Verified compilation of space-efficient reversible circuits

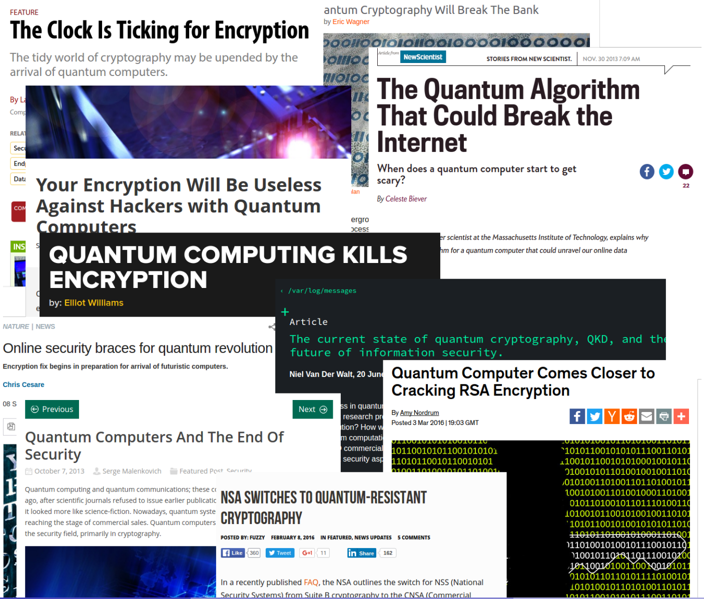
Matthew Amy[1] Martin Roetteler[2] Krysta Svore[2]

[1] University of Waterloo
[2] Microsoft Research

July 25, 2017

Quantum computing

Theory:



FEATURE

The Clock Is Ticking for Encryption

The tidy world of cryptography may be upended by the arrival of quantum computers.

antum Cryptography Will Break The Bank

by Eric Wagner

NewScientist STORIES FROM NEW SCIENTIST. NOV. 30 2013 7:09 AM

The Quantum Algorithm That Could Break the Internet

When does a quantum computer start to get scary?

By Celeste Biever

22

er scientist at the Massachusetts Institute of Technology, explains why

hm for a quantum computer that could unravel our online data

Your Encryption Will Be Useless Against Hackers with Quantum Computers

QUANTUM COMPUTING KILLS ENCRYPTION

by: Elliot Williams

‹ /var/log/messages

\+ Article

The current state of quantum cryptography, QKD, and the future of information security.

Niel Van Der Walt, 20 June

NATURE | NEWS

Online security braces for quantum revolution

Encryption fix begins in preparation for arrival of futuristic computers.

Chris Cesare

Previous Next

Quantum Computers And The End Of Security

October 7, 2013 Serge Malenkovich Featured Post, Security

Quantum computing and quantum communications; these c
ago, after scientific journals refused to issue earlier publicatic
it looked more like science-fiction. Nowadays, quantum syste
reaching the stage of commercial sales. Quantum computers
the security field, primarily in cryptography.

Quantum Computer Comes Closer to Cracking RSA Encryption

By Amy Nordrum

Posted 3 Mar 2016 | 19:03 GMT

NSA SWITCHES TO QUANTUM-RESISTANT CRYPTOGRAPHY

POSTED BY: FUZZY FEBRUARY 8, 2016 IN FEATURED, NEWS UPDATES 5 COMMENTS

Like 360 Tweet G+1 11 Share 162

In a recently published FAQ, the NSA outlines the switch for NSS (National Security Systems) from Suite B cryptography to the CNSA (Commercial

Quantum computing

Reality:

> *Quantum computing is weakened by the high degree of overhead required to perform classical computations reversibly (and to correct errors)*

Reversible computing

Every operation must be invertible

- $x \wedge y = 0 \implies x = ???, y = ???$
- Can't re-use memory without "uncomputing" its value first

To perform classical functions reversibly, embed in a larger space

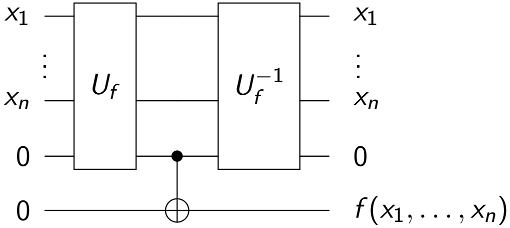
- $\mathit{Toffoli}(x, y, z) = (x, y, z \oplus (x \wedge y))$
- $\mathit{Toffoli}(x, y, 0) = (x, y, x \wedge y)$

Reclaiming space

Naïve "reversibilification": replace every AND gate with a Toffoli

- Temporary bits are called ancillas
- Uses space linear(!) in the number of AND gates

Bennett's trick: copy out result of a computation & uncompute

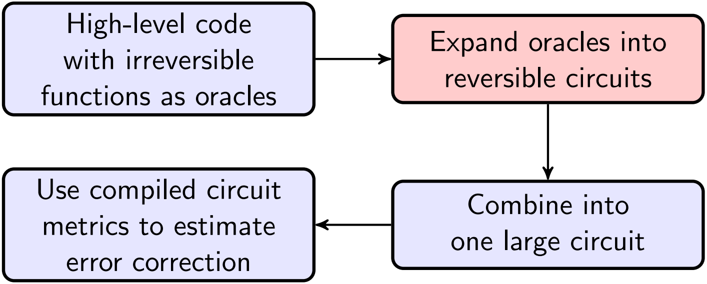


Resource estimation

Quantum compilers $\equiv$ *resource estimators*

- Estimate how much overhead a real implementation incurs

Typical design flow (e.g. Quipper, QCL):



High-level code with irreversible functions as oracles

Expand oracles into reversible circuits

Combine into one large circuit

Use compiled circuit metrics to estimate error correction

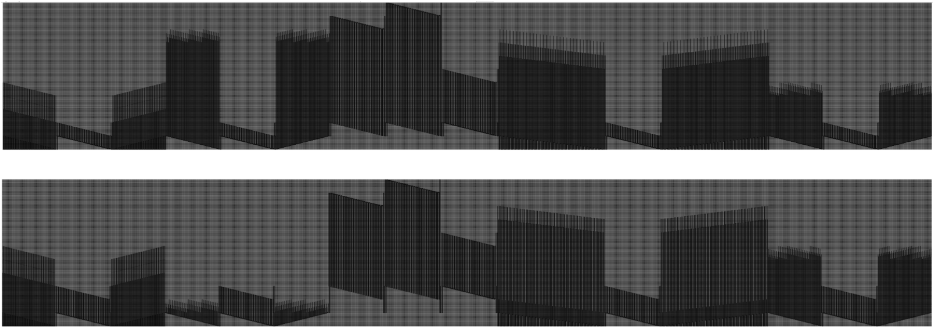
Ex. The QLS algorithm has an estimated logical space blowup of $\times 10^6$!

Why verify?

Resource estimates vary wildly between compilers

Typical hardware verification doesn't scale, since reversible circuits are monolithic & generally not reusable

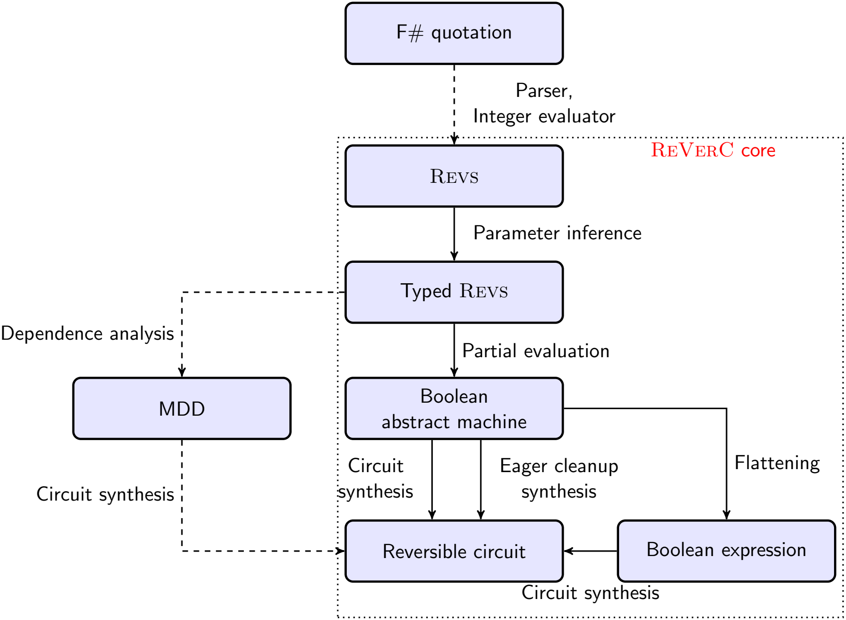
- Think assembly without labels or jumps



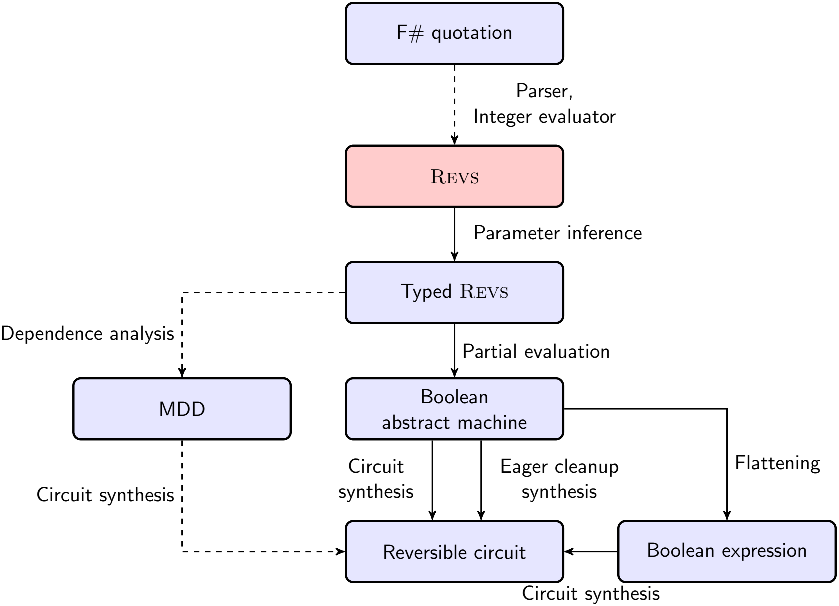
ReVerC

- Compiler for the F# embedded DSL Revs
- Performs optimizations for space-efficiency
- Formally verified in $F^\star$
- Includes a BDD-based assertion-checker for program verification

Compiler architecture



Revs



F# quotation

Parser,
Integer evaluator

Revs

Parameter inference

Typed Revs

Dependence analysis

Partial evaluation

MDD

Boolean
abstract machine

Circuit
synthesis

Eager cleanup
synthesis

Flattening

Circuit synthesis

Reversible circuit

Boolean expression

Circuit synthesis

REVS by example

n-bit adder

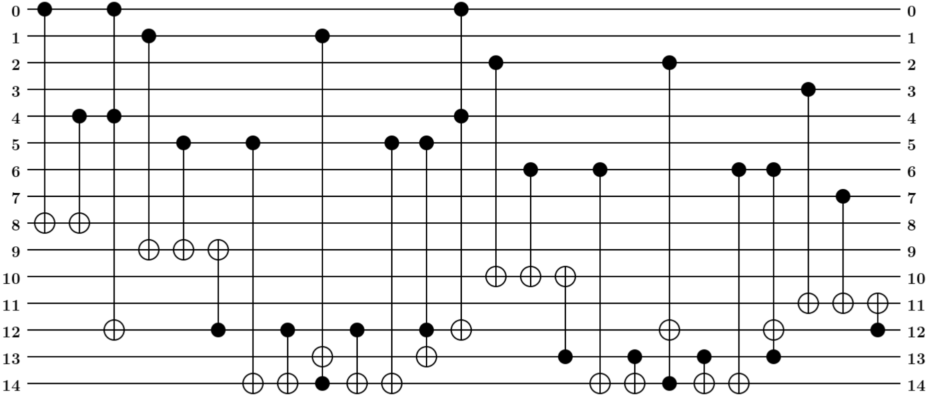
```
let adder n = <@
  fun a b ->
    let maj a b c = (a ∧ (b ⊕ c)) ⊕ (b ∧ c)
    let result = Array.zeroCreate(n)
    let mutable carry = false

    result.[0] ← a.[0] ⊕ b.[0]
    for i in 1 .. n-1 do
      carry ← maj a.[i-1] b.[i-1] carry
      result.[i] ← a.[i] ⊕ b.[i] ⊕ carry
      assert result.[i] = (a.[i] ⊕ b.[i] ⊕ carry)
    result
@>
```

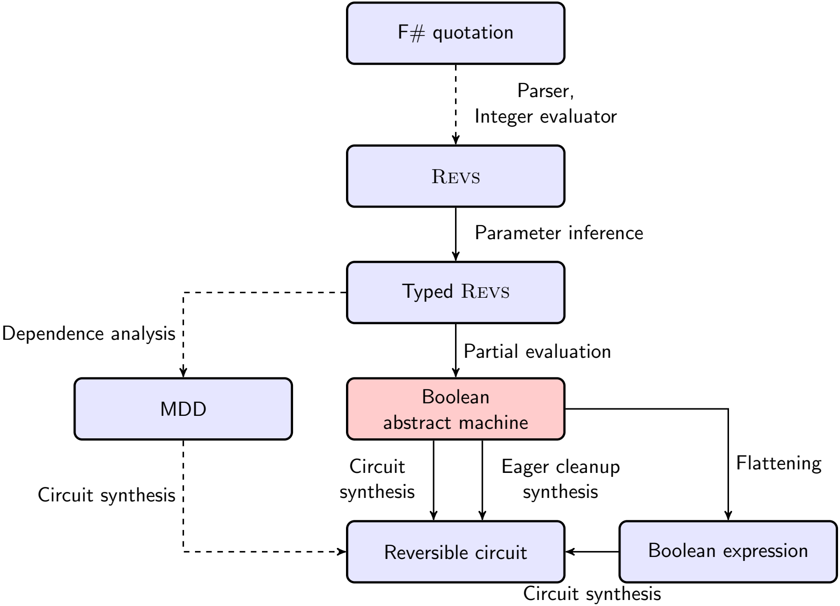
**Note: all control is compile-time static

REVS by example

n-bit adder



Boolean abstract machine



Boolean abstract machine

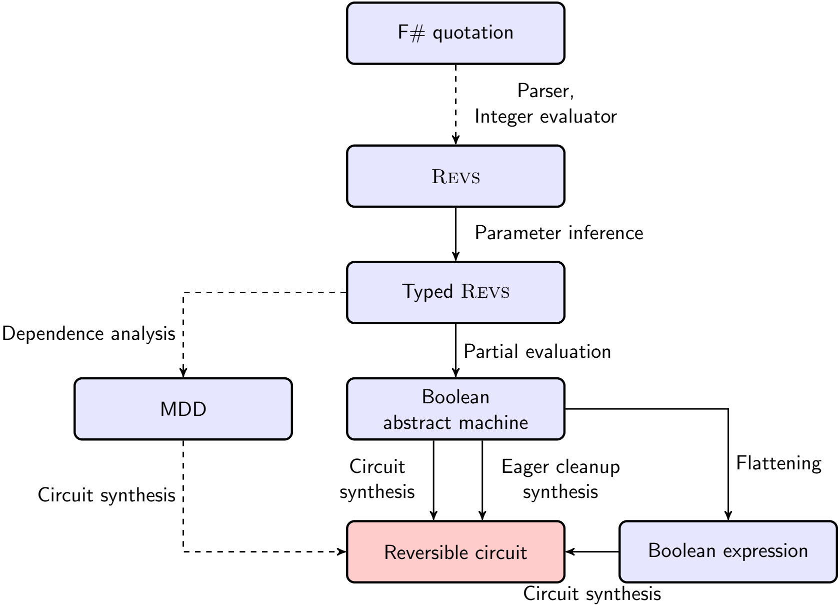
We use partial evaluation to reduce Revs to a sequence of assignments

- Lvalue most be a new, 0-valued store location
- RHS is a Boolean expression
- Semantics & transformation coincide → easier verification!

After expanding & assigning unique locations to a 4-bit adder:

```
(* result = alloc(4), carry_0 = alloc(1) *)
result.[0] ← a.[0] ⊕ b.[0]
carry_1    ← (a.[0] ∧ (b.[0] ⊕ carry_0)) ⊕ (b.[0] ∧ carry_0)
result.[1] ← a.[1] ⊕ b.[1] ⊕ carry_1
carry_2    ← (a.[1] ∧ (b.[1] ⊕ carry_1)) ⊕ (b.[1] ∧ carry_1)
result.[2] ← a.[2] ⊕ b.[2] ⊕ carry_2
carry_3    ← (a.[2] ∧ (b.[2] ⊕ carry_2)) ⊕ (b.[2] ∧ carry_2)
result.[3] ← a.[3] ⊕ b.[3] ⊕ carry_3
```

Circuit compilation



F# quotation

Parser,
Integer evaluator

Revs

Parameter inference

Typed Revs

Dependence analysis

Partial evaluation

MDD

Boolean
abstract machine

Circuit synthesis

Circuit
synthesis

Eager cleanup
synthesis

Flattening

Reversible circuit

Boolean expression

Circuit synthesis

Eager Cleanup

A.K.A. garbage collection

```
(* result = alloc(4), carry0 = alloc(1) *)
1 result.[0] ← a.[0] ⊕ b.[0]
2 carry1     ← (a.[0] ∧ (b.[0] ⊕ carry0)) ⊕ (b.[0] ∧ carry0)
3 result.[1] ← a.[1] ⊕ b.[1] ⊕ carry1
4 carry2     ← (a.[1] ∧ (b.[1] ⊕ carry1)) ⊕ (b.[1] ∧ carry1)
5 result.[2] ← a.[2] ⊕ b.[2] ⊕ carry2
6 carry3     ← (a.[2] ∧ (b.[2] ⊕ carry2)) ⊕ (b.[2] ∧ carry2)
7 result.[3] ← a.[3] ⊕ b.[3] ⊕ carry3
```

After line 4, we can garbage-collect $\texttt{carry}_1$ and reuse its space for $\texttt{carry}_3$

Problem: we can't overwrite $\texttt{carry}_1$ with the 0 state
Solution: each location i is associated with an expression $\kappa(i)$ s.t.

$$i \oplus \kappa(i) = 0$$

Eager Cleanup

```
1  c1 ← a.[0] ∧ b.[0]
2  c2 ← (a.[1] ∧ (b.[1] ⊕ c1)) ⊕ (b.[1] ∧ c1)
3  clean c1 (* c1 ← c1 ⊕ κ(c1) *)
4  c3 ← (a.[2] ∧ (b.[2] ⊕ c2)) ⊕ (b.[2] ∧ c2)
5  clean c2 (* c2 ← c2 ⊕ κ(c2) *)
6
```

l	$\kappa(c_1)$	$\kappa(c_2)$	$\kappa(c_1)$
1	0	0	0
2			
3			
4			
5			
6			

Eager Cleanup

```
1 c1 ← a.[0] ∧ b.[0]
2 c2 ← (a.[1] ∧ (b.[1] ⊕ c1)) ⊕ (b.[1] ∧ c1)
3 clean c1 (* c1 ← c1 ⊕ κ(c1) *)
4 c3 ← (a.[2] ∧ (b.[2] ⊕ c2)) ⊕ (b.[2] ∧ c2)
5 clean c2 (* c2 ← c2 ⊕ κ(c2) *)
6
```

l	$\kappa(c_1)$	$\kappa(c_2)$	$\kappa(c_1)$
1	0	0	0
2	$a_0 \wedge b_0$	0	0
3			
4			
5			
6			

Eager Cleanup

```
1 c₁ ← a.[0] ∧ b.[0]
2 c₂ ← (a.[1] ∧ (b.[1] ⊕ c₁)) ⊕ (b.[1] ∧ c₁)
3 clean c₁ (* c₁ ← c₁ ⊕ κ(c₁) *)
4 c₃ ← (a.[2] ∧ (b.[2] ⊕ c₂)) ⊕ (b.[2] ∧ c₂)
5 clean c₂ (* c₂ ← c₂ ⊕ κ(c₂) *)
6
```

l	$\kappa(c_1)$	$\kappa(c_2)$	$\kappa(c_1)$
1	0	0	0
2	$a_0 \wedge b_0$	0	0
3	$a_0 \wedge b_0$	$(a_1 \wedge (b_1 \oplus c_1)) \oplus (b_1 \wedge c_1)$	0
4			
5			
6			

Eager Cleanup

```
1  c₁ ← a.[0] ∧ b.[0]
2  c₂ ← (a.[1] ∧ (b.[1] ⊕ c₁)) ⊕ (b.[1] ∧ c₁)
3  clean c₁ (* c₁ ← c₁ ⊕ κ(c₁) *)
4  c₃ ← (a.[2] ∧ (b.[2] ⊕ c₂)) ⊕ (b.[2] ∧ c₂)
5  clean c₂ (* c₂ ← c₂ ⊕ κ(c₂) *)
6
```

l	$\kappa(c_1)$	$\kappa(c_2)$	$\kappa(c_1)$
1	0	0	0
2	$a_0 \wedge b_0$	0	0
3	$a_0 \wedge b_0$	$(a_1 \wedge (b_1 \oplus c_1)) \oplus (b_1 \wedge c_1)$	0
4	0	$(a_1 \wedge (b_1 \oplus (a_0 \wedge b_0))) \oplus (b_1 \wedge (a_0 \wedge b_0))$	0
5			
6			

Eager Cleanup

```
1  c₁ ← a.[0] ∧ b.[0]
2  c₂ ← (a.[1] ∧ (b.[1] ⊕ c₁)) ⊕ (b.[1] ∧ c₁)
3  clean c₁ (* c₁ ← c₁ ⊕ κ(c₁) *)
4  c₃ ← (a.[2] ∧ (b.[2] ⊕ c₂)) ⊕ (b.[2] ∧ c₂)
5  clean c₂ (* c₂ ← c₂ ⊕ κ(c₂) *)
6
```

l	$\kappa(c_1)$	$\kappa(c_2)$	$\kappa(c_1)$
1	0	0	0
2	$a_0 \wedge b_0$	0	0
3	$a_0 \wedge b_0$	$(a_1 \wedge (b_1 \oplus c_1)) \oplus (b_1 \wedge c_1)$	0
4	0	$(a_1 \wedge (b_1 \oplus (a_0 \wedge b_0))) \oplus (b_1 \wedge (a_0 \wedge b_0))$	0
5	0	$(a_1 \wedge (b_1 \oplus (a_0 \wedge b_0))) \oplus (b_1 \wedge (a_0 \wedge b_0))$	$(a_2 \wedge (b_2 \oplus c_2)) \oplus (b_2 \wedge c_2)$
6			

Eager Cleanup

```
1 c₁ ← a.[0] ∧ b.[0]
2 c₂ ← (a.[1] ∧ (b.[1] ⊕ c₁)) ⊕ (b.[1] ∧ c₁)
3 clean c₁ (* c₁ ← c₁ ⊕ κ(c₁) *)
4 c₃ ← (a.[2] ∧ (b.[2] ⊕ c₂)) ⊕ (b.[2] ∧ c₂)
5 clean c₂ (* c₂ ← c₂ ⊕ κ(c₂) *)
6
```

l	$\kappa(c_1)$	$\kappa(c_2)$	$\kappa(c_1)$
1	0	0	0
2	$a_0 \wedge b_0$	0	0
3	$a_0 \wedge b_0$	$(a_1 \wedge (b_1 \oplus c_1)) \oplus (b_1 \wedge c_1)$	0
4	0	$(a_1 \wedge (b_1 \oplus (a_0 \wedge b_0))) \oplus (b_1 \wedge (a_0 \wedge b_0))$	0
5	0	$(a_1 \wedge (b_1 \oplus (a_0 \wedge b_0))) \oplus (b_1 \wedge (a_0 \wedge b_0))$	$(a_2 \wedge (b_2 \oplus c_2)) \oplus (b_2 \wedge c_2)$
6	0	0	???

Verification

Formal verification of ReVerC[1] carried out in F*

$\sim$ 2000 lines of code

$\sim$ 2200 lines of proof code, written in 1 "person month"

Main theorems:

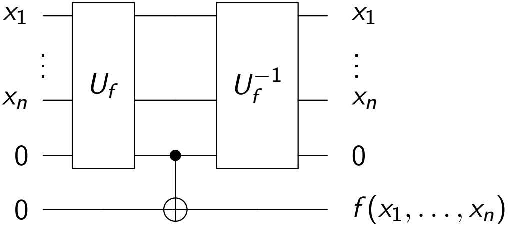
- Circuit synthesis produces correct output
- Circuit synthesis cleans all intermediate ancillas
- Each abstract machine compiler preserves the semantics
- All optimizations correct, etc.

---

[1] https://github.com/msr-quarc/ReVerC

Verifying Bennett

The Bennett trick:

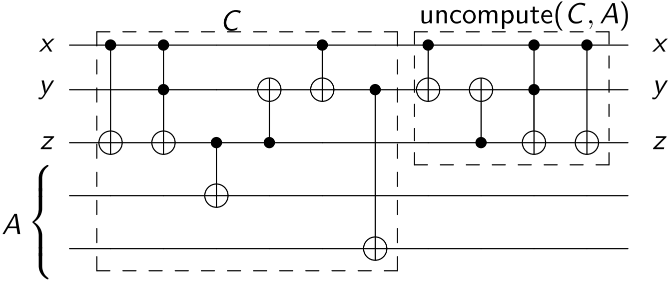


Works because the middle gate does not affect bits used in U_f

Verifying Bennett

A generalized Bennett method

Given a circuit C and set of bits A, we can uncompute C on $\overline{A}$ if no bits of A are used as controls in C



Verifying Bennett

```
val bennett : C:circuit -> copy:circuit -> st:state ->
  Lemma (requires (wfCirc C /\ disjoint (uses C) (mods copy)))
        (ensures  (agree_on st
                     (evalCirc (C@copy@(rev C)) st)
                     (uses C)))
let bennett C copy st =
  let st', st'' = evalCirc C st, evalCirc (C@copy) st in
    eval_mod st' copy;
    ctrls_sub_uses (rev C);
    evalCirc_state_swap (rev C) st' st'' (uses C);
    rev_inverse C st

val uncompute_mixed_inverse : C:circuit -> A:set int -> st:state ->
  Lemma (requires (wfCirc C /\ disjoint A (ctrls C)))
        (ensures  (agree_on st
                     (evalCirc (rev (uncompute C A)) (evalCirc C st))
                     (complement A)))
let uncompute_mixed_inverse C A st =
  uncompute_agree C A st;
  uncompute_ctrls_subset C A;
  evalCirc_state_swap (rev (uncompute C A))
                      (evalCirc C st)
                      (evalCirc (uncompute C A) st)
                      (complement A);
  rev_inverse (uncompute C A) st
```

Experiments

Bit counts with eager cleanup $\sim$ to state-of-the-art compiler

Benchmark	Revs (eager)			ReVerC (eager)		
	bits	gates	Toffolis	bits	gates	Toffolis
carryRippleAdd 32	129	467	124	**113**	361	90
carryRippleAdd 64	257	947	252	**225**	745	186
mult 32	128	6016	4032	128	6016	4032
mult 64	256	24320	16256	256	24320	16256
carryLookahead 32	**109**	1036	344	146	576	146
carryLookahead 64	**271**	3274	1130	376	1649	428
modAdd 32	65	188	62	65	188	62
modAdd 64	129	380	126	129	380	126
cucarroAdder 32	65	98	32	65	98	32
cucarroAdder 64	129	194	64	129	194	64
ma4	17	24	8	17	24	8
SHA-2 round	**353**	2276	754	449	1796	**594**
MD5	7905	82624	27968	**4769**	70912	**27520**

Conclusion

- Formalized an irreversible language Revs
- Designed a new eager cleaning method based on cleanup expressions
- Implemented & formally verified a compiler (ReVerC) in F*

Take aways

- Proving theorems about real code is not unreasonably difficult
- Design code in such a way to minimize the scope of difficult logic

Thank you!

Questions?