Fast verified post-quantum software, part 1: RAM subroutines

D. J. Bernstein

Performance pressure ⇒ tons of new crypto software ⇒
many mistakes passing tests ⇒ frequent security disasters.


e.g. 2019.09 “Produced signatures were valid but leaked information on the private key”: Falcon.

e.g. 2019.10 “Minerva attack can recover private keys from smart cards, cryptographic libraries”.

e.g. 2020.08 “A key-recovery timing attack on . . . FrodoKEM”.

e.g. 2020.12 “It looks like the FrodoKEM team also fixed the timing oracle [GJN20] badly and caused a more serious security problem while trying to do that.”
Performance pressure ⇒ tons of new crypto software ⇒ many mistakes passing tests ⇒ frequent security disasters.

e.g. 2019.06 “Warning: Google Researcher Drops Windows 10 Zero-Day Security Bomb”:

- modular inverse.

- e.g. 2019.09 “Produced signatures were valid but leaked information on the private key”: Falcon.

- e.g. 2019.10 “Minerva attack can recover private keys from smart cards, cryptographic libraries”.

- e.g. 2020.08 “A key-recovery timing attack on : : : FrodoKEM”.

- e.g. 2020.12 “It looks like the FrodoKEM team also fixed the timing oracle [GJN20] badly and caused a more serious security problem while trying to do that.”

Many functions × many CPUs

Keccak (SHA-3) team maintains “Keccak Code Package” with >20 optimized implementations of Keccak: AVX2, NEON, etc.

Also, for “parallel Keccak”, many further implementations.
Fast verified post-quantum software, part 1: RAM subroutines

D. J. Bernstein

Performance pressure ⇒ tons of new crypto software ⇒ many mistakes passing tests ⇒ frequent security disasters.

e.g. 2019.06 “Warning: Google Researcher Drops Windows 10 Zero-Day Security Bomb”:

modular inverse.

e.g. 2019.09 “Produced signatures were valid but leaked information on the private key”: Falcon.

e.g. 2019.10 “Minerva attack can recover private keys from smart cards, cryptographic libraries”.

e.g. 2020.08 “A key-recovery timing attack on ... FrodoKEM”.

e.g. 2020.12 “It looks like the FrodoKEM team also fixed the timing oracle [GJN20] badly and caused a more serious security problem while trying to do that.”

Many functions × many CPUs

Keccak (SHA-3) team maintains “Keccak Code Package” with >20 optimized implementations of Keccak: AVX2, NEON, etc. Also, for “parallel Keccak”, many further implementations.
Fast verified post-quantum software, part 1: RAM subroutines

D. J. Bernstein

Performance pressure ⇒ tons of new crypto software ⇒ many mistakes passing tests ⇒ frequent security disasters.

e.g. 2019.06 “Warning: Google Researcher Drops Windows 10 Zero-Day Security Bomb”:

modular inverse.

e.g. 2019.09 “Produced signatures were valid but leaked information on the private key”: Falcon.

e.g. 2019.10 “Minerva attack can recover private keys from smart cards, cryptographic libraries”.

e.g. 2020.08 “A key-recovery timing attack on ... FrodoKEM”.

e.g. 2020.12 “It looks like the FrodoKEM team also fixed the timing oracle [GJN20] badly and caused a more serious security problem while trying to do that.”

Many functions × many CPUs

Keccak (SHA-3) team maintains “Keccak Code Package” with >20 optimized implementations of Keccak: AVX2, NEON, etc.

Also, for “parallel Keccak”, many further implementations.
e.g. 2019.09 “Produced signatures were valid but leaked information on the private key”: Falcon.

e.g. 2019.10 “Minerva attack can recover private keys from smart cards, cryptographic libraries”.

e.g. 2020.08 “A key-recovery timing attack on . . . FrodoKEM”.

e.g. 2020.12 “It looks like the FrodoKEM team also fixed the timing oracle [GJN20] badly and caused a more serious security problem while trying to do that.”

Many functions × many CPUs
Keccak (SHA-3) team maintains “Keccak Code Package” with >20 optimized implementations of Keccak: AVX2, NEON, etc. Also, for “parallel Keccak”, many further implementations.
e.g. 2019.09 “Produced signatures were valid but leaked information on the private key”: Falcon.
e.g. 2019.10 “Minerva attack can recover private keys from smart cards, cryptographic libraries”.
e.g. 2020.08 “A key-recovery timing attack on … FrodoKEM”.
e.g. 2020.12 “It looks like the FrodoKEM team also fixed the timing oracle [GJN20] badly and caused a more serious security problem while trying to do that.”

Many functions $\times$ many CPUs
Keccak (SHA-3) team maintains “Keccak Code Package” with $>20$ optimized implementations of Keccak: AVX2, NEON, etc. Also, for “parallel Keccak”, many further implementations. Why not portable C code using “optimizing” compiler? Slower.
e.g. 2019.09 “Produced signatures were valid but leaked information on the private key”: Falcon.
e.g. 2019.10 “Minerva attack can recover private keys from smart cards, cryptographic libraries”.
e.g. 2020.08 “A key-recovery timing attack on ... FrodoKEM”.
e.g. 2020.12 “It looks like the FrodoKEM team also fixed the timing oracle [GJN20] badly and caused a more serious security problem while trying to do that.”

Many functions $\times$ many CPUs
Keccak (SHA-3) team maintains “Keccak Code Package” with $>20$ optimized implementations of Keccak: AVX2, NEON, etc. Also, for “parallel Keccak”, many further implementations.

Why not portable C code using “optimizing” compiler? Slower.
Post-quantum crypto is going down same path: AVX2, ARM Cortex-M4, Cortex-A7, Cortex-A53, Zen, AVX-512, RISC-V, ...
Many functions × many CPUs
Keccak (SHA-3) team maintains “Keccak Code Package” with >20 optimized implementations of Keccak: AVX2, NEON, etc. Also, for “parallel Keccak”, many further implementations.
Why not portable C code using “optimizing” compiler? Slower.
Post-quantum crypto is going down same path: AVX2, ARM Cortex-M4, Cortex-A7, Cortex-A53, Zen, AVX-512, RISC-V, ...

Some good news
For some types of optimized code, and some types of specs:
Without insane levels of effort, can have an automated guarantee that the optimized code does what the spec says.
Many functions × many CPUs
Keccak (SHA-3) team maintains “Keccak Code Package” with >20 optimized implementations of Keccak: AVX2, NEON, etc. Also, for “parallel Keccak”, many further implementations.

Why not portable C code using “optimizing” compiler? Slower.

Post-quantum crypto is going down same path: AVX2, ARM Cortex-M4, Cortex-A7, Cortex-A53, Zen, AVX-512, RISC-V, …
Many functions × many CPUs

Keccak (SHA-3) team maintains “Keccak Code Package” with >20 optimized implementations of Keccak: AVX2, NEON, etc. Also, for “parallel Keccak”, many further implementations.

Why not portable C code using “optimizing” compiler? Slower.

Post-quantum crypto is going down same path: AVX2, ARM Cortex-M4, Cortex-A7, Cortex-A53, Zen, AVX-512, RISC-V, …
Many functions × many CPUs
Keccak (SHA-3) team maintains “Keccak Code Package” with >20 optimized implementations of Keccak: AVX2, NEON, etc. Also, for “parallel Keccak”, many further implementations.

Why not portable C code using “optimizing” compiler? Slower.

Post-quantum crypto is going down same path: AVX2, ARM Cortex-M4, Cortex-A7, Cortex-A53, Zen, AVX-512, RISC-V, …

Some good news
For some types of optimized code, and some types of specs: Without insane levels of effort, can have an automated guarantee that the optimized code does what the spec says.
Many functions × many CPUs

Keccak (SHA-3) team maintains “Keccak Code Package” with >20 optimized implementations of Keccak: AVX2, NEON, etc. Also, for “parallel Keccak”, many further implementations.

Why not portable C code using “optimizing” compiler? Slower.

Post-quantum crypto is going down same path: AVX2, ARM Cortex-M4, Cortex-A7, Cortex-A53, Zen, AVX-512, RISC-V, …

Some good news

For some types of optimized code, and some types of specs:

Without insane levels of effort, can have an automated guarantee that the optimized code does what the spec says.

Security reviewer still has to check whether the spec is secure and has to check for bugs in the verification tools—but saves tons of time in checking code optimized for each CPU.
Many functions × many CPUs

Keccak (SHA-3) team maintains “Keccak Code Package” with >20 optimized implementations: AVX2, NEON, etc. For “parallel Keccak”, further implementations.

Why not portable C code using “optimizing” compiler? Slower.

Quantum crypto is going down same path: AVX2, ARM Cortex-M4, Cortex-A7, Cortex-A53, Zen, AVX-512, RISC-V, ...

Some good news

For some types of optimized code, and some types of specs:

Without insane levels of effort, can have an automated guarantee that the optimized code does what the spec says.

Security reviewer still has to check whether the spec is secure and has to check for bugs in the verification tools—but saves tons of time in checking code optimized for each CPU.

What exactly is “the spec”?

Starting in 1960, CACM published algorithms—written in ALGOL. Why do we tolerate algorithms written in English “pseudocode”?
Many functions × many CPUs
Keccak (SHA-3) team maintains
"Keccak Code Package" with
> 20 optimized implementations
of Keccak: AVX2, NEON, etc.
Also, for "parallel Keccak",
many further implementations.

Why not portable C code using
"optimizing" compiler? Slower.
Crypto is going down same path: AVX2, ARM Cortex-M4, Cortex-A7, Cortex-A53, Zen, AVX-512, RISC-V, ...

Some good news
For some types of optimized code, and some types of specs:
Without insane levels of effort, can have an automated guarantee that the optimized code does what the spec says.

Security reviewer still has to check whether the spec is secure and has to check for bugs in the verification tools—but saves tons of time in checking code optimized for each CPU.

What exactly is “the spec”?
Starting in 1960, CACM published algorithms—written in ALGOL. Why do we tolerate algorithms written in English "pseudocode"?
Some good news

For some types of optimized code, and some types of specs:
Without insane levels of effort, can have an automated guarantee that the optimized code does what the spec says.

Security reviewer still has to check whether the spec is secure and has to check for bugs in the verification tools—but saves tons of time in checking code optimized for each CPU.

What exactly is “the spec”?
Starting in 1960, CACM published algorithms—written in ALGOL. Why do we tolerate algorithms written in English “pseudocode”?...
Some good news
For some types of optimized code, and some types of specs:
Without insane levels of effort, can have an automated guarantee that the optimized code does what the spec says.

Security reviewer still has to check whether the spec is secure and has to check for bugs in the verification tools—but saves tons of time in checking code optimized for each CPU.

What exactly is “the spec”?
Starting in 1960, CACM published algorithms—written in ALGOL. Why do we tolerate algorithms written in English “pseudocode”?
Some good news
For some types of optimized code, and some types of specs:
Without insane levels of effort, can have an automated guarantee that the optimized code does what the spec says.
Security reviewer still has to check whether the spec is secure and has to check for bugs in the verification tools—but saves tons of time in checking code optimized for each CPU.

What exactly is “the spec”? Starting in 1960, CACM published algorithms—written in ALGOL. Why do we tolerate algorithms written in English “pseudocode”? “Easier to read than ref”: that’s because ref
• was forced to be in C,
• often tries to be constant time,
• sometimes tries to be fast.
Some good news

For some types of optimized code, and some types of specs:

Without insane levels of effort, can have an automated guarantee that the optimized code does what the spec says.

Security reviewer still has to check whether the spec is secure and has to check for bugs in the verification tools—but saves tons of time in checking code optimized for each CPU.

What exactly is “the spec”?

Starting in 1960, CACM published algorithms—written in ALGOL. Why do we tolerate algorithms written in English “pseudocode”?

“Easier to read than ref”: that’s because ref

• was forced to be in C,
• often tries to be constant time,
• sometimes tries to be fast.

No conflict between spec being (1) easy to read, (2) executable.

Verify spec = ref = avx2 = ⋯ .

Security reviewers focus on spec.
Some good news

For some types of optimized code, and some types of specs:

Without insane levels of effort, can have an automated guarantee that the optimized code does what the spec says.

Security reviewer still has to check whether the spec is secure and has to check for bugs in the verification tools—but saves tons of time in checking code optimized for each CPU.

What exactly is “the spec”?

Starting in 1960, CACM published algorithms—written in ALGOL. Why do we tolerate algorithms written in English “pseudocode”? “Easier to read than ref”: that’s because ref

• was forced to be in C,
• often tries to be constant time,
• sometimes tries to be fast.

No conflict between spec being (1) easy to read, (2) executable.
Verify spec = ref = avx2 = ···.
Security reviewers focus on spec.

Case study: RAM subroutines

Many algorithms rely on RAM. CPU RAM instructions leak secret addresses through timing. Can we eliminate timing leaks?
Some good news
For some types of optimized code, and some types of specs:
Without insane levels of effort, can have an automated guarantee
that the optimized code does what the spec says.
Security reviewer still has to check whether the spec is secure
and has to check for bugs in the verification tools—but saves tons of time in checking
code optimized for each CPU.

What exactly is “the spec”?
Starting in 1960, CACM published algorithms—written in ALGOL. Why do we tolerate algorithms written in English “pseudocode”?
“Easier to read than ref”: that’s because ref
- was forced to be in C,
- often tries to be constant time,
- sometimes tries to be fast.
No conflict between spec being (1) easy to read, (2) executable.
Verify spec = ref = avx2 = ···.
Security reviewers focus on spec.

Case study: RAM subroutines
Many algorithms rely on RAM. CPU RAM instructions leak secret addresses through timing. Can we eliminate this?
Some good news
For some types of optimized code, and some types of specs: Without insane levels of effort, can have an automated guarantee that the optimized code does what the spec says. Security reviewer still has to check whether the spec is secure and has to check for bugs in the verification tools—but saves tons of time in checking code optimized for each CPU.

What exactly is “the spec”?
Starting in 1960, CACM published algorithms—written in ALGOL. Why do we tolerate algorithms written in English “pseudocode”? “Easier to read than ref”: that’s because ref
• was forced to be in C,
• often tries to be constant time,
• sometimes tries to be fast.

No conflict between spec being (1) easy to read, (2) executable.
Verify spec = ref = avx2 = ···.
Security reviewers focus on spec.

Case study: RAM subroutines
Many algorithms rely on RAM. CPU RAM instructions leak secret addresses through timing. Can we eliminate timing leaks?
What exactly is “the spec”?  
Starting in 1960, CACM published algorithms—written in ALGOL. Why do we tolerate algorithms written in English “pseudocode”?  

“Easier to read than ref”: that’s because ref
- was forced to be in C,
- often tries to be constant time,
- sometimes tries to be fast.

No conflict between spec being (1) easy to read, (2) executable. Verify spec = ref = avx2 = ···. Security reviewers focus on spec.

Case study: RAM subroutines
Many algorithms rely on RAM. CPU RAM instructions leak secret addresses through timing. Can we eliminate timing leaks?
What exactly is “the spec”?

Starting in 1960, CACM published algorithms—written in ALGOL. Why do we tolerate algorithms written in English “pseudocode”? “Easier to read than ref”: that’s because ref

- was forced to be in C,
- often tries to be constant time,
- sometimes tries to be fast.

No conflict between spec being (1) easy to read, (2) executable.

Verify spec = ref = avx2 = ···.

Security reviewers focus on spec.

Case study: RAM subroutines

Many algorithms rely on RAM. CPU RAM instructions leak secret addresses through timing. Can we eliminate timing leaks?

Yes! Replace CPU RAM insns with software to simulate RAM.
What exactly is “the spec”?  
Starting in 1960, CACM published algorithms—written in ALGOL. Why do we tolerate algorithms written in English “pseudocode”?  

“Easier to read than ref”: that’s because ref
• was forced to be in C,
• often tries to be constant time,
• sometimes tries to be fast.

No conflict between spec being (1) easy to read, (2) executable.
Verify spec = ref = avx2 = ··· .
Security reviewers focus on spec.

Case study: RAM subroutines
Many algorithms rely on RAM. CPU RAM instructions leak secret addresses through timing. Can we eliminate timing leaks?
Yes! Replace CPU RAM insns with software to simulate RAM.

Speedup #1: Use sorting to efficiently simulate parallel RAM.
What exactly is “the spec”?

Starting in 1960, CACM published algorithms—written in ALGOL. Why do we tolerate algorithms written in English “pseudocode”?

“Easier to read than ref”: that’s because ref
• was forced to be in C,
• often tries to be constant time,
• sometimes tries to be fast.

No conflict between spec being (1) easy to read, (2) executable. Verify spec = ref = avx2 = ···. Security reviewers focus on spec.

Case study: RAM subroutines

Many algorithms rely on RAM. CPU RAM instructions leak secret addresses through timing. Can we eliminate timing leaks?

Yes! Replace CPU RAM insns with software to simulate RAM.

Speedup #1: Use sorting to efficiently simulate parallel RAM.

Speedup #2: Sometimes same permutation is applied to many inputs. Precompute “control bits” for permutation.
What exactly is “the spec”? Starting in 1960, CACM published algorithms—written in ALGOL. Why do we tolerate algorithms in English “pseudocode”? “Easier to read than ref”: because ref was forced to be in C, often tries to be constant time, and sometimes tries to be fast.

Let’s not mistake between spec being easy to read, (2) executable.

Verify spec = ref = avx2 = ···

Security reviewers focus on spec.

Case study: RAM subroutines

Many algorithms rely on RAM. CPU RAM instructions leak secret addresses through timing. Can we eliminate timing leaks?

Yes! Replace CPU RAM insns with software to simulate RAM.

Speedup #1: Use sorting to efficiently simulate parallel RAM.

Speedup #2: Sometimes same permutation is applied to many inputs. Precompute “control bits” for permutation.

2018 Bernstein: speed records for sorting integer arrays. Verified constant-time software.


This software is already used inside current software releases for Classic McEliece (sorting and permutations), NTRU (sorting), NTRU Prime (sorting).
What exactly is “the spec”?

Starting in 1960, CACM published algorithms—written in ALGOL.

Why do we tolerate algorithms written in English “pseudocode”?

“Easier to read than ref”:

- ref was forced to be in C,
- often tries to be constant time,
- sometimes tries to be fast.

No conflict between spec being (1) easy to read, (2) executable.

Verify spec = ref = avx2 = ··· .

Security reviewers focus on spec.

Case study: RAM subroutines

Many algorithms rely on RAM.

CPU RAM instructions leak secret addresses through timing.

Can we eliminate timing leaks?

Yes! Replace CPU RAM insns with software to simulate RAM.

Speedup #1: Use sorting to efficiently simulate parallel RAM.

Speedup #2: Sometimes same permutation is applied to many inputs. Precompute “control bits” for permutation.

2018 Bernstein: speed records for sorting integer arrays.

Verified constant-time software.

2020 Bernstein: speed records for constant-time permutations.

HOL Light proof of algorithm.

Coming soon: verification of the permutation software.

This software is already used inside current software releases for Classic McEliece (sorting and permutations), NTRU, NTRU Prime (sorting).
What exactly is “the spec”?

Starting in 1960, CACM published algorithms—written in ALGOL. Why do we tolerate algorithms written in English “pseudocode”? “Easier to read than ref”:

- was forced to be in C,
- often tries to be constant time,
- sometimes tries to be fast.

No conflict between spec being (1) easy to read, (2) executable.

Verify spec = ref = avx2 = ···.

Security reviewers focus on spec.

Case study: RAM subroutines

Many algorithms rely on RAM. CPU RAM instructions leak secret addresses through timing. Can we eliminate timing leaks?

Yes! Replace CPU RAM insns with software to simulate RAM.

Speedup #1: Use sorting to efficiently simulate parallel RAM.

Speedup #2: Sometimes same permutation is applied to many inputs. Precompute “control bits” for permutation.

2018 Bernstein: speed records for sorting integer arrays. Verified constant-time software.

2020 Bernstein: speed records for constant-time permutations. HOL Light proof of algorithm.

Coming soon: verification of the permutation software.

This software is already used inside current software releases for Classic McEliece (sorting and permutations), NTRU (sorting), NTRU Prime (sorting).
Case study: RAM subroutines

Many algorithms rely on RAM. CPU RAM instructions leak secret addresses through timing. Can we eliminate timing leaks?

Yes! Replace CPU RAM insns with software to simulate RAM.

Speedup #1: Use sorting to efficiently simulate *parallel* RAM.

Speedup #2: Sometimes same permutation is applied to many inputs. Precompute “control bits” for permutation.

2018 Bernstein: speed records for sorting integer arrays. Verified constant-time software.

2020 Bernstein: speed records for constant-time permutations. HOL Light proof of algorithm.

Coming soon: verification of the permutation software.

This software is already used inside current software releases for Classic McEliece (sorting and permutations), NTRU (sorting), NTRU Prime (sorting).
Case study: RAM subroutines

Many algorithms rely on RAM. CPU RAM instructions leak secret addresses through timing. Can we eliminate timing leaks? Replace CPU RAM instructions with software to simulate RAM.

1. Use sorting to efficiently simulate parallel RAM.
2. Sometimes same permutation is applied to many inputs. Precompute "control bits" for permutation.

2018 Bernstein: speed records for sorting integer arrays. Verified constant-time software.

2020 Bernstein: speed records for constant-time permutations. HOL Light proof of algorithm.

Coming soon: verification of the permutation software.

This software is already used inside current software releases for Classic McEliece (sorting and permutations), NTRU (sorting), NTRU Prime (sorting).

The conventional path

Imagine an optimizing compiler automatically converting spec → fast binary for whichever CPU. “Compiler verification”: prove that the compiler always works correctly.

If all of this is done, great!
Case study: RAM subroutines

Many algorithms rely on RAM. CPU RAM instructions leak secret addresses through timing. Can we eliminate timing leaks? Yes! Replace CPU RAM insns with software to simulate RAM.

Speedup #1: Use sorting to efficiently simulate parallel RAM.

Speedup #2: Sometimes the same permutation is applied to many inputs. Precompute "control bits" for permutation.

2018 Bernstein: speed records for sorting integer arrays. Verified constant-time software.

2020 Bernstein: speed records for constant-time permutations. HOL Light proof of algorithm.

Coming soon: verification of the permutation software.

This software is already used inside current software releases for Classic McEliece (sorting and permutations), NTRU (sorting), NTRU Prime (sorting).

The conventional path...

Imagine an optimizing compiler automatically converting spec → fast binary for whichever CPU.

"Compiler verification": prove that the compiler always works correctly.

If all of this is done, great!
Case study: RAM subroutines

Many algorithms rely on RAM. CPU RAM instructions leak secret addresses through timing. Can we eliminate timing leaks? Yes! Replace CPU RAM insns with software to simulate RAM.

Speedup #1: Use sorting to efficiently simulate parallel RAM.

Speedup #2: Sometimes same permutation is applied to many inputs. Precompute “control bits” for permutation.

2018 Bernstein: speed records for sorting integer arrays. Verified constant-time software.

2020 Bernstein: speed records for constant-time permutations. HOL Light proof of algorithm.

Coming soon: verification of the permutation software.

This software is already used inside current software releases for Classic McEliece (sorting and permutations), NTRU (sorting), NTRU Prime (sorting).

The conventional path

Imagine an optimizing compiler automatically converting spec → fast binary for whichever CPU.

“Compiler verification”: prove that the compiler always works correctly.

If all of this is done, great!
2018 Bernstein: speed records for sorting integer arrays. Verified constant-time software.

2020 Bernstein: speed records for constant-time permutations. HOL Light proof of algorithm.

Coming soon: verification of the permutation software.

This software is already used inside current software releases for Classic McEliece (sorting and permutations), NTRU (sorting), NTRU Prime (sorting).

---

The conventional path

Imagine an optimizing compiler automatically converting spec → fast binary for whichever CPU.

“Compiler verification”: prove that the compiler always works correctly.

If all of this is done, great!
2018 Bernstein: speed records for sorting integer arrays. Verified constant-time software.

2020 Bernstein: speed records for constant-time permutations. HOL Light proof of algorithm.

Coming soon: verification of the permutation software.

This software is already used inside current software releases for Classic McEliece (sorting and permutations), NTRU (sorting), NTRU Prime (sorting).

The conventional path
Imagine an optimizing compiler automatically converting spec → fast binary for whichever CPU.

“Compiler verification”: prove that the compiler always works correctly.

If all of this is done, great!

Reality: Again, look at Keccak. Speedups > automated speedups > verified automated speedups.
Bernstein: speed records for sorting integer arrays.
Verified constant-time software.

Bernstein: speed records for constant-time permutations.
HOL Light proof of algorithm.

Coming soon: verification of the permutation software.
This software is already used inside current software releases for Classic McEliece (sorting and permutations), NTRU (sorting), NTRU Prime (sorting).

The conventional path
Imagine an optimizing compiler automatically converting spec → fast binary for whichever CPU.

“Compiler verification”: prove that the compiler always works correctly.

If all of this is done, great!

Reality: Again, look at Keccak.

Speedups > automated speedups > verified automated speedups.

Verifying fast software
Optimization experts:
spec → opt → opt2 → opt3 → opt4 → opt5 → ... → avx2.
Some manual steps, some tools.
CPUs share some steps.
2018 Bernstein: speed records for sorting integer arrays.
Verified constant-time software.

2020 Bernstein: speed records for constant-time permutations.
HOL Light proof of algorithm.

Coming soon: verification of the permutation software.
This software is already used inside current software releases for Classic McEliece (sorting and permutations), NTRU (sorting), NTRU Prime (sorting).

The conventional path
Imagine an optimizing compiler automatically converting spec → fast binary for whichever CPU.

“Compiler verification”: prove that the compiler always works correctly.

If all of this is done, great!

Reality: Again, look at Keccak.

Speedups > automated speedups > verified automated speedups.
The conventional path

Imagine an optimizing compiler automatically converting spec → fast binary for whichever CPU.

“Compiler verification”:
prove that the compiler always works correctly.

If all of this is done, great!

Reality: Again, look at Keccak.

Speedups >
automated speedups >
verified automated speedups.

Verifying fast software

Optimization experts:
spec → opt → opt2 → opt3 → opt4 → opt5 → ··· → avx2.
Some manual steps, some tools.
CPUs share some steps.
The conventional path
Imagine an optimizing compiler automatically converting spec $\rightarrow$ fast binary for whichever CPU.

“Compiler verification”: prove that the compiler always works correctly.

If all of this is done, great!

Reality: Again, look at Keccak.

Speedups $>$
automated speedups $>$
verified automated speedups.

Verifying fast software
Optimization experts:
spec $\rightarrow$ opt $\rightarrow$ opt2 $\rightarrow$ opt3 $\rightarrow$ opt4 $\rightarrow$ opt5 $\rightarrow$ $\cdots$ $\rightarrow$ avx2.
Some manual steps, some tools. CPUs share some steps.
The conventional path

Imagine an optimizing compiler automatically converting \( \text{spec} \rightarrow \) fast binary for whichever CPU.

“Compiler verification”: prove that the compiler always works correctly.

If all of this is done, great!

Reality: Again, look at Keccak.

Speedups > automated speedups > verified automated speedups.

Verifying fast software

Optimization experts:

\[ \text{spec} \rightarrow \text{opt} \rightarrow \text{opt2} \rightarrow \text{opt3} \rightarrow \text{opt4} \rightarrow \text{opt5} \rightarrow \cdots \rightarrow \text{avx2}. \]

Some manual steps, some tools. CPUs share some steps.

“Translation validation”: verify equivalence of tool output to tool input. Doesn’t require verifying that the tool always works.

The conventional path
Imagine an optimizing compiler automatically converting spec → fast binary for whichever CPU.

“Compiler verification”:
prove that the compiler always works correctly.

If all of this is done, great!

Reality: Again, look at Keccak.
Speedups > automated speedups > verified automated speedups.

Verifying fast software
Optimization experts:
spec → opt → opt2 → opt3 → opt4 → opt5 → ··· → avx2.
Some manual steps, some tools.
CPUs share some steps.

“Translation validation”:
verify equivalence of tool output to tool input.

Doesn’t require verifying that the tool always works.

“Transformation verification”:
verify equivalence of manual output to manual input.

Allowing new verification chains
For verification, suffices to build spec ↔ verif ↔ verif2 ↔ verif3 ↔···↔ avx2.
Don’t try to force this chain to match the development path spec → opt → opt2 → opt3 → opt4 →···→ avx2.
Imagine an optimizing compiler automatically converting spec → fast binary for whichever CPU.

"Compiler verification": prove that the compiler always works correctly.

If all of this is done, great!

Reality: Again, look at Keccak.

Speedups > automated speedups > verified automated speedups.

---

Verifying fast software

Optimization experts:
spec → opt → opt2 → opt3 → opt4 → opt5 → ··· → avx2.

Some manual steps, some tools. CPUs share some steps.

"Translation validation": verify equivalence of tool output to tool input. Doesn’t require verifying that the tool always works.

"Transformation verification": verify equivalence of manual output to manual input.

---

Allowing new verification chains

For verification, suffices to build spec ↔ verif ↔ verif2 ↔ verif3 ↔ ··· ↔ avx2.

Don’t try to force this chain to match the development path spec → opt → opt2 → opt3 → opt4 → opt5 → ··· → avx2.
The conventional path
Imagine an optimizing compiler automatically converting spec → fast binary for whichever CPU.

"Compiler verification": prove that the compiler always works correctly.
If all of this is done, great!
Reality: Again, look at Keccak.

Speedups > automated speedups > verified automated speedups.

---

Verifying fast software
Optimization experts:
spec → opt → opt2 → opt3 → opt4 → opt5 → ··· → avx2.
Some manual steps, some tools. CPUs share some steps.

“Translation validation”: verify equivalence of tool output to tool input.
Doesn’t require verifying that the tool always works.


---

Allowing new verification chains
For verification, suffices to build
spec ↔ verif ↔ verif2 ↔ verif3 ↔ ··· ↔ avx2.
Don’t try to force this chain to match the development path
spec → opt → opt2 → opt3 → opt4 → opt5 → ··· → avx2.
Verifying fast software

Optimization experts:

spec → opt → opt2 → opt3 →
opt4 → opt5 → ··· → avx2.

Some manual steps, some tools.
CPUs share some steps.

“Translation validation”:
verify equivalence of
tool output to tool input.
Doesn’t require verifying
that the tool \textit{always} works.

“Transformation verification”:
verify equivalence of
manual output to manual input.

Allowing new verification chains

For verification, suffices to build
spec ↔ verif ↔ verif2 ↔
verif3 ↔ ··· ↔ avx2.

Don’t try to force this chain to
match the development path
spec → opt → opt2 → opt3 →
opt4 → opt5 → ··· → avx2.
Verifying fast software

Optimization experts:

spec → opt → opt2 → opt3 →
opt4 → opt5 → ··· → avx2.

Some manual steps, some tools.
CPUs share some steps.

“Translation validation”:
verify equivalence of
tool output to tool input.
Doesn’t require verifying
that the tool always works.

“Transformation verification”:
verify equivalence of
manual output to manual input.

Allowing new verification chains

For verification, suffices to build
spec ↔ verif ↔ verif2 ↔
verif3 ↔ ··· ↔ avx2.

Don’t try to force this chain to
match the development path
spec → opt → opt2 → opt3 →
opt4 → opt5 → ··· → avx2.

Separation promotes independent
speedups in (1) the development
process and (2) the verification
process: e.g., vectorization is
often challenging for development
but trivial for verification.