Sorting integer arrays: security, speed, and verification

D. J. Bernstein

University of Illinois at Chicago,
Ruhr-University Bochum

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

ty of Illinois at Chicago,
iversity Bochum

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

TCB: po
that is r
the user

ays:
d verification

is at Chicago,
ochum

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

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

ion

ago,

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 (TC

TCB: portion of computer s
that is responsible for enforc
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.

Bob's 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.

Bob's 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.

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.

Trusted computing base (TCB)

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

Bob's 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

en:

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

message is
ctually sent.

rity" systems
his property.
ve modified
sage.

Trusted computing base (TCB)

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

Bob's 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

Trusted computing base (TCB)

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

Bob's 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.

Bob's 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.

Bob's 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.

Bob's 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.

Bob's 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.

curity 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.

icy for 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

s talk:

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	VM C	
Alice data	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	VM C	
Alice data	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.
e 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.

Cryptogr

How doe
that inc
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:

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.

Cryptography

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

Cryptographic solu
Message-authentic

Alice's messag
↓
authenticated me
↓ untru
authenticated me
↓
Alice's messag

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.

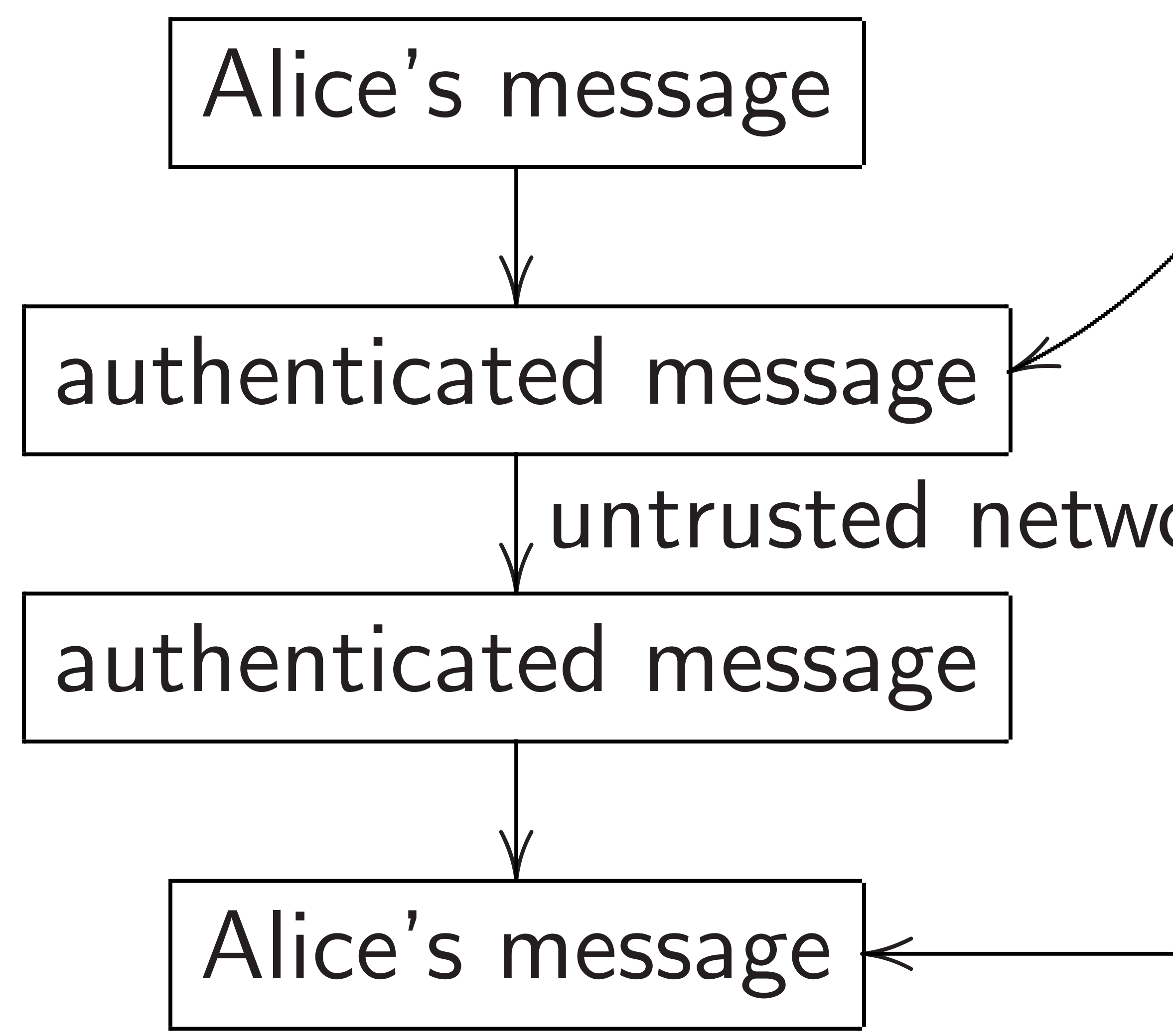
Browser in VM C isn't in TCB.
Can't touch data in VM A,
if TCB works correctly.

Alice also runs many VMs.

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:

VM A	VM C	
Alice data	Charlie data	...

TCB stops each VM from touching data in other VMs.

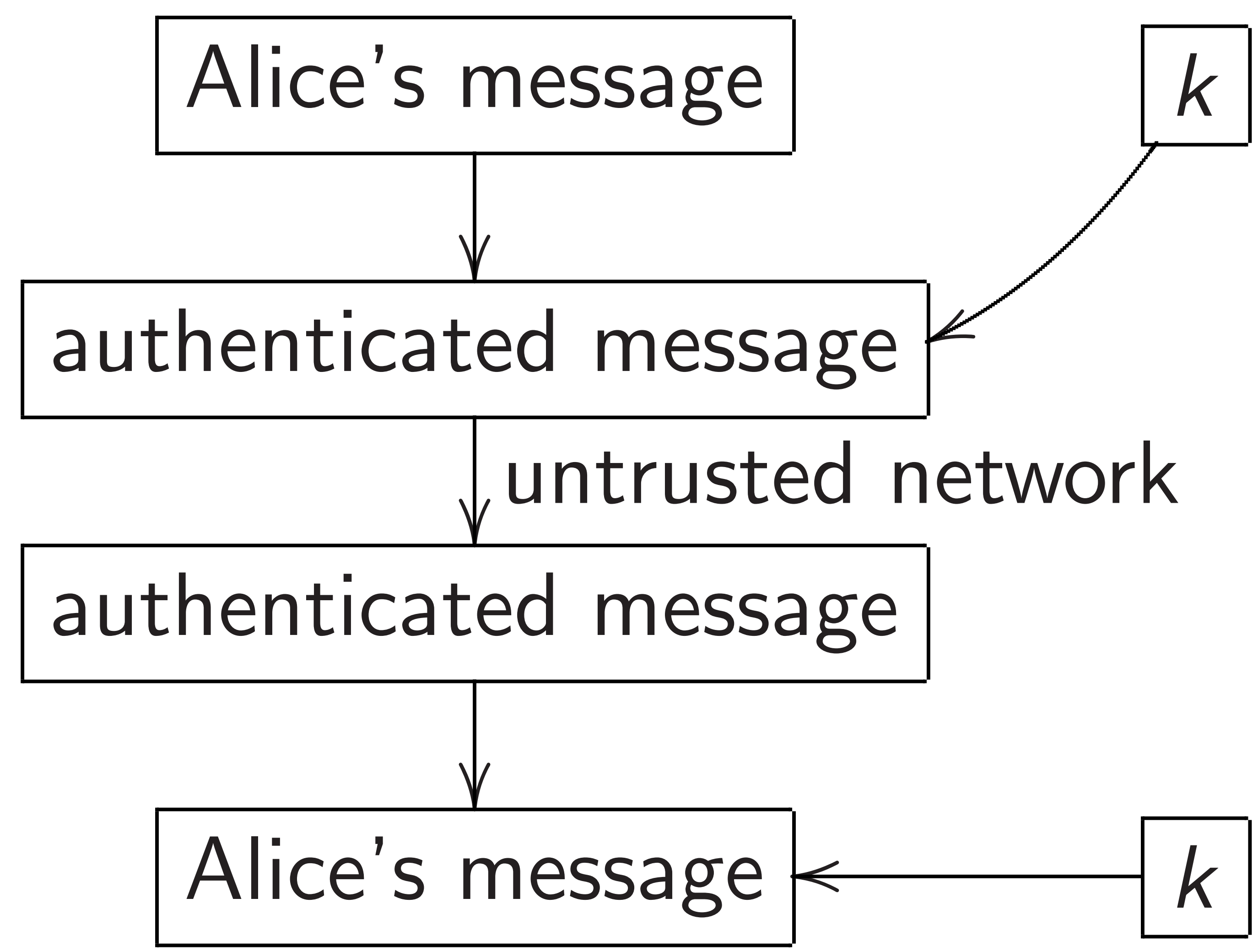
Browser in VM C isn't in TCB. Can't touch data in VM A, if TCB works correctly.

Alice also runs many VMs.

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:

VM A	VM C	
Alice data	Charlie data	. . .

TCB stops each VM from touching data in other VMs.

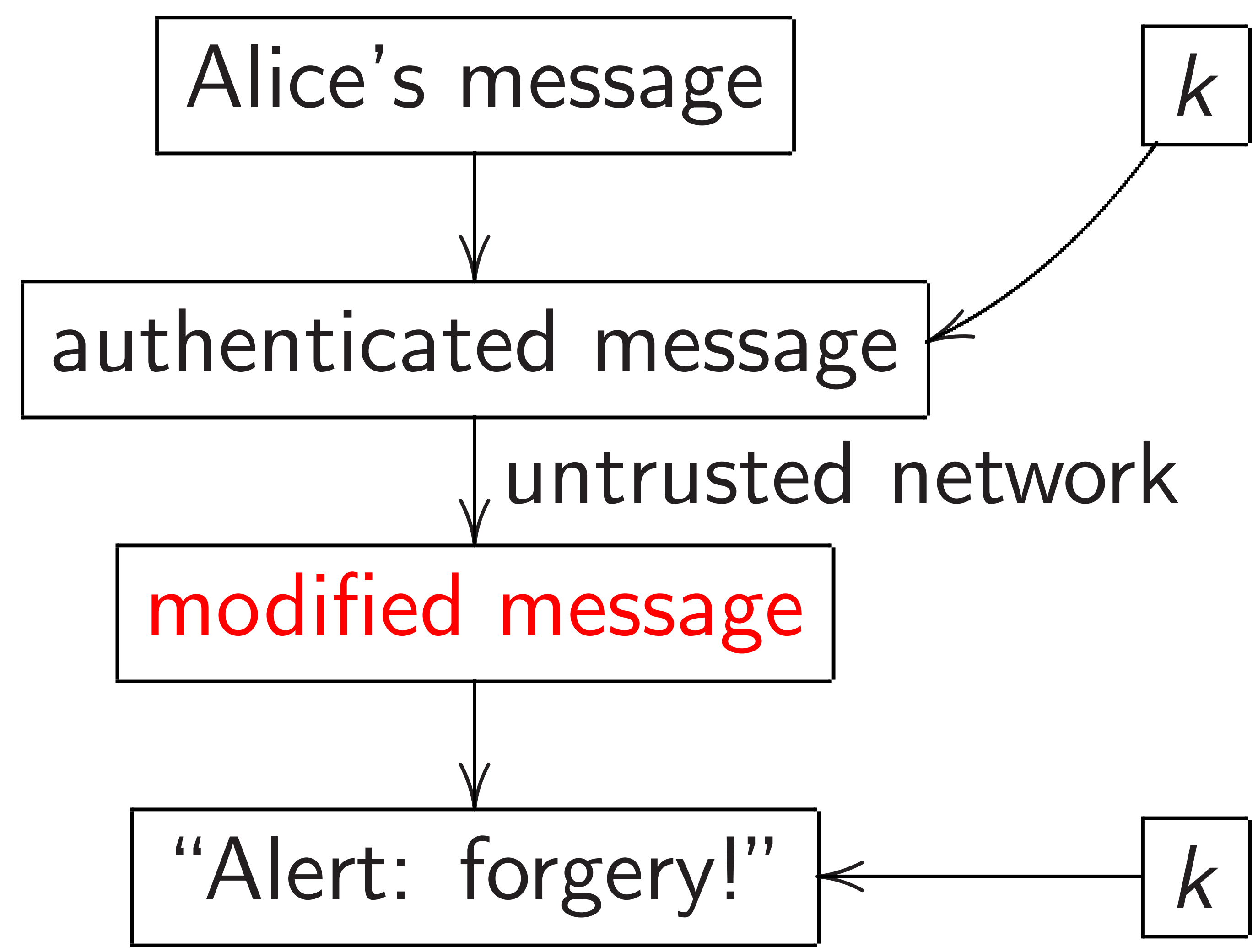
Browser in VM C isn't in TCB. Can't touch data in VM A, if TCB works correctly.

Alice also runs many VMs.

Cryptography

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

Cryptographic solution: Message-authentication codes.



ecurity strategy:

ect computer systems
a much smaller TCB.

audit the TCB.

runs many VMs:

A | VM C
ata | Charlie data | . . .

ps each VM from
data in other VMs.

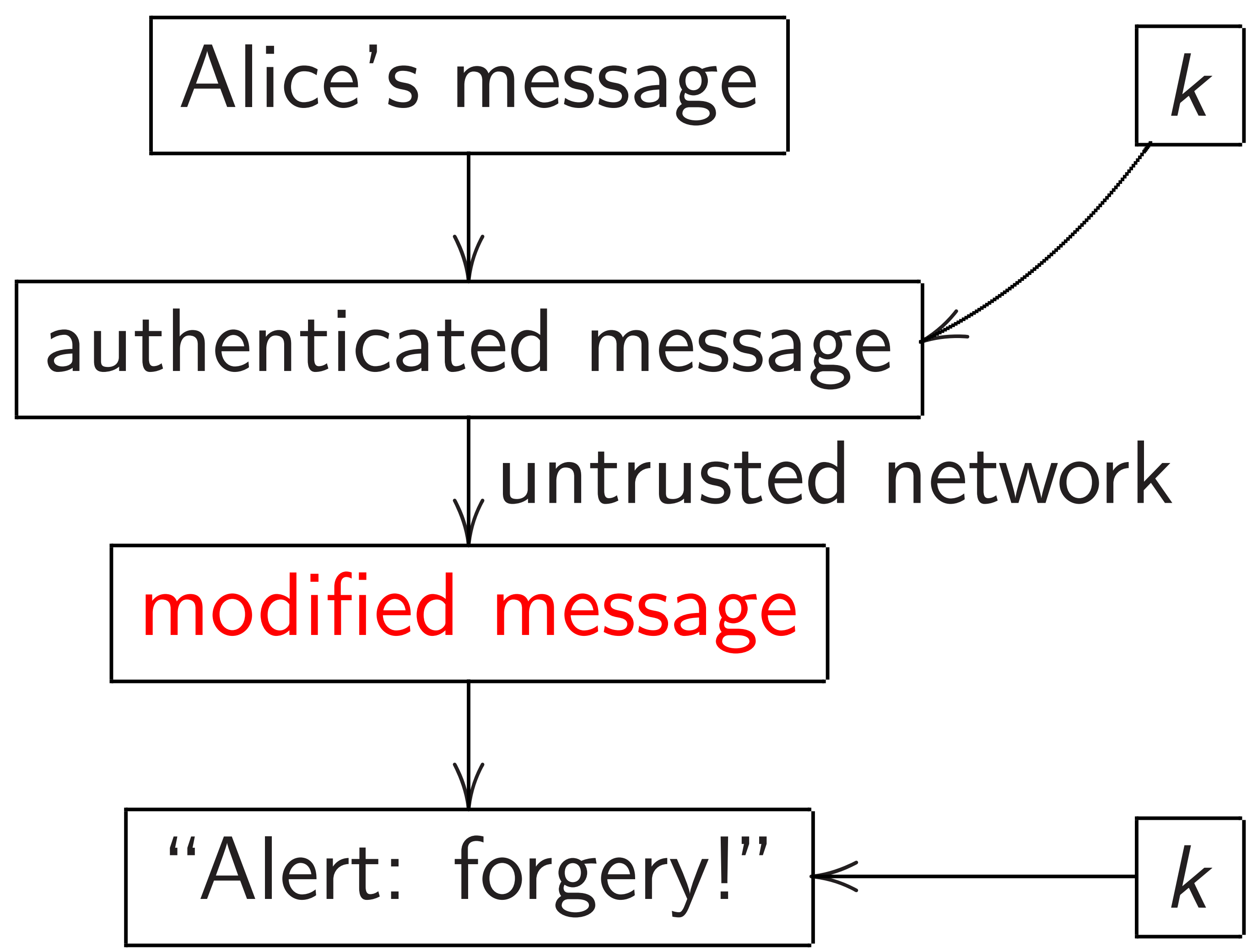
in VM C isn't in TCB.
uch data in VM A,
works correctly.

o runs many VMs.

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.

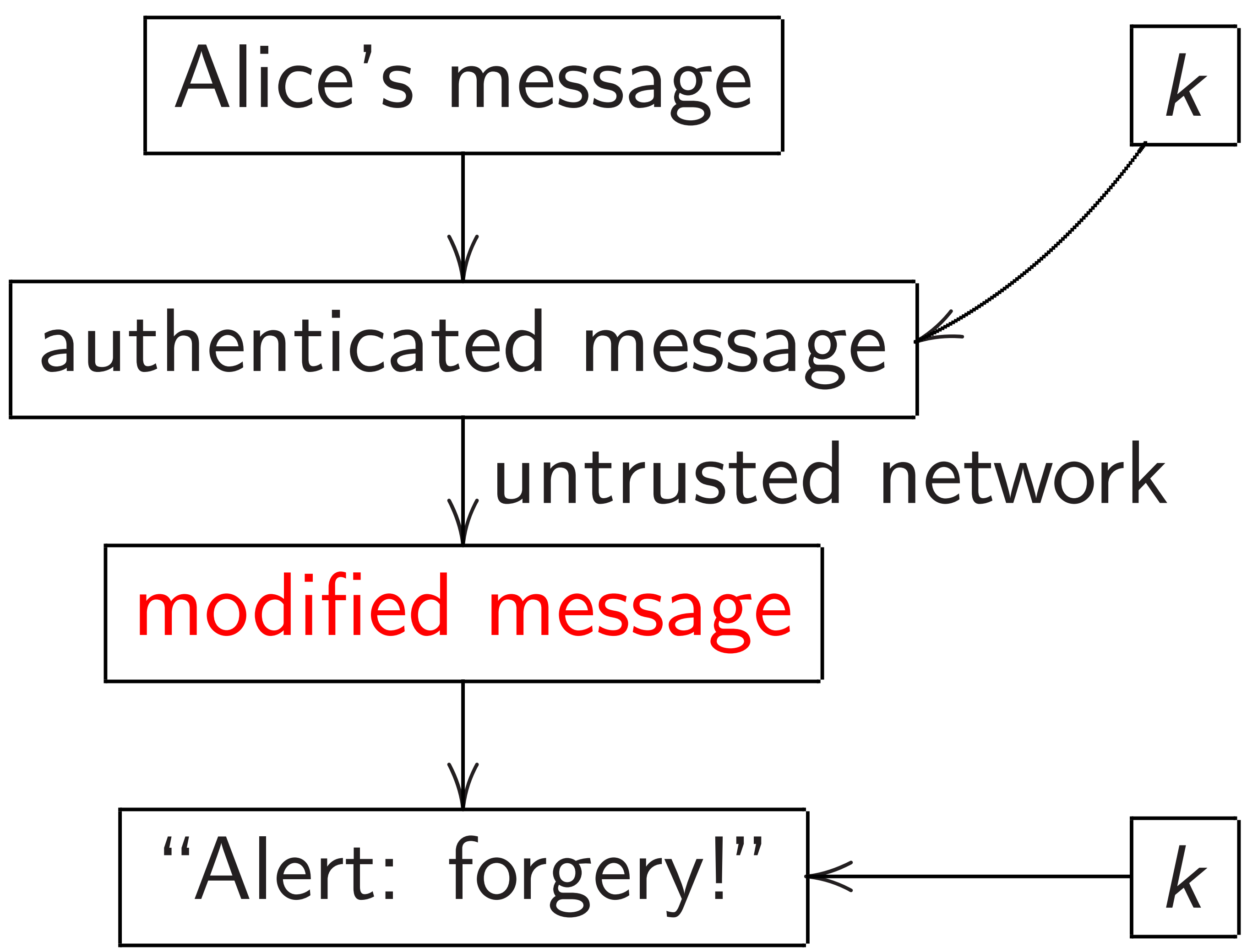
isn't in TCB.
in VM A,
ectly.

ny VMs.

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.

. . .

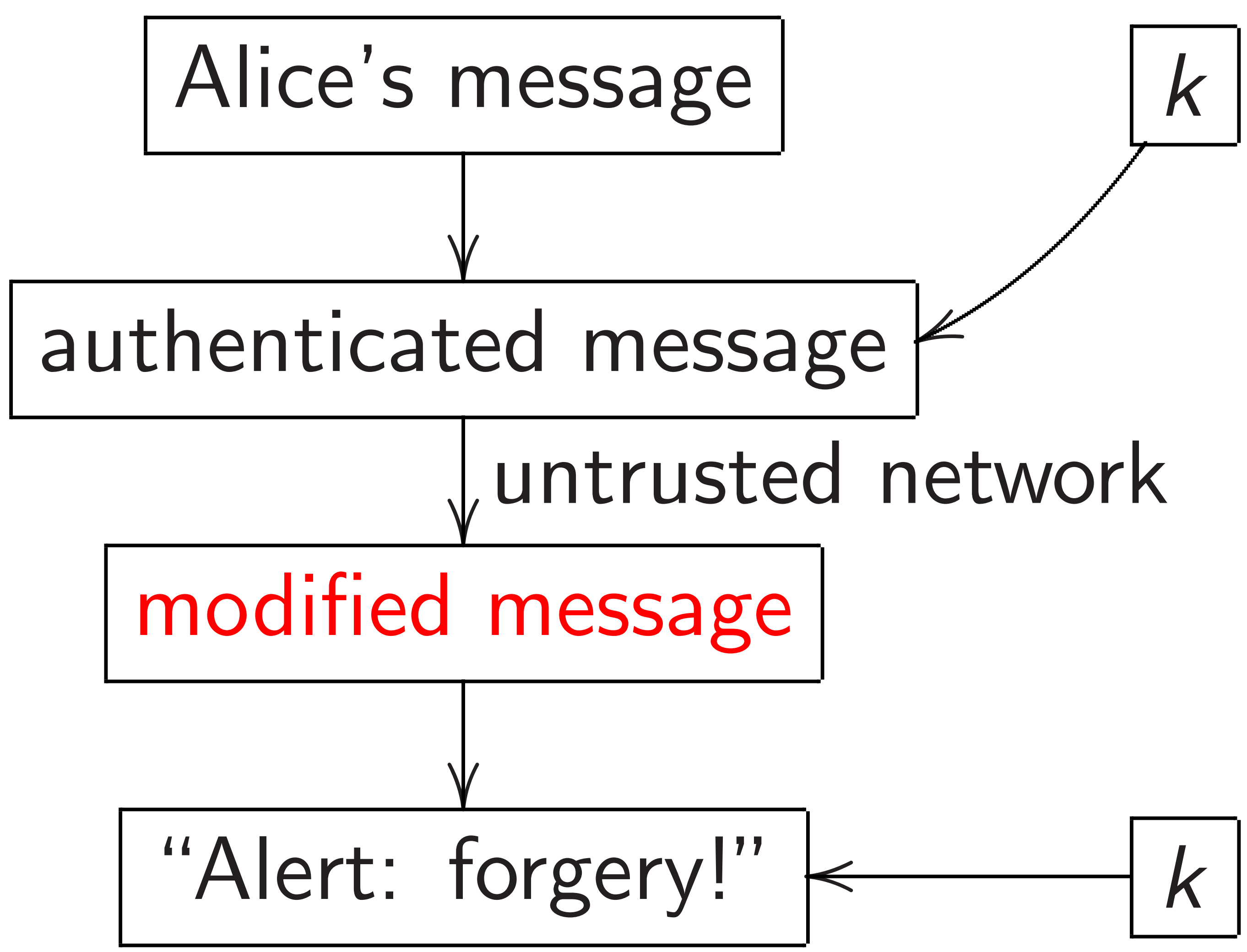
.

CB.

Cryptography

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

Cryptographic solution: Message-authentication codes.



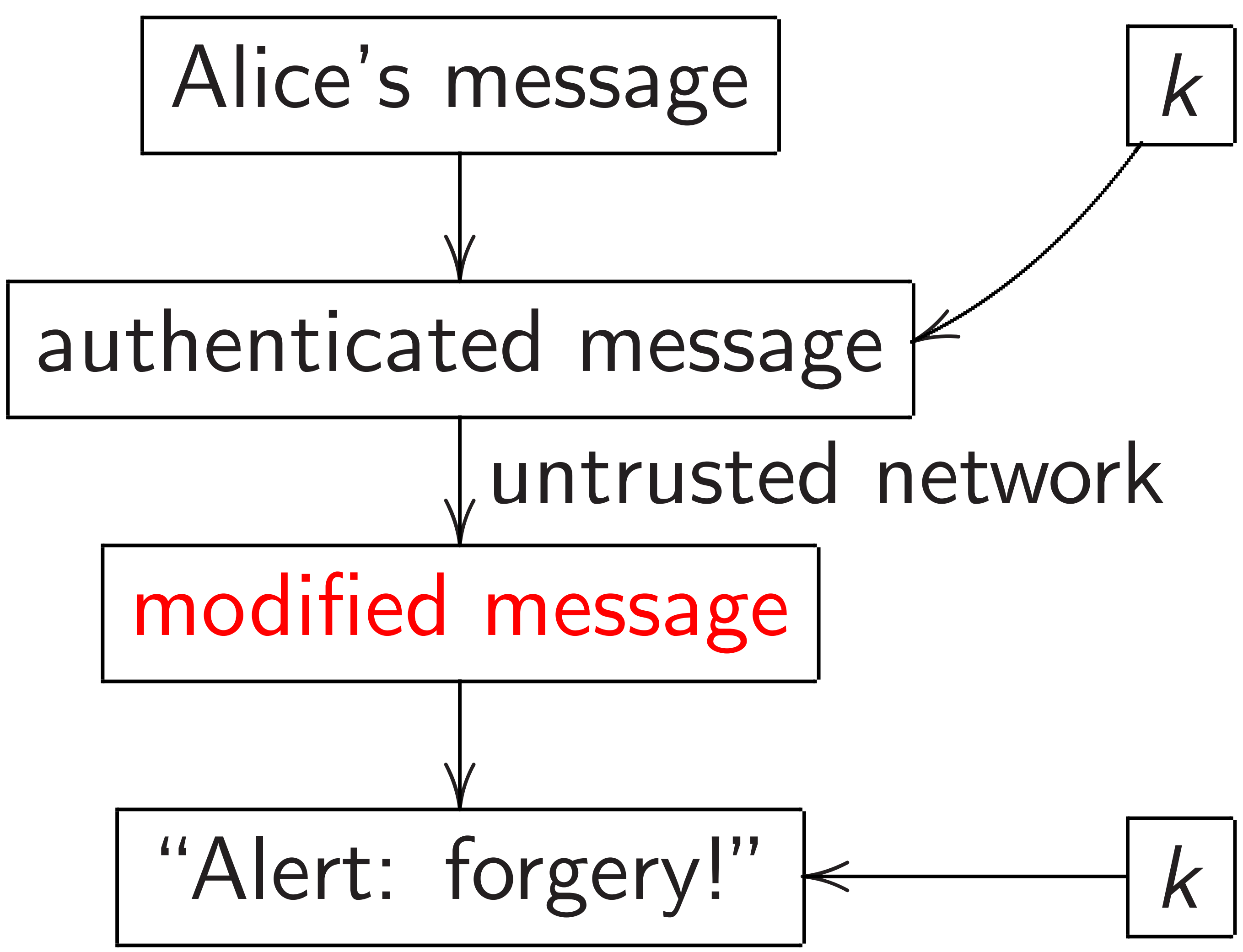
Important for Alice and Bob to share the same secret k.

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.



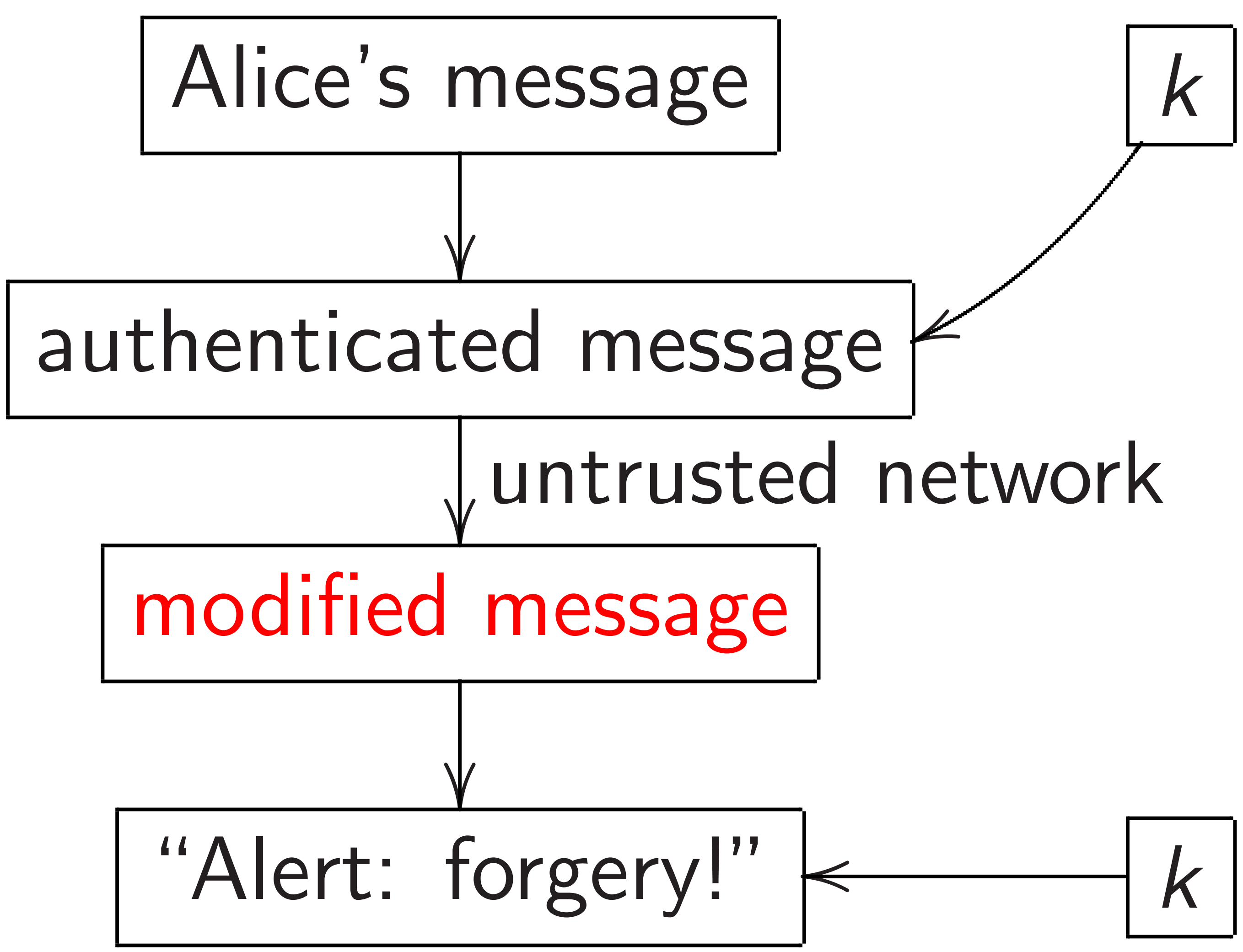
Important for Alice and Bob to share the same secret k.

What if attacker was spying on their communication of k?

Cryptography

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

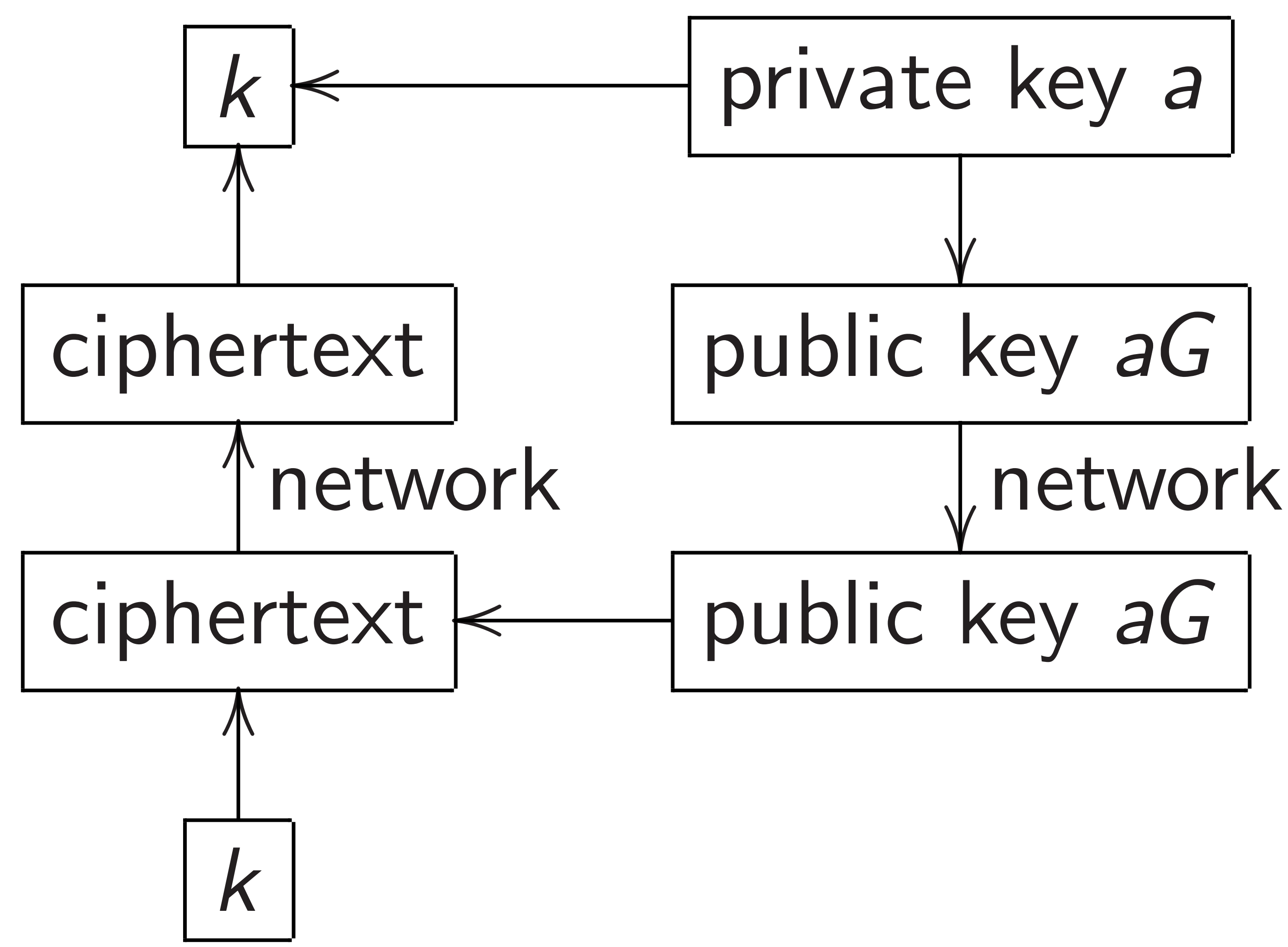
Cryptographic solution: Message-authentication codes.



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

What if attacker was spying on their communication of k?

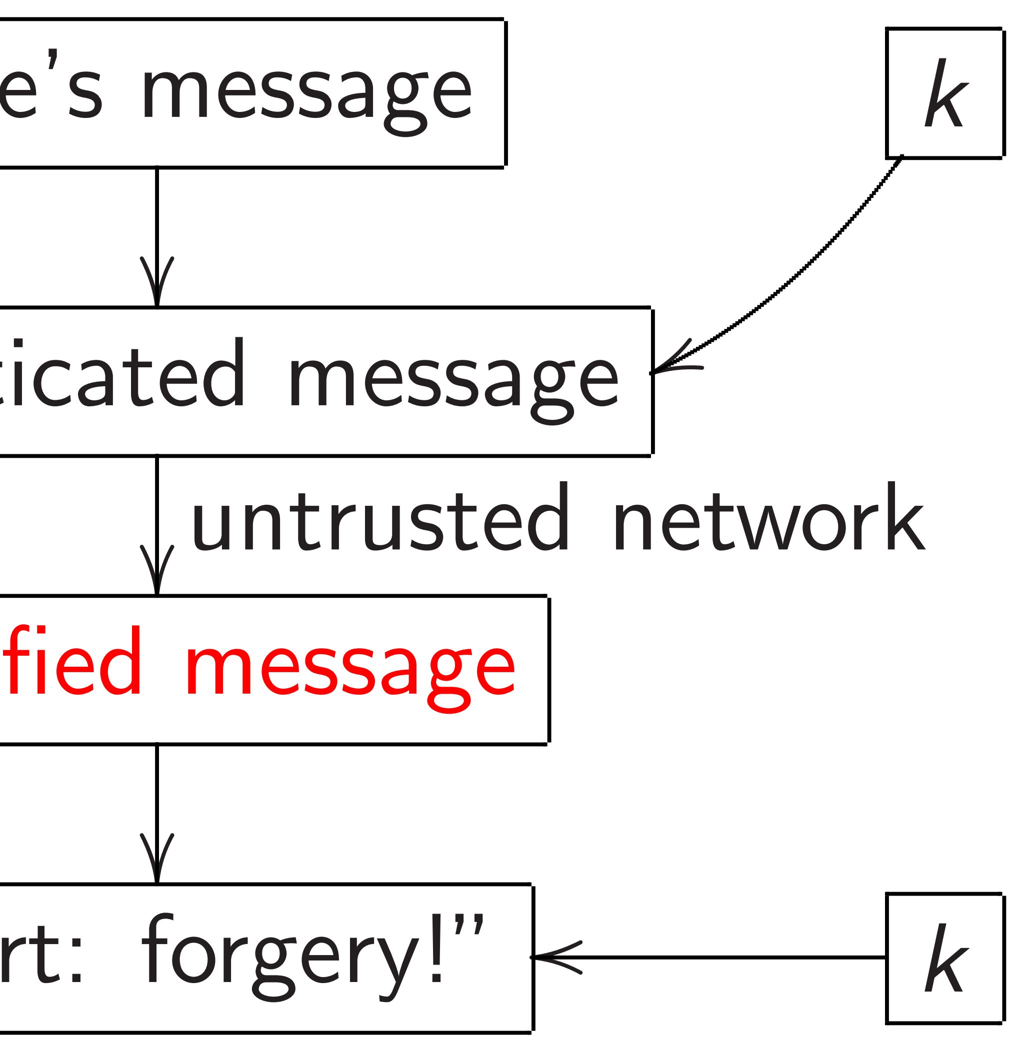
Solution 1: Public-key encryption.



raphy

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

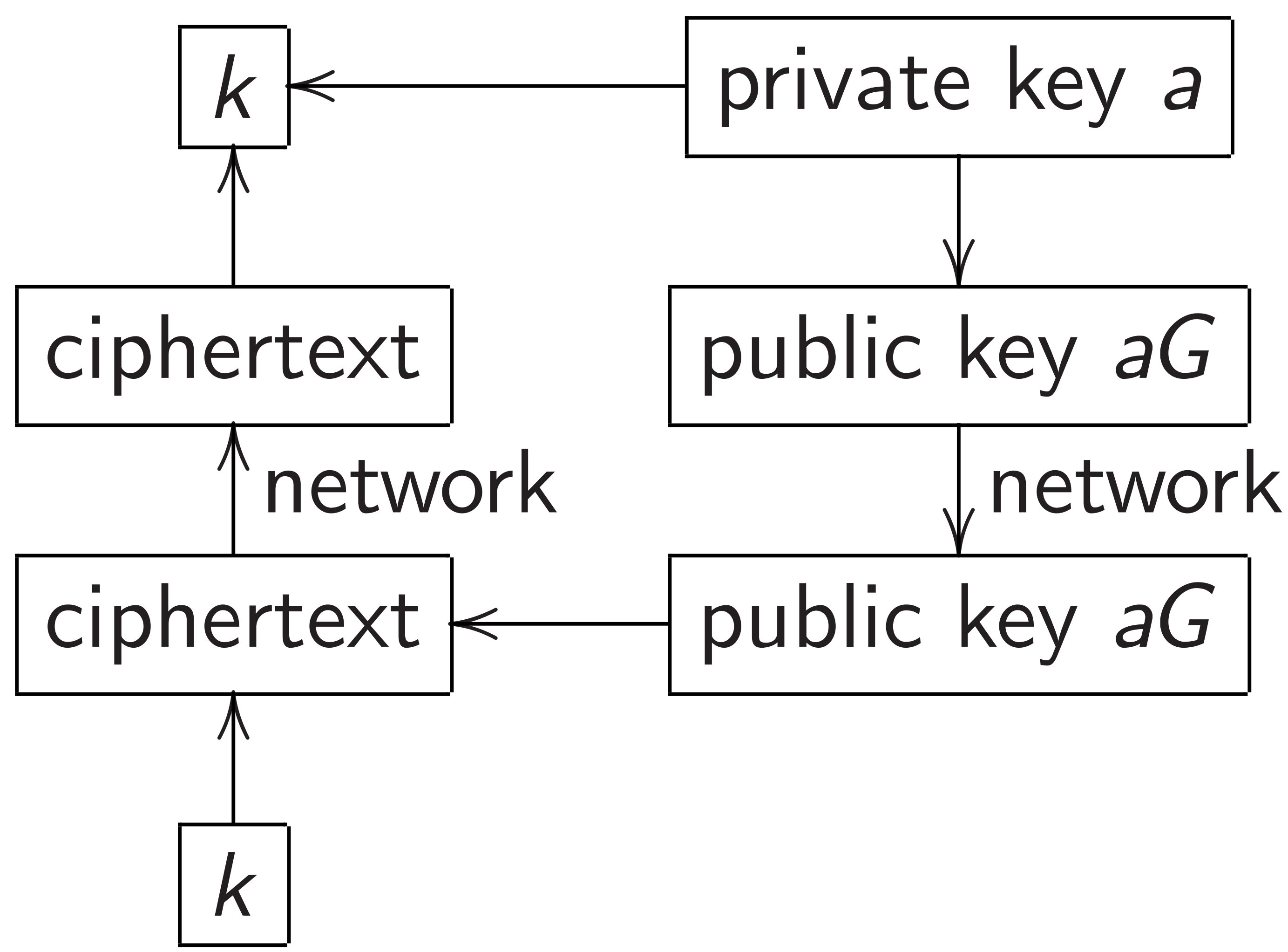
raphic solution:
e-authentication codes.



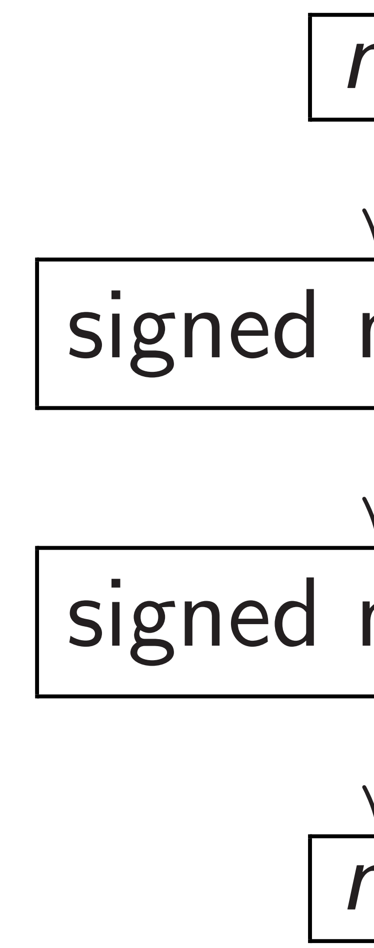
Important for Alice and Bob
to share the same secret k.

What if attacker was spying
on their communication of k?

Solution 1:
Public-key encryption.

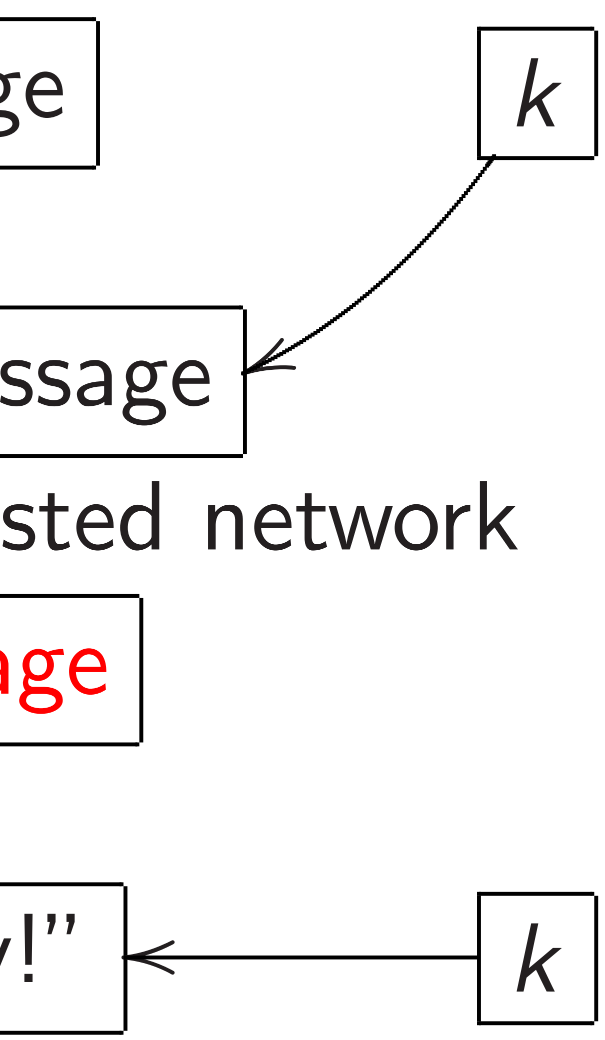


Solution
Public-k



aptop know
work data
top?

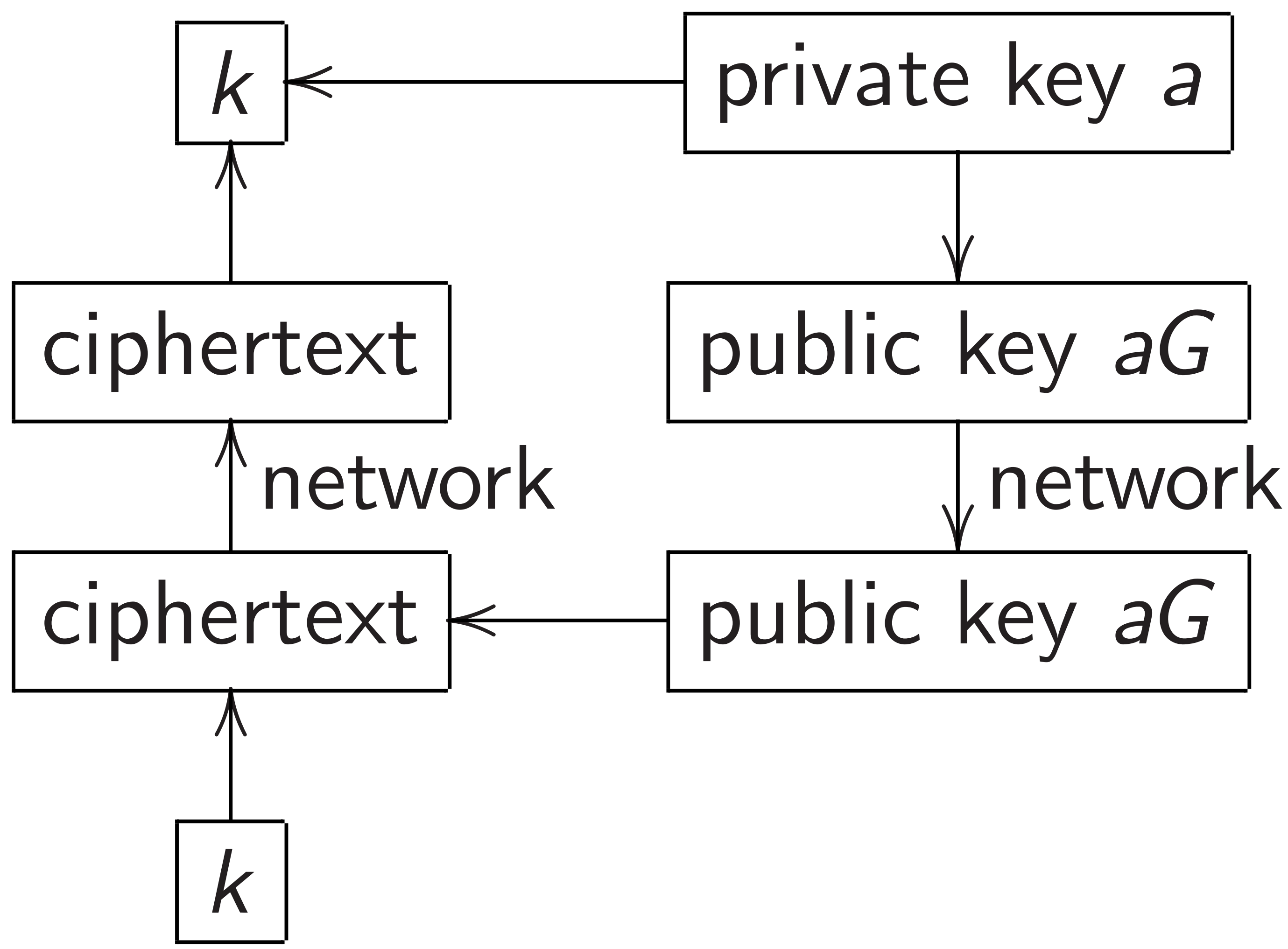
ution:
cation codes.



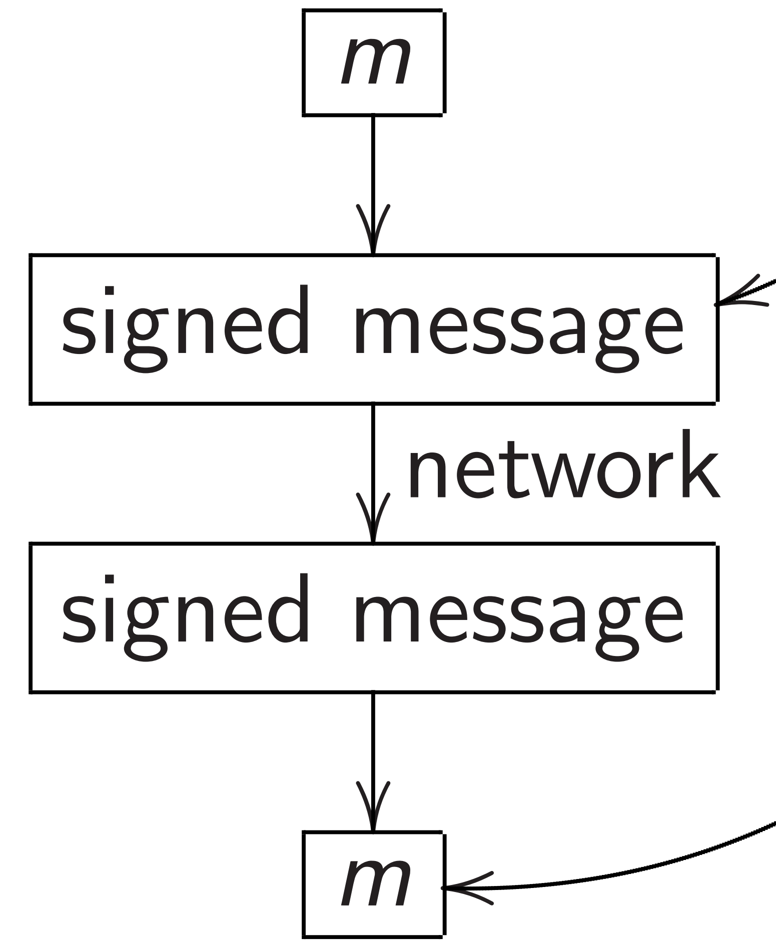
Important for Alice and Bob
to share the same secret k.

What if attacker was spying
on their communication of k?

Solution 1:
Public-key encryption.



Solution 2:
Public-key signatu



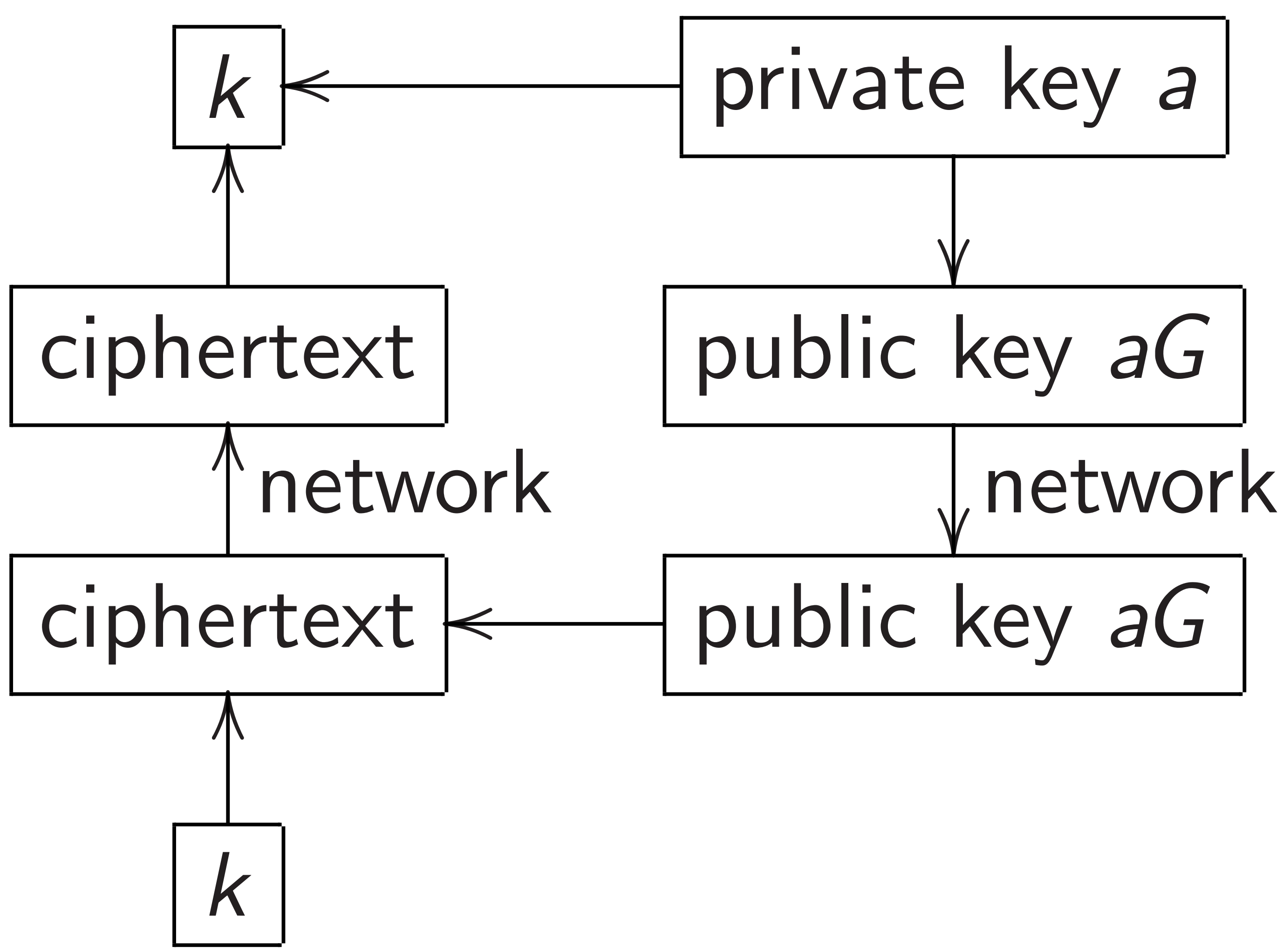
w

es.

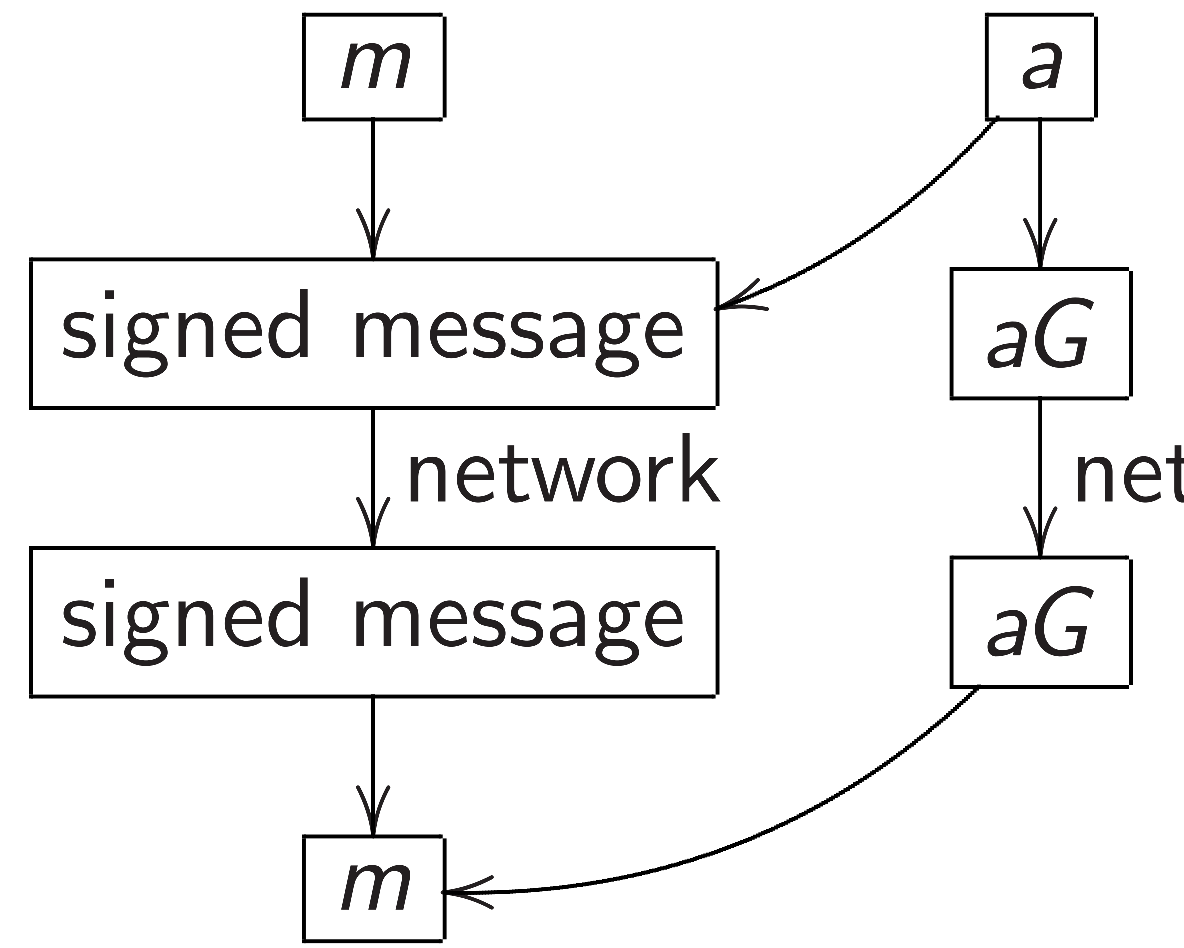
Important for Alice and Bob to share the same secret k.

What if attacker was spying on their communication of k?

Solution 1: Public-key encryption.



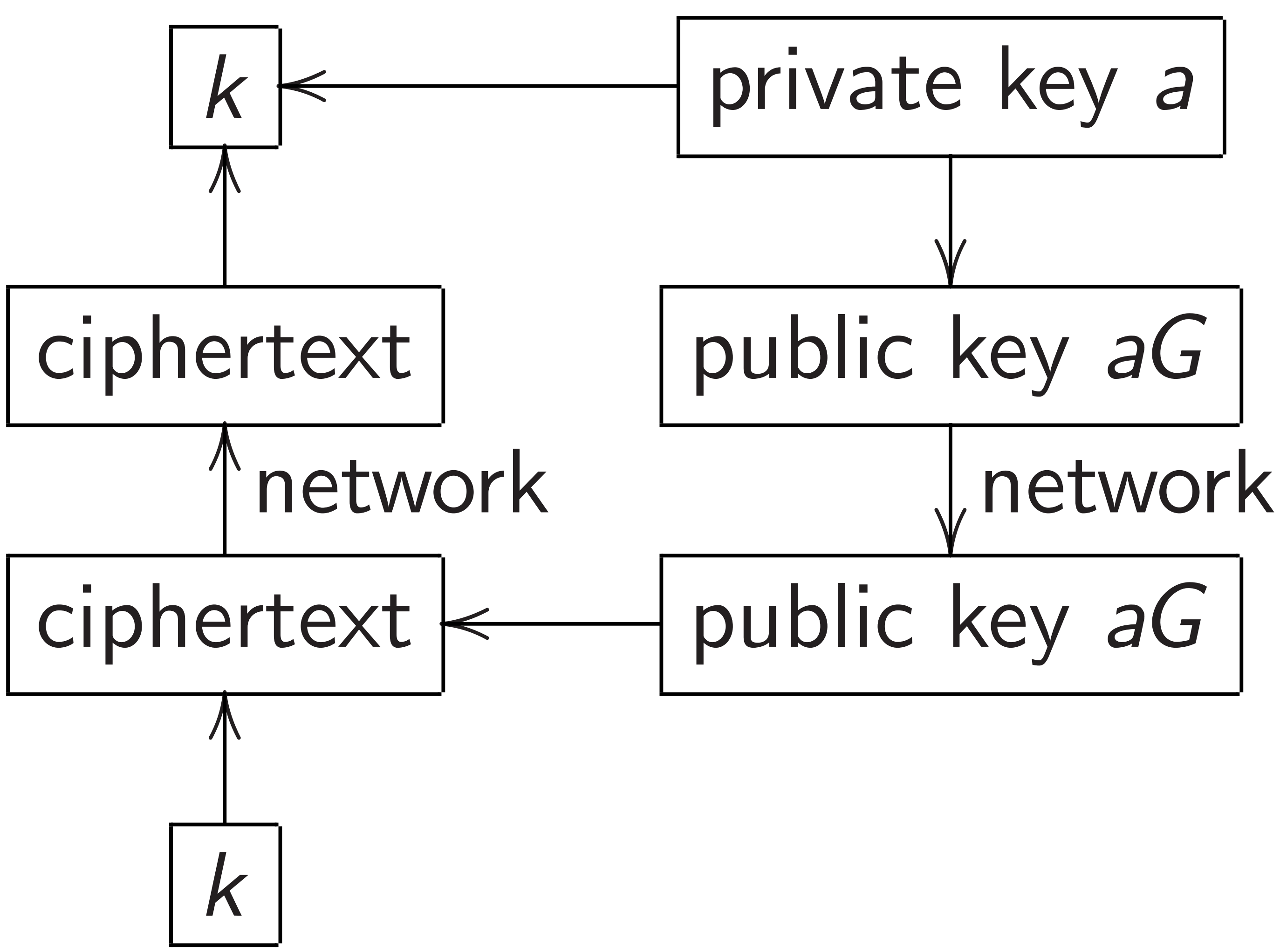
Solution 2: Public-key signatures.



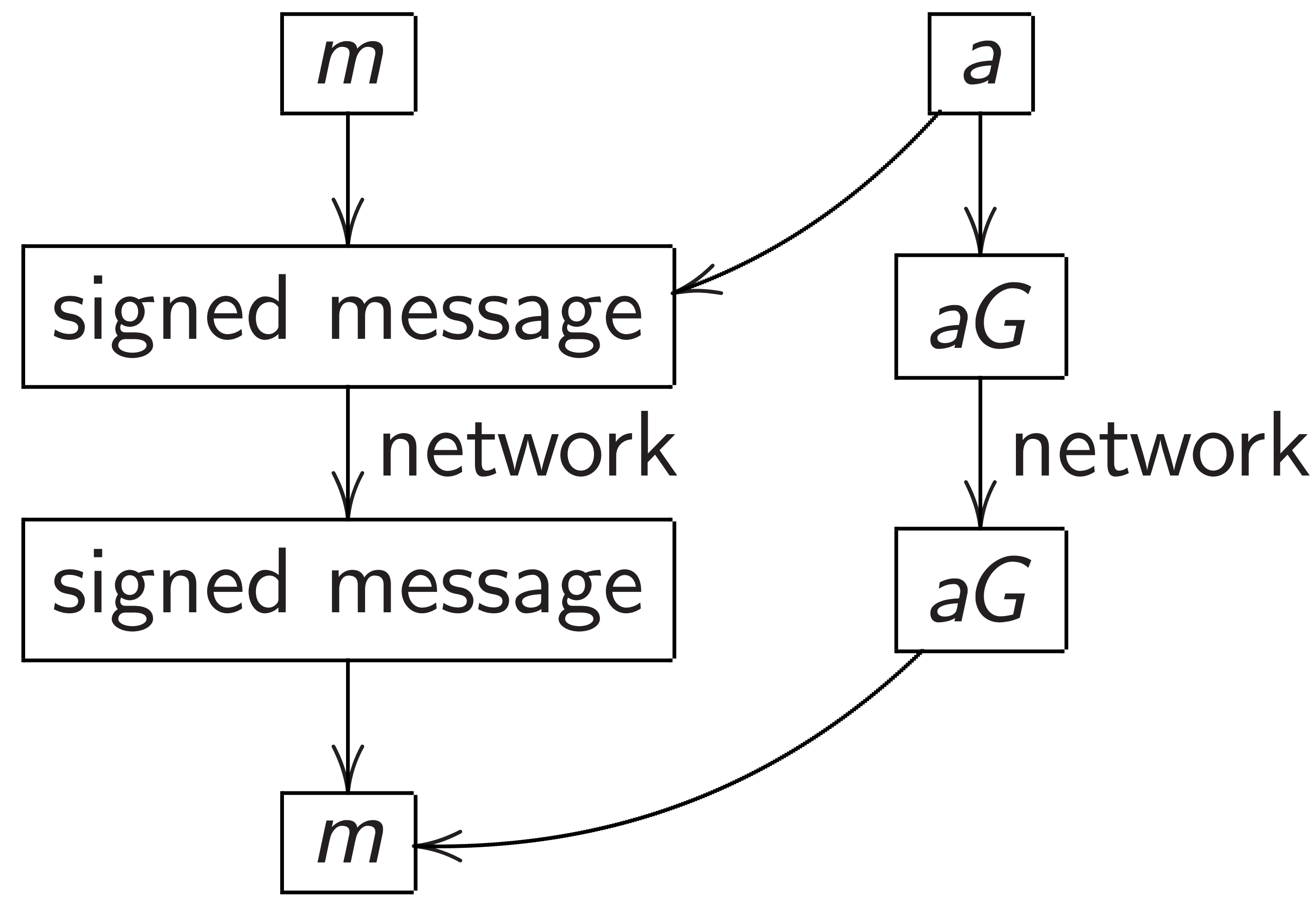
Important for Alice and Bob
to share the same secret k.

What if attacker was spying
on their communication of k?

Solution 1:
Public-key encryption.



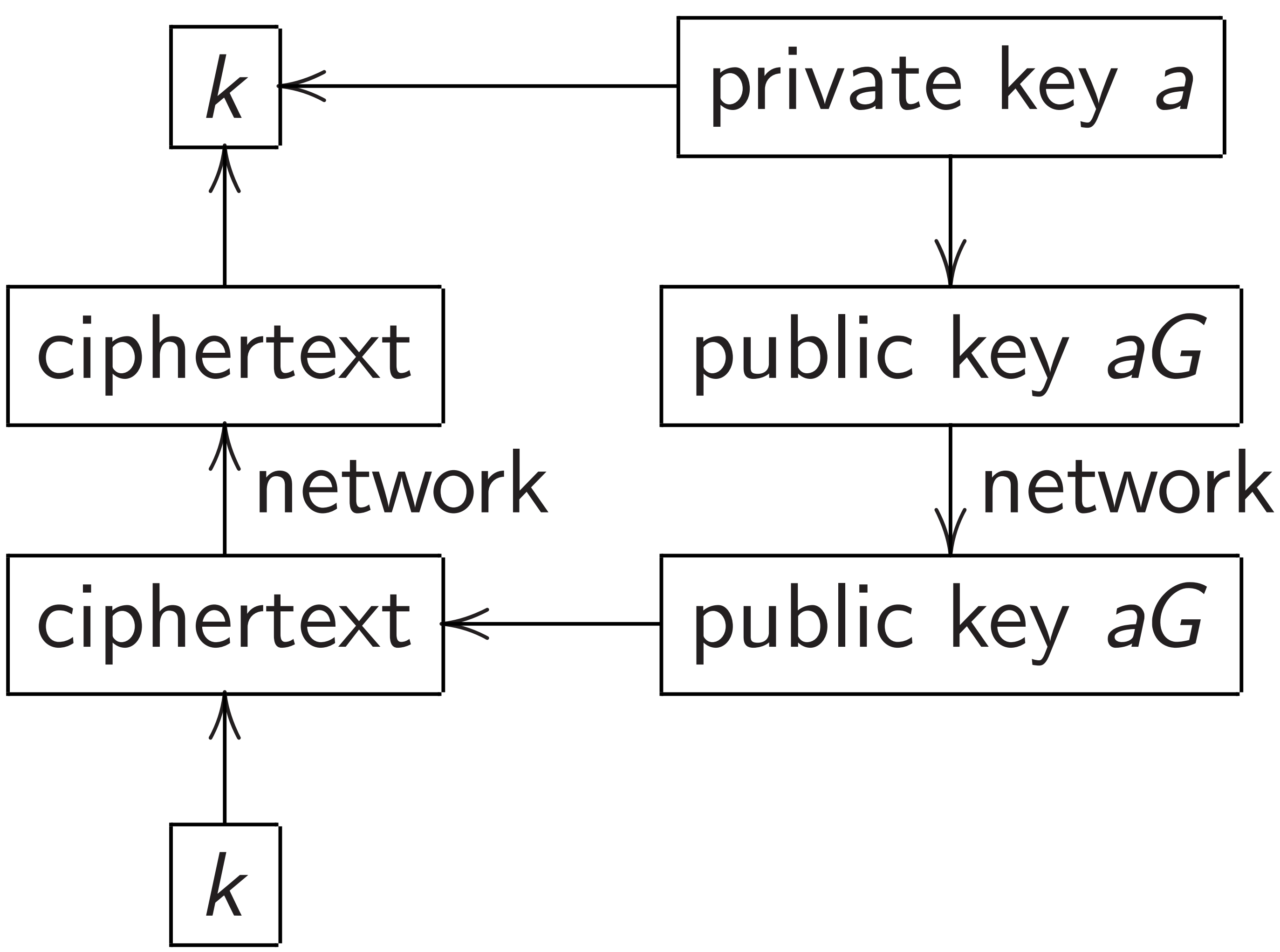
Solution 2:
Public-key signatures.



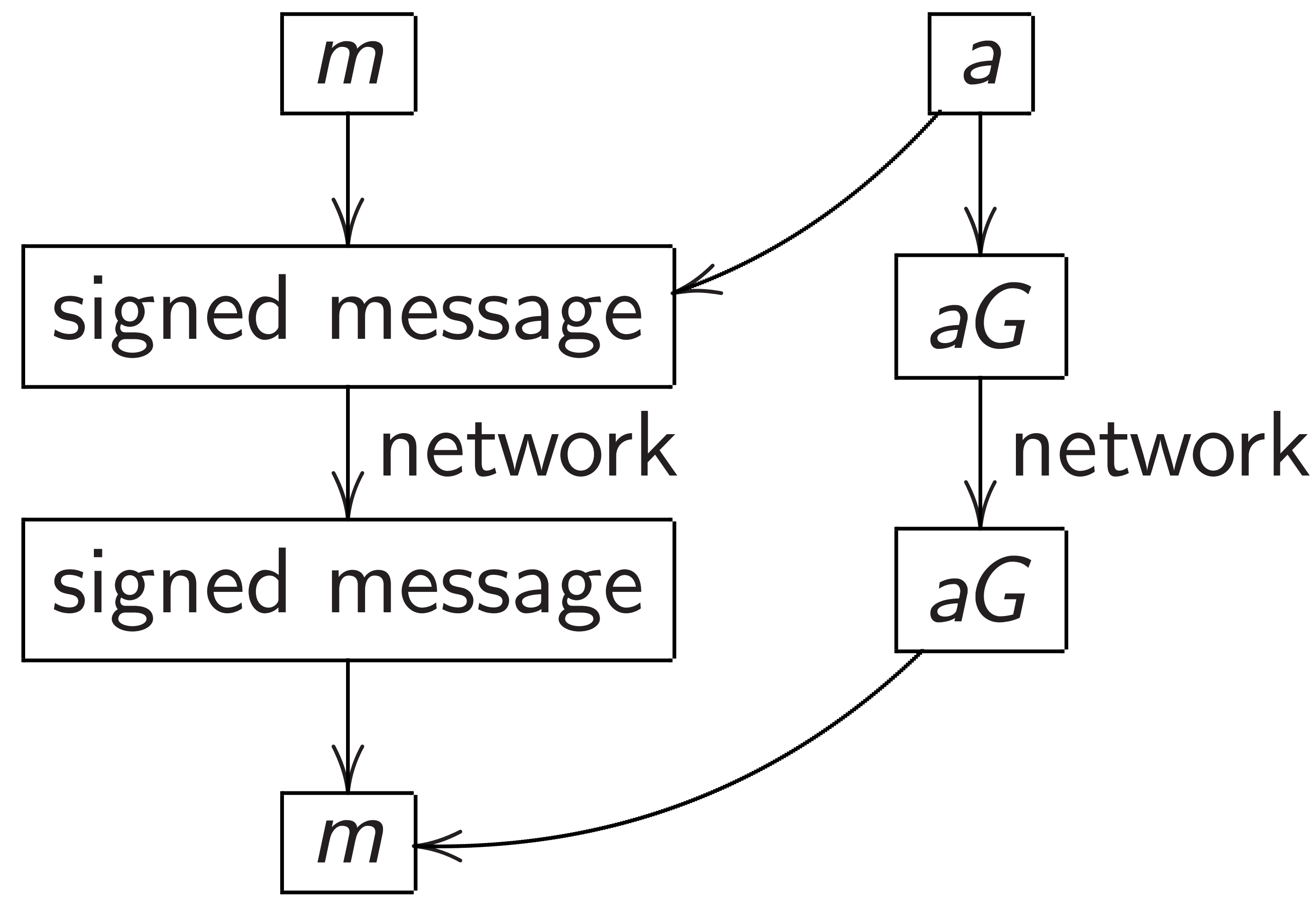
Important for Alice and Bob
to share the same secret k.

What if attacker was spying
on their communication of k?

Solution 1:
Public-key encryption.



Solution 2:
Public-key signatures.

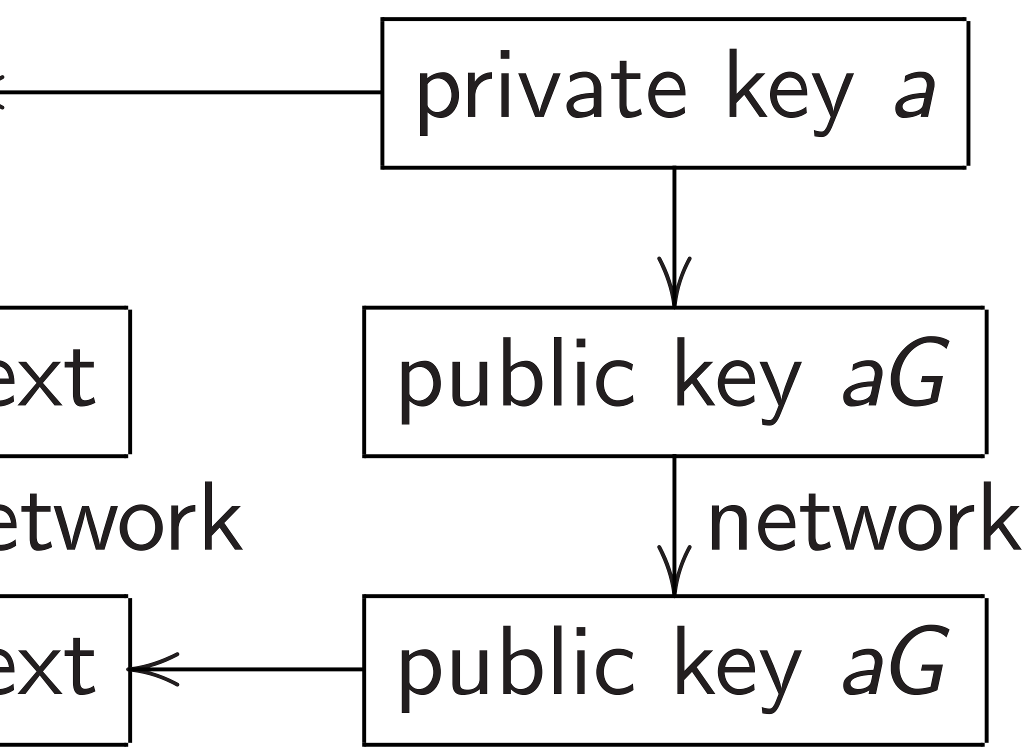


No more shared secret k
but Alice still has secret a.
Cryptography requires TCB
to protect secrecy of keys,
even if user has no other secrets.

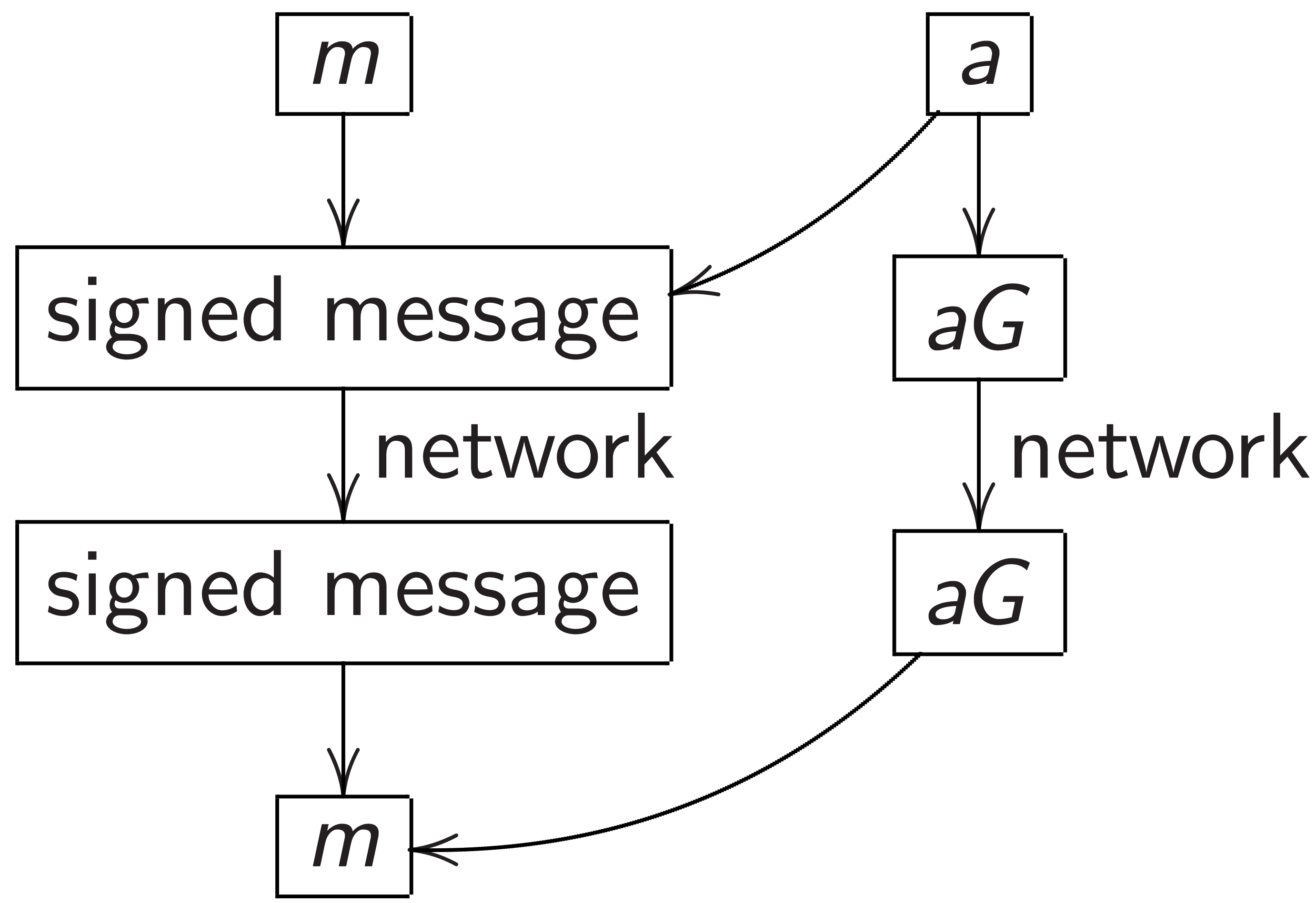
nt for Alice and Bob
the same secret k.

attacker was spying
communication of k?

1:
ey 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.

Constan

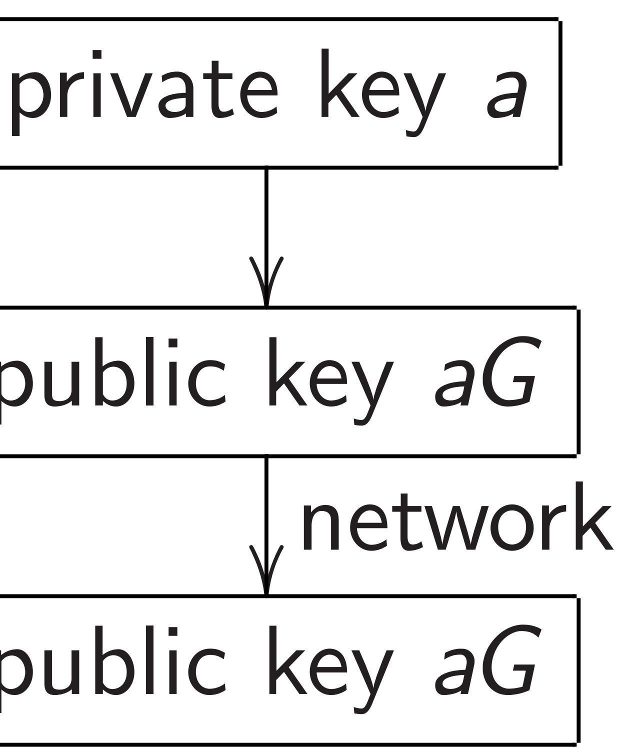
Large po
optimiza
addresse

Conside
instructi
parallel
store-to-
branch p

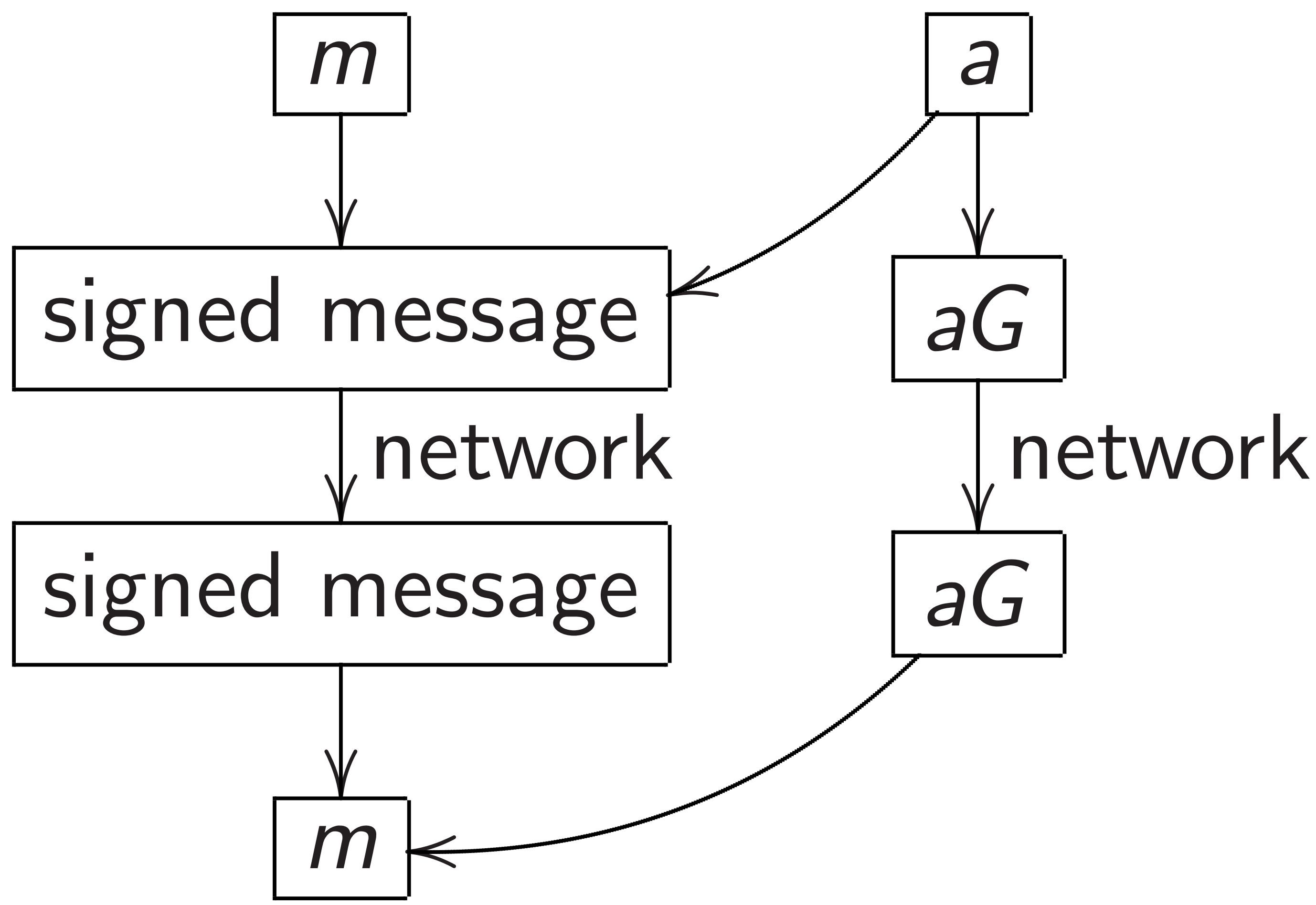
e and Bob
secret k.

vas spying
cation 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

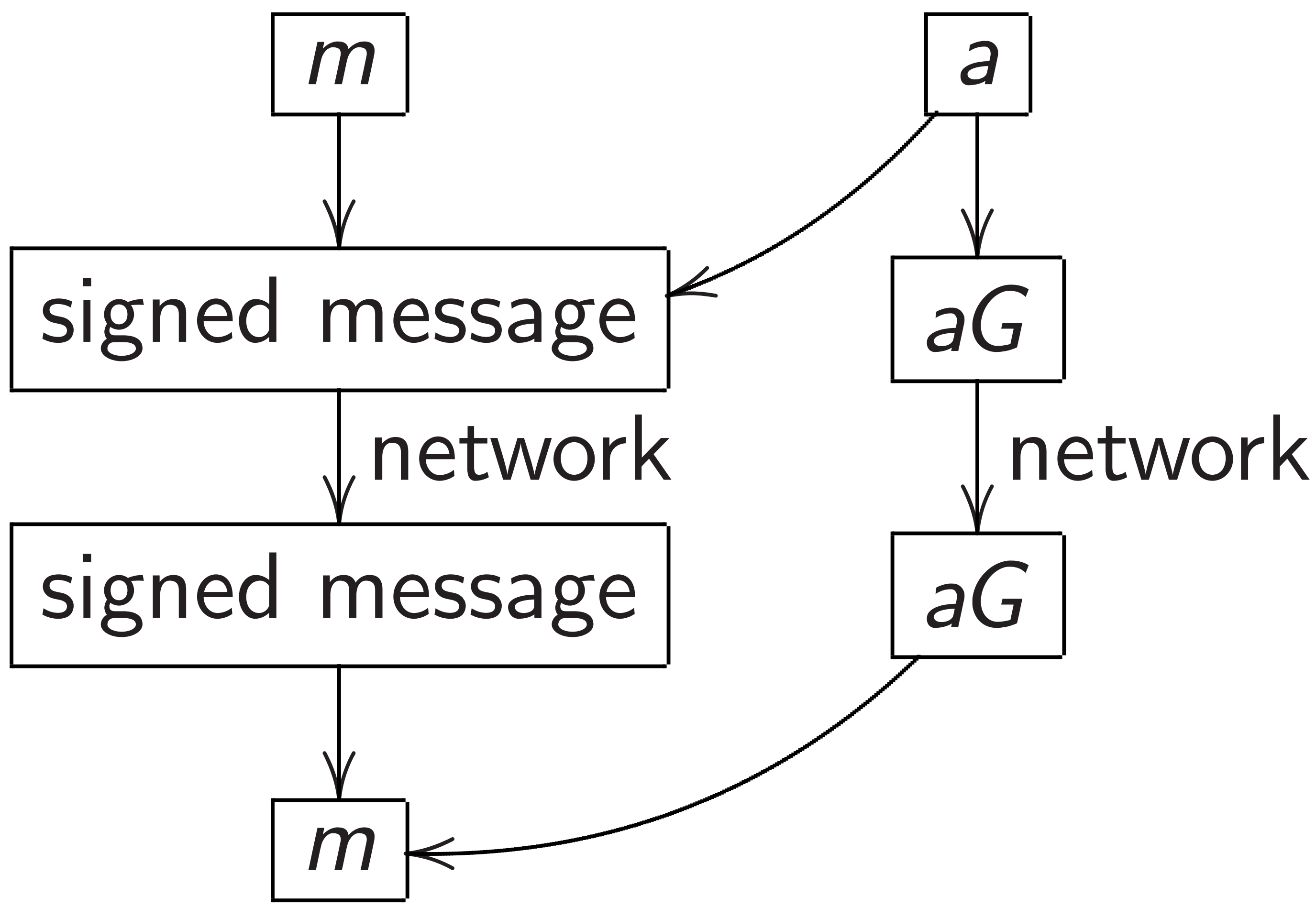
Large portion of C
optimizations depe
addresses of memo

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

k?

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.

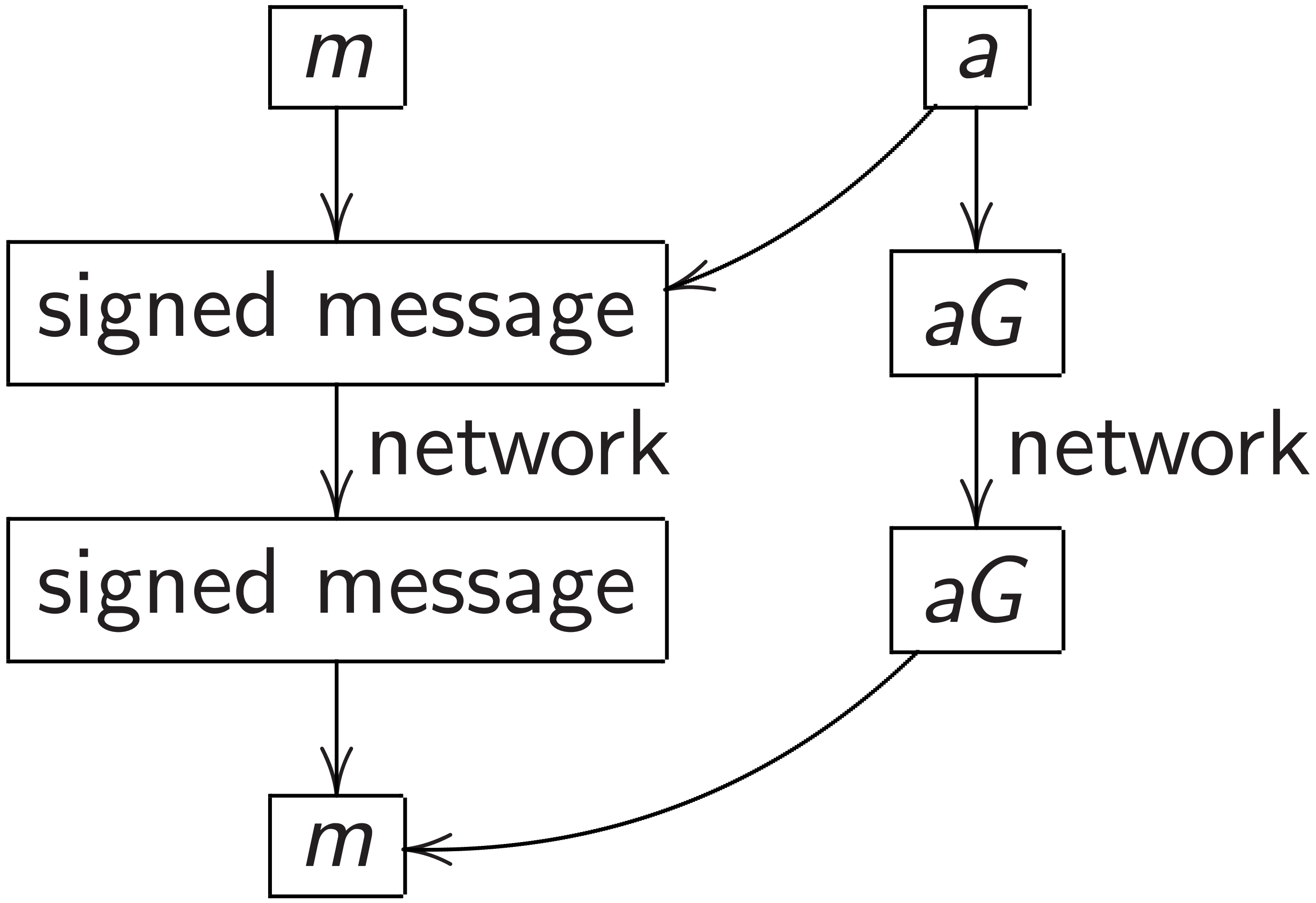
Constant-time software

Large portion of CPU hardw
optimizations depending on
addresses of memory locatio

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.

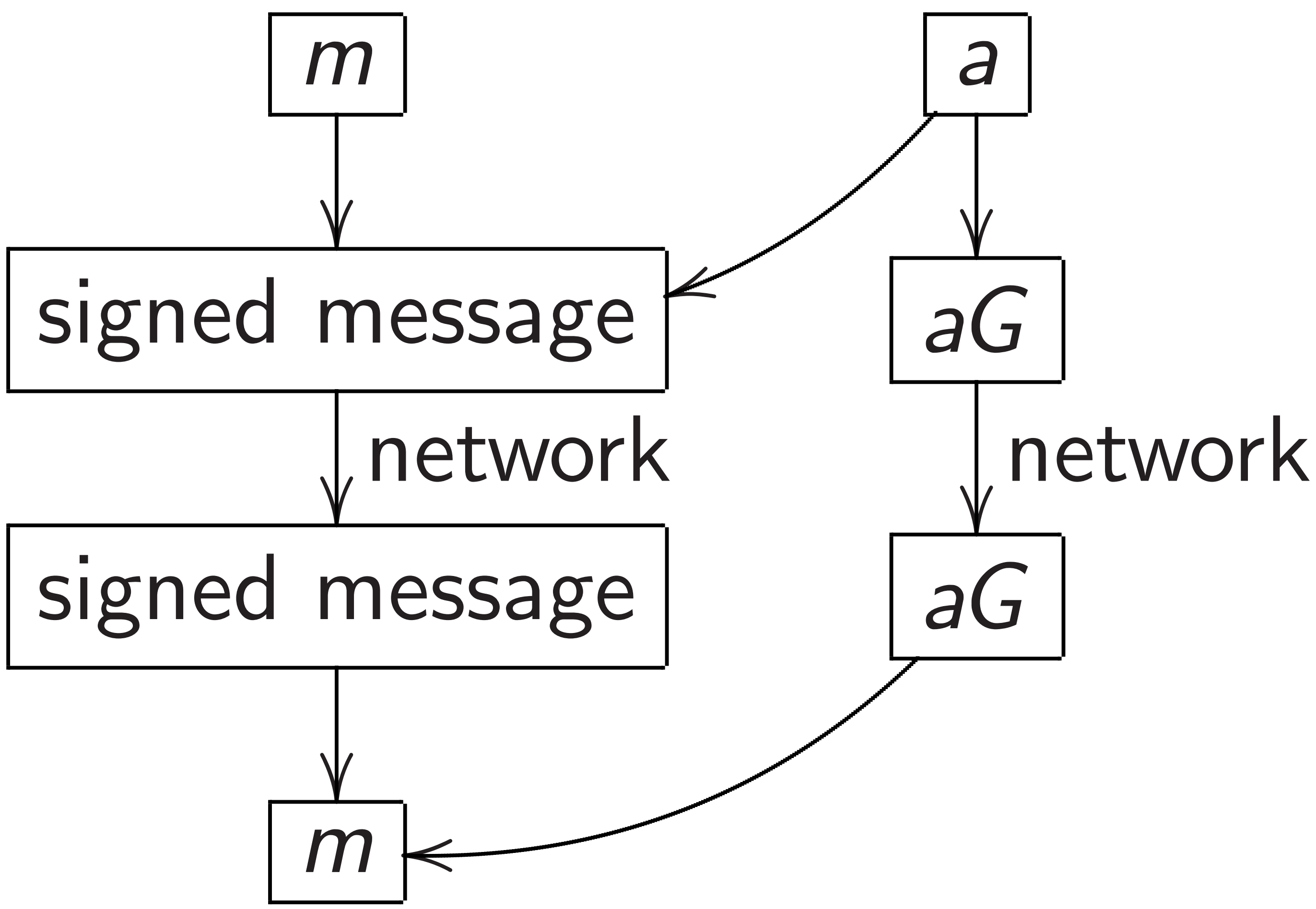
Constant-time software

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

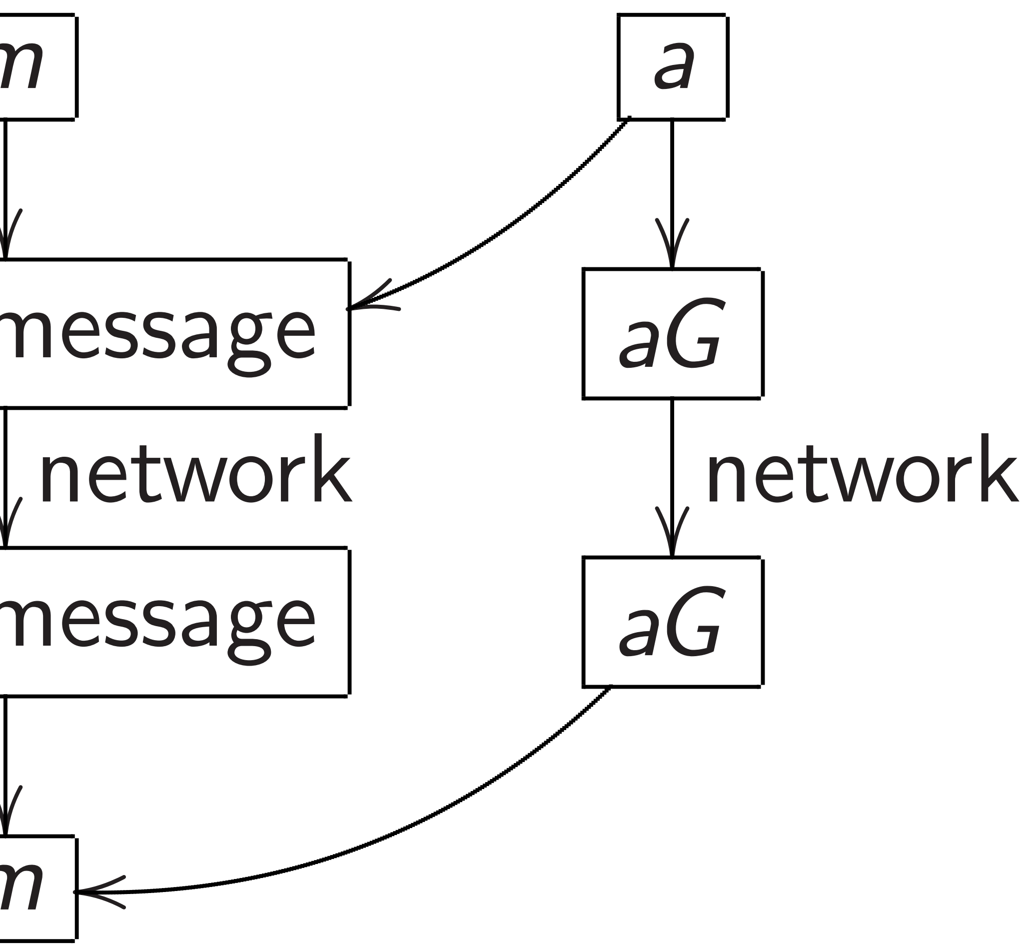
Large portion of CPU hardware: optimizations depending on addresses of memory locations.

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

Many attacks show that this portion of the CPU has trouble keeping secrets. e.g. RIDL: 2019 Schaik–Milburn–Österlund–Frigo–Maisuradze–Razavi–Bos–Giuffrida.

2:

ey signatures.



e shared secret k

e still has secret a.

graphy requires TCB

ect secrecy of keys,

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 show that this portion of the CPU has trouble keeping secrets. e.g. RIDL: 2019 Schaik–Milburn–Österlund–Frigo–Maisuradze–Razavi–Bos–Giuffrida.

Typical

Understa

But deta

not expo

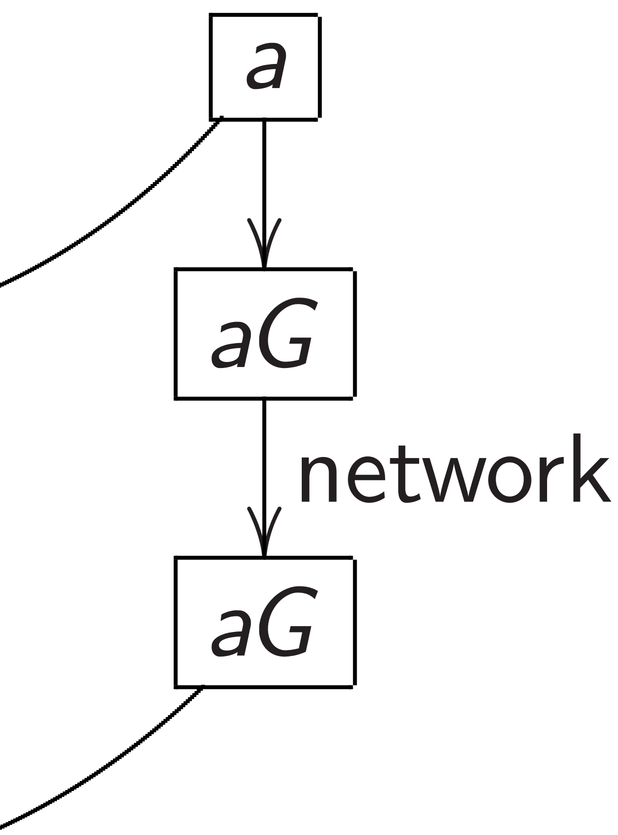
Try to p

This bec

Tweak t

to try to

res.



ecret k
secret a.
quires TCB
cy of keys,
o 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 show that this portion of the CPU has trouble keeping secrets. e.g. RIDL: 2019 Schaik–Milburn–Österlund–Frigo–Maisuradze–Razavi–Bos–Giuffrida.

Typical literature c

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

twork

EB

,

crets.

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 show that this portion of the CPU has trouble keeping secrets. e.g. RIDL: 2019 Schaik–Milburn–Österlund–Frigo–Maisuradze–Razavi–Bos–Giuffrida.

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

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 show that this portion of the CPU has trouble keeping secrets. e.g. RIDL: 2019 Schaik–Milburn–Österlund–Frigo–Maisuradze–Razavi–Bos–Giuffrida.

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 show that this portion of the CPU has trouble keeping secrets. e.g. RIDL: 2019 Schaik–Milburn–Österlund–Frigo–Maisuradze–Razavi–Bos–Giuffrida.

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 show that this portion of the CPU has trouble keeping secrets. e.g. RIDL: 2019 Schaik–Milburn–Österlund–Frigo–Maisuradze–Razavi–Bos–Giuffrida.

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.

t-time software

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

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

ttacks show that this
of the CPU has trouble
secrets. e.g. RIDL: 2019
Milburn–Österlund–Frigo–
dze–Razavi–Bos–Giuffrida.

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-

tware

CPU hardware:
ending on
ory locations.

hing,
g,
ks,
arding,
etc.

w that this
J has trouble
.g. RIDL: 2019
Osterlund–Frigo–
vi–Bos–Giuffrida.

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
(1987 Goldreich, 1
Oblivious RAM; 20
domain-specific fo

are:

ns.

s
ble
2019
Frigo–
ffrida.

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
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
Attacker
breaking
public-ke
e.g., fin

on this topic:

ortion of CPU.

en proprietary,

curity review.

ks further.

complicated.

ed software

known attacks.

his 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 withi
Attacker has quan
breaking today's n
public-key crypto (
e.g., finding *a* give

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

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.

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

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.
990 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

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

on:

ovsky:
tein:
peed)

rtion
ut
s.
.

d
s: e.g.,
tness
aks.

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.

udy: Constant-time sorting

risk within 10 years:
r has quantum computer
; today's most popular
ey crypto (RSA and ECC;
ding a given aG).

undreds of people
69 complete proposals
national competition for
antum crypto standards.

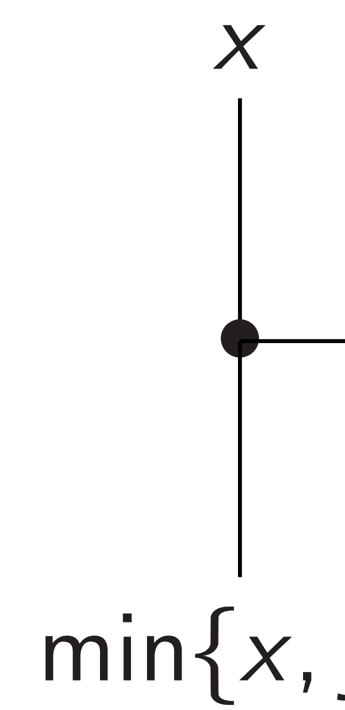
ine in some submissions:
y of secret integers.
768 32-bit integers.

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 cor
Warning
compiler

Even eas

tant-time sorting

10 years:
tum computer
nost popular
(RSA and ECC;
en aG).

f people
te proposals
ompetition for
pto standards.

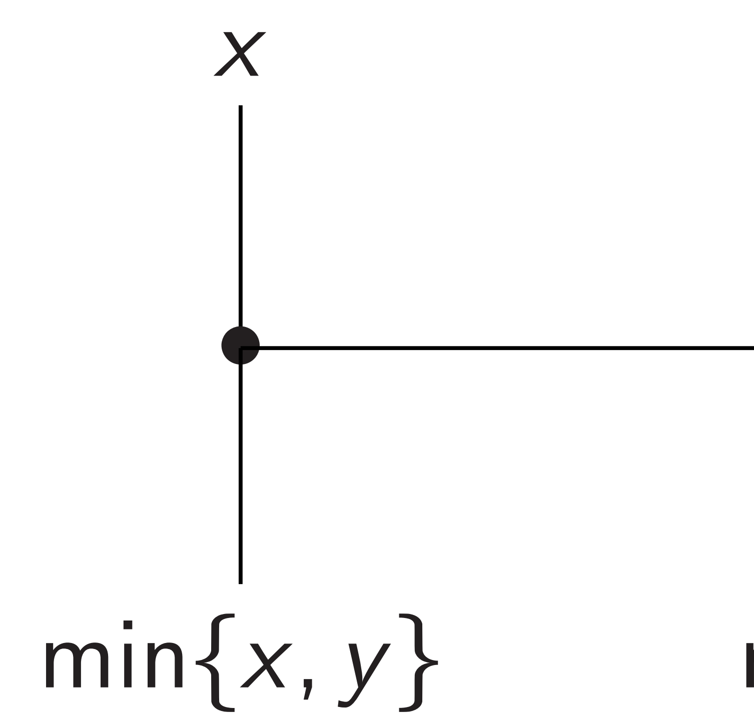
e submissions:
t integers.
t integers.

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 solu
a **comparator** sor



Easy constant-tim
Warning: C stand
compiler to screw

Even easier exercis

sorting

puter
lar
ECC;

ls
for
rds.

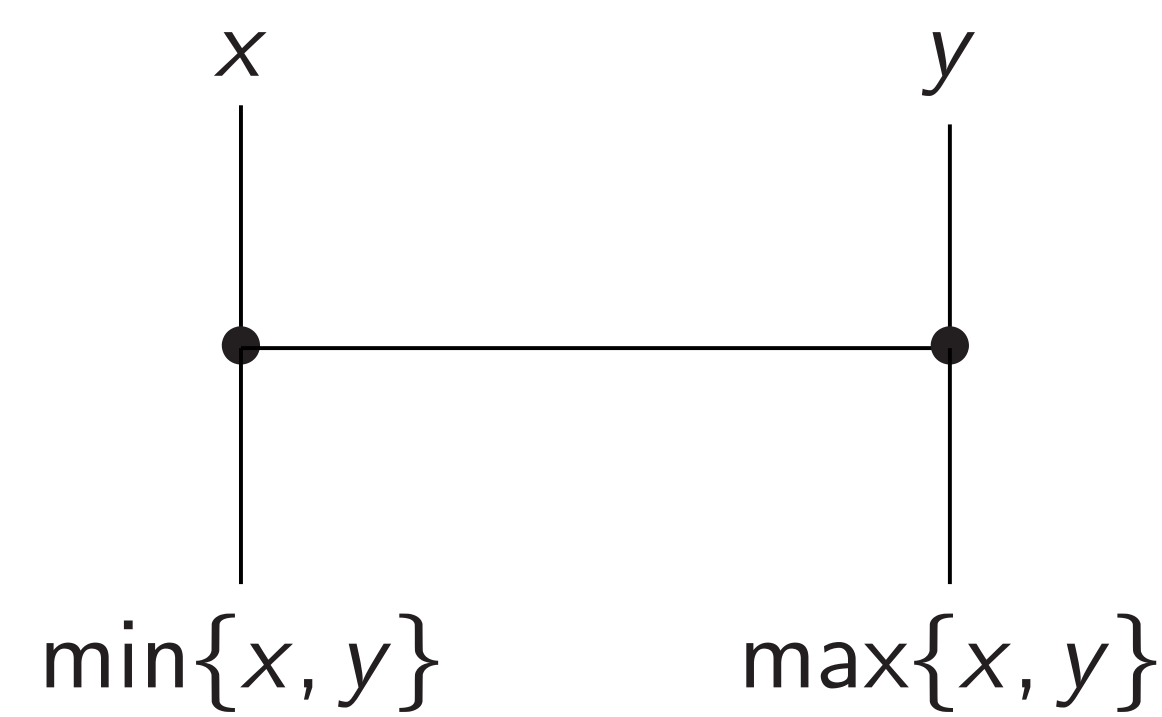
ons:

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 inte



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

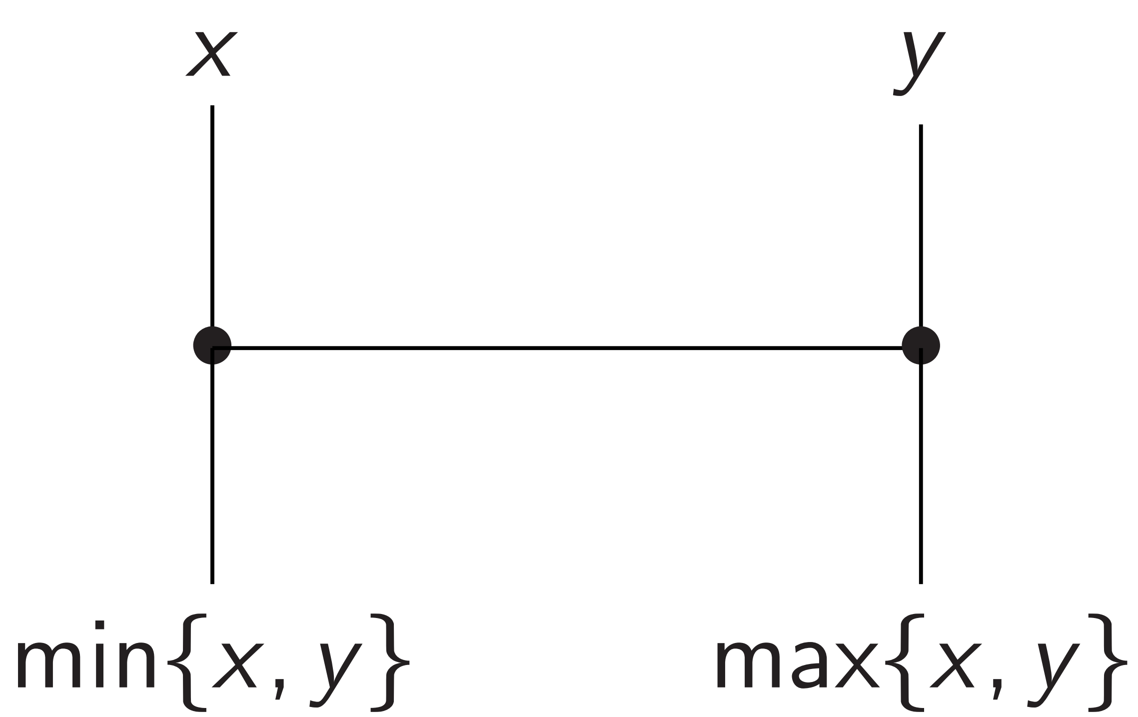
Even easier exercise in asm.

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.

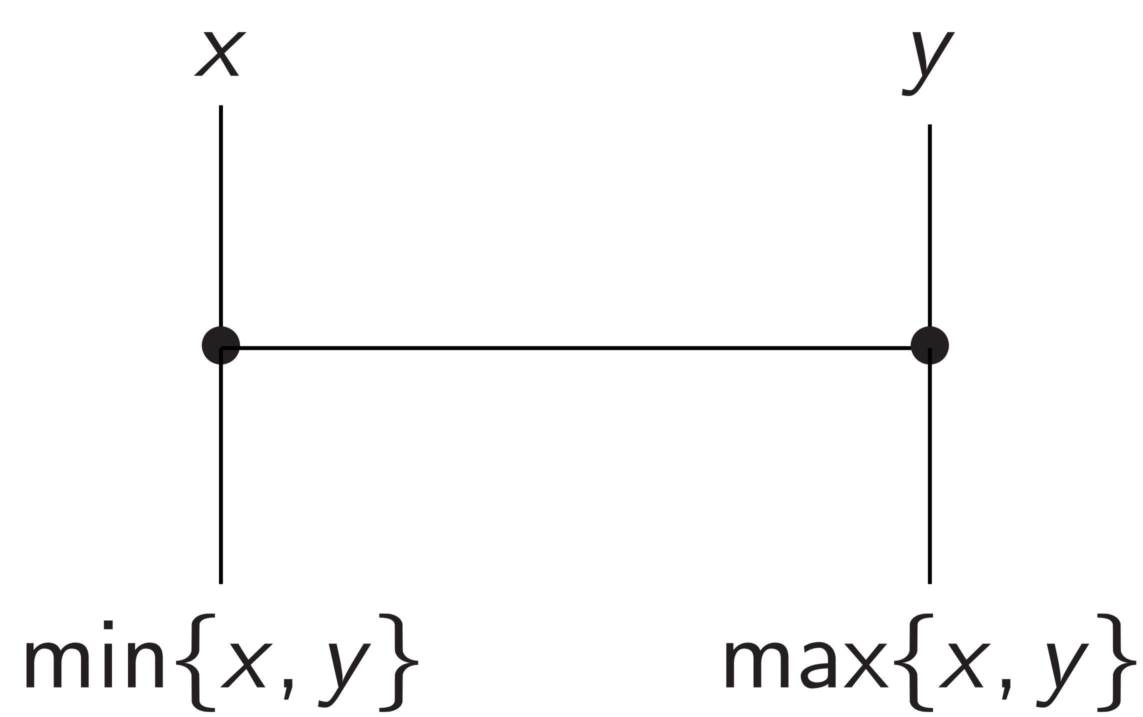
Even easier exercise in asm.

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
t-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.

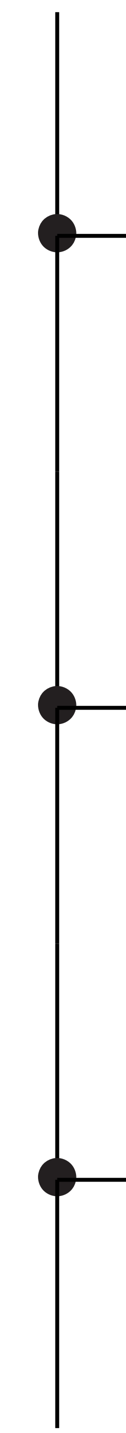


Easy constant-time exercise in C.
Warning: C standard allows
compiler to screw this up.

Even easier exercise in asm.

Combine
sorting

Example

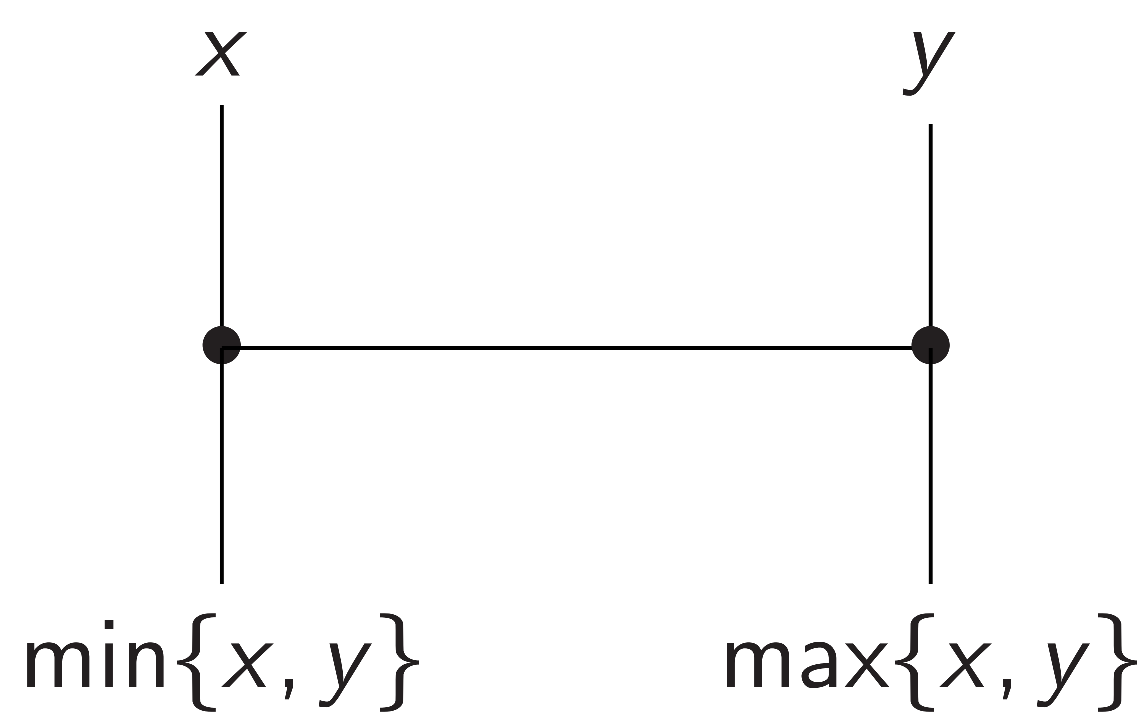


t data
t addresses?

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

competition:
d as
ing algorithm."
radix sort
hes.
s in radix sort
crets.

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

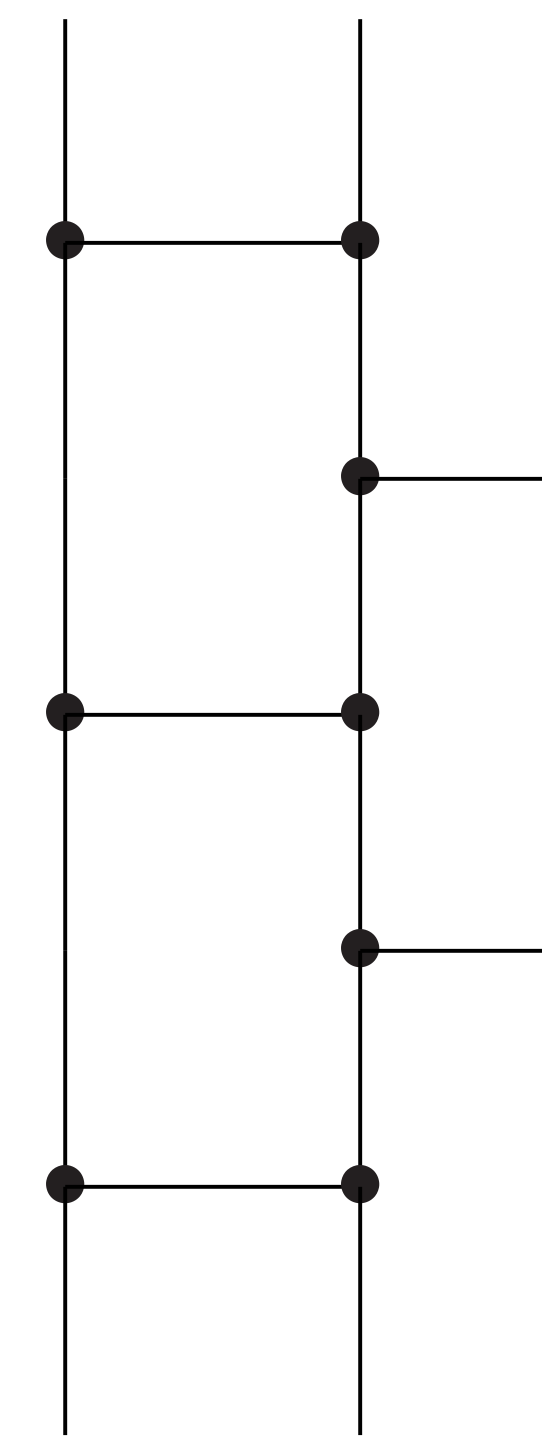


Easy constant-time exercise in C.
Warning: C standard allows
compiler to screw this up.

Even easier exercise in asm.

Combine compara
sorting network f

Example of a sorti



s?

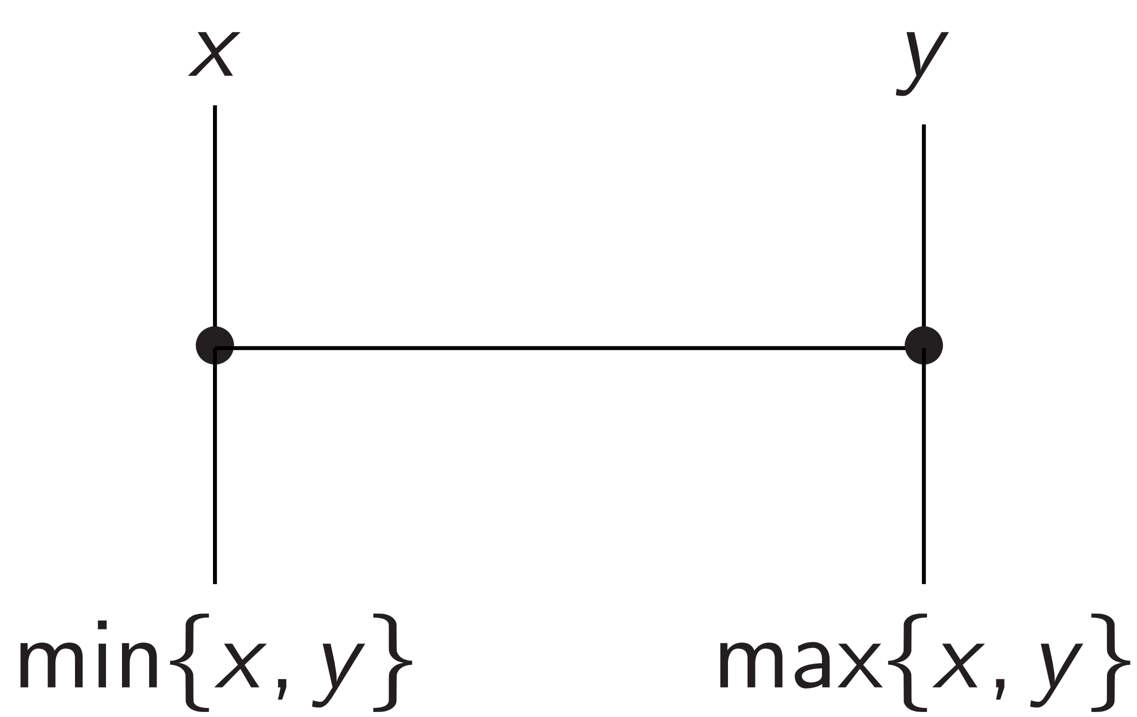
ly
data.

ion:

thm.”

sort

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

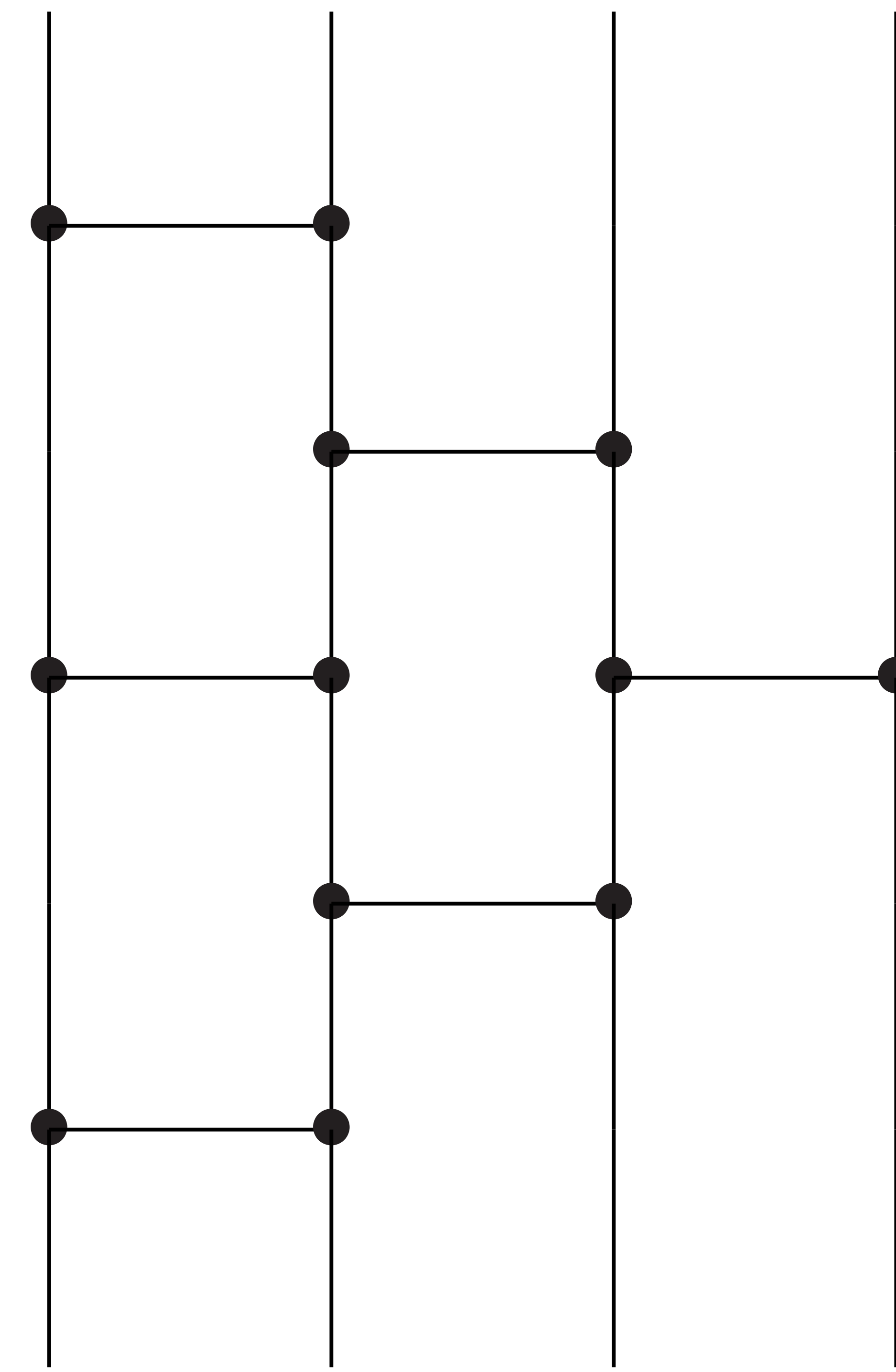


Easy constant-time exercise in C.
Warning: C standard allows
compiler to screw this up.

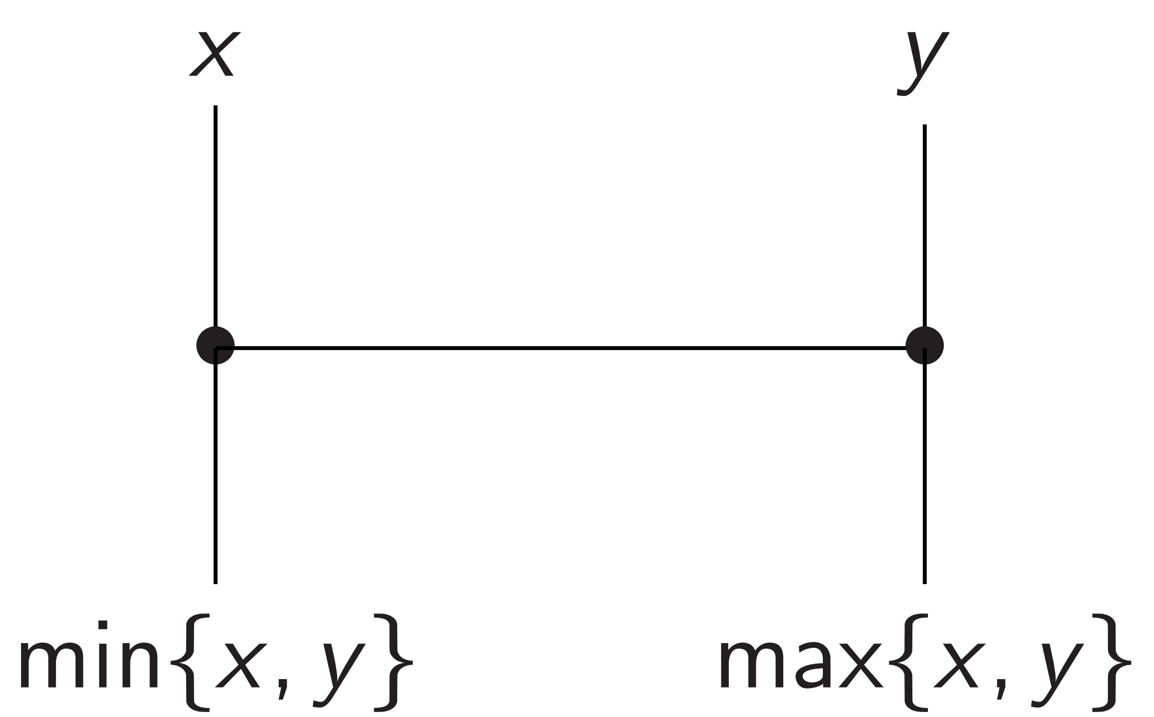
Even easier exercise in asm.

Combine comparators into a
sorting network for more in

Example of a sorting networ



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

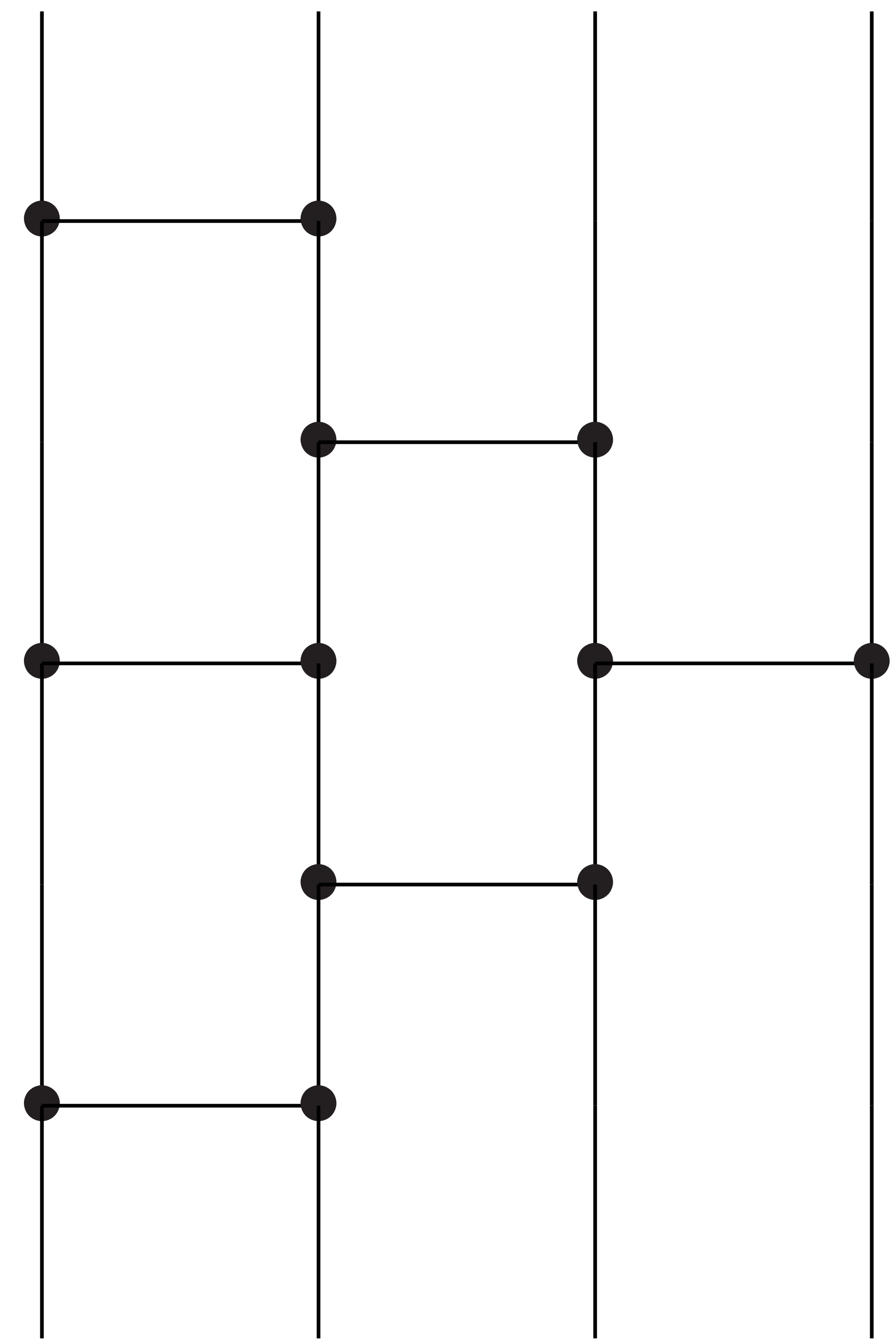


Easy constant-time exercise in C.
Warning: C standard allows compiler to screw this up.

Even easier exercise in asm.

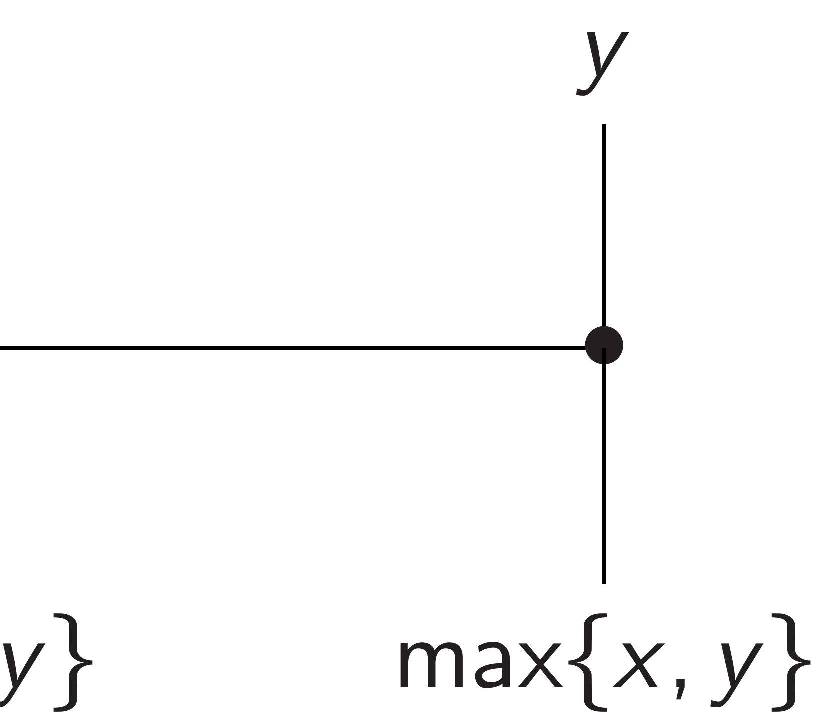
Combine comparators into a **sorting network** for more inputs.

Example of a sorting network:



ion of solution:

arator sorting 2 integers.



nstant-time exercise in C.

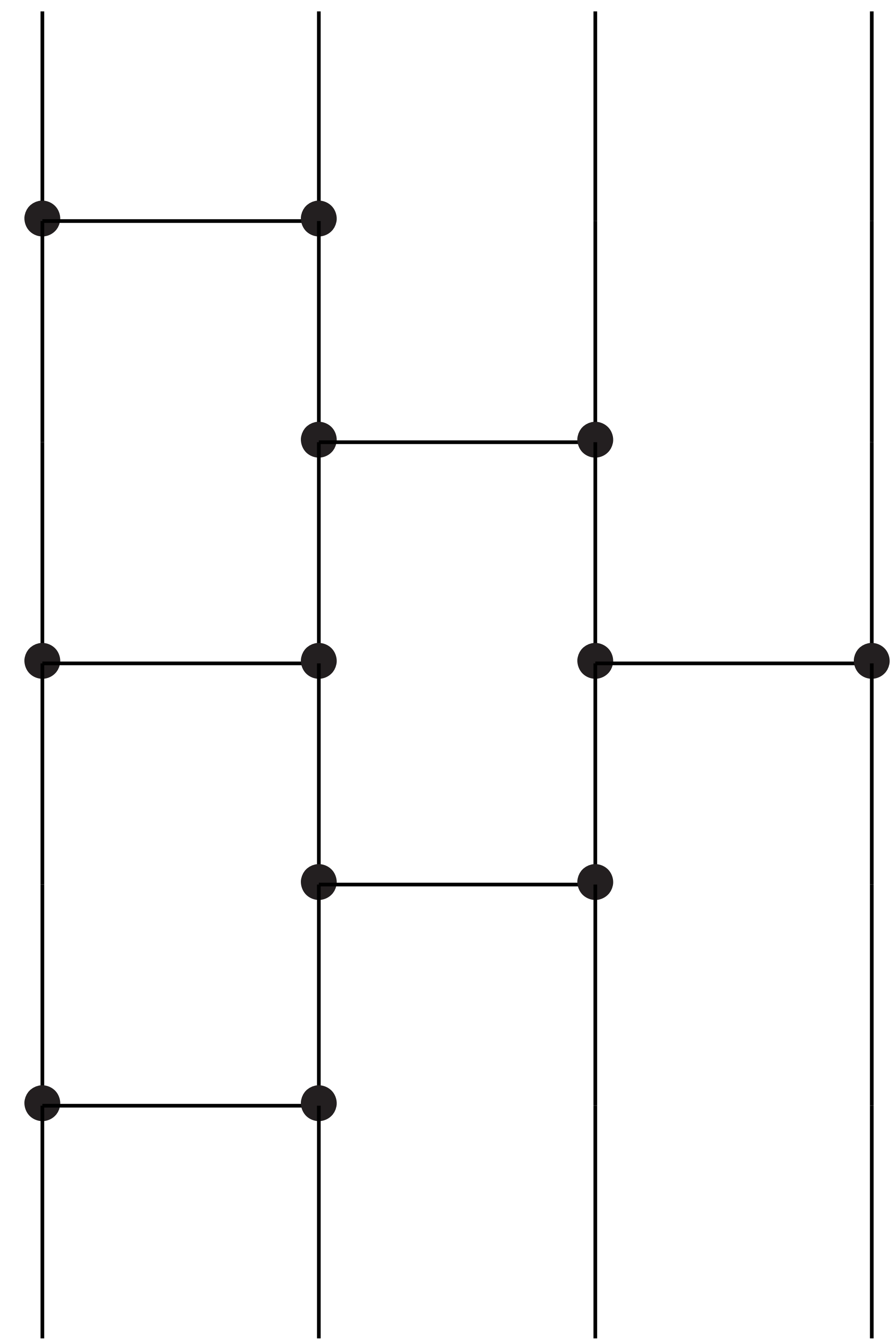
;: C standard allows

r to screw this up.

sier exercise in asm.

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

Example of a sorting network:



Positions

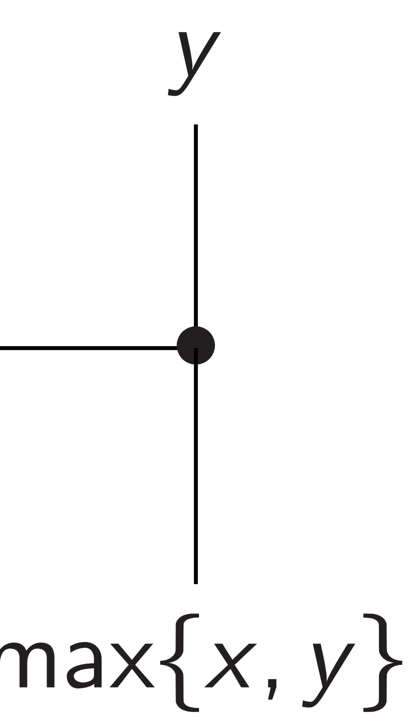
in a sort

independ

Naturally

ution:

ting 2 integers.

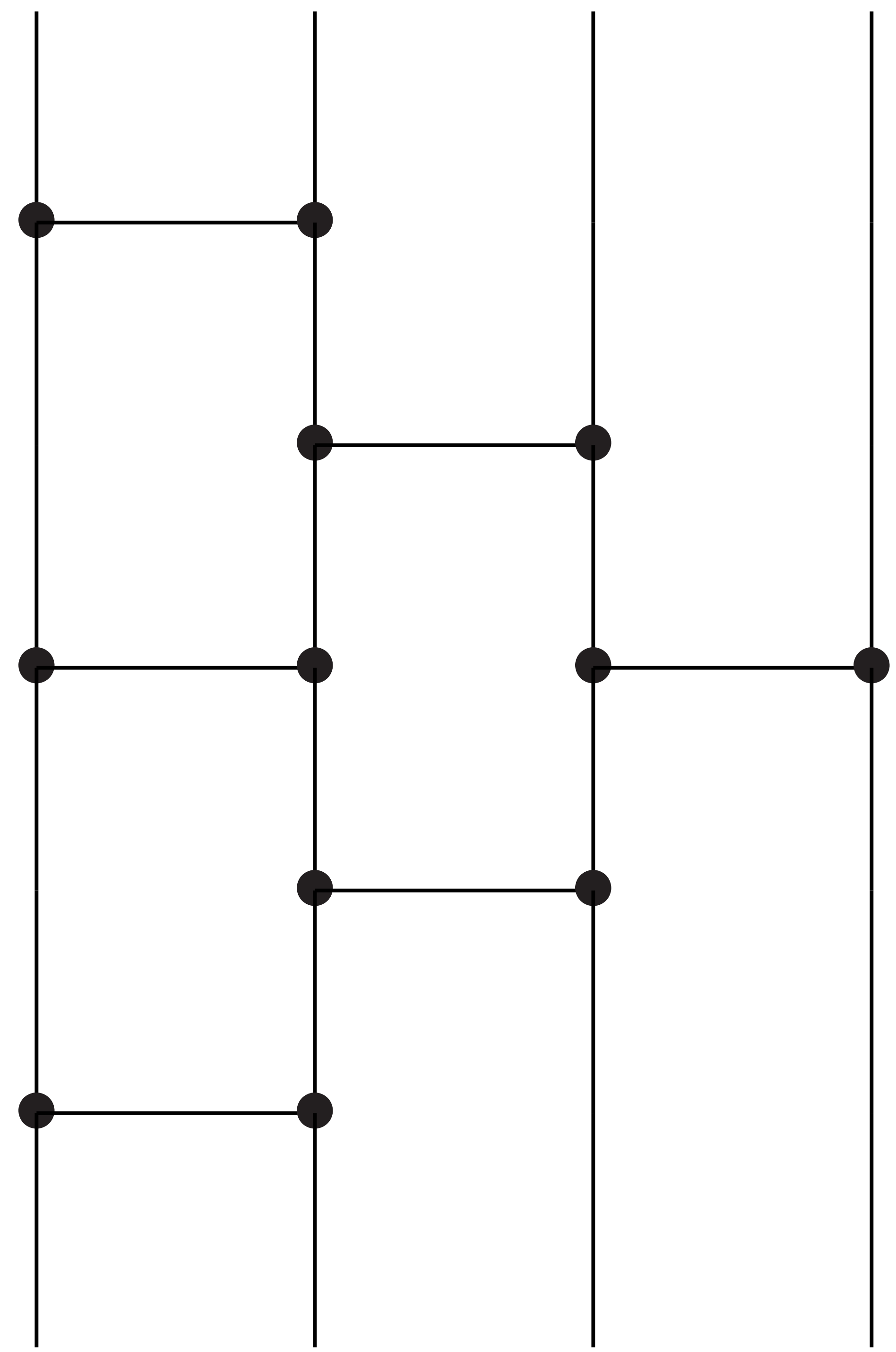


e exercise in C.
ard allows
this up.

se in asm.

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

Example of a sorting network:



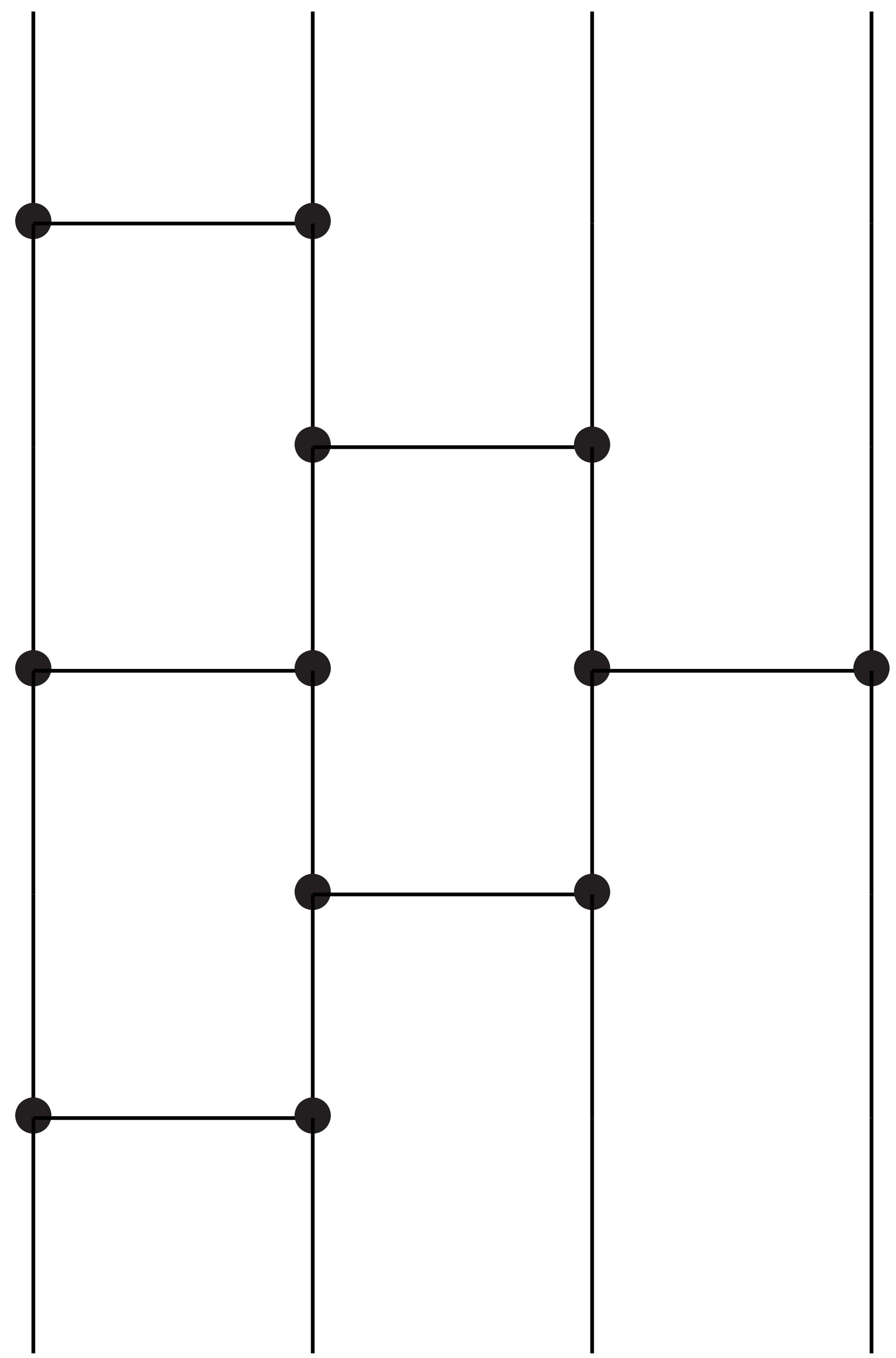
Positions of compa
in a sorting netwo
independent of th
Naturally constant

egers.

in C.

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

Example of a sorting network:

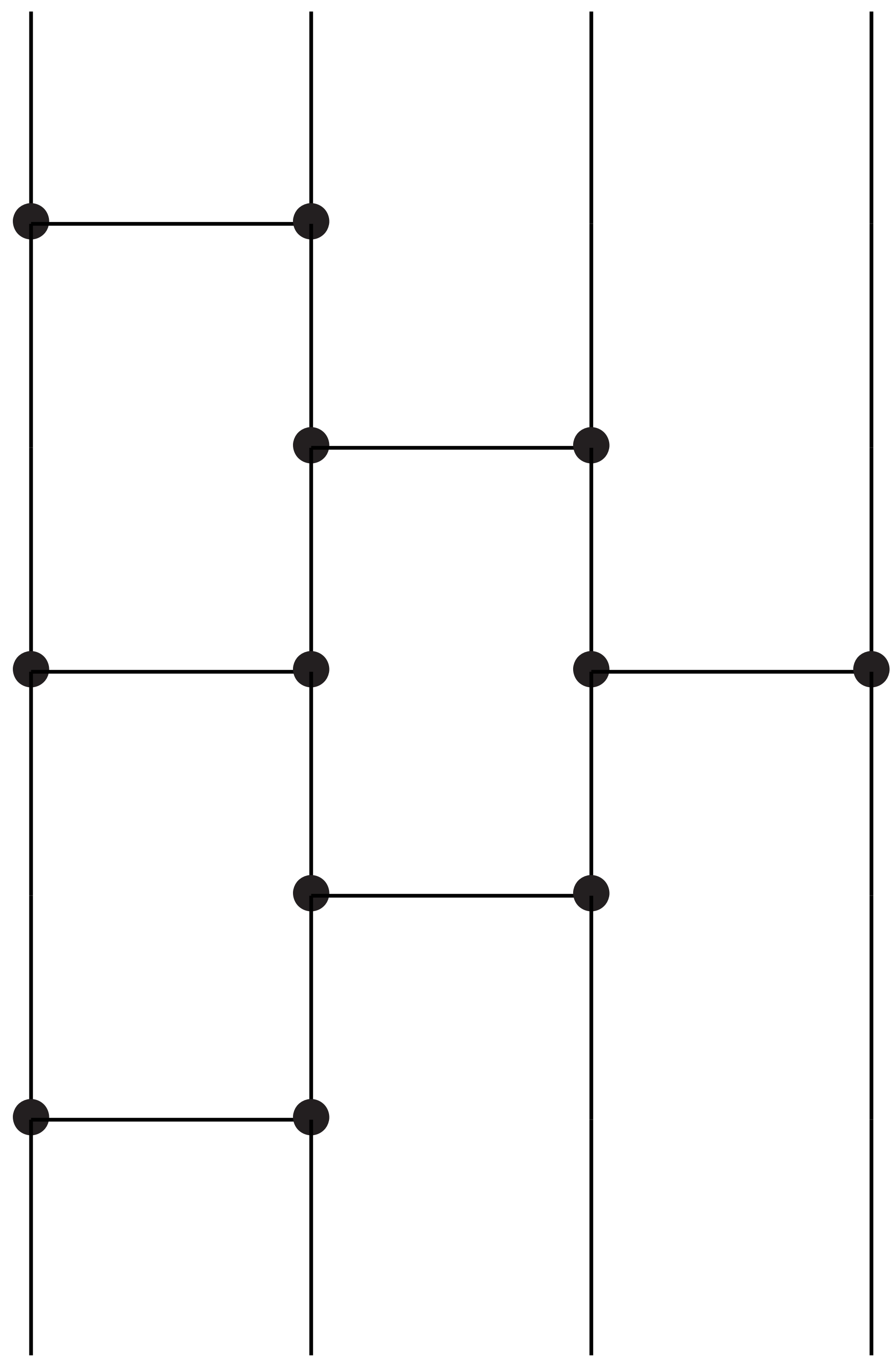


Positions of comparators in a sorting network are independent of the input.

Naturally constant-time.

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

Example of a sorting network:

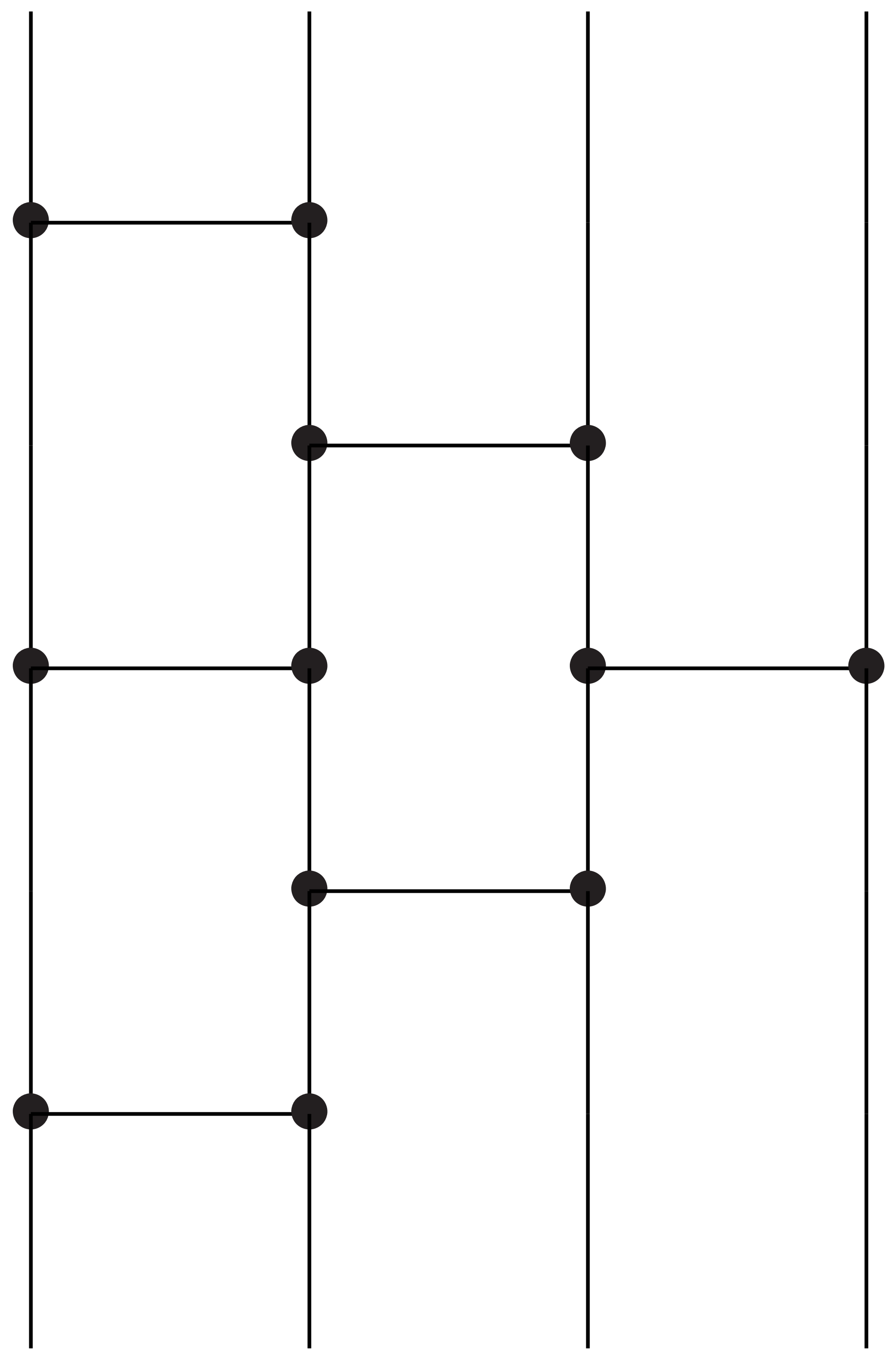


Positions of comparators in a sorting network are independent of the input.

Naturally constant-time.

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

Example of a sorting network:

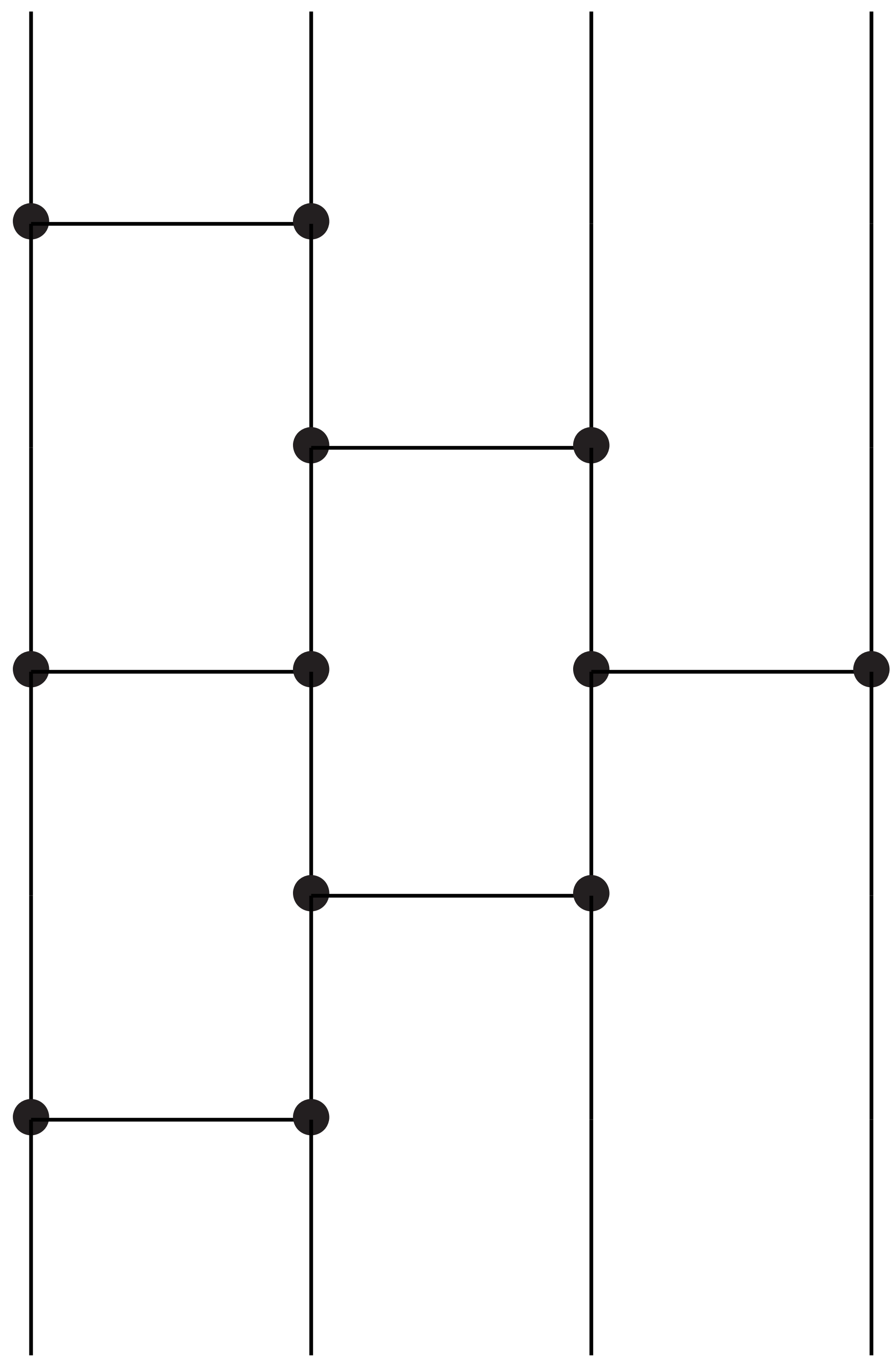


Positions of comparators in a sorting network are independent of the input. Naturally constant-time.

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

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

Example of a sorting network:



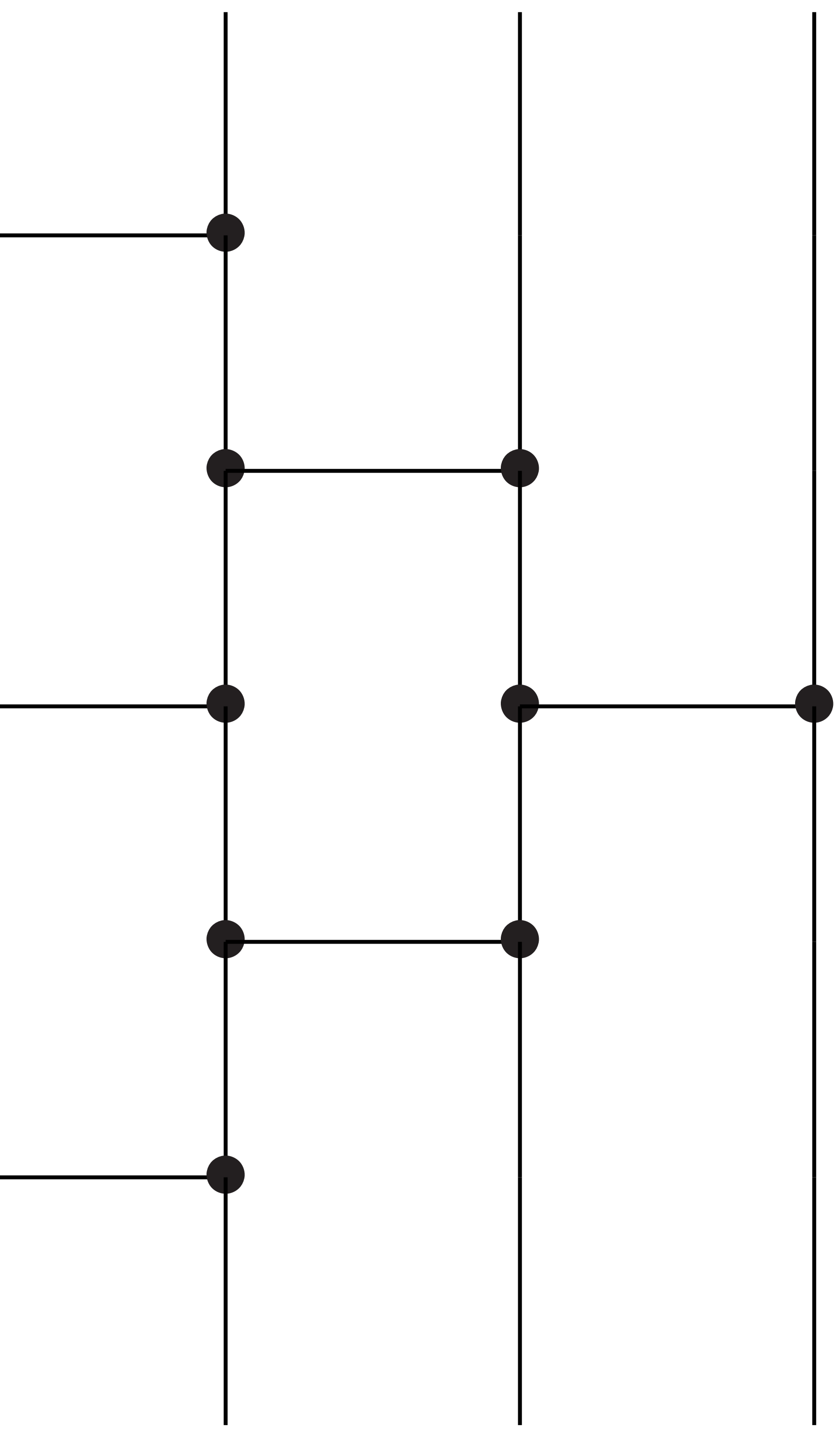
Positions of comparators in a sorting network are independent of the input. Naturally constant-time.

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.

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.

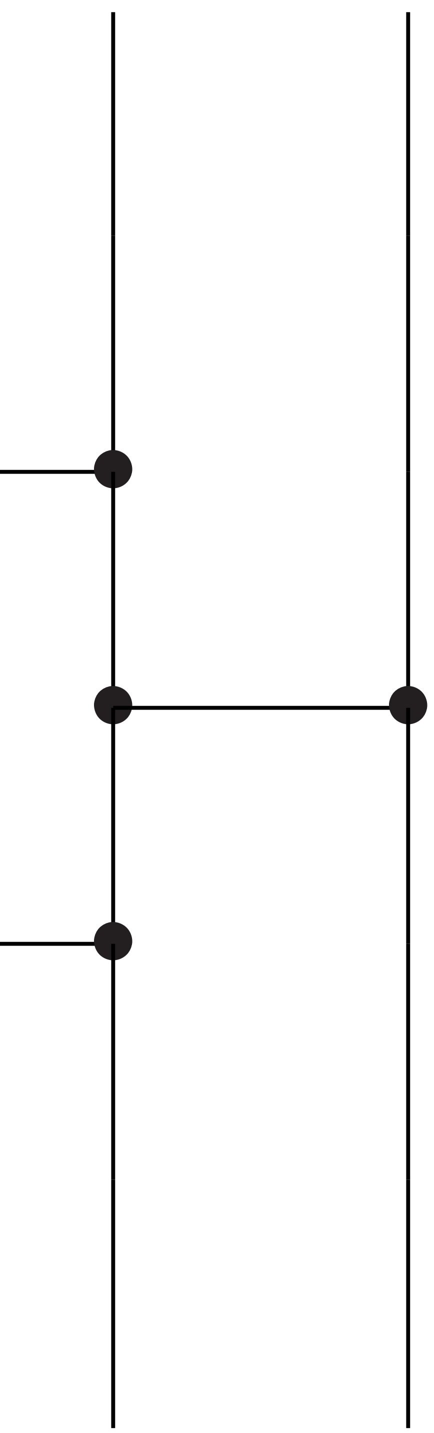
But $(n^2 - n)/2$ comparators produce complaints about performance as n increases.

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

    for
      fo



  }
}
```

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
  }
}
```

n

nputs.

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.

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

a serious issue in the
antum competition.
s evaluation criterion;
ke to stress this once
the forum that we'd
ke 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 n

$\approx n(\log_2$
Much fa

Warning
of Batch
require n
Also, Wi
networks
handling

arators
rk are
e input.
t-time.

omparators
ts about
increases.

issue in the
npetition.
on criterion;
s this once
n that we'd
nore 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 "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

s

ne

n;

e

d

orm-

; 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 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
l

New: djl
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

How mu
by refusi
quicksor

Cycles o
to sort n

26948 s
22812 h
17748 k
16980 i
12672 s

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

nparators.
bubble sort.

ther descriptions
ng networks
power of 2.
ays "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

How much speed
by refusing to use
quicksort, radix so

Cycles on Intel Ha
to sort $n = 768$ 32

26948 `stdsort` (v
22812 `herf` (varia
17748 `krasnov` (v
16980 `ipp 2019.5`
12672 `sid1607` (v

n of
ge",
of
rge"

t.

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

How much speed did we los
by refusing to use variable-ti
quicksort, radix sort, etc.?

Cycles on Intel Haswell CPU
to sort $n = 768$ 32-bit intege

26948 `stdsort` (variable-tim
22812 `herf` (variable-time)
17748 `krasnov` (variable-tim
16980 `ipp 2019.5` (variable
12672 `sid1607` (variable-tim

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

How much speed did we lose by refusing to use variable-time quicksort, radix sort, etc.?

Cycles on Intel Haswell CPU core to sort $n = 768$ 32-bit integers:

26948 `stdsort` (variable-time)
22812 `herf` (variable-time)
17748 `krasnov` (variable-time)
16980 `ipp 2019.5` (variable-time)
12672 `sid1607` (variable-time)

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

How much speed did we lose by refusing to use variable-time quicksort, radix sort, etc.?

Cycles on Intel Haswell CPU core to sort $n = 768$ 32-bit integers:

26948 `stdsort` (variable-time)
22812 `herf` (variable-time)
17748 `krasnov` (variable-time)
16980 `ipp 2019.5` (variable-time)
12672 `sid1607` (variable-time)
5964 `djbsort` (constant-time)

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

How much speed did we lose by refusing to use variable-time quicksort, radix sort, etc.?

Cycles on Intel Haswell CPU core to sort $n = 768$ 32-bit integers:

26948 `stdsort` (variable-time)
22812 `herf` (variable-time)
17748 `krasnov` (variable-time)
16980 `ipp 2019.5` (variable-time)
12672 `sid1607` (variable-time)
5964 `djbsort` (constant-time)

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

How much speed did we lose by refusing to use variable-time quicksort, radix sort, etc.?

Cycles on Intel Haswell CPU core to sort $n = 768$ 32-bit integers:

26948 `stdsort` (variable-time)
22812 `herf` (variable-time)
17748 `krasnov` (variable-time)
16980 `ipp 2019.5` (variable-time)
12672 `sid1607` (variable-time)
5964 `djbsort` (constant-time)

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

How much speed did we lose by refusing to use variable-time quicksort, radix sort, etc.?

Cycles on Intel Haswell CPU core to sort $n = 768$ 32-bit integers:

26948 `stdsort` (variable-time)
22812 `herf` (variable-time)
17748 `krasnov` (variable-time)
16980 `ipp 2019.5` (variable-time)
12672 `sid1607` (variable-time)
5964 `djbsort` (constant-time)

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

How much speed did we lose by refusing to use variable-time quicksort, radix sort, etc.?

Cycles on Intel Haswell CPU core to sort $n = 768$ 32-bit integers:

26948 `stdsort` (variable-time)
22812 `herf` (variable-time)
17748 `krasnov` (variable-time)
16980 `ipp 2019.5` (variable-time)
12672 `sid1607` (variable-time)
5964 `djbsort` (constant-time)

No slowdown. New speed records!

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

The slowdown for constant time

How much speed did we lose by refusing to use variable-time quicksort, radix sort, etc.?

Cycles on Intel Haswell CPU core to sort $n = 768$ 32-bit integers:

26948 `stdsort` (variable-time)
22812 `herf` (variable-time)
17748 `krasnov` (variable-time)
16980 `ipp 2019.5` (variable-time)
12672 `sid1607` (variable-time)
5964 `djbsort` (constant-time)

No slowdown. New speed records!

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

The slowdown for constant time

How much speed did we lose by refusing to use variable-time quicksort, radix sort, etc.?

Cycles on Intel Haswell CPU core to sort $n = 768$ 32-bit integers:

26948 `stdsort` (variable-time)
22812 `herf` (variable-time)
17748 `krasnov` (variable-time)
16980 `ipp 2019.5` (variable-time)
12672 `sid1607` (variable-time)
5964 `djbsort` (constant-time)

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

How much speed did we lose by refusing to use variable-time quicksort, radix sort, etc.?

Cycles on Intel Haswell CPU core to sort $n = 768$ 32-bit integers:

26948 `stdsort` (variable-time)
22812 `herf` (variable-time)
17748 `krasnov` (variable-time)
16980 `ipp 2019.5` (variable-time)
12672 `sid1607` (variable-time)
5964 `djbsort` (constant-time)

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

How much speed did we lose by refusing to use variable-time quicksort, radix sort, etc.?

Cycles on Intel Haswell CPU core to sort $n = 768$ 32-bit integers:

26948 `stdsort` (variable-time)
22812 `herf` (variable-time)
17748 `krasnov` (variable-time)
16980 `ipp 2019.5` (variable-time)
12672 `sid1607` (variable-time)
5964 `djbsort` (constant-time)

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

ch speed did we lose
ing to use variable-time
t, radix sort, etc.?

n Intel Haswell CPU core
$n = 768$ 32-bit integers:

`tdsort` (variable-time)
`erf` (variable-time)
`rasnov` (variable-time)
`pp 2019.5` (variable-time)
`id1607` (variable-time)
`jbsort` (constant-time)

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 v

Test the
random
decreasi

constant time

did we lose
variable-time
rt, etc.?

swell CPU core
2-bit integers:

variable-time)
ble-time)
variable-time)
5 (variable-time)
variable-time)
constant-time)

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

e

ime

J core

ers:

ne)

ne)

-time)

ne)

me)

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?

vn 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 worl

thm
hms?

n do
ers +
gers.

m a
er.

lower.

Verification

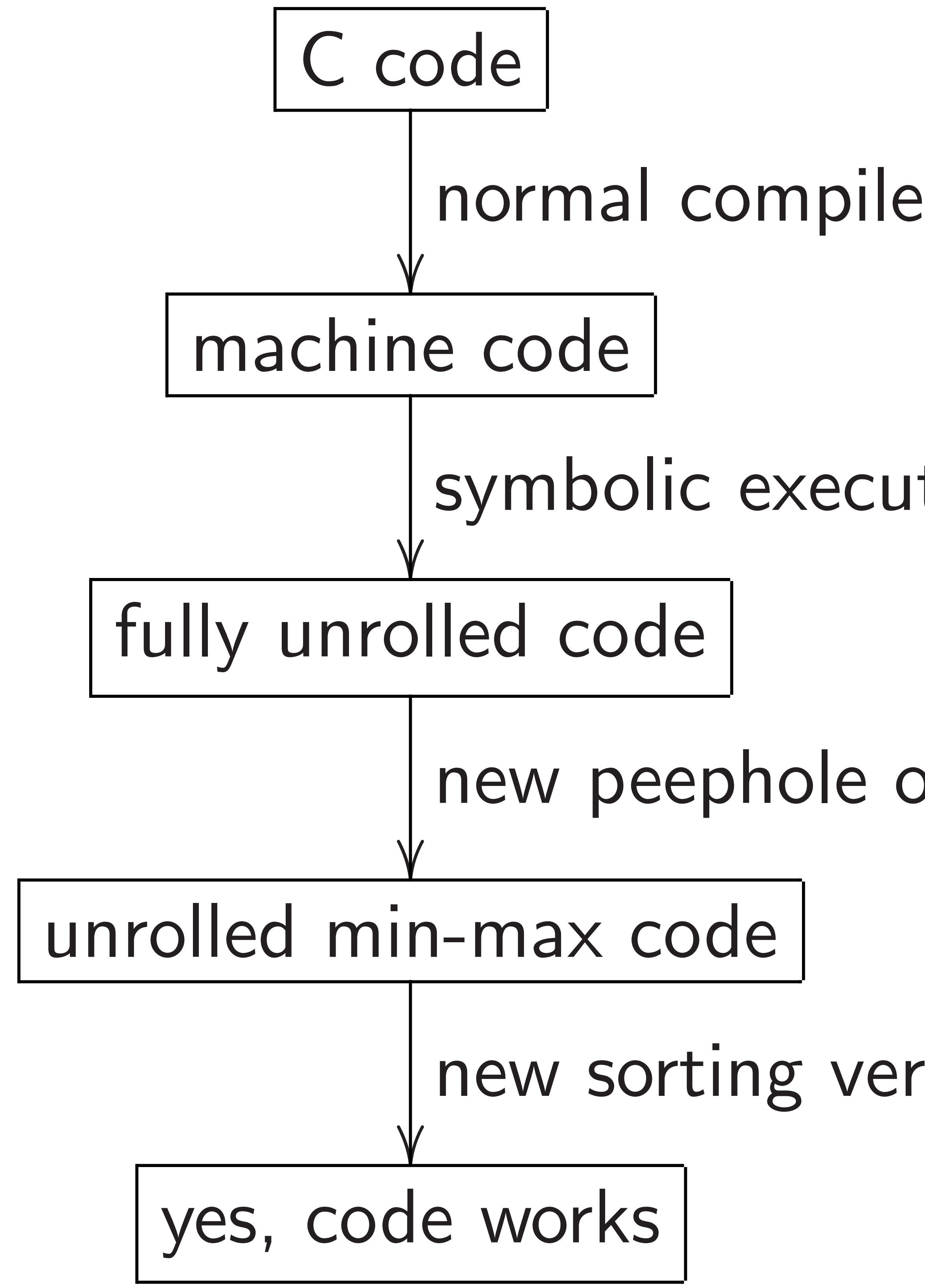
Sorting software is in the TCB.
Does it work correctly?

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):



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):

$$
\boxed{\text{C code}}
\xrightarrow{\text{normal compiler}}
\boxed{\text{machine code}}
\xrightarrow{\text{symbolic execution}}
\boxed{\text{fully unrolled code}}
\xrightarrow{\text{new peephole optimizer}}
\boxed{\text{unrolled min-max code}}
\xrightarrow{\text{new sorting verifier}}
\boxed{\text{yes, code works}}
$$

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 sev
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 several tiny n
eliminating byte sp
a few missing vect

CB.

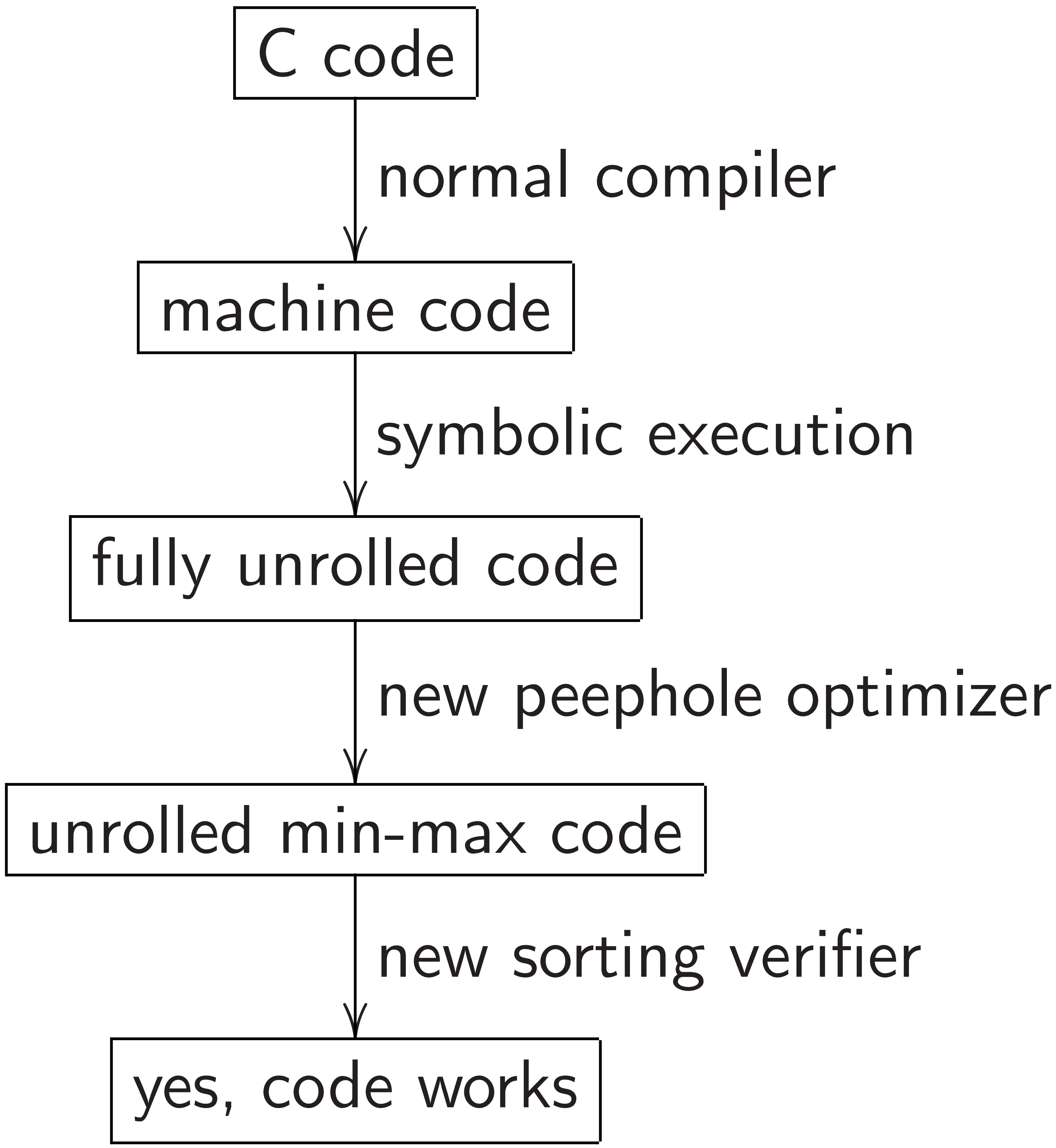
many
puts,
work.

uts

lems

y.

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



Symbolic execution:
use existing angr.io toolkit
with several tiny new patche
eliminating byte splitting, ac
a few missing vector instruct

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

$$\begin{tikzcd}
\boxed{\text{C code}} \arrow[d, "\text{normal compiler}"] \\
\boxed{\text{machine code}} \arrow[d, "\text{symbolic execution}"] \\
\boxed{\text{fully unrolled code}} \arrow[d, "\text{new peephole optimizer}"] \\
\boxed{\text{unrolled min-max code}} \arrow[d, "\text{new sorting verifier}"] \\
\boxed{\text{yes, code works}}
\end{tikzcd}$$

Symbolic execution:

use existing `angr.io` toolkit, with several 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.io` toolkit,
with several 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):

$$\begin{tikzcd}
\boxed{\text{C code}} \arrow[d, "\text{normal compiler}"] \\
\boxed{\text{machine code}} \arrow[d, "\text{symbolic execution}"] \\
\boxed{\text{fully unrolled code}} \arrow[d, "\text{new peephole optimizer}"] \\
\boxed{\text{unrolled min-max code}} \arrow[d, "\text{new sorting verifier}"] \\
\boxed{\text{yes, code works}}
\end{tikzcd}$$

Symbolic execution:
use existing `angr.io` toolkit, with several 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.io toolkit, with several 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
fast uin

sorting

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.io toolkit,
with several 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`
(verified fast `int3`
verified portable `i`
fast `uint32`, fast `i`

sorting.cr.yp.t

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

Web site shows ho
use the verificatio

Next release plann
verified ARM NEO

r

tion

ptimizer

ifier

Symbolic execution:
use existing angr.io toolkit, with several 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 fast `int32` on AVX verified portable `int32`, fast `uint32`, fast `float32`):

`sorting.cr.yp.to`

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.io` toolkit, with several 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 fast `int32` on AVX2, verified portable `int32`, fast `uint32`, fast `float32`):

`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.