Temporal Staging for Correct-by-Construction Cryptographic Hardware*

Yakir Forman [1] Bill Harrison [2]

[1]Two Six Technologies, Inc.

[2]Idaho National Laboratories

* 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.

Temporal Staging for Correct-by-Construction Cryptographic Hardware

- Software *program transformation* approach [Burstall, Dijkstra, Scherlis,. . .] applied to deriving correct, performant *hardware*
  - Semantics-preserving transformations : *Reference* $\rightsquigarrow \ldots \rightsquigarrow$ *Implementation*
- Here, all transformations take place in a *functional HLS language*
  - Resulting formally verified hardware designs included in FHE accelerators currently being fabricated

Temporal Staging for Correct-by-Construction Cryptographic Hardware

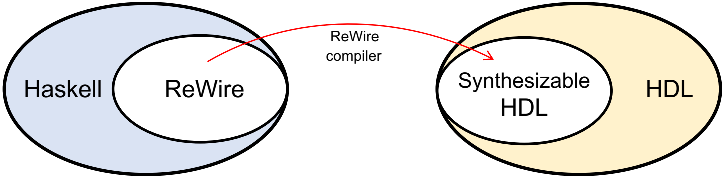
- Software *program transformation* approach [Burstall, Dijkstra, Scherlis,. . .] applied to deriving correct, performant *hardware*
  - Semantics-preserving transformations : *Reference* $\rightsquigarrow \ldots \rightsquigarrow$ *Implementation*
- Here, all transformations take place in a *functional HLS language*
  - Resulting formally verified hardware designs included in FHE accelerators currently being fabricated
- Meet my coauthor, Yakir Forman (who did 90% of the work):

Spring 2022

Localization and Cantor Spectrum for Quasiperiodic Discrete Schrödinger Operators with Asymmetric, Smooth, Cosine-Like Sampling Functions

Yakir Moshe Forman
Yale University Graduate School of Arts and Sciences, yakir.forman@yale.edu

ReWire Functional High Level Synthesis Language



- 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)
```

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
```

Stream Semantics [NFM23]

$$\underbrace{((40,25,20),(),(0,0))}_{tick0},\ \underbrace{((41,25,20),(),(48,37))}_{tick1},\ \underbrace{((40,25,20),(),(50,36))}_{tick2},\ \ldots$$

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

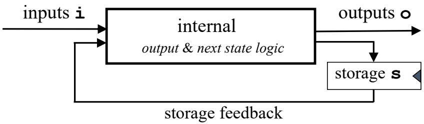
```
data Ans a = DC | Val a     -- "don't care" and "valid"
pcsa :: W8 → Re W8 () (Ans (W8, W8)) ()
pcsa a     = do
               b  ← signal DC
               c  ← signal DC
               a' ← signal (Val (f a b c))
               pcsa a'
```

Stream Semantics

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

Semantics & Staging Functions

Mealy Machine



Corresponding ReWire monad

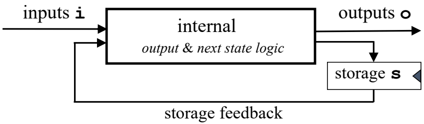
```
type M s       = StateT s Identity
 -- ReWire monad
type Re i s o = ReacT i o (M s)
 -- consume/produce inputs & outputs synchronously
signal :: o → Re i s o i
```

- Formal Semantics [NFM23] is stream of "snapshots" : $Stream\ (i\ ,\ s\ ,\ o)$
- Staging Functions

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

Semantics & Staging Functions

Mealy Machine

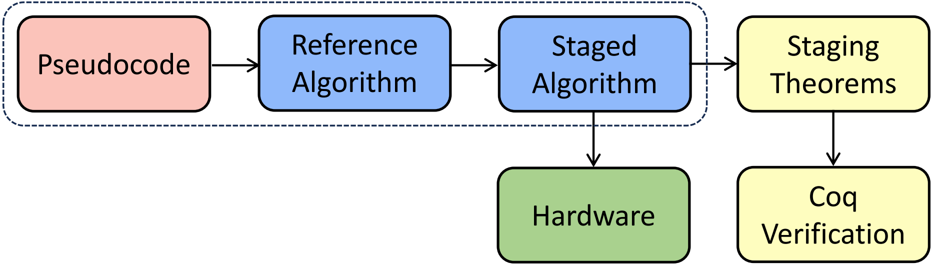


Corresponding ReWire monad

```
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 Functions
  - `(stage x)` turns computation `x` into single cycle of hardware device

Temporal Staging Methodology



- Staging transformation: applying `stage` functions to parts of reference algorithm
- `stage` functions are *akin* to lift functions of monad transformers

Temporal Staging Methodology

Imperative Algorithm

$\backslash\ a_1\ a_2\ a_3 \rightarrow$

$\quad$ **do**

$\qquad x_1\ \ a_1$

$\qquad x_2\ \ a_2$

$\qquad x_3\ \ a_3$

- Pseudocode Transliterated to Haskell
- "Imperative" $\Rightarrow$ use State Monad

Staged Algorithm in ReWire

$\backslash\ a_1 \rightarrow$

$\quad$ **do**

$\qquad a_2 \leftarrow \mathtt{stage}\ (x_1\ \ a_1)$

$\qquad a_3 \leftarrow \mathtt{stage}\ (x_2\ \ a_2)$

$\qquad \mathtt{stage}\ (x_3\ \ a_3)$

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

BLAKE2 Case Study

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 Pseudocode *

```
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 Pseudocode *

```
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.
```

Reference in ReWire (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
```

Pseudocode ⇝ Reference Algorithm ⇝ Staged Algorithm

Blake2 Function Pseudocode*

```
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.
```

Reference Algorithm

```
_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
```

Pseudocode ⇝ Reference Algorithm ⇝ Staged Algorithm

Blake2 Function Pseudocode*

```
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 Algorithm

```
_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
```

Staging Theorems

Theorem (Staging Theorem)

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

$$[\![\, \mathtt{stage}\ \mathtt{x} >\!\!>= \mathtt{f} \,]\!]\ (\mathtt{i}, \mathtt{s}, \mathtt{o})\ (\mathtt{i}' \lhd \mathtt{is}) = (\mathtt{i},\ \mathtt{s},\ \mathtt{o}) \lhd [\![\, \mathtt{f}\ \mathtt{a} \,]\!]\ (\mathtt{i}',\ \mathtt{s}'\ ,\ \mathtt{Nothing})\ \mathtt{is}$$
$$\quad \textbf{where}$$
$$\qquad (\mathtt{a}\ ,\ \mathtt{s}') = \mathtt{runST}\ [\![\, \mathtt{x} \,]\!]\ \mathtt{s}$$

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

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

Correctness Theorem*

- `refb2b`: reference version of BLAKE2b
- `cycle` formalizes the action of the device on a single input
- Let `staged` be the unrolling:

```
staged = cycle Start >>= cycle >>= cycle >>= cycle >>= cycle >>= cycle
```

Theorem (Correctness)

Staged and Reference Algorithms compute identical values on identical inputs; i.e.,

$$o = a$$

where

$$
\begin{aligned}
(\mathtt{a}\,,\,\_) &= \mathtt{runST}\,(\mathtt{refb2b}\,(\mathtt{m}_0,\,\mathtt{m}_1,\,\mathtt{m}_2,\,\mathtt{m}_3,\,\mathtt{p}))\;\mathtt{s} \\
(\underbrace{\_ \lhd \ldots \lhd \_}_{\text{wait 6 cycles}} \lhd (\_,\_,\mathtt{o})) &= [\![\mathtt{staged}]\!]\;(\mathtt{i},\,\mathtt{s},\,\mathtt{o})\;(\mathtt{m}_0 \lhd \mathtt{m}_1 \lhd \mathtt{m}_2 \lhd \mathtt{m}_3 \lhd \mathtt{p} \lhd \mathtt{is})
\end{aligned}
$$

Summary, Conclusions, & Future Work

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.

Summary, Conclusions, & Future Work

- Summary
  - Functional HLS is a vector for transferring software science to hardware design
  - Temporal Staging slices computations by clock cycle, whereas classic SW staging separates static from dynamic [Scherlis,Taha,. . .]
- Related Work
  - Previous work [NFM23] used ReWire to model/verify complex, highly optimized Verilog designs for FHE
  - Here, we use ReWire for design, formal verification, *and* implementation
- Performance
  - Extensive performance evaluation TBD
  - . . .although individual designs (e.g., BLAKE2b) are sufficiently performant [Moore23] to be included with ASICs currently in fabrication
- Future Work
  - At Two Six, other crypto algorithms (e.g., AES256) have been implemented & formally verified using Temporal Staging in ReWire

THANKS!