Correct-by-construction Cryptographic Hardware via Explicit Staging Transformations*

Yakir Forman Bill Harrison

High Assurance Solutions
Two Six Technologies, Inc.
Arlington Virginia

* This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) program *Data Protection in Virtual Environments* (DPRIVE). The views, opinions and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Background: Staging Transformations

“Staging Transformations” have been around a while

- *Pass Separation* transformation (Jørring&Scherlis86)
  - Program transformation/annotation partitioning into compile-time and run-time parts
- Code constructor in MetaML (Taha&Sheard00)
  - “`1 + 2`” is an expression of type `int`
  - “`< 1 + 2 >`” is an expression of type `code(int)` that, if you run it, will produce `3`

Background: Staging Transformations

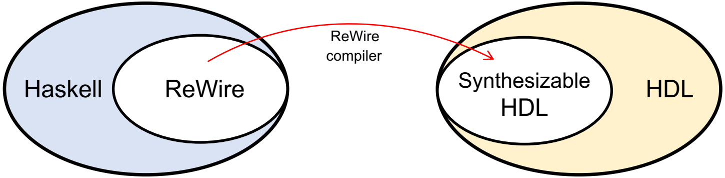
“Staging Transformations” have been around a while

- *Pass Separation* transformation (Jørring&Scherlis86)
  - Program transformation/annotation partitioning into compile-time and run-time parts
- Code constructor in MetaML (Taha&Sheard00)
  - “`1 + 2`” is an expression of type `int`
  - “`< 1 + 2 >`” is an expression of type `code(int)` that, if you run it, will produce `3`

Today: Haskell/ReWire `stage` functions

- Staging transformation: just applying `stage` to part of algorithm
- `stage x` turns computation `x` into single cycle of hardware device
- `stage` functions are *akin* to lift functions of monad transformers

Background: ReWire Language & Toolchain



- Inherits Haskell's good qualities
  - Pure functions, strong types, monads, equational reasoning, etc.
- ReWire compiler produces Verilog, VHDL, or FIRRTL
- Freely Available: `https://github.com/twosixlabs/rewire`
- ReWire Formalization in ITP Systems (Isabelle, Coq, Agda)

Carry-Save Adders in ReWire

Carry-Save Addition (CSA) as Pure Function

```
f :: W8 → W8 → W8 → (W8, W8)
f a b c    = ( ((a & b) || (a & c) || (b & c) ) << '0' , a ⊕ b ⊕ c )
```

Running in GHCi

```
ghci> f 40 25 20
      (48,37)
ghci> f 41 25 20
      (50,36)
```

twosix TECHNOLOGIES

Carry-Save Adders in ReWire

Carry-Save Addition (CSA) as Pure Function

```
f :: W8 → W8 → W8 → (W8, W8)
f a b c    = ( ((a & b) || (a & c) || (b & c) ) << '0' , a ⊕ b ⊕ c )
```

CSA Device in ReWire

```
csa :: (W8, W8, W8) → Re (W8, W8, W8) () (W8, W8) ()
csa (a, b, c) = do
                  i ← signal (f a b c)
                  csa i     -- N.b., tail-recursive
```

Carry-Save Adders in ReWire

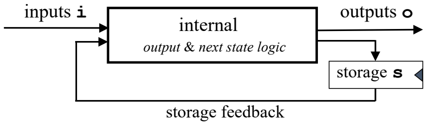
Carry-Save Addition (CSA) as Pure Function

```
f :: W8 → W8 → W8 → (W8, W8)
f a b c     = ( ((a & b) || (a & c) || (b & c) ) << '0' , a ⊕ b ⊕ c )
```

CSA Device in ReWire

```
csa :: (W8, W8, W8) → Re (W8, W8, W8) () (W8, W8) ()
csa (a, b, c) = do
              i ← signal (f a b c)
              csa i      -- N.b., tail-recursive
```

Mealy Machine



Corresponding ReWire monad

```
type M s       = StateT s Identity
 -- ReWire monad
type Re i s o = ReacT i o (M s)
 -- consume/produce inputs & outputs synchronously
signal :: o → Re i s o i
```

Carry-Save Adders in ReWire

Carry-Save Addition (CSA) as Pure Function

```
f :: W8 → W8 → W8 → (W8, W8)
f a b c    = ( ((a & b) || (a & c) || (b & c) ) << '0' , a ⊕ b ⊕ c )
```

CSA Device in ReWire

```
csa :: (W8, W8, W8) → Re (W8, W8, W8) () (W8, W8) ()
csa (a, b, c) = do
                  i ← signal (f a b c)
                  csa i     -- N.b., tail-recursive
```

Stream Semantics [NFM23]

```
((40,25,20),(),(0,0)), ((41,25,20),(),(48,37)), ((40,25,20),(),(50,36)), ...
```

tick0 *tick1* *tick2*

Carry-Save Adders in ReWire

Carry-Save Addition (CSA) as Pure Function

```
f :: W8 → W8 → W8 → (W8, W8)
f a b c     = ( ((a & b) || (a & c) || (b & c) ) << '0' , a ⊕ b ⊕ c )
```

CSA Device in ReWire

```
csa :: (W8, W8, W8) → Re (W8, W8, W8) () (W8, W8) ()
csa (a, b, c) = do
                  i ← signal (f a b c)
                  csa i     -- N.b., tail-recursive
```

Stream Semantics [NFM23]

(48,37) = f 40 25 20

((40,25,20),(),(0,0)), ((41,25,20),(),(48,37)), ((40,25,20),(),(50,36)), ...

(50,36) = f 41 25 20

Carry-Save Adders in ReWire

Carry-Save Addition (CSA) as Pure Function

```
f :: W8 → W8 → W8 → (W8, W8)
f a b c    = ( ((a & b) || (a & c) || (b & c) ) << '0' , a ⊕ b ⊕ c )
```

CSA Device in ReWire

```
csa :: (W8, W8, W8) → Re (W8, W8, W8) () (W8, W8) ()
csa (a, b, c) = do
                 i ← signal (f a b c)
                 csa i      -- N.b., tail-recursive
```

ReWire Compiler

```
$ rwc CSA.hs --verilog
$ ls -l CSA.v
  -rw-r--r--  1 william.harrison  staff  2159 Nov 14 08:33 CSA.v
```

Carry-Save Adders in ReWire

“Curried” CSA takes inputs one per cycle

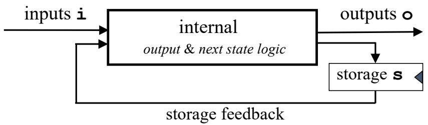
```
data Ans a = DC | Val a     -- "don't care" and "valid"
pcsa :: W8 → Re W8 () (Ans (W8, W8)) ()
pcsa a      = do
                b  ← signal DC
                c  ← signal DC
                a' ← signal (Val (f a b c))
                pcsa a'
```

Stream Semantics

```
(40,(),DC), (25,(),DC), (20,(),DC), (41,(),Val (48,37)), ...
```

Semantics & Staging Functions

Mealy Machine



Corresponding ReWire monad

```
type M s         = StateT s Identity
 -- ReWire monad
type Re i s o = ReacT i o (M s)
 -- consume/produce inputs & outputs synchronously
signal :: o → Re i s o i
```

- Formal Semantics [NFM23] is stream of “snapshots” : `Stream (i , s , o)`
- Staging Function*s*

```
stage :: M s a → Re i s (Maybe o) i
stage x = do
             lift x
             i' ← signal Nothing
             return i'
```

Correct-by-construction Cryptographic Hardware via Explicit Staging Transformations

Intuitive Storyboard of Technique

Imperative Algorithm

$$\backslash\; a_1\; a_2\; a_3 \rightarrow$$
$$\textbf{do}$$
$$x_1\; a_1$$
$$x_2\; a_2$$
$$x_3\; a_3$$

- Pseudocode Transliterated to Haskell
- "Imperative" ⇒ use State Monad

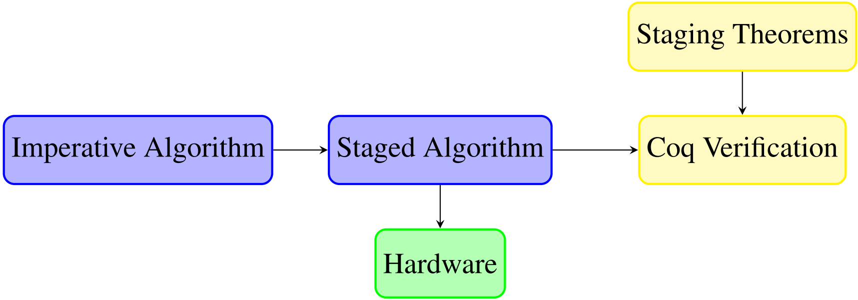
Staged Algorithm in ReWire

$$\backslash\; a_1 \rightarrow$$
$$\textbf{do}$$
$$a_2 \leftarrow \texttt{stage}\;(x_1\; a_1)$$
$$a_3 \leftarrow \texttt{stage}\;(x_2\; a_2)$$
$$\texttt{stage}\;(x_3\; a_3)$$

- Performant HW via ReWire compiler
- Coq Theorems relate $\texttt{stage}(x_i)$ to x_i

Correct-by-construction Cryptographic Hardware via Explicit Staging Transformations

Intuitive Storyboard of Technique



Today: BLAKE2

Background

- Cryptographic hash function
  - Input: message blocks of 16 64-bit words
  - Output: 8 64-bit words
- Can be used for pseudorandom number generation, e.g., in openFHE library
- Defined as imperative pseudocode in
  - *RFC 7693: BLAKE2 Cryptographic Hash and Message Authentication Function*

Cryptographic Functions in ReWire

Functions are just Functions

Blake2 Mixing Function*

```
FUNCTION G( v[0..15], a, b, c, d, x, y )
|
|   v[a] := (v[a] + v[b] + x) mod 2**w
|   v[d] := (v[d] ^ v[a]) >>> R1
|   v[c] := (v[c] + v[d])     mod 2**w
|   v[b] := (v[b] ^ v[c]) >>> R2
|   v[a] := (v[a] + v[b] + y) mod 2**w
|   v[d] := (v[d] ^ v[a]) >>> R3
|   v[c] := (v[c] + v[d])     mod 2**w
|   v[b] := (v[b] ^ v[c]) >>> R4
|
|   RETURN v[0..15]
|
END FUNCTION.
```

**RFC 7693: BLAKE2 Cryptographic Hash and Message Authentication Function*

Cryptographic Functions in ReWire

Functions are just Functions

Blake2 Mixing Function*

```
FUNCTION G( v[0..15], a, b, c, d, x, y )
|
|   v[a] := (v[a] + v[b] + x) mod 2**w
|   v[d] := (v[d] ^ v[a]) >>> R1
|   v[c] := (v[c] + v[d])     mod 2**w
|   v[b] := (v[b] ^ v[c]) >>> R2
|   v[a] := (v[a] + v[b] + y) mod 2**w
|   v[d] := (v[d] ^ v[a]) >>> R3
|   v[c] := (v[c] + v[d])     mod 2**w
|   v[b] := (v[b] ^ v[c]) >>> R4
|
|   RETURN v[0..15]
|
END FUNCTION.
```

ReWire Realization (pretty printed by hand)

```
_G :: Reg → Reg → Reg → Reg → Reg → Reg → M ()
_G a b c d x y = do
    a <== a + b + x
    d <== (d ^ a) >>> _R1
    c <== c + d
    b <== (b ^ c) >>> _R2
    a <== a + b + y
    d <== (d ^ a) >>> _R3
    c <== c + d
    b <== (b ^ c) >>> _R4
```

**RFC 7693: BLAKE2 Cryptographic Hash and Message Authentication Function*

Checking against RFC7369

Screenshot from RFC7693, Appendix A

```
BLAKE2b-512("abc") = BA 80 A5 3F 98 1C 4D 0D 6A 27 97 B6 9F 12 F6 E9
                     4C 21 2F 14 68 5A C4 B7 4B 12 BB 6F DB FF A2 D1
                     7D 87 C5 39 2A AB 79 2D C2 52 D5 DE 45 33 CC 95
                     18 D3 8A A8 DB F1 92 5A B9 23 86 ED D4 00 99 23
```

Checking against RFC7369

Screenshot from RFC7693, Appendix A

```
BLAKE2b-512("abc") = BA 80 A5 3F 98 1C 4D 0D 6A 27 97 B6 9F 12 F6 E9
                     4C 21 2F 14 68 5A C4 B7 4B 12 BB 6F DB FF A2 D1
                     7D 87 C5 39 2A AB 79 2D C2 52 D5 DE 45 33 CC 95
                     18 D3 8A A8 DB F1 92 5A B9 23 86 ED D4 00 99 23
```

Run Tests in Haskell

```
$ ghci Blake2b-reference.hs
GHCi, version 9.2.5: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling (Blake2b-reference.hs, interpreted )
ghci> _BLAKE2b_512 "abc"

                BA 80 A5 3F 98 1C 4D 0D 6A 27 97 B6 9F 12 F6 E9
                4C 21 2F 14 68 5A C4 B7 4B 12 BB 6F DB FF A2 D1
                7D 87 C5 39 2A AB 79 2D C2 52 D5 DE 45 33 CC 95
                18 D3 8A A8 DB F1 92 5A B9 23 86 ED D4 00 99 23
```

Correct-by-Construction Cryptographic Hardware via Explicit Staging Transformations

Blake2 Function*

```
FUNCTION F( h[0..7], m[0..15], t, f )
|
|      // Initialize local work vector v[0..15]
|      ...
|      v[12] := v[12] ^ (t mod 2**w)
|      v[13] := v[13] ^ (t >> w)
|      IF f = TRUE THEN
|      |   v[14] := v[14] ^ 0xFF..FF
|      END IF.
|
|      // Cryptographic mixing
|      ...
|
|      FOR i = 0 TO 7 DO
|      |   h[i] := h[i] ^ v[i] ^ v[i + 8]
|      END FOR.
|
|      RETURN h[0..7]
|
END FUNCTION.
```

ReWire Realization

```
_F :: W 128 → Bit → M ()
_F t f = do
              init_work_vector
              V12 <== V12 ^ lowword t
              V13 <== V13 ^ highword t
              if f then
                  V14 <== V13 ^ 0xF...F
                else
                  return ()
              cryptomixing
              xor_two_halves
```

* From: *RFC 7693: BLAKE2 Cryptographic Hash and Message Authentication Function*

Correct-by-Construction Cryptographic Hardware via Explicit Staging Transformations

Blake2 Function*

```
FUNCTION F( h[0..7], m[0..15], t, f )
|
|       // Initialize local work vector v[0..15]
|       ...
|       v[12] := v[12] ^ (t mod 2**w)
|       v[13] := v[13] ^ (t >> w)
|       IF f = TRUE THEN
|       |   v[14] := v[14] ^ 0xFF..FF
|       END IF.
|
|       // Cryptographic mixing
|       ...
|
|       FOR i = 0 TO 7 DO
|       |   h[i] := h[i] ^ v[i] ^ v[i + 8]
|       END FOR.
|
|       RETURN h[0..7]
|
END FUNCTION.
```

(Staged) ReWire Realization

```
_F :: W 128 → Bit → Re ()
_F t f = do
    stage $ init_work_vector
            V12 <== V12 ^ lowword t
            V13 <== V13 ^ highword t
            if f then
                V14 <== V13 ^ 0xF...F
              else
                return ()
    stage   cryptomixing
    stage   xor_two_halves
```

* From: *RFC 7693: BLAKE2 Cryptographic Hash and Message Authentication Function*

Staging Theorems

Theorem (Staging Theorem)

For all snapshots $(\mathtt{i}\ ,\ \mathtt{s}\ ,\ \mathtt{o})$ *and input streams* $(\mathtt{i}' \triangleleft \mathtt{is})$,

$$\llbracket\, \mathtt{stage}\ \mathtt{x} >\!\!>\!\!= \mathtt{f}\, \rrbracket\, (\mathtt{i}, \mathtt{s}, \mathtt{o})\ (\mathtt{i}' \triangleleft \mathtt{is}) = (\mathtt{i},\ \mathtt{s},\ \mathtt{o}) \triangleleft \llbracket\, \mathtt{f}\, \rrbracket\ \mathtt{i}'\ (\mathtt{i}',\ \mathtt{s}'\ ,\ \mathtt{Nothing})\ \mathtt{is}$$
$$\textbf{where}$$
$$(\mathtt{a}\ ,\ \mathtt{s}') = \mathtt{runST}\, \llbracket\, \mathtt{x}\, \rrbracket\ \mathtt{s}$$

- Each flavor of `stage` has a similar theorem
- All are formalized and proved in Coq

*The symbol $\triangleleft$ is stream "cons".

Correctness Theorem*

- `refb2b` describes an imperative (state-monadic) version of BLAKE2b
- `stagedb2b` formalizes the action of the device on a single input
- Let `six` be the unrolling:

```
stagedb2b Start >>= stagedb2b >>= stagedb2b >>= stagedb2b >>= stagedb2b >>= stagedb2b
```

Theorem (Correctness)

$$\mathtt{out}_7\,(\llbracket \mathtt{six} \rrbracket\,(\mathtt{i},\,\mathtt{s},\,\mathtt{o})\,\mathtt{ins}) = \mathtt{fst}\,(\mathtt{runST}\,(\mathtt{refb2b}\,(\mathtt{m}_0,\,\mathtt{m}_1,\,\mathtt{m}_2,\,\mathtt{m}_3,\,\mathtt{p}))\,\mathtt{s})$$

where

$$\mathtt{ins} = \mathtt{m}_0 \lhd \mathtt{m}_1 \lhd \mathtt{m}_2 \lhd \mathtt{m}_3 \lhd \mathtt{p} \lhd \mathtt{is}$$

*Proved in Coq using staging theorems.

Summary & Conclusions

Correct-by-construction Cryptographic Hardware via Explicit Staging Transformations

IEEE Spectrum 12/22/23

Hardware Verification in the large

- DARPA DPRIVE Project with Duality; starting Phase 3
- Verifying Aggressively Optimized Hardware Accelerators for FHE
- See *Formalized High Level Synthesis with Applications to Cryptographic Hardware* [NASA Formal Methods 2023] for semantics, etc.

THANKS!