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."

ified
antum software,
RAM subroutines

ernstein

---

ance pressure ⇒
new crypto software ⇒
istakes passing tests ⇒
security disasters.

9.06 “Warning: Google
ner Drops Windows
-Day Security Bomb”:
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 fu

Keccak
“Keccak
>20 opt
of Kecca
Also, for
many fu

tware,
outines

ure ⇒
o software ⇒
ssing tests ⇒
disasters.

rning: Google
Windows
rity Bomb":

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 ×

Keccak (SHA-3) t
"Keccak Code Pac
>20 optimized im
of Keccak: AVX2,
Also, for "parallel
many further impl

$\Rightarrow$

$\Rightarrow$

ogle

”:

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 CP

Keccak (SHA-3) team main
“Keccak Code Package” wit
$>$20 optimized implementat
of Keccak: AVX2, NEON, e
Also, for “parallel Keccak”,
many further implementatio

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.

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 × 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, . . .

9.09 "Produced signatures
id but leaked information
rivate key": Falcon.

9.10 "Minerva attack can
private keys from smart
ryptographic libraries".

0.08 "A key-recovery
ttack on … FrodoKEM".

0.12 "It looks like the
EM team also fixed the
racle [GJN20] badly and
a more serious security
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.

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 go

For some
and som

Without
can have
that the
does wh

duced signatures
ked information
": Falcon.

erva attack can
ys from smart
nic libraries".

ey-recovery
. . FrodoKEM".

oks like the
also fixed the
N20] badly and
ious security
ng 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.

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
and some types of

Without insane lev
can have an auton
that the optimized
does what the spe

natures
ation

k can
hart
s".

y
KEM".

ne
the
and
ity
hat."

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
and some types of specs:

Without insane levels of effo
can have an automated guar
that the optimized code
does what the spec says.

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, . . .

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.

nctions × many CPUs

(SHA-3) team maintains
Code Package" with
imized implementations
ak: AVX2, NEON, etc.
"parallel Keccak",
rther implementations.

t portable C code using
zing" compiler? Slower.

antum crypto is going
me path: AVX2, ARM
M4, Cortex-A7, Cortex-
n, 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 ex

Starting
algorithr
Why do
written i

many CPUs

eam maintains
ckage" with
plementations
NEON, etc.
Keccak",
ementations.

C code using
piler? Slower.

pto is going
AVX2, ARM
-A7, Cortex-
2, 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 "t

Starting in 1960, C
algorithms—writte
Why do we tolerat
written in English

Us

tains
th
ions
tc.

ns.

sing
ver.

ng
RM
ex-
/, . . .

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 pub
algorithms—written in ALG
Why do we tolerate algorith
written in English “pseudoco

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 $\texttt{spec} = \texttt{ref} = \texttt{avx2} = \cdots$.
Security reviewers focus on `spec`.

ood news

e types of optimized code,
e types of specs:

insane levels of effort,
e an automated guarantee
optimized code
at the spec says.

reviewer still has to
hether the spec is secure
to check for
the verification tools—
s tons of time in checking
timized 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` = $\cdots$. Security reviewers focus on `spec`.

Case stu

Many al
CPU RA
secret ac
Can we

optimized code,
f specs:

vels of effort,
nated guarantee
d code
c says.

still has to
e spec is secure
for
ation tools—
time in checking
r 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 $\texttt{spec} = \texttt{ref} = \texttt{avx2} = \cdots$. Security reviewers focus on `spec`.

Case study: RAM

Many algorithms r
CPU RAM instruc
secret addresses th
Can we eliminate t

l code,

ort,
rantee

ecure

s—
ecking
U.

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` = $\cdots$. Security reviewers focus on `spec`.

Case study: RAM subroutin

Many algorithms rely on RA
CPU RAM instructions leak
secret addresses through tim
Can we eliminate timing lea

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` = $\cdots$. 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 $\texttt{spec} = \texttt{ref} = \texttt{avx2} = \cdots$. 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` = $\cdots$. 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` = $\cdots$. 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.

xactly is “the spec”?

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

to read than `ref`”:
ecause `ref`
rced to be in C,
tries to be constant time,
imes tries to be fast.

lict between `spec` being
to read, (2) executable.
pec = `ref` = `avx2` = · · ·.
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 Be
for sorti
Verified

2020 Be
for const
HOL Lig

Coming
of the p

This sof
inside cu
for Class
permuta
NTRU P

the spec"?

CACM published
en in ALGOL.
te algorithms
"pseudocode"?

an `ref`":
f
e in C,
constant time,
to be fast.

en `spec` being
(2) executable.
f = `avx2` = · · · .
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: sp
for sorting integer
Verified constant-t

2020 Bernstein: sp
for constant-time
HOL Light proof o

Coming soon: veri
of the permutation

This software is al
inside current softw
for Classic McEliec
permutations), NT
NTRU Prime (sort

.
blished
OL.
ms
ode”?

time,
.
eing
able.
$= \cdots$.
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 recor
for sorting integer arrays.
Verified constant-time softw

2020 Bernstein: speed recor
for constant-time permutati
HOL Light proof of algorith

Coming soon: verification
of the permutation software

This software is already usec
inside current software relea
for Classic McEliece (sorting
permutations), NTRU (sorti
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).

udy: RAM subroutines

gorithms rely on RAM.
AM instructions leak
ddresses through timing.
eliminate timing leaks?

place CPU RAM insns
tware to simulate RAM.

#1: Use sorting to
y simulate *parallel* RAM.

#2: Sometimes
rmutation is applied
inputs. Precompute
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 con

Imagine
automat
fast bina

"Compil
prove th
always w

If all of

subroutines

rely on RAM.
tions leak
hrough timing.
timing leaks?

U RAM insns
imulate RAM.

sorting to
e *parallel* RAM.

netimes
is applied
Precompute
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

Imagine an optimi
automatically conv
fast binary for whi

“Compiler verifica
prove that the cor
always works corre

If all of this is don

es

AM.

ning.
ks?

ns
AM.

RAM.

e
on.

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 comp
automatically converting sp
fast binary for whichever CP

"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 $\rightarrow$ 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 $\rightarrow$ 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.

rnstein: speed records
ng integer arrays.
constant-time software.

rnstein: speed records
tant-time permutations.
ght proof of algorithm.

soon: verification
ermutation software.

tware is already used
urrent software releases
sic McEliece (sorting and
tions), NTRU (sorting),
Prime (sorting).

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

Optimiz
`spec` $\rightarrow$
`opt4` $\rightarrow$
Some m
CPUs sh

peed records
arrays.
time software.

peed records
permutations.
of algorithm.

ification
n software.

ready used
ware releases
ce (sorting and
TRU (sorting),
ting).

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 softw

Optimization expe
`spec` $\rightarrow$ `opt` $\rightarrow$ `op`
`opt4` $\rightarrow$ `opt5` $\rightarrow$ ·
Some manual step
CPUs share some

ds

are.

ds
ons.
m.

d
ses
; and
ng),

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` $\rightarrow$ `opt`
`opt4` $\rightarrow$ `opt5` $\rightarrow \cdots \rightarrow$ `avx`
Some manual steps, some to
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 `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.

ventional path

an optimizing compiler
tically converting `spec` $\rightarrow$
ary for whichever CPU.

er verification":
at the compiler
vorks correctly.

this is done, great!

Again, look at Keccak.

s $>$
ed speedups $>$
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.

"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

For verif
`spec` $\leftrightarrow$
`verif3`

Don't tr
match tl
`spec` $\rightarrow$
`opt4` $\rightarrow$

path

zing compiler
verting spec $\rightarrow$
chever CPU.

tion":
npiler
ectly.

e, great!

ok at Keccak.

ps >
l 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.

"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 verif

For verification, su
`spec` $\leftrightarrow$ `verif` $\leftrightarrow$
`verif3` $\leftrightarrow \cdots \leftrightarrow$ a

Don't try to force
match the develop
`spec` $\rightarrow$ `opt` $\rightarrow$ op
`opt4` $\rightarrow$ `opt5` $\rightarrow$ ·

iler

ec →

U.

cak.

s.

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 ch

For verification, suffices to b

spec ↔ verif ↔ verif2 ↔

verif3 ↔ · · · ↔ avx2.

Don't try to force this chain

match the development path

spec → opt → opt2 → opt

opt4 → opt5 → · · · → avx

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.

"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` $\leftrightarrow$ `verif` $\leftrightarrow$ `verif2` $\leftrightarrow$ `verif3` $\leftrightarrow \cdots \leftrightarrow$ `avx2`.

Don't try to force this chain to match the development path

`spec` $\rightarrow$ `opt` $\rightarrow$ `opt2` $\rightarrow$ `opt3` $\rightarrow$ `opt4` $\rightarrow$ `opt5` $\rightarrow \cdots \rightarrow$ `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.