Does cryptographic software work correctly?

1. The scale of the problem

Daniel J. Bernstein

University of Illinois at Chicago; Ruhr University Bochum

CVE-2018-0733, an OpenSSL bug

"Because of an implementation bug the PA-RISC `CRYPTO_memcmp` function is effectively reduced to only comparing the least significant bit of each byte." Bug introduced 2016.05.

CVE-2018-0733, an OpenSSL bug

"Because of an implementation bug the PA-RISC `CRYPTO_memcmp` function is effectively reduced to only comparing the least significant bit of each byte." Bug introduced 2016.05.

How severe is this? "This allows an attacker to forge messages that would be considered as authenticated in an amount of tries lower than that guaranteed by the security claims of the scheme."

CVE-2018-0733, an OpenSSL bug

“Because of an implementation bug the PA-RISC `CRYPTO_memcmp` function is effectively reduced to only comparing the least significant bit of each byte.” Bug introduced 2016.05.

How severe is this? “This allows an attacker to forge messages that would be considered as authenticated in an amount of tries lower than that guaranteed by the security claims of the scheme.”

— Yes, 2^{16} is “lower than” 2^{128}.

CVE-2017-3738, another OpenSSL bug

Don't care about PA-RISC? How about Intel?

"There is an overflow bug in the AVX2 Montgomery multiplication procedure used in exponentiation with 1024-bit moduli."
Bug introduced 2013.07.

CVE-2017-3738, another OpenSSL bug

Don't care about PA-RISC? How about Intel?

"There is an overflow bug in the AVX2 Montgomery multiplication procedure used in exponentiation with 1024-bit moduli."
Bug introduced 2013.07.

"Attacks against DH1024 are considered just feasible"

CVE-2017-3738, another OpenSSL bug

Don’t care about PA-RISC? How about Intel?

“There is an overflow bug in the AVX2 Montgomery multiplication procedure used in exponentiation with 1024-bit moduli.”
Bug introduced 2013.07.

“Attacks against DH1024 are considered just feasible”
— How much time? How much hardware?

CVE-2017-3738, continued

Are you safe if you aren't using DH1024? "Analysis suggests that attacks against RSA and DSA as a result of this defect would be very difficult to perform and are not believed likely."

CVE-2017-3738, continued

Are you safe if you aren't using DH1024? "Analysis suggests that attacks against RSA and DSA as a result of this defect would be very difficult to perform and are not believed likely."

— Really? How much public scrutiny
has the actual computation received from cryptanalysts?

CVE-2017-3738, continued

Are you safe if you aren't using DH1024? "Analysis suggests that attacks against RSA and DSA as a result of this defect would be very difficult to perform and are not believed likely."

— Really? How much public scrutiny
has the actual computation received from cryptanalysts?

What this looks like to me: "We have analyzed our new cryptosystem and concluded that attacks are not likely."

CVE-2017-3738, continued

Are you safe if you aren't using DH1024? "Analysis suggests that attacks against RSA and DSA as a result of this defect would be very difficult to perform and are not believed likely."

— Really? How much public scrutiny
has the actual computation received from cryptanalysts?

What this looks like to me: "We have analyzed our new cryptosystem and concluded that attacks are not likely."
— Don't we require attack analyses to be published and reviewed?

CVE-2017-3738, continued

Are you safe if you aren't using DH1024? "Analysis suggests that attacks against RSA and DSA as a result of this defect would be very difficult to perform and are not believed likely."

— Really? How much public scrutiny
has the actual computation received from cryptanalysts?

What this looks like to me: "We have analyzed our new cryptosystem and concluded that attacks are not likely."
— Don't we require attack analyses to be published and reviewed?

2019.12: Similar OpenSSL advisory for CVE-2019-1551.

Part of the CVE-2017-3738 patch

```
@@ -1093,7 +1093,9 @@
  vmovdqu  -8+32*2-128($ap),$TEMP2

  mov      $r1, %rax
+ vpblendd \$0xfc, $ZERO, $ACC9, $ACC9 # correct $ACC3
  imull    $n0, %eax
+ vpaddq   $ACC9,$ACC4,$ACC4           # correct $ACC3
  and      \$0x1fffffff, %eax

  imulq    16-128($ap),%rbx
@@ -1329,15 +1331,12 @@
```

2019.09: bug announced in Falcon software

“The consequences of these bugs are the following:

- Produced signatures were valid but **leaked information on the private key**. [emphasis added]
- Performance was artificially inflated: . . .

The fact that these bugs existed in the first place shows that the traditional development methodology (i.e. ‘being super careful’) has failed.”

2019.09: bug announced in Falcon software

"The consequences of these bugs are the following:

- Produced signatures were valid but **leaked information on the private key**. [emphasis added]
- Performance was artificially inflated: . . .

The fact that these bugs existed in the first place shows that the traditional development methodology (i.e. 'being super careful') has failed."

2018.01: Similar bug announced in Dilithium software (which "can easily be exploited to recover the secret key").

2019.09: bug announced in Falcon software

"The consequences of these bugs are the following:

- Produced signatures were valid but **leaked information on the private key**. [emphasis added]
- Performance was artificially inflated: . . .

The fact that these bugs existed in the first place shows that the traditional development methodology (i.e. 'being super careful') has failed."

2018.01: Similar bug announced in Dilithium software (which "can easily be exploited to recover the secret key").

2020.07: NIST post-quantum competition announces Dilithium and Falcon as the two lattice-based signature-system finalists.

Cryptography is notoriously hard to review

Mathematical complications in cryptography lead to subtle bugs.

Cryptography is notoriously hard to review

Mathematical complications in cryptography lead to subtle bugs.

Side-channel countermeasures add more complexity.

Cryptography is notoriously hard to review

Mathematical complications in cryptography lead to subtle bugs.

Side-channel countermeasures add more complexity.

Post-quantum cryptography: even more complex.

Cryptography is notoriously hard to review

Mathematical complications in cryptography lead to subtle bugs.

Side-channel countermeasures add more complexity.

Post-quantum cryptography: even more complex.

Cryptography is applied to large volumes of data.
Often individual cryptographic computations are time-consuming.
Pursuit of speed $\Rightarrow$ many different cryptographic systems, and
cryptographic code optimized in many ways for particular CPUs.

Cryptography is notoriously hard to review

Mathematical complications in cryptography lead to subtle bugs.

Side-channel countermeasures add more complexity.

Post-quantum cryptography: even more complex.

Cryptography is applied to large volumes of data.
Often individual cryptographic computations are time-consuming.
Pursuit of speed ⇒ many different cryptographic systems, and cryptographic code optimized in many ways for particular CPUs.

e.g. Keccak Code Package: >20 implementations of SHA-3.
e.g. Google added hand-written Cortex-A7 asm to Linux kernel for Speck128/128-XTS, then switched to (faster) Adiantum-XChaCha.

Is open-source software bug-free?

Eric S. Raymond, 1999: “Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone. Or, less formally, ‘Given enough eyeballs, all bugs are shallow.’ ”

Is open-source software bug-free?

Eric S. Raymond, 1999: “Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone. Or, less formally, ‘Given enough eyeballs, all bugs are shallow.’ ”

— “Beta-tester”: Ultimately, the unhappy user?

Is open-source software bug-free?

Eric S. Raymond, 1999: “Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone. Or, less formally, ‘Given enough eyeballs, all bugs are shallow.’ ”

— “Beta-tester”: Ultimately, the unhappy user?

— “Almost every problem”: That’s not “all bugs”!
Don’t we care about the exceptions, the bugs not found quickly?
Rare bugs can be devastating, especially for security!

More reasons for skepticism

— How do we know how many exceptions there are?
How many people are looking for unobvious bugs in our code?

More reasons for skepticism

— How do we know how many exceptions there are?
How many people are looking for unobvious bugs in our code?

— How can there be enough people looking for bugs
when most developers prefer writing new code?

More reasons for skepticism

— How do we know how many exceptions there are?
How many people are looking for unobvious bugs in our code?

— How can there be enough people looking for bugs
when most developers prefer writing new code?

— ESR advocates a development methodology
that releases a constant flood of new bugs.
Doesn't this make his "law" automatically true?
Is this the correctness metric that users want?

So we should use closed source?

“Closed source stops attackers from finding bugs.”

So we should use closed source?

"Closed source stops attackers from finding bugs."
— Serious attackers extract, disassemble, decompile the code, and understand it without our code comments, function names, etc.

So we should use closed source?

“Closed source stops attackers from finding bugs.”
— Serious attackers extract, disassemble, decompile the code, and understand it without our code comments, function names, etc.

“Closed source scares away some lazy academics, so we have fewer public bug announcements to deal with.”

So we should use closed source?

“Closed source stops attackers from finding bugs.”
— Serious attackers extract, disassemble, decompile the code, and understand it without our code comments, function names, etc.

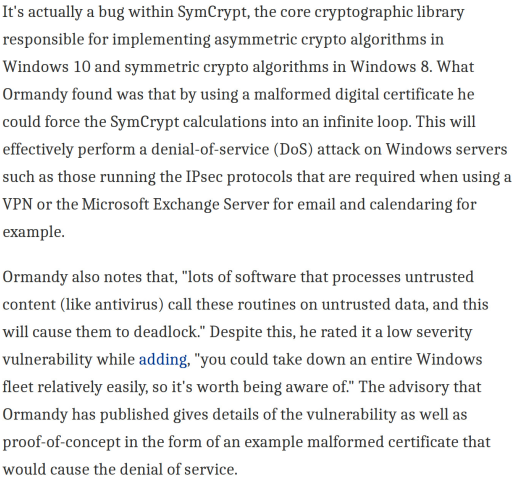
“Closed source scares away some lazy academics, so we have fewer public bug announcements to deal with.”
— Sounds plausible, but is the delay worthwhile?
e.g. Infineon deployed RSALib very widely before its keygen was broken by 2017 Nemec–Sys–Svenda–Klinec–Matyas “ROCA”.

Forbes Billionaires Innovation Leadership Money Business Small Business Lifes

210,878 views | Jun 12, 2019, 08:10am

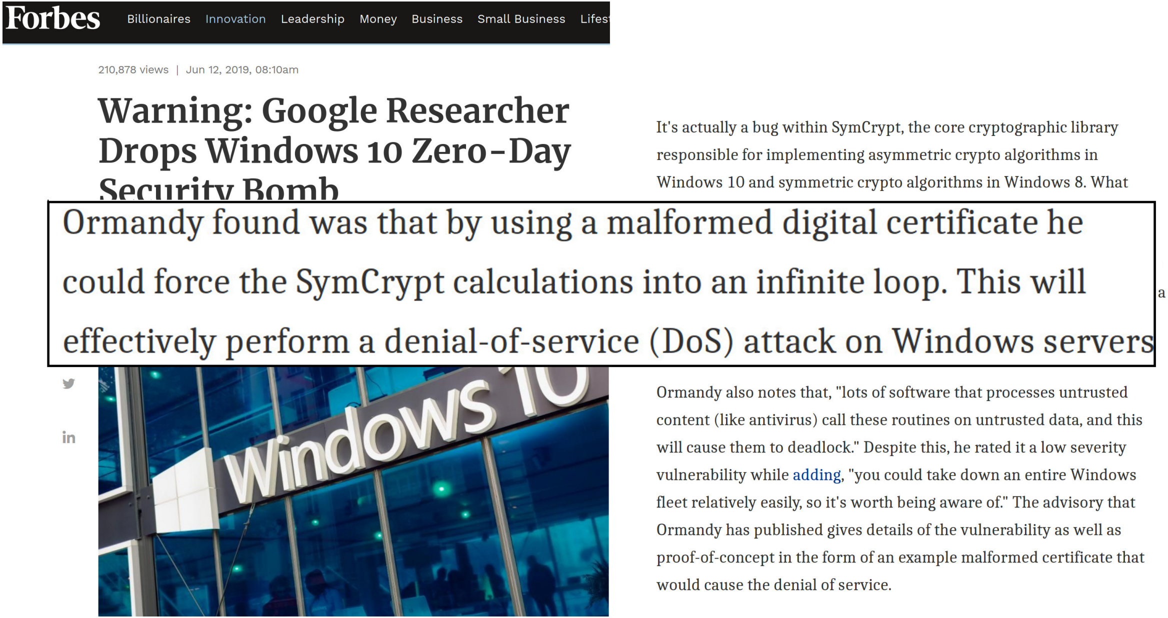
Warning: Google Researcher Drops Windows 10 Zero-Day Security Bomb

Davey Winder Senior Contributor
Cybersecurity
I report and analyse breaking cybersecurity and privacy stories



It's actually a bug within SymCrypt, the core cryptographic library responsible for implementing asymmetric crypto algorithms in Windows 10 and symmetric crypto algorithms in Windows 8. What Ormandy found was that by using a malformed digital certificate he could force the SymCrypt calculations into an infinite loop. This will effectively perform a denial-of-service (DoS) attack on Windows servers such as those running the IPsec protocols that are required when using a VPN or the Microsoft Exchange Server for email and calendaring for example.

Ormandy also notes that, "lots of software that processes untrusted content (like antivirus) call these routines on untrusted data, and this will cause them to deadlock." Despite this, he rated it a low severity vulnerability while adding, "you could take down an entire Windows fleet relatively easily, so it's worth being aware of." The advisory that Ormandy has published gives details of the vulnerability as well as proof-of-concept in the form of an example malformed certificate that would cause the denial of service.



210,878 views | Jun 12, 2019, 08:10am

Warning: Google Researcher Drops Windows 10 Zero-Day Security Bomb

It's actually a bug within SymCrypt, the core cryptographic library responsible for implementing asymmetric crypto algorithms in Windows 10 and symmetric crypto algorithms in Windows 8. What

Ormandy found was that by using a malformed digital certificate he could force the SymCrypt calculations into an infinite loop. This will effectively perform a denial-of-service (DoS) attack on Windows servers

a

Ormandy also notes that, "lots of software that processes untrusted content (like antivirus) call these routines on untrusted data, and this will cause them to deadlock." Despite this, he rated it a low severity vulnerability while adding, "you could take down an entire Windows fleet relatively easily, so it's worth being aware of." The advisory that Ormandy has published gives details of the vulnerability as well as proof-of-concept in the form of an example malformed certificate that would cause the denial of service.

Does cryptographic software work correctly?

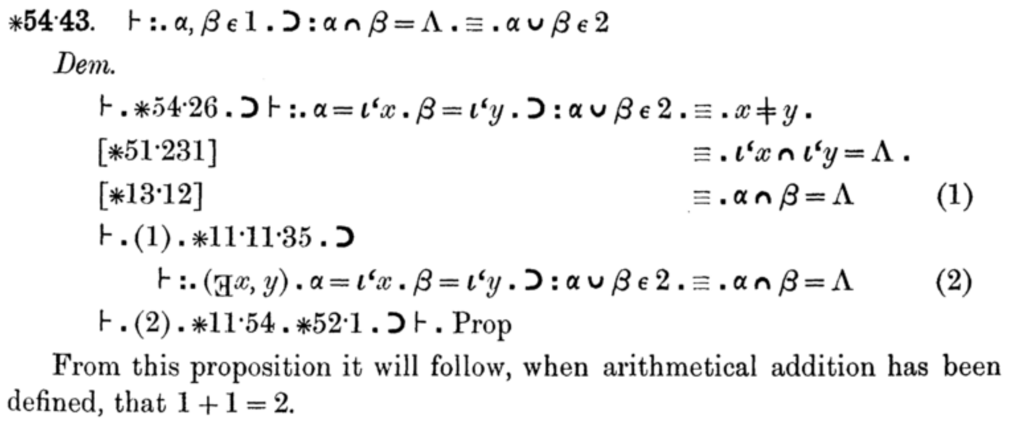
2. Computer-verified proofs

Daniel J. Bernstein

University of Illinois at Chicago; Ruhr University Bochum

Formal logic to the rescue?

Whitehead and Russell, *Principia Mathematica*, volume 1,
1st edition (1910), page 379:



∗54·43. $\vdash :. \alpha, \beta \in 1 . \supset : \alpha \cap \beta = \Lambda . \equiv . \alpha \cup \beta \in 2$

Dem.

$$\vdash . {*}54{\cdot}26 . \supset \vdash :. \alpha = \iota\text{‘}x . \beta = \iota\text{‘}y . \supset : \alpha \cup \beta \in 2 . \equiv . x \neq y .$$

$$[{*}51{\cdot}231] \qquad \equiv . \iota\text{‘}x \cap \iota\text{‘}y = \Lambda .$$

$$[{*}13{\cdot}12] \qquad \equiv . \alpha \cap \beta = \Lambda \qquad (1)$$

$$\vdash . (1) . {*}11{\cdot}11{\cdot}35 . \supset$$

$$\vdash :. (\exists x, y) . \alpha = \iota\text{‘}x . \beta = \iota\text{‘}y . \supset : \alpha \cup \beta \in 2 . \equiv . \alpha \cap \beta = \Lambda \qquad (2)$$

$$\vdash . (2) . {*}11{\cdot}54 . {*}52{\cdot}1 . \supset \vdash . \text{Prop}$$

From this proposition it will follow, when arithmetical addition has been defined, that $1 + 1 = 2$.

Formal verification today

Require code reviewer to *prove* correctness.

Require proofs to pass a proof-checking computer program.

Formal verification today

Require code reviewer to *prove* correctness.
Require proofs to pass a proof-checking computer program.

Mathematicians rarely use these proof-checking tools today.
Proving crypto code correct is tedious.

Formal verification today

Require code reviewer to *prove* correctness.
Require proofs to pass a proof-checking computer program.

Mathematicians rarely use these proof-checking tools today.
Proving crypto code correct is tedious. But not impossible!
Latest EverCrypt release: verified software for Curve25519, Ed25519, ChaCha20, Poly1305, AES-CTR (if CPU has AES-NI), AES-GCM (same), MD5, SHA-1, SHA-2, SHA-3, BLAKE2.

Formal verification today

Require code reviewer to *prove* correctness.
Require proofs to pass a proof-checking computer program.

Mathematicians rarely use these proof-checking tools today.
Proving crypto code correct is tedious. But not impossible!
Latest EverCrypt release: verified software for Curve25519, Ed25519, ChaCha20, Poly1305, AES-CTR (if CPU has AES-NI), AES-GCM (same), MD5, SHA-1, SHA-2, SHA-3, BLAKE2.

Good: High confidence that subtle bugs are gone
(in the code; but worry about bugs in compiler, CPU, . . .).

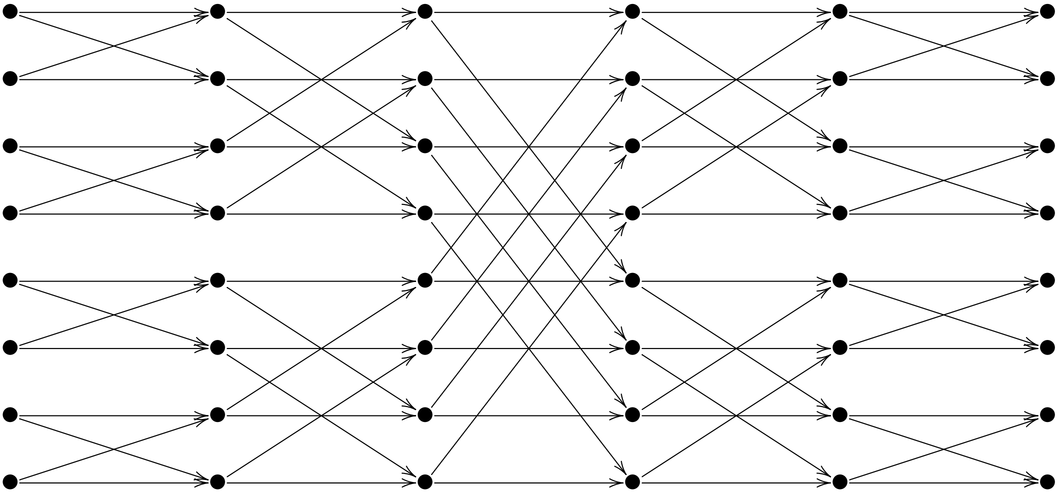
Formal verification today

Require code reviewer to *prove* correctness.
Require proofs to pass a proof-checking computer program.

Mathematicians rarely use these proof-checking tools today.
Proving crypto code correct is tedious. But not impossible!
Latest EverCrypt release: verified software for Curve25519, Ed25519, ChaCha20, Poly1305, AES-CTR (if CPU has AES-NI), AES-GCM (same), MD5, SHA-1, SHA-2, SHA-3, BLAKE2.

Good: High confidence that subtle bugs are gone
(in the code; but worry about bugs in compiler, CPU, . . .).
Bad: Tons of effort for each implementation.
e.g. EverCrypt doesn't have fast software for smartphone CPUs.

Case study: Beneš networks



Computing control bits for Beneš networks

Long literature on Beneš networks. Energy-efficient. Low latency.

Computing control bits for Beneš networks

Long literature on Beneš networks. Energy-efficient. Low latency.

1968 Stone: Fast algorithm that, given a permutation of 2^m inputs, computes Beneš-network control bits applying that permutation.

Computing control bits for Beneš networks

Long literature on Beneš networks. Energy-efficient. Low latency.

1968 Stone: Fast algorithm that, given a permutation of 2^m inputs, computes Beneš-network control bits applying that permutation.

1981 Lev–Pippenger–Valiant, 1982 Nassimi–Sahni, 1996 Lee–Liew, etc.: Fast parallel algorithms to compute control bits.

Computing control bits for Beneš networks

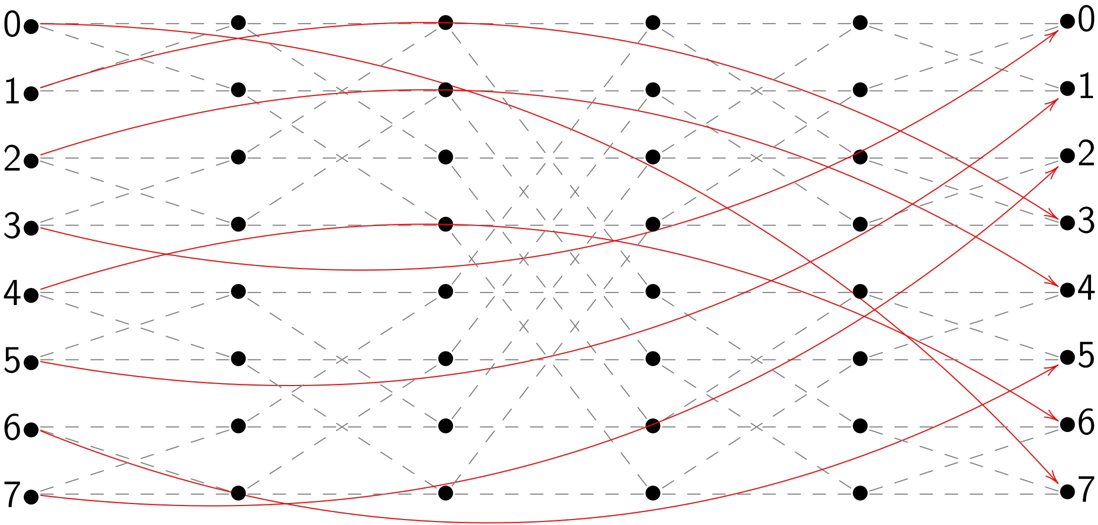
Long literature on Beneš networks. Energy-efficient. Low latency.

1968 Stone: Fast algorithm that, given a permutation of 2^m inputs, computes Beneš-network control bits applying that permutation.

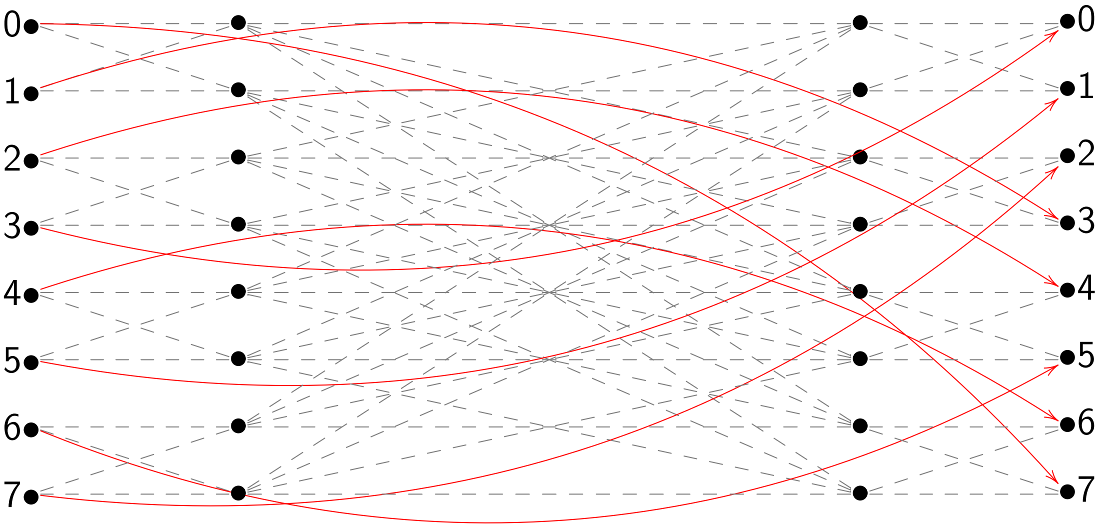
1981 Lev–Pippenger–Valiant, 1982 Nassimi–Sahni, 1996 Lee–Liew, etc.: Fast parallel algorithms to compute control bits.

Post-quantum crypto (e.g., Classic McEliece) uses fast constant-time software to compute and apply control bits.
Is this software always computing the right control bits?

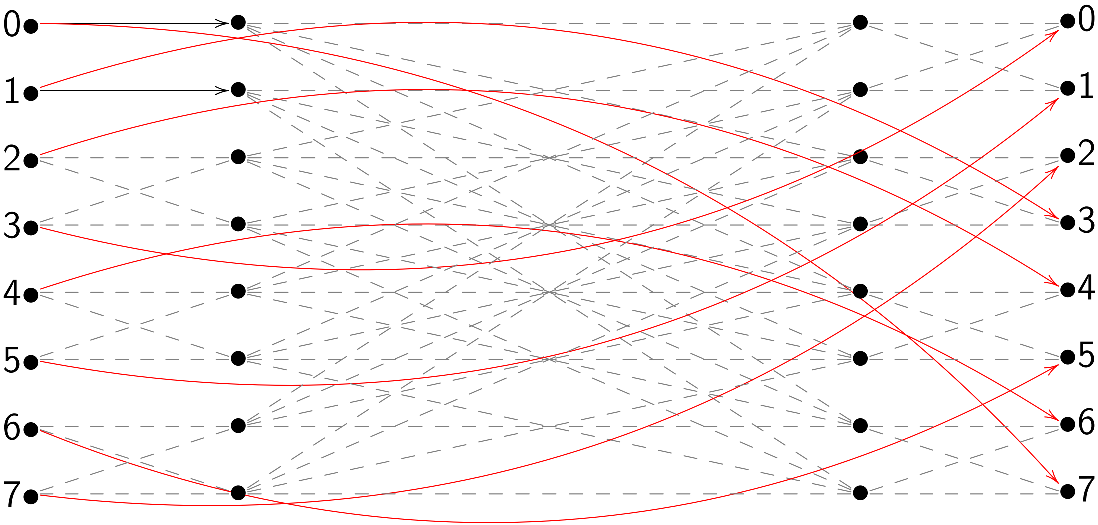
Stone's algorithm



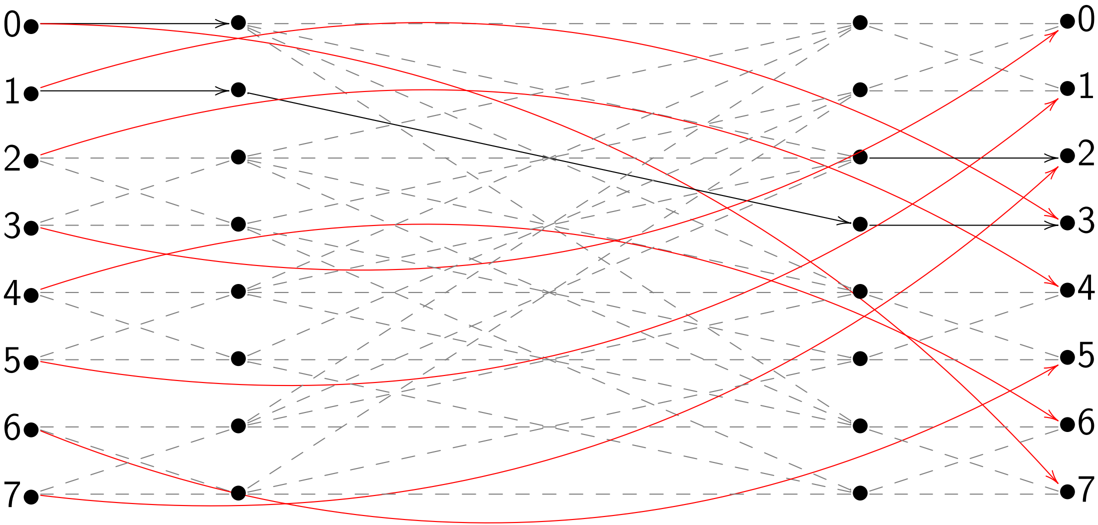
Stone's algorithm



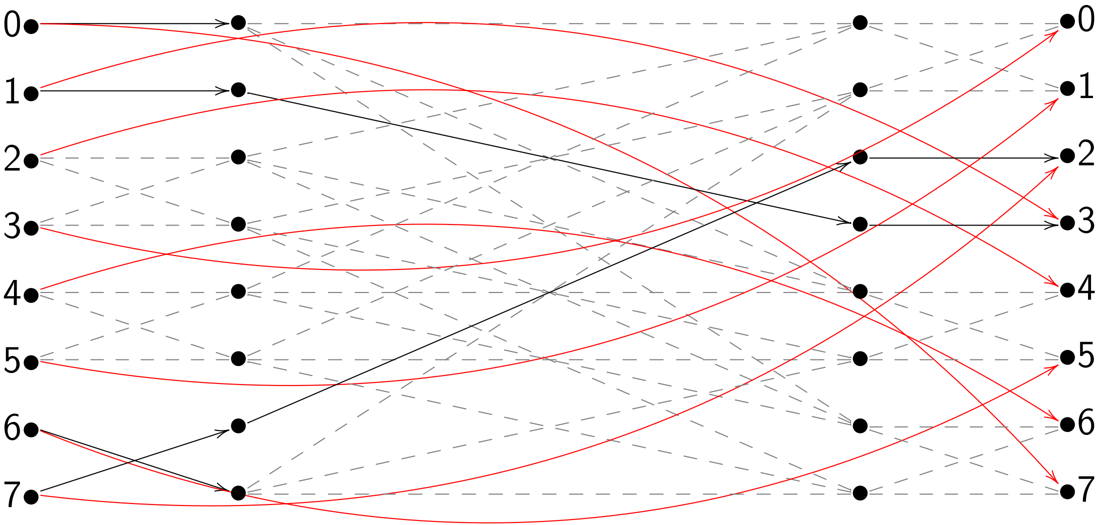
Stone's algorithm



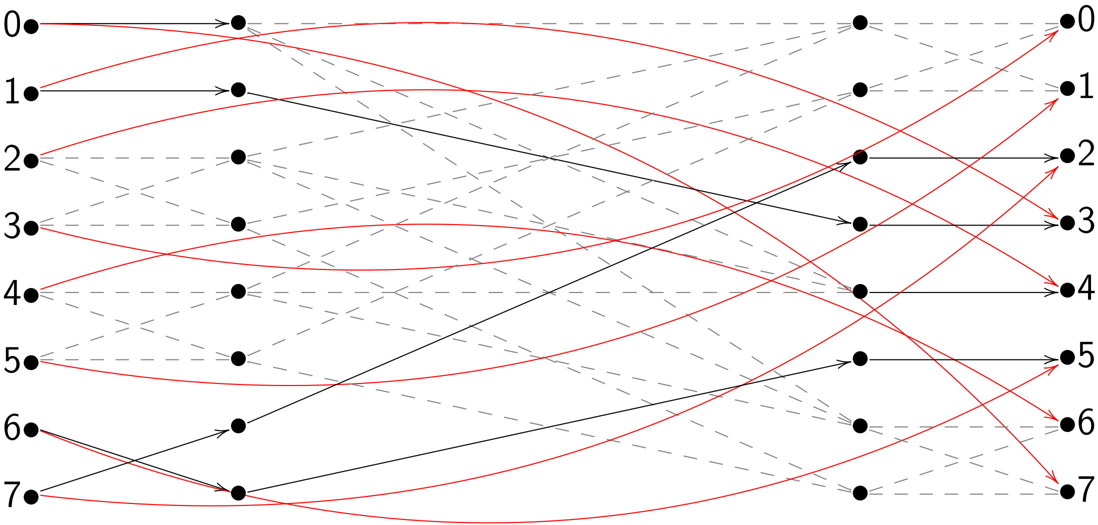
Stone's algorithm



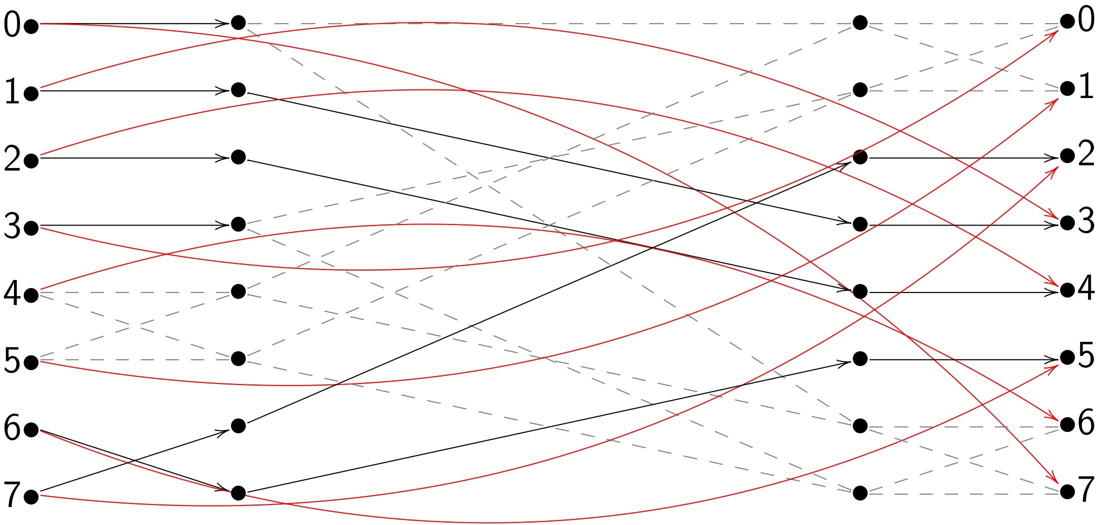
Stone's algorithm



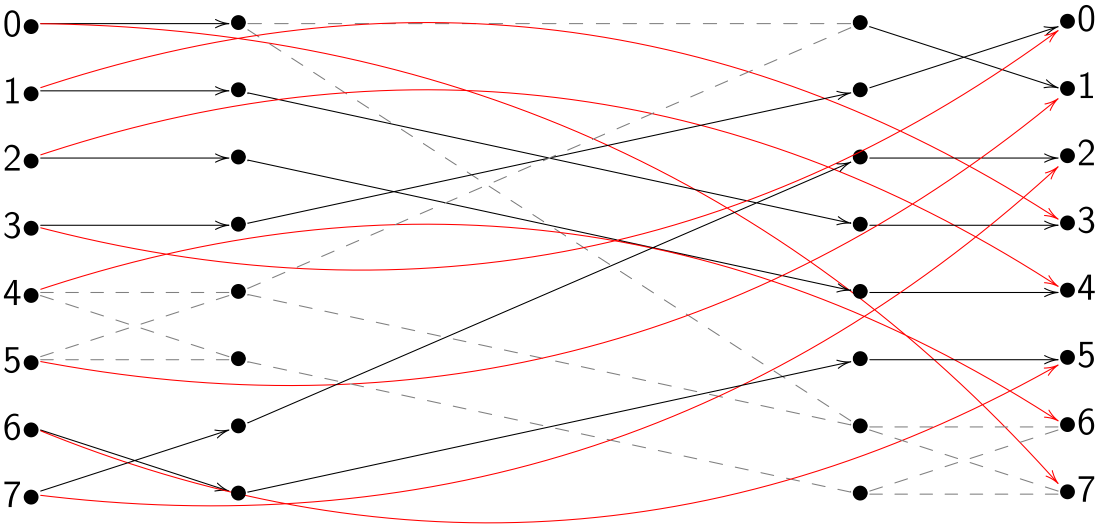
Stone's algorithm



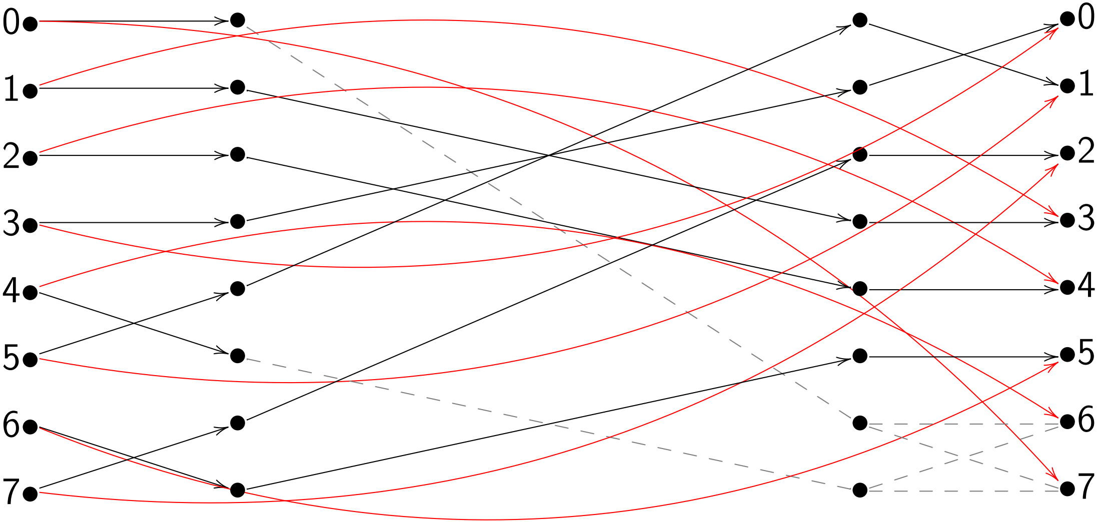
Stone's algorithm



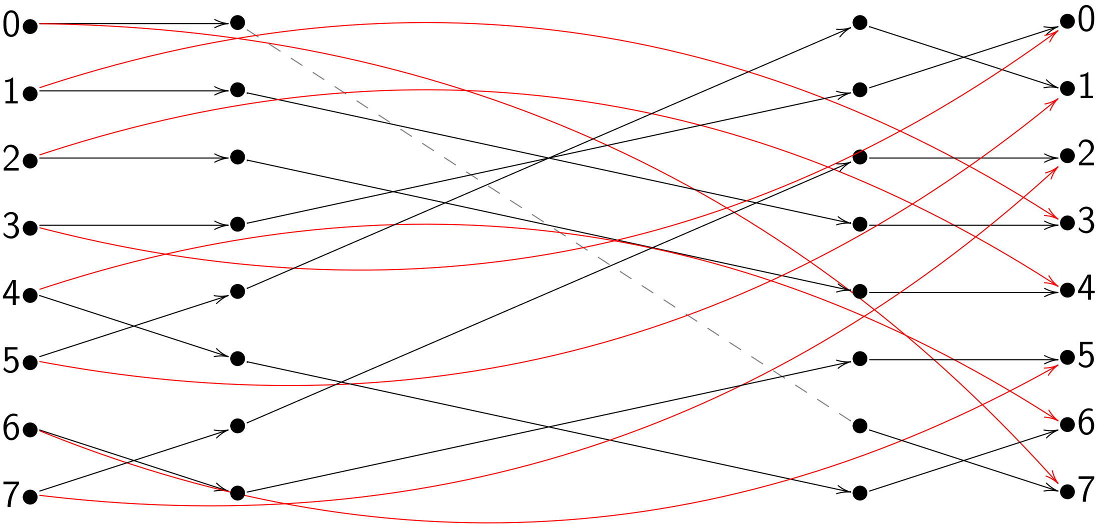
Stone's algorithm



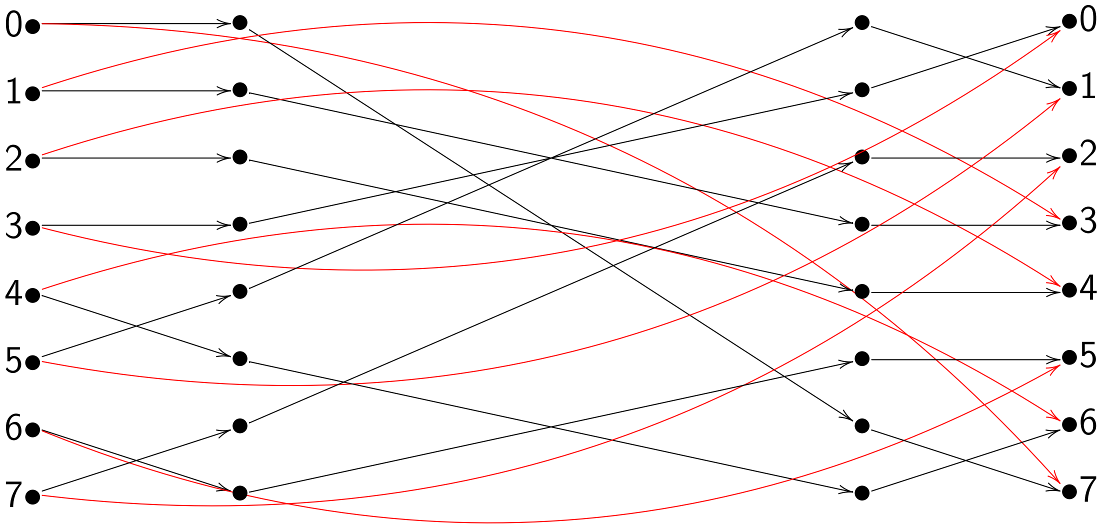
Stone's algorithm



Stone's algorithm



Stone's algorithm



Control-bit formulas

"Verified fast formulas for control bits for permutation networks", `https://cr.yp.to/papers.html#controlbits`:

Start with any permutation π of $\{0, 1, \ldots, 2b-1\}$.

Compute first control bits $f_0, f_1, \ldots, f_{b-1}$ and last control bits $\ell_0, \ell_1, \ldots, \ell_{b-1}$ according to particular formulas in terms of π.

Define $F(x) = x \oplus f_{\lfloor x/2 \rfloor}$; $L(x) = x \oplus \ell_{\lfloor x/2 \rfloor}$; $M(x) = F(\pi(L(x)))$.

Control-bit formulas

"Verified fast formulas for control bits for permutation networks", `https://cr.yp.to/papers.html#controlbits`:

Start with any permutation π of $\{0, 1, \ldots, 2b - 1\}$.

Compute first control bits $f_0, f_1, \ldots, f_{b-1}$ and last control bits $\ell_0, \ell_1, \ldots, \ell_{b-1}$ according to particular formulas in terms of π.

Define $F(x) = x \oplus f_{\lfloor x/2 \rfloor}$; $L(x) = x \oplus \ell_{\lfloor x/2 \rfloor}$; $M(x) = F(\pi(L(x)))$.

Pages 4–7 of paper: Detailed math proof that $M(x) \equiv x \pmod 2$.

Control-bit formulas

"Verified fast formulas for control bits for permutation networks", https://cr.yp.to/papers.html#controlbits:

Start with any permutation π of $\{0, 1, \ldots, 2b - 1\}$.

Compute first control bits $f_0, f_1, \ldots, f_{b-1}$ and last control bits $\ell_0, \ell_1, \ldots, \ell_{b-1}$ according to particular formulas in terms of π.

Define $F(x) = x \oplus f_{\lfloor x/2 \rfloor}$; $L(x) = x \oplus \ell_{\lfloor x/2 \rfloor}$; $M(x) = F(\pi(L(x)))$.

Pages 4–7 of paper: Detailed math proof that $M(x) \equiv x \pmod 2$.
Pages 21–66 of paper: Proof verified by HOL Light.

Verifying claimed theorems in HOL Light

In a new Debian Stretch VM: `# apt install git make camlp5`

As a new user, download and compile HOL Light:

```
$ git clone https://github.com/jrh13/hol-light.git
$ cd hol-light; make
```

Download someone's claimed HOL Light theorems: e.g.,

```
$ wget https://cr.yp.to/2020/controlbits-20200923.ml
```

Start HOL Light (takes a few minutes to verify built-in theorems):

```
$ ocaml
# #use "hol.ml";;
```

Ask HOL Light to verify the claimed theorems:

```
# #use "controlbits-20200923.ml";;
```

Defining a mathematical function in HOL Light

```
let xor1 = new_definition
  `xor1 (n:num) = if EVEN n then n+1 else n-1`;;
```

i.e. `xor1(0)` is 1; `xor1(1)` is 0; `xor1(2)` is 3; `xor1(3)` is 2; etc.

Defining a mathematical function in HOL Light

```
let xor1 = new_definition
  `xor1 (n:num) = if EVEN n then n+1 else n-1`;;
```

i.e. `xor1(0)` is 1; `xor1(1)` is 0; `xor1(2)` is 3; `xor1(3)` is 2; etc.

`num` means nonnegative integers: $\{0, 1, 2, \ldots\}$.
`EVEN n` means True (`T`) if `n` is even, else False (`F`).
`n+1` means what you think it means.

Defining a mathematical function in HOL Light

```
let xor1 = new_definition
  `xor1 (n:num) = if EVEN n then n+1 else n-1`;;
```

i.e. `xor1(0)` is 1; `xor1(1)` is 0; `xor1(2)` is 3; `xor1(3)` is 2; etc.

`num` means nonnegative integers: $\{0, 1, 2, \ldots\}$.
`EVEN n` means True (`T`) if `n` is even, else False (`F`).
`n+1` means what you think it means.

Warning: `n-1` doesn't mean exactly what you think it means.
If `n` is `0:num` then `n-1` is `0`. Error-prone definition of `-`. Yikes!
Analogy: `+` on `int` in C isn't math $+$ on integers; can overflow.

Quantifiers in HOL Light

"f is an involution" means: every x has $f(f(x)) = x$.

```
let involution = new_definition
  `involution (f:A->A) <=> !x. f(f x) = x`;;
```

Quantifiers in HOL Light

“f is an involution” means: every x has $f(f(x)) = x$.

```
let involution = new_definition
  `involution (f:A->A) <=> !x. f(f x) = x`;;
```

`f:A->A` is a function from `A` to `A`. Can write `f x` for `f(x)`.

Quantifiers in HOL Light

"f is an involution" means: every x has $f(f(x)) = x$.

```
let involution = new_definition
  `involution (f:A->A) <=> !x. f(f x) = x`;;
```

`f:A->A` is a function from `A` to `A`. Can write `f x` for `f(x)`.

`!x` in HOL Light means "for all `x` of this type".
HOL Light type-checker automatically chooses type of `x` as `A`
since `x` is an `f` input (and an `f` output). Or can write `!x:A`.

Quantifiers in HOL Light

“f is an involution” means: every x has $f(f(x)) = x$.

```
let involution = new_definition
  `involution (f:A->A) <=> !x. f(f x) = x`;;
```

`f:A->A` is a function from `A` to `A`. Can write `f x` for `f(x)`.

`!x` in HOL Light means “for all `x` of this type”.
HOL Light type-checker automatically chooses type of `x` as `A`
since `x` is an `f` input (and an `f` output). Or can write `!x:A`.

In `xor1` definition could have written `xor1 n = ...`.
Type-checker would have assumed `num` since `EVEN` wants a `num`.

Quantifiers in HOL Light

"f is an involution" means: every x has $f(f(x)) = x$.

```
let involution = new_definition
  `involution (f:A->A) <=> !x. f(f x) = x`;;
```

`f:A->A` is a function from `A` to `A`. Can write `f x` for `f(x)`.

`!x` in HOL Light means "for all `x` of this type".
HOL Light type-checker automatically chooses type of `x` as `A`
since `x` is an `f` input (and an `f` output). Or can write `!x:A`.

In `xor1` definition could have written `xor1 n = ....`
Type-checker would have assumed `num` since `EVEN` wants a `num`.
Can even say `involution f = ...`; type-checker will invent an `A`.

Verified theorems in HOL Light: `thm`

```
# xor1_involution;;
val it : thm = |- involution xor1
```

Verified theorems in HOL Light: `thm`

```
# xor1_involution;;
val it : thm = |- involution xor1
```

Always carefully check theorem statements and definitions: e.g.,

```
# xor1;;
val it : thm = |- !n. xor1 n =
  (if EVEN n then n + 1 else n - 1)
```

Verified theorems in HOL Light: `thm`

```
# xor1_involution;;
val it : thm = |- involution xor1
```

Always carefully check theorem statements and definitions: e.g.,

```
# xor1;;
val it : thm = |- !n. xor1 n =
  (if EVEN n then n + 1 else n - 1)
```

Also check (before running it!) that `controlbits-20200923.ml` didn't override HOL Light.

Verified theorems in HOL Light: `thm`

```
# xor1_involution;;
val it : thm = |- involution xor1
```

Always carefully check theorem statements and definitions: e.g.,

```
# xor1;;
val it : thm = |- !n. xor1 n =
  (if EVEN n then n + 1 else n - 1)
```

Also check (before running it!) that `controlbits-20200923.ml` didn't override HOL Light. Harder: check OCaml, gcc, OS, CPU.

Proving theorems in HOL Light

Somewhere inside `controlbits-20200923.ml`:

```
let xor1_involution = prove(
  `involution xor1`,
  MESON_TAC[xor1xor1;involution]);;
```

`MESON_TAC`: “model elimination subgoal oriented” theorem-proving tactic . . . meaning: this follows trivially.

Proving theorems in HOL Light

Somewhere inside `controlbits-20200923.ml`:

```
let xor1_involution = prove(
  `involution xor1`,
  MESON_TAC[xor1xor1;involution]);;
```

`MESON_TAC`: “model elimination subgoal oriented” theorem-proving tactic . . . meaning: this follows trivially.

```
# involution;;
val it : thm = |- !f. involution f <=> (!x. f (f x) = x)
```

Proving theorems in HOL Light

Somewhere inside `controlbits-20200923.ml`:

```
let xor1_involution = prove(
  `involution xor1`,
  MESON_TAC[xor1xor1;involution]);;
```

`MESON_TAC`: "model elimination subgoal oriented" theorem-proving tactic . . . meaning: this follows trivially.

```
# involution;;
val it : thm = |- !f. involution f <=> (!x. f (f x) = x)
# xor1xor1;;
val it : thm = |- !n. xor1 (xor1 n) = n
```

Proving theorems in HOL Light, continued

```
let xor1xor1 = prove(
  `!n. xor1(xor1 n) = n`,
  MESON_TAC[xor1xor1_ifodd;xor1xor1_ifeven;EVEN_OR_ODD]);;
```

Proving theorems in HOL Light, continued

```
let xor1xor1 = prove(
  `!n. xor1(xor1 n) = n`,
  MESON_TAC[xor1xor1_ifodd;xor1xor1_ifeven;EVEN_OR_ODD]);;

# EVEN_OR_ODD;;
val it : thm = |- !n. EVEN n \/ ODD n
```

Proving theorems in HOL Light, continued

```
let xor1xor1 = prove(
  `!n. xor1(xor1 n) = n`,
  MESON_TAC[xor1xor1_ifodd;xor1xor1_ifeven;EVEN_OR_ODD]);;
```

```
# EVEN_OR_ODD;;
val it : thm = |- !n. EVEN n \/ ODD n
# xor1xor1_ifeven;;
val it : thm = |- !n. EVEN n ==> xor1 (xor1 n) = n
```

Proving theorems in HOL Light, continued

```
let xor1xor1 = prove(
  `!n. xor1(xor1 n) = n`,
  MESON_TAC[xor1xor1_ifodd;xor1xor1_ifeven;EVEN_OR_ODD]);;

# EVEN_OR_ODD;;
val it : thm = |- !n. EVEN n \/ ODD n
# xor1xor1_ifeven;;
val it : thm = |- !n. EVEN n ==> xor1 (xor1 n) = n
# xor1xor1_ifodd;;
val it : thm = |- !n. ODD n ==> xor1 (xor1 n) = n
```

Sometimes proofs feel a bit more complicated

```
let pow_num_bijection = prove(
  `!p:A->A. bijection p ==> !n. bijection (p pow_num n)`,
  GEN_TAC THEN DISCH_TAC THEN
  INDUCT_TAC THENL
  [ REWRITE_TAC[pow_num_0;bijection_I]
  ; REWRITE_TAC[suc_isadd1] THEN
    ASM_MESON_TAC[pow_num_plus1;bijection_composes]
  ]);;
```

So we're done?

```
# middleperm_parity;;
val it : thm = |- !p x. bijection p ==>
  (ODD (middleperm p x) <=> ODD x)
```

So we know $M(x) \equiv x \pmod 2$.

So we're done?

```
# middleperm_parity;;
val it : thm = |- !p x. bijection p ==>
   (ODD (middleperm p x) <=> ODD x)
```

So we know $M(x) \equiv x \pmod{2}$. With marginally more effort:
$\pi \mapsto$ full sequence of control bits $\mapsto$ Beneš network $\mapsto$ same π.

So we're done?

```
# middleperm_parity;;
val it : thm = |- !p x. bijection p ==>
  (ODD (middleperm p x) <=> ODD x)
```

So we know $M(x) \equiv x \pmod 2$. With marginally more effort: $\pi \mapsto$ full sequence of control bits $\mapsto$ Beneš network $\mapsto$ same π.

What we actually want to know: this **software** is computing the same control bits, and this **software** is then applying the same π.

"Software" includes Python script in paper; reference C code; gcc output from the C code; optimized assembly language; etc.

Solution: More proofs?

CompCert is a compiler with

- a formal definition of a C-like input language;
- a formal definition of (e.g.) an “ARM assembly language” (at least some instructions), maybe perfectly matching ARM;
- a formally verified proof that, for each input program, the output program is equivalent to the input program.

Solution: More proofs?

CompCert is a compiler with

- a formal definition of a C-like input language;
- a formal definition of (e.g.) an “ARM assembly language” (at least some instructions), maybe perfectly matching ARM;
- a formally verified proof that, for each input program, the output program is equivalent to the input program.

So: write C-like code, prove it applies π. Compile with CompCert.

Solution: More proofs?

CompCert is a compiler with

- a formal definition of a C-like input language;
- a formal definition of (e.g.) an “ARM assembly language” (at least some instructions), maybe perfectly matching ARM;
- a formally verified proof that, for each input program, the output program is equivalent to the input program.

So: write C-like code, prove it applies π. Compile with CompCert.

Oops: the output is too slow, and have to pay to use CompCert.

Solution: More proofs?

CompCert is a compiler with

- a formal definition of a C-like input language;
- a formal definition of (e.g.) an “ARM assembly language” (at least some instructions), maybe perfectly matching ARM;
- a formally verified proof that, for each input program, the output program is equivalent to the input program.

So: write C-like code, prove it applies π. Compile with CompCert.

Oops: the output is too slow, and have to pay to use CompCert.

So: write assembly, prove it applies π.

Solution: More proofs?

CompCert is a compiler with

- a formal definition of a C-like input language;
- a formal definition of (e.g.) an "ARM assembly language" (at least some instructions), maybe perfectly matching ARM;
- a formally verified proof that, for each input program, the output program is equivalent to the input program.

So: write C-like code, prove it applies π. Compile with CompCert.

Oops: the output is too slow, and have to pay to use CompCert.

So: write assembly, prove it applies π. Feasible? Yes.

Solution: More proofs?

CompCert is a compiler with

- a formal definition of a C-like input language;
- a formal definition of (e.g.) an “ARM assembly language” (at least some instructions), maybe perfectly matching ARM;
- a formally verified proof that, for each input program, the output program is equivalent to the input program.

So: write C-like code, prove it applies π. Compile with CompCert.

Oops: the output is too slow, and have to pay to use CompCert.

So: write assembly, prove it applies π. Feasible? Yes. Tedious? Yes.

Does cryptographic software work correctly?

3. Symbolic testing

Daniel J. Bernstein

University of Illinois at Chicago; Ruhr University Bochum

Testing

Testing is great. Test everything. Design for tests.

Why wasn't the PA-RISC `CRYPTO_memcmp` software in OpenSSL run through millions of tests on random inputs?
And tests on inputs differing in just a few positions?
SUPERCOP crypto test framework has always done this.

Testing

Testing is great. Test everything. Design for tests.

Why wasn't the PA-RISC `CRYPTO_memcmp` software in OpenSSL
run through millions of tests on random inputs?
And tests on inputs differing in just a few positions?
SUPERCOP crypto test framework has always done this.

Good reaction to a bug:
"How can I build fast automated tests to catch this kind of bug?"
Even better to ask question before bug happens.

The most important complaint about testing

Testing can miss attacker-triggerable bugs for rare inputs.

The most important complaint about testing

Testing can miss attacker-triggerable bugs for rare inputs.

e.g. 2019.11 paper from Nath and Sarkar points out
bugs with probability $\approx 1/2^{64}$ in the fastest code for Curve448:

“On certain kinds of inputs, the code will lead to overflow conditions and hence to incorrect results.

The most important complaint about testing

Testing can miss attacker-triggerable bugs for rare inputs.

e.g. 2019.11 paper from Nath and Sarkar points out bugs with probability $\approx 1/2^{64}$ in the fastest code for Curve448:

"On certain kinds of inputs, the code will lead to overflow conditions and hence to incorrect results. This, however, is a very low probability event and cannot be captured using some randomly generated known answer tests (KATs). . . .

The most important complaint about testing

Testing can miss attacker-triggerable bugs for rare inputs.

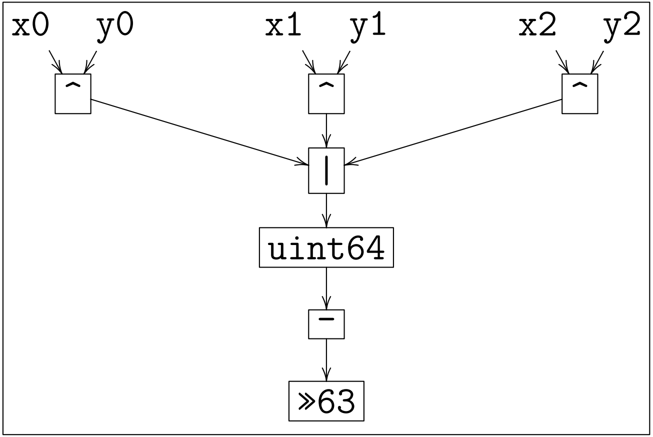
e.g. 2019.11 paper from Nath and Sarkar points out bugs with probability $\approx 1/2^{64}$ in the fastest code for Curve448:

"On certain kinds of inputs, the code will lead to overflow conditions and hence to incorrect results. This, however, is a very low probability event and cannot be captured using some randomly generated known answer tests (KATs). . . . We believe that it is important to have proofs of correctness of the reduction algorithms to ensure that the algorithms works correctly for all possible inputs."

Symbolic testing: beyond testing particular inputs

```
.globl CRYPTO_memcmp
CRYPTO_memcmp:
xor    %rax,%rax
xor    %r10,%r10
cmp    $0x0,%rdx
je     no_data
cmp    $0x10,%rdx
jne    loop
mov    (%rdi),%r10
mov    0x8(%rdi),%r11
mov    $0x1,%rdx
xor    (%rsi),%r10
xor    0x8(%rsi),%r11
or     %r11,%r10
cmovne %rdx,%rax
repz retq
loop:
mov    (%rdi),%r10b
lea    0x1(%rdi),%rdi
xor    (%rsi),%r10b
lea    0x1(%rsi),%rsi
or     %r10b,%al
dec    %rdx
jne    loop
neg    %rax
shr    $0x3f,%rax
no_data:
repz retq
```

Arithmetic DAG for all 3-byte inputs:



The power of modern reverse-engineering tools

Easy to use `angr.io` for automatic **symbolic execution**:
machine-language software ➜ arithmetic DAG.
Simplifies analysis: simpler instructions, no memory, no jumps.

The power of modern reverse-engineering tools

Easy to use `angr.io` for automatic **symbolic execution**:
machine-language software ➜ arithmetic DAG.
Simplifies analysis: simpler instructions, no memory, no jumps.

Limitation, sometimes exponential blowup: `angr` splits universes whenever it reaches an input-dependent branch or address.
. . . which we try to avoid in crypto anyway.

The power of modern reverse-engineering tools

Easy to use `angr.io` for automatic **symbolic execution**:
machine-language software ➜ arithmetic DAG.
Simplifies analysis: simpler instructions, no memory, no jumps.

Limitation, sometimes exponential blowup: `angr` splits universes whenever it reaches an input-dependent branch or address.
. . . which we try to avoid in crypto anyway.

`angr` (via Z3 SMT solver) often sees equivalence of small DAGs.
e.g. sees that OpenSSL `x86_64` `CRYPTO_memcmp` on 3-byte inputs outputs 0 if `x0==y0` and `x1==y1` and `x2==y2`,
and outputs 1 otherwise. Similarly for other input lengths.

```c
#include <openssl/crypto.h>

unsigned char x[N];
unsigned char y[N];
int z;

int main()
{
  z = CRYPTO_memcmp(x,y,N);
  return 0;
}
```

```
#!/usr/bin/env python3

import sys
import angr

N = int(sys.argv[1]) if len(sys.argv) > 1 else 16

proj = angr.Project(’cmp%d’%N)
state = proj.factory.full_init_state()

state.options |= {
  angr.options.ZERO_FILL_UNCONSTRAINED_MEMORY
}
```

```
x = {}
xaddr = proj.loader.find_symbol('x').rebased_addr
for i in range(N):
  x[i] = state.solver.BVS('x%d'%i,8)
  state.mem[xaddr+i].char = x[i]

y = {}
yaddr = proj.loader.find_symbol('y').rebased_addr
for i in range(N):
  y[i] = state.solver.BVS('y%d'%i,8)
  state.mem[yaddr+i].char = y[i]

simgr = proj.factory.simgr(state)
simgr.run()
```

```
assert len(simgr.errored) == 0
print(’%d universes’ % len(simgr.deadended))
for exit in simgr.deadended:
  zaddr = proj.loader.find_symbol(’z’).rebased_addr
  z = exit.mem[zaddr].int.resolved
  print(’out = %s’ % z)

  xeqy = True
  for i in range(N):
    xeqy = state.solver.And(xeqy,x[i]==y[i])
  xney = state.solver.Not(xeqy)
  for bugs in ((z!=0,z!=1),(z!=0,xeqy),(z!=1,xney)):
    assert not exit.satisfiable(extra_constraints=bugs)
```

Symbolic execution with better equivalence testing

What if the DAG is too complicated for the SMT solver?
Answer: **Build smarter tools to recognize DAG equivalence.**

Symbolic execution with better equivalence testing

What if the DAG is too complicated for the SMT solver?
Answer: **Build smarter tools to recognize DAG equivalence.**

Case study, software library from `sorting.cr.yp.to`:

- New speed records for sorting of in-memory integer arrays. This is a subroutine in some post-quantum cryptosystems.
- Side-channel countermeasures: no secret branch conditions; no secret array indices.
- New tool verifies correct sorting of all size-N inputs. No need for manual review of per-CPU optimized code.