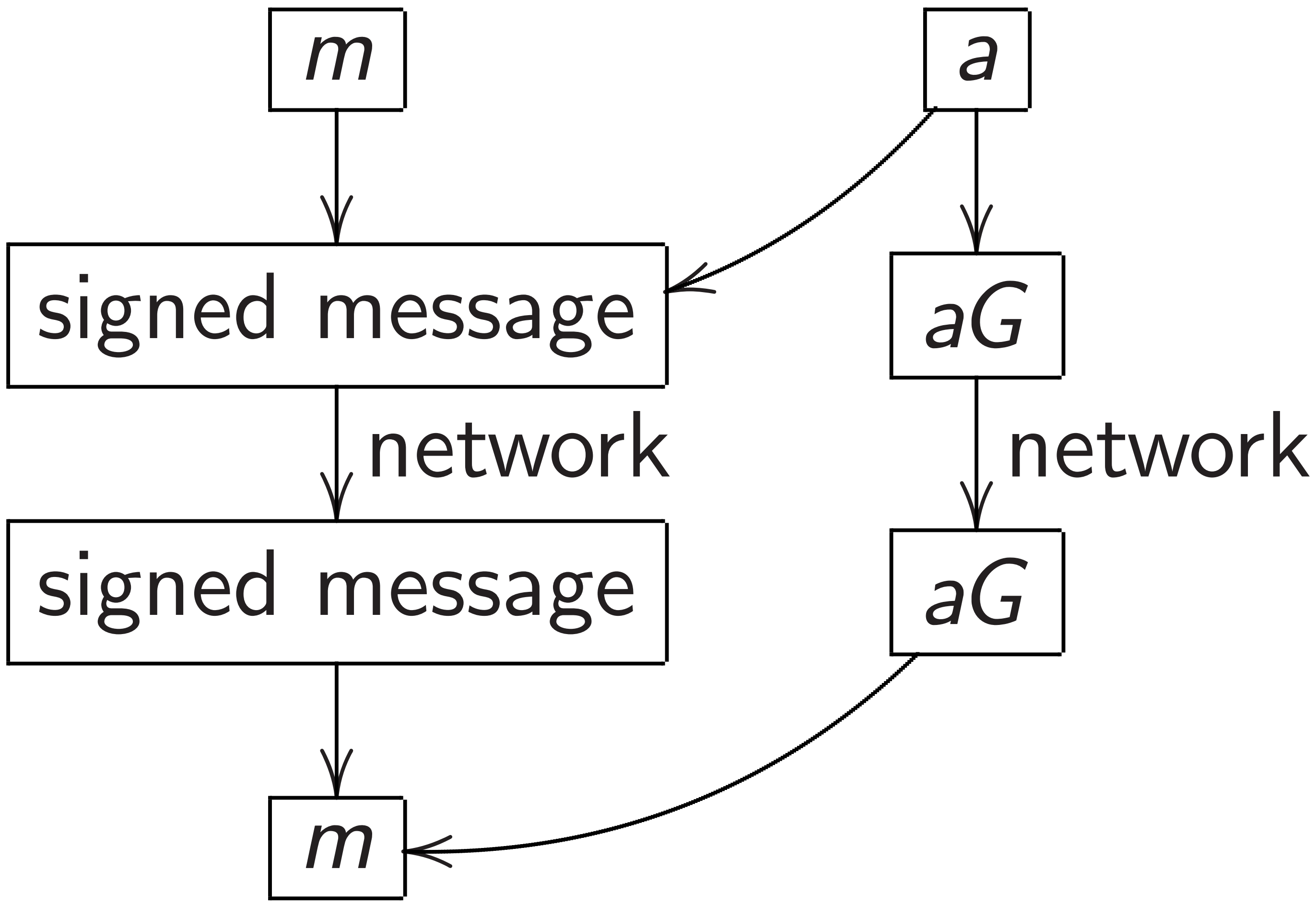
Cryptographic part of the TCB is huge. Many implementations of many cryptographic primitives.

Most complications are for speed.

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

Solution 2:

Public-key signatures.



m

a

signed message

aG

network

network

signed message

aG

m

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.

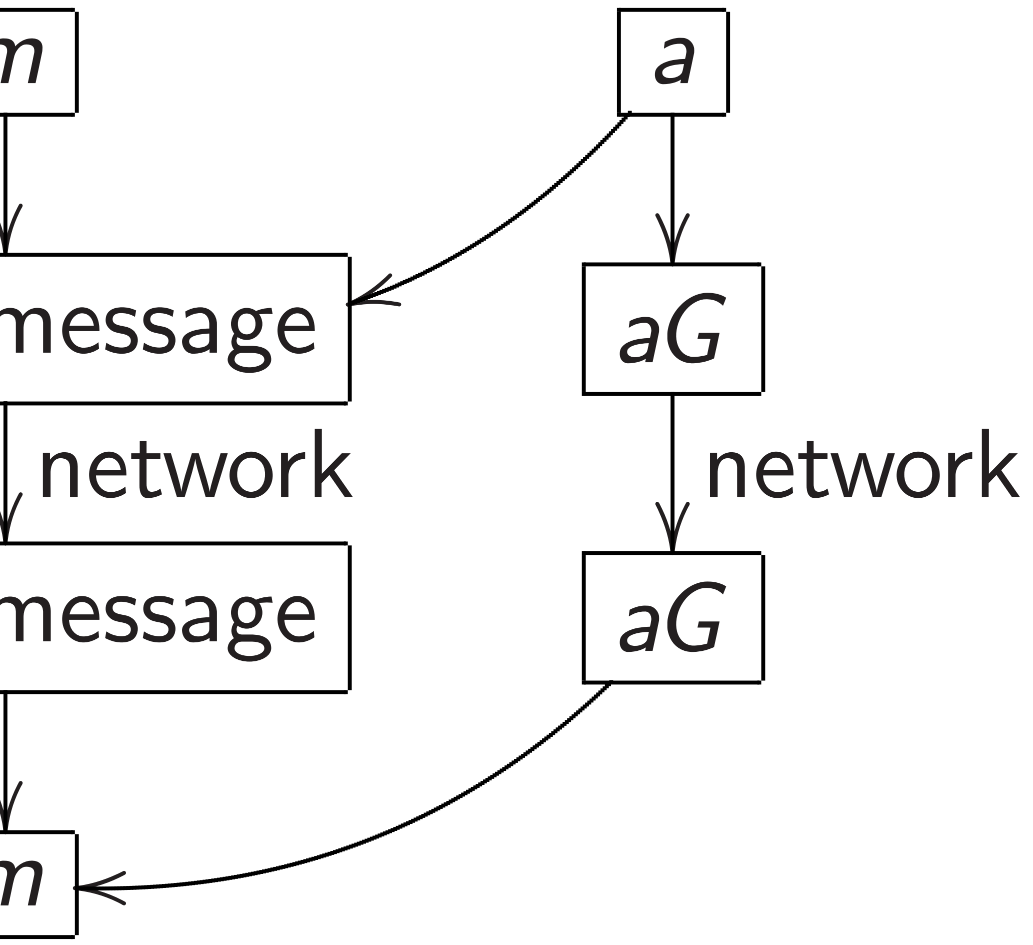
Most complications are for speed.

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.

2:

ey signatures.



world: software for
cation/encryption/sigs
and carefully audited ⇒
ographic security failures.

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

“Keccak

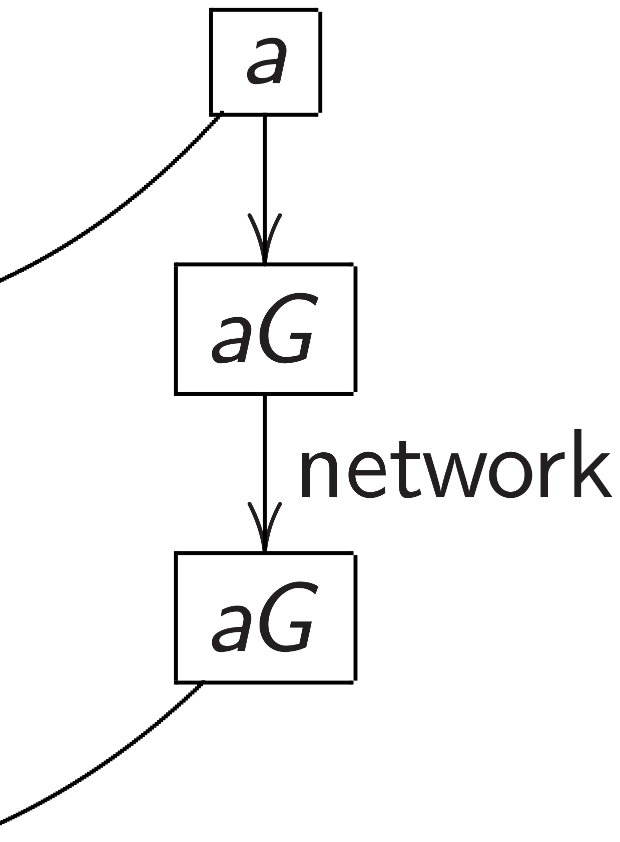
>20 opt

of Kecca

Includes

many fu

res.



ftware for
cryption/sigs
ully audited ⇒
security failures.

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) t
"Keccak Code Pac
>20 optimized im
of Keccak: AVX2,
Includes "parallel
many further impl

twork

igs
d ⇒
ilures.

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 main
"Keccak Code Package" wit
>20 optimized implementat
of Keccak: AVX2, NEON, e
Includes "parallel Keccak":
many further implementatio

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.

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.

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

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.

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.

rld:

raphic part of the TCB
Many implementations
cryptographic primitives.

mplications are for speed.

ruary 2018: Google adds
peck cipher to Linux
sing hand-written asm
1 Cortex-A7 processors.

2018: Google switches
eck to ChaCha12, again
nd-written assembly.
t 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.

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 pe
cryptogr
e.g. NIS
really lik
optimize
⇒ More

t of the TCB
plementations
phic primitives.

ns are for speed.

3: Google adds
er to Linux
written asm
A7 processors.

ogle switches
aCha12, again
n assembly.
20? Speed.

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 c
cryptographic perf
e.g. NIST, May 20
really like to see m
optimized impleme
⇒ More and more

CB
ons
itives.

speed.

adds
x
m
ors.

es
gain
y.
.

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 a
cryptographic performance.
e.g. NIST, May 2018: "we'd
really like to see more platfo
optimized implementations"
⇒ More and more software.

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”.
⇒ More and more software.

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".
⇒ More and more software.

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

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”. ⇒ 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.

(SHA-3) team maintains
Code Package" with
timized implementations
ak: AVX2, NEON, etc.
"parallel Keccak":
rther implementations.

t portable C code using
zing" compiler? Slower.

example: many different
es in NIST competition
-quantum public-key
aphy. (See next talk.)

verlap in implementations,
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".
⇒ 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 a

Large po
optimiza
addresse

Consider
instructi
parallel c
store-to-
branch p

eam maintains
ckage" with
plementations
NEON, etc.
Keccak":
ementations.

C code using
piler? Slower.

many different
competition
public-key
e next talk.)

mplementations,
me of code.

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 C
optimizations depe
addresses of memo

Consider data cac
instruction caching
parallel cache ban
store-to-load forwa
branch prediction,

tains
th
ions
tc.

ns.

sing
wer.

ferent
ion

k.)

tions,
e.

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 hardw
optimizations depending on
addresses of memory locatio

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

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.

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.

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

eople still complain about
aphic performance.
T, May 2018: "we'd
e to see more platform-
ed implementations".
and more software.

curity failures from
computations: e.g.,
17-3732, CVE-2017-3736,
17-3738 in OpenSSL.

curity failures from
time computations: e.g.
18-0495, CVE-2018-0737,
18-5407 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.

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

Typical

Underst
But det
not exp

Try to p
This be

Tweak t
to try t

complain about
formance.
018: "we'd
nore platform-
entations".
e software.

ures from
tions: e.g.,
CVE-2017-3736,
OpenSSL.

ures from
putations: e.g.
CVE-2018-0737,
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.

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

Typical literature o

Understand this p
But details are oft
not exposed to sec

Try to push attack
This becomes very

Tweak the attacke
to try to stop the

about

d

orm-

.

,

3736,

..

e.g.

0737,

..

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 top

Understand this portion of C
But details are often proprie
not exposed to security revie

Try to push attacks further.
This becomes very complica

Tweak the attacked software
to try to stop the known att

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.

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.

For researchers: This is great!

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.

For researchers: This is great!

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

attacks

ortion of CPU hardware:
ntions depending on
es of memory locations.

r data caching,
on caching,
cache banks,
-load forwarding,
prediction, etc.

ttacks (e.g. TLBleed from
as–Razavi–Bos–Giuffrida)
at this portion of the CPU
ble 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.

For researchers: This is great!

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

The “co
Don’t gi
to this p
(1987 G
Obliviou
domain-

CPU hardware:
ending on
ory locations.

hing,
g,
ks,
arding,
etc.

. TLBleed from
–Bos–Giuffrida)
rtion of the CPU
g 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.

For researchers: This is great!

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

The “constant-tim
Don’t give any sec
to this portion of t
(1987 Goldreich, 1
Oblivious RAM; 20
domain-specific fo

ware:

ons.

from

frida)

e CPU

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” solutio
Don’t give any secrets
to this portion of the CPU.
(1987 Goldreich, 1990 Ostro
Oblivious RAM; 2004 Berns
domain-specific for better sp

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)

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)

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

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)

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.

literature on this topic:

and this portion of CPU.
ails are often proprietary,
osed to security review.

ush attacks further.
comes very complicated.

he attacked software
› stop the known attacks.

archers: This is great!

tors: This is a nightmare.
ears of security failures.
idence 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)

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 stu

Subrouti
Classic N
Gravity-S
LEDApk
sort arra
e.g. sort

Typical s
merge sc
choose l
based on
also bran

How to
without

on this topic:

ortion of CPU.
en proprietary,
curity review.

ks further.
complicated.

ed software
known attacks.

This is great!

is a nightmare.
urity failures.
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)

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: Const

Subroutine in (e.g
Classic McEliece,
Gravity-SPHINCS,
LEDApkc, NTRU
sort array of secre
e.g. sort 768 32-bi

Typical sorting alg
merge sort, quicks
choose load/store
based on secret da
also branch based

How to sort secret
without any secret

pic:

CPU.

tary,

ew.

ted.

e

tacks.

nt!

tmare.

res.

urity.

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 s

Subroutine in (e.g.) BIG QU
Classic McEliece, GeMSS,
Gravity-SPHINCS, LEDAker
LEDApkc, NTRU Prime, Ro
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. Usual
also branch based on secret

How to sort secret data
without any secret addresses

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?

nstant-time" solution:
ve any secrets
portion of the CPU.
oldreich, 1990 Ostrovsky:
s RAM; 2004 Bernstein:
specific for better speed)

alysis: Need this portion
PU to be correct, but
ed it to keep secrets.
uditing much easier.

atch for attitude and
ce of CPU designers: e.g.,
ues errata for correctness
ot for information leaks.

Case study: Constant-time sorting

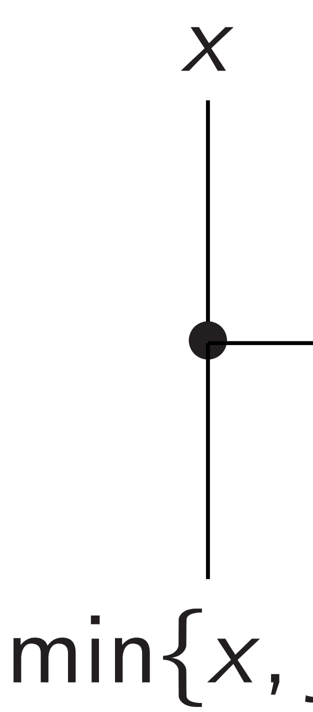
Subroutine in (e.g.) BIG QUAKE, Classic McEliece, GeMSS, Gravity-SPHINCS, LEDAkem, LEDApkc, NTRU Prime, Round2: sort array of secret integers. e.g. sort 768 32-bit integers.

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?

Foundat

a **compa**



Easy cor
Warning
compiler

Even eas

ne” solution:
crets
the CPU.
1990 Ostrovsky:
004 Bernstein:
r better speed)

ed this portion
correct, but
eep secrets.
uch easier.

ttitude and
J designers: e.g.,
for correctness
mation leaks.

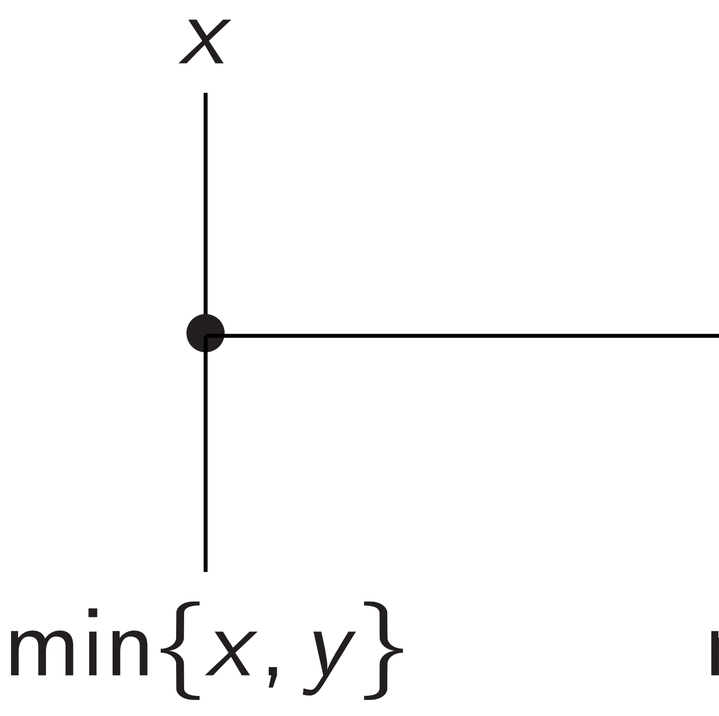
Case study: Constant-time sorting

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 solu
a **comparator** sor



Easy constant-tim
Warning: C stand
compiler to break

Even easier exercis

on:

ovsky:

tein:

peed)

rtion

ut

s.

r

d

s: e.g.,

tness

aks.

Case study: Constant-time sorting

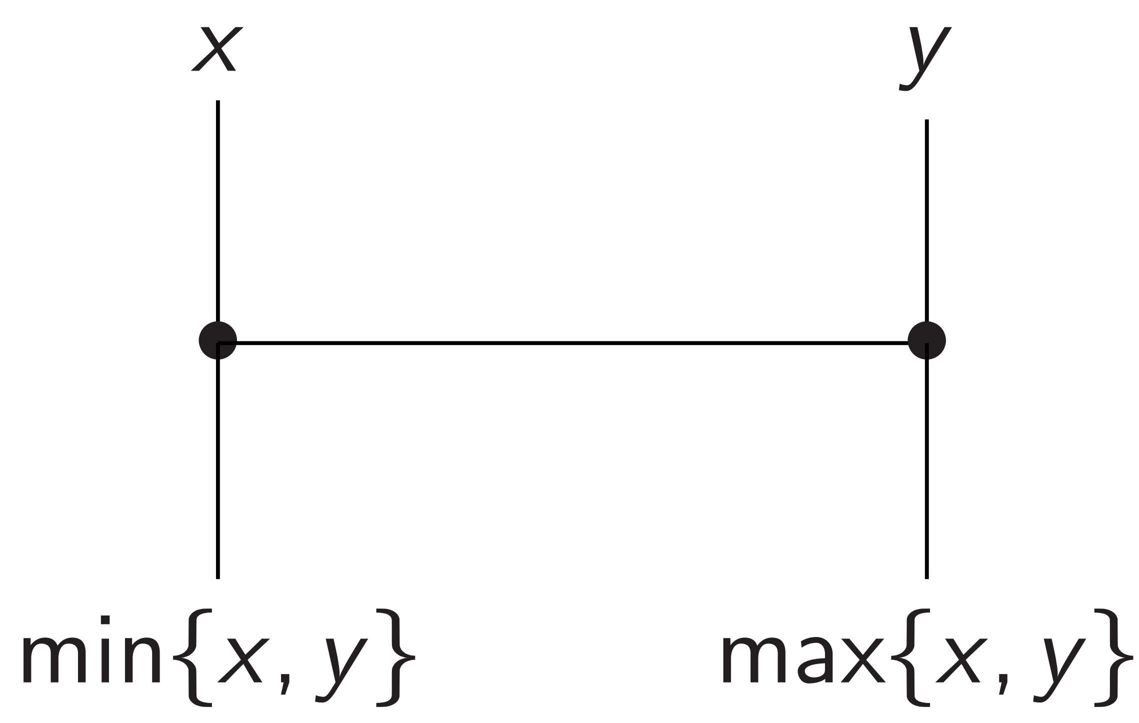
Subroutine in (e.g.) BIG QUAKE, Classic McEliece, GeMSS, Gravity-SPHINCS, LEDAkem, LEDApkc, NTRU Prime, Round2: sort array of secret integers. e.g. sort 768 32-bit integers.

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 inte



Easy constant-time exercise

Warning: C standard allows compiler to break the solutio

Even easier exercise in asm.

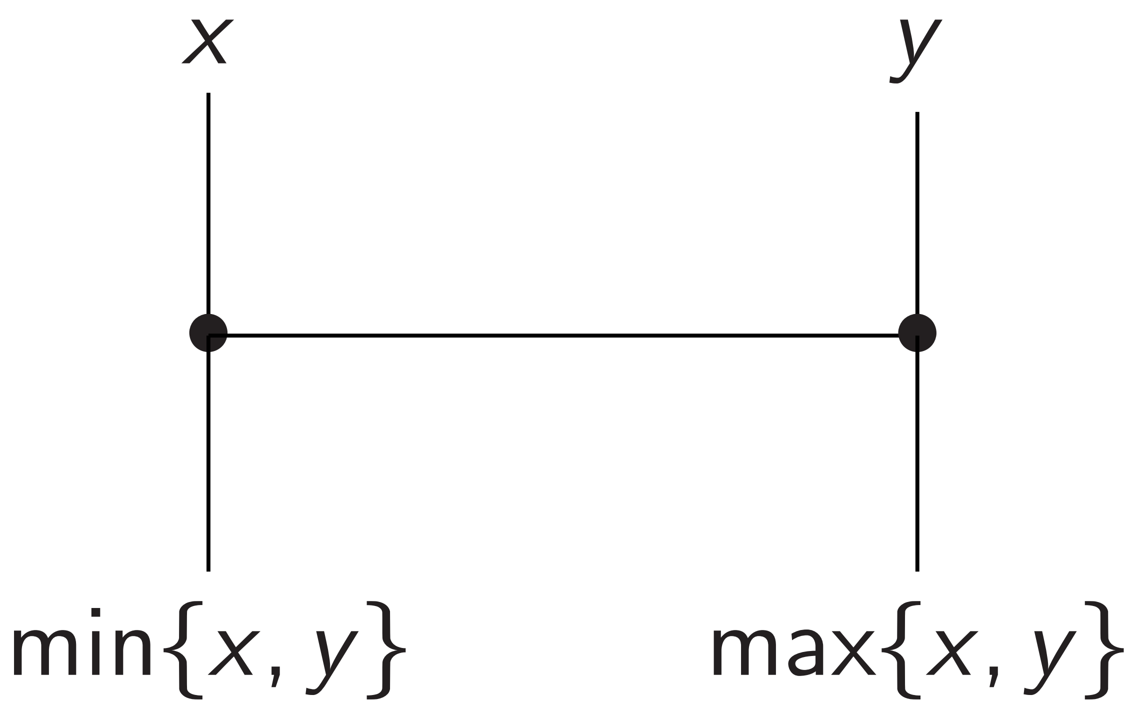
Case study: Constant-time sorting

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.

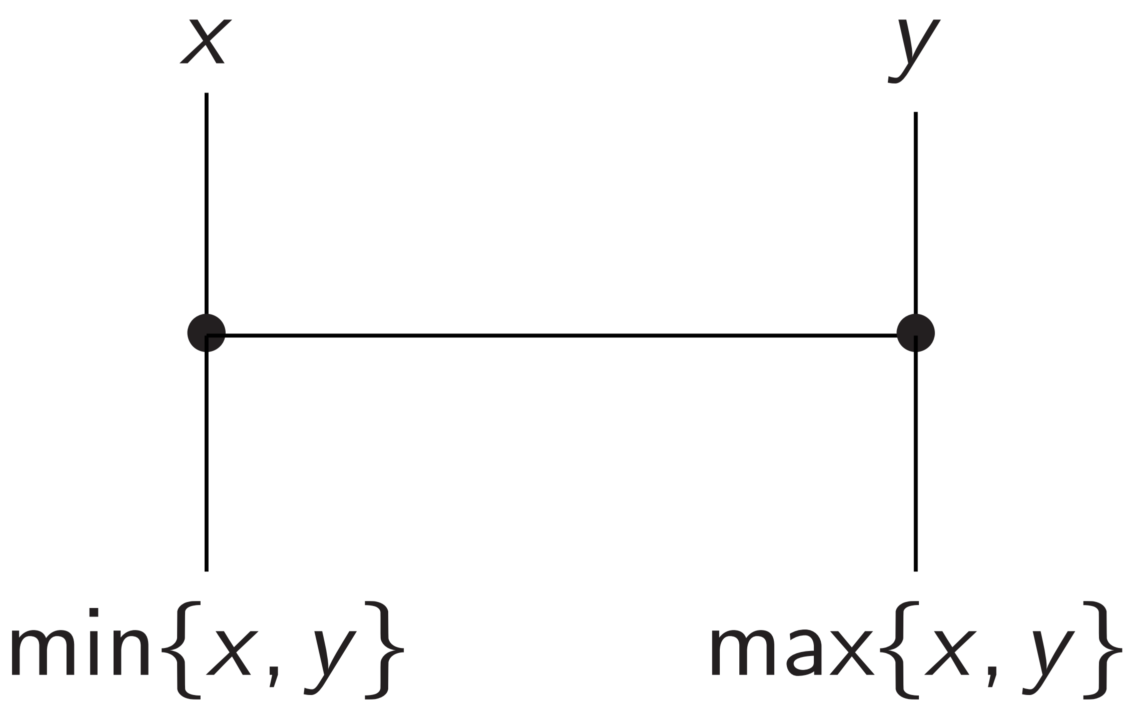
udy: Constant-time sorting

ine in (e.g.) BIG QUAKE,
McEliece, GeMSS,
SPHINCS, LEDAkem,
c, NTRU Prime, Round2:
y of secret integers.
768 32-bit integers.

sorting algorithms—
ort, quicksort, etc.—
oad/store addresses
n secret data. Usually
nch based on secret data.

sort secret data
any secret addresses?

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

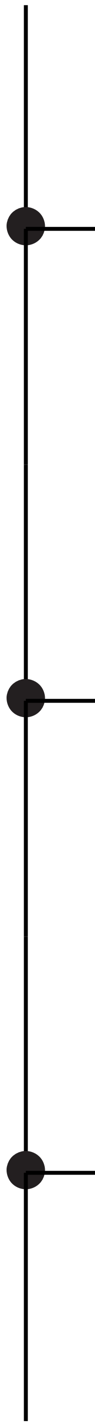


Easy constant-time exercise in C.
Warning: C standard allows
compiler to break the solution.

Even easier exercise in asm.

Combine
sorting

Example



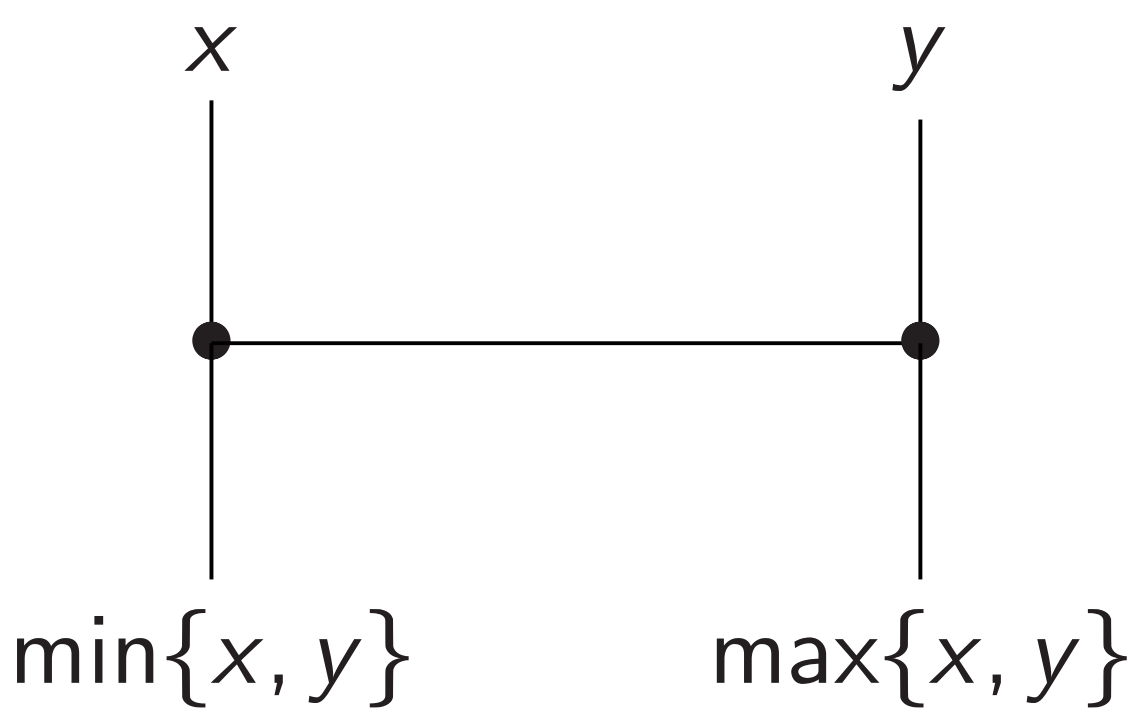
tant-time sorting

.) BIG QUAKE,
GeMSS,
LEDAkem,
Prime, Round2:
t integers.
t integers.

gorithms—
ort, etc.—
addresses
ata. Usually
on secret data.

t data
t addresses?

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

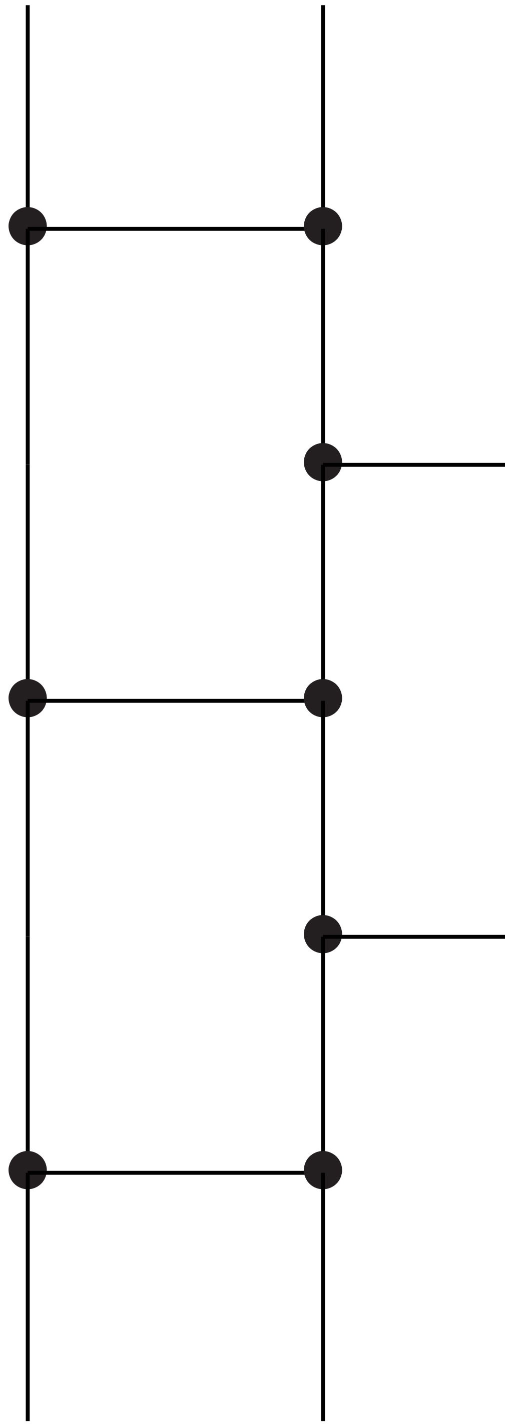


Easy constant-time exercise in C.
Warning: C standard allows
compiler to break the solution.

Even easier exercise in asm.

Combine comparat
sorting network f

Example of a sorti



sorting

JAKE,

n,

ound2:

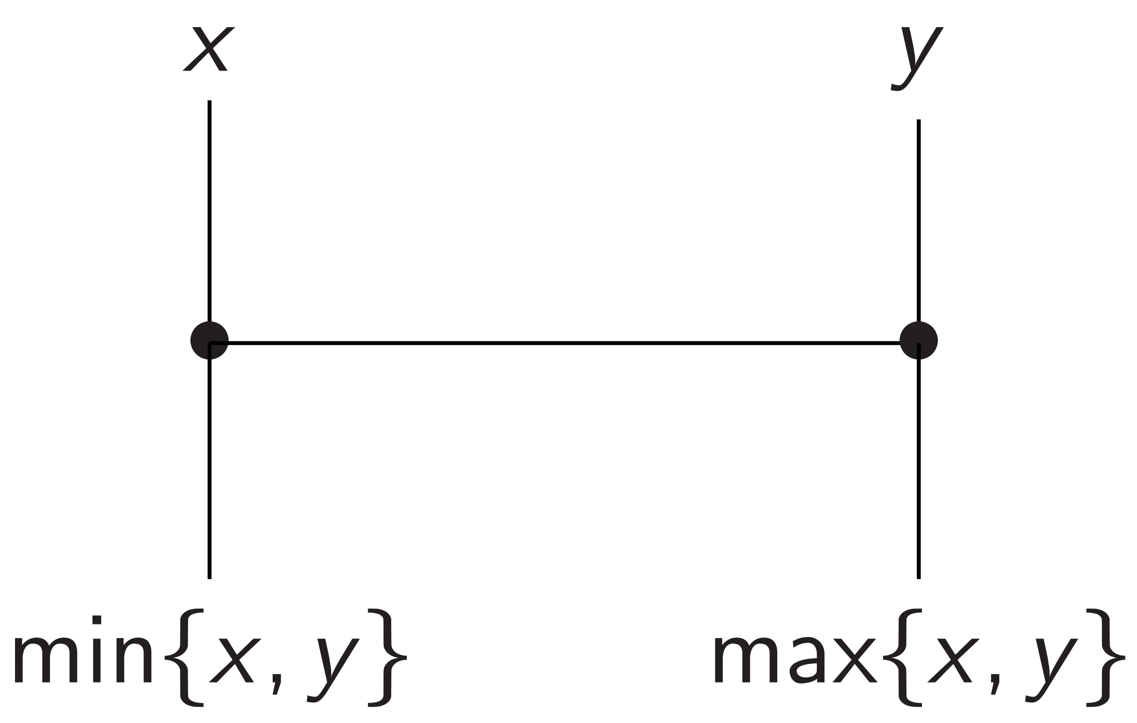
ly

data.

s?

Foundation of solution:

a **comparator** sorting 2 integers.



Easy constant-time exercise in C.

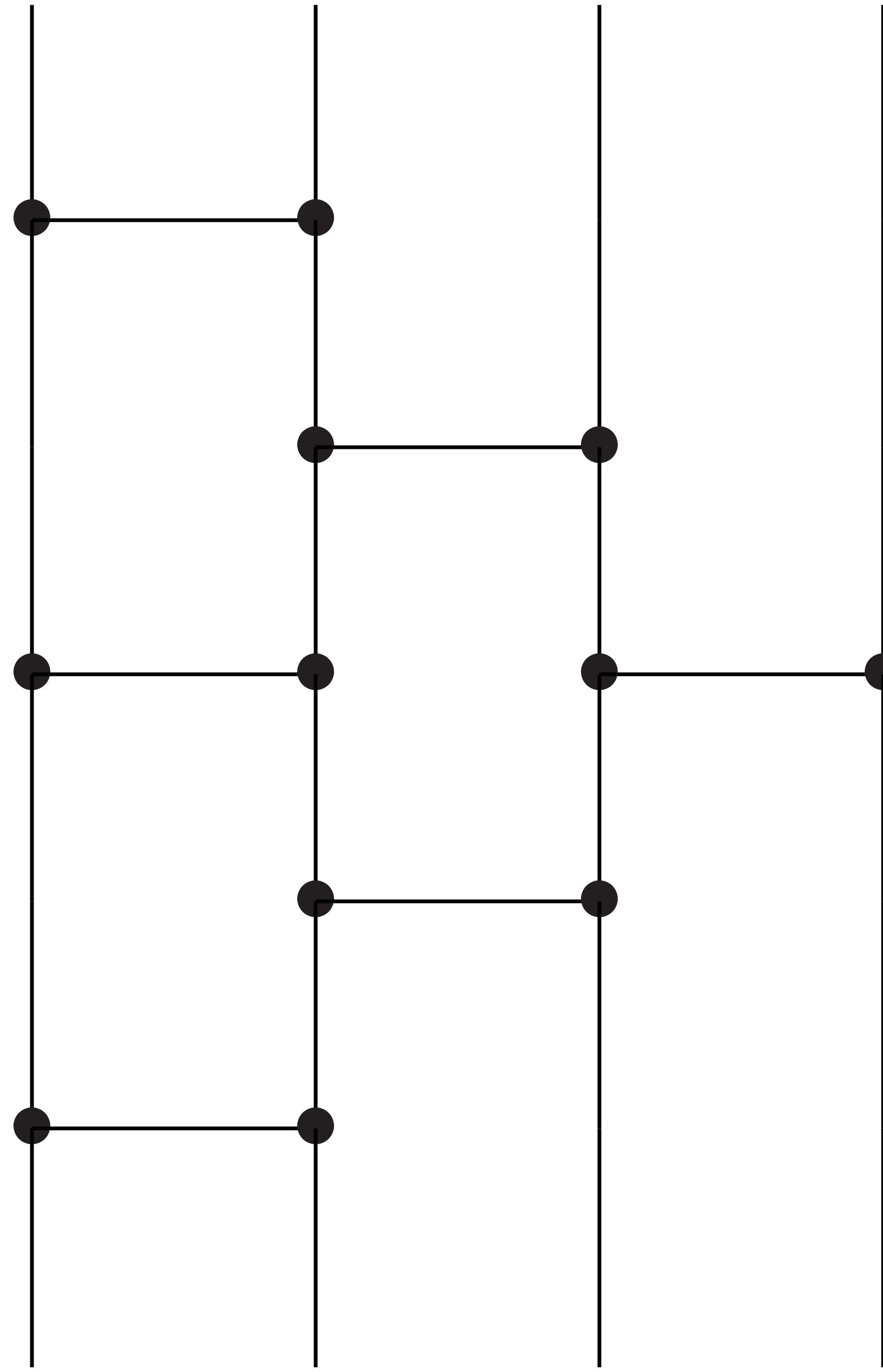
Warning: C standard allows compiler to break the solution.

Even easier exercise in asm.

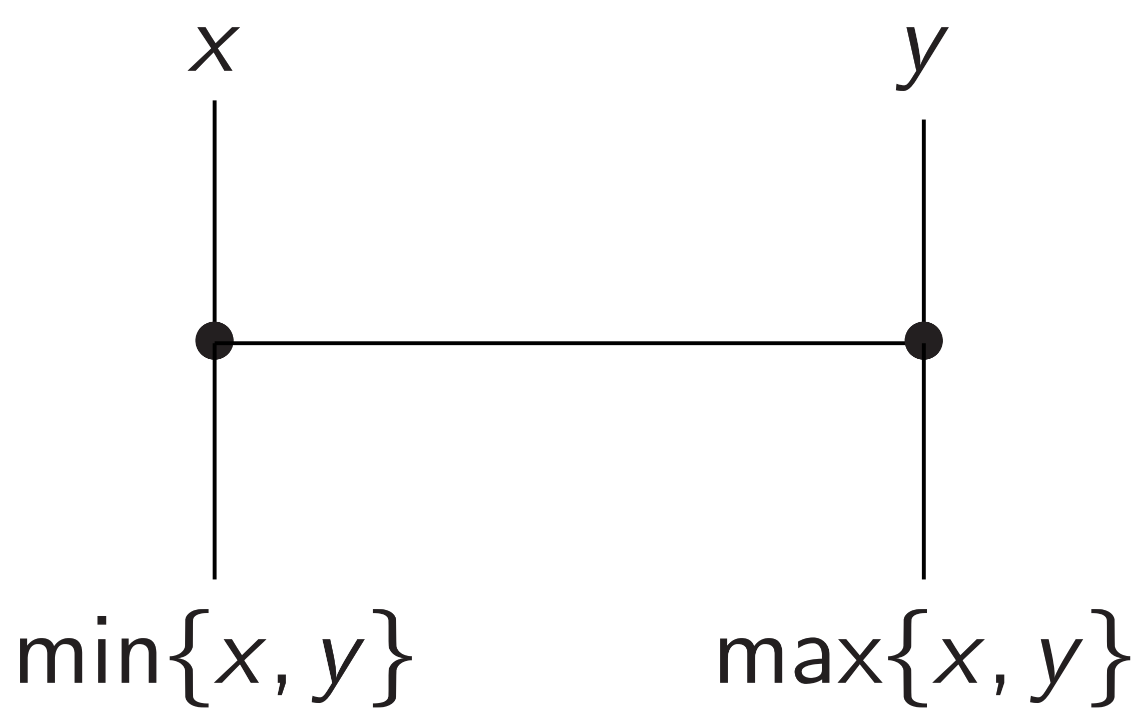
Combine comparators into a

sorting network for more in

Example of a sorting networ



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

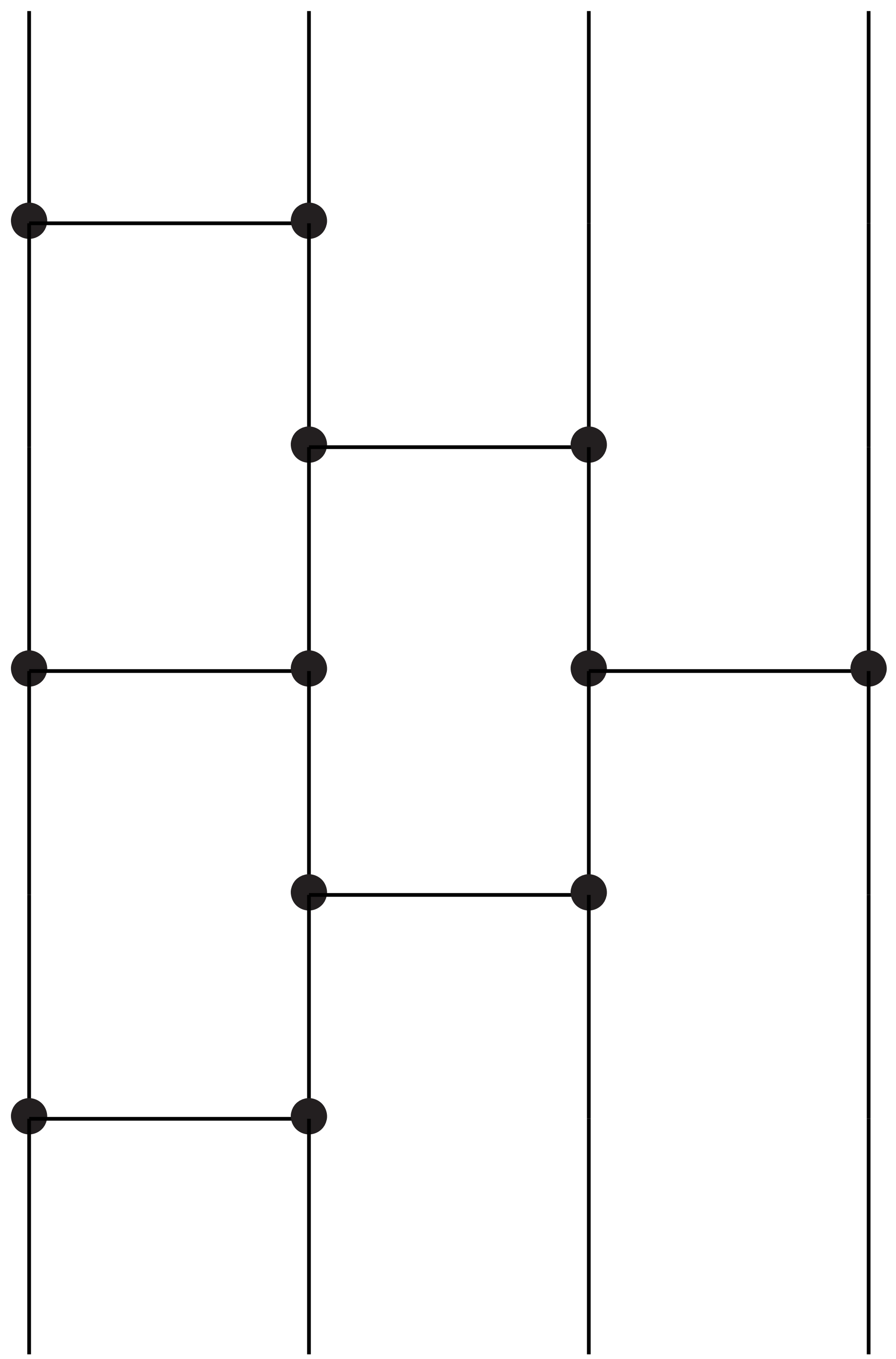


Easy constant-time exercise in C.
Warning: C standard allows compiler to break the solution.

Even easier exercise in asm.

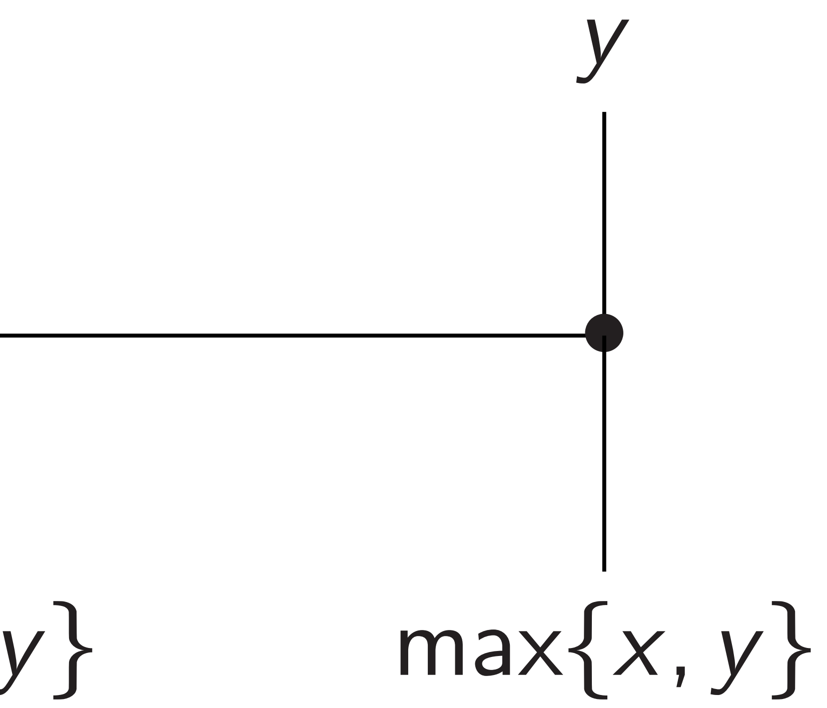
Combine comparators into a **sorting network** for more inputs.

Example of a sorting network:



ion of solution:

arator sorting 2 integers.



nstant-time exercise in C.

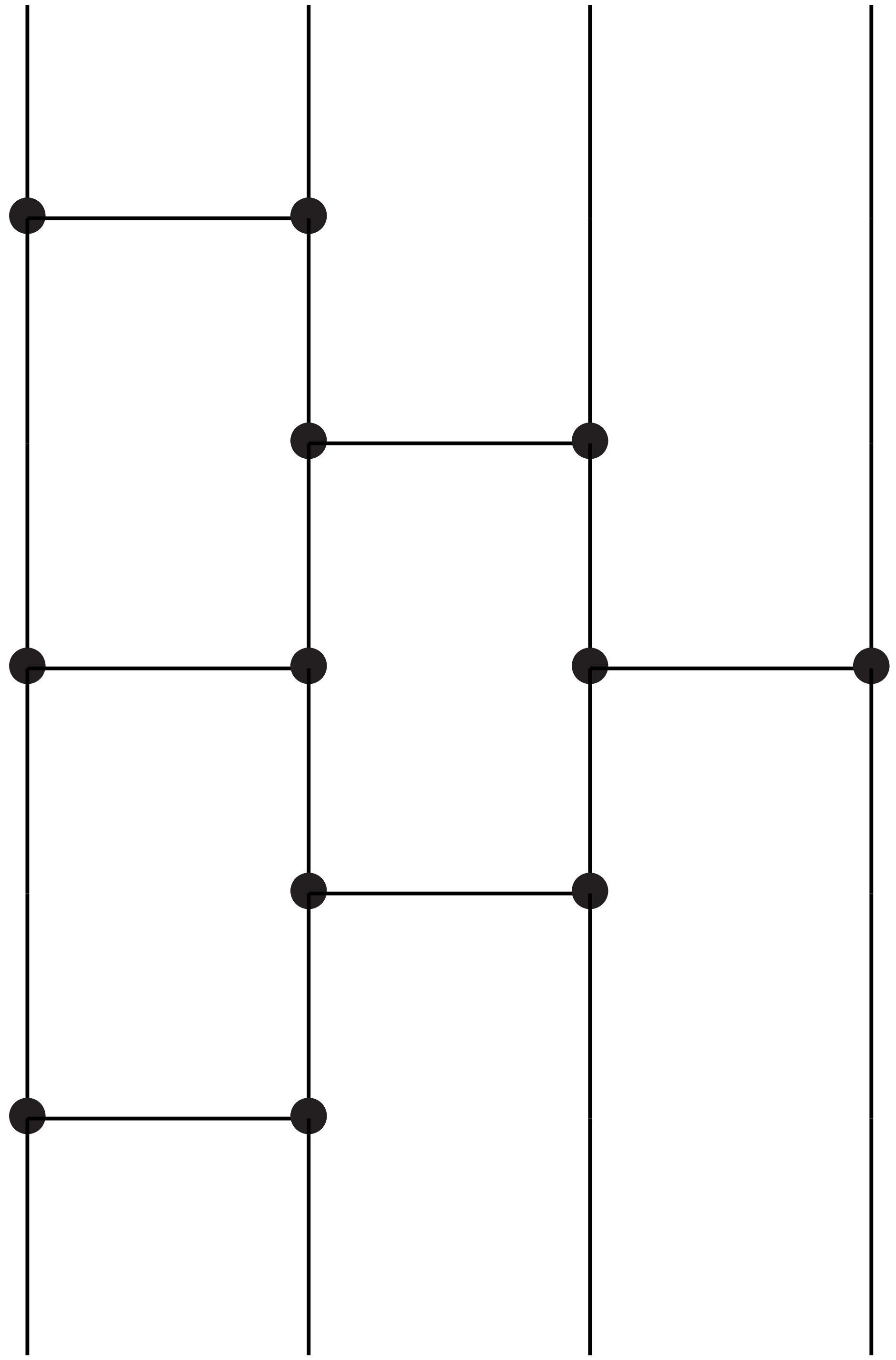
: C standard allows

r to break the solution.

sier exercise in asm.

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

Example of a sorting network:



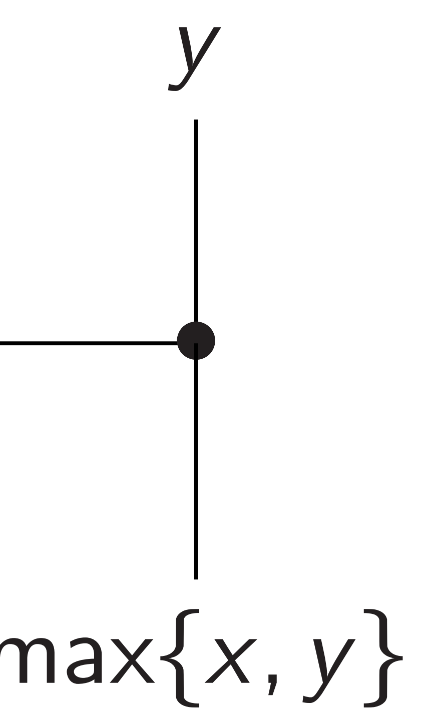
Position

in a sort

independ

Naturall

ution:
ting 2 integers.

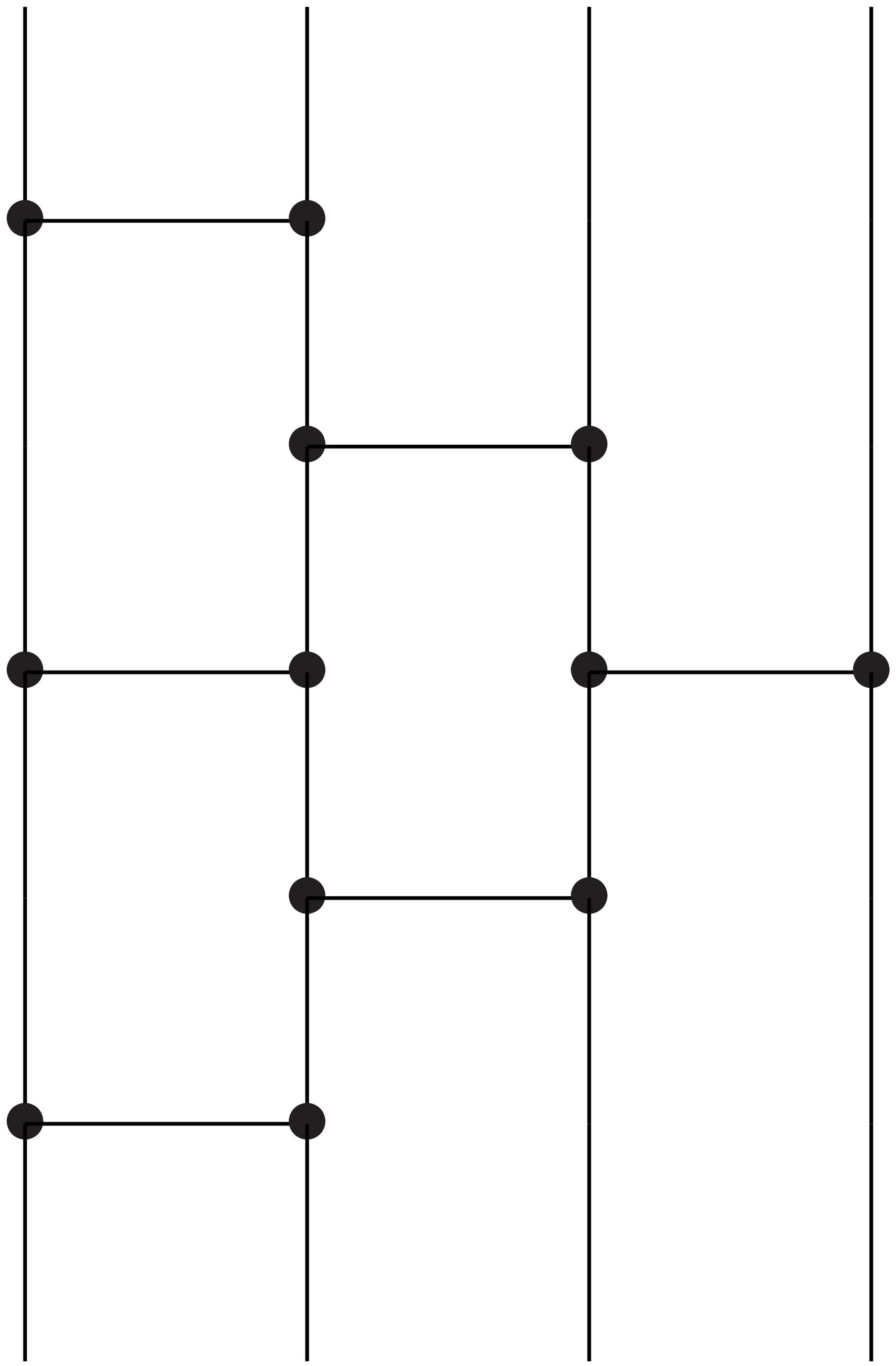


e exercise in C.
ard allows
the solution.

se in asm.

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

Example of a sorting network:



Positions of compa
in a sorting netwo
independent of the
Naturally constant

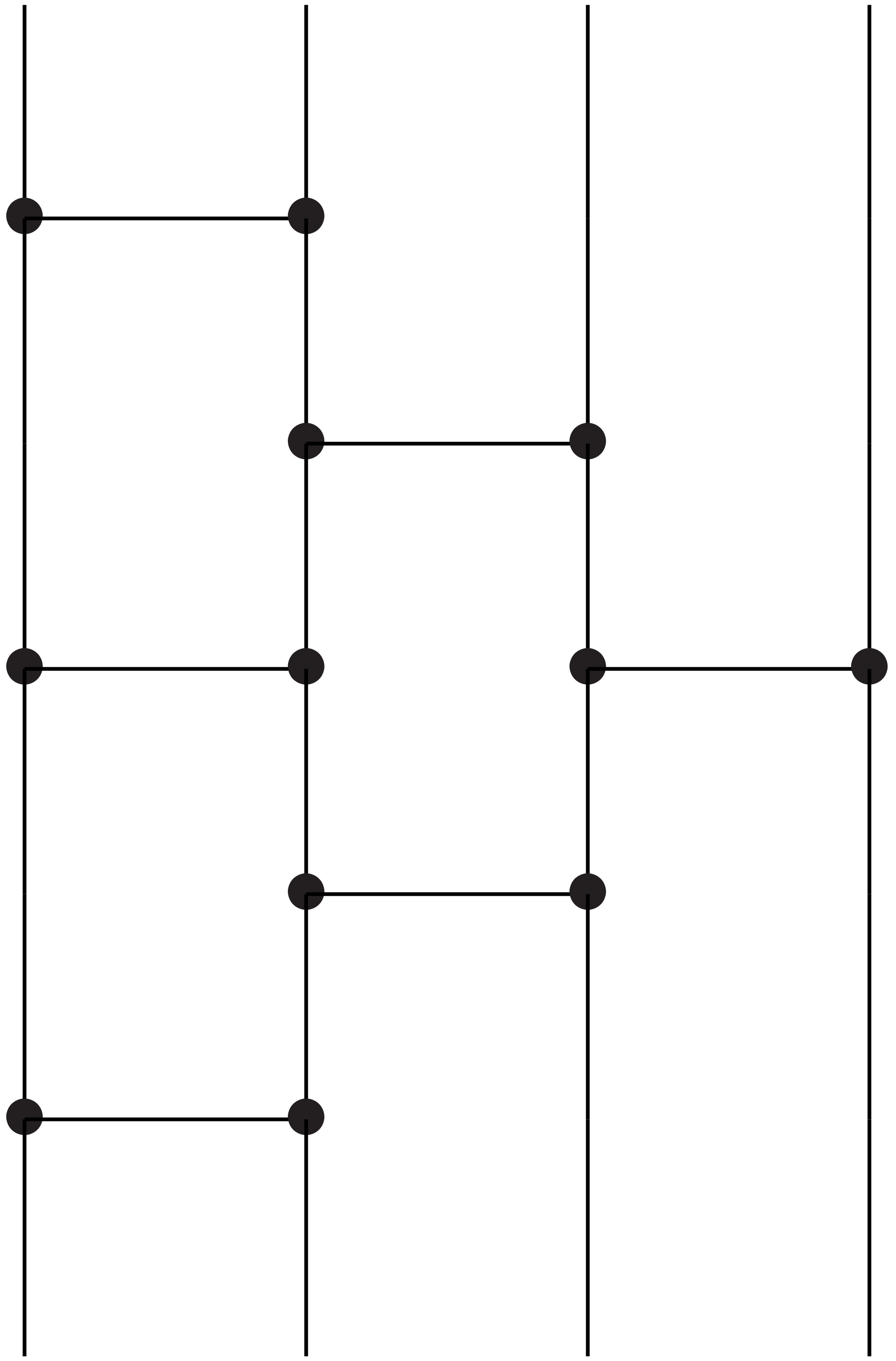
egers.

in C.

on.

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

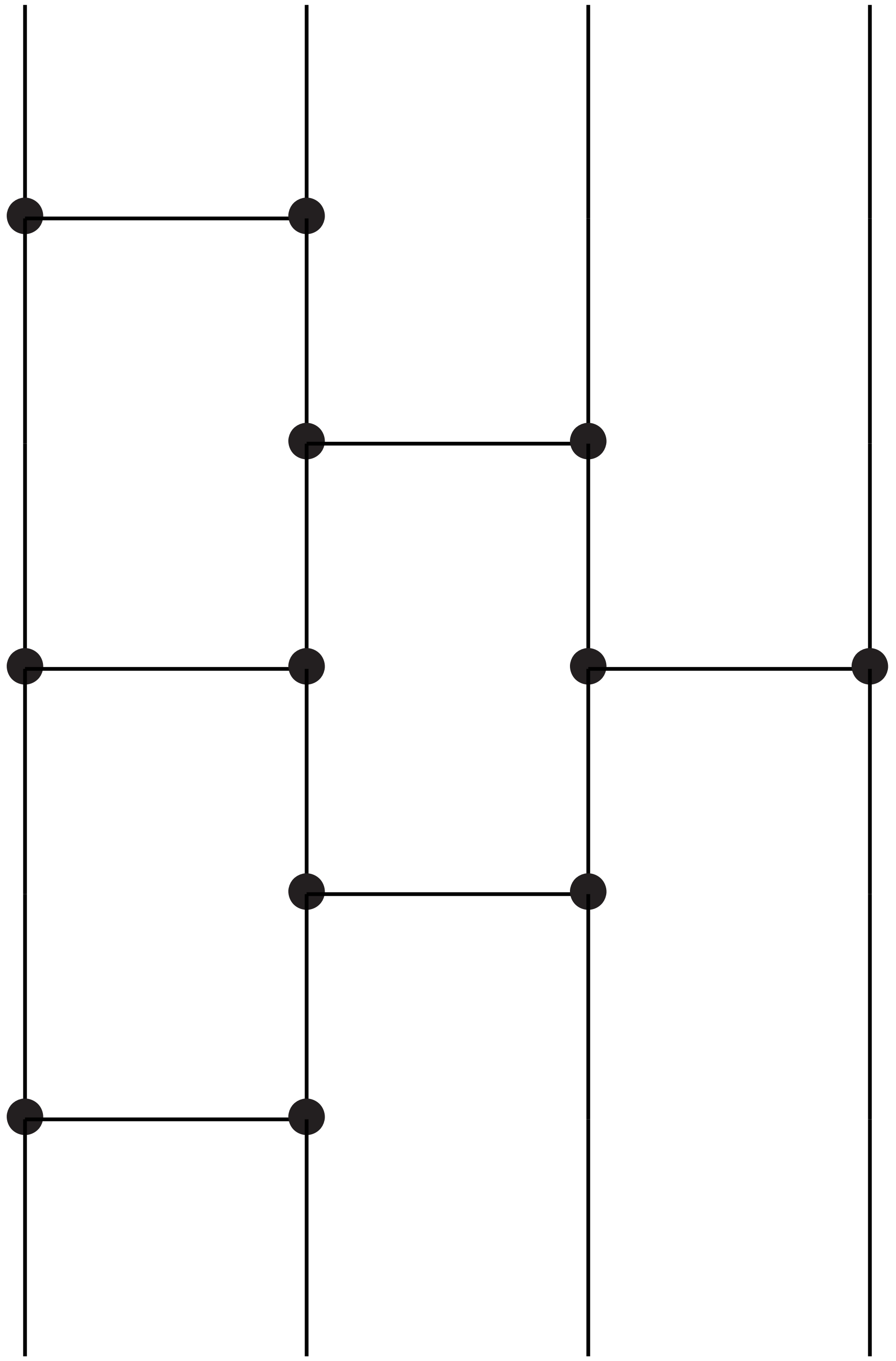
Example of a sorting network:



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

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

Example of a sorting network:

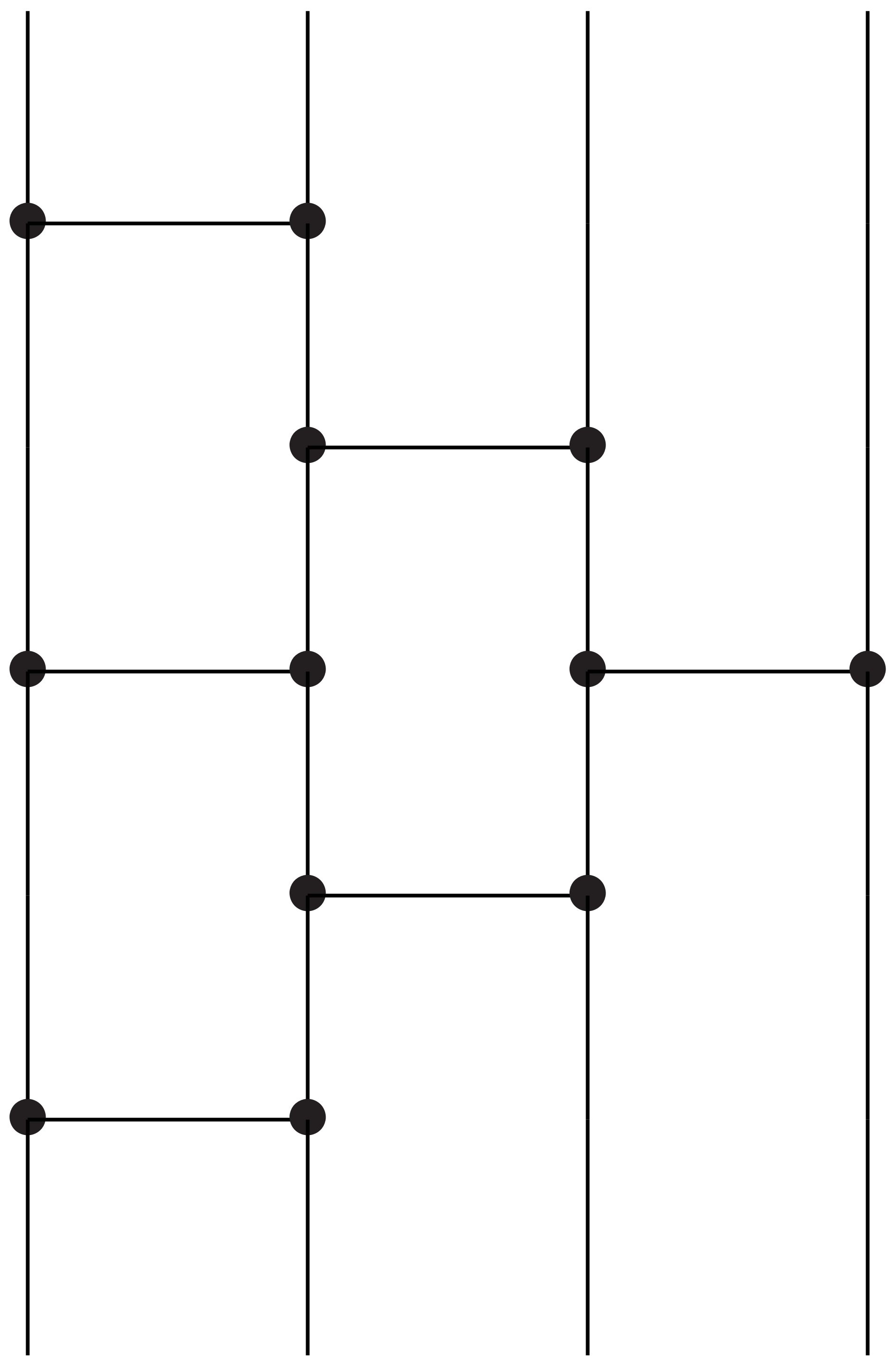


Positions of comparators in a sorting network are independent of the input.

Naturally constant-time.

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

Example of a sorting network:

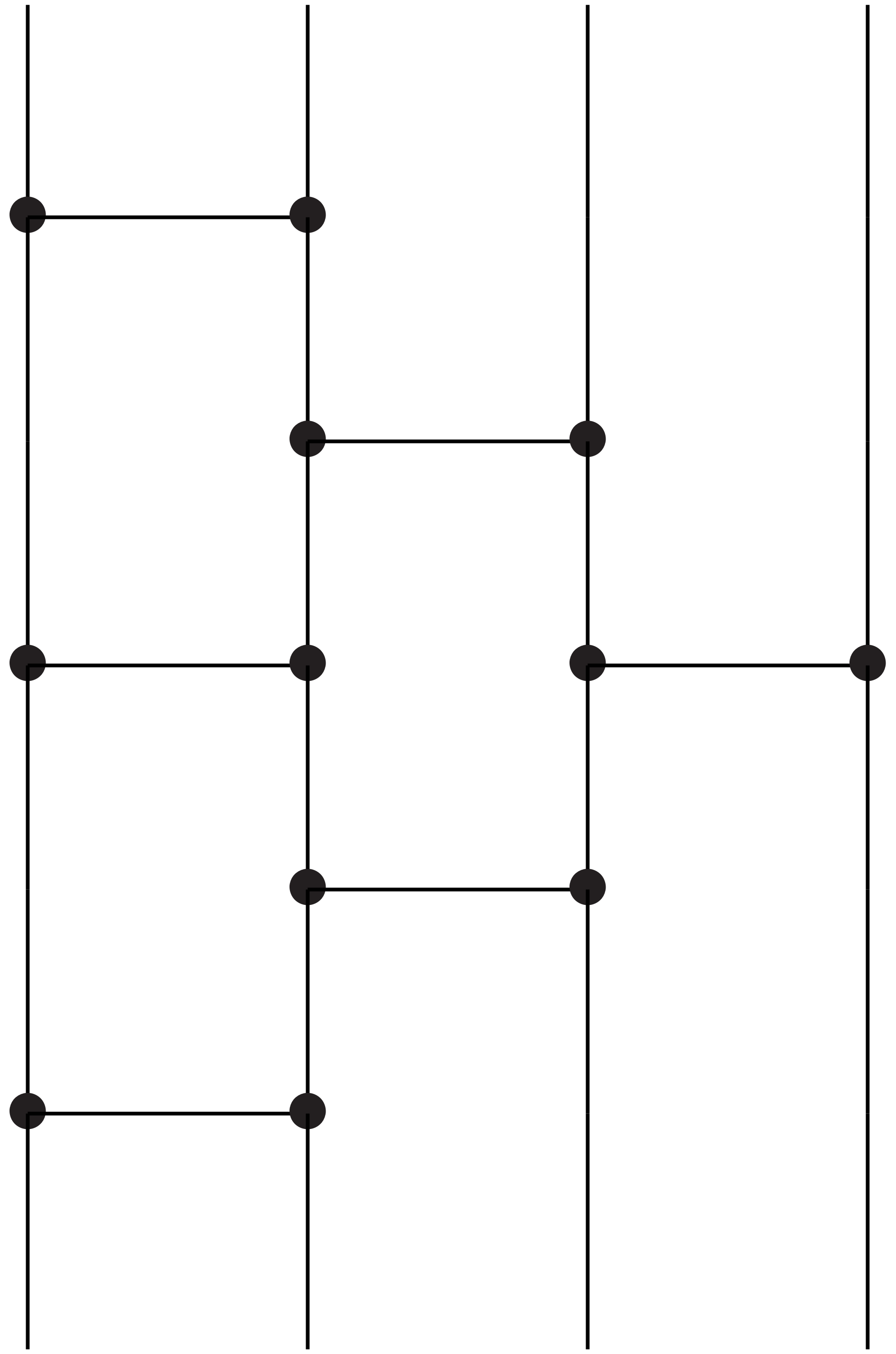


Positions of comparators in a sorting network are independent of the input. Naturally constant-time.

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.”

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

Example of a sorting network:



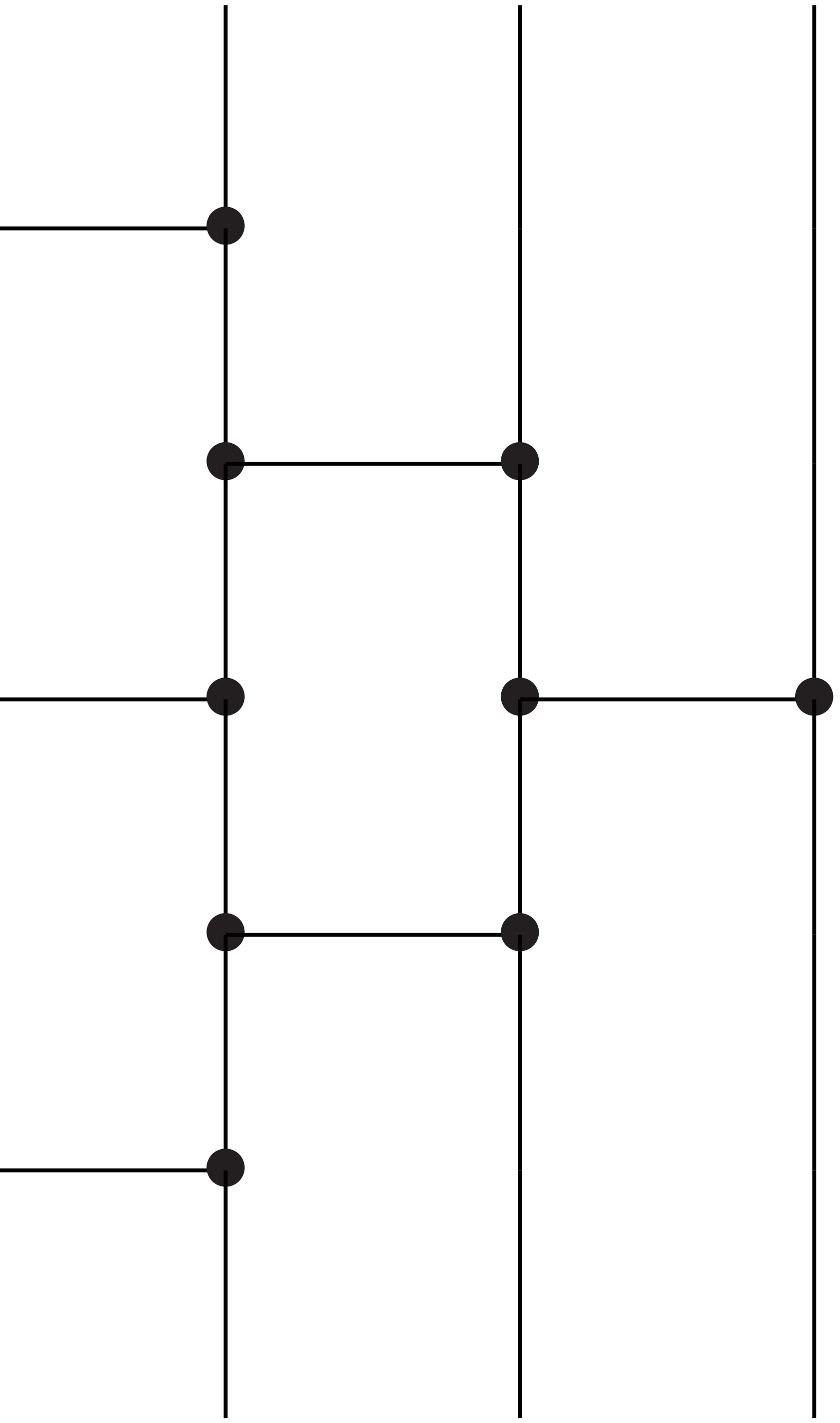
Positions of comparators in a sorting network are independent of the input. Naturally constant-time.

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.

e comparators into a **network** for more inputs.

e of a sorting network:



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

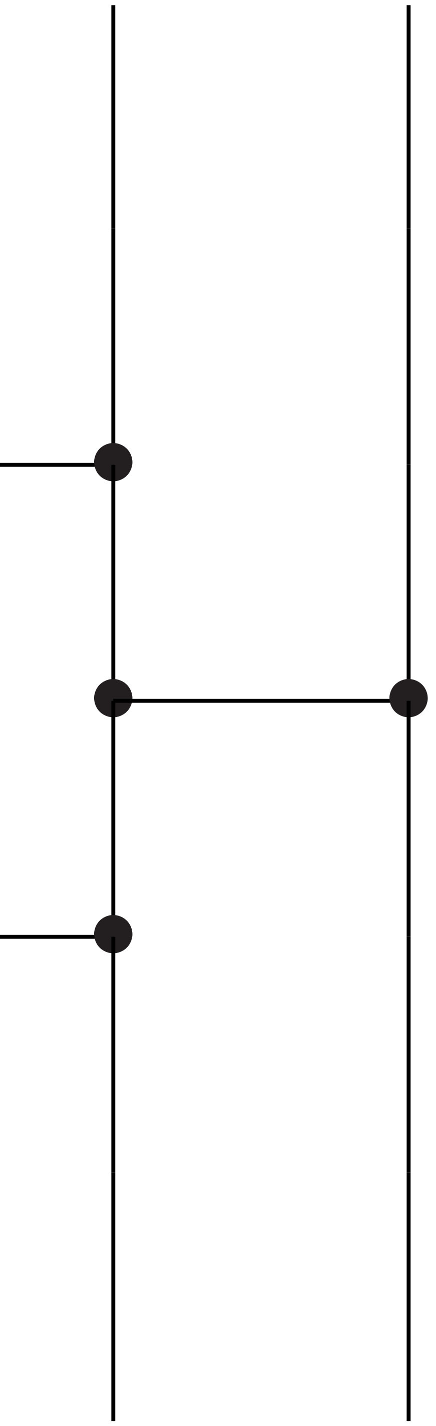
But remember all the people complaining about speed: e.g., “We would be happy to hear that fixed weight sampling is efficient on a variety of platforms . . . We have not yet been convinced that this is the case.”

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

```
void in
{ int64
  if (n
  t = 1
  while
  for (
    for
      i


    for
      f



  }
}
```

tors into a
for more inputs.

ng network:



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(
{ int64 t,p,q,i;
  if (n < 2) ret
  t = 1;
  while (t < n -
  for (p = t;p >
    for (i = 0;i
      if (!(i &
        minmax(x
    for (q = t;q
      for (i = 0
        if (!(i
          minmax
  }
}
```

n

nputs.

k:

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

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);
  }
}
```

s of comparators
ing network are
dent of the input.
y constant-time.

ember all the people
ning about speed: e.g.,
uld be happy to hear that
ight sampling is efficient
iety of platforms . . .
e not yet been convinced
s is the case.”

)/2 comparators in bubble
duce complaints about
ance 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
1973 Kn
which is
1968 Ba
sorting n

$\approx n(\log_2$
Much fa

Warning
of Batch
require *n*
Also, W
networks
handling

arators
rk are
e input.
t-time.

the people
t speed: e.g.,
ppy to hear that
ling is efficient
tforms . . .
been convinced
se.”

rators in bubble
plaints about
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 t
1973 Knuth “merg
which is a simplifie
1968 Batcher “odd
sorting networks.

$\approx n(\log_2 n)^2/4$ com
Much faster than

Warning: many ot
of Batcher's sortin
require n to be a p
Also, Wikipedia sa
networks . . . are n
handling arbitrarily

e
.g.,
r that
cient

nced

bubble
ut

```
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
1973 Knuth “merge exchang
which is a simplified version
1968 Batcher “odd-even me
sorting networks.

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

Warning: many other descri
of Batcher’s sorting network
require n to be a power of 2
Also, Wikipedia says “Sortin
networks . . . are not capable
handling arbitrarily large inp

```
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."

```
t32_sort(int32 *x,int64 n)
 t,p,q,i;
 < 2) return;
;
 (t < n - t) t += t;
p = t;p > 0;p >>= 1) {
 (i = 0;i < n - p;++i)
f (!(i & p))
 minmax(x+i,x+i+p);
 (q = t;q > p;q >>= 1)
or (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 co

Cons

Berns

La

“NTRU

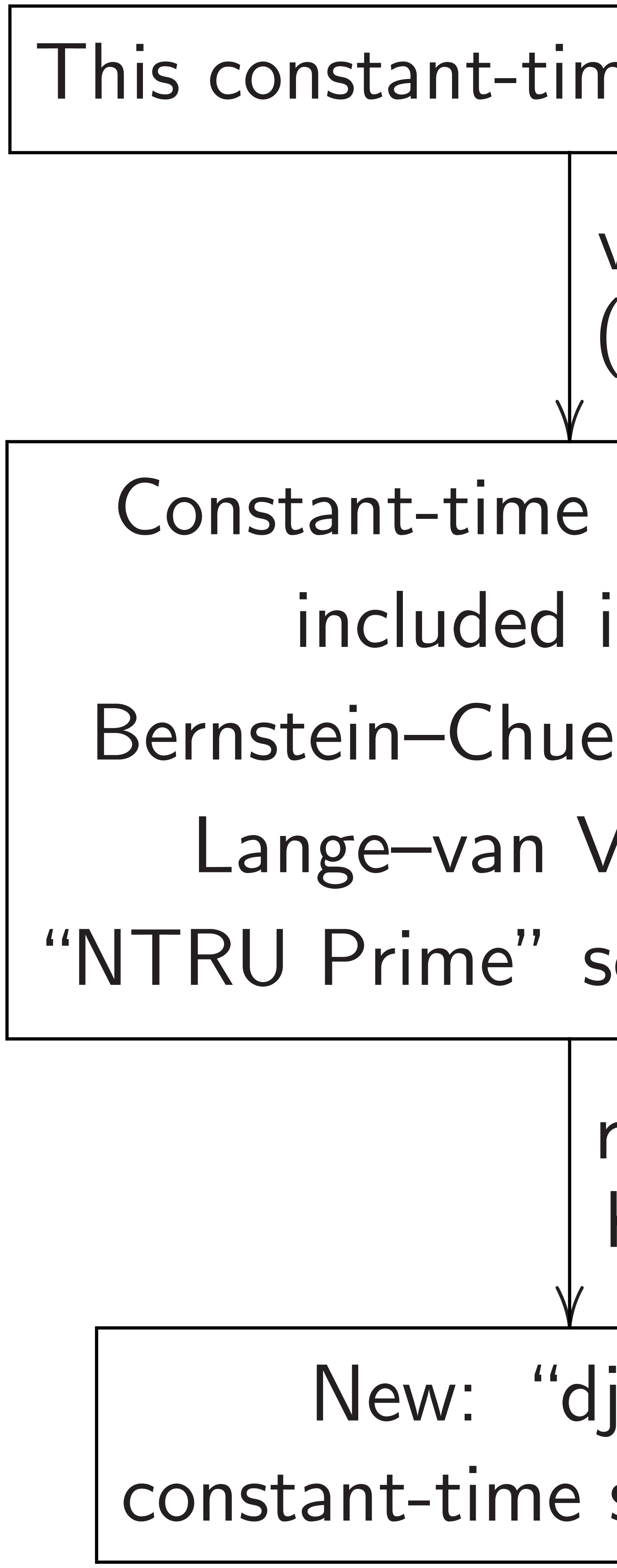
const

```
int32 *x,int64 n)

urn;

 t) t += t;
 0;p >>= 1) {
 < n - p;++i)
p))
+i,x+i+p);
 > p;q >>= 1)
;i < n - q;++i)
& p))
(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-tim

v
(

Constant-time
included i
Bernstein–Chue
Lange–van V
“NTRU Prime” s

r
I

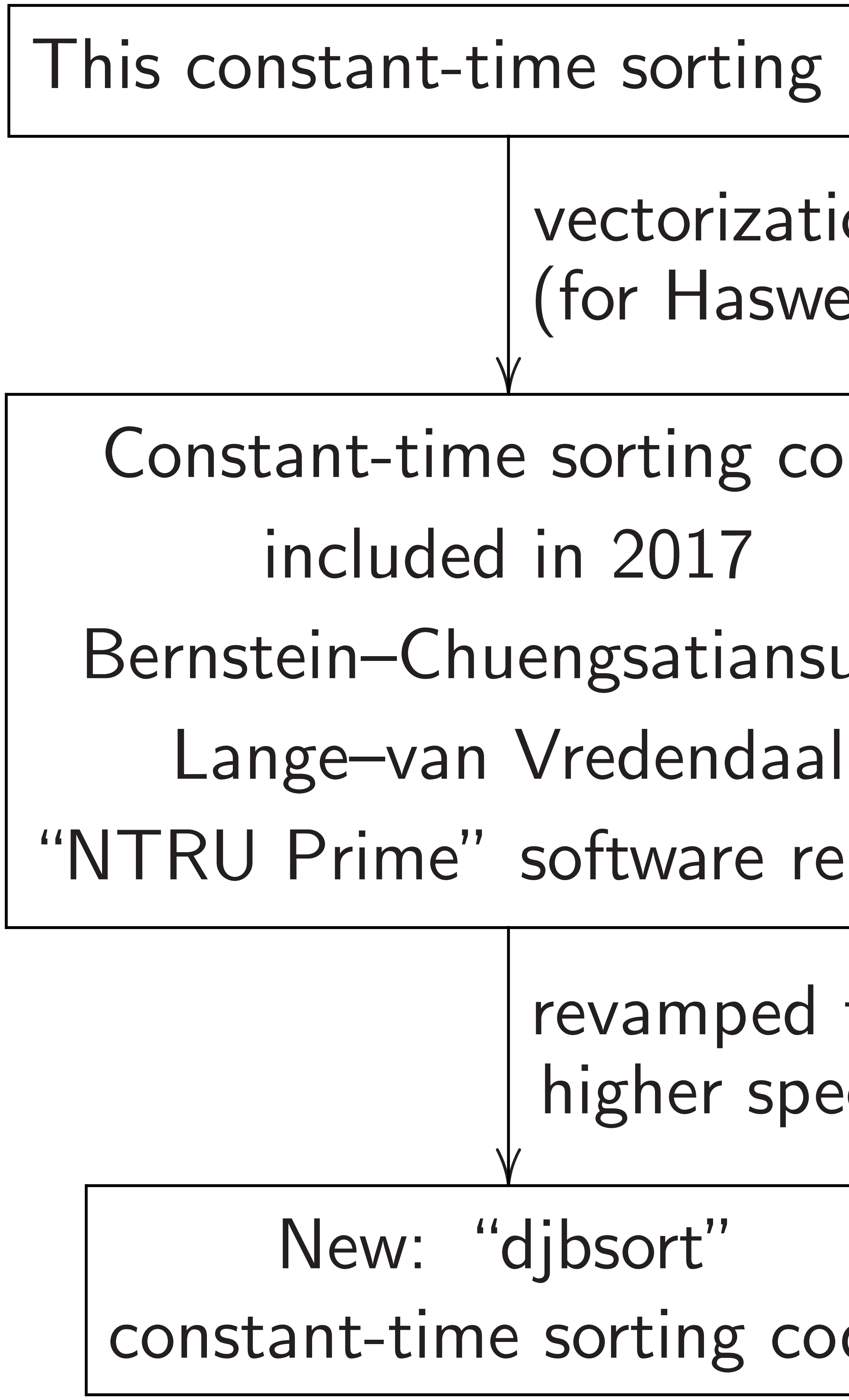
New: “dj
constant-time s

```
int64 n)



t;
1) {
++i)


;
= 1)
q;++i)

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

vectorizati
(for Haswe

Constant-time sorting co
included in 2017
Bernstein–Chuengsatiansu
Lange–van Vredendaal
"NTRU Prime" software re

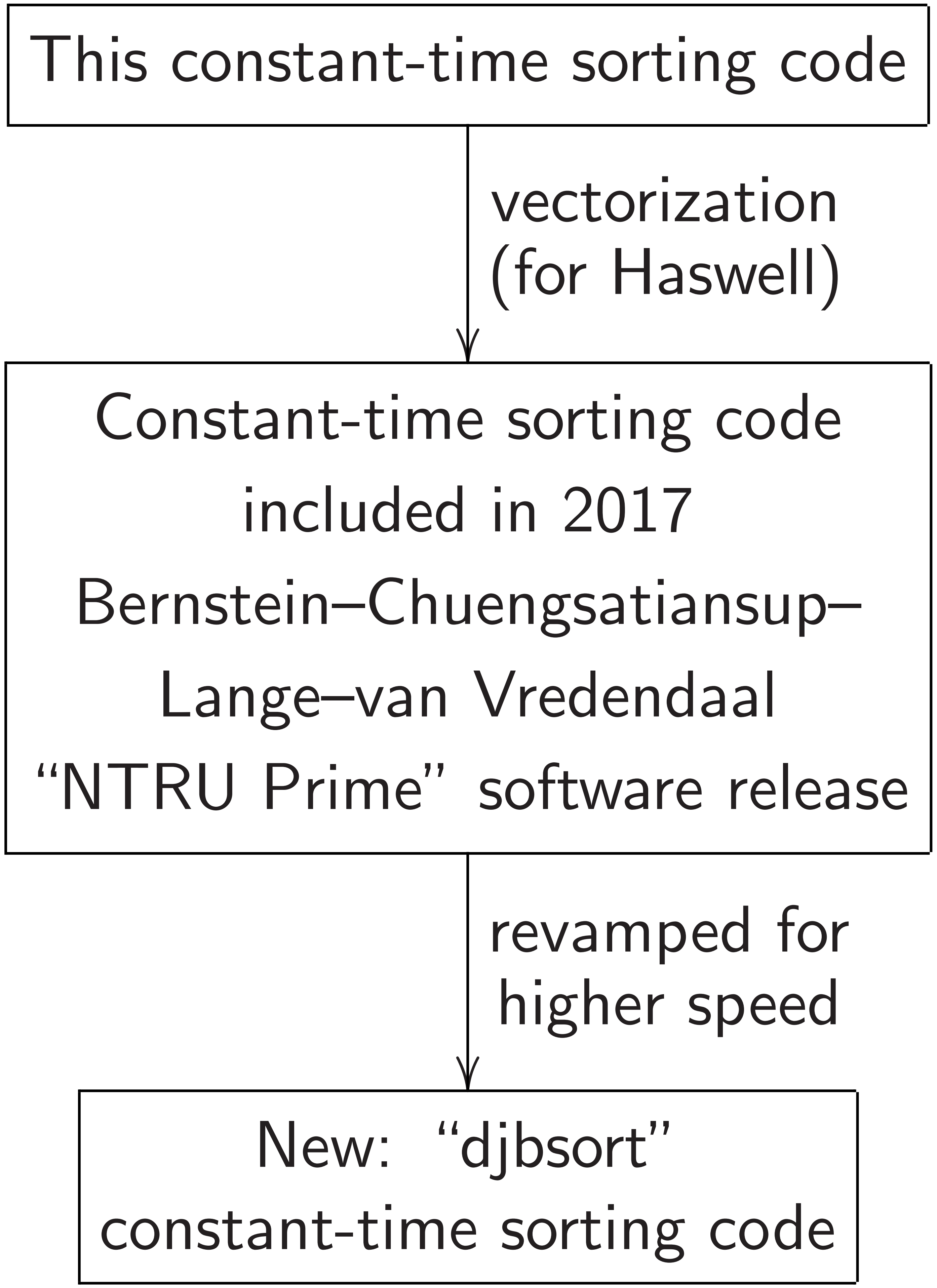
revamped
higher spe

New: "djbsort"
constant-time sorting co

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

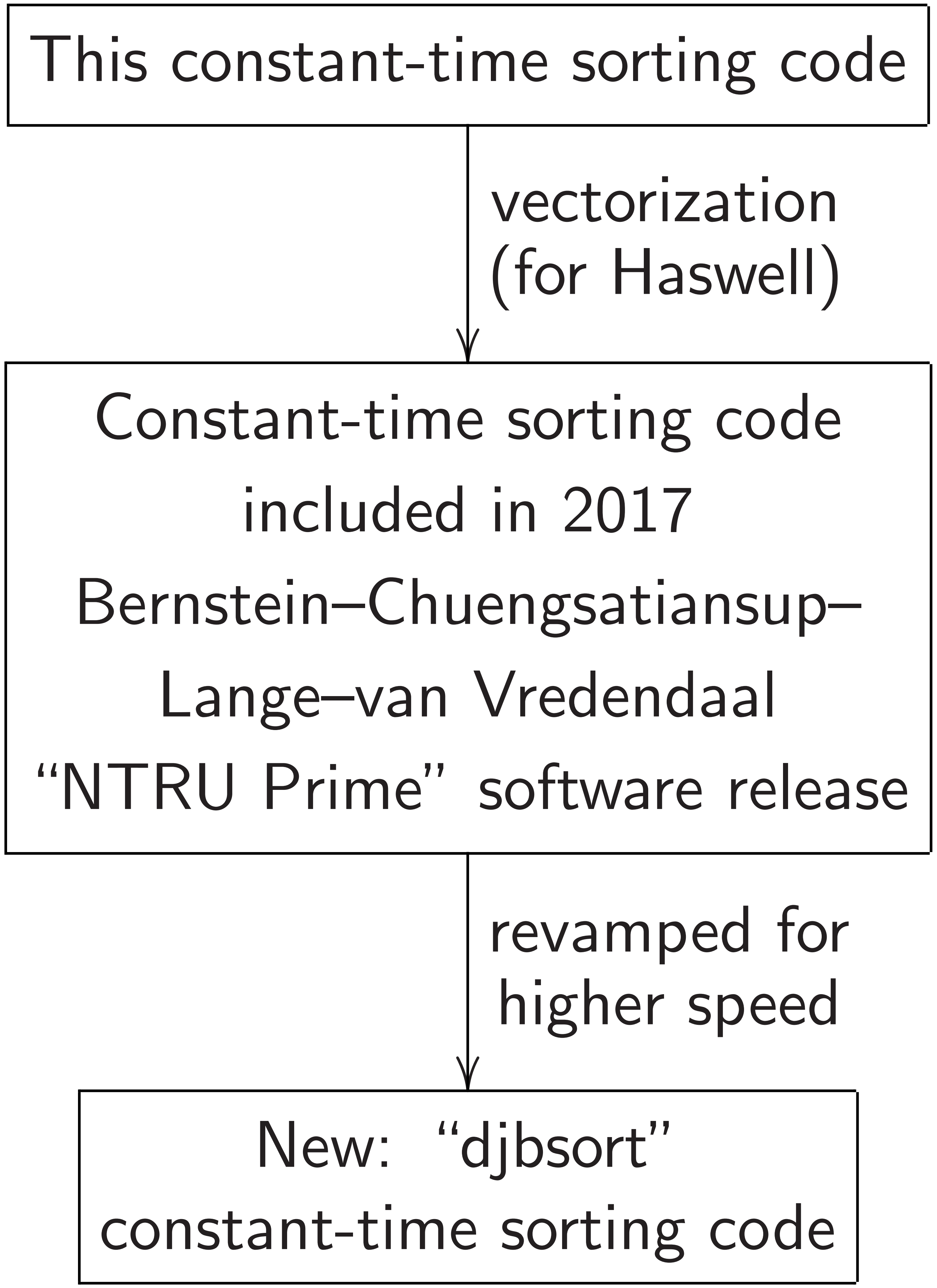
revamped for higher speed

New: "djbsort" constant-time sorting code

s slide: C translation of
nuth "merge exchange",
a simplified version of
tcher "odd-even merge"
networks.

$n)^2/4$ comparators.
ster than bubble sort.

: many other descriptions
ner's sorting networks
n to be a power of 2.
ikipedia says "Sorting
s . . . are not capable of
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 slow

Massive
Includes
sorting u
on mode
2015 Gu

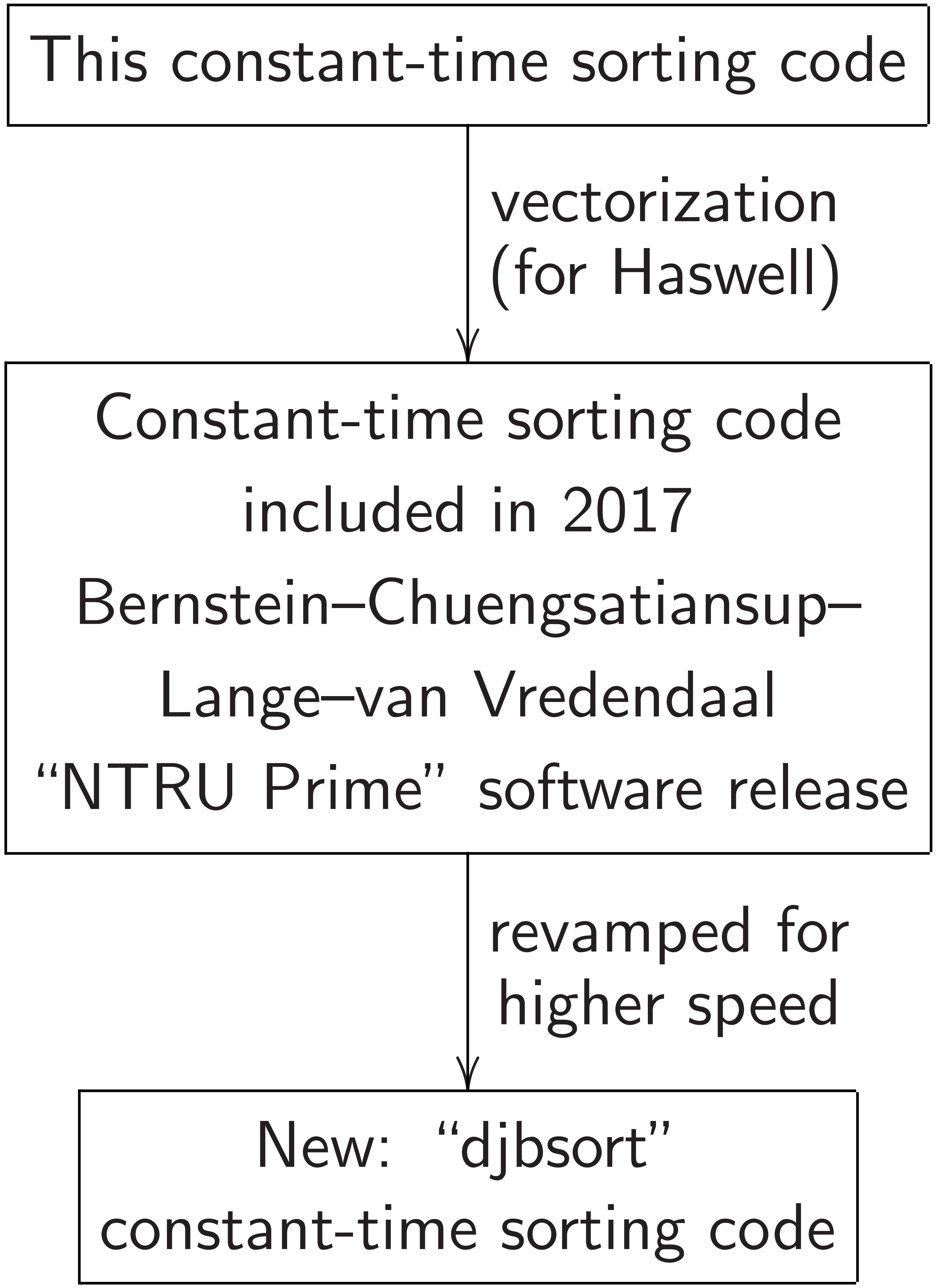
Haswell
25608 s
21844 h

15136 k

translation of
ge exchange",
ed version of
d-even merge"

nparators.
bubble sort.

ther descriptions
g networks
power of 2.
ys "Sorting
not capable of
y 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

Massive fast-sortir
Includes several ef
sorting using AVX
on modern Intel C
2015 Gueron–Kras

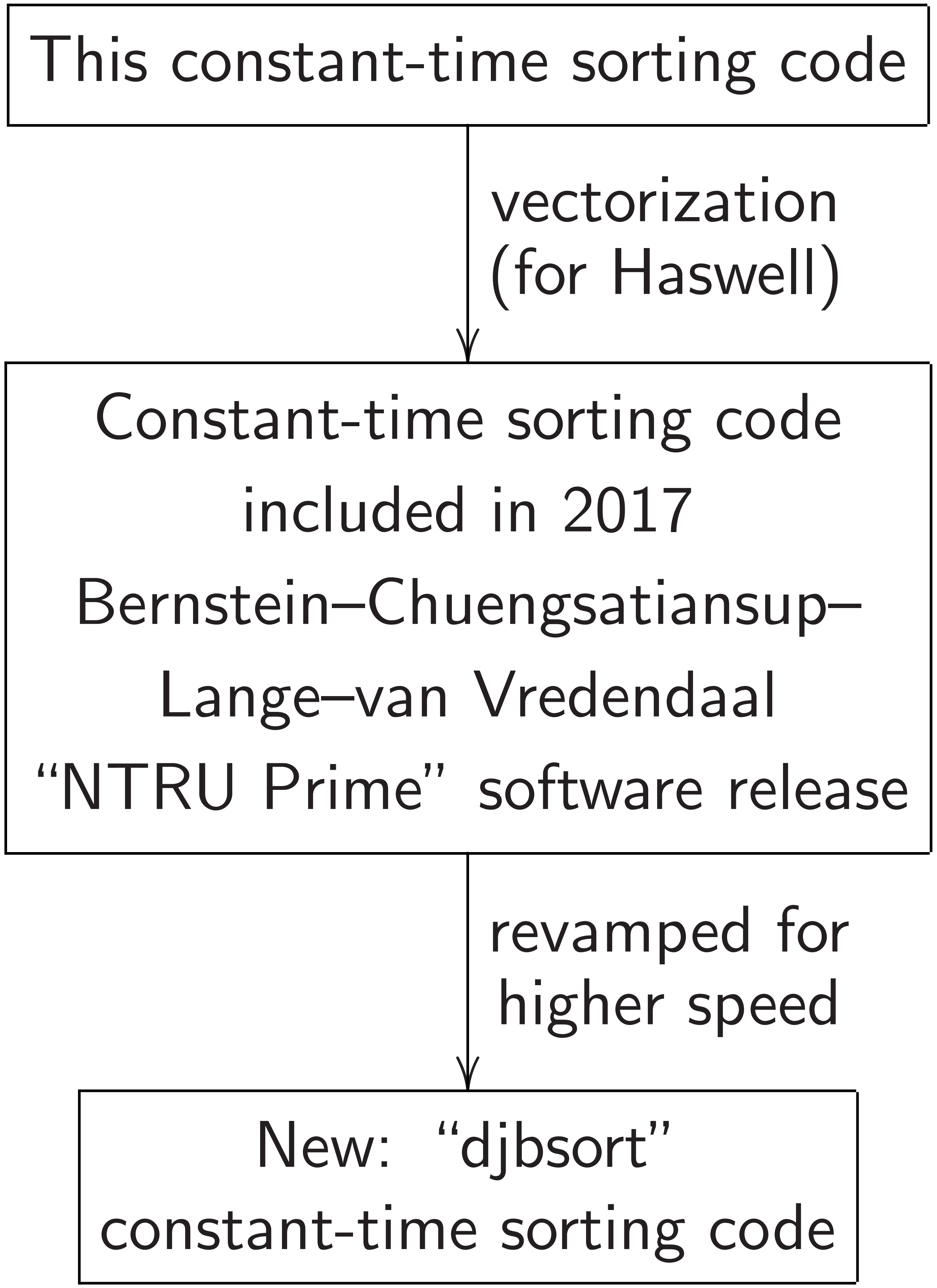
Haswell (titan0)
25608 stdsort
21844 herf

15136 krasnov

n of
ge",
of
rge"

rt.

ptions
s
.
ng
e of
uts."



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

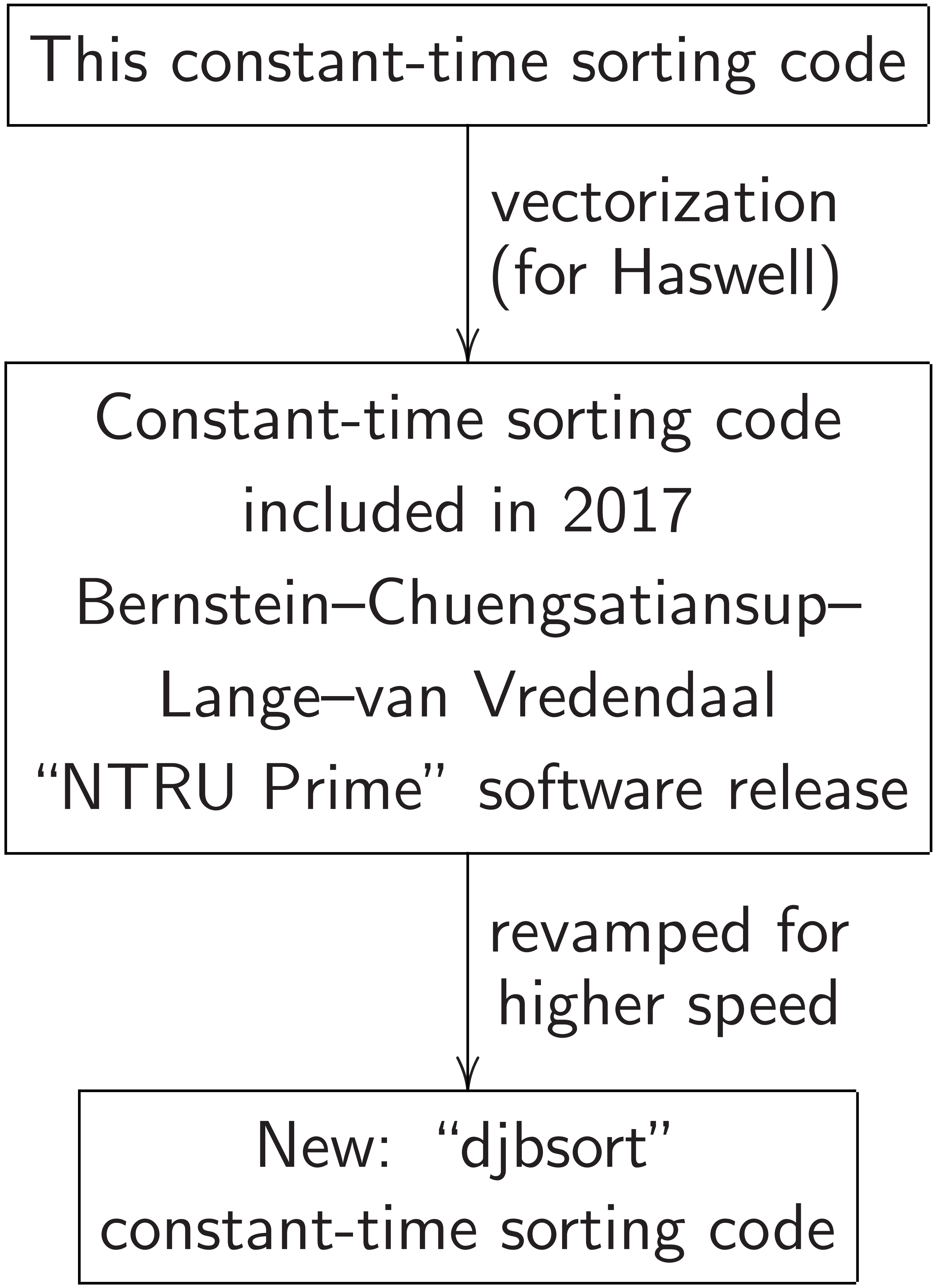
New: "djbsort" constant-time sorting code

The slowdown for constant t

Massive fast-sorting literatur
Includes several efforts to op
sorting using AVX2 instructi
on modern Intel CPUs: e.g.
2015 Gueron–Krasnov quick

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

15136 `krasnov`



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

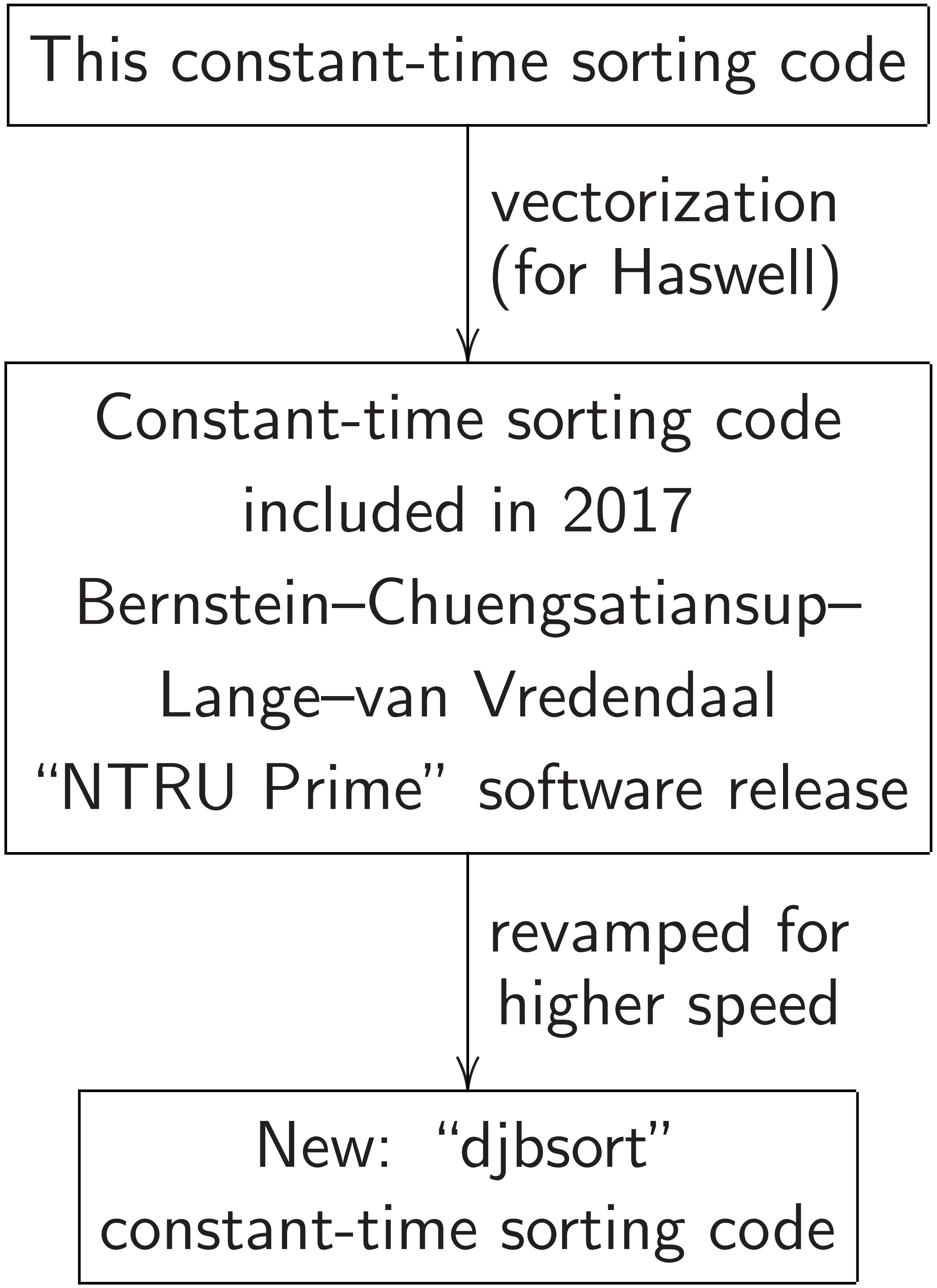
The slowdown for constant time

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`



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.

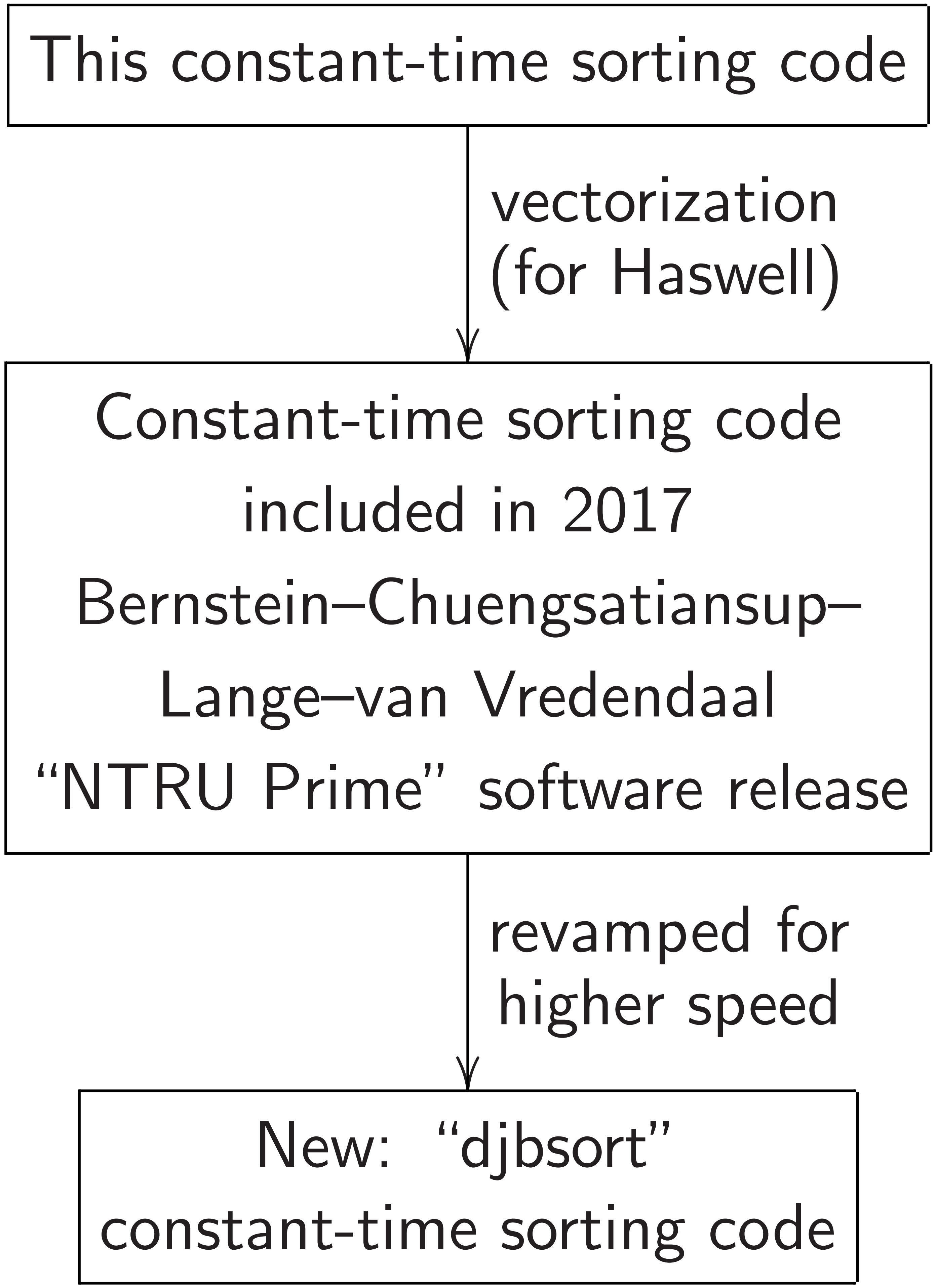
Haswell (`titan0`) cycles, $n = 768$:

25608 `stdsort`

21844 `herf`

18548 `oldavx2` (2017 BCLvV)

15136 `krasnov`



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

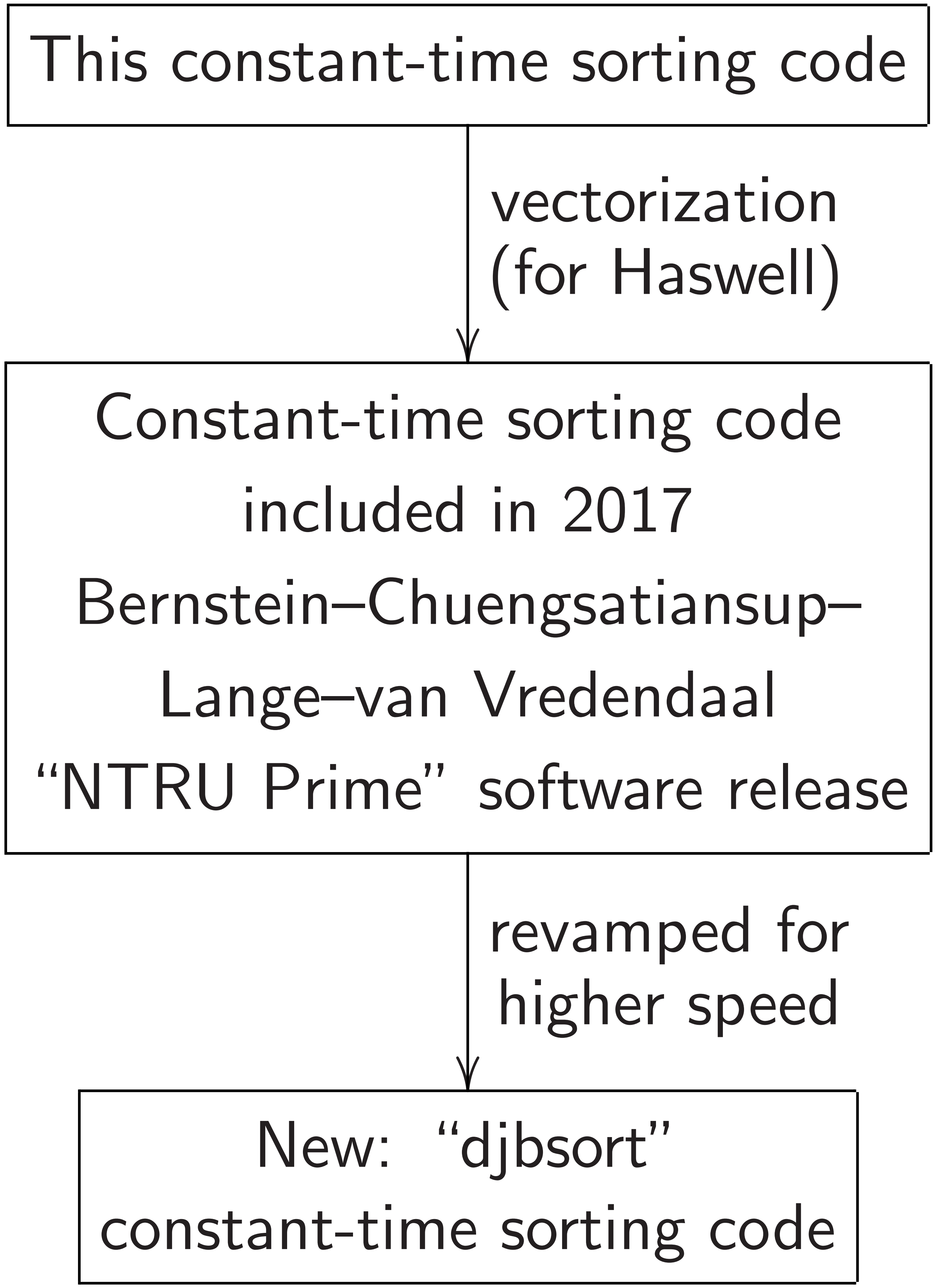
New: "djbsort" constant-time sorting code

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`
18548	`oldavx2` (2017 BCLvV)
15136	`krasnov`
6596	`avx2` (2018 djbsort)



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

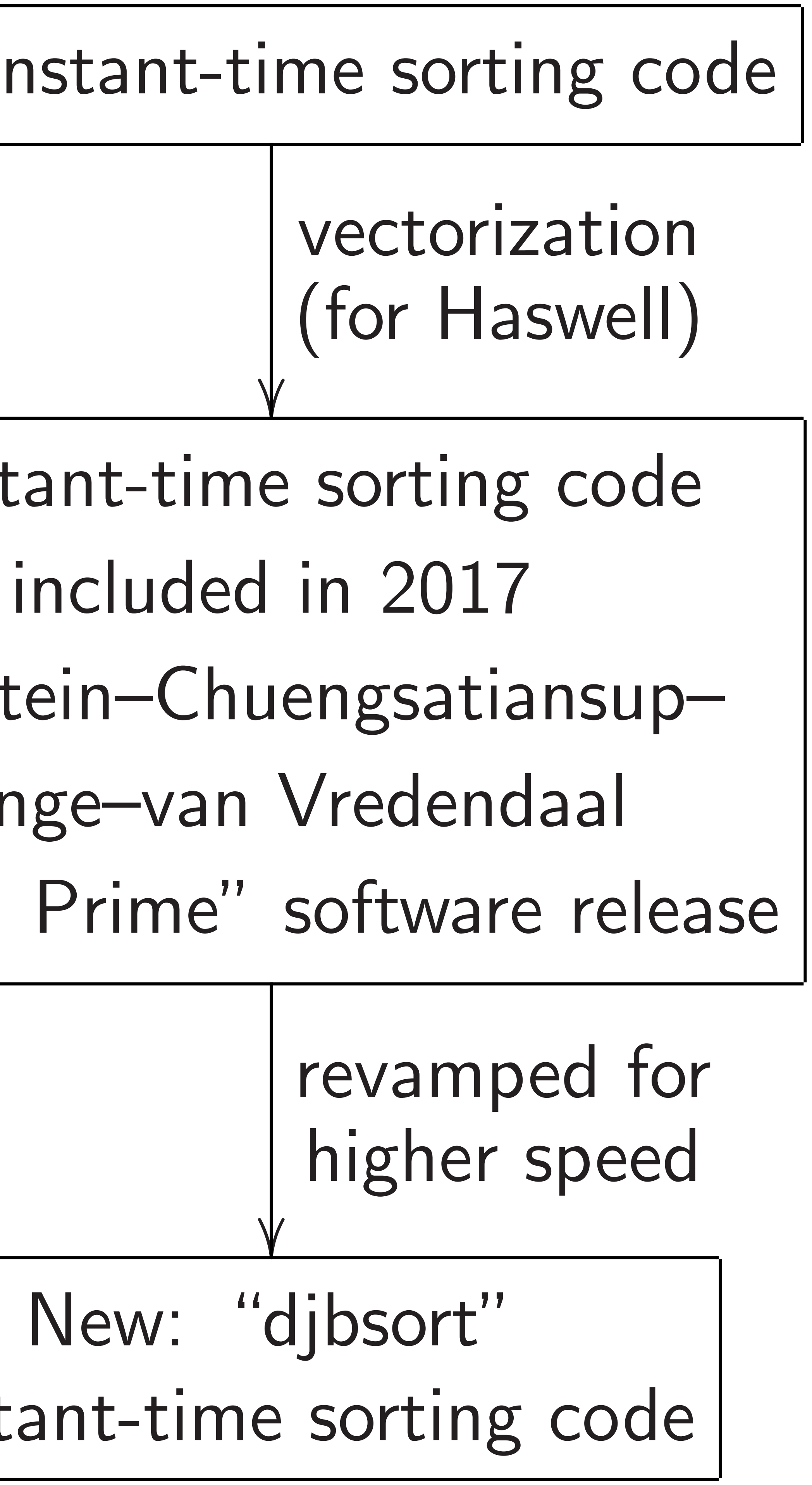
The slowdown for constant time

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`
18548	`oldavx2` (2017 BCLvV)
15136	`krasnov`
6596	`avx2` (2018 djbsort)

No slowdown. New speed records!



nstant-time sorting code

vectorization
(for Haswell)

tant-time sorting code
included in 2017
tein–Chuengsatiansup–
nge–van Vredendaal
Prime” software release

revamped for
higher speed

New: “djbsort”
tant-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
18548 oldavx2 (2017 BCLvV)
15136 krasnov
 6596 avx2 (2018 djbsort)
```

No slowdown. New speed records!

How car
beat sta

ne sorting code

vectorization
(for Haswell)

sorting code
n 2017
ngsatiansup–
Vredendaal
oftware release

revamped for
higher speed

bsort"
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`
18548	`oldavx2` (2017 BCLvV)
15136	`krasnov`
6596	`avx2` (2018 djbsort)

No slowdown. New speed records!

How can an n(log
beat standard n lo

code

on
ll)

de

up–

lease

for
ed

de

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 $n(\log n)^2$ algori
beat standard $n \log n$ algorit

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?

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?

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

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?

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.

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?

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.

vdown for constant time

fast-sorting literature.

several efforts to optimize

using AVX2 instructions

ern Intel CPUs: e.g.

ueron–Krasnov quicksort.

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

`tdsort`

`erf`

`ldavx2` (2017 BCLvV)

`rasnov`

`vx2` (2018 djbsort)

down. New speed records!

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.

Verificat

Sorting s

Does it w

Test the

random

decreasi

constant time

ng literature.

forts to optimize

2 instructions

PUs: e.g.

snov quicksort.

cycles, $n = 768$:

2017 BCLvV)

djbsort)

w speed records!

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

Does it work corre

Test the sorting so

random inputs, inc

decreasing inputs.

time

re.

ptimize

ons

sort.

$= 768$:

V)

ecords!

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 TC

Does it work correctly?

Test the sorting software on

random inputs, increasing in

decreasing inputs. Seems to

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.

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.

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

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

n an $n(\log n)^2$ algorithm
ndard $n \log n$ algorithms?

well-known trends
design, reflecting
ental hardware costs
us operations.

ycle, Haswell core can do
ops on 32-bit integers +
ops on 32-bit integers.

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

onal 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.

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

ma

fully

unrolled

yes,

$n)^2$ algorithm

g n algorithms?

wn trends

flecting

ware costs

ons.

ell core can do

2-bit integers +

2-bit integers.

nteger from a

much slower.

n: much slower.

Verification

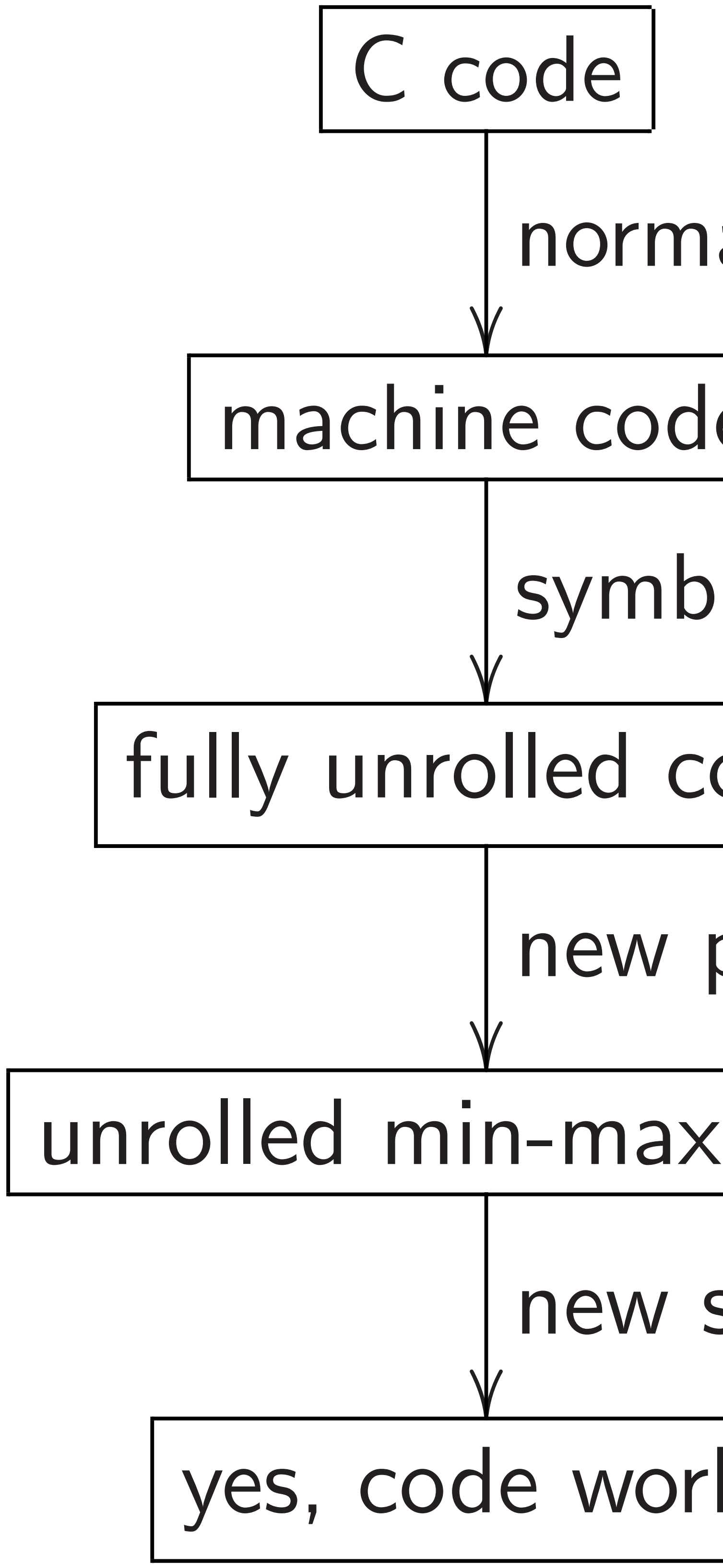
Sorting software is in the TCB.
Does it work correctly?

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

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



C code

norm

machine cod

symb

fully unrolled c

new p

unrolled min-max

new s

yes, code wor

thm
hms?

n do
ers +
ers.

m a
er.

lower.

Verification

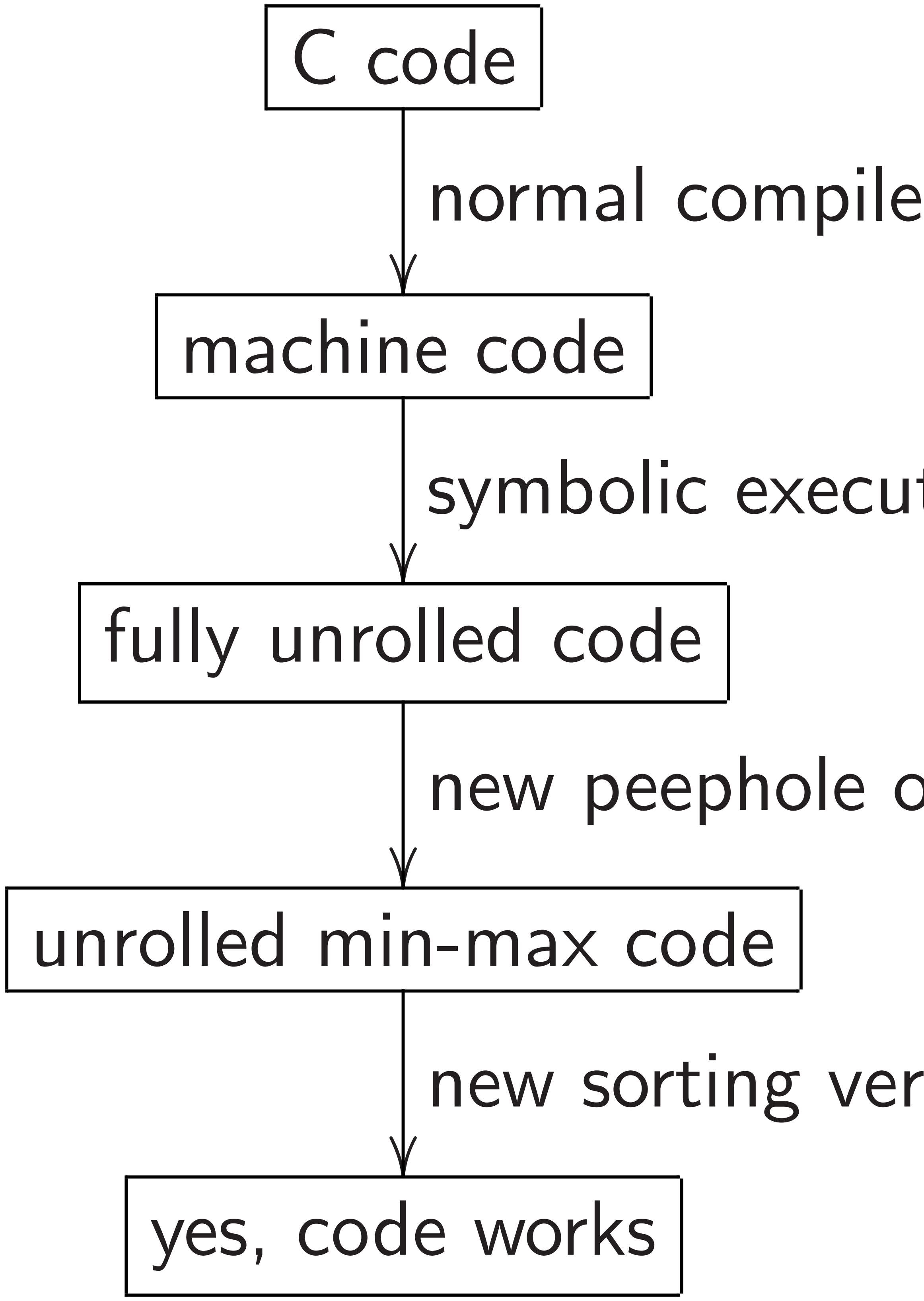
Sorting software is in the TCB.
Does it work correctly?

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 compile

machine code

symbolic execut

fully unrolled code

new peephole o

unrolled min-max code

new sorting ver

yes, code works

Verification

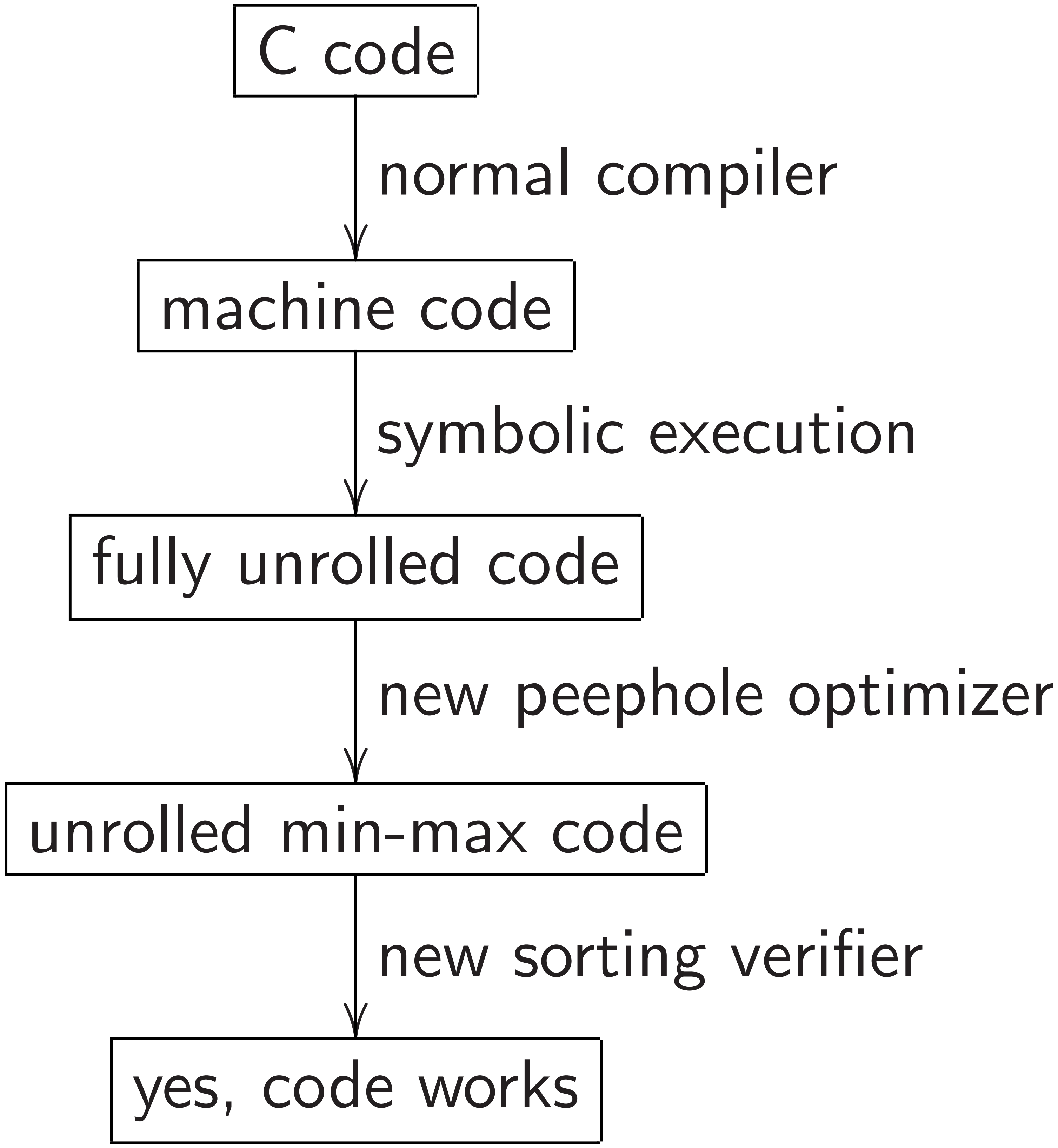
Sorting software is in the TCB. Does it work correctly?

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):



tion

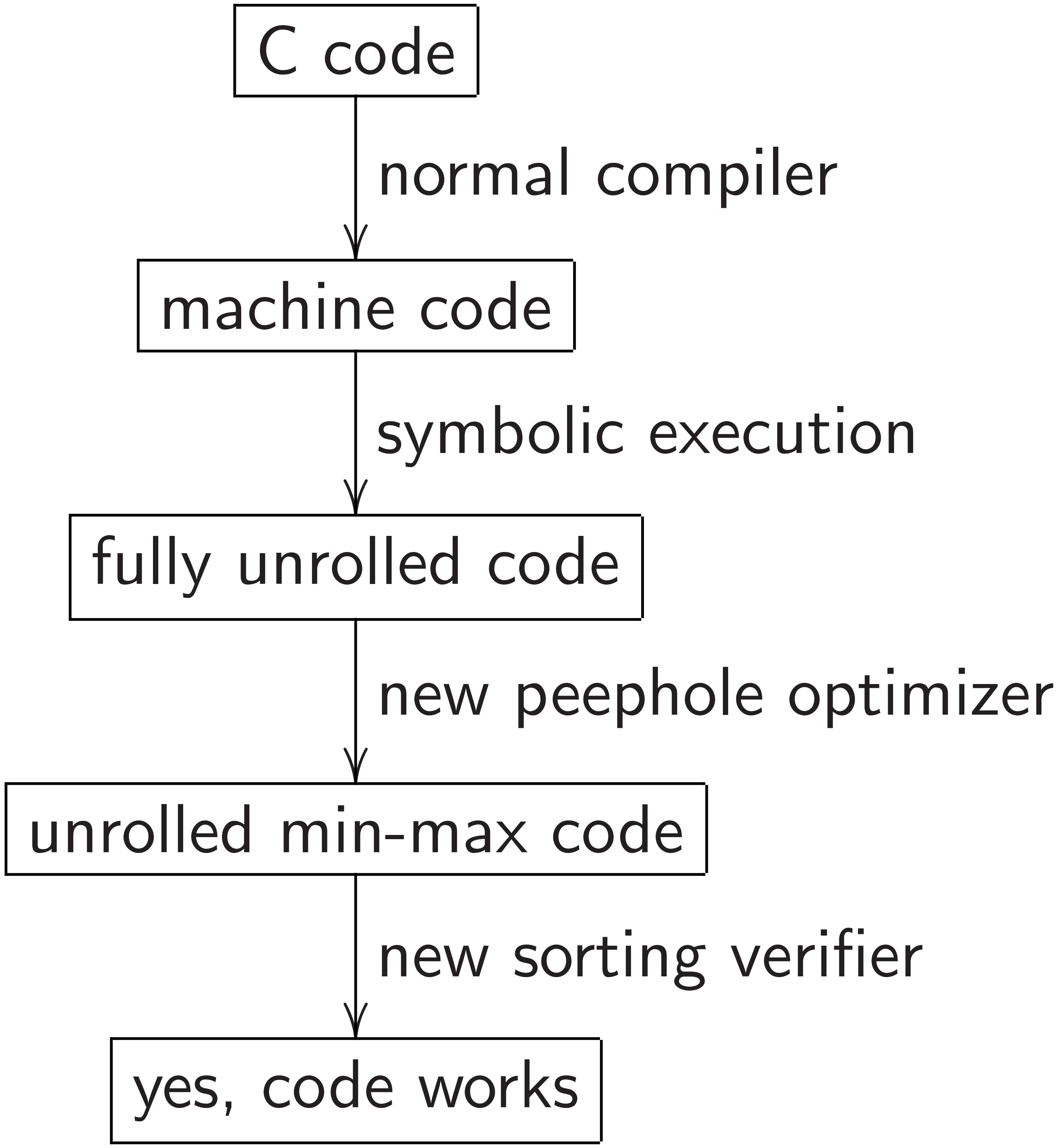
software is in the TCB.
work correctly?

sorting software on many
inputs, increasing inputs,
ng inputs. Seems to work.

there *occasional* inputs
his sorting software
sort correctly?

Many security problems
occasional inputs
CB 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

Symboli
use exist
with tiny
eliminati
a few m

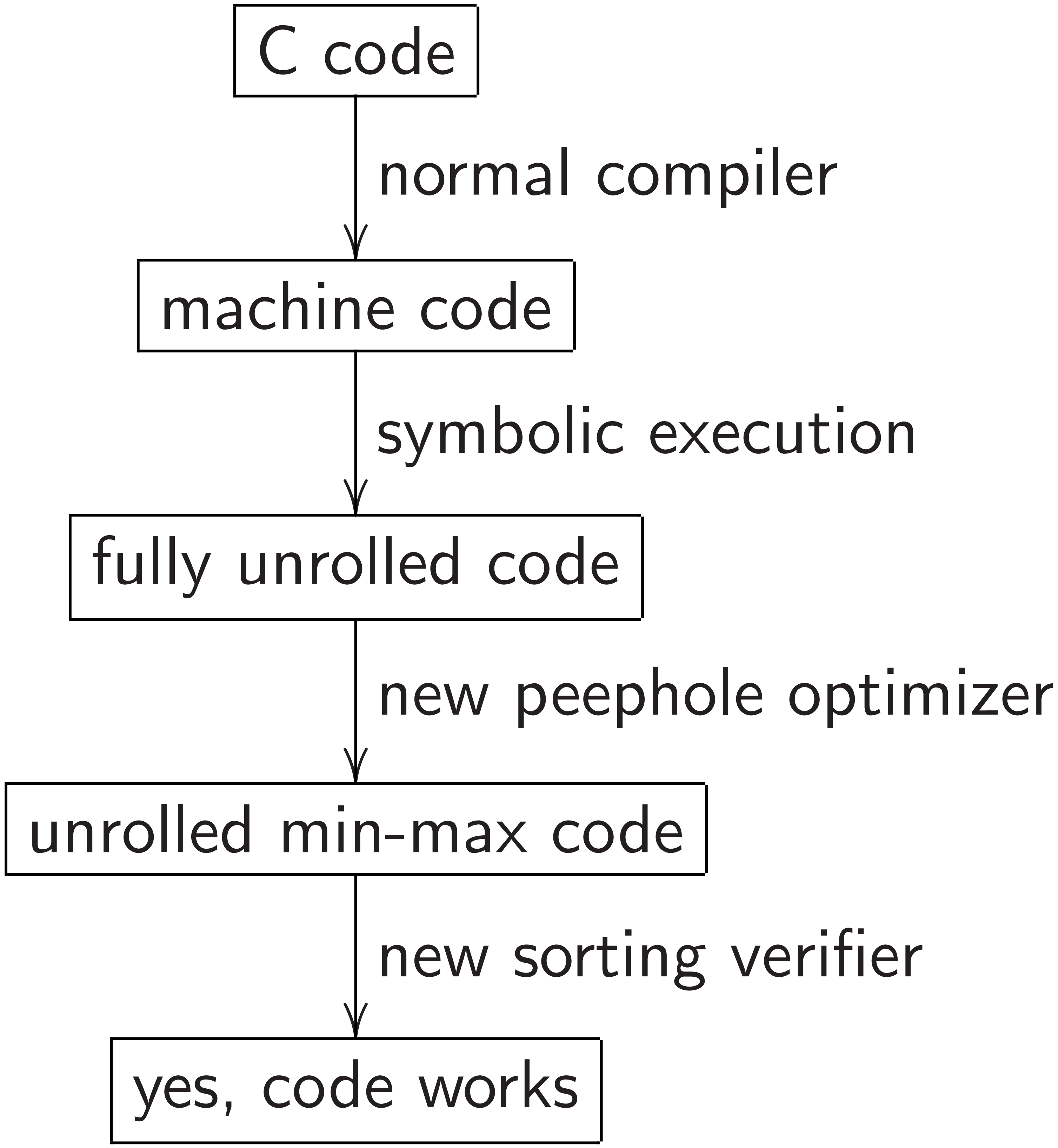
s in the TCB.
ectly?

oftware on many
creasing inputs,
Seems to work.

asional inputs
software
tly?

curity problems
inputs
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 executio
use existing “angr”
with tiny new patc
eliminating byte sp
a few missing vect

CB.

many

puts,

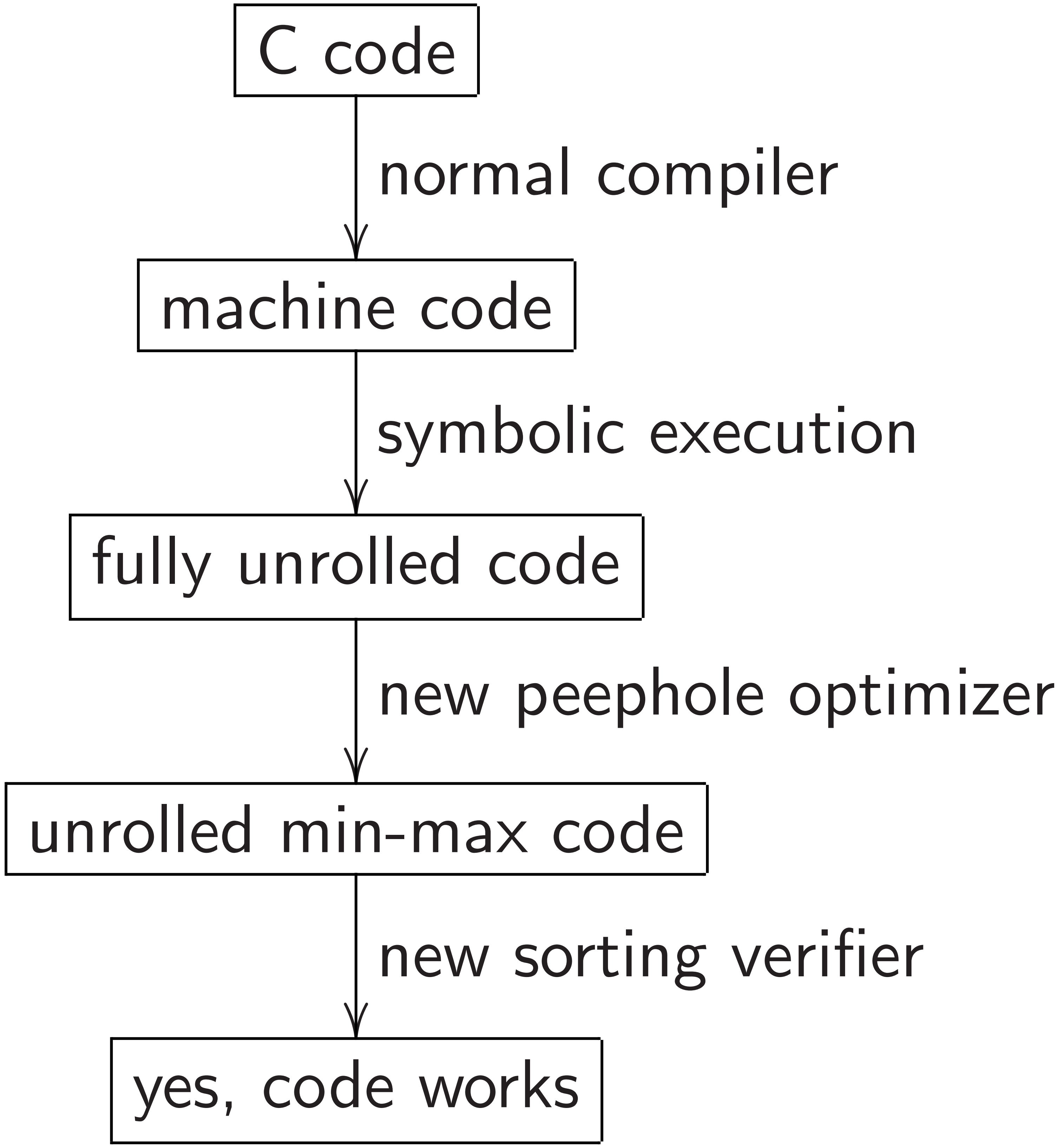
work.

uts

lems

y.

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

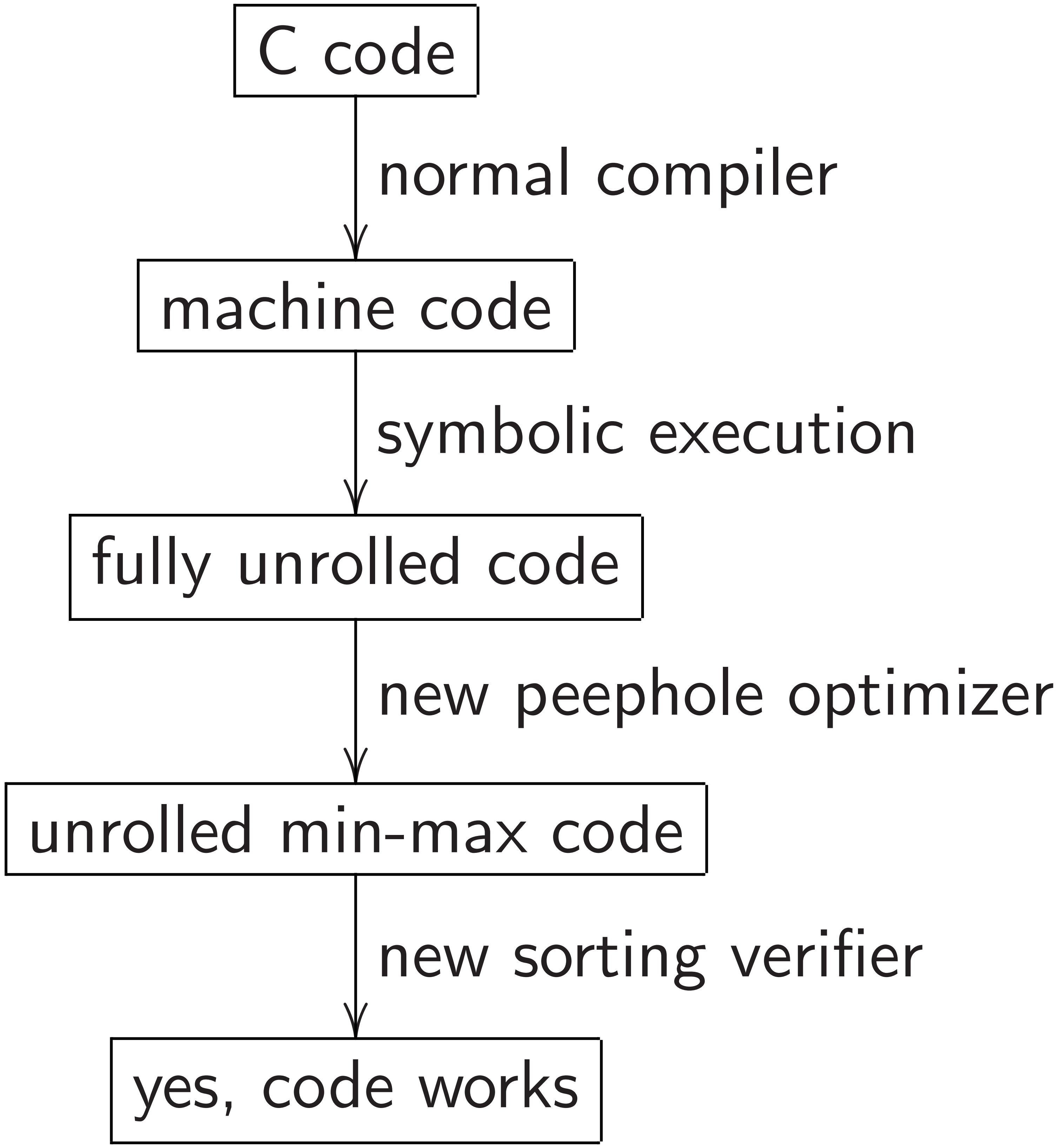
new sorting verifier

yes, code works

Symbolic execution:

use existing “angr” library,
with tiny new patches for
eliminating byte splitting, ac
a few missing vector instruct

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

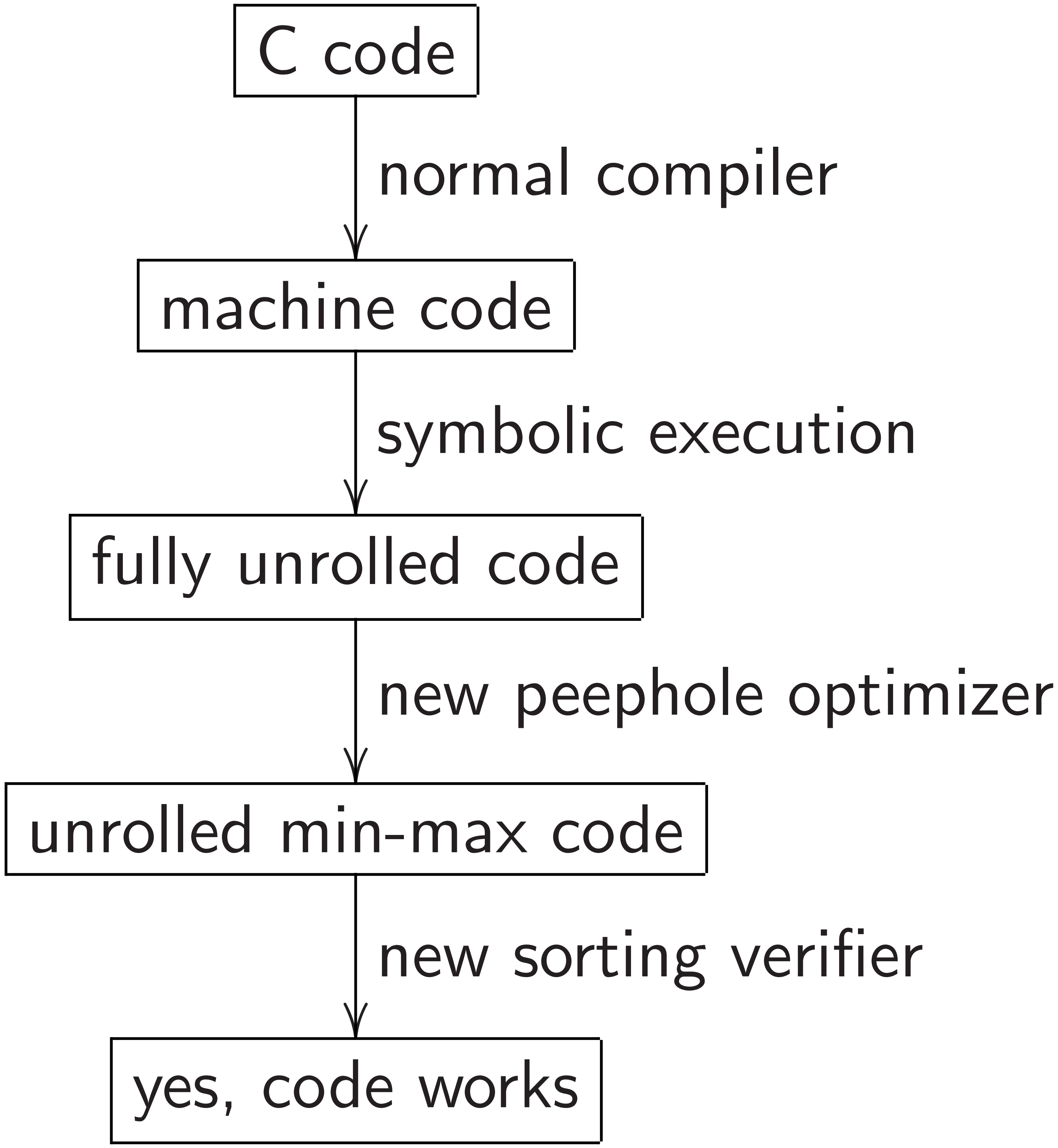
new sorting verifier

yes, code works

Symbolic execution:

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

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

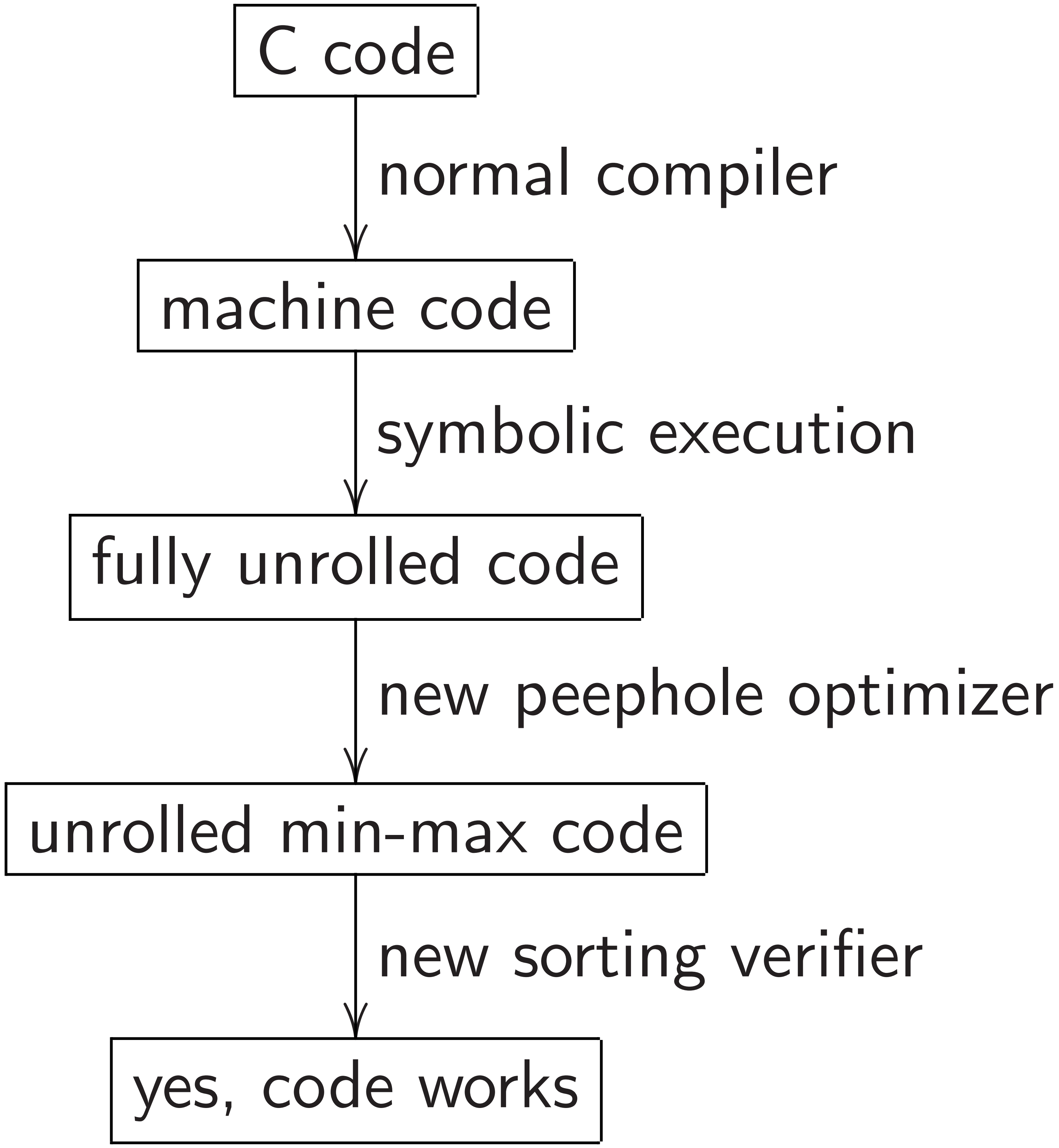
new sorting verifier

yes, code works

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

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

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

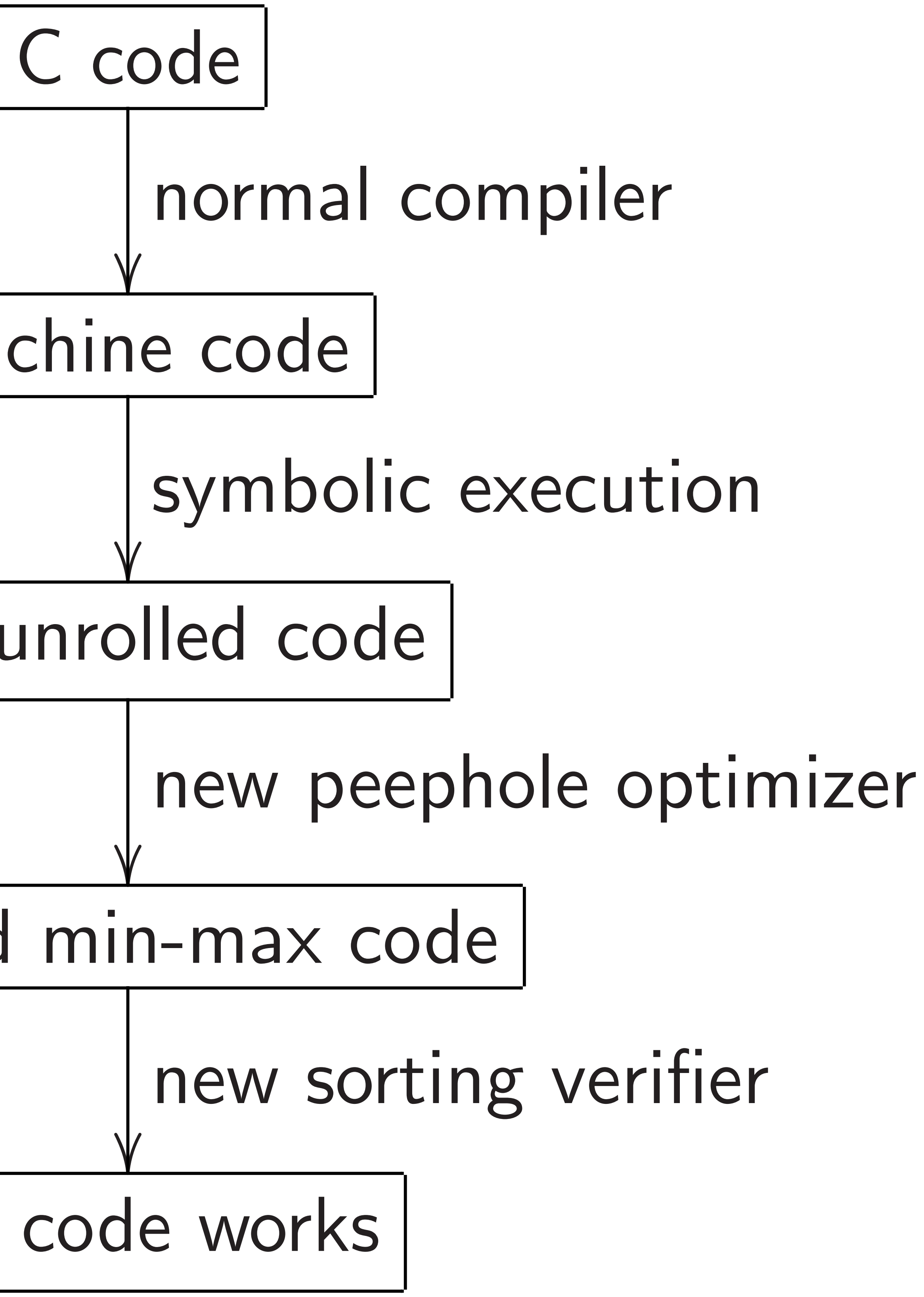
yes, code works

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

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.

n used *n* (e.g., 768):



C code

normal compiler

chine code

symbolic execution

unrolled code

new peephole optimizer

d min-max code

new sorting verifier

code works

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
verified
verified

https:/

Includes
automat
simple b
verificati

Web site
use the

Next rel
verified

e.g., 768):

al compiler

e

olic execution

ode

peephole optimizer

code

sorting verifier

ks

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 rel
verified AVX2 code
verified portable c

https://sorting

Includes the sortin
automatic build-ti
simple benchmarki
verification tools.

Web site shows ho
use the verification

Next release plann
verified ARM NEC

r

tion

ptimizer

ifier

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.t

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

Web site shows how to
use the verification tools.

Next release planned:
verified ARM NEON code.

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.

c execution:
ting "angr" library,
y new patches for
ing byte splitting, adding
issing vector instructions.

e optimizer:
e instruction patterns
nt to min, max.

verifier: decompose
o merging networks.
ach merging network
neralization of 2007
evi–Litman, correction of
ung–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 futu

I don't t
fundame

- crypto
- stoppi
- makin

See the

Firefox h
verified c
Curve25

I'm work
post-qua

n:
” library,
ches for
plitting, adding
tor instructions.

r:
on patterns
max.

ecompose
networks.
ng network
on of 2007
, correction of
kumar.

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
fundamental tensi

- crypto performa
- stopping timing
- making sure soft

See the sorting ex

Firefox has already
verified constant-t
Curve25519+ChaC

I’m working on eas
post-quantum cod

dding
tions.

ns

.

k

on of

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 betwee

- crypto performance,
- stopping timing attacks,
- making sure software work

See the sorting example.

Firefox has already deployed
verified constant-time softwa
Curve25519+ChaCha20+Po

I'm working on easier verific
post-quantum code, faster c

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.