Verification in Quantum Computing

Matthew Amy

University of Waterloo & Institute for Quantum Computing

Design Automation for Quantum Computing
November 16th, 2017
Quantum computing

Theory:

The Clock Is Ticking for Encryption

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

Your Encryption Will Be Useless Against Hackers with Quantum Computers

Online security braces for quantum revolution

Quantum Computers And The End Of Security

NSA SWITCHES TO QUANTUM-RESISTANT CRYPTOGRAPHY

Matthew Amy (IQC)
Quantum computing

Reality:

*Quantum computing is weakened by a high degree of overhead*

Sources of overhead:

- Intrinsic overhead of an algorithm
  
  *e.g. overhead of Grover’s search*

- Overhead incurred at the logical layer due to reversibility
  
  *e.g. $g : |x\rangle|y\rangle \rightarrow |x\rangle|y \oplus f(x)\rangle$*

- Additional overhead at the physical layer due to error correction
Algorithmic overhead: Additional query of $f$, $4n - 8$ Toffolis
Logical overhead: $1600$ qubits, $> 2 \times$ the number of gates
Physical overhead: $2^{38}$ times as many “executions of SHA-256”
Resource estimation

Estimate how much resources (time & space) a realistic implementation of an algorithm uses

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

1. High-level code with irreversible functions as oracles
2. Compile oracles into reversible circuits
3. Optimize circuit(s)
4. Use compiled circuit metrics to estimate error correction

Errors can (and do) occur at any stage!
Example
Eager cleanup bug

Without optimization:

\[
\begin{array}{c}
a_0 \\
a_1 \\
0 \\
0 \\
0 \\
0 \\
0 \\
\end{array}
\]

With optimization:

\[
\begin{array}{c}
a_0 \\
a_1 \\
0 \\
0 \\
0 \\
0 \\
a_0 \oplus a_1 \\
\end{array}
\]
Why verify?

1.) *Quantum resource estimates are being used to guide real security policies*
   - Open Quantum Safe ([https://openquantumsafe.org/](https://openquantumsafe.org/))

2.) *Resource estimates vary wildly between compilers*
   e.g. for binary welded tree \((n = 100, s = 100)\)
   - ScaffCC gives 571805 qubits, 33966707 gates
   - Quipper gives 314/1932 qubits, 30424410/36257210 gates
Why verify formally?

3.) Testing capability is limited
   - Quantum simulation doesn’t scale
   - Circuits are special-purpose and monolithic
Verifying a resource analysis design flow

1. High-level code with irreversible functions as oracles
2. Compile oracles into reversible circuits
3. Optimize circuit(s)
4. Use compiled circuit metrics to estimate error correction

Program verification

- Prove properties of expected behaviour for specific programs
- Properties may not be true of all programs, e.g. integer overflow
- Techniques include abstract interpretation (Entanglement analysis), model checking (Quantum model-checker), type systems (Quipper), formal proof (Quantum Hoare Logic)

Quantum-specific challenges:

- What are the program properties of interest?
Verifying a resource analysis design flow

1. High-level code with irreversible functions as oracles
2. Compile oracles into reversible circuits
3. Optimize circuit(s)
4. Use compiled circuit metrics to estimate error correction

Compiler verification
- Compiled program executes as expected
- Techniques include translation validation (per program), formal proof (all programs)

- CompCert, , ReVerC

Quantum-specific challenges:
- Explicit clean-up and reuse of memory
- Probabilistic semantics
Formal proof in compiler verification
ML-like language with dependent types developed at MSR

What are Dependent types?
- **Types** may depend on terms – i.e. Array n
- Corresponds to predicate logic (*Curry-Howard isomorphism*)

What are they useful for? **writing logical specifications/theorems**

```plaintext
val head : l:List{not (is_Empty l)} -> Tot int
val insert_is_heap : h:Heap -> i:int ->
  Lemma (is_heap h ⇒ is_heap (insert h i))
val compile_correct :
  Lemma (∀ P:program, i:inputs.
    eval_program P i = eval_assembly (compile P) i)
```

How do we verify specifications/theorems are correct?
- F* compiler checks specifications with SMT solver
  **caveat:** typically have to write lemmas & induction structure.
ReVerC (arXiv:1603.01635)
https://github.com/msr-quarc/ReVerC

Reversible circuit compiler for the F# embedded DSL Revs

- Compiles irreversible code into reversible circuits
- Performs optimizations for space-efficiency
- Formally verified in F*
- Includes a BDD-based assertion-checker for program verification & additional translation validation
Compiler architecture

F# quotation

Parser, Integer evaluator

Typed ReVS

Parameter inference

Typed ReVS

Partial evaluation

Boolean abstract machine

Circuit synthesis

Eager cleanup synthesis

Reversible circuit

Boolean expression

Circuit synthesis

Dependence analysis

MDD

Circuit synthesis

ReVERC core
Revs

- F# quotation
- Parser, Integer evaluator
- Revs
- Parameter inference
- Typed Revs
- Partial evaluation
- Typed Revs
- Boolean abstract machine
- Circuit synthesis
- Eager cleanup synthesis
- Reversible circuit
- Boolean expression
- Flattening

- Dependence analysis
- MDD
- Circuit synthesis
\textbf{Revs}

\textbf{Var} \ x, \ \textbf{Bool} \ b \in \{0, 1\} = \mathbb{B}, \ \textbf{Nat} \ i, j \in \mathbb{N}, \ \textbf{Loc} \ l \in \mathbb{N}

\textbf{Val} \ \nu ::= \text{unit} \mid l \mid \text{reg} \ l_1 \ldots l_n \mid \lambda x. t

\textbf{Term} \ t ::= \text{let} \ x = t_1 \ \text{in} \ t_2 \mid \lambda x. t

\quad \mid (t_1 \ t_2)

\quad \mid t_1; t_2

\quad \mid x

\quad \mid t_1 \leftarrow t_2

\quad \mid b \mid t_1 \oplus t_2 \mid t_1 \land t_2

\quad \mid \text{reg} \ t_1 \ldots t_n \mid t.[i] \mid t.[i..j] \mid \text{append} \ t_1 \ t_2 \mid \text{rotate} \ i \ t

\quad \mid \text{clean} \ t \mid \text{assert} \ t
Revs by example

*n*-bit adder

```ocaml
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
let s0 a =  
let a2 = rot 2 a  
let a13 = rot 13 a  
let a22 = rot 22 a  
let t = Array.zeroCreate 32  
for i in 0 .. 31 do  
    t.[i] ← a2.[i] ⊕ a13.[i] ⊕ a22.[i]  

let s1 a =  
let a6 = rot 6 a  
let a11 = rot 11 a  
let a25 = rot 25 a  
let t = Array.zeroCreate 32  
for i in 0 .. 31 do  
    t.[i] ← a6.[i] ⊕ a11.[i] ⊕ a25.[i]

let ch e f g =  
let t = Array.zeroCreate 32  
for i in 0 .. 31 do  
    t.[i] ← e.[i] ∧ f.[i] ∧ g.[i]  
t

fun k w x →  
let hash x =  
let a = x.[0..31],  
b = x.[32..63],  
c = x.[64..95],  
d = x.[96..127],  
e = x.[128..159],  
f = x.[160..191],  
g = x.[192..223],  
h = x.[224..255]
(%modAdd 32) (ch e f g) h  
(%modAdd 32) (s0 a) h  
(%modAdd 32) w h  
(%modAdd 32) k h  
(%modAdd 32) d h  
(%modAdd 32) (ma a b c) h  
(%modAdd 32) (s1 e) h  
for i in 0 .. n - 1 do  
    hash (rot 32*i x)
x
Typed Revs

- F# quotation
- Parser, Integer evaluator
- Revs
- Parameter inference
- Typed Revs
- Partial evaluation
- Dependence analysis
- MDD
  - Circuit synthesis
- Typed Revs
  - Boolean abstract machine
    - Circuit synthesis
    - Eager cleanup synthesis
    - Flattening
- Reversible circuit
  - Boolean expression
  - Circuit synthesis
Typed Revs

**Type** \[ T ::= X \mid \text{Unit} \mid \text{Bool} \mid \text{Reg } n \mid T_1 \rightarrow T_2 \]

Inferred type system with statically typed registers sizes

- Main purpose is to simplify the job of the compiler
  - Simpler compiler ⇒ easier verification!

- Verification-light
  - Prevents out-of-bounds register accesses
  - Sanity check for register sizes

```ocaml
let f = fun a : Reg 8 -> ... in
let a = Array.zeroCreate 8 in
let b = Array.zeroCreate 16 in
f a
f b
```
Type/parameter inference

Basic idea: solve a system of integer linear arithmetic constraints

- e.g. \((x = \text{Reg } y) \land (y \geq z - 3) \land (y \geq 8)\)
- let \(c = \text{append a b} \rightarrow (c : \text{Reg } x) \land (a : \text{Reg } y) \land (b : \text{Reg } z) \land (x \geq y + z)\)

Solver overview:

- Solve equalities by unification
- Merge arithmetic constraints & reduce to normal form
- For constraints \(x \geq n\), set \(x = n\)
- Check remaining arithmetic constraints are satisfied

Caveat: doesn’t always find a solution
Boolean abstract machine

- F# quotation
  - Parser, Integer evaluator
  - Parameter inference
- Typed REVS
  - Partial evaluation
- Typed REVS
  - Dependence analysis
  - MDD
- Boolean abstract machine
  - Circuit synthesis
  - Eager cleanup synthesis
- Boolean expression
  - Flattening
- Reversible circuit
  - Circuit synthesis
Boolean abstract machine

Only one operation:

assign a store location to the result of a Boolean expression

Partial evaluation used to transform \texttt{REVS} code into the abstract machine

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

**Strictly more general than reversible circuits**
Example
Adder circuit

```ocaml
fun a b ->
  let carry_ex a b c = (a ∧ (b ⊕ c)) ⊕ (b ∧ c)
  let result = Array.zeroCreate(4)
  let mutable carry = false

  result.[0] ← a.[0] ⊕ b.[0]
  for i in 1 .. 4-1 do
    carry ← carry_ex a.[i-1] b.[i-1] carry
    result.[i] ← a.[i] ⊕ b.[i] ⊕ carry
    assert (result.[i] = (a.[i] ⊕ b.[i] ⊕ carry))
  result

↓ partial evaluation

(* 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
```
Recall
Reversible computing

Every operation must be invertible
- $x \land 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
- $\text{Toffoli}(x, y, z) = (x, y, z \oplus (x \land y))$
- $\text{Toffoli}(x, y, 0) = (x, y, x \land y)$
Recall

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

\[
\begin{align*}
    U_f \\
    \vdots \\
    x_n \\
    0 \\
    0
\end{align*}
\]

\[
\begin{align*}
    U_f^{-1} \\
    \vdots \\
    x_n \\
    0 \\
    f(x_1, \ldots, x_n)
\end{align*}
\]
Circuit compilation

F# quotation

Parser, Integer evaluator

REVS

Parameter inference

Typed REVS

Partial evaluation

Boolean abstract machine

Circuit synthesis

Eager cleanup synthesis

Reversible circuit

Circuit synthesis

Boolean expression

MDD

Dependence analysis

Circuit synthesis

Flattening
Eager Cleanup
A.K.A. garbage collection

(* result = alloc(4), carry\_0 = alloc(1) *)

1. \texttt{result}[0] \leftarrow \texttt{a}[0] \oplus \texttt{b}[0]
2. \texttt{carry\_1} \leftarrow (\texttt{a}[0] \land (\texttt{b}[0] \oplus \texttt{carry\_0})) \oplus (\texttt{b}[0] \land \texttt{carry\_0})
3. \texttt{result}[1] \leftarrow \texttt{a}[1] \oplus \texttt{b}[1] \oplus \texttt{carry\_1}
4. \texttt{carry\_2} \leftarrow (\texttt{a}[1] \land (\texttt{b}[1] \oplus \texttt{carry\_1})) \oplus (\texttt{b}[1] \land \texttt{carry\_1})
5. \texttt{result}[2] \leftarrow \texttt{a}[2] \oplus \texttt{b}[2] \oplus \texttt{carry\_2}
6. \texttt{carry\_3} \leftarrow (\texttt{a}[2] \land (\texttt{b}[2] \oplus \texttt{carry\_2})) \oplus (\texttt{b}[2] \land \texttt{carry\_2})
7. \texttt{result}[3] \leftarrow \texttt{a}[3] \oplus \texttt{b}[3] \oplus \texttt{carry\_3}

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 \texttt{i} is associated with an expression \(\kappa(i)\) s.t.

\[ i \oplus \kappa(i) = 0 \]
Interpretations

Compilation methods defined by providing interpretations $\mathcal{I}$ of the abstract machine

An interpretation consists of a domain $D$ and two operations

\[
\text{assign} : D \times \mathbb{N} \times \text{BExp} \rightarrow D
\]
\[
\text{eval} : D \times \mathbb{N} \times \text{State} \rightarrow \mathbb{B}.
\]

**Semantic function eval is provided to unify verification**
Circuit synthesis

\[ B \exp ::= 0 \mid 1 \mid i \mid \neg B \mid B_1 \oplus B_2 \mid B_1 \land B_2 \]

To be reversible compiled expression must have the form \( i \oplus B \)

\[ i \oplus (B_1 \oplus B_2) \rightarrow \]

\[ i \oplus (B_1 \land B_2) \rightarrow \]
Eager Cleanup
A.K.A. garbage collection

In the 4-bit adder example, after the assignment
\[ \text{carry}_2 \leftarrow (a_[1] \land (b_[1] \oplus \text{carry}_1)) \oplus (b_[1] \land \text{carry}_1) \]
the location of \text{carry}_1 is no longer in use, so we can reuse it for \text{carry}_3

Problems: we can’t overwrite \text{carry}_1 with the ”0” state

Solution: if \text{carry}_1 is in the state \( B \), \( \text{carry}_1 \oplus B = 0 \)
\[ \Rightarrow \text{location } i \text{ is associated with an expression } \kappa(i) \text{ such that } i \oplus \kappa(i) = 0 \]
Eager Cleanup

1. $c_1 \leftarrow a.[0] \land b.[0]$

2. $c_2 \leftarrow (a.[1] \land (b.[1] \oplus c_1)) \oplus (b.[1] \land c_1)$

3. \textbf{clean} $c_1$ (* $c_1 \leftarrow c_1 \oplus \kappa(c_1)$ *)

4. $c_3 \leftarrow (a.[2] \land (b.[2] \oplus c_2)) \oplus (b.[2] \land c_2)$

5. \textbf{clean} $c_2$ (* $c_2 \leftarrow c_2 \oplus \kappa(c_2)$ *)

<table>
<thead>
<tr>
<th>$l$</th>
<th>$\kappa(c_1)$</th>
<th>$\kappa(c_2)$</th>
<th>$\kappa(c_1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Eager Cleanup

1. \[c_1 \leftarrow a.[0] \land b.[0]\]
2. \[c_2 \leftarrow (a.[1] \land (b.[1] \oplus c_1)) \oplus (b.[1] \land c_1)\]
3. \textbf{clean } c_1 (* c_1 \leftarrow c_1 \oplus \kappa(c_1) *)
4. \[c_3 \leftarrow (a.[2] \land (b.[2] \oplus c_2)) \oplus (b.[2] \land c_2)\]
5. \textbf{clean } c_2 (* c_2 \leftarrow c_2 \oplus \kappa(c_2) *)

<table>
<thead>
<tr>
<th>(l)</th>
<th>(\kappa(c_1))</th>
<th>(\kappa(c_2))</th>
<th>(\kappa(c_1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>(a_0 \land b_0)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Eager Cleanup

1. \[ c_1 \leftarrow a[0] \land b[0] \]

2. \[ c_2 \leftarrow (a[1] \land (b[1] \oplus c_1)) \oplus (b[1] \land c_1) \]

3. \textbf{clean} \( c_1 \) (* \( c_1 \leftarrow c_1 \oplus \kappa(c_1) * \)*)

4. \[ c_3 \leftarrow (a[2] \land (b[2] \oplus c_2)) \oplus (b[2] \land c_2) \]

5. \textbf{clean} \( c_2 \) (* \( c_2 \leftarrow c_2 \oplus \kappa(c_2) * \)*)

<table>
<thead>
<tr>
<th>( l )</th>
<th>( \kappa(c_1) )</th>
<th>( \kappa(c_2) )</th>
<th>( \kappa(c_1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>( a_0 \land b_0 )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>( a_0 \land b_0 )</td>
<td>( (a_1 \land (b_1 \oplus c_1)) \oplus (b_1 \land c_1) )</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Eager Cleanup

1. \[ c_1 \leftarrow a\cdot[0] \land b\cdot[0] \]
2. \[ c_2 \leftarrow (a\cdot[1] \land (b\cdot[1] \oplus c_1)) \oplus (b\cdot[1] \land c_1) \]
3. \textbf{clean } c_1 (* \; c_1 \leftarrow c_1 \oplus \kappa(c_1) *)
4. \[ c_3 \leftarrow (a\cdot[2] \land (b\cdot[2] \oplus c_2)) \oplus (b\cdot[2] \land c_2) \]
5. \textbf{clean } c_2 (* \; c_2 \leftarrow c_2 \oplus \kappa(c_2) *)

<table>
<thead>
<tr>
<th>(l)</th>
<th>(\kappa(c_1))</th>
<th>(\kappa(c_2))</th>
<th>(\kappa(c_1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>(a_0 \land b_0)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>(a_0 \land b_0)</td>
<td>((a_1 \land (b_1 \oplus c_1)) \oplus (b_1 \land c_1))</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>((a_1 \land (b_1 \oplus (a_0 \land b_0))) \oplus (b_1 \land (a_0 \land b_0)))</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Eager Cleanup

1. \( c_1 \leftarrow a.[0] \land b.[0] \)
2. \( c_2 \leftarrow (a.[1] \land (b.[1] \oplus c_1)) \oplus (b.[1] \land c_1) \)
3. clean \( c_1 \) (* \( c_1 \leftarrow c_1 \oplus \kappa(c_1) *) \)
4. \( c_3 \leftarrow (a.[2] \land (b.[2] \oplus c_2)) \oplus (b.[2] \land c_2) \)
5. clean \( c_2 \) (* \( c_2 \leftarrow c_2 \oplus \kappa(c_2) *) \)

<table>
<thead>
<tr>
<th>( l )</th>
<th>( \kappa(c_1) )</th>
<th>( \kappa(c_2) )</th>
<th>( \kappa(c_1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>( a_0 \land b_0 )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>( a_0 \land b_0 )</td>
<td>( (a_1 \land (b_1 \oplus c_1)) \oplus (b_1 \land c_1) )</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>( (a_1 \land (b_1 \oplus (a_0 \land b_0))) \oplus (b_1 \land (a_0 \land b_0)) )</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>( (a_1 \land (b_1 \oplus (a_0 \land b_0))) \oplus (b_1 \land (a_0 \land b_0)) )</td>
<td>( (a_2 \land (b_2 \oplus c_2)) \oplus (b_2 \land c_2) )</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Eager Cleanup

1. \( c_1 \leftarrow a.[0] \land b.[0] \)
2. \( c_2 \leftarrow (a.[1] \land (b.[1] \oplus c_1)) \oplus (b.[1] \land c_1) \)
3. \textbf{clean } c_1 (* c_1 \leftarrow c_1 \oplus \kappa(c_1) *)
4. \( c_3 \leftarrow (a.[2] \land (b.[2] \oplus c_2)) \oplus (b.[2] \land c_2) \)
5. \textbf{clean } c_2 (* c_2 \leftarrow c_2 \oplus \kappa(c_2) *)

<table>
<thead>
<tr>
<th>( l )</th>
<th>( \kappa(c_1) )</th>
<th>( \kappa(c_2) )</th>
<th>( \kappa(c_1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>( a_0 \land b_0 )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>( a_0 \land b_0 )</td>
<td>( a_1 \land (b_1 \oplus c_1) ) ( \oplus (b_1 \land c_1) )</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>( a_1 \land (b_1 \oplus (a_0 \land b_0)) ) ( \oplus (b_1 \land (a_0 \land b_0)) )</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>( a_1 \land (b_1 \oplus (a_0 \land b_0)) ) ( \oplus (b_1 \land (a_0 \land b_0)) )</td>
<td>( a_2 \land (b_2 \oplus c_2) ) ( \oplus (b_2 \land c_2) )</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>???</td>
</tr>
</tbody>
</table>
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:

\[
x \rightarrow U_f \rightarrow U_f^{-1} \rightarrow f(x_1, \ldots, x_n)
\]

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

```ocaml
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;
  ctrl_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
```

Matthew Amy (IQC)
(\* Circuit synthesis correctness \*)

val compile_bexp_correct : ah:ancHeap -> targ:int ->
                         exp:boolExp -> st:state ->

  Lemma (requires (zeroHeap st ah /
                             disjoint (elts ah) (vars exp) /
                             not (Set.mem targ (elts ah)) /
                             not (Set.mem targ (vars exp))))

  (ensures (compileBexpEval ah targ exp st =
                   (lookup st targ) <> evalBexp exp st))
(* Circuit synthesis cleans ancillas *)

val compile_with_cleanup : ah:ancHeap -> targ:int ->
    exp:boolExp -> st:state ->

Lemma (requires (zeroHeap st ah /
    disjoint (elts ah) (vars exp) /
    not (Set.mem targ (elts ah)) /
    not (Set.mem targ (vars exp)))))

(ensures (zeroHeap (compileBexpCleanEvalSt ah targ exp st)
    (first (compileBexpClean ah targ exp))))
(* "Circuit" interpretation preserves semantics *)

```ocaml
type valid_circ_state (cs: circState) (init : state ) =
   (forall l l'. not (l = l') ==> not (lookup cs. subs l = lookup cs. subs l')) /
   disjoint (vals cs. subs) (elts cs.ah) /
   zeroHeap init cs.ah /
   zeroHeap (evalCirc cs. gates init) cs.ah /
   (forall bit. Set.mem bit (vals cs.subs) ==> (lookup cs. zero bit = true ==> lookup (evalCirc cs. gates init) bit = false))
```

```ocaml
type equiv_state (cs: circState) (bs: boolState) (init : state ) =
   cs. top = forall i. circEval cs init i = boolEval bs init i
```

```ocaml
val assign_pres_equiv : cs: circState -> bs: boolState -> l: int ->
   bexp: boolExp -> init: state ->
   Lemma (requires (valid_circ_state cs init /
   equiv_state cs bs init))
   (ensures (valid_circ_state (circAssign cs l bexp) init /
     equiv_state (circAssign cs l bexp) (boolAssign bs l bexp) init))
```
Verification

(* "Eager cleanup" interpretation preserves semantics *)

type valid_GC_state (cs: circGCState) (init: state) =
  (forall l l'. not (l = l') ==> 
    not (lookup cs.symtab l = lookup cs.symtab l')) /\ 
  (disjoint (vals cs.symtab) (elts cs.ah)) /\ 
  (zeroHeap init cs.ah) /\ 
  (zeroHeap (evalCirc cs.gates init) cs.ah) /\ 
  (forall bit. Set.mem bit (vals cs.symtab) ==> 
    (disjoint (vars (lookup cs.cvals bit)) (elts cs.ah))) /\ 
  (forall bit. Set.mem bit (vals cs.symtab) ==> 
    (b2t(lookup cs.isanc bit) ==> lookup init bit = false)) /\ 
  (forall bit. Set.mem bit (vals cs.symtab) ==> 
    (evalBexp (BXor (BVar bit, (lookup cs.cvals bit)))
    (evalCirc cs.gates init) = lookup init bit))

type equiv_state (cs: circGCState) (bs: boolState) (init: state) =
  cs.top = forall i. circGCEval cs init i = boolEval bs init i

val assign_pres_equiv : cs: circGCState -> bs: boolState -> l: int ->
  bexp: boolExp -> init: state ->
  Lemma (requires (valid_GC_state cs init /\ equiv_state cs bs init))
  (ensures (valid_GC_state (circGCAssign cs l bexp) init /\
      equiv_state (circGCAssign cs l bexp)
      (boolAssign bs l bexp) init))
## Experiments

Bit counts with eager cleanup ~ to state-of-the-art compiler

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

Given a circuit $C$, can we verify that $C$ implements a unitary matrix $U$? What about an optimized circuit $C'$?

The reversible case

- Classical CAD techniques such as miteres & BDDs or SAT solvers applicable here
- BDD-based verification in ReVerC starts thrashing at $\sim 75$ bits with $8$ Gb memory
- May be able to go further with functional coverage techniques

The quantum case

- Decision diagram-based techniques applied in the past (QuIDD)
- Limited by size of unitaries
Sum-over-paths

A space-efficient, natural mathematical description of unitaries

\[
R_z(\theta) : |x\rangle \mapsto e^{2\pi i \theta x} |x\rangle
\]

\[
H : |x\rangle \mapsto \frac{1}{\sqrt{2}} \sum_{y \in \{0,1\}} \omega^{4xy} |y\rangle
\]

\[\text{Toffoli}_n : |x_1 x_2 \cdots x_n\rangle \mapsto |x_1 x_2 \cdots (x_1 \land x_2 \land \cdots \land x_n)\rangle\]

\[\text{Adder}_n : |x\rangle |y\rangle |0\rangle \mapsto |x\rangle |y\rangle |x + y\rangle\]

\[\text{QFT}_n : |x\rangle \mapsto \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} e^{2\pi i xy/2^n} |y\rangle\]

In general:

\[
U : |x\rangle \mapsto \frac{1}{\sqrt{2^k}} \sum_{y \in \{0,1\}^k} e^{2\pi ip(x,y)} |f(x, y)\rangle
\]

** Efficiently composable & computable from a circuit representation!**
An equivalence checking methodology

Basic fact:

\[ U = I \iff H^\otimes n U H^\otimes n |0\rangle = |0\rangle \]

To check equivalence of a circuit \( C \) w.r.t. a circuit or specification \( C' \),

1. Compute sum-over-paths representations \( U_C \) and \( U_{C'} \)
2. Construct quantum miter 
   \[ U = H^\otimes n U_C \circ U_{C'}^\dagger H^\otimes n \]
3. If 
   \[ U : |x\rangle \mapsto \frac{1}{\sqrt{2}^k} \sum_{y \in \{0,1\}^k} e^{2\pi ip(x,y)} |f(x, y)\rangle, \]
   verify
   \[ \frac{1}{\sqrt{2}^k} \sum_{y \in \{0,1\}^k, f(0,y)=0} e^{2\pi ip(0,y)} = 1 \]

If \( p \in \mathbb{Z}_2[x, y] \), then step 3 reduces to \( \#SAT \). Moreover, if \( \deg(p) \leq 2 \), step 3 is efficiently computable (Montanaro, arXiv:1607.08473)
Symbolic reductions

*Can we do better for other polynomials?*

Recall: for Clifford $+ T$, $p \in \mathbb{Z}_8[x, y]$

\[
\frac{1}{\sqrt{2^{k+1}}} \sum_{y \in \{0,1\}^k} \omega^{4y'q(x,y)+r(x,y)} |f(x, y)\rangle = \frac{1}{\sqrt{2^{k-1}}} \sum_{y \in \{0,1\}^k} \omega^{r(x,y)} |f(x, y)\rangle \quad (1)
\]

\[
\frac{1}{\sqrt{2^{k+1}}} \sum_{y \in \{0,1\}^k} \omega^{2y'+4y'q(x,y)+r(x,y)} |f(x, y)\rangle = \frac{1}{\sqrt{2^k}} \sum_{y \in \{0,1\}^k} \omega^{1+6q(x,y)+r(x,y)} |f(x, y)\rangle \quad (2)
\]

Using just relation (1), possible to verify a number of optimized arithmetic operators on 32-bit registers against specifications in seconds
Conclusion

- Formalized an irreversible language \texttt{REVS}
- Designed a new eager cleaning method based on cleanup expressions
- Implemented & formally verified a compiler (\texttt{REVERC}) in F* 

Take aways

- Proving theorems about real code is \textbf{not} unreasonably difficult
- Design code in such a way to minimize the scope of difficult logic
Going forward

Formally verify *quantum* circuit compilers
- Verifying library function implementations
- Verifying optimization

Develop methods for
- Functional coverage?
Thank you!

Questions?