

Sorting integer arrays:
security, speed, and verification

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.

integer arrays:

speed, and verification

ernstein

1

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.

2

Trusted

TCB: po

that is re

the users

ays:
d verification

1

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.

2

Trusted computing

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

1

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.

2

Trusted computing base (TCB)

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

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.

Trusted computing base (TCB)

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

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.

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.

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.

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.

laptop screen:

From: Alice

Thank you for your

contribution. We received

some interesting papers,

unfortunately your

assumes this message is

from Alice actually sent.

Today's "security" systems

guarantee this property.

Someone could have modified

the message.

2

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.

3

Example

1. Attack

in a c

Linux

en:

your

We received
ing papers,
ately your

message is
actually sent.

urity" systems
his property.
ve modified
sage.

2

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.

3

Examples of attac

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

2

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.

3

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.

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)

portion of computer system
responsible for enforcing
s' security policy.

policy for this talk:

page is displayed on
screen as "From: Alice"
message is from Alice.

works correctly,
message is guaranteed
from Alice, no matter what
of the system does.

3

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?

4

Classic s

Researchit
to have

g base (TCB)

computer system
for enforcing
policy.

this talk:

ayed on

From: Alice"

om Alice.

ectly,

uaranteed

no matter what
tem does.

3

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?

4

Classic security str

Rearchitect compu
to have a much sr

3

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?

4

Classic security strategy:

Rearchitect computer system 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.

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:



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:



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:



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.

rowser is in the TCB.

n the TCB. Etc.

TCB has many bugs,

g many security holes.

oe of fixing this?

4

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.

5

Cryptogr

How doe
that inco
is from A

Cryptogr
Message

Alic

authent

authent

Alic

Attack strategies:

buffer overflow

er to control

Alice's laptop.

buffer overflow

er to control

Bob's laptop.

the TCB.

the TCB.

B. Etc.

many bugs,

curity holes.

g 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.

Cryptography

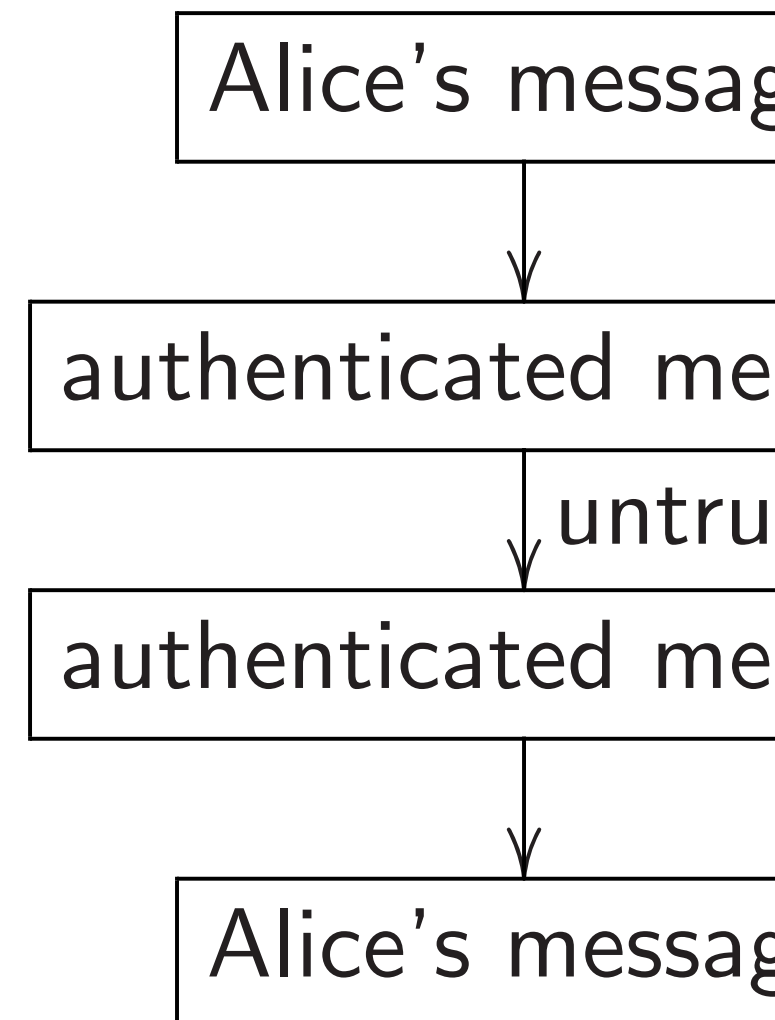
How does Bob's laptop

that incoming network

is from Alice's laptop?

Cryptographic solution

Message-authentication



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.

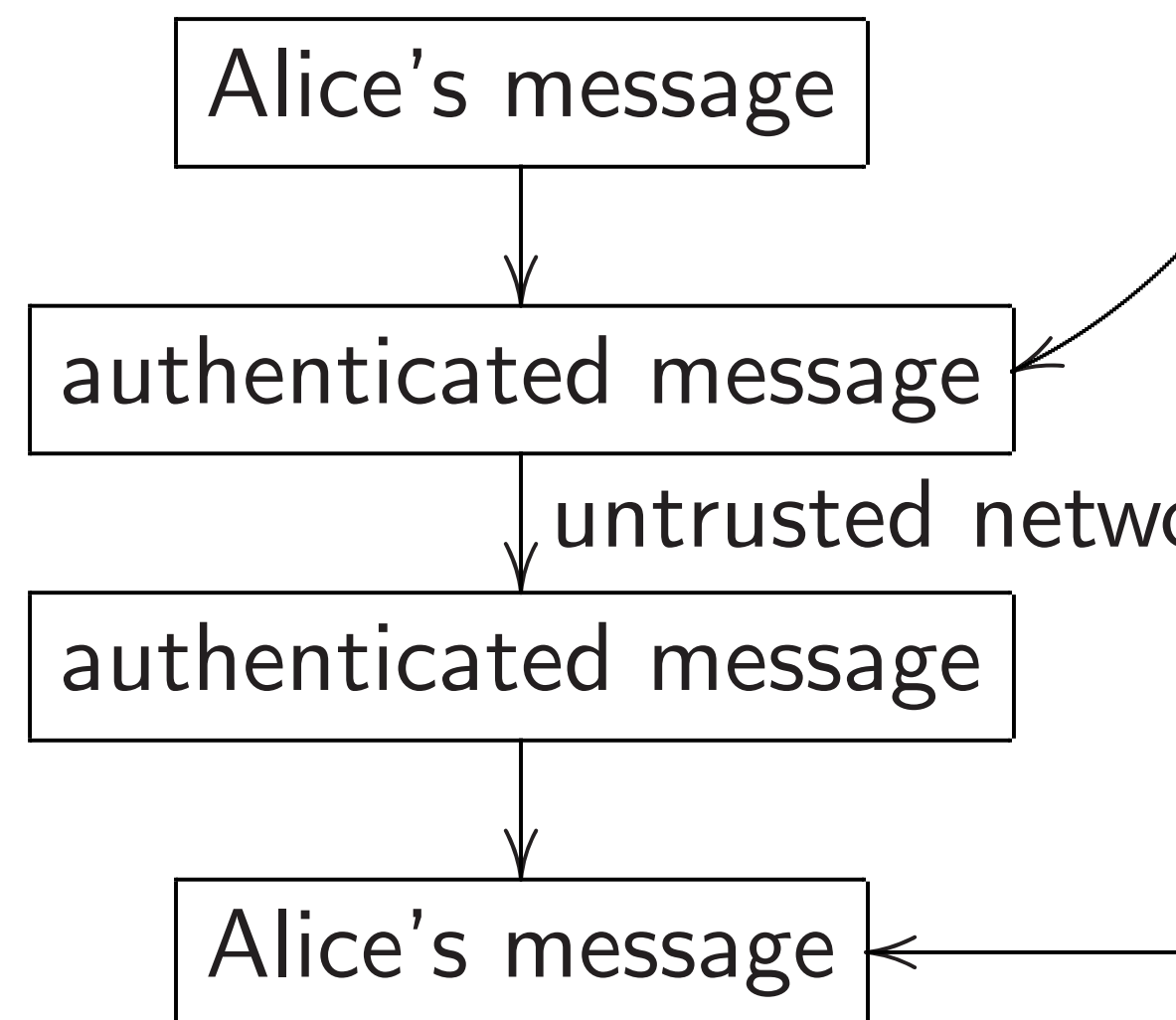
Alice also runs many VMs.

Cryptography

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

Cryptographic solution:

Message-authentication code



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.

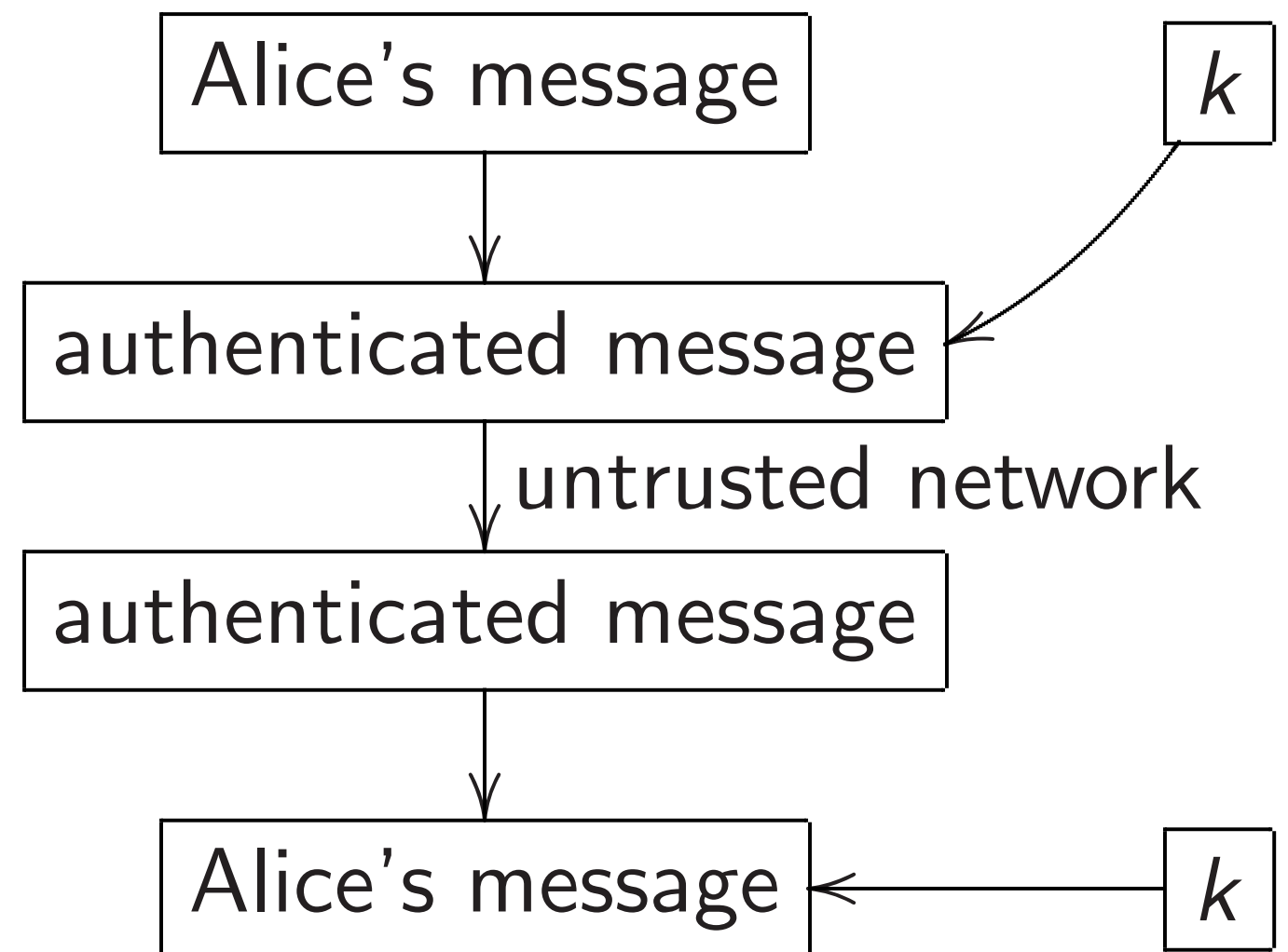
Alice also runs many VMs.

Cryptography

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

Cryptographic solution:

Message-authentication codes.



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.

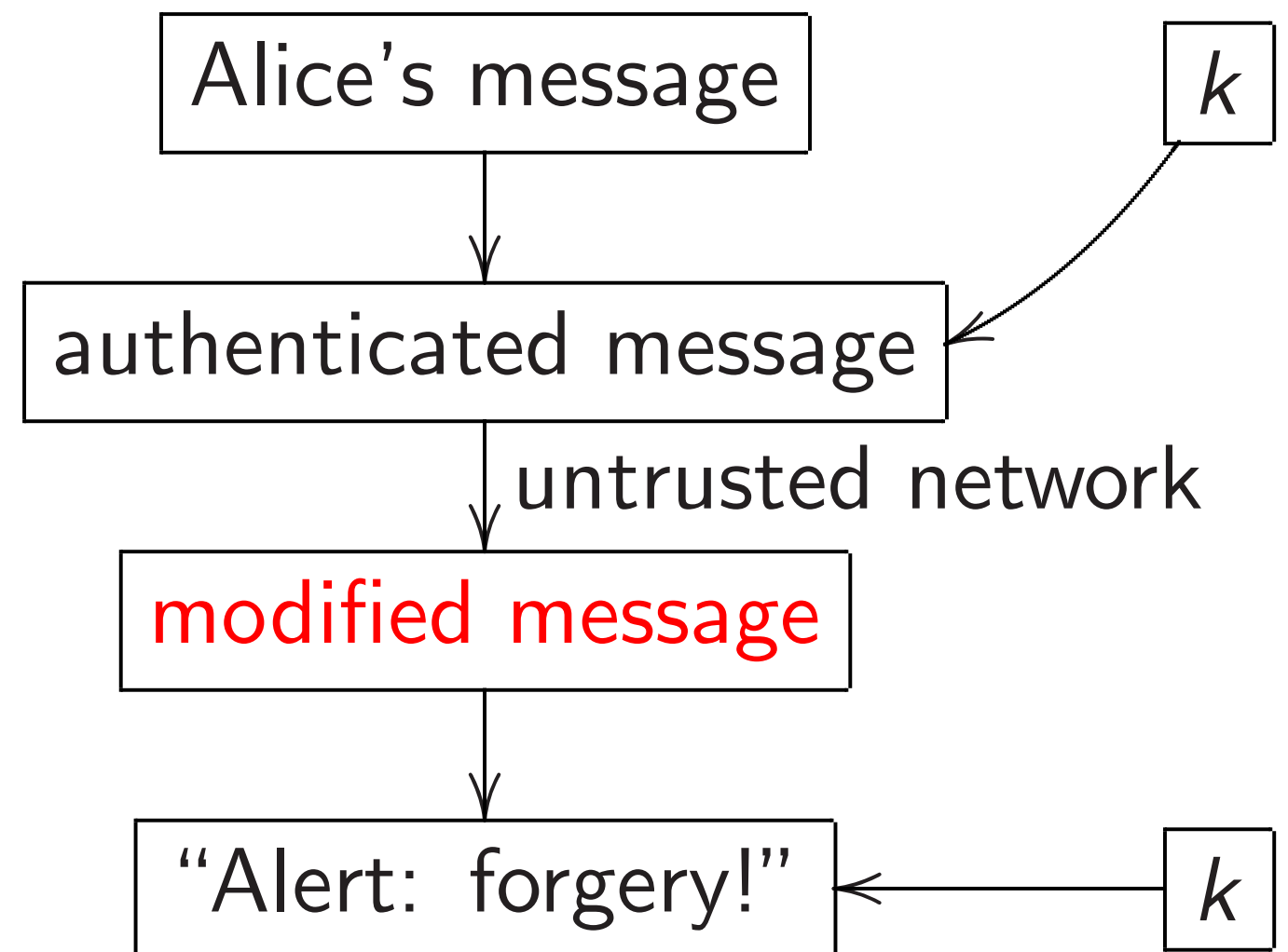
Alice also runs many VMs.

Cryptography

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

Cryptographic solution:

Message-authentication codes.



security strategy:

protect computer systems
with a much smaller TCB.

regularly audit the TCB.

OS runs many VMs:



protects each VM from
accessing data in other VMs.

data in VM C isn't in TCB.

data in VM A,
works correctly.

OS runs many VMs.

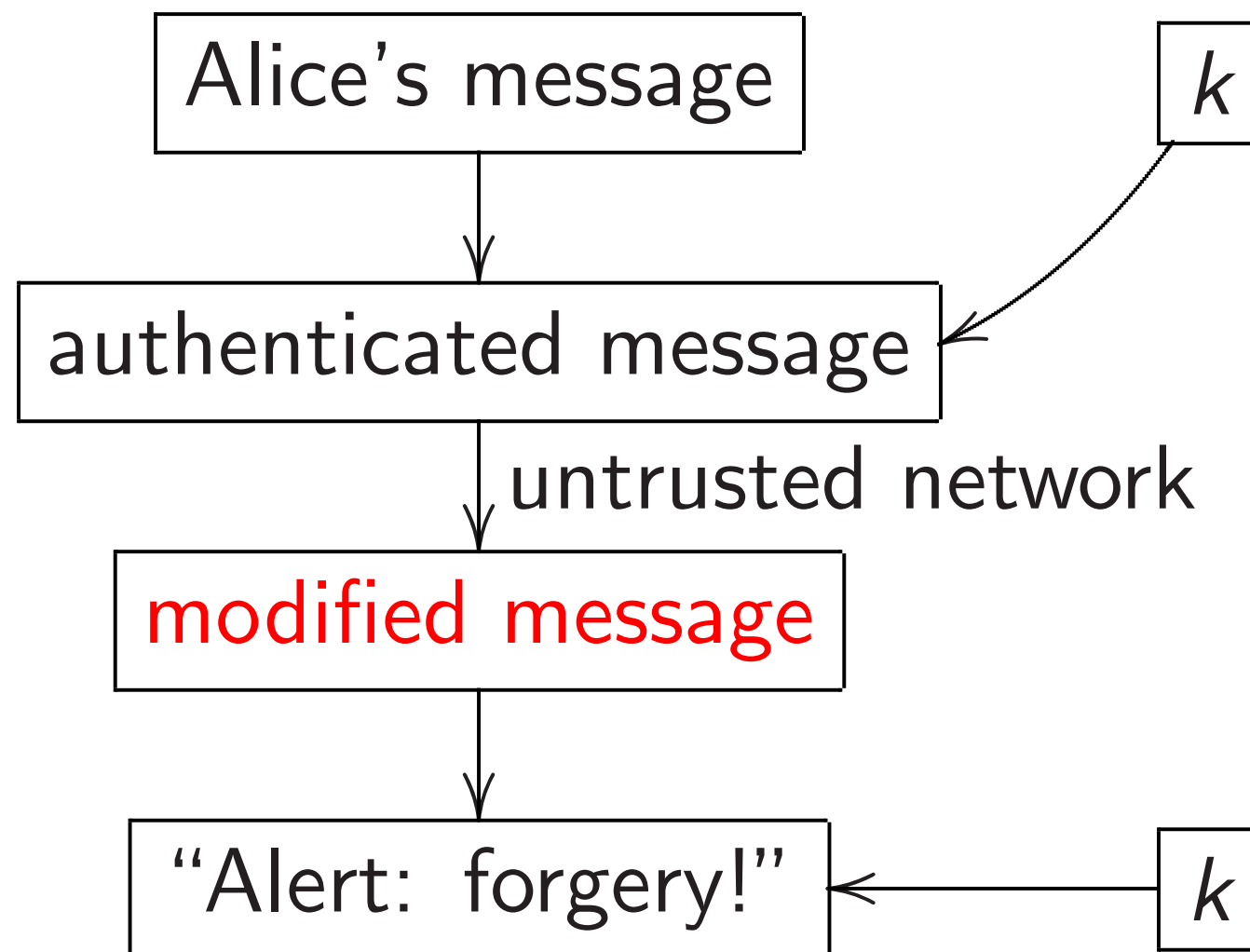
5

Cryptography

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

Cryptographic solution:

Message-authentication codes.



6

Important
to share
What if
on their

strategy:

computer systems

smaller TCB.

TCB.

VMs:

VM C
 Charlie data

...

VM from

other VMs.

isn't in TCB.

in VM A,

ectly.

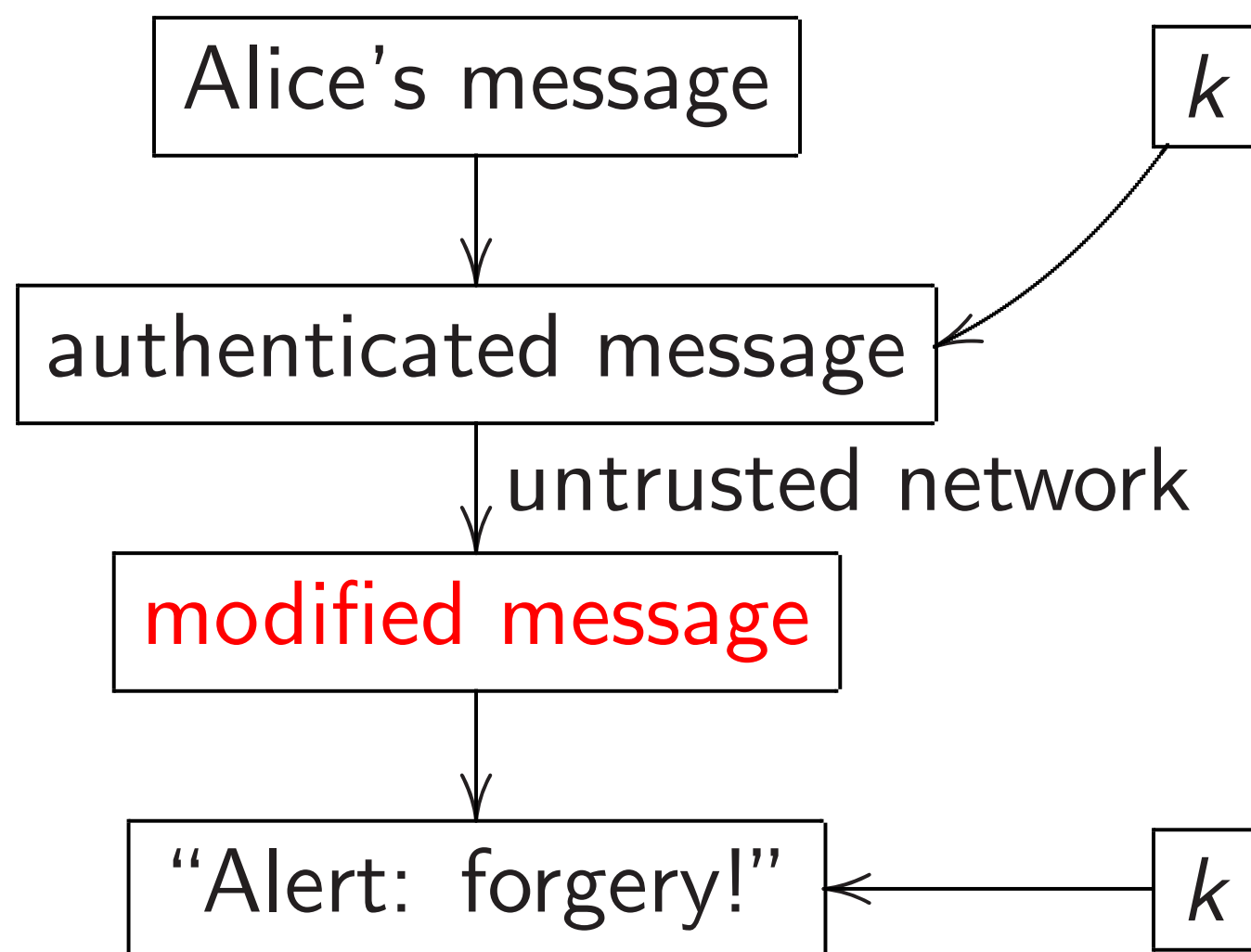
ny VMs.

Cryptography

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

Cryptographic solution:

Message-authentication codes.



Important for Alice

to share the same

What if attacker w

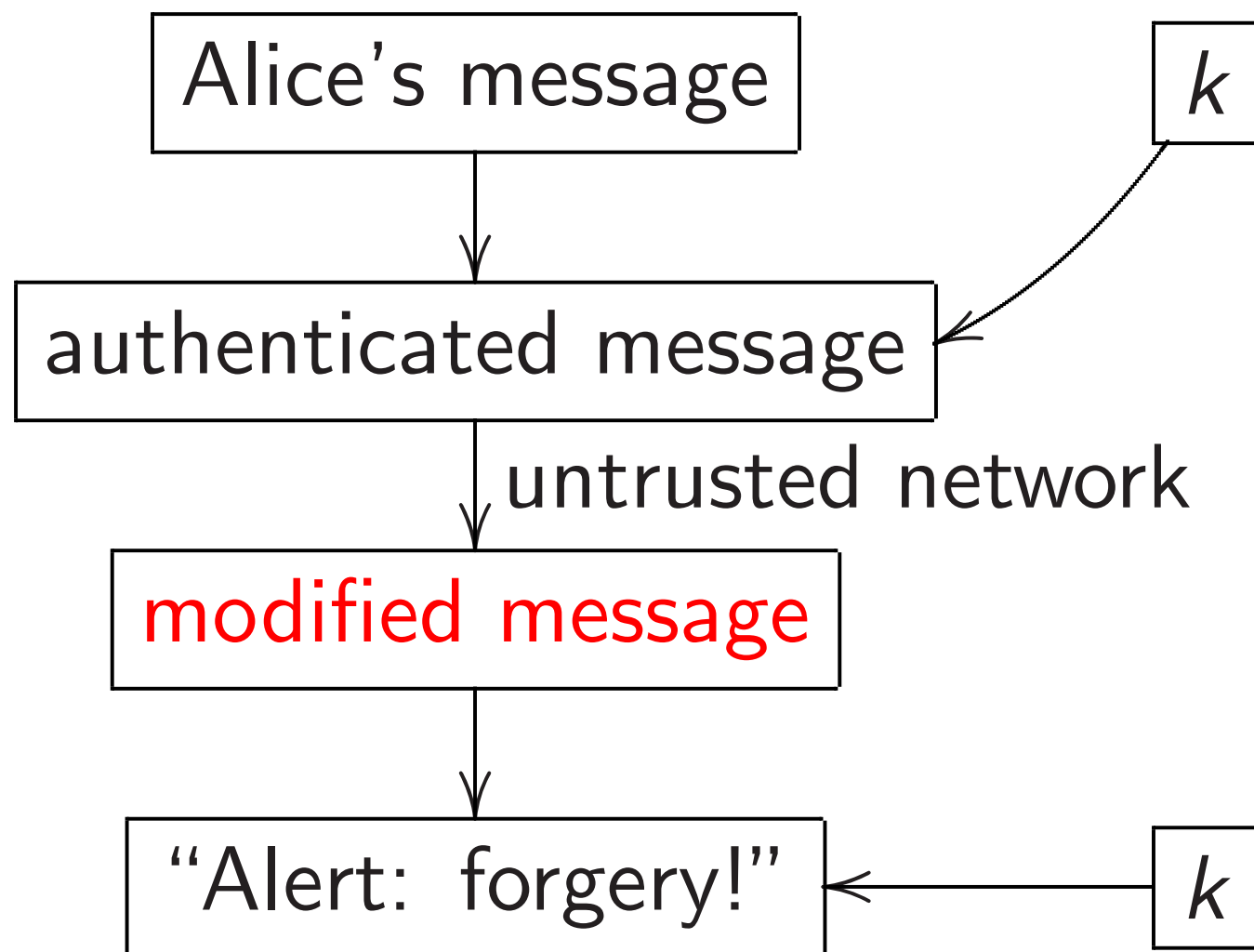
on their communic

5

Cryptography

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

Cryptographic solution:
Message-authentication codes.



6

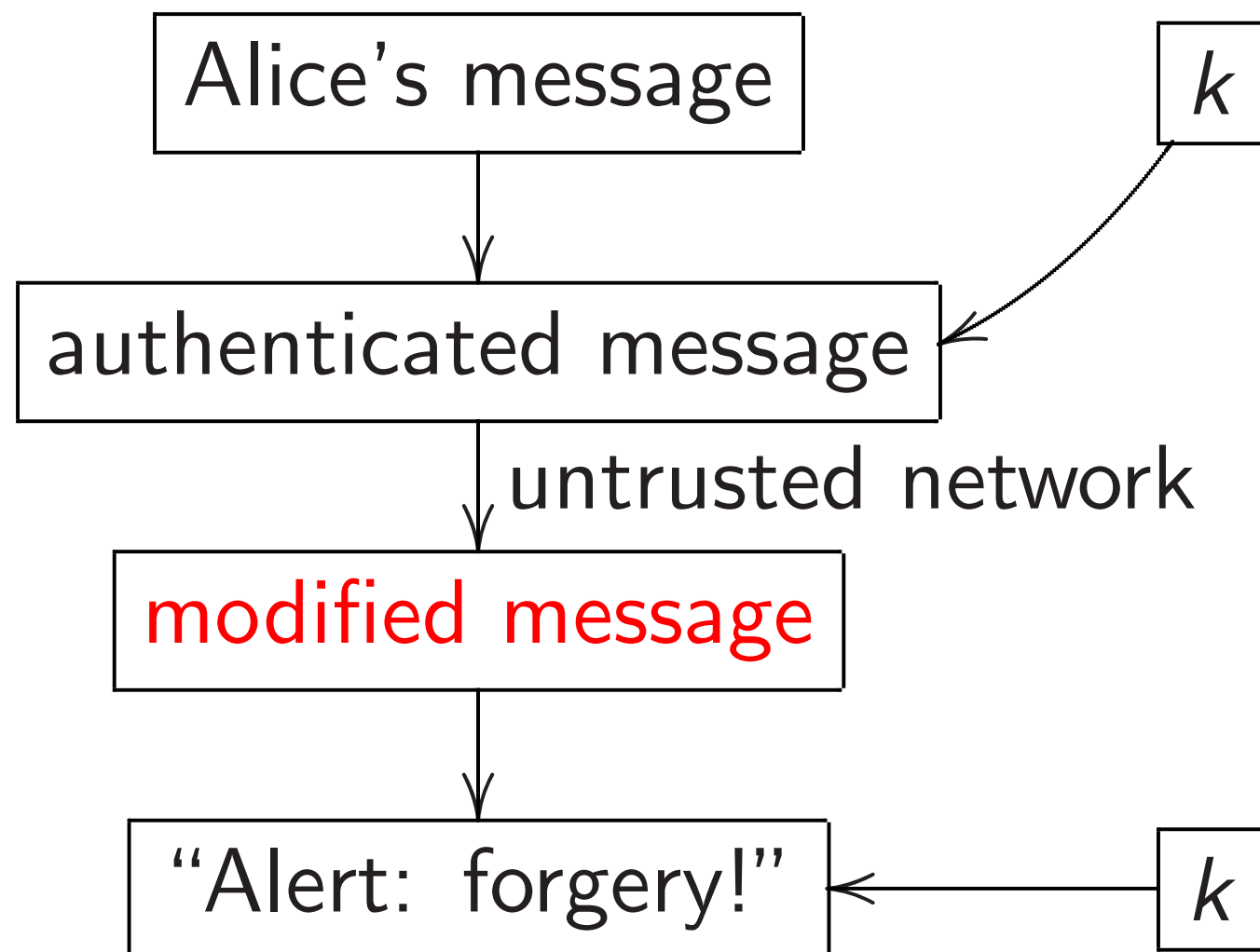
Important for Alice and Bob to share the same secret k .

What if attacker was spying on their communication of k ?

Cryptography

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

Cryptographic solution:
Message-authentication codes.



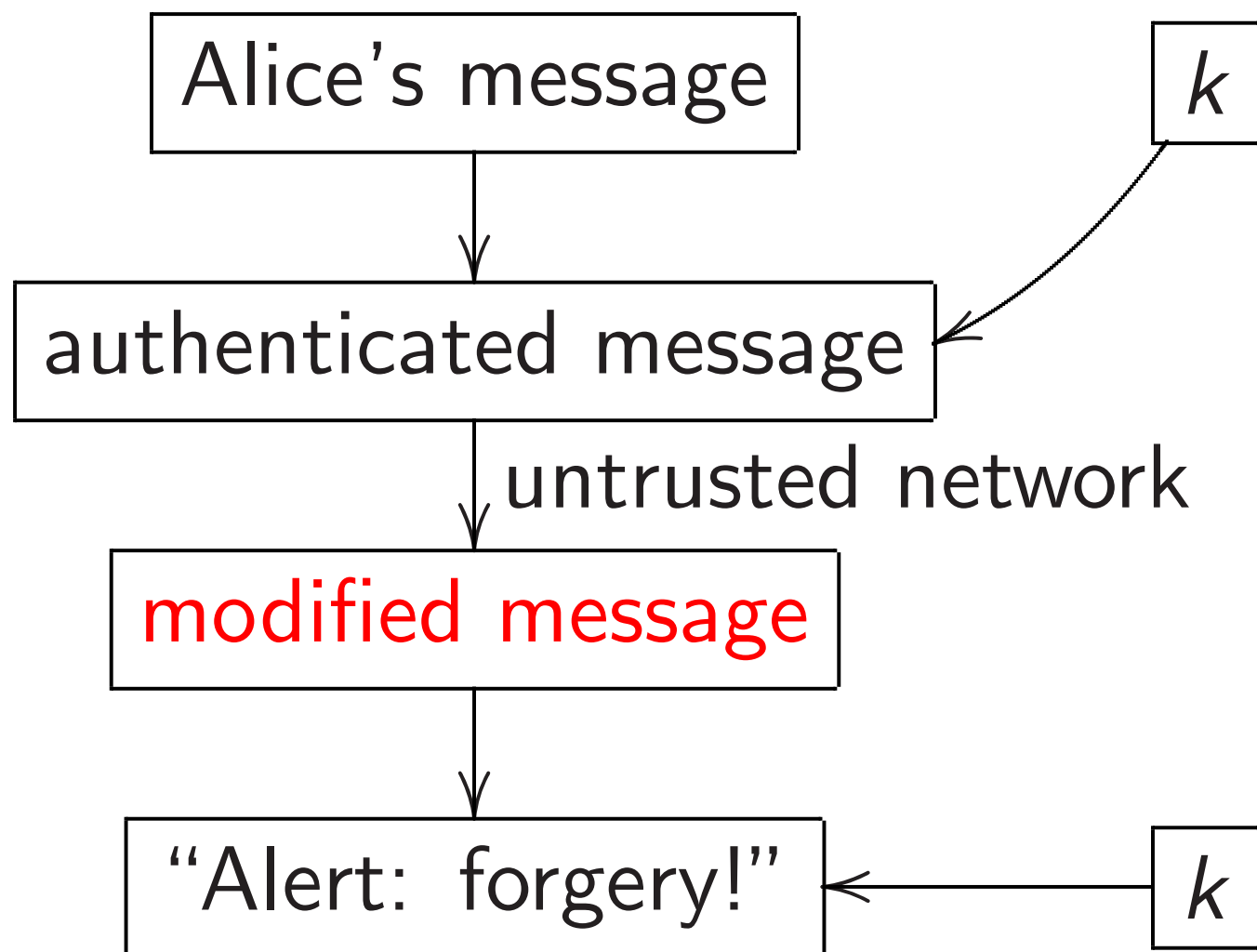
Important for Alice and Bob to share the same secret k .

What if attacker was spying on their communication of k ?

Cryptography

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

Cryptographic solution:
Message-authentication codes.

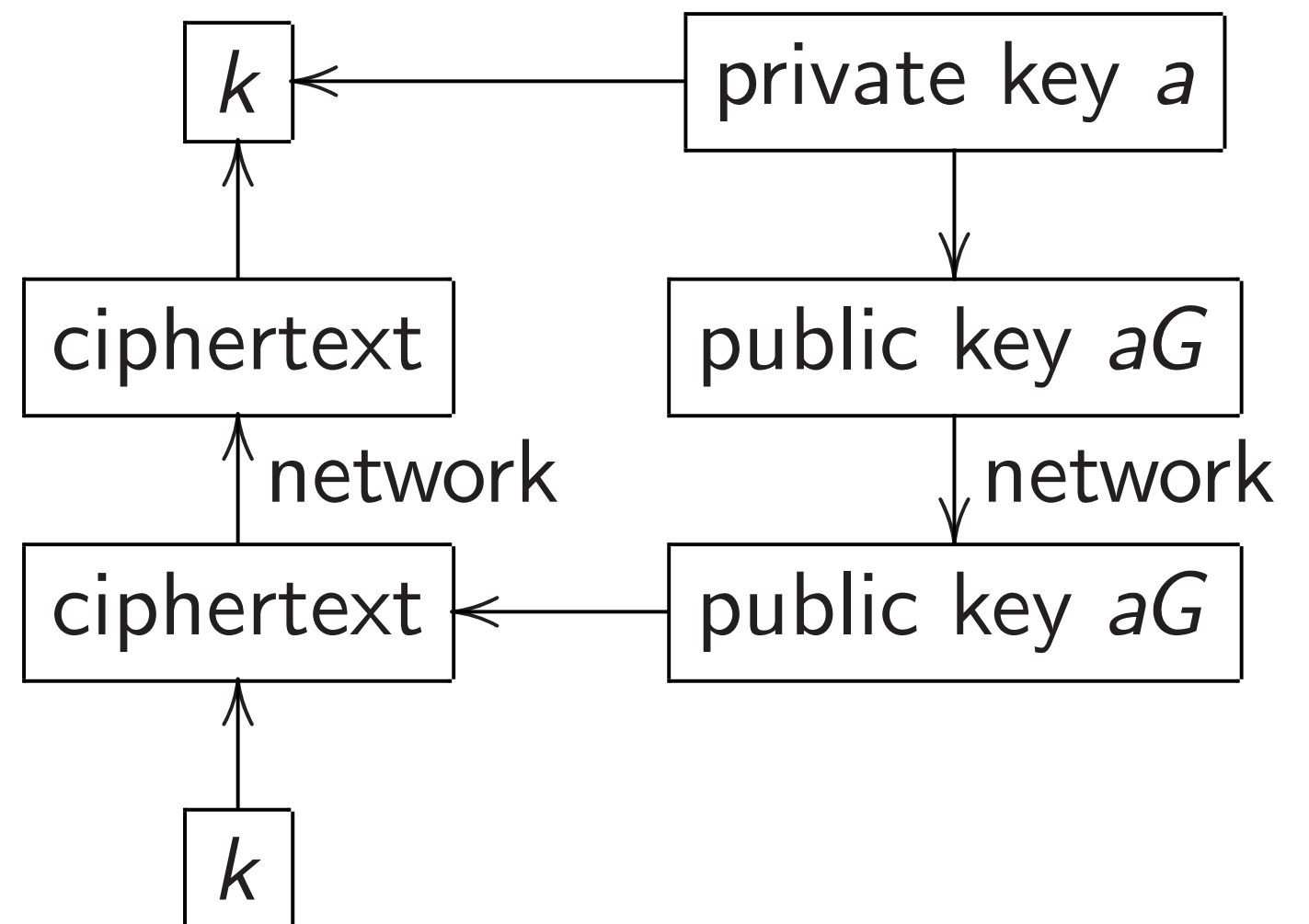


Important for Alice and Bob to share the same secret k .

What if attacker was spying on their communication of k ?

Solution 1:

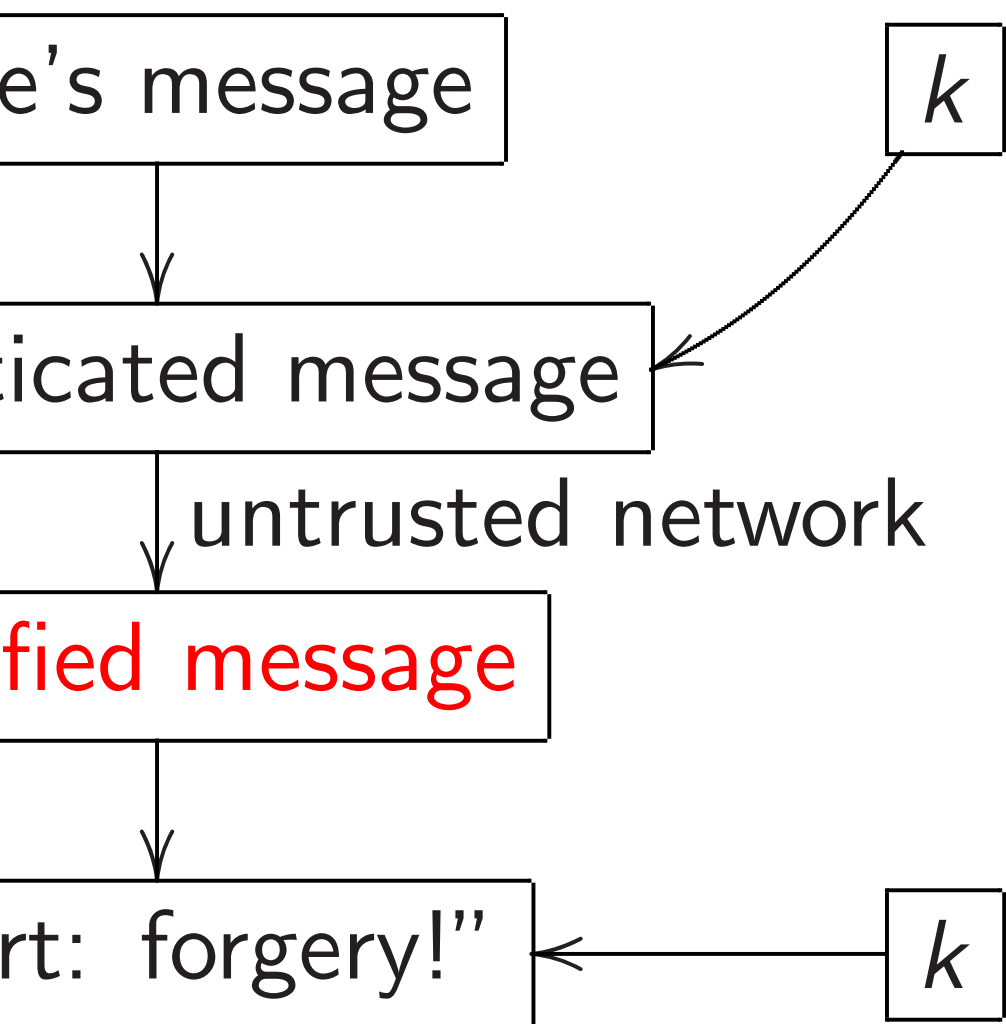
Public-key encryption.



graphy

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

raphic solution:
e-authentication codes.



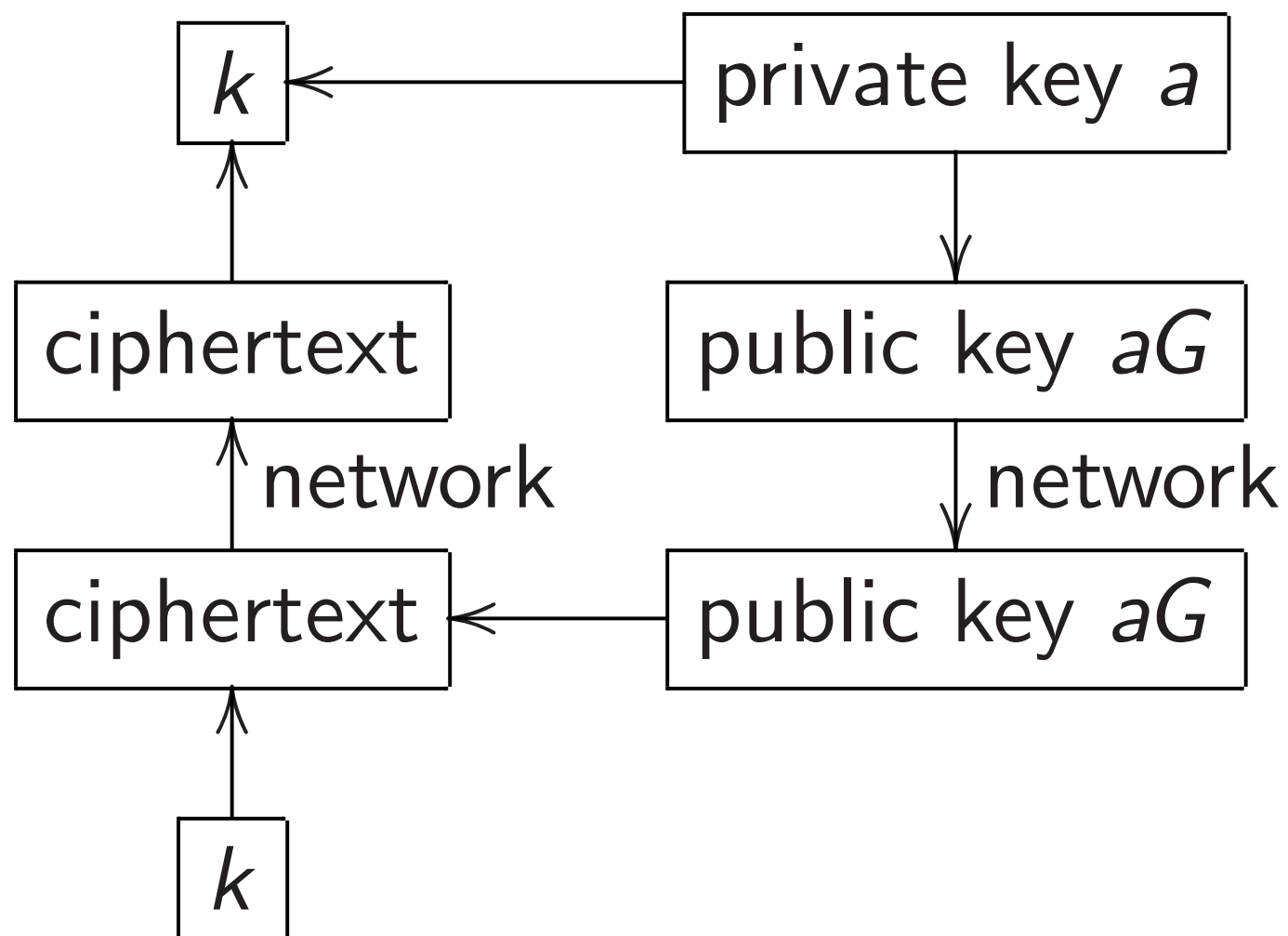
6

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.



7

Solution
Public-k

signed r

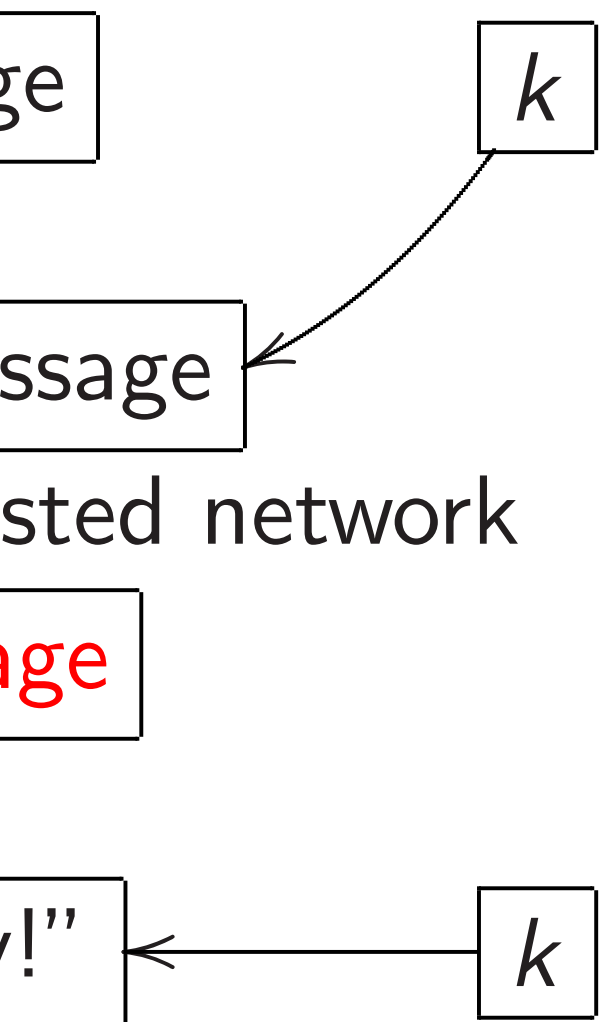
signed r

signed r

6

laptop know
work data
top?

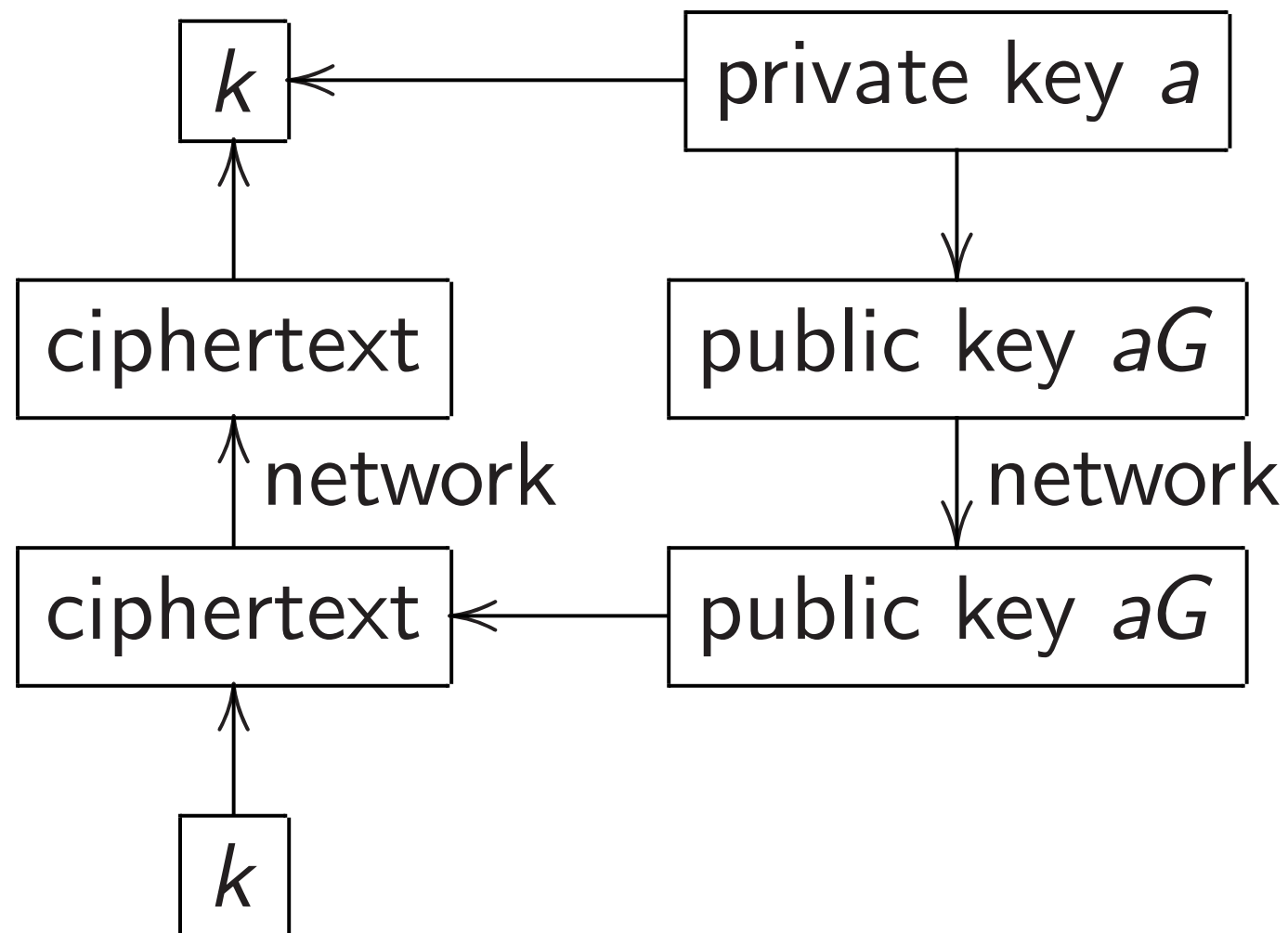
ution:
ication codes.



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

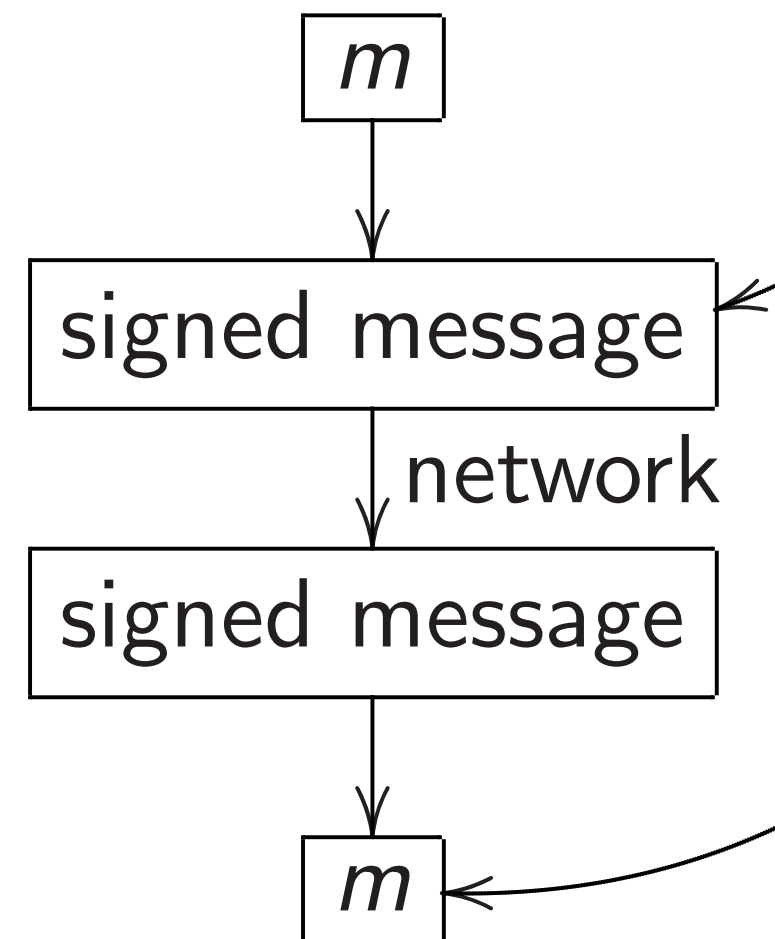
What if attacker was spying
on their communication of k ?

Solution 1:
Public-key encryption.



7

Solution 2:
Public-key signatu



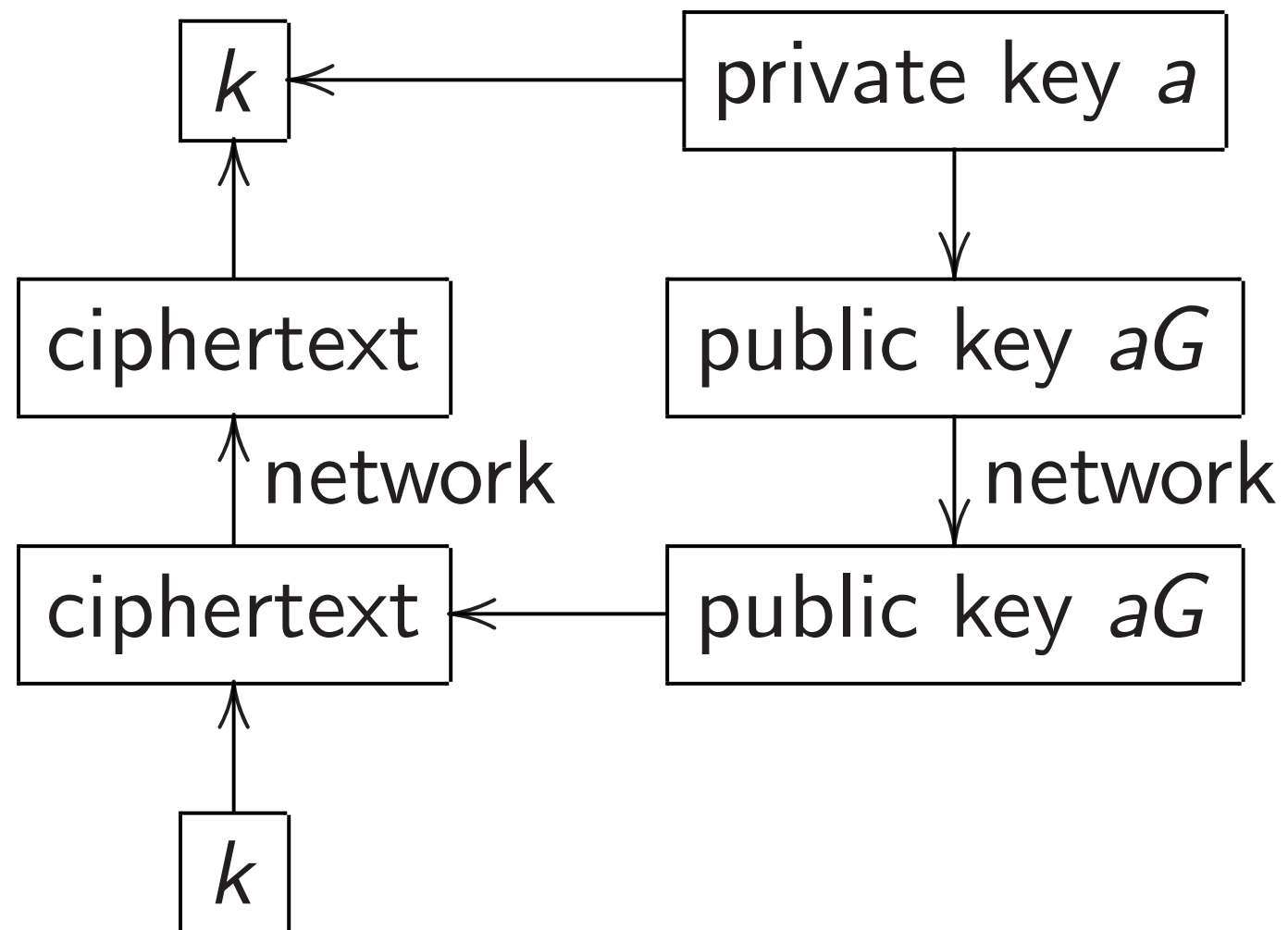
6

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

What if attacker was spying on their communication of k ?

Solution 1:

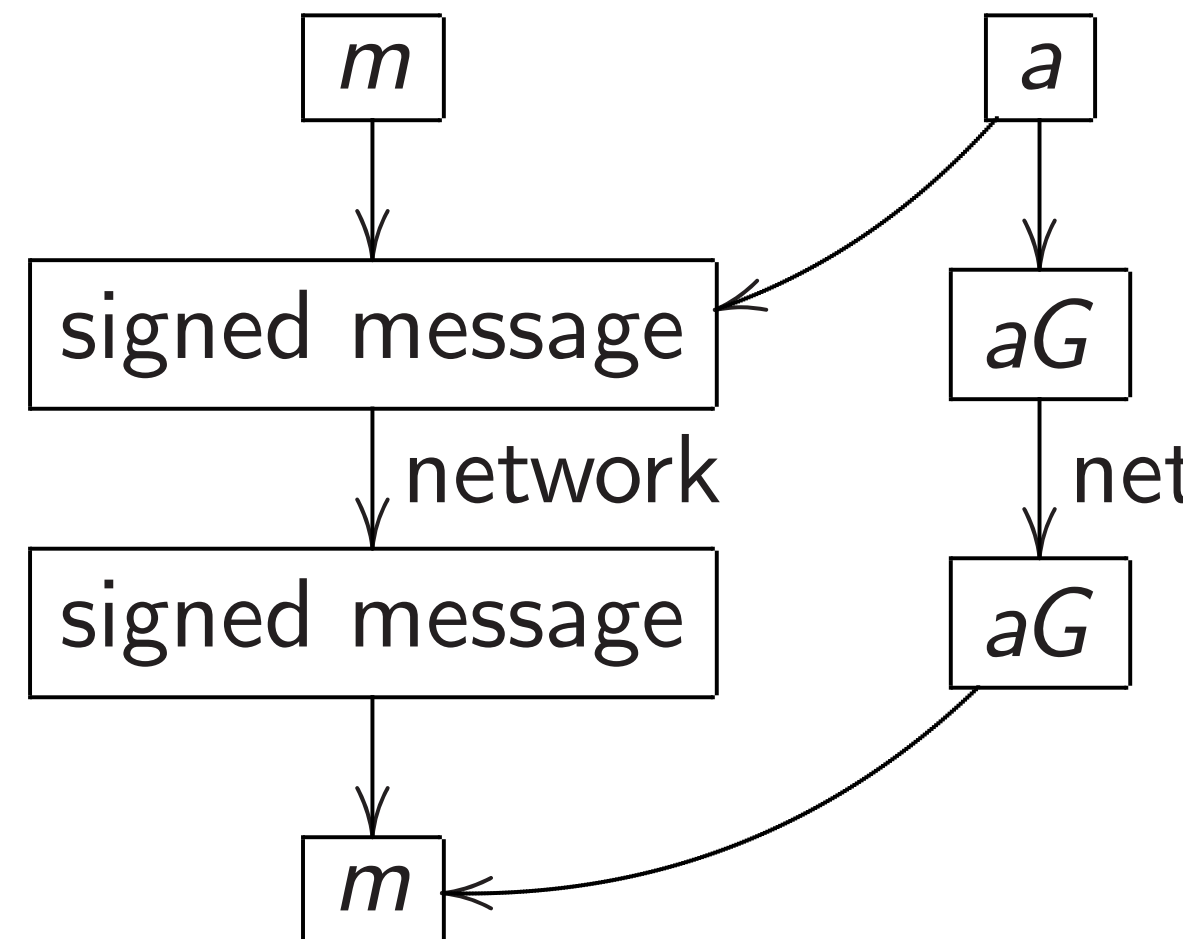
Public-key encryption.



7

Solution 2:

Public-key signatures.

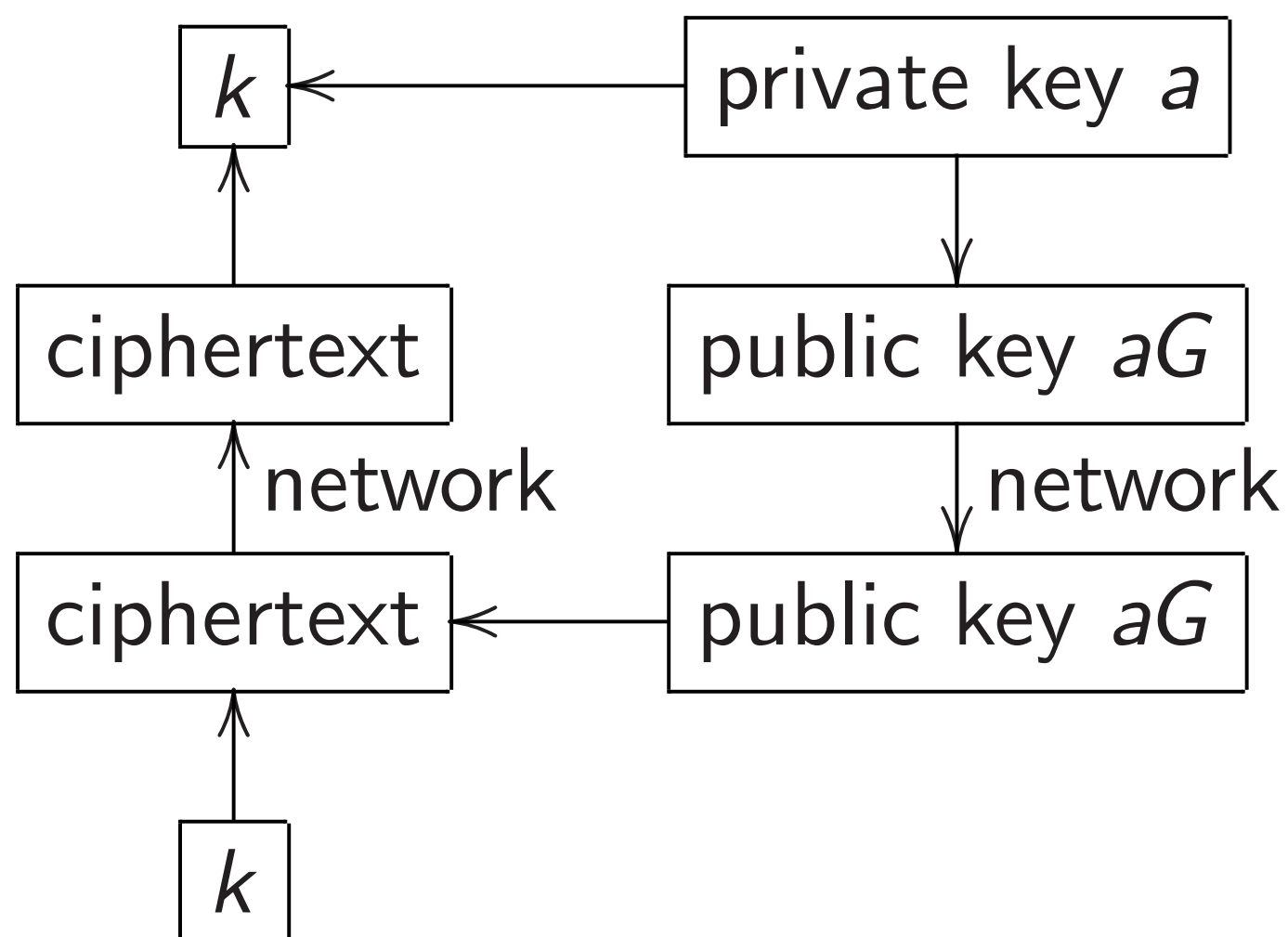


Important for Alice and Bob to share the same secret k .

What if attacker was spying on their communication of k ?

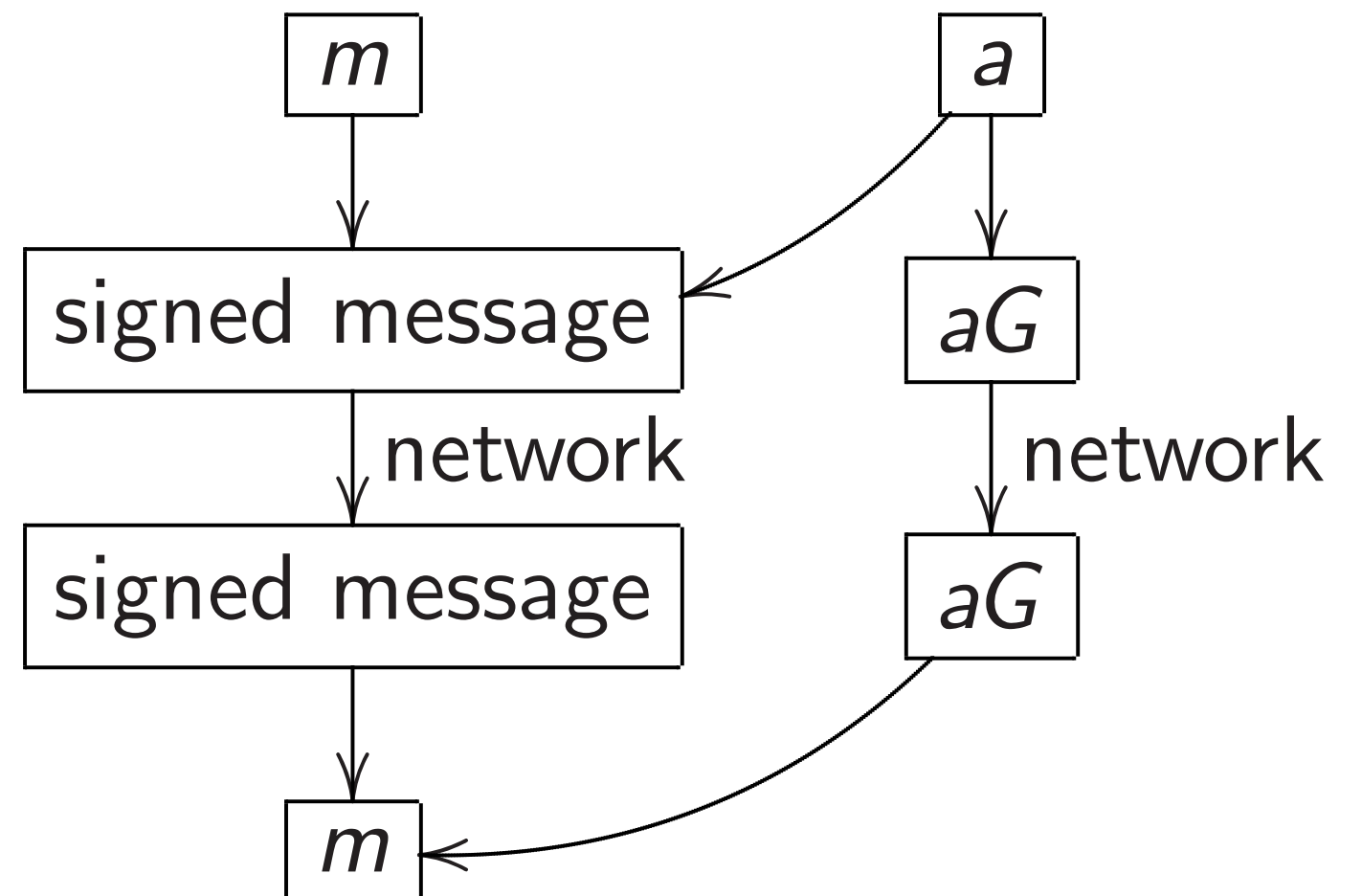
Solution 1:

Public-key encryption.



Solution 2:

Public-key signatures.

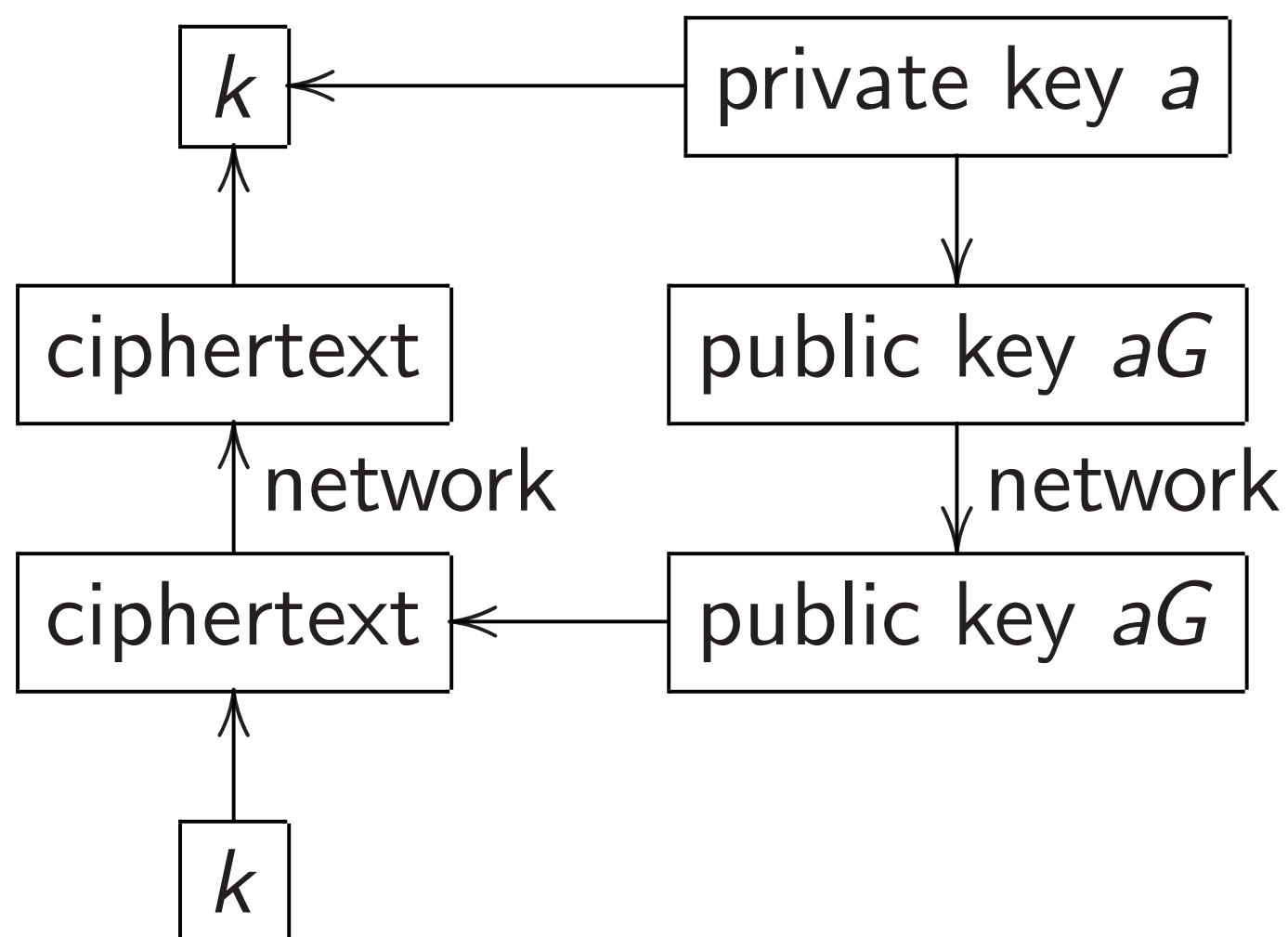


Important for Alice and Bob to share the same secret k .

What if attacker was spying on their communication of k ?

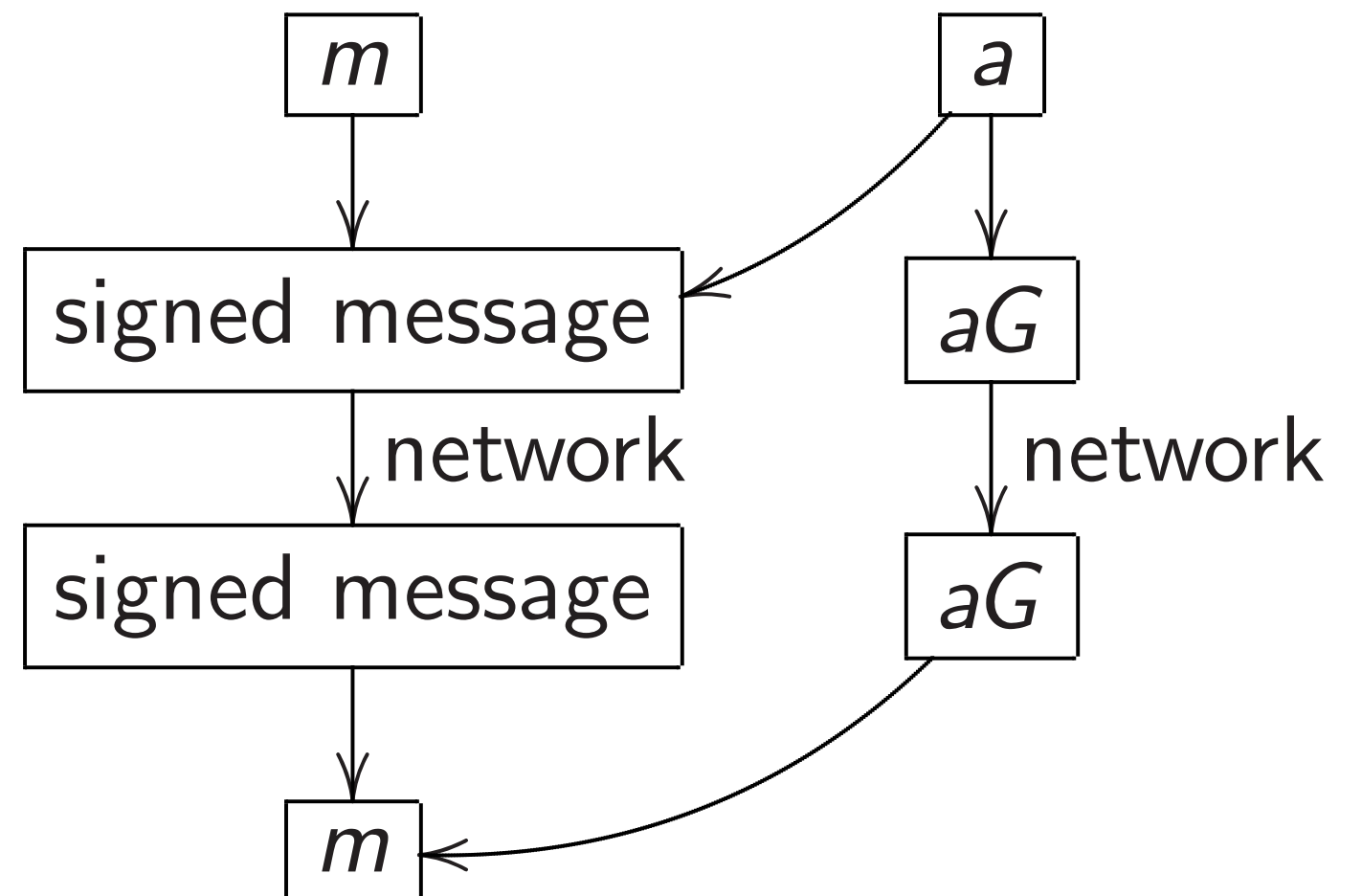
Solution 1:

Public-key encryption.



Solution 2:

Public-key signatures.



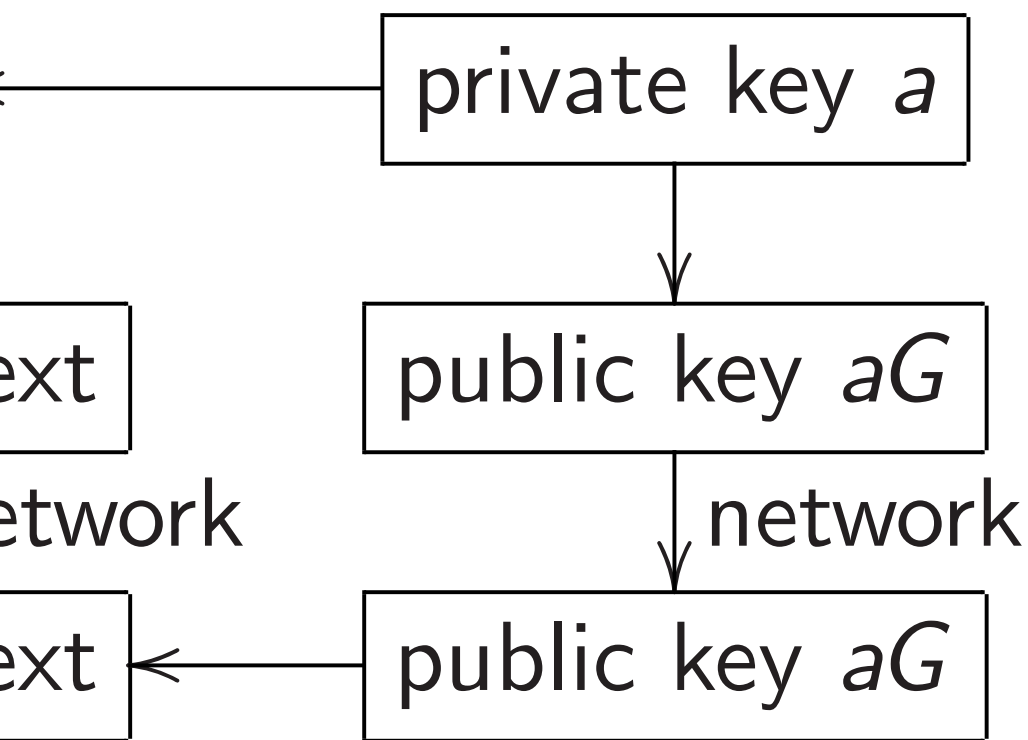
No more shared secret k but Alice still has secret a .

Cryptography requires TCB to protect secrecy of keys, even if user has no other secrets.

nt for Alice and Bob
the same secret k .

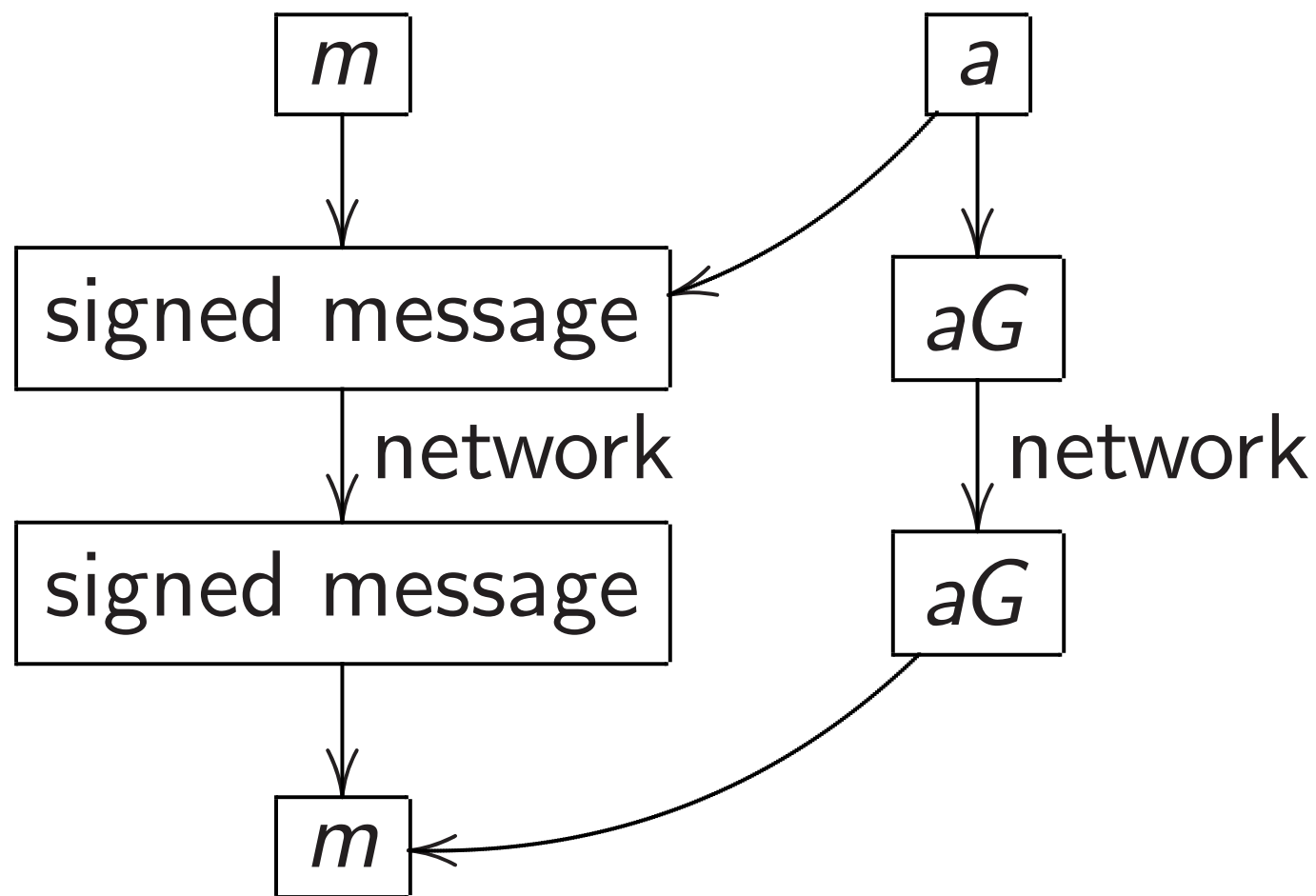
attacker was spying
communication of k ?

1:
key encryption.



7

Solution 2:
Public-key signatures.



No more shared secret k
but Alice still has secret a .
Cryptography requires TCB
to protect secrecy of keys,
even if user has no other secrets.

8

Constant

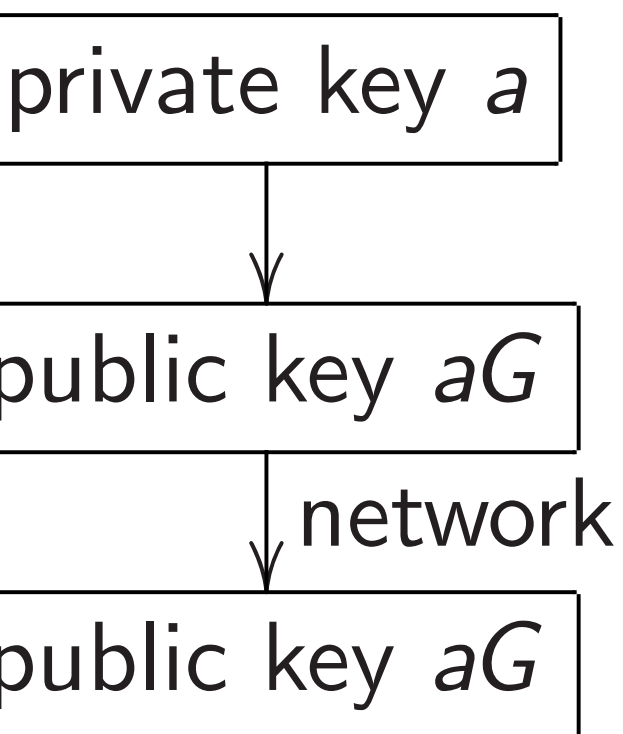
Large po
optimiza
addresse

Consider
instructi
parallel c
store-to-
branch p

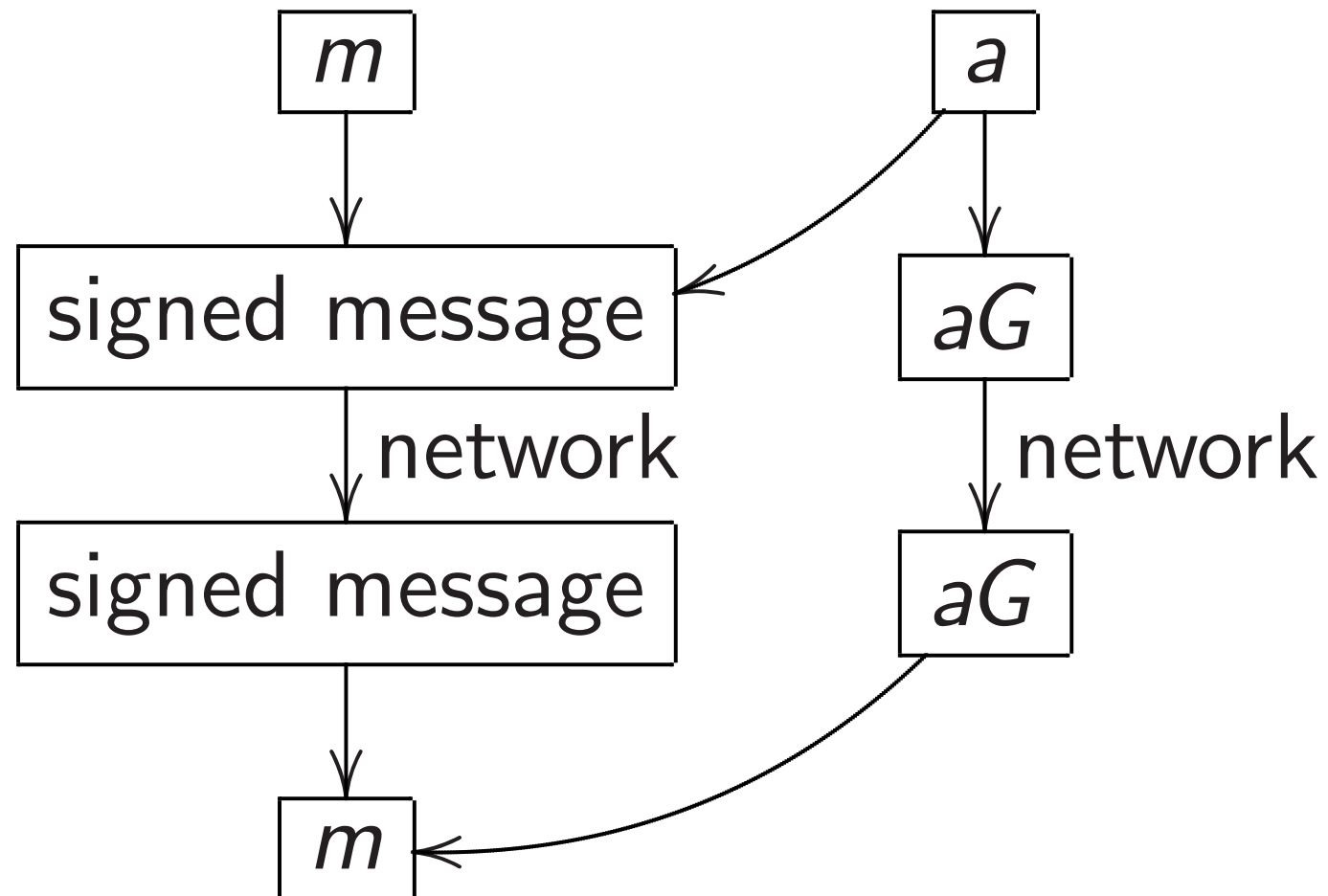
e and Bob
secret k .

was spying
cagation of k ?

tion.



Solution 2:
Public-key signatures.



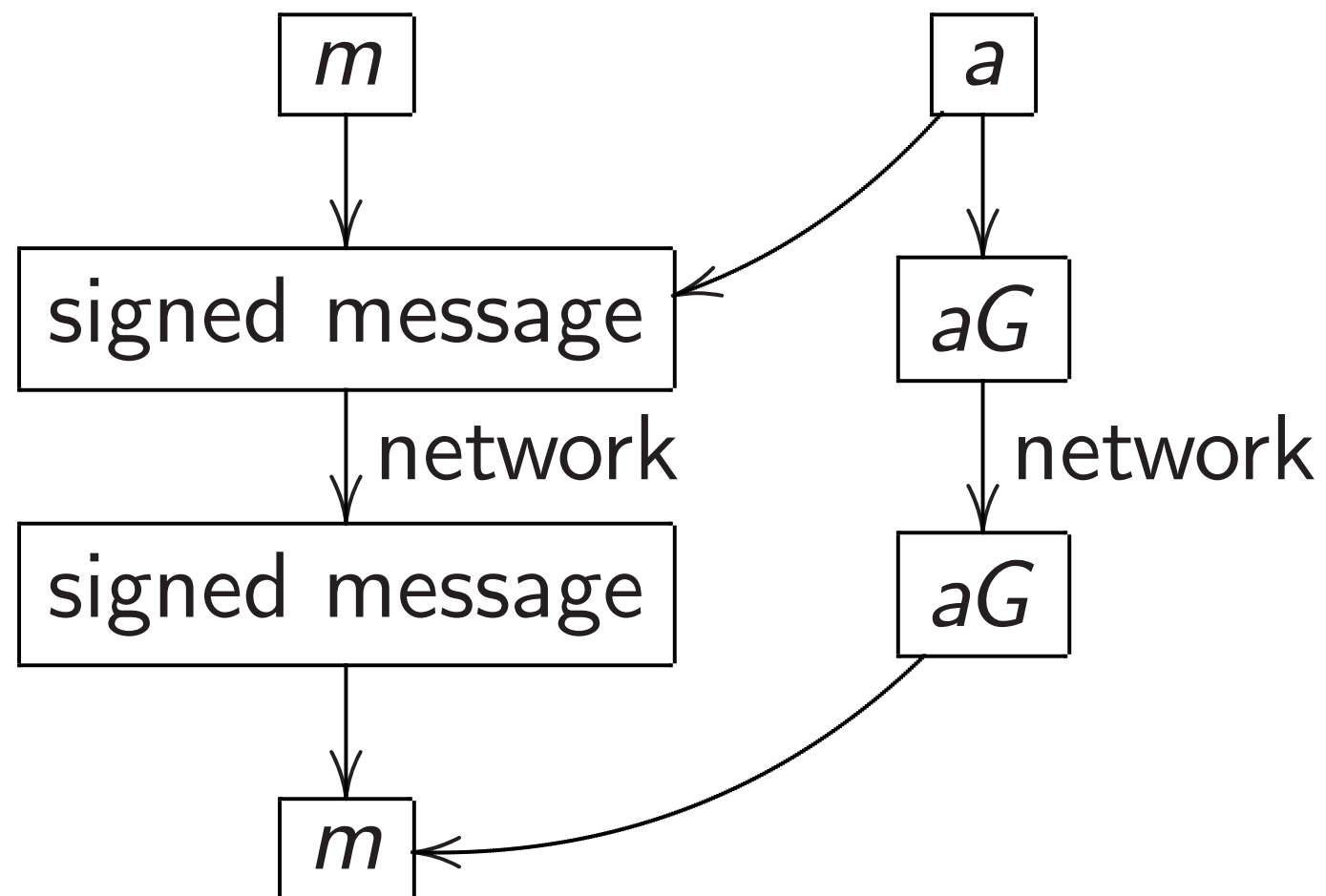
No more shared secret k
but Alice still has secret a .
Cryptography requires TCB
to protect secrecy of keys,
even if user has no other secrets.

Constant-time soft

Large portion of C
optimizations depe
addresses of memo

Consider data cach
instruction caching
parallel cache bank
store-to-load forward
branch prediction,

Solution 2:
Public-key signatures.



No more shared secret k
but Alice still has secret a .
Cryptography requires TCB
to protect secrecy of keys,
even if user has no other secrets.

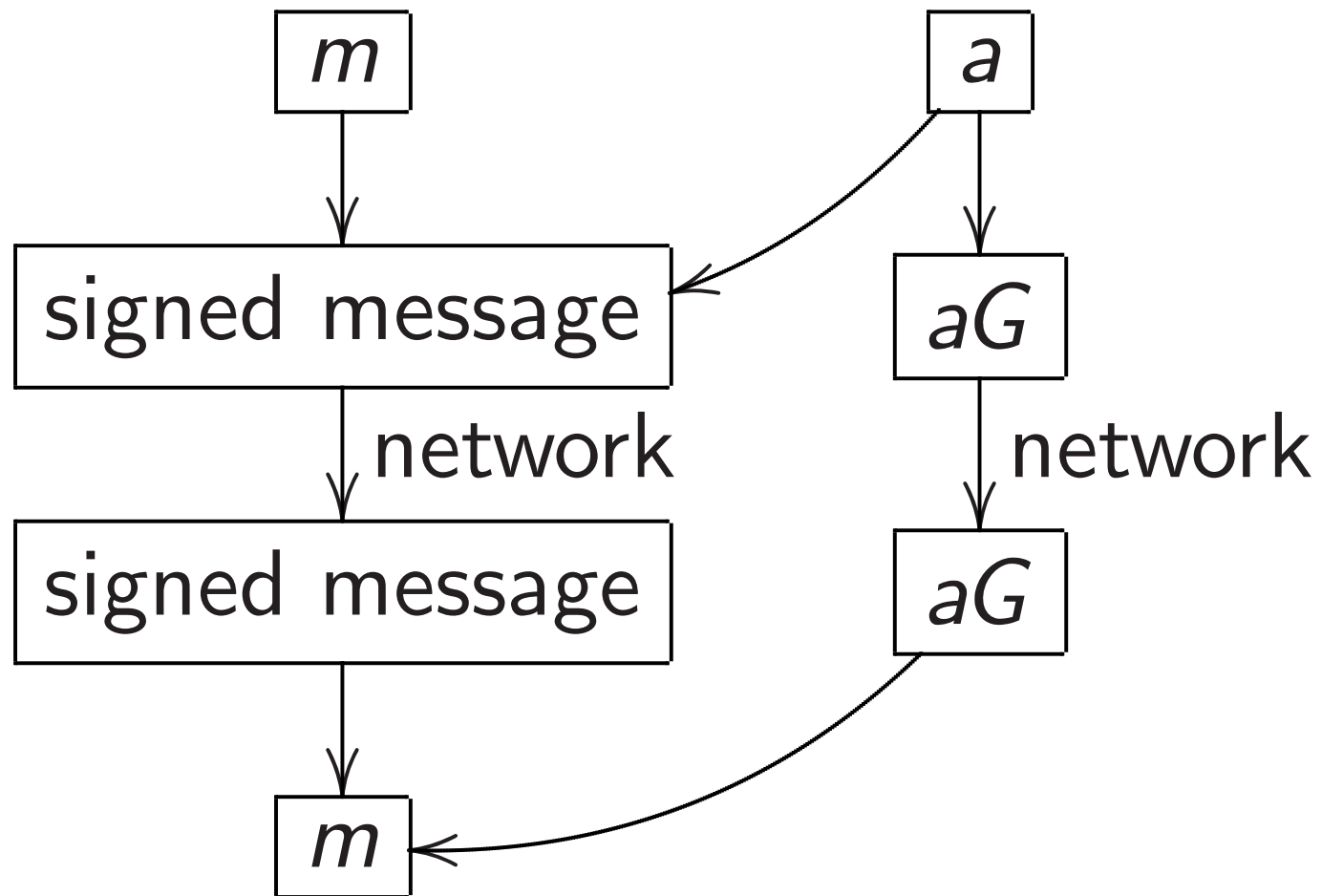
Constant-time software

Large portion of CPU hardware
optimizations depending on
addresses of memory location

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

Solution 2:

Public-key signatures.



No more shared secret k

but Alice still has secret a .

Cryptography requires TCB

to protect secrecy of keys,

even if user has no other secrets.

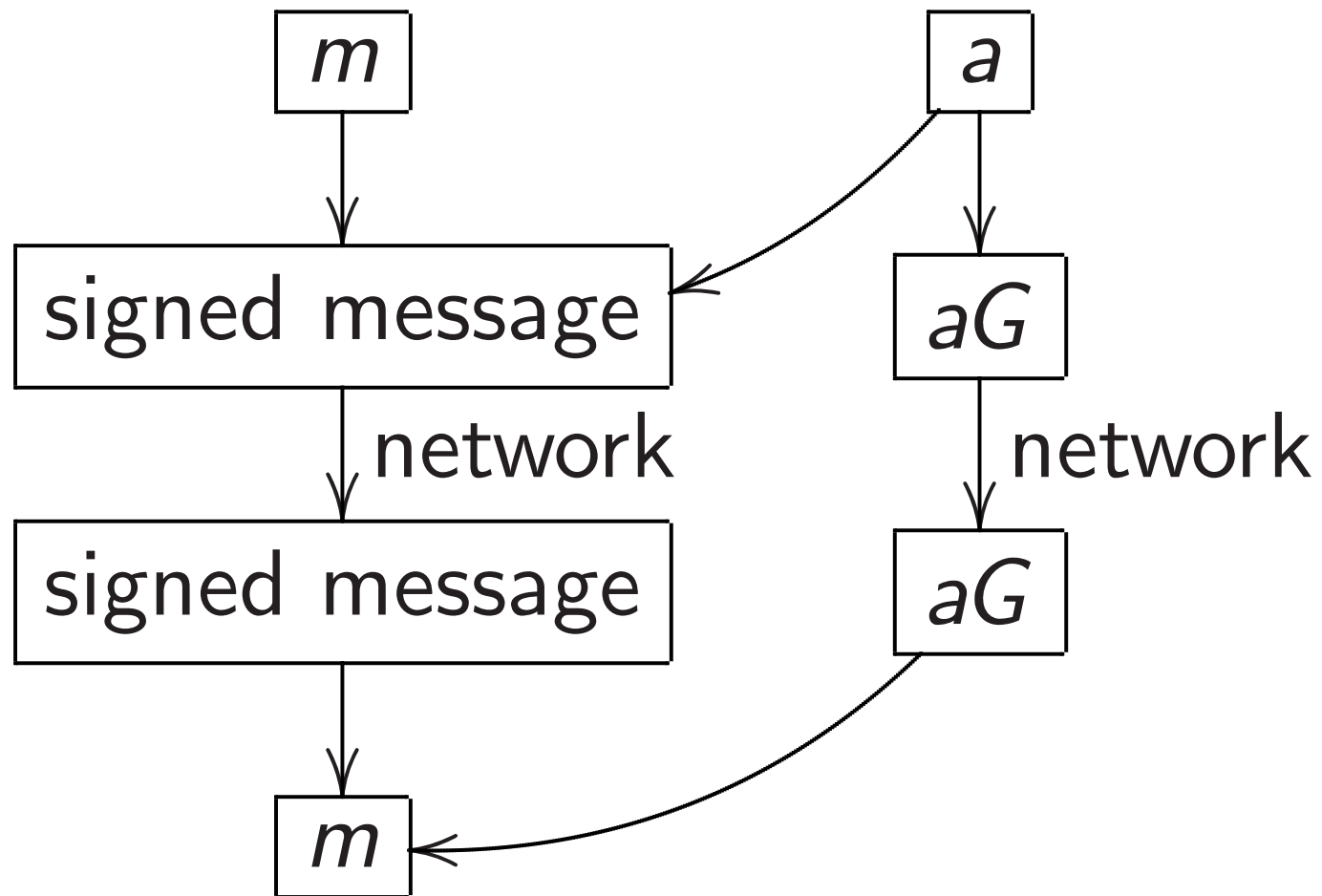
Constant-time software

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.

Solution 2:

Public-key signatures.



No more shared secret k
but Alice still has secret a .

Cryptography requires TCB
to protect secrecy of keys,
even if user has no other secrets.

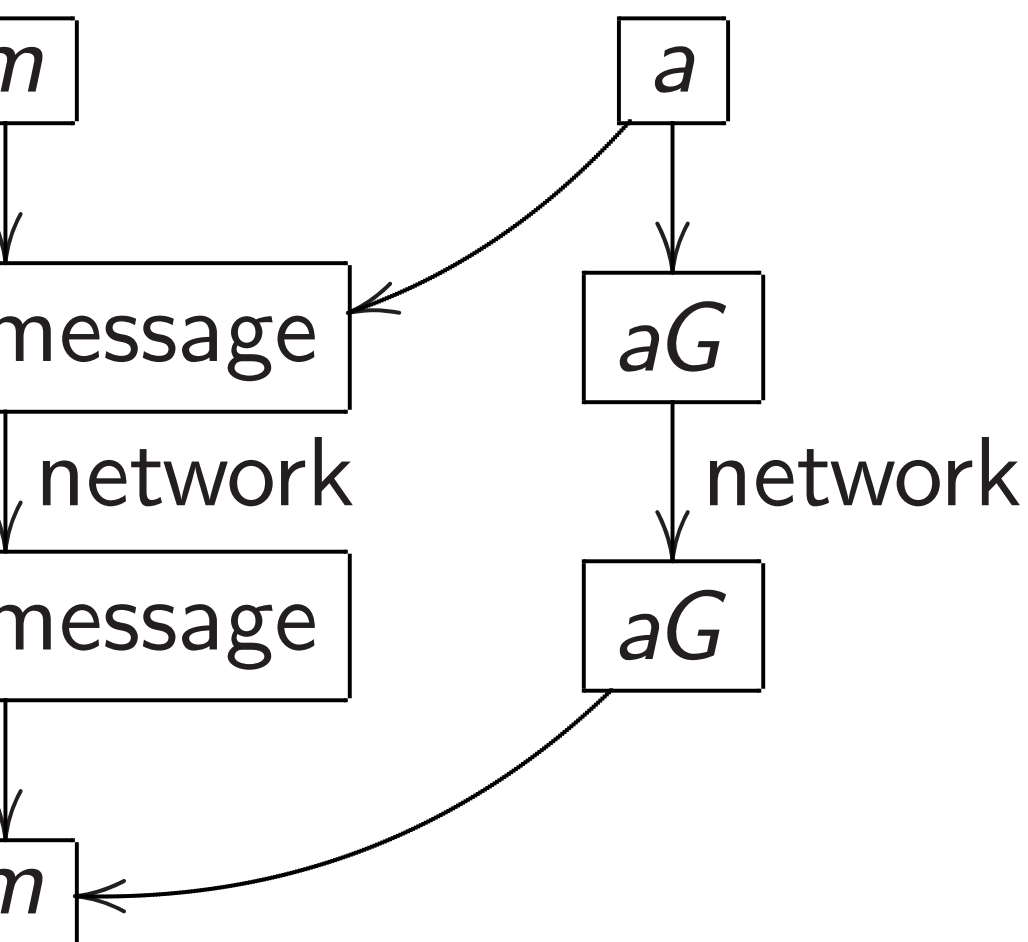
Constant-time software

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.

2:
key signatures.



shared secret k
still has secret a .

graphology requires TCB

ect secrecy of keys,

user has no other secrets.

8

Constant-time software

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.

9

Typical

Understa

But deta

not expo

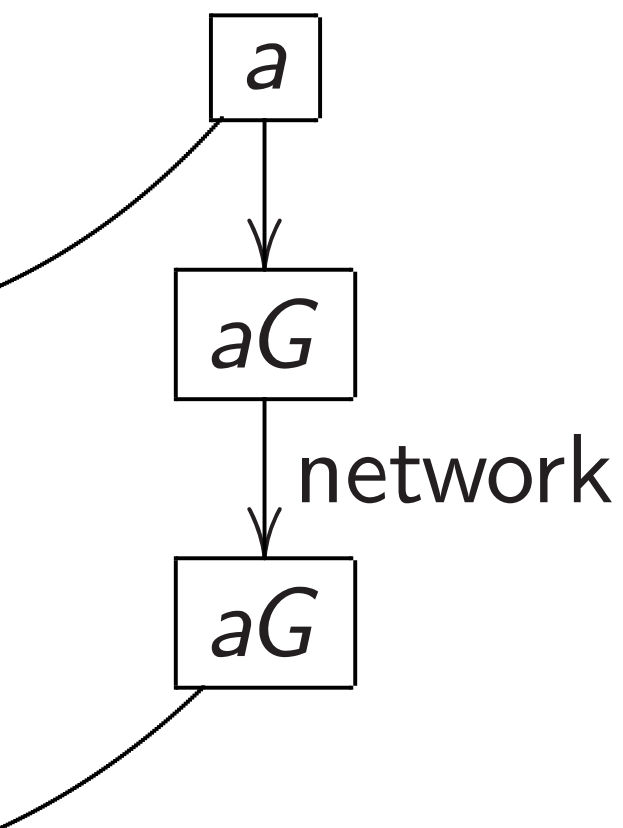
Try to p

This bec

Tweak t

to try to

res.

secret k secret a .**requires TCB****leakage of keys,**

to other secrets.

Constant-time software

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 po
But details are oft
not exposed to sec

Try to push attack
This becomes very

Tweak the attacke
to try to stop the

Constant-time software

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 review

Try to push attacks further.
This becomes very complicated

Tweak the attacked software
to try to stop the known att

Constant-time software

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.

Constant-time software

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!

Constant-time software

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.

Real-time software

Portion of CPU hardware:
Configurations depending on
Locations of memory locations.

Cache data caching,
Branch prediction,
Branch caching,
Cache banks,
Branch load forwarding,
Branch prediction, etc.

Attacks (e.g. TLBleed from
Spectre–Razavi–Bos–Giuffrida)
that this portion of the CPU
is unable to keep secrets.

9

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.

10

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

software

CPU hardware:

depending on
memory locations.

thing,

g,

ks,

arding,

etc.

. TLBleed from

–Bos–Giuffrida)

tion of the CPU

g secrets.

9

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.

10

The “constant-time

Don’t give any sec

to this portion of t

(1987 Goldreich, 1

Oblivious RAM; 20

domain-specific fo

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 Ostrom
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,

posed to security review.

ush attacks further.

comes very complicated.

he attacked software

o 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

Serious r

Attacker

breaking

public-ke

e.g., find

on this topic:

portion of CPU.

en proprietary,
security review.

ks further.

y complicated.

ed software

known attacks.

This is great!

is a nightmare.

urity failures.

uture 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

Serious risk within

Attacker has quan

breaking today’s m

public-key crypto

e.g., finding a give

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

Serious risk within 10 years:

Attacker has quantum comp

breaking today's most popul

public-key crypto (RSA and

e.g., finding a given aG).

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

Serious risk within 10 years:

Attacker has quantum computer

breaking today's most popular

public-key crypto (RSA and ECC;

e.g., finding a given aG).

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

Serious risk within 10 years:

Attacker has quantum computer
breaking today's most popular
public-key crypto (RSA and ECC;
e.g., finding a given aG).

2017: Hundreds of people
submit 69 complete proposals
to international competition for
post-quantum crypto standards.

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

Serious risk within 10 years:

Attacker has quantum computer

breaking today's most popular

public-key crypto (RSA and ECC;

e.g., finding a given aG).

2017: Hundreds of people

submit 69 complete proposals

to international competition for

post-quantum crypto standards.

Subroutine in some submissions:

sort array of secret integers.

e.g. sort 768 32-bit integers.

constant-time" solution:
 ve any secrets
 portion of the CPU.
 Goldreich, 1990 Ostrovsky:
 s RAM; 2004 Bernstein:
 specific for better speed)
 analysis: Need this portion
 PU to be correct, but
 ed it to keep secrets.
 auditing much easier.
 watch for attitude and
 ce of CPU designers: e.g.,
 ues errata for correctness
 ot for information leaks.

Case study: Constant-time sorting

Serious risk within 10 years:
 Attacker has quantum computer
 breaking today's most popular
 public-key crypto (RSA and ECC;
 e.g., finding a given aG).

2017: Hundreds of people
 submit 69 complete proposals
 to international competition for
 post-quantum crypto standards.

Subroutine in some submissions:
 sort array of secret integers.
 e.g. sort 768 32-bit integers.

How to
 without

ne" solution:

crets

the CPU.

1990 Ostrovsky:

2004 Bernstein:

(for better speed)

ed this portion

correct, but

keep secrets.

much easier.

attitude and

designers: e.g.,

for correctness

information leaks.

Case study: Constant-time sorting

Serious risk within 10 years:

Attacker has quantum computer
breaking today's most popular
public-key crypto (RSA and ECC;
e.g., finding a given aG).

2017: Hundreds of people
submit 69 complete proposals
to international competition for
post-quantum crypto standards.

Subroutine in some submissions:
sort array of secret integers.
e.g. sort 768 32-bit integers.

How to sort secret
without any secret

Case study: Constant-time sorting

Serious risk within 10 years:

Attacker has quantum computer
breaking today's most popular
public-key crypto (RSA and ECC;
e.g., finding a given aG).

2017: Hundreds of people
submit 69 complete proposals
to international competition for
post-quantum crypto standards.

Subroutine in some submissions:
sort array of secret integers.
e.g. sort 768 32-bit integers.

How to sort secret data
without any secret addresses

Case study: Constant-time sorting

Serious risk within 10 years:

Attacker has quantum computer breaking today's most popular public-key crypto (RSA and ECC; e.g., finding a given aG).

2017: Hundreds of people submit 69 complete proposals to international competition for post-quantum crypto standards.

Subroutine in some submissions: sort array of secret integers.
e.g. sort 768 32-bit integers.

How to sort secret data without any secret addresses?

Case study: Constant-time sorting

Serious risk within 10 years:

Attacker has quantum computer breaking today's most popular public-key crypto (RSA and ECC; e.g., finding a given aG).

2017: Hundreds of people submit 69 complete proposals to international competition for post-quantum crypto standards.

Subroutine in some submissions: sort array of secret integers.
e.g. sort 768 32-bit integers.

How to sort secret data without any secret addresses?

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

Case study: Constant-time sorting

Serious risk within 10 years:

Attacker has quantum computer breaking today's most popular public-key crypto (RSA and ECC; e.g., finding a given aG).

2017: Hundreds of people submit 69 complete proposals to international competition for post-quantum crypto standards.

Subroutine in some submissions: sort array of secret integers.
e.g. sort 768 32-bit integers.

How to sort secret data without any secret addresses?

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

One submission to competition: “Radix sort is used as **constant-time** sorting algorithm.”

Some versions of radix sort avoid secret branches.

Case study: Constant-time sorting

Serious risk within 10 years:

Attacker has quantum computer breaking today's most popular public-key crypto (RSA and ECC; e.g., finding a given aG).

2017: Hundreds of people submit 69 complete proposals to international competition for post-quantum crypto standards.

Subroutine in some submissions: sort array of secret integers.
e.g. sort 768 32-bit integers.

How to sort secret data without any secret addresses?

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

One submission to competition: “Radix sort is used as **constant-time** sorting algorithm.”

Some versions of radix sort avoid secret branches.

But data addresses in radix sort still depend on secrets.

Today: Constant-time sorting

risk within 10 years:

has quantum computer

today's most popular

crypto (RSA and ECC;

finding a given aG).

hundreds of people

59 complete proposals

national competition for

quantum crypto standards.

line in some submissions:

y of secret integers.

768 32-bit integers.

How to sort secret data
without any secret addresses?

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

One submission to competition:

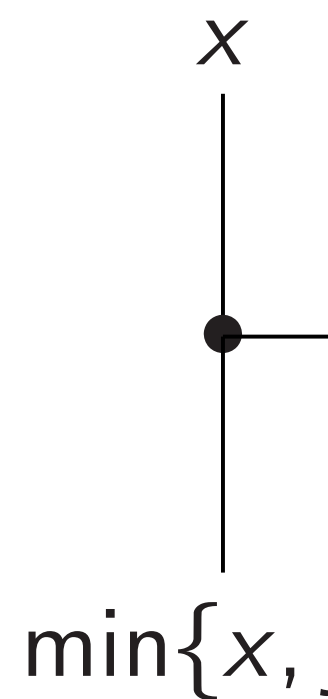
“Radix sort is used as
constant-time sorting algorithm.”

Some versions of radix sort
avoid secret branches.

But data addresses in radix sort
still depend on secrets.

Foundat

a **compa**



Easy con

Warning

compiler

Even eas

constant-time sorting

10 years:

quantum computer

most popular

(RSA and ECC;
even aG).

of people

submitted proposals

in competition for

crypto standards.

of submissions:

for integers.

for integers.

How to sort secret data
without any secret addresses?

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

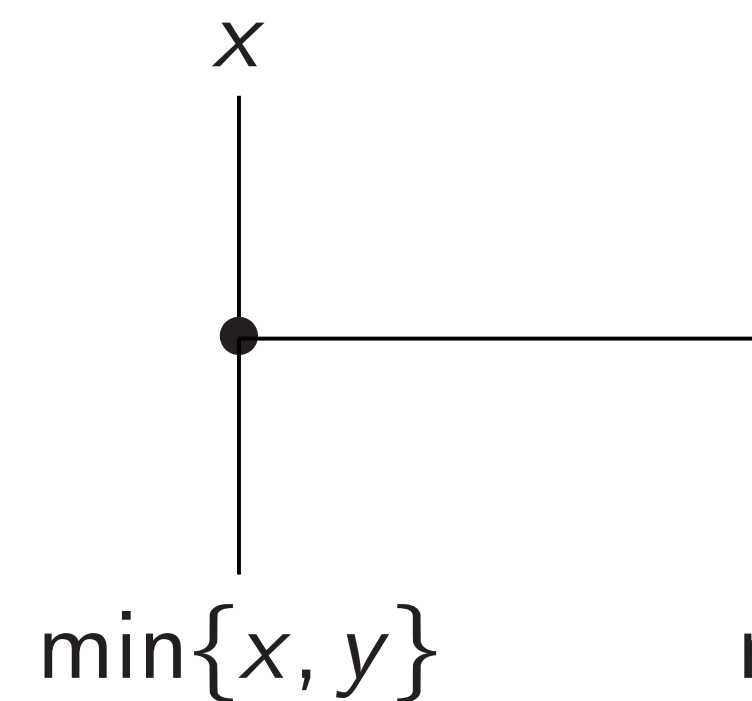
One submission to competition:

“Radix sort is used as
constant-time sorting algorithm.”

Some versions of radix sort
avoid secret branches.

But data addresses in radix sort
still depend on secrets.

Foundation of solution
is a **comparator** sorting



Easy constant-time

Warning: C standard

compiler to screw

Even easier exercise

How to sort secret data
without any secret addresses?

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

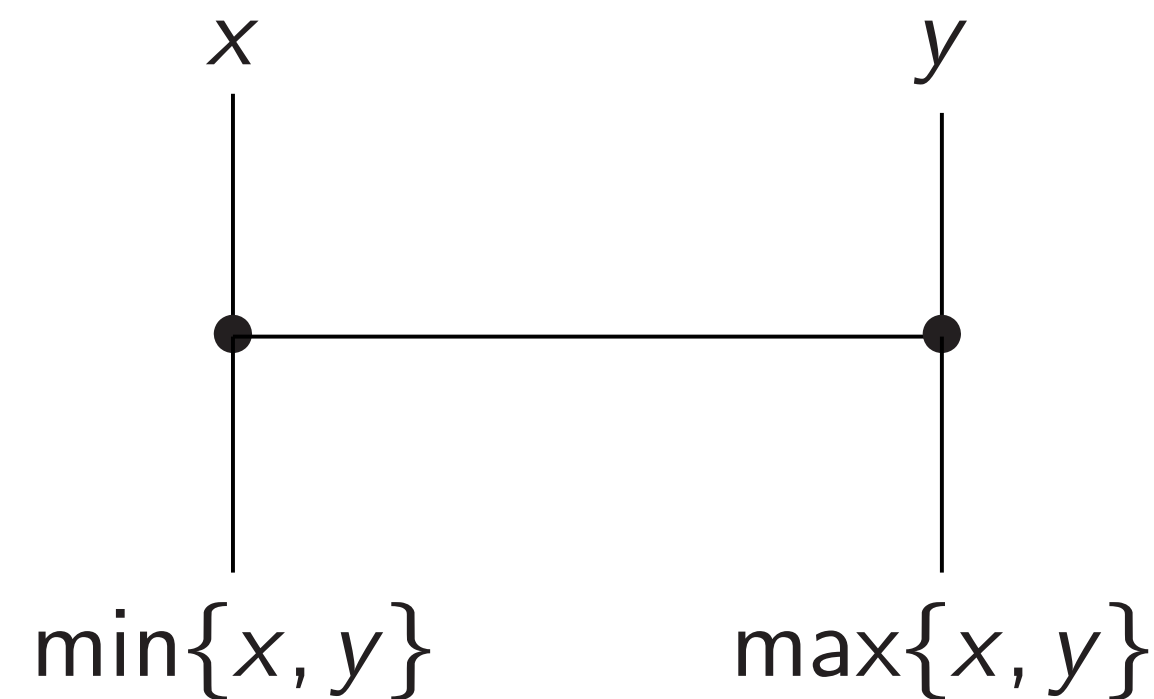
One submission to competition:

“Radix sort is used as
constant-time sorting algorithm.”

Some versions of radix sort
avoid secret branches.

But data addresses in radix sort
still depend on secrets.

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



Easy constant-time exercise
Warning: C standard allows
compiler to screw this up.

Even easier exercise in asm.

How to sort secret data
without any secret addresses?

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

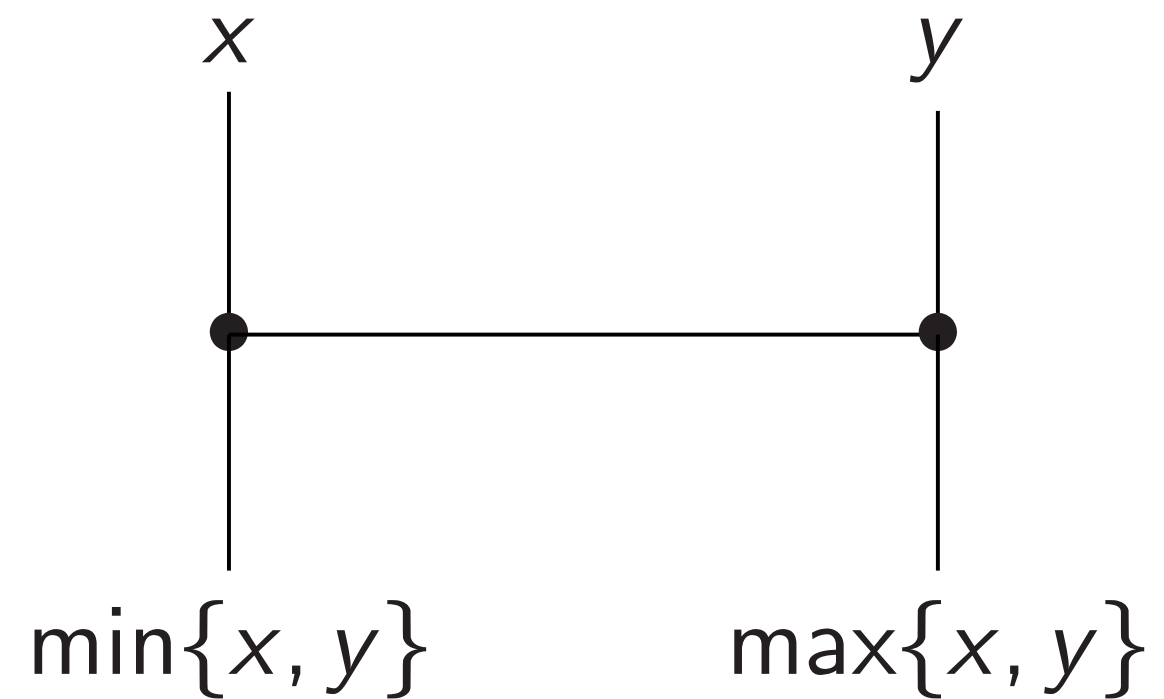
One submission to competition:

“Radix sort is used as
constant-time sorting algorithm.”

Some versions of radix sort
avoid secret branches.

But data addresses in radix sort
still depend on secrets.

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



Easy constant-time exercise in C.

Warning: C standard allows
compiler to screw this up.

Even easier exercise in asm.

sort secret data

any secret addresses?

sorting algorithms—

ort, quicksort, etc.—

oad/store addresses

n secret data. Usually

nch based on secret data.

mission to competition:

sort is used as

constant-time sorting algorithm.”

ersions of radix sort

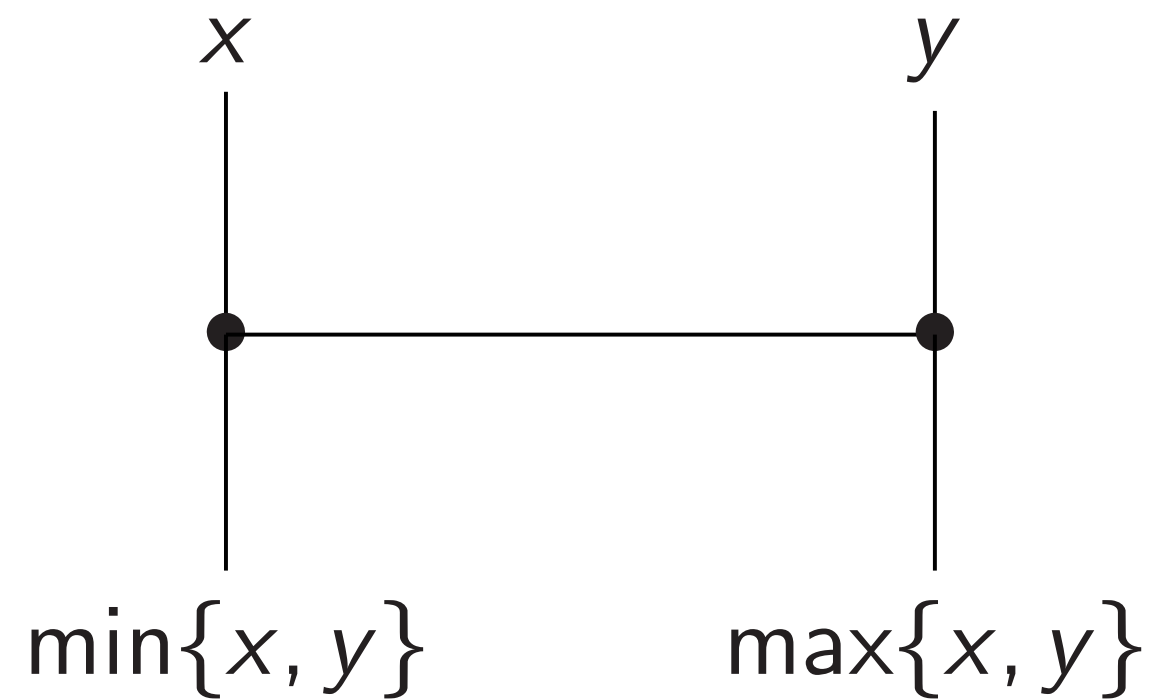
cret branches.

a addresses in radix sort

end on secrets.

Foundation of solution:

a **comparator** sorting 2 integers.



Easy constant-time exercise in C.

Warning: C standard allows compiler to screw this up.

Even easier exercise in asm.

Combined

sorting

Example



secret data
secret addresses?
algorithms—
sort, etc.—
secret addresses
secret data. Usually
on secret data.

competition:

and as
"sorting algorithm."

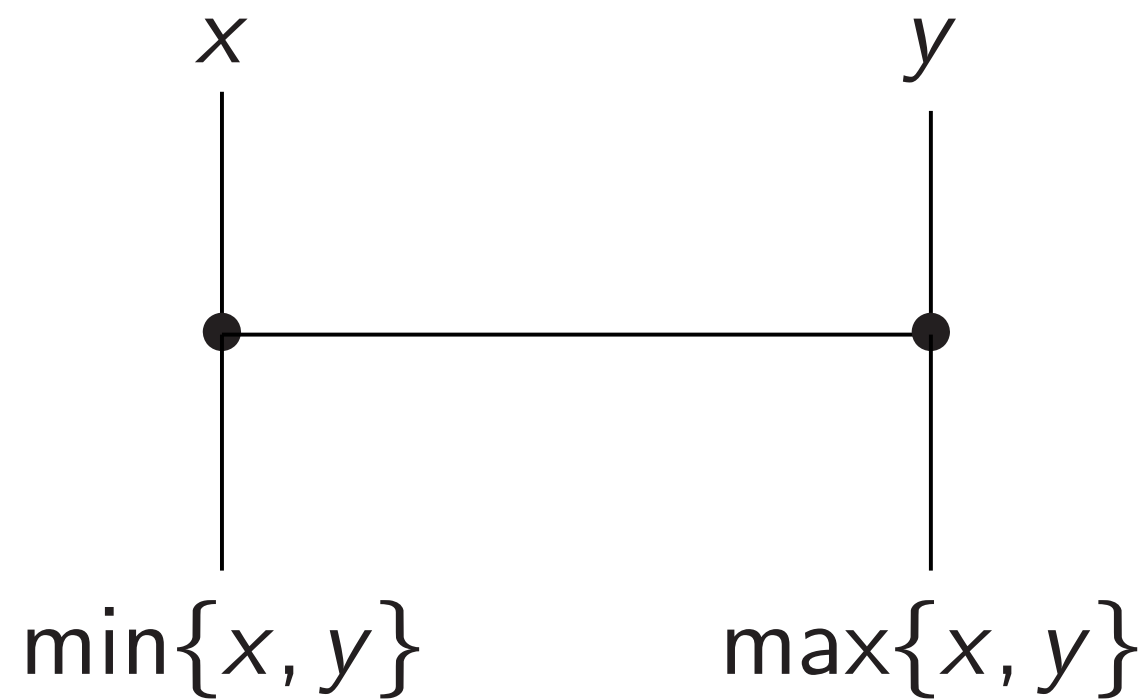
radix sort

thes.

s in radix sort

crets.

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

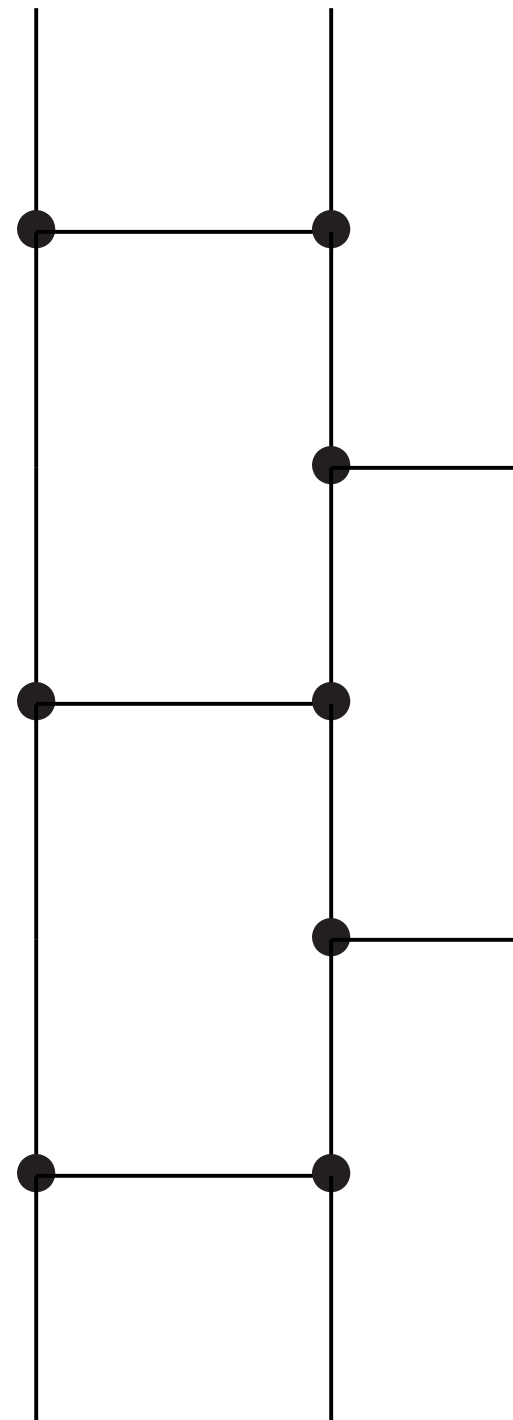


Easy constant-time exercise in C.
Warning: C standard allows
compiler to screw this up.

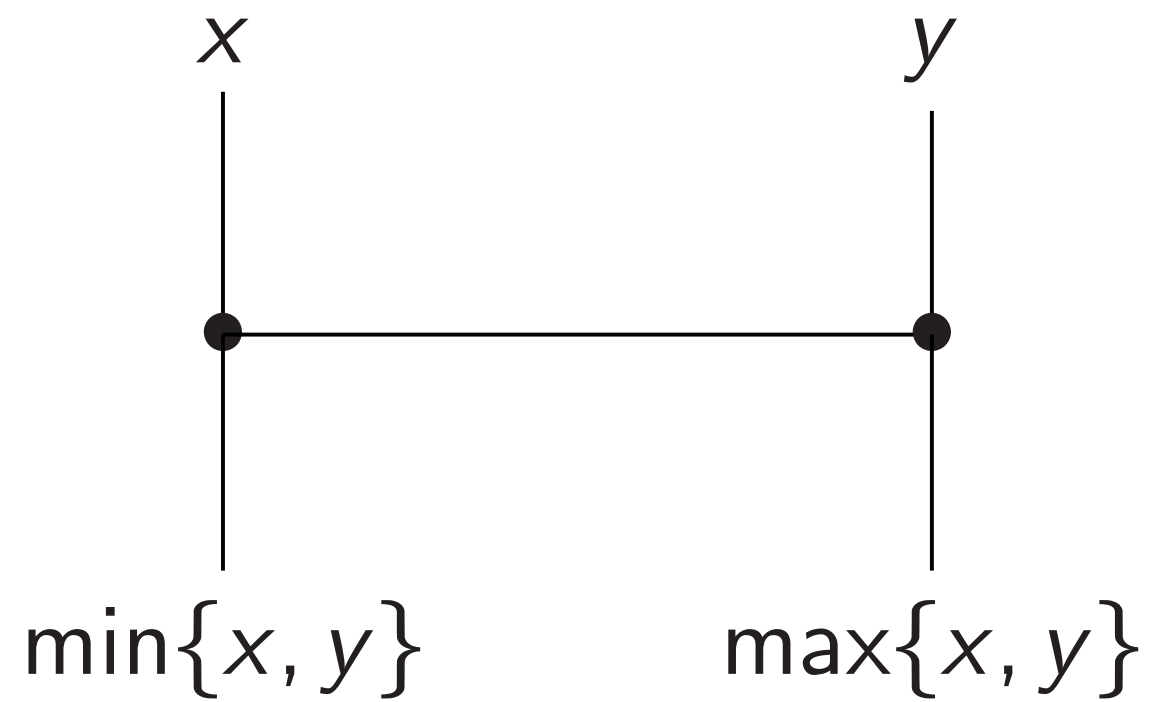
Even easier exercise in asm.

Combine comparators
sorting network for

Example of a sorting



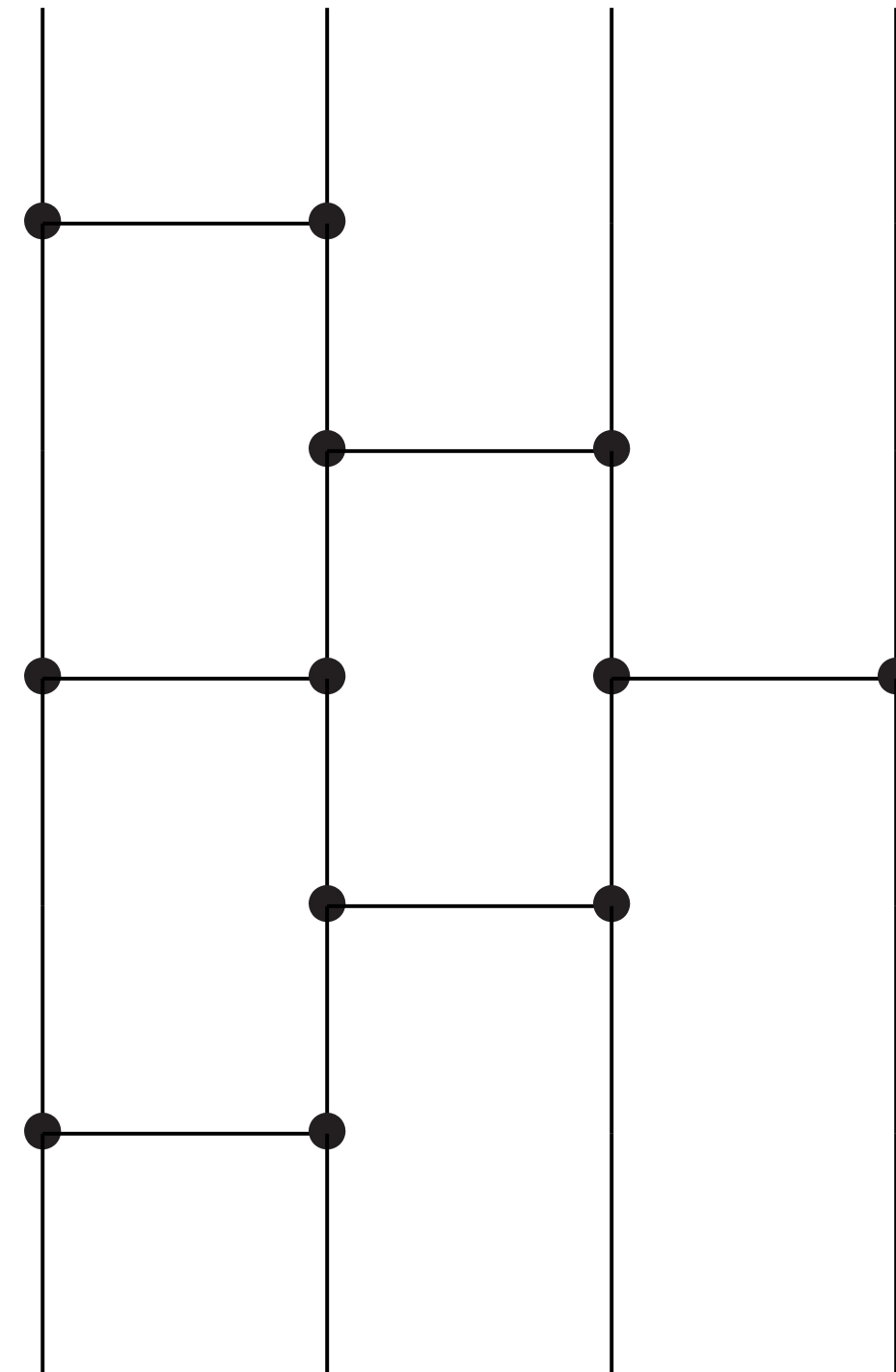
Foundation of solution:
a **comparator** sorting 2 integers.



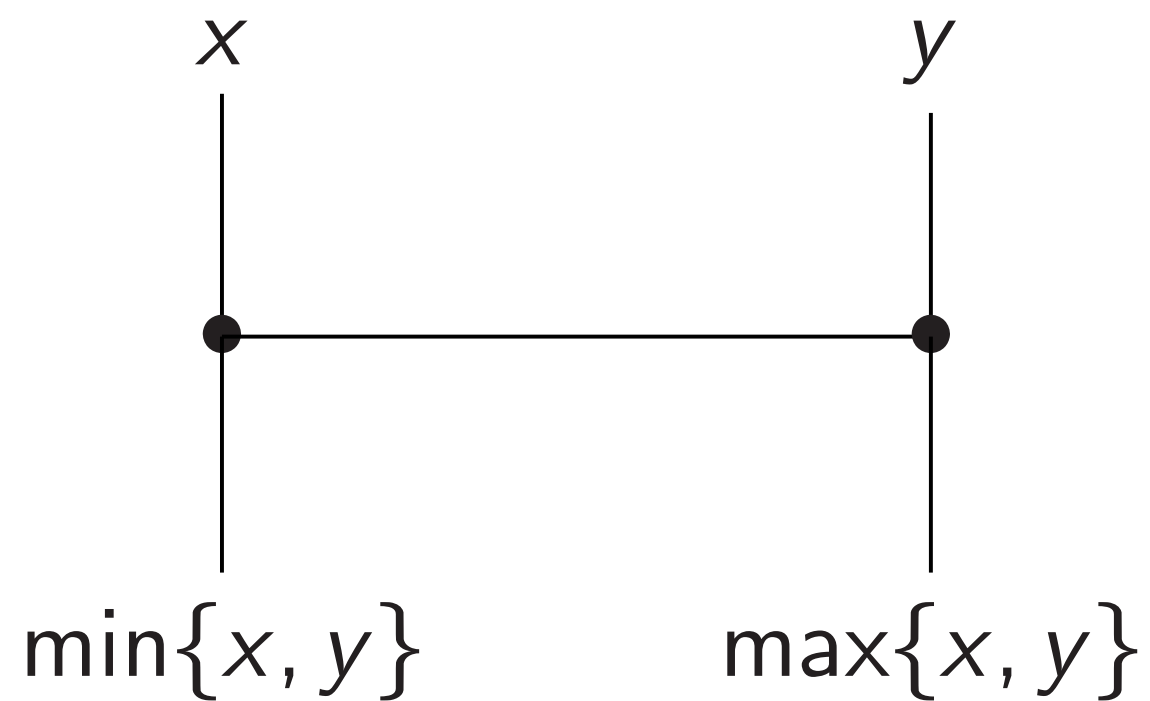
Easy constant-time exercise in C.
Warning: C standard allows
compiler to screw this up.
Even easier exercise in asm.

Combine comparators into a
sorting network for more in

Example of a sorting network



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



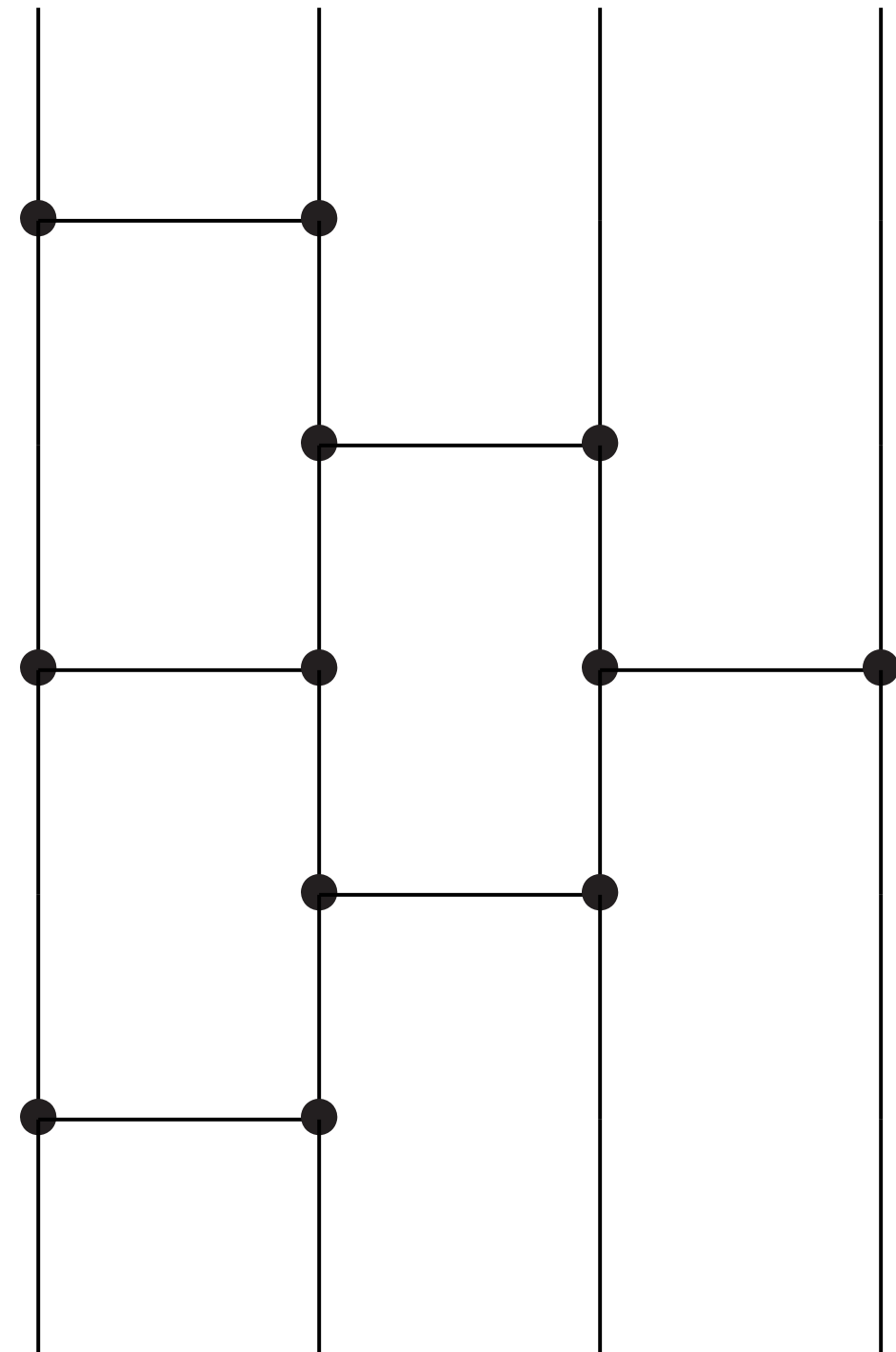
Easy constant-time exercise in C.

Warning: C standard allows
compiler to screw this up.

Even easier exercise in asm.

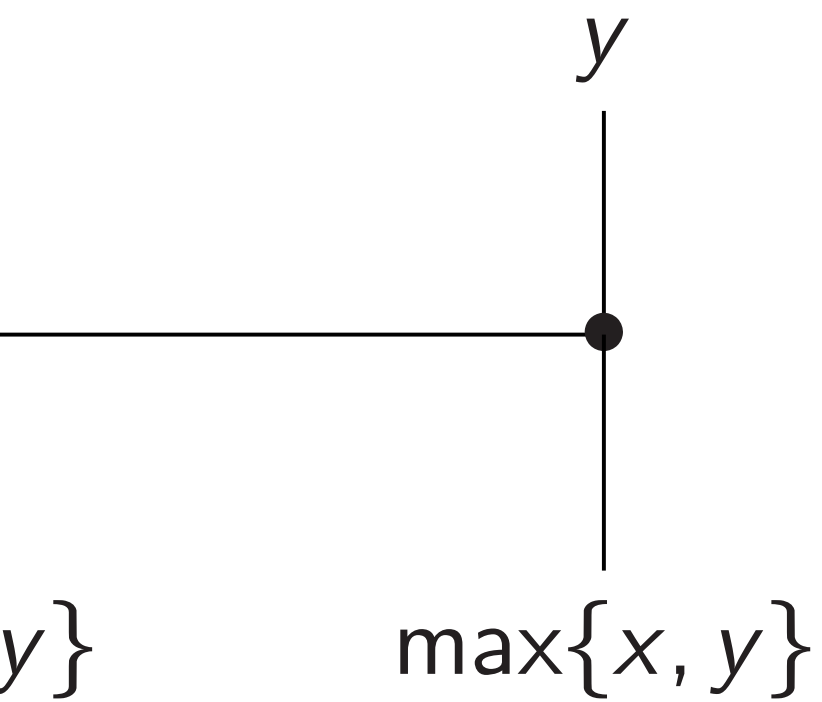
Combine comparators into a
sorting network for more inputs.

Example of a sorting network:



ion of solution:

Comparator sorting 2 integers.



constant-time exercise in C.

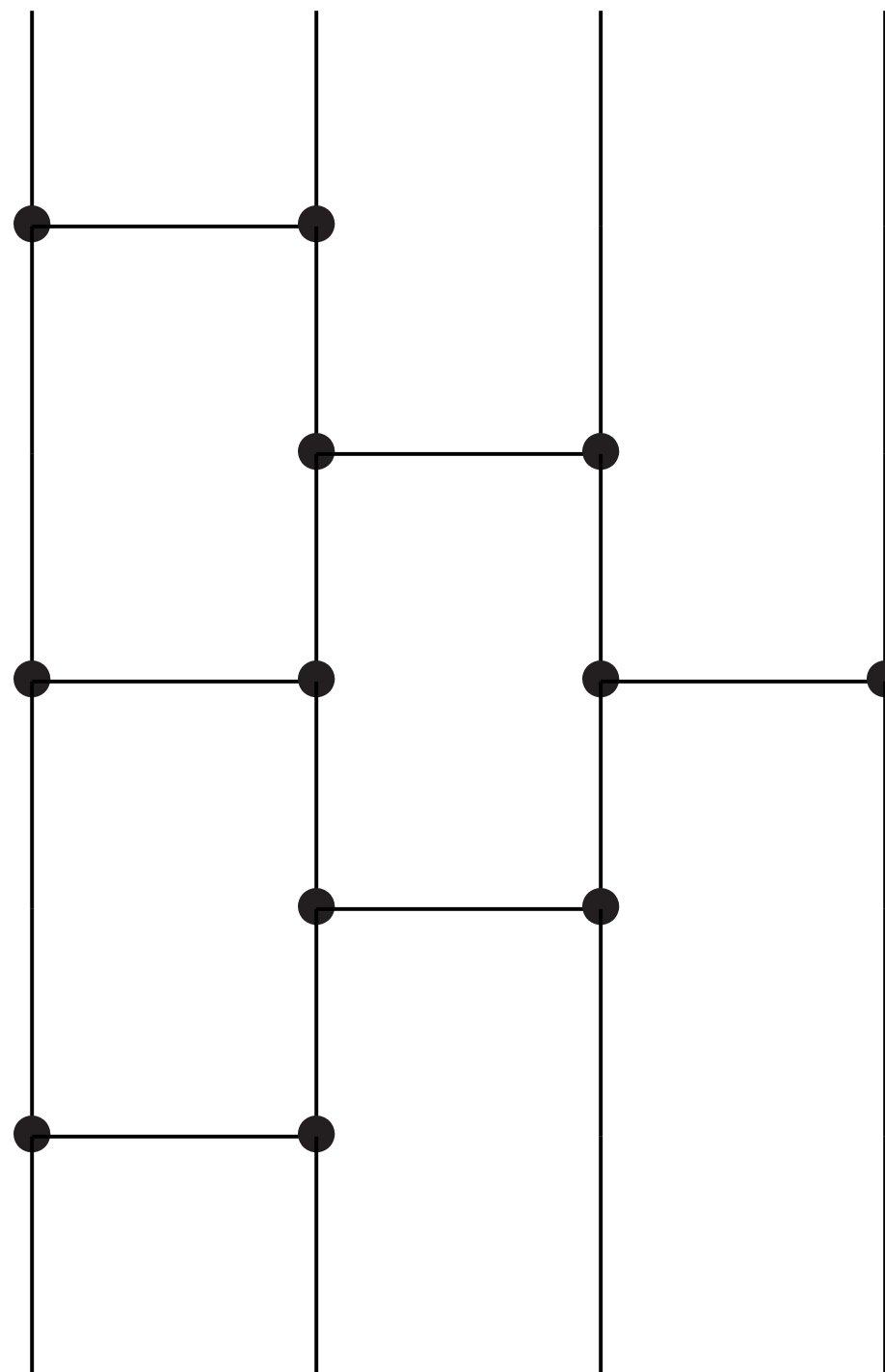
: C standard allows

to screw this up.

similar exercise in asm.

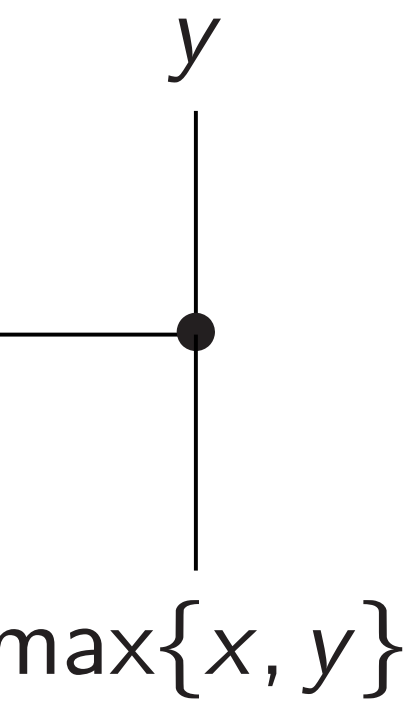
Combine comparators into a
sorting network for more inputs.

Example of a sorting network:



Positions
in a sort
independ
Naturally

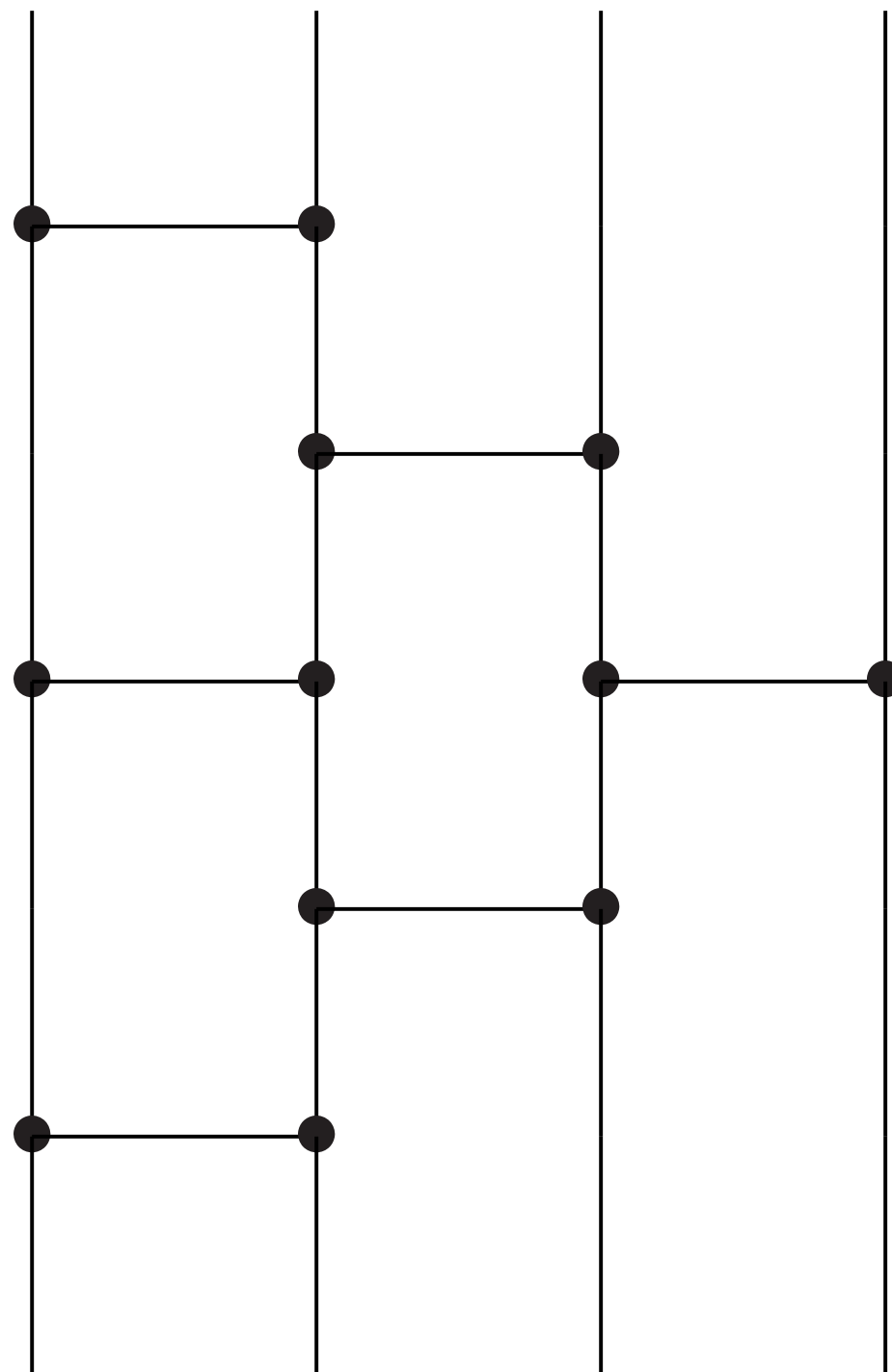
ation:
 ting 2 integers.



e exercise in C.
 ard allows
 this up.
 se in asm.

Combine comparators into a
sorting network for more inputs.

Example of a sorting network:

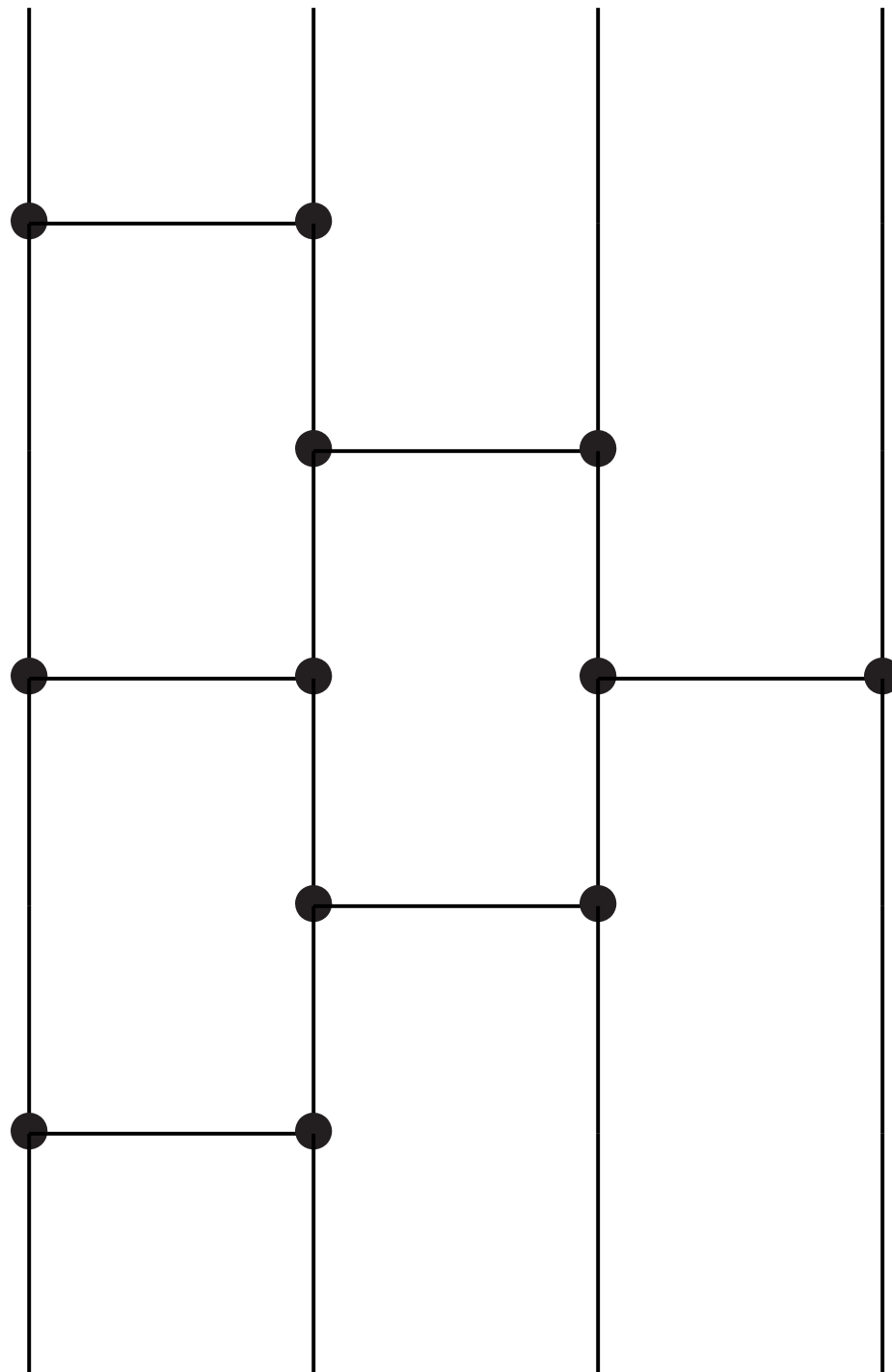


Positions of compa
 in a sorting network
 independent of the
 Naturally constant

egers.

Combine comparators into a
sorting network for more inputs.

Example of a sorting network:

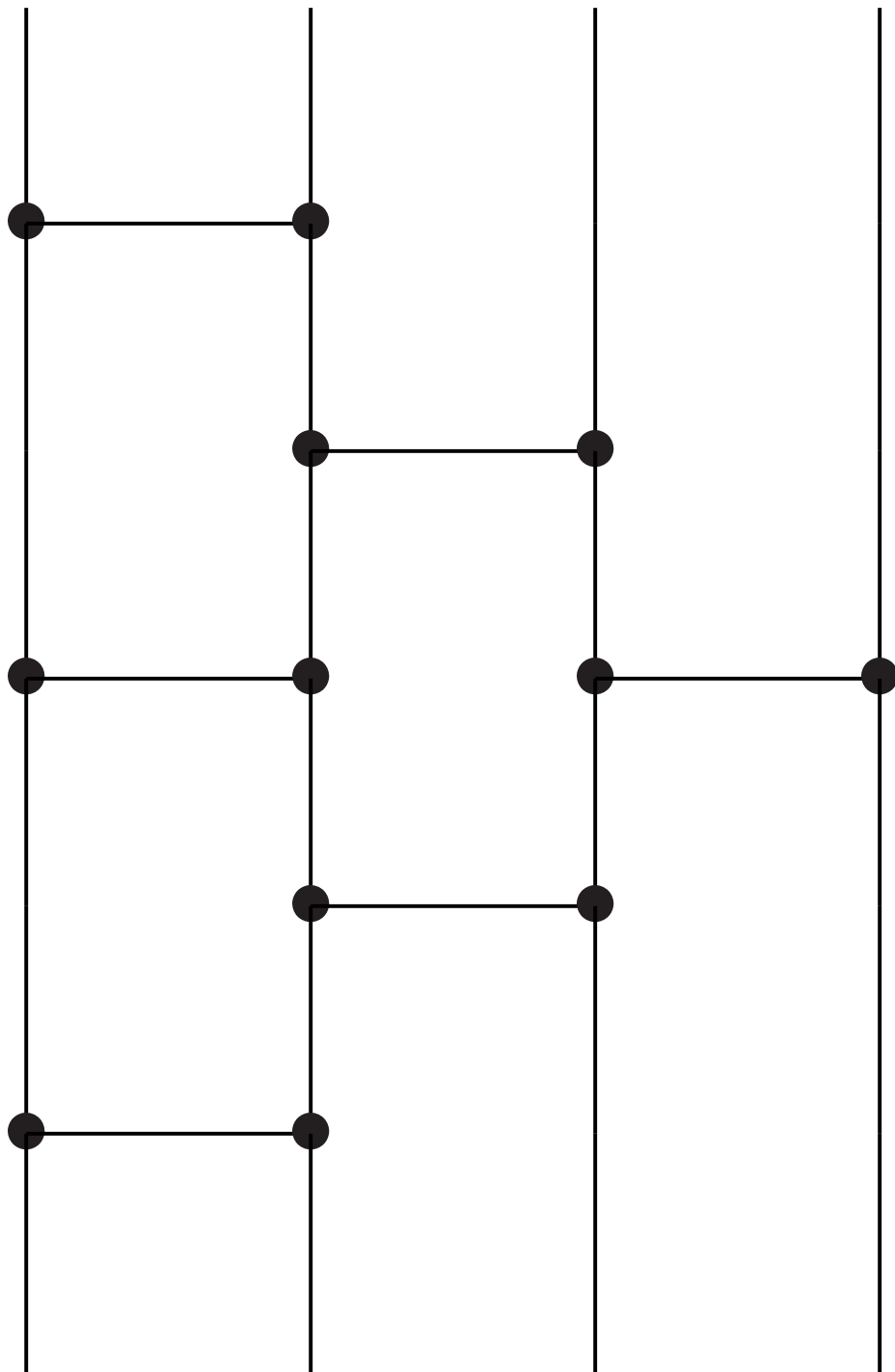


in C.

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

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

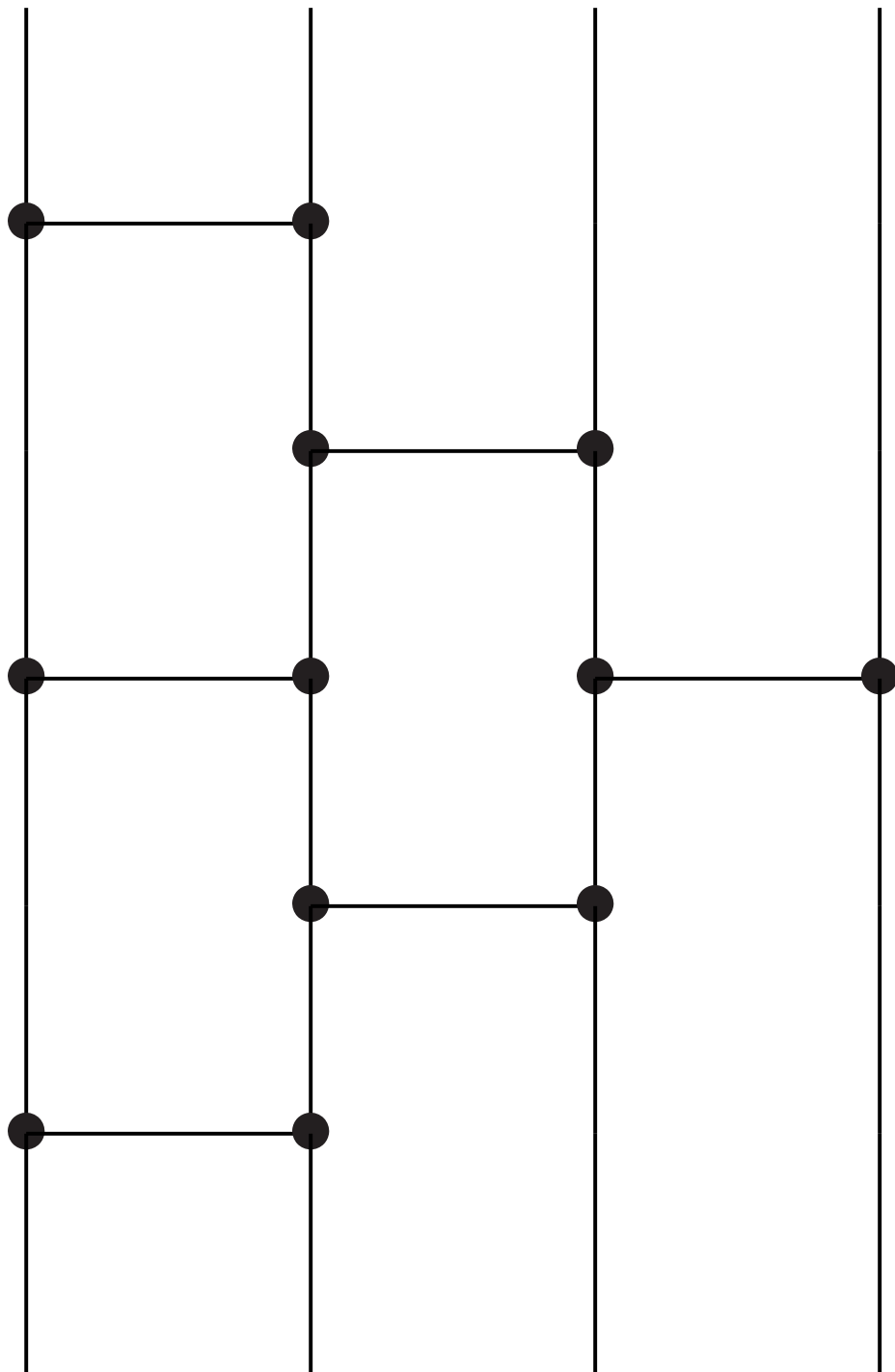
Example of a sorting network:



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:

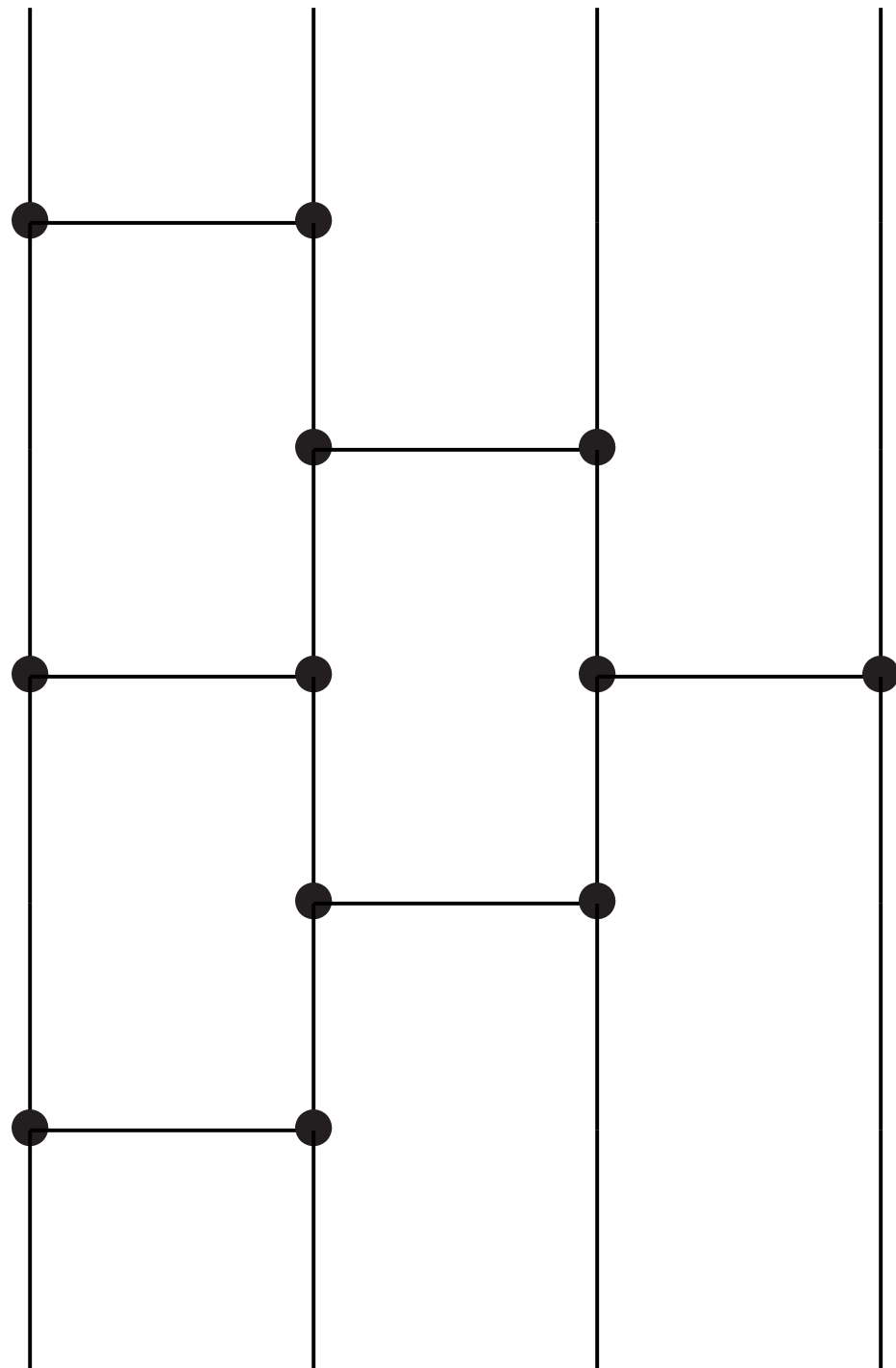


Positions of comparators in a sorting network are independent of the input. Naturally constant-time.

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

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.

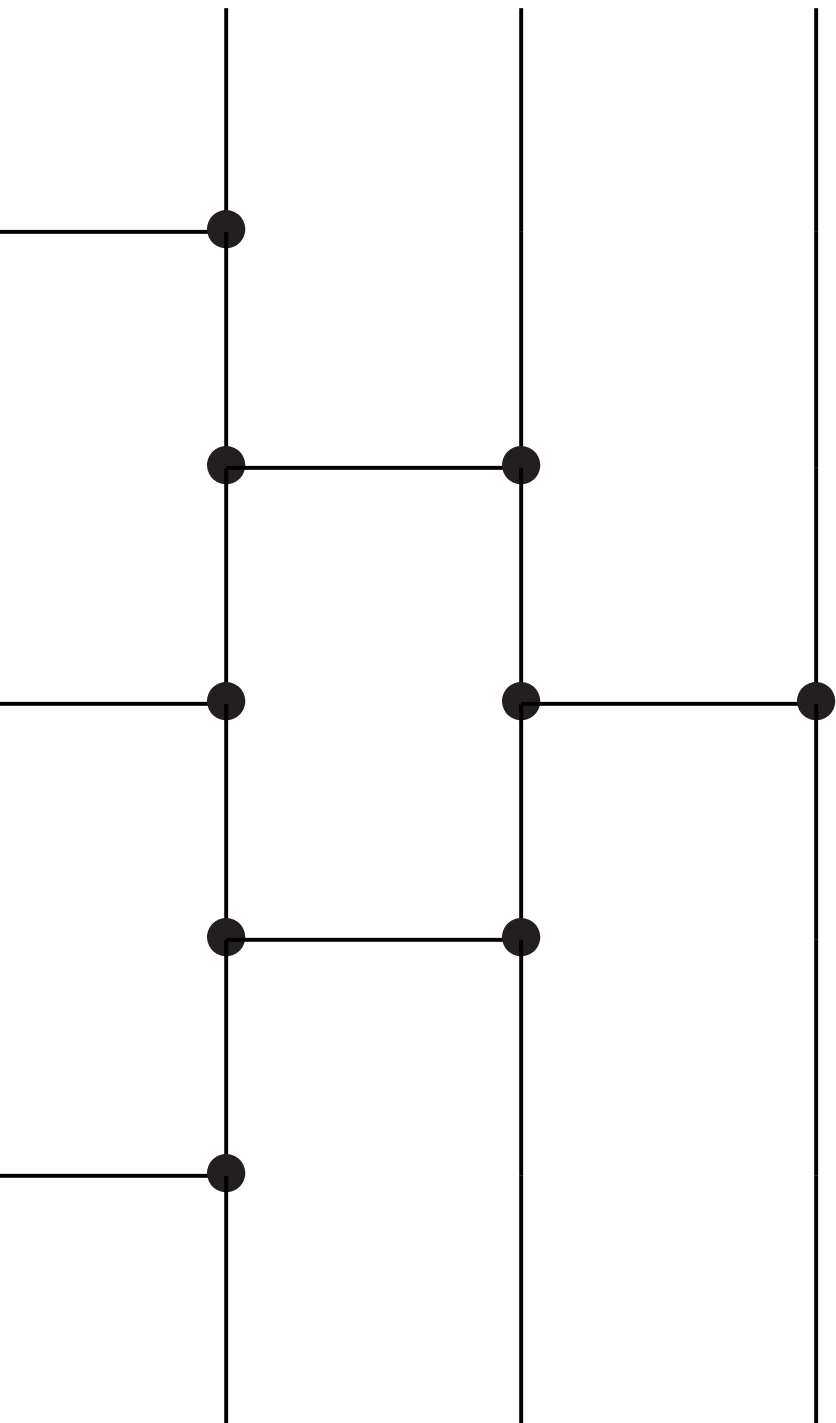
But $(n^2 - n)/2$ comparators produce complaints about performance as n increases.

Speed is a serious issue in the post-quantum competition.

“Cost” is evaluation criterion; “we’d like to stress this once again on the forum that we’d really like to see more platform-optimized implementations”; etc.

comparators into a
network for more inputs.

of a sorting network:



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

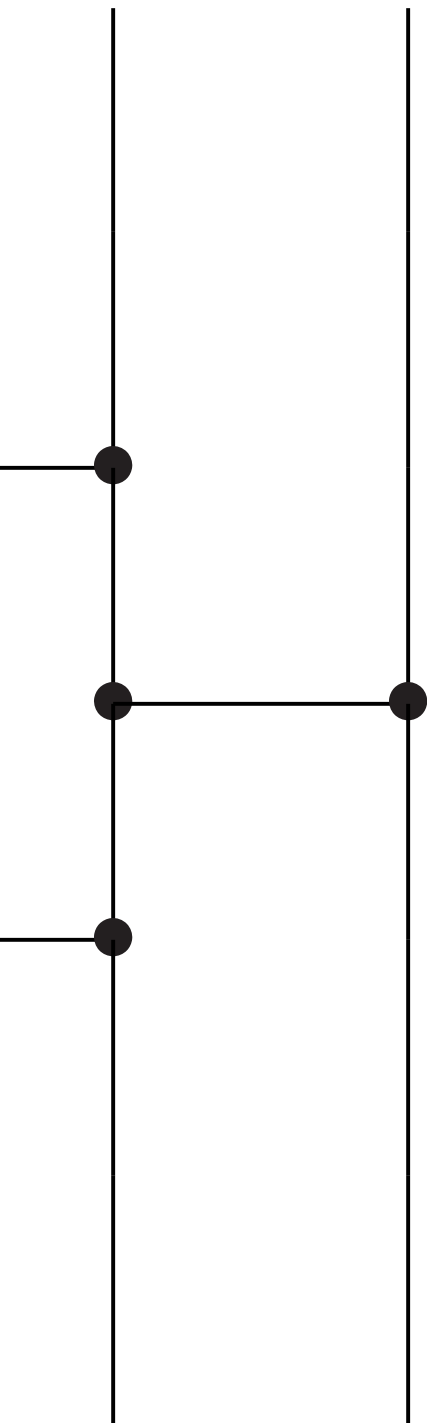
But $(n^2 - n)/2$ comparators
produce complaints about
performance as n increases.

Speed is a serious issue in the
post-quantum competition.

“Cost” is evaluation criterion;
“we’d like to stress this once
again on the forum that we’d
really like to see more platform-
optimized implementations”; etc.

```
void int
{ int64
  if (n
    t = 1
  while
  for (j
    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 $(n^2 - n)/2$ comparators
produce complaints about
performance as n increases.

Speed is a serious issue in the
post-quantum competition.

“Cost” is evaluation criterion;
“we’d like to stress this once
again on the forum that we’d
really like to see more platform-
optimized implementations”; etc.

```
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
    }
  }
}
```

inputs.

k:

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

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

Speed is a serious issue in the post-quantum competition.

“Cost” is evaluation criterion; “we’d like to stress this once again on the forum that we’d really like to see more platform-optimized implementations”; etc.

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

Speed is a serious issue in the post-quantum competition.

“Cost” is evaluation criterion; “we’d like to stress this once again on the forum that we’d really like to see more platform-optimized implementations”; etc.

```
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.

$(n - 1)/2$ comparators
complaints about
ance as n increases.

a serious issue in the
antum competition.

s evaluation criterion;
ke to stress this once

n the forum that we'd

re to see more platform-
ed implementations"; etc.

```
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 r
 $\approx n(\log_2$
Much fa

Warning
of Batch
require n
Also, W

networks
handling

comparators
 work are
 the input.
 time.
 comparators
 about
 increases.
 issue in the
 competition.
 on criterion;
 s this once
 n that we'd
 more platform-
 entations"; etc.

```

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 "ode
 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

```

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 exchange
 which is a simplified version
 1968 Batcher “odd-even me
 sorting networks.

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

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.**”

```

17
int32_sort(int32 *x, int64 n)
{
    t, p, q, i;
    if (n < 2) return;
    ;
    (t < n - t) t += t;
    p = t; p > 0; p >>= 1) {
        (i = 0; i < n - p; ++i)
        if (!(i & p))
            minmax(x+i, x+i+p);
        (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.**”

18

This co

Const

Berns

La

“NTRU

const

```

int32 *x,int64 n)
return;

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-time

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

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

vectorization
(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 coo

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

slide: C translation of
 with “merge exchange”,
 a simplified version of
 tcher “odd-even merge”
 networks.

$n)^2/4$ comparators.

ster than bubble sort.

: many other descriptions
 er’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

2015 Gu

AVX2 (H

quicksor

≈45 cyc

≈55 cyc

Slower t

impleme

the faste

aware of

IPP: Inte

Performa

translation of
 "merge exchange",
 "odd-even merge"

comparators.

bubble sort.

Other descriptions

of networks

with power of 2.

It says "Sorting

is not capable of

handling 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-sorting

2015 Gueron–Kras

AVX2 (Haswell) on

quicksort. For 32-

≈ 45 cycles/byte for

≈ 55 cycles/byte for

Slower than "the r

implemented of IPP

the fastest in-mem

aware of": 32, 40

IPP: Intel's Integra

Performance Prim

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 literature

2015 Gueron–Krasnov: AVX

AVX2 (Haswell) optimization

quicksort. For 32-bit integer

≈ 45 cycles/byte for $n \approx 2^{10}$

≈ 55 cycles/byte for $n \approx 2^{20}$

Slower than “the radix sort

implemented of IPP, which is

the fastest in-memory sort w

aware of”: 32, 40 cycles/by

IPP: Intel’s Integrated

Performance Primitives libra

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.

2015 Gueron–Krasnov: AVX and AVX2 (Haswell) optimization of quicksort. For 32-bit integers:
 ≈ 45 cycles/byte for $n \approx 2^{10}$,
 ≈ 55 cycles/byte for $n \approx 2^{20}$.

Slower than “the radix sort implemented of IPP, which is the fastest in-memory sort we are aware of”: 32, 40 cycles/byte.

IPP: Intel’s Integrated Performance Primitives library.

constant-time sorting code

vectorization
(for Haswell)

constant-time sorting code
included in 2017

stein–Chuengsatiansup–
nge–van Vredendaal

“Prime” software release

revamped for
higher speed

New: “djbsort”

constant-time sorting code

The slowdown for constant time

Massive fast-sorting literature.

2015 Gueron–Krasnov: AVX and
AVX2 (Haswell) optimization of
quicksort. For 32-bit integers:

≈ 45 cycles/byte for $n \approx 2^{10}$,

≈ 55 cycles/byte for $n \approx 2^{20}$.

Slower than “the radix sort
implemented of IPP, which is
the fastest in-memory sort we are
aware of”: 32, 40 cycles/byte.

IPP: Intel’s Integrated
Performance Primitives library.

Constant
again on

the sorting code

vectorization
(for Haswell)

sorting code
in 2017

ingsatiansup-
/redendaal

software release

revamped for
higher speed

bsort"
sorting code

The slowdown for constant time

Massive fast-sorting literature.

2015 Gueron–Krasnov: AVX and AVX2 (Haswell) optimization of quicksort. For 32-bit integers:

≈ 45 cycles/byte for $n \approx 2^{10}$,

≈ 55 cycles/byte for $n \approx 2^{20}$.

Slower than “the radix sort implemented of IPP, which is the fastest in-memory sort we are aware of”: 32, 40 cycles/byte.

IPP: Intel’s Integrated Performance Primitives library.

Constant-time resu
again on Haswell C

The slowdown for constant time

Massive fast-sorting literature.

2015 Gueron–Krasnov: AVX and AVX2 (Haswell) optimization of quicksort. For 32-bit integers:

≈ 45 cycles/byte for $n \approx 2^{10}$,

≈ 55 cycles/byte for $n \approx 2^{20}$.

Slower than “the radix sort implemented of IPP, which is the fastest in-memory sort we are aware of”: 32, 40 cycles/byte.

IPP: Intel’s Integrated Performance Primitives library.

Constant-time results, again on Haswell CPU core:

The slowdown for constant time

Massive fast-sorting literature.

2015 Gueron–Krasnov: AVX and AVX2 (Haswell) optimization of quicksort. For 32-bit integers:
 ≈ 45 cycles/byte for $n \approx 2^{10}$,
 ≈ 55 cycles/byte for $n \approx 2^{20}$.

Slower than “the radix sort implemented of IPP, which is the fastest in-memory sort we are aware of”: 32, 40 cycles/byte.

IPP: Intel’s Integrated Performance Primitives library.

Constant-time results,
again on Haswell CPU core:

The slowdown for constant time

Massive fast-sorting literature.

2015 Gueron–Krasnov: AVX and AVX2 (Haswell) optimization of quicksort. For 32-bit integers:
 ≈ 45 cycles/byte for $n \approx 2^{10}$,
 ≈ 55 cycles/byte for $n \approx 2^{20}$.

Slower than “the radix sort implemented of IPP, which is the fastest in-memory sort we are aware of”: 32, 40 cycles/byte.

IPP: Intel’s Integrated Performance Primitives library.

Constant-time results, again on Haswell CPU core:

2017 BCLvV:

6.5 cycles/byte for $n \approx 2^{10}$,
 33 cycles/byte for $n \approx 2^{20}$.

The slowdown for constant time

Massive fast-sorting literature.

2015 Gueron–Krasnov: AVX and AVX2 (Haswell) optimization of quicksort. For 32-bit integers:

≈ 45 cycles/byte for $n \approx 2^{10}$,

≈ 55 cycles/byte for $n \approx 2^{20}$.

Slower than “the radix sort implemented of IPP, which is the fastest in-memory sort we are aware of”: 32, 40 cycles/byte.

IPP: Intel’s Integrated Performance Primitives library.

Constant-time results, again on Haswell CPU core:

2017 BCLvV:

6.5 cycles/byte for $n \approx 2^{10}$,

33 cycles/byte for $n \approx 2^{20}$.

2018 djbsort:

2.5 cycles/byte for $n \approx 2^{10}$,

15.5 cycles/byte for $n \approx 2^{20}$.

The slowdown for constant time

Massive fast-sorting literature.

2015 Gueron–Krasnov: AVX and AVX2 (Haswell) optimization of quicksort. For 32-bit integers:

≈ 45 cycles/byte for $n \approx 2^{10}$,

≈ 55 cycles/byte for $n \approx 2^{20}$.

Slower than “the radix sort implemented of IPP, which is the fastest in-memory sort we are aware of”: 32, 40 cycles/byte.

IPP: Intel’s Integrated Performance Primitives library.

Constant-time results, again on Haswell CPU core:

2017 BCLvV:

6.5 cycles/byte for $n \approx 2^{10}$,

33 cycles/byte for $n \approx 2^{20}$.

2018 djbsort:

2.5 cycles/byte for $n \approx 2^{10}$,

15.5 cycles/byte for $n \approx 2^{20}$.

No slowdown. New speed records!

The slowdown for constant time

Massive fast-sorting literature.

2015 Gueron–Krasnov: AVX and AVX2 (Haswell) optimization of quicksort. For 32-bit integers:

≈ 45 cycles/byte for $n \approx 2^{10}$,

≈ 55 cycles/byte for $n \approx 2^{20}$.

Slower than “the radix sort implemented of IPP, which is the fastest in-memory sort we are aware of”: 32, 40 cycles/byte.

IPP: Intel’s Integrated Performance Primitives library.

Constant-time results, again on Haswell CPU core:

2017 BCLvV:

6.5 cycles/byte for $n \approx 2^{10}$,

33 cycles/byte for $n \approx 2^{20}$.

2018 djbsort:

2.5 cycles/byte for $n \approx 2^{10}$,

15.5 cycles/byte for $n \approx 2^{20}$.

No slowdown. New speed records!

Warning: Comparison for $n \approx 2^{20}$ involves microarchitecture details beyond Haswell core. Should measure all code on same CPU.

slowdown for constant time

fast-sorting literature.

Antonov–Krasnov: AVX and

(Haswell) optimization of

code. For 32-bit integers:

6.5 cycles/byte for $n \approx 2^{10}$,

33 cycles/byte for $n \approx 2^{20}$.

Antonov “the radix sort

variant of IPP, which is

the best in-memory sort we are

aware of”: 32, 40 cycles/byte.

Intel’s Integrated

Performance Primitives library.

Constant-time results,
again on Haswell CPU core:

2017 BCLvV:

6.5 cycles/byte for $n \approx 2^{10}$,

33 cycles/byte for $n \approx 2^{20}$.

2018 djbsort:

2.5 cycles/byte for $n \approx 2^{10}$,

15.5 cycles/byte for $n \approx 2^{20}$.

No slowdown. New speed records!

Warning: Comparison for $n \approx 2^{20}$

involves microarchitecture details

beyond Haswell core. Should

measure all code on same CPU.

How can

beat sta

constant time

g literature.

snov: AVX and

optimization of

bit integers:

or $n \approx 2^{10}$,

or $n \approx 2^{20}$.

radix sort

P, which is

emory sort we are

cycles/byte.

ated

itives library.

Constant-time results,
again on Haswell CPU core:

2017 BCLvV:

6.5 cycles/byte for $n \approx 2^{10}$,

33 cycles/byte for $n \approx 2^{20}$.

2018 djbsort:

2.5 cycles/byte for $n \approx 2^{10}$,

15.5 cycles/byte for $n \approx 2^{20}$.

No slowdown. New speed records!

Warning: Comparison for $n \approx 2^{20}$

involves microarchitecture details

beyond Haswell core. Should

measure all code on same CPU.

How can an $n(\log$
beat standard $n \log$

Constant-time results,
again on Haswell CPU core:

2017 BCLvV:

6.5 cycles/byte for $n \approx 2^{10}$,

33 cycles/byte for $n \approx 2^{20}$.

2018 djbsort:

2.5 cycles/byte for $n \approx 2^{10}$,

15.5 cycles/byte for $n \approx 2^{20}$.

No slowdown. New speed records!

Warning: Comparison for $n \approx 2^{20}$
involves microarchitecture details
beyond Haswell core. Should
measure all code on same CPU.

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

Constant-time results,
again on Haswell CPU core:

2017 BCLvV:

6.5 cycles/byte for $n \approx 2^{10}$,

33 cycles/byte for $n \approx 2^{20}$.

2018 djbsort:

2.5 cycles/byte for $n \approx 2^{10}$,

15.5 cycles/byte for $n \approx 2^{20}$.

No slowdown. New speed records!

Warning: Comparison for $n \approx 2^{20}$
involves microarchitecture details
beyond Haswell core. Should
measure all code on same CPU.

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

Constant-time results,
again on Haswell CPU core:

2017 BCLvV:

6.5 cycles/byte for $n \approx 2^{10}$,

33 cycles/byte for $n \approx 2^{20}$.

2018 djbsort:

2.5 cycles/byte for $n \approx 2^{10}$,

15.5 cycles/byte for $n \approx 2^{20}$.

No slowdown. New speed records!

Warning: Comparison for $n \approx 2^{20}$
involves microarchitecture details
beyond Haswell core. Should
measure all code on same CPU.

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.

Constant-time results,
again on Haswell CPU core:

2017 BCLvV:

6.5 cycles/byte for $n \approx 2^{10}$,
33 cycles/byte for $n \approx 2^{20}$.

2018 djbsort:

2.5 cycles/byte for $n \approx 2^{10}$,
15.5 cycles/byte for $n \approx 2^{20}$.

No slowdown. New speed records!

Warning: Comparison for $n \approx 2^{20}$
involves microarchitecture details
beyond Haswell core. Should
measure all code on same CPU.

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.

Constant-time results,
again on Haswell CPU core:

2017 BCLvV:

6.5 cycles/byte for $n \approx 2^{10}$,
33 cycles/byte for $n \approx 2^{20}$.

2018 djbsort:

2.5 cycles/byte for $n \approx 2^{10}$,
15.5 cycles/byte for $n \approx 2^{20}$.

No slowdown. New speed records!

Warning: Comparison for $n \approx 2^{20}$
involves microarchitecture details
beyond Haswell core. Should
measure all code on same CPU.

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.

Real-time results,

on Haswell CPU core:

CLvV:

ops/byte for $n \approx 2^{10}$,

ops/byte for $n \approx 2^{20}$.

Sort:

ops/byte for $n \approx 2^{10}$,

ops/byte for $n \approx 2^{20}$.

down. New speed records!

: Comparison for $n \approx 2^{20}$

microarchitecture details

Haswell core. Should

all code on same CPU.

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 v

Test the

random

decreasin

ults,
 CPU core:
 $n \approx 2^{10}$,
 $n \approx 2^{20}$.
 $n \approx 2^{10}$,
 or $n \approx 2^{20}$.
 w speed records!
 ison for $n \approx 2^{20}$
 itecture details
 ore. Should
 on same CPU.

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.

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.

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

well-known trends

design, reflecting

mental hardware costs

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?

own trends

reflecting

ware costs

ons.

ell core can do

2-bit integers +

2-bit integers.

integer from a

much slower.

n: much slower.

Verification

Sorting software is in the TCB.

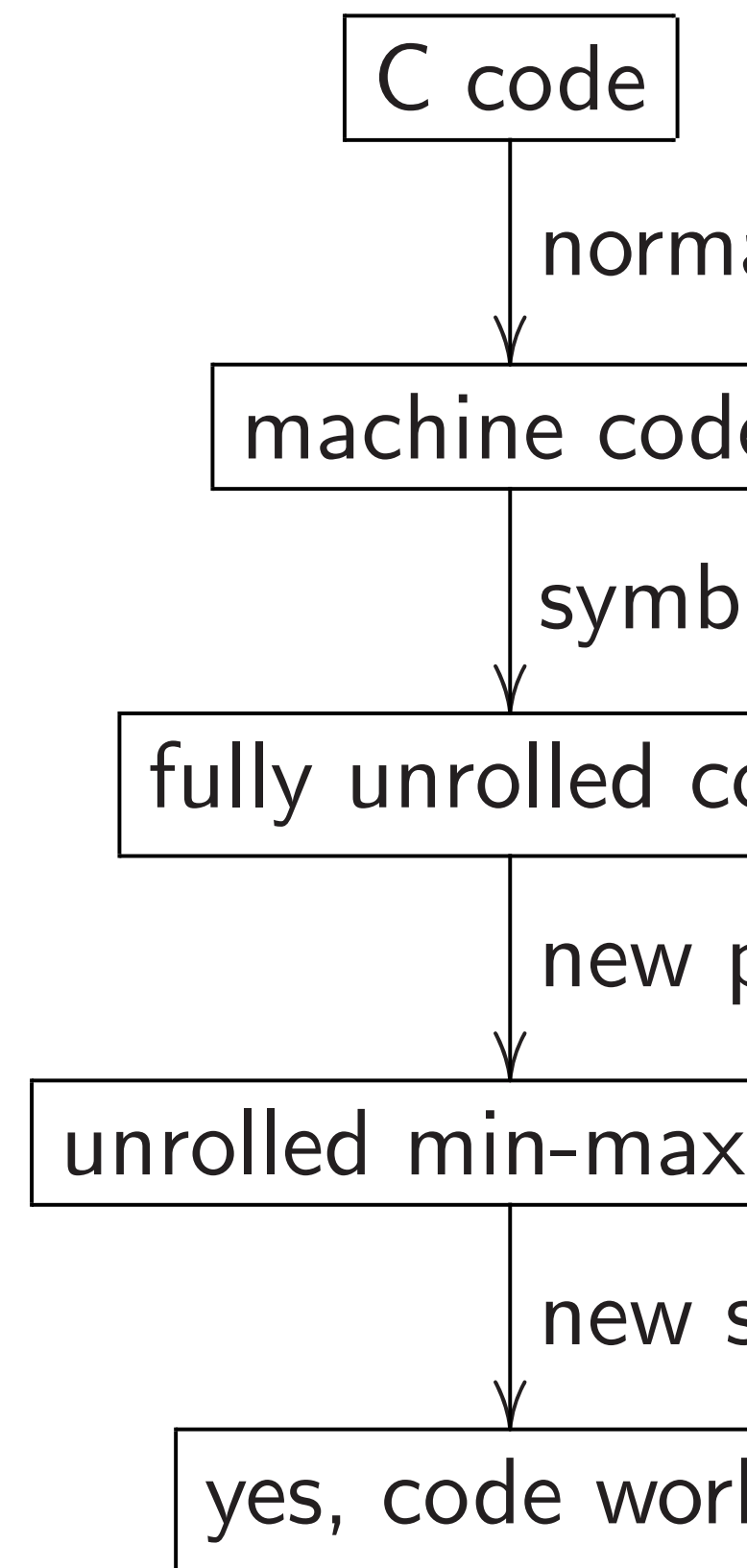
Does it work correctly?

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



Verification

Sorting software is in the TCB.

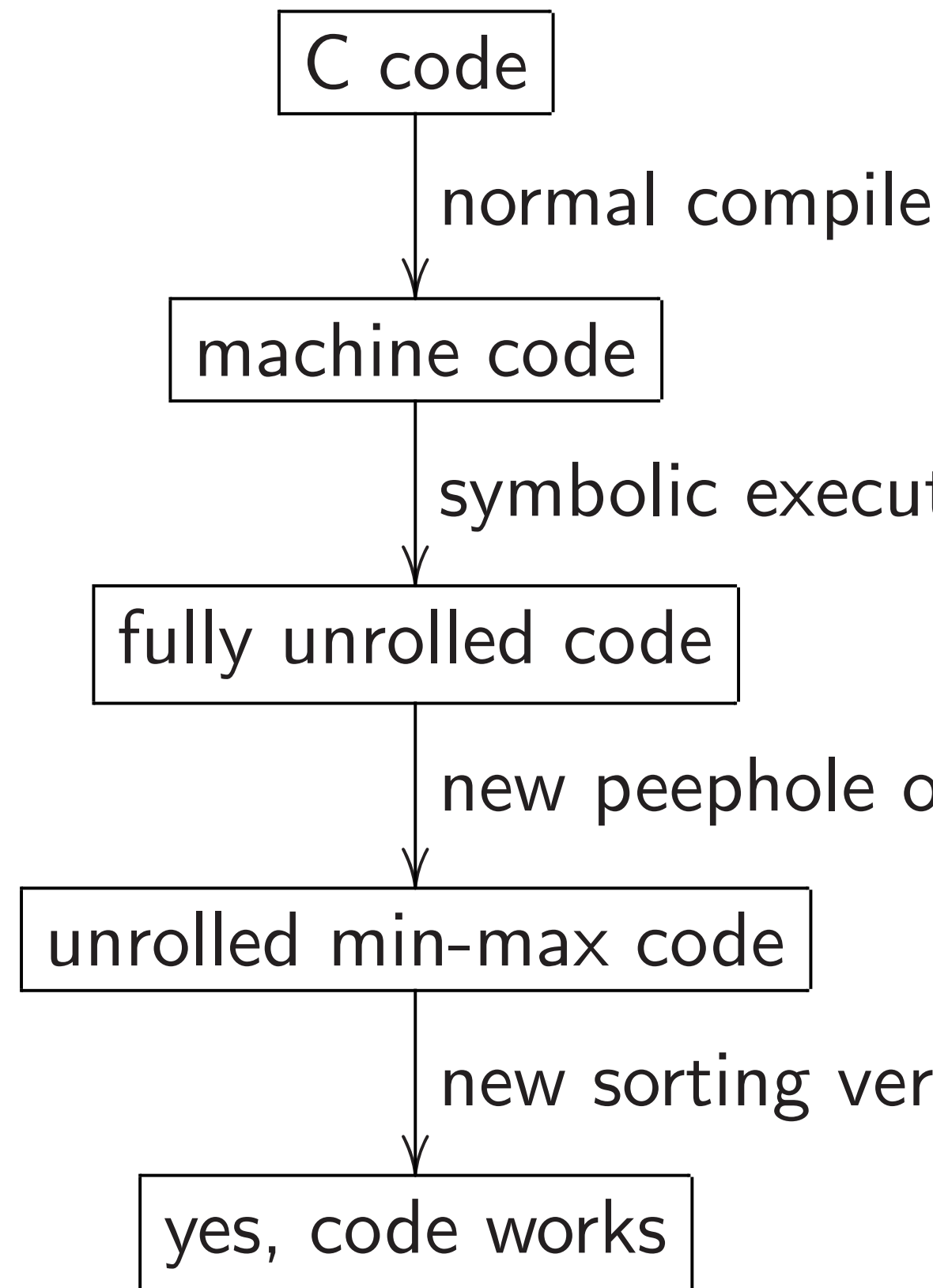
Does it work correctly?

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):



Verification

Sorting software is in the TCB.

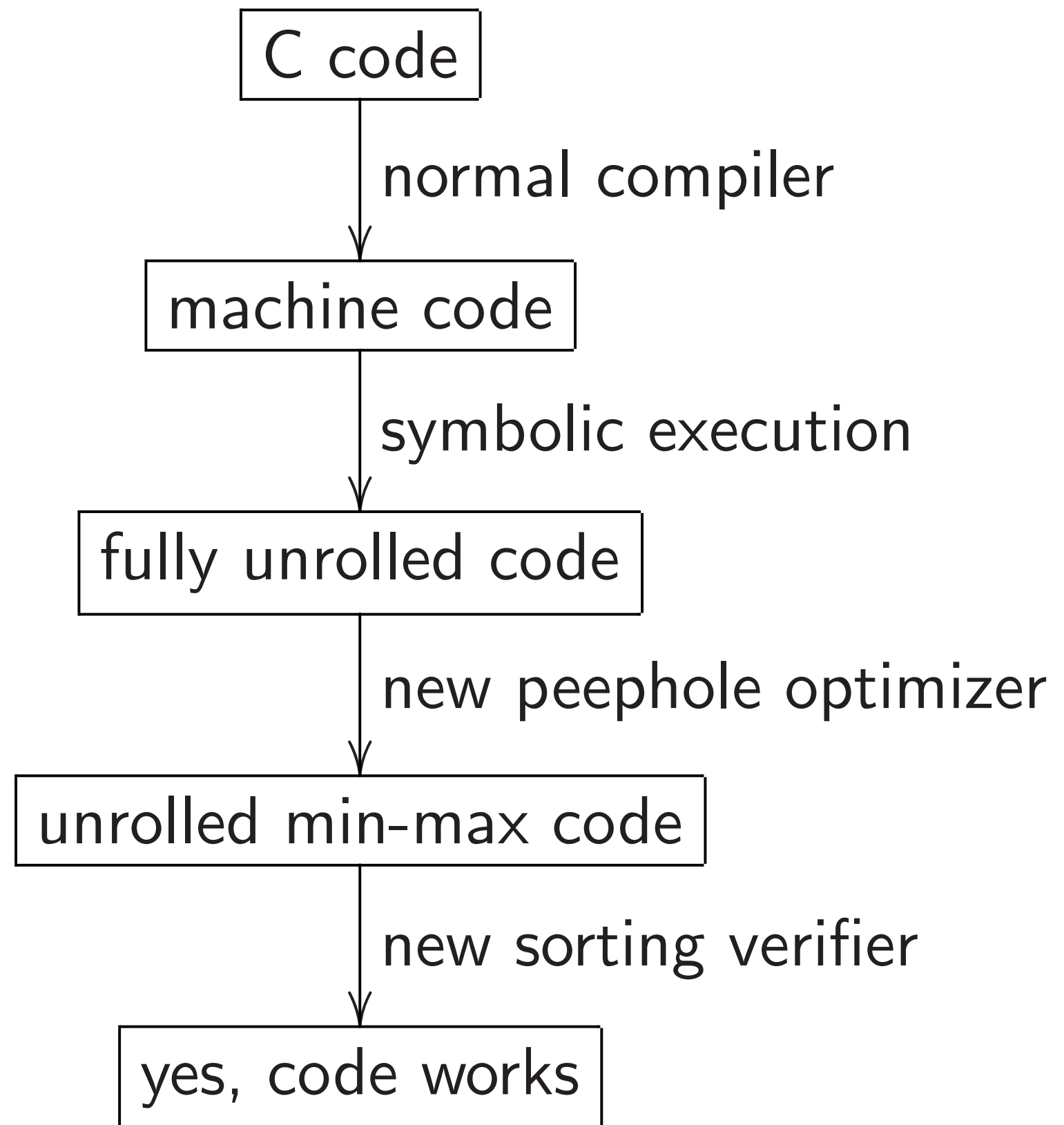
Does it work correctly?

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):



ion

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

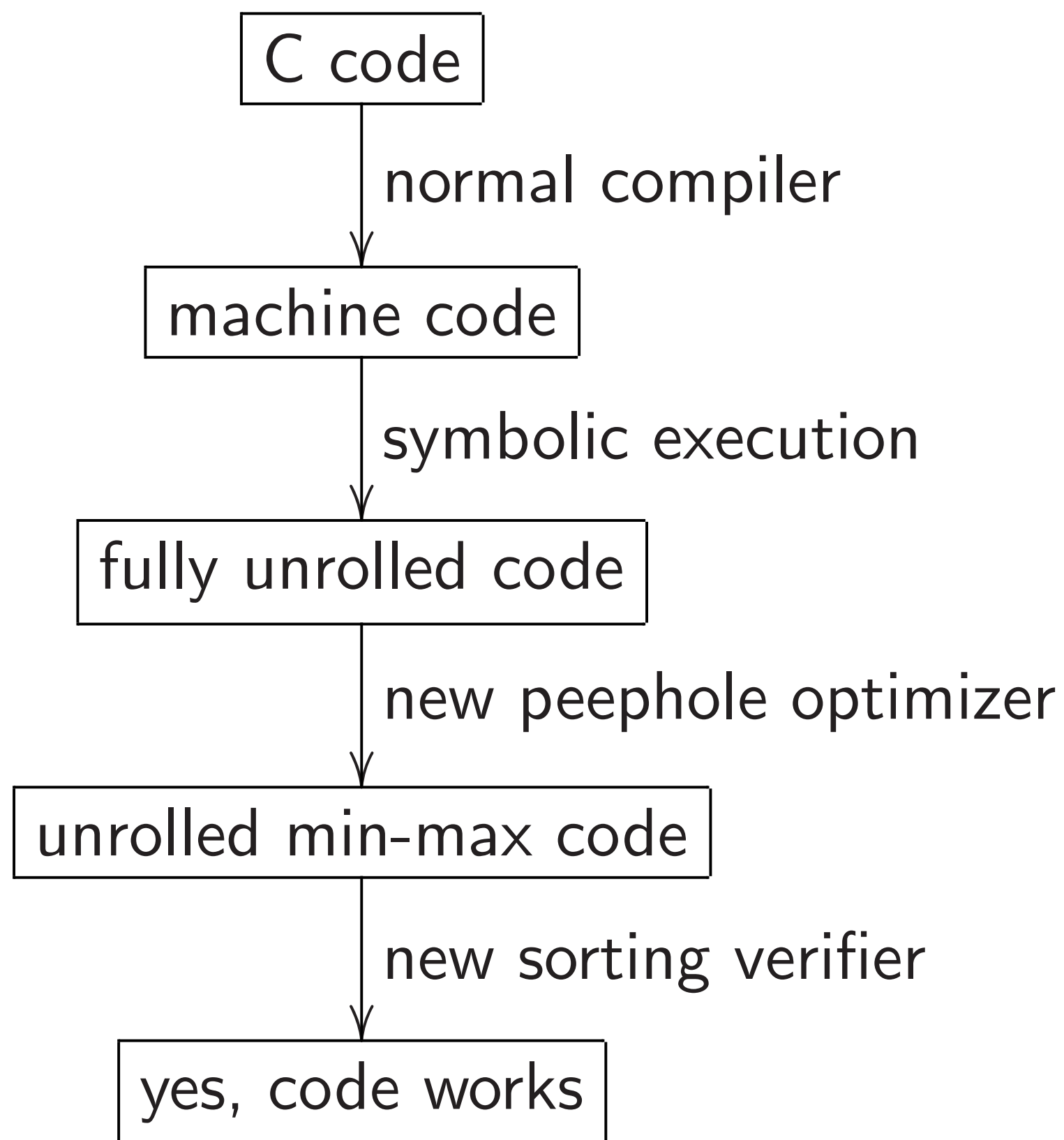
sort correctly?

Many security problems

occasional inputs

CB works incorrectly.

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



Symbolic

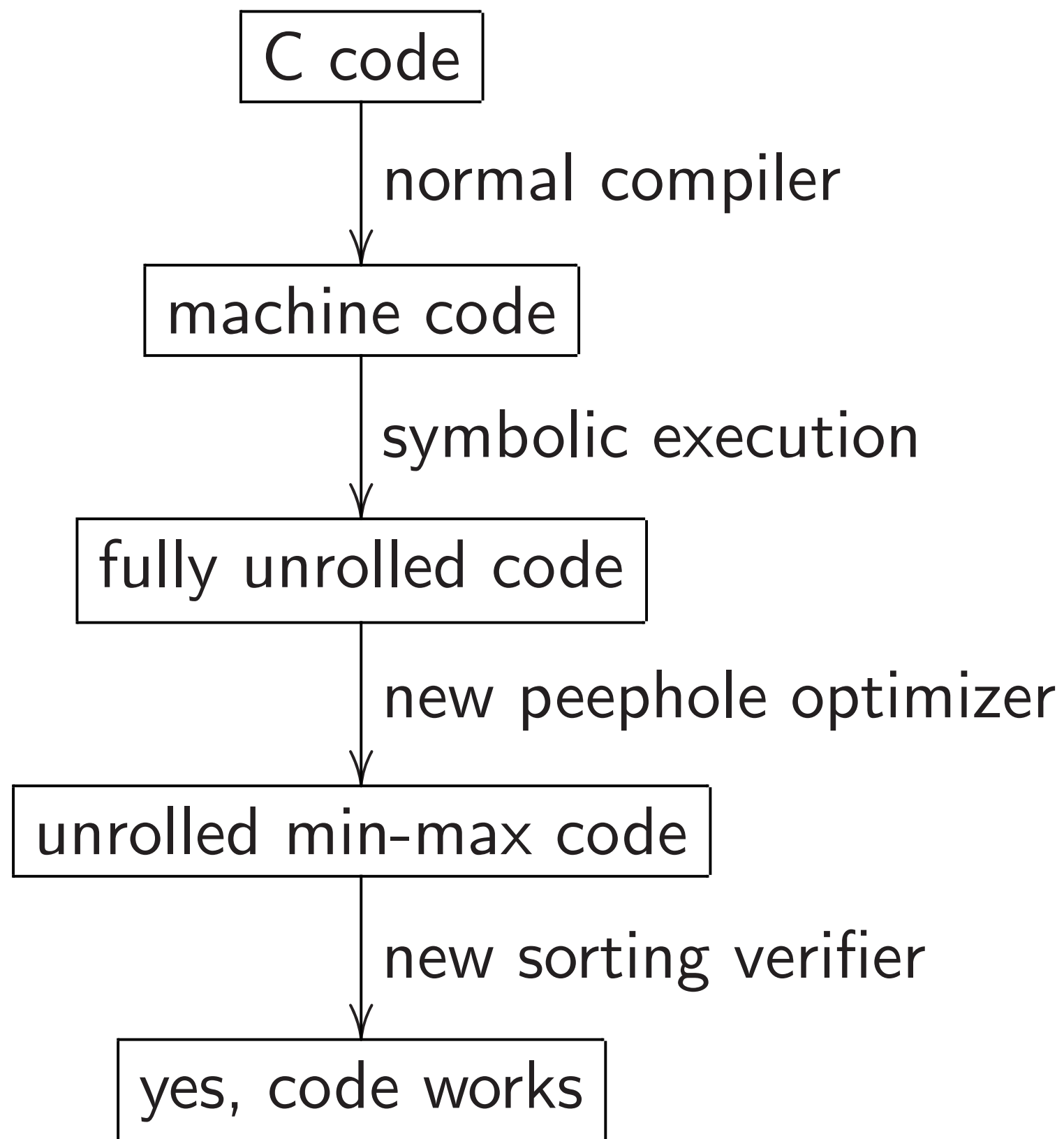
use exist

with tiny

eliminati

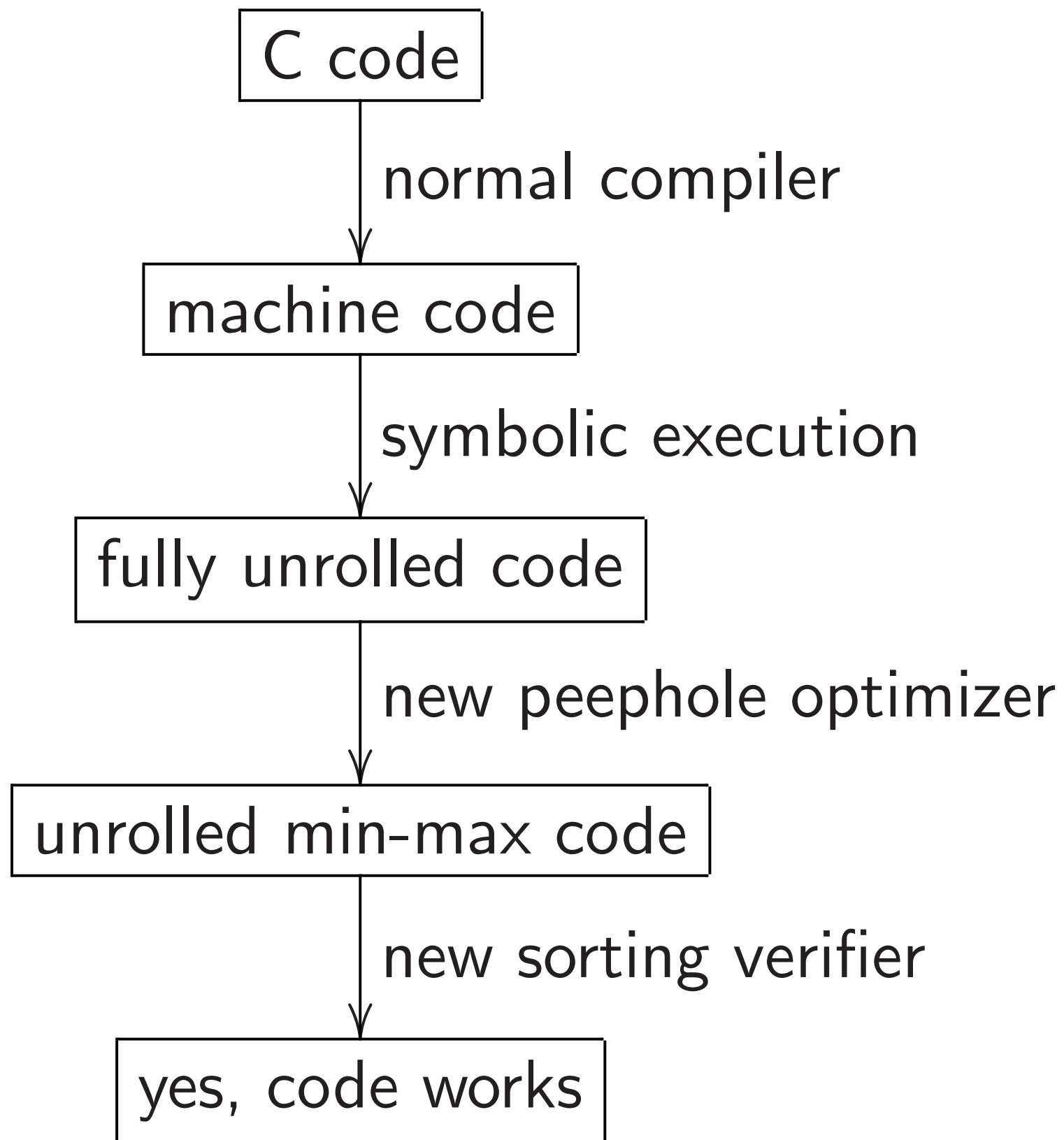
a few m

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



Symbolic execution
 use existing “angr”
 with tiny new patch
 eliminating byte sp
 a few missing vect

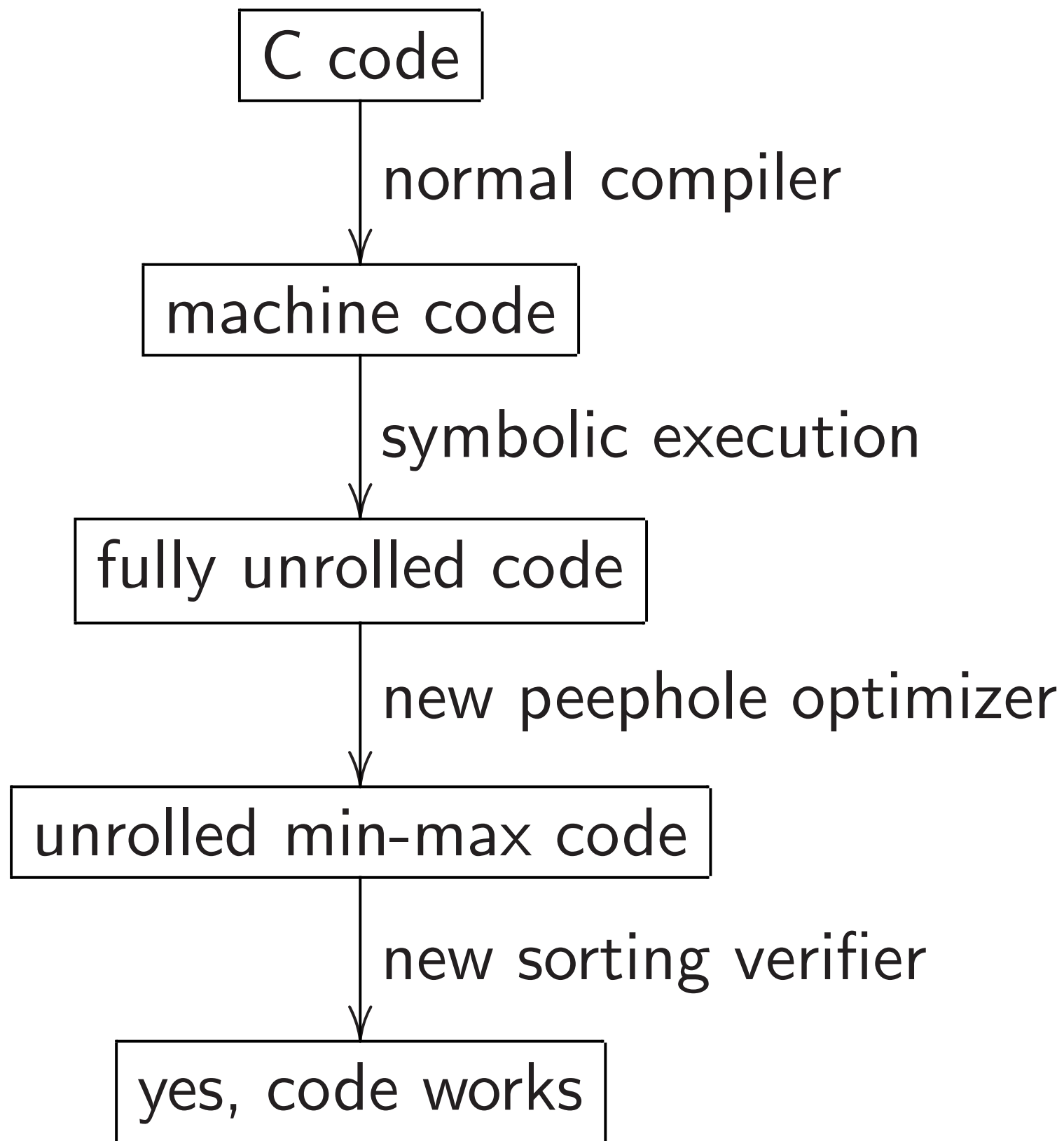
For each used n (e.g., 768):



Symbolic execution:

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

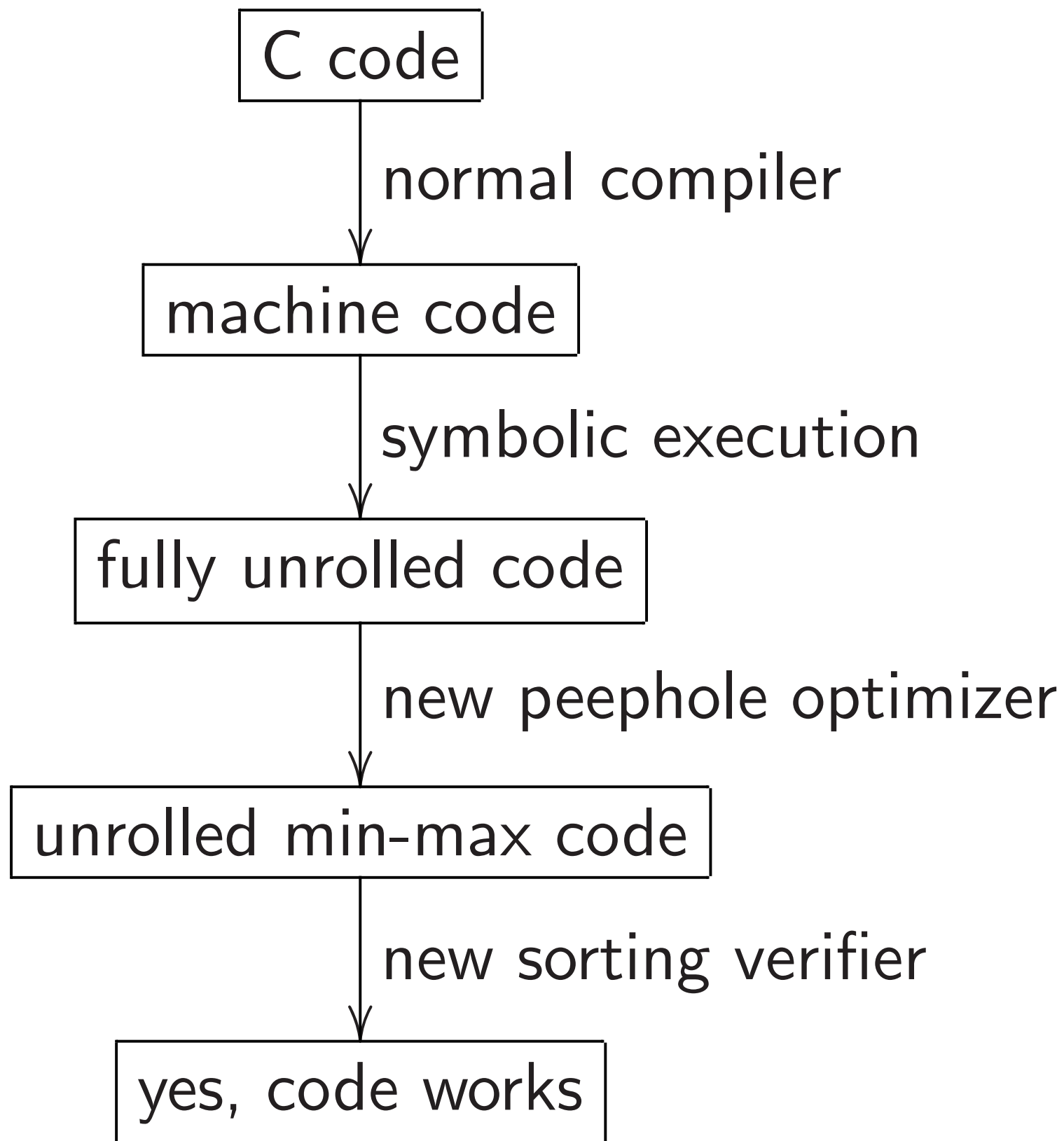
For each used n (e.g., 768):



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):



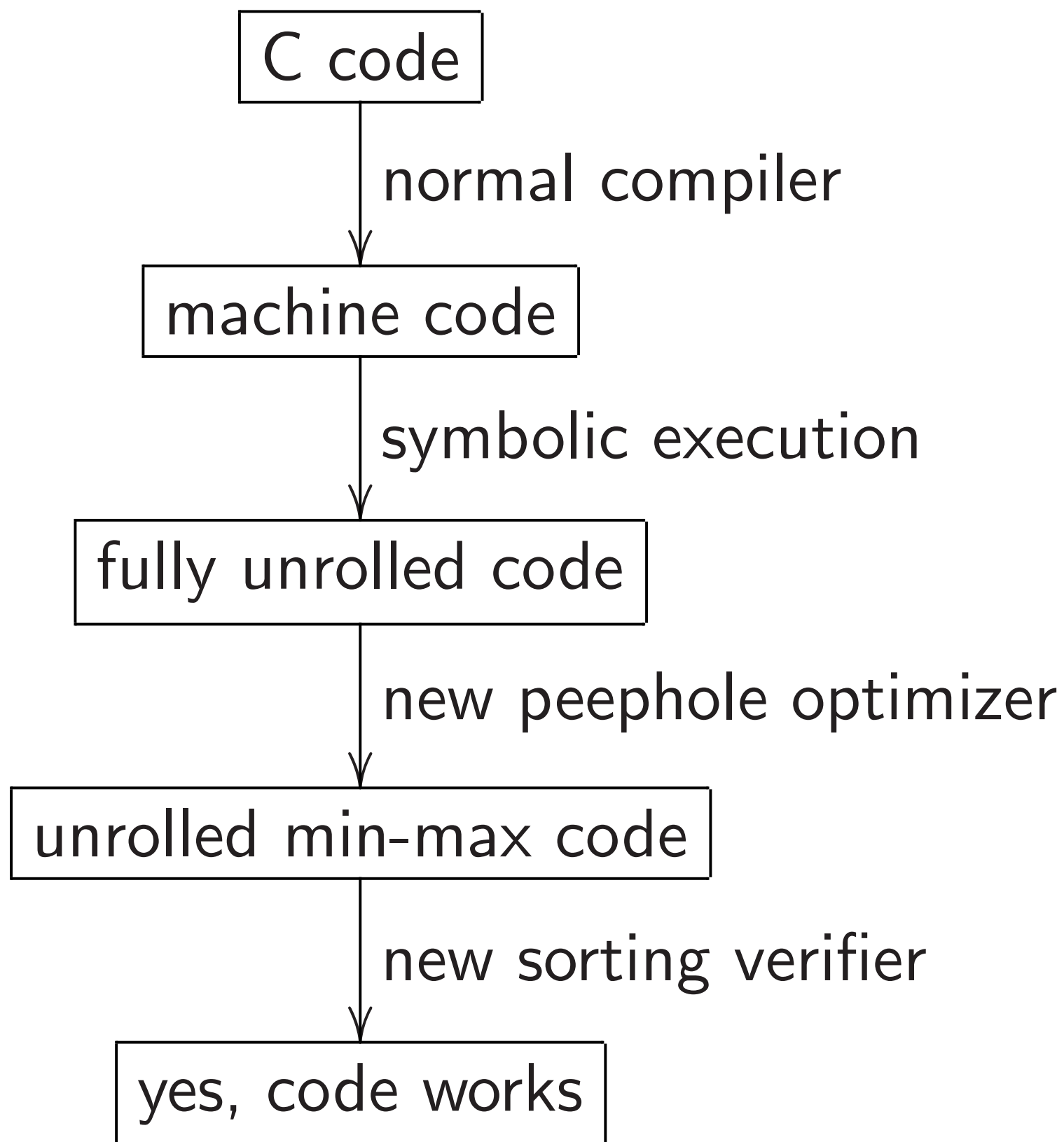
Symbolic execution:

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):



Symbolic execution:

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

Peephole optimizer:

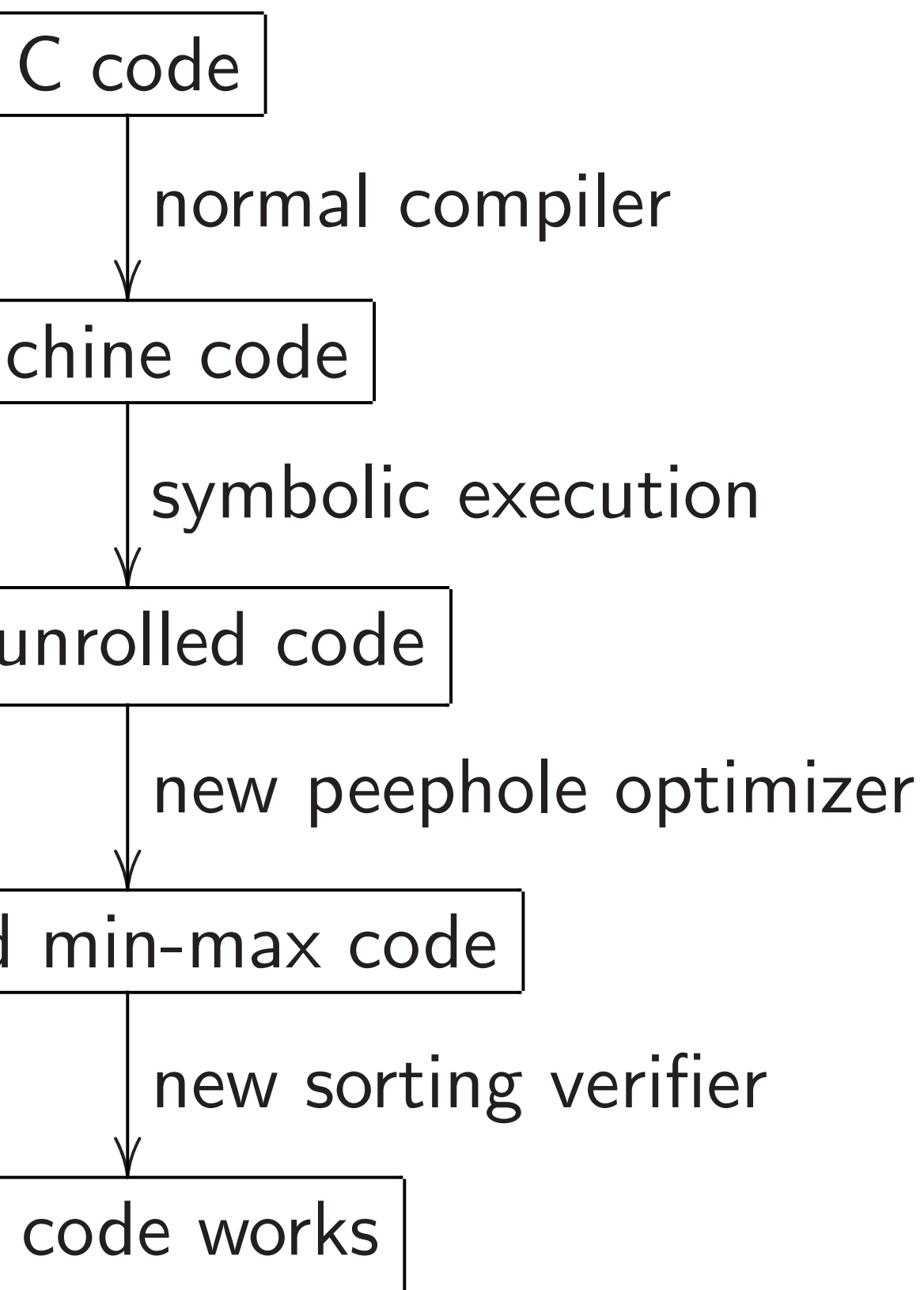
recognize instruction patterns equivalent to min, max.

Sorting verifier: decompose DAG into merging networks.

Verify each merging network using generalization of 2007

Even–Levi–Litman, correction of 1990 Chung–Ravikumar.

used n (e.g., 768):



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.

First djbb
 verified :

<https://>

Includes
 automat
 simple b
 verificati

Web site
 use the v

Next relat
 verified
 and veri

e.g., 768):

al compiler

e

olic execution

ode

peephole optimizer

code

orting 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.

First djbsort release
verified int32 on

<https://sorting>

Includes the sorting
automatic build-time
simple benchmark
verification tools.

Web site shows how
use the verification

Next release planned
verified ARM NEON
and verified portable

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.

First djbsort release,
verified `int32` on AVX2:

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

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
and verified portable 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.

First djbsort release,
verified int32 on AVX2:
<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
and verified portable code.