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.
Sorting integer arrays: 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.

Trusted computing base (TCB)

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

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.
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.
From: Alice

Thank you for your submission. We received many interesting papers, and unfortunately your message appears genuine.

Bob assumes this message is something Alice actually sent.

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

Examples of attack strategies:
1. Attacker uses buffer overflow in a device driver to control Linux kernel on Alice’s laptop.

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.

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
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.

Examples of attack strategies:
1. Attacker uses buffer overflow in a device driver to control Linux kernel on Alice's laptop.
Bob’s laptop screen:
From: Alice
Thank you for your submission. We received many interesting papers, and unfortunately your...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.

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.
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?
Trusted computing base (TCB)

Portion of computer system responsible for enforcing users’ security policy.

Security policy for this talk:

Message is displayed on screen as “From: Alice” message is from Alice.

If TCB works correctly, 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?

Classic security strategy:

Rearchitect computer systems to have a much smaller TCB.
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?

Classic security strategy:

- Rearchitect computer systems to have a much smaller TCB.
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?

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.
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:

\[
\begin{array}{ccc}
\text{VM A} & \text{VM C} & \ldots \\
\text{Alice data} & \text{Charlie data} & \\
\end{array}
\]

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:

<table>
<thead>
<tr>
<th>VM A</th>
<th>VM C</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice data</td>
<td>Charlie data</td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>VM A</th>
<th>VM C</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice data</td>
<td>Charlie data</td>
<td></td>
</tr>
</tbody>
</table>

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.
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:

<table>
<thead>
<tr>
<th>VM A</th>
<th>VM C</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice data</td>
<td>Charlie data</td>
<td>...</td>
</tr>
</tbody>
</table>

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.

Alice’s message

\[ \cdot \cdot \cdot \]

k

\[ \cdot \cdot \cdot \]

\[ \cdot \cdot \cdot \]

untrusted network

\[ \cdot \cdot \cdot \]

\[ \cdot \cdot \cdot \]

authenticated message

\[ \cdot \cdot \cdot \]

\[ \cdot \cdot \cdot \]

Alice’s message

\[ \cdot \cdot \cdot \]

k

\[ \cdot \cdot \cdot \]

\[ \cdot \cdot \cdot \]
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:

<table>
<thead>
<tr>
<th>VM A</th>
<th>VM C</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice data</td>
<td>Charlie data</td>
<td>...</td>
</tr>
</tbody>
</table>

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.

Alice’s message
\[ \rightarrow \]
\[ \rightarrow \]
untrusted network
\[ \rightarrow \]
\[ \rightarrow \]
authenticated message
Alice’s message
Examples of attack strategies:
1. Attacker uses buffer overflow in a device driver to control Linux kernel on Alice's laptop.
2. Attacker uses buffer overflow in a web browser to control disk files on Bob's laptop.

Device driver is in the TCB.
Web browser is in the TCB.
CPU is in the TCB. Etc.
Massive TCB has many bugs, including many security holes.
Any hope of fixing this?

Classic security strategy:
Rearchitect computer systems to have a much smaller TCB.
Carefully audit the TCB.

e.g. Bob runs many VMs:

VM A
Alice data

VM C
Charlie data

TCB stops each VM from touching data in other VMs.
Browser in VM C isn’t in TCB. Can’t touch data in VM A, if TCB works correctly.

Alice also runs many VMs.

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

Cryptographic solution:
Message-authentication codes.

Alice’s message
cryptographically authenticated
untrusted network
authenticated message
authenticated message
Alice’s message
Classic security strategy:
Rearchitect computer systems to have a much smaller TCB.
Carefully audit the TCB.
e.g. Bob runs many VMs:

<table>
<thead>
<tr>
<th>VM A</th>
<th>VM C</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice data</td>
<td>Charlie data</td>
<td></td>
</tr>
</tbody>
</table>

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.

```
Alice’s message
↓
authenticated message
↓
untrusted network
↓
authenticated message
↓
Alice’s message
```

```
Alice’s message
k
←
authenticated message
↓
untrusted network
↓
authenticated message
↓
Alice’s message
k
```
Classic security strategy:
Rearchitect computer systems to have a much smaller TCB.
Carefully audit the TCB.
e.g. Bob runs many VMs:

<table>
<thead>
<tr>
<th>VM A</th>
<th>VM C</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice data</td>
<td>Charlie data</td>
<td>...</td>
</tr>
</tbody>
</table>

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.

Alice’s message

<table>
<thead>
<tr>
<th>authenticated message</th>
</tr>
</thead>
<tbody>
<tr>
<td>untrusted network</td>
</tr>
</tbody>
</table>

modified message

<table>
<thead>
<tr>
<th>“Alert: forgery!”</th>
</tr>
</thead>
</table>

k

k
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 | ... |

... stops each VM from accessing 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.

Alice’s message

\[ k \]

\[ \text{authenticated message} \]

untrusted network

\[ \text{modified message} \]

“Alert: forgery!”

Alice and Bob need to share the same secret \( k \).

What if attacker was spying on their communication of \( k \)?
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.

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 $k$?

Alice’s message
\[ \downarrow \]
authenticated message
\[ \downarrow \]
untrusted network
\[ \downarrow \]
modified message
\[ \downarrow \]
“Alert: forgery!”
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.

Alice’s message

authenticated message

untrusted network

modified message

“Alert: forgery!”

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.

```
Alice's message
↓
authenticated message
untrusted network
↓
modified message
↓
"Alert: forgery!"
```

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

What if attacker was spying on their communication of $k$?

Solution 1: Public-key encryption.

```
k
private key a
↓
ciphertext
network
↓
ciphertext
network
public key aG
←
k
```

```
private key a
↓
ciphertext
network
↓
ciphertext
network
public key aG
←
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.

Solution 2: Public-key signatures.
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.

Solution 2: Public-key signatures.
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.

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.

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

Constant-time software
Large portion of CPU hardware:
optimizations depending on addressed
Consider data caching,
instruction caching,
parallel cache banks,
store-to-load forwarding,
branch prediction, etc.
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.

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.
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.

$$
\begin{array}{c}
m \\
\downarrow \\
signed \text{ message} \\
\downarrow \\
network \\
\downarrow \\
signed \text{ message} \\
\downarrow \\
aG \\
\downarrow \\
network \\
\downarrow \\
m \\
\end{array}
$$

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.

Solution 2: Public-key signatures.

$$
\begin{array}{c}
m \\
\downarrow \\
signed \text{ message} \\
\downarrow \\
network \\
\downarrow \\
signed \text{ message} \\
\downarrow \\
aG \\
\downarrow \\
aG \\
\downarrow \\
m \\
\end{array}
$$

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.
Solution 2: Public-key signatures.

\[
\begin{align*}
&m &\rightarrow &\text{signed message} \\
&\text{signed message} &\rightarrow &\text{network} \\
&\text{network} &\rightarrow &\text{signed message} \\
&\text{signed message} &\rightarrow &aG \\
&\text{network} &\rightarrow &aG \\
&\text{network} &\rightarrow &aG \\
&aG &\rightarrow &m
\end{align*}
\]

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.
Solution 2: Public-key signatures.

- Signed message
- Network
- $m$  
- $aG$
- $a$
- $a$
- $m$
- $k$
- $a$
- $G$
- $a$
- $m$
- $G$

The shared secret $k$ is no more shared, 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–Osterlund–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.
Solution 2: Public-key signatures.

\[
\text{signed message} \rightarrow \text{network} \rightarrow aG \rightarrow \text{network} \rightarrow \text{signed message}
\]

\[
\text{signed message} \rightarrow \text{network} \rightarrow aG \rightarrow \text{network} \rightarrow \text{signed message}
\]

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.

---

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.
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.
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 Schalk–Milburn–Osterlund–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.

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)
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–Osterlund–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.

The “constant-time” solution:
Don’t give any secrets to this portion of the CPU.

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–
¨
Osterlund–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.

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)
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.
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.
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.

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 $aG$).
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. This is great! 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.

Case study: Constant-time sorting
Serious risk within 10 years: Attacker has quantum computer breaking today’s most popular public-key crypto (e.g., finding $aG$).
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.

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.

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.
The “constant-time” solution:

Don’t give any secrets to this portion of the CPU. (1987 Goldreich, 1990 Ostrovsky: oblivious RAM; 2004 Bernstein: specific for better speed)

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.

How to sort secret data without any secret addresses?
The "constant-time" solution:

Don't give any secrets to this portion of the CPU.


This made this portion correct, but kept secrets. Much easier.

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.

How to sort secret data without any secret addresses?
The "constant-time" solution:
Don't give any secrets
to this portion of the CPU.
(1987 Goldreich, 1990 Ostrovsky:
Oblivious RAM; 2004 Bernstein:
domain-specific for better speed)

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

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

Case study: Constant-time sorting
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 $aG$).

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.
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; finding $a \cdot G$).

Hundreds of people submit 69 complete proposals to international competition for post-quantum crypto standards. One in some submissions:
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.

Foundation of solution:
a comparator sorting 2 integers.

$x$

$\min\{x, y\}$ $\max\{x, y\}$

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

10 years:
Quantum computer
most popular
(RSA and ECC; then \( aG \)).

Hundreds of people
Submit proposals
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.

Foundation of solution:
a comparator sorting 2 integers.

\[
\begin{align*}
\text{min} \{x, y\} & \quad \text{min} \{x, y\} \\
\text{max} \{x, y\} & \quad \text{max} \{x, y\}
\end{align*}
\]

Easy constant-time exercise in C.
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.

\[
\begin{align*}
\min \{x, y\} & \quad \max \{x, y\} \\
\end{align*}
\]

Easy constant-time exercise in C.
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.

\[
\begin{align*}
x & \quad y \\
\min\{x, y\} & \quad \max\{x, y\}
\end{align*}
\]

Easy constant-time exercise in C. 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.

\[
\begin{align*}
\min\{x, y\} & \quad & \max\{x, y\}
\end{align*}
\]

Easy constant-time exercise in C.

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.

\[
\begin{align*}
\min\{x, y\} & \quad \text{x y} \\
\max\{x, y\} & \quad \text{• •}
\end{align*}
\]

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:

\[
\begin{align*}
\text{• •} & \quad \text{• • • •}
\end{align*}
\]
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.

\[ \min\{x, y\} \quad \max\{x, y\} \]

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:
Foundation of solution:
a comparator sorting 2 integers.

\[
\begin{align*}
\min\{x, y\} & \quad \max\{x, y\} \\
x & \quad y
\end{align*}
\]

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:
Foundation of solution:

A comparator sorting 2 integers.

\[
\begin{align*}
\min\{x; y\} & \quad \max\{x; y\}
\end{align*}
\]

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:

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

Naturally constant-time.
Foundation of solution: a comparator sorting 2 integers.

\[
\begin{align*}
&\text{min} \{ x; y \} \\
&\text{max} \{ x; y \}
\end{align*}
\]

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:

\[
\begin{array}{c}
  \text{Positions of comparators in a sorting network are independent of the input.}
  \text{Naturally constant-time.}
\end{array}
\]
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:

![Sorting Network Diagram]

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

But \( \frac{n^2 - n}{2} \) 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.
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.

```c
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);
  }
}
```
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.

```c
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);
}
```
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.

```c
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);
  }
}
```
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.

```c
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);
    }
}
```
Positions of comparators in a sorting network are independent of the input. Naturally constant-time.

\[ \left( \frac{n}{2} - \frac{n}{2} \right) / 2 \] 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.

```c
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.

\[ n \log_2 n \approx n \log_2 n \]

\[ 2 \] 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."
Positions of comparators in a sorting network are independent of the input. Naturally constant-time. But $(n^2 = n)$ 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.

---

```c
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."
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.

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 network require \(n\) to be a power of 2.

Also, Wikipedia says “Sorting networks ... are not capable of handling arbitrarily large inputs."
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.

≈ $n(\log_2 n)^2/4$ 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.”
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.

\(\approx n \left(\log_2 n\right)^2/4\) 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.”
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 >>= 1) {
        for (i = 0; i < n - p; ++i)
            if (!(i & p))
                minmax(x + i, x + i + p);
        for (q = t; ; q >>= 1)
            for (i = 0; i < n - q; ++i)
                if (!(i & p))
                    minmax(x + i + p, x + i + q);
    }
}

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

$\approx n (\log_2 n)^2 / 4$ comparators.

Much faster than bubble sort.

Warning: many other descriptions of Batcher’s sorting networks require $n$ to be a power of 2.

Also, Wikipedia says “Sorting networks . . . are not capable of handling arbitrarily large inputs.”

This constant-time sorting code vectorization (for Haswell)

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

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

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

\[ \approx n (\log_2 n)^2 / 4 \] comparators.

Much faster than bubble sort.

Warning: many other descriptions of Batcher’s sorting networks require \( n \) to be a power of 2.

Also, Wikipedia says “Sorting networks ... are not capable of handling arbitrarily large inputs.”

This constant-time sorting code vectorization (for Haswell)

Constant-time sorting code included in 2017 Bernstein–Chuengsatiansupa–Lange–van Vredendaal “NTRU Prime” software release revamped for higher speed

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

\[ \approx n \left( \log_2 n \right)^2 / 4 \] comparators.

Much faster than bubble sort.

Warning: many other descriptions of Batcher’s sorting networks require \( n \) to be a power of 2. Also, Wikipedia says “Sorting networks . . . are not capable of handling arbitrarily large inputs.”
This constant-time sorting code
vectorization (for Haswell)

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

revamped for higher speed

New: djbsort
constant-time sorting code

The slowdown for constant time sorting
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:

- stdsort (variable-time): 26948 cycles
- herf (variable-time): 22812 cycles
- krasnov (variable-time): 17748 cycles
- ipp 2019.5 (variable-time): 16980 cycles
- sid1607 (variable-time): 12672 cycles
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
can handle arbitrarily large inputs.

The slowdown for constant time

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

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 to sort $n = 768$ 32-bit integers:

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdsort</td>
<td>26948</td>
</tr>
<tr>
<td>herf</td>
<td>22812</td>
</tr>
<tr>
<td>krasnov</td>
<td>17748</td>
</tr>
<tr>
<td>ipp 2019.5</td>
<td>16980</td>
</tr>
<tr>
<td>sid1607</td>
<td>12672</td>
</tr>
<tr>
<td>djbsort</td>
<td>5964</td>
</tr>
</tbody>
</table>

No slowdown. New speed records!
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.?

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cycles on Intel Haswell CPU core</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdsort (variable-time)</td>
<td>26948</td>
</tr>
<tr>
<td>herf (variable-time)</td>
<td>22812</td>
</tr>
<tr>
<td>krasnov (variable-time)</td>
<td>17748</td>
</tr>
<tr>
<td>ipp 2019.5 (variable-time)</td>
<td>16980</td>
</tr>
<tr>
<td>sid1607 (variable-time)</td>
<td>12672</td>
</tr>
<tr>
<td>djbsort (constant-time)</td>
<td>5964</td>
</tr>
</tbody>
</table>

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?
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 $\text{stdsort}$ (variable-time)
- 22812 $\text{herf}$ (variable-time)
- 17748 $\text{krasnov}$ (variable-time)
- 16980 $\text{ipp 2019.5}$ (variable-time)
- 12672 $\text{sid1607}$ (variable-time)
- 5964 $\text{djbsort}$ (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.
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:

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

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.

Verification
Sorting software is in the TCB.
Does it work correctly?
Test the sorting software on many
random inputs, increasing inputs,
and decreasing inputs. Seems to work.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdsort</td>
<td>26948</td>
</tr>
<tr>
<td>herf</td>
<td>22812</td>
</tr>
<tr>
<td>krasnov</td>
<td>17748</td>
</tr>
<tr>
<td>ipp 2019.5</td>
<td>16980</td>
</tr>
<tr>
<td>sid1607</td>
<td>12672</td>
</tr>
<tr>
<td>djbsort</td>
<td>5964</td>
</tr>
</tbody>
</table>

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.

Verification

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

Test the sorting software on many random inputs, increasing inputs, decreasing inputs.
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:

<table>
<thead>
<tr>
<th></th>
<th>stdsort (variable-time)</th>
<th>herf (variable-time)</th>
<th>krasnov (variable-time)</th>
<th>ipp 2019.5 (variable-time)</th>
<th>sid1607 (variable-time)</th>
<th>djbsort (constant-time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>26948</td>
<td>22812</td>
<td>17748</td>
<td>16980</td>
<td>12672</td>
<td>5964</td>
</tr>
</tbody>
</table>

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.
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.
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 $8 \text{ min ops on 32-bit integers} + 8 \text{ max ops on 32-bit integers}$. 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.

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
How can an $n^2$ 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.

C code
normal compiler
↓ ↓ 
machine code
symbolic execution
↓ ↓ 
fully unrolled code
new peephole optimizer
↓ ↓ 
unrolled min-max code
new sorting verifier
↓ ↓ 
yes, code works
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" operations on 32-bit integers + 8 "max" operations 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.

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

\[
\begin{align*}
&\text{C code} \\
&\downarrow\downarrow \\
&\text{normal compiler} \\
&\downarrow\downarrow \\
&\text{machine code} \\
&\downarrow\downarrow \\
&\text{symbolic execution} \\
&\downarrow\downarrow \\
&\text{fully unrolled code} \\
&\downarrow\downarrow \\
&\text{new peephole optimizer} \\
&\downarrow\downarrow \\
&\text{unrolled min-max code} \\
&\downarrow\downarrow \\
&\text{new sorting verifier} \\
&\downarrow\downarrow \\
&\text{yes, code works}
\end{align*}
\]
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.
Verification

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

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

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

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

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

```
C code
  ↓ normal compiler
  ↓ machine code
  ↓ symbolic execution
  ↓ fully unrolled code
  ↓ new peephole optimizer
  ↓ unrolled min-max code
  ↓ new sorting verifier
  ↓ yes, code works
```

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

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.
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.

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
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.
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):

- **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):

1. C code
2. normal compiler
3. machine code
4. symbolic execution
5. fully unrolled code
6. new peephole optimizer
7. unrolled min-max code
8. new sorting verifier
9. 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.

Current djbsort release (verified fast int32 on AVX2, verified portable int32, fast uint32, fast float32):

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

C code
\[\rightarrow\] normal compiler
\[\rightarrow\] machine code
\[\rightarrow\] symbolic execution
\[\rightarrow\] fully unrolled code
\[\rightarrow\] new peephole optimizer
\[\rightarrow\] unrolled min-max code
\[\rightarrow\] new sorting verifier
\[\rightarrow\] 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.

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 ... tools.
Web site shows how to
use the verification tools.
Next release planned:
verified ARM NEON code.

Current djbsort release
(verified fast int32 on AVX2,
verified portable int32 ,
fast uint32 , fast float32 ... 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.
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.