Sorting integer arrays: security, speed, and verification

D. J. Bernstein

Bob’s laptop screen:

From: Alice

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

Bob assumes this message is something Alice actually sent.

But today’s “security” systems fail to guarantee this property. Attacker could have modified or forged the message.
Sorting integer arrays: security, speed, and verification

D. J. Bernstein

Bob's laptop screen:

From: Alice

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

Bob assumes this message is something Alice actually sent.

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

Trusted computing base (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 laptop screen:

From: Alice

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

Bob assumes this message is something Alice actually sent.

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

Trusted computing base (TCB)

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

Security policy for this talk: If message is displayed on Bob’s screen as “From: Alice” then message is from Alice.
Bob’s laptop screen:

```
From: Alice

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

Bob assumes this message is something Alice actually sent.

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

---

**Trusted computing base (TCB)**

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

Security policy for this talk:

If message is displayed on Bob’s screen as “From: Alice” then message is from Alice.

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

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

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

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

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

"Security" systems today fail to guarantee this property. Attacker could have modified or forged the message.

Trusted computing base (TCB)
TCB: portion of computer system that is responsible for enforcing the users' security policy.

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

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

Examples of attack strategies:
1. Attacker uses buffer overflow in a device driver to control Linux kernel on Alice's laptop.
Trusted computing base (TCB)

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

Security policy for this talk:
If message is displayed on Bob’s screen as “From: Alice” then message is from Alice.

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

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

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

Security policy for this talk:
If message is displayed on Bob’s screen as “From: Alice” then message is from Alice.

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

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

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

Security policy for this talk:
If message is displayed on Bob’s screen as “From: Alice” then message is from Alice.

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

Examples of attack strategies:

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

2. Attacker uses buffer overflow in a web browser to control disk files on Bob’s laptop.
Trusted computing base (TCB)

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

Security policy for this talk:
If message is displayed on Bob’s screen as “From: Alice” then message is from Alice.

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

Examples of attack strategies:

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

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

Device driver is in the TCB.
Web browser is in the TCB.
CPU is in the TCB. Etc.
Trusted computing base (TCB)

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

Security policy for this talk:
If message is displayed on Bob’s screen as “From: Alice” then message is from Alice.

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

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

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

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

Computer system for enforcing policy.

This talk:
Play on From: Alice" from Alice.

Directly, guaranteed no matter what 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 system to have a much smaller TCB.
Trusted computing base (TCB)

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

Security policy for this talk:
If message is displayed on Bob's screen as "From: Alice" then message is from Alice.
If TCB works correctly, then message is guaranteed to be from Alice, no matter what the rest of the system does.

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

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

Massive TCB has many bugs, including many security holes.

Any hope of fixing this?

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:

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

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.

Alic...
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>
</table>
| 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.
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
untrusted network
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.
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.

Alice’s message

```
authenticated message
```

untrusted network

```
modified message
```

“Alert: forgery!”
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>
</tr>
</thead>
<tbody>
<tr>
<td>Alice data</td>
<td>Charlie data</td>
</tr>
</tbody>
</table>

... 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.

Important for Alice and Bob 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.

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
  - 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?
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

\[ \rightarrow \]

authenticated message

\[ \rightarrow \]

untrusted network

modified message

\[ \rightarrow \]

“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.

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.
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.
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.
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.

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.

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.

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.

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 1: Public-key encryption.

Solution 2: Public-key signatures.

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 $k$?

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.

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

No more shared secret \( k \) but Alice still has secret \( a \).

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

**Constant-time software**

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

Consider data caching, instruction caching, parallel cache banks, store-to-load forwarding, branch prediction, etc.
Solution 2: Public-key signatures.

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

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

---

**Constant-time software**

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

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

Many attacks (e.g. TLBleed from 2018 Gras–Razavi–Bos–Giuffrida) show that this portion of the CPU has trouble keeping secrets.
Solution 2:
Public-key signatures.

\[ m \]
\[ \downarrow \downarrow \]
\[ a \]
\[ \downarrow \downarrow \]
\[ \leftarrow \leftarrow \]
signed message
network
\[ \downarrow \downarrow \]
\[ aG \]
network
\[ \downarrow \downarrow \]
\[ \leftarrow \leftarrow \]
signed message
network

No more shared secret \( k \)
but Alice still has secret \( a \).

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

Constant-time software
Large portion of CPU hardware:
optimizations depending on
addresses of memory locations.

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

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

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.

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

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

Typical literature on this topic:

Understand this portion of CPU.

But details are often proprietary, not exposed to security review.

Try to push attacks further.

This becomes very complicated.

Tweak the attacked software to try to stop the known attacks.
Solution 2: Public-key signatures.

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

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

Typical literature on this topic:

Understand this portion of CPU.

But details are often proprietary, not exposed to security review.

Try to push attacks further. This becomes very complicated.

Tweak the attacked software to try to stop the known attacks.
Constant-time software

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

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

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

Typical literature on this topic:

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

Try to push attacks further. This becomes very complicated.

Tweak the attacked software to try to stop the known attacks.
Constant-time software

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

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

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

Typical literature on this topic:
Understand this portion of CPU. But details are often proprietary, not exposed to security review.

Try to push attacks further. This becomes very complicated.

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

For researchers: This is great!
Constant-time software

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

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

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

Typical literature on this topic:
Understand this portion of CPU. But details are often proprietary, not exposed to security review.

Try to push attacks further. This becomes very complicated.

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

For researchers: This is great!
For auditors: This is a nightmare.
Many years of security failures. No confidence in future security.
Constant-time software

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

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

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

Typical literature on this topic:
Understand this portion of CPU.
But details are often proprietary,
not exposed to security review.

Try to push attacks further.
This becomes very complicated.

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

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

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:
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.
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. Details are often proprietary, not exposed to security review. Try to push attacks further. Comes very complicated. Tweak the attacked software to stop the known attacks. Researchers: This is great! Auditors: This is a nightmare. 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 a given 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. Tweaked software and known attacks. This is great! 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 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 $aG$).
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.
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)

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.


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?
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.
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 $aG$).
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.

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.
Case study: Constant-time sorting

10 years:
Quantum computer
Serious risk within 10 years:
Attacker has quantum computer
breaking today's most popular
crypto (RSA and ECC; even $aG$).

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
colorful sorting algorithm.”

Some versions of radix sort
avoid secret branches.
But data addresses in radix sort
still depend on secrets.

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

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 $\mathbf{G}$).

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.

Foundation of solution:

comparator sorting 2 integers.

$x$ $y$

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

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

Even easier exercise in asm.
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.

\[
x \quad y
\]

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

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

Even easier exercise in asm.
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:
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.

Combine comparators into a sorting network for more inputs.

Example of a sorting network:
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.

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

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

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

Easy constant-time exercise in C.

Warning: C standard allows compiler to screw this up.

Even easier exercise in asm.

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

Example of a sorting network:
Foundation of solution:

- Comparator sorting 2 integers.
- \( x \) \( y \)
- \( \min \{ x; y \} \) \( \max \{ x; y \} \)

Easy constant-time exercise in C.

Warning: C standard allows compiler to screw this up.

Even easier exercise in asm.

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.
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:

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:

```
\begin{center}
\begin{tikzpicture}
  \begin{scope}[every node/.style={circle,fill,inner sep=0.5ex}]
    \node (a1) at (0,0) {};
    \node (a2) at (0,1) {};
    \node (b1) at (1,0) {};
    \node (b2) at (1,1) {};
    \node (c1) at (2,0) {};
    \node (c2) at (2,1) {};
    \node (d1) at (3,0) {};
    \node (d2) at (3,1) {};
    \end{scope}
  \begin{scope}[every path/.style={ultra thick,draw=black}]
    \path (a1) edge (a2);
    \path (a2) edge (b1);
    \path (a2) edge (b2);
    \path (b1) edge (c1);
    \path (b2) edge (c2);
    \path (c2) edge (d1);
    \path (c2) edge (d2);
  \end{scope}
\end{tikzpicture}
\end{center}
```

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

```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.

```
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. 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);
    }
}
```

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

\(\approx n\left(\log_2 n\right)\) 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 \((n^2 - n) = 2n\) 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.

\[\approx n \left(\log_2 n\right)^2 / 4\] 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.”
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);
    }
}
```

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.

\( \approx n(\log_2 n)^2/4 \) 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.”
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.”
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.”
void int32_sort(int32 *x, int64 n)
{ int64 t, p, q, i;
if (n < 2) return;
t = 1;
while (t < n - t) t += t;
for (p = t; p > 0; p >>= 1) {
for (i = 0; i < n - p; ++i)
if (!(i & p))
minmax(x + i, x + i + p);
for (q = t; q > p; q >>= 1)
for (i = 0; i < n - q; ++i)
if (!(i & p))
minmax(x + i + p, x + i + q);
}
}

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

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

Much faster than bubble sort.

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

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

This constant-time sorting code vectorization (for Haswell)

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

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

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

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

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

revamped for higher speed

New: “djbsort”
constant-time sorting code

The slowdown for constant time sorting is large.
Massive fast-sorting literature.
2015 Gueron–Krasnov: AVX and AVX2 (Haswell) optimization of quicksort. For 32-bit integers:
\( \approx 45 \) cycles/byte for \( n \approx 2^{10} \),
\( \approx 55 \) cycles/byte for \( n \approx 2^{20} \).
Slower than “the radix sort implemented in IPP, which is the fastest in-memory sort we are aware of”: 32, 40 cycles/byte.

IPP: Intel’s Integrated Performance Primitives library.
This constant-time sorting code vectorization (for Haswell)

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

revamped for higher speed

New: “djbsort”

constant-time sorting code

The slowdown for constant time

Massive fast-sorting literature.

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

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

IPP: Intel’s Integrated Performance Primitives library.
This constant-time sorting code

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

revamped for higher speed

New: “djbsort”
constant-time sorting code

The slowdown for constant time
Massive fast-sorting literature.
2015 Gueron–Krasnov: AVX and AVX2 (Haswell) optimization of quicksort. For 32-bit integers:
≈45 cycles/byte for $n \approx 2^{10}$,
≈55 cycles/byte for $n \approx 2^{20}$.
Slower than “the radix sort implemented of IPP, which is the fastest in-memory sort we are aware of”: 32, 40 cycles/byte.

IPP: Intel’s Integrated Performance Primitives library.
This constant-time sorting code

vectorization (for Haswell)

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

revamped for higher speed

New: “djbsort” constant-time sorting code

The slowdown for constant time

Massive fast-sorting literature.

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

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

IPP: Intel’s Integrated Performance Primitives library.
Massive fast-sorting literature.

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

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

IPP: Intel’s Integrated Performance Primitives library.
This constant-time sorting code vectorization (for Haswell)

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

New: "djbsort" constant-time sorting code

The slowdown for constant time

Massive fast-sorting literature.

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

\[ \approx 45 \text{ cycles/byte for } n \approx 2^{10}, \]

\[ \approx 55 \text{ cycles/byte for } n \approx 2^{20}. \]

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

IPP: Intel’s Integrated Performance Primitives library.
The slowdown for constant time
Massive fast-sorting literature.
2015 Gueron–Krasnov: AVX and
AVX2 (Haswell) optimization of
quicksort. For 32-bit integers:
≈45 cycles/byte for \( n \approx 2^{10} \),
≈55 cycles/byte for \( n \approx 2^{20} \).
Slower than “the radix sort
implemented of IPP, which is
the fastest in-memory sort we are
aware of”: 32, 40 cycles/byte.
IPP: Intel’s Integrated
Performance Primitives library.
The slowdown for constant time
Massive fast-sorting literature.
2015 Gueron–Krasnov: AVX and
AVX2 (Haswell) optimization of
quicksort. For 32-bit integers:
\[ \approx 45 \text{ cycles/byte for } n \approx 2^{10}, \]
\[ \approx 55 \text{ cycles/byte for } n \approx 2^{20}. \]
Slower than “the radix sort
implemented of IPP, which is
the fastest in-memory sort we are
aware of”: 32, 40 cycles/byte.
IPP: Intel’s Integrated
Performance Primitives library.

Constant-time results,
again on Haswell CPU core:
The slowdown for constant time 
Massive fast-sorting literature.

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

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

IPP: Intel’s Integrated 
Performance Primitives library.

Constant-time results, 
again on Haswell CPU core:

2017 BCLvV:
6.5 cycles/byte for \( n \approx 2^{10} \),
33 cycles/byte for \( n \approx 2^{20} \).
The slowdown for constant time

Massive fast-sorting literature.

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

\[ \approx 45 \text{ cycles/byte for } n \approx 2^{10}, \]
\[ \approx 55 \text{ cycles/byte for } n \approx 2^{20}. \]

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

IPP: Intel’s Integrated Performance Primitives library.

Constant-time results, again on Haswell CPU core:

2017 BCLvV:

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

2018 djbsort:

2.5 cycles/byte for \( n \approx 2^{10}, \)
15.5 cycles/byte for \( n \approx 2^{20}. \)
The slowdown for constant time
Massive fast-sorting literature.
2015 Gueron–Krasnov: AVX and
AVX2 (Haswell) optimization of
quicksort. For 32-bit integers:
≈ 45 cycles/byte for $n \approx 2^{10}$,
≈ 55 cycles/byte for $n \approx 2^{20}$.
Slower than “the radix sort
implemented of IPP, which is
the fastest in-memory sort we are
aware of”: 32, 40 cycles/byte.
IPP: Intel’s Integrated
Performance Primitives library.

Constant-time results,
again on Haswell CPU core:

2017 BCLvV:
6.5 cycles/byte for $n \approx 2^{10}$,
33 cycles/byte for $n \approx 2^{20}$.

2018 djbsort:
2.5 cycles/byte for $n \approx 2^{10}$,
15.5 cycles/byte for $n \approx 2^{20}$.
No slowdown. New speed records!
The slowdown for constant time
Massive fast-sorting literature.

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

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

IPP: Intel’s Integrated Performance Primitives library.

Constant-time results, again on Haswell CPU core:

2017 BCLvV:
\[ 6.5 \text{ cycles/byte for } n \approx 2^{10}, \]
\[ 33 \text{ cycles/byte for } n \approx 2^{20}. \]

2018 djb sort:
\[ 2.5 \text{ cycles/byte for } n \approx 2^{10}, \]
\[ 15.5 \text{ cycles/byte for } n \approx 2^{20}. \]

No slowdown. New speed records!

Warning: Comparison for \( n \approx 2^{20} \) involves microarchitecture details beyond Haswell core. Should measure all code on same CPU.
The slowdown for constant time fast-sorting literature.

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

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

IPP: Intel’s Integrated Performance Primitives library.

Constant-time results, again on Haswell CPU core:

2017 BCLvV:
6.5 cycles/byte for \( n \approx 2^{10} \),
33 cycles/byte for \( n \approx 2^{20} \).

2018 djbsort:
2.5 cycles/byte for \( n \approx 2^{10} \),
15.5 cycles/byte for \( n \approx 2^{20} \).

No slowdown. New speed records!

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

How can an \( n (\log n)^2 \) algorithm beat standard \( n \log n \) algorithms?
The slowdown for constant time sorting literature.

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

\[ \approx 45 \text{ cycles/byte for } n \approx 2^{10}, \]

\[ \approx 55 \text{ cycles/byte for } n \approx 2^{20}. \]

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

IPP: Intel's Integrated Performance Primitives library.

Constant-time results, again on Haswell CPU core:

2017 BCLvV:

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

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

2018 djbsort:

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

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

No slowdown. New speed records!

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

How can an \( n(\log n)^2 \) algorithm beat standard \( n \log n \) algorithms?
The slowdown for constant time
Massive fast-sorting literature.
2015 Gueron–Krasnov: AVX and
AVX2 (Haswell) optimization of
quicksort. For 32-bit integers:
\[ \approx 45 \text{ cycles/byte for } n \approx 2^{10}, \]
\[ \approx 55 \text{ cycles/byte for } n \approx 2^{20}. \]
Slower than "the radix sort
implemented of IPP, which is
the fastest in-memory sort we are
aware of": 32, 40 cycles/byte.
IPP: Intel's Integrated
Performance Primitives library.

Constant-time results,
again on Haswell CPU core:
2017 BCLvV:
6.5 cycles/byte for \( n \approx 2^{10} \),
33 cycles/byte for \( n \approx 2^{20} \).

2018 djbsort:
2.5 cycles/byte for \( n \approx 2^{10} \),
15.5 cycles/byte for \( n \approx 2^{20} \).

No slowdown. New speed records!
Warning: Comparison for \( n \approx 2^{20} \)
involves microarchitecture details
beyond Haswell core. Should
measure all code on same CPU.

How can an \( n(\log n)^2 \) algorithm
beat standard \( n \log n \) algorithms?
Constant-time results, again on Haswell CPU core:

2017 BCLvV:
6.5 cycles/byte for $n \approx 2^{10}$,
33 cycles/byte for $n \approx 2^{20}$.

2018 djbsort:
2.5 cycles/byte for $n \approx 2^{10}$,
15.5 cycles/byte for $n \approx 2^{20}$.

No slowdown. New speed records!

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

How can an $n(\log n)^2$ algorithm beat standard $n \log n$ algorithms?
Constant-time results, again on Haswell CPU core:

2017 BCLvV:
6.5 cycles/byte for $n \approx 2^{10}$,
33 cycles/byte for $n \approx 2^{20}$.

2018 djbsort:
2.5 cycles/byte for $n \approx 2^{10}$,
15.5 cycles/byte for $n \approx 2^{20}$.

No slowdown. New speed records!

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

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

Answer: well-known trends in CPU design, reflecting fundamental hardware costs of various operations.
Constant-time results, again on Haswell CPU core:

2017 BCLvV:
6.5 cycles/byte for $n \approx 2^{10}$,
33 cycles/byte for $n \approx 2^{20}$.

2018 djbsort:
2.5 cycles/byte for $n \approx 2^{10}$,
15.5 cycles/byte for $n \approx 2^{20}$.

No slowdown. New speed records!

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

---

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

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

Every cycle, Haswell core can do 8 “min” ops on 32-bit integers + 8 “max” ops on 32-bit integers.
Constant-time results, again on Haswell CPU core:

2017 BCLvV:
6.5 cycles/byte for $n \approx 2^{10}$, 33 cycles/byte for $n \approx 2^{20}$.

2018 djbsort:
2.5 cycles/byte for $n \approx 2^{10}$, 15.5 cycles/byte for $n \approx 2^{20}$.

No slowdown. New speed records!

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

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

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

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

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

Conditional branch: much slower.
Constant-time results, again on Haswell CPU core:

2017 BCLvV:
6 cycles/byte for $n \approx 2^{10}$,
33 cycles/byte for $n \approx 2^{20}$.

2018 djbsort:
6 cycles/byte for $n \approx 2^{10}$,
15 cycles/byte for $n \approx 2^{20}$.

No slowdown. New speed records!

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

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

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

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

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

Conditional branch: much slower.

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.
Constant-time results, again on Haswell CPU core:

2017 BCLvV:
6 \text{ cycles/byte for } n \approx 2^{10},
33 \text{ cycles/byte for } n \approx 2^{20}.

2018 djbsort:
6 \text{ cycles/byte for } n \approx 2^{10},
15 \text{ cycles/byte for } n \approx 2^{20}.

No slowdown. New speed records!

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

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

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

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

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

Conditional branch: much slower.

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.
Constant-time results, again on Haswell CPU core:

2017 BCLvV:
6 : 5 cycles/byte for $n \approx 2^{10}$,
33 cycles/byte for $n \approx 2^{20}$.

2018 djbsort:
2 : 5 cycles/byte for $n \approx 2^{10}$,
15 : 6 cycles/byte for $n \approx 2^{20}$.

No slowdown. New speed records!

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

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

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

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

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

Conditional branch: much slower.

Verification

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

Test the sorting software on
random inputs, increasing inputs,
and 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.
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.

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
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.

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
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
\[\rightarrow\]
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
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
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):

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

Symbolic execution:

use existing “angr” library, with 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.

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

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

Symbolic execution: use existing “angr” library, with tiny new patches for eliminating byte splitting, adding a few missing vector instructions.
Sorting software is in the TCB. Does it work correctly? Test the sorting software on many random inputs, increasing inputs, decreasing inputs. Seems to work. But are there occasional inputs where this sorting software fails to sort correctly? History: Many security problems involve occasional inputs where TCB works incorrectly.

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

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

Symbolic execution: use existing “angr” library, with tiny new patches for eliminating byte splitting, adding a few missing vector instructions.
For each used $n$ (e.g., 768):

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

Symbolic execution:
use existing “angr” library, with tiny new patches for eliminating byte splitting, adding a few missing vector instructions.
For each used $n$ (e.g., 768):

C code

normal compiler

downward

machine code

symbolic execution

downward

fully unrolled code

new peephole optimizer

downward

unrolled min-max code

new sorting verifier

downward

yes, code works

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

Peephole optimizer:
recognize instruction patterns equivalent to min, max.
For each used $n$ (e.g., 768):

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. yes, code works

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

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

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

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

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

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

Sorting verifier: decompose DAG into merging networks.

First djbsort release, verified int32 on AVX2:
https://sorting.cr.yp.to
Includes the sorting code; automatic build-time tests; simple benchmarking program; verification tools.
Web site shows how to use the verification tools.
Next release planned: verified ARM NEON code and verified portable code.
Symbolic execution:
use existing “angr” library, with tiny new patches for eliminating byte splitting, adding a few missing vector instructions.

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

Sorting verifier: decompose DAG into merging networks.

First djbsort release, verified int32 on AVX2:
https://sorting.cr.yp.to
Includes the sorting code; automatic build-time tests; simple benchmarking program; verification tools.
Web site shows how to use the verification tools.
Next release planned: verified ARM NEON code and verified portable code.
Symbolic execution:
use existing “angr” library, with tiny new patches for eliminating byte splitting, adding a few missing vector instructions.

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

Sorting verifier: decompose DAG into merging networks.

First djbsort release, verified int32 on AVX2:
https://sorting.cr.yp.to
Includes the sorting code; automatic build-time tests; simple benchmarking program; verification tools.
Web site shows how to use the verification tools.
Next release planned: verified ARM NEON code and verified portable code.
Symbolic execution:
use existing “angr” library,
with tiny new patches for
eliminating byte splitting, adding
a few missing vector instructions.

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

Sorting verifier: decompose DAG into merging networks.
Verify each merging network using generalization of 2007

First djbsort release,
verified int32 on AVX2:
https://sorting.cr.yp.to
Includes the sorting code;
automatic build-time tests;
simple benchmarking program;
verification tools.
Web site shows how to use the verification tools.
Next release planned:
verified ARM NEON code
and verified portable code.