# High assurance crypto-development with Cryptol

Cryptol Team | May 2nd, 2011
Levent Erkök

*The Cryptol team, past and present:*
Sally Browning, Magnus Carlsson, Levent Erkök, Sigbjorn Finne, Andy Gill, Fergus Henderson, John Launchbury, Jeff Lewis, Lee Pike, John Matthews, Thomas Nordin, Mark Shields, Joel Stanley, Frank Seaton Taylor, Jim Teisher, Philip Weaver, Adam Wick

| galois |

## About Galois

high assurance
research and development

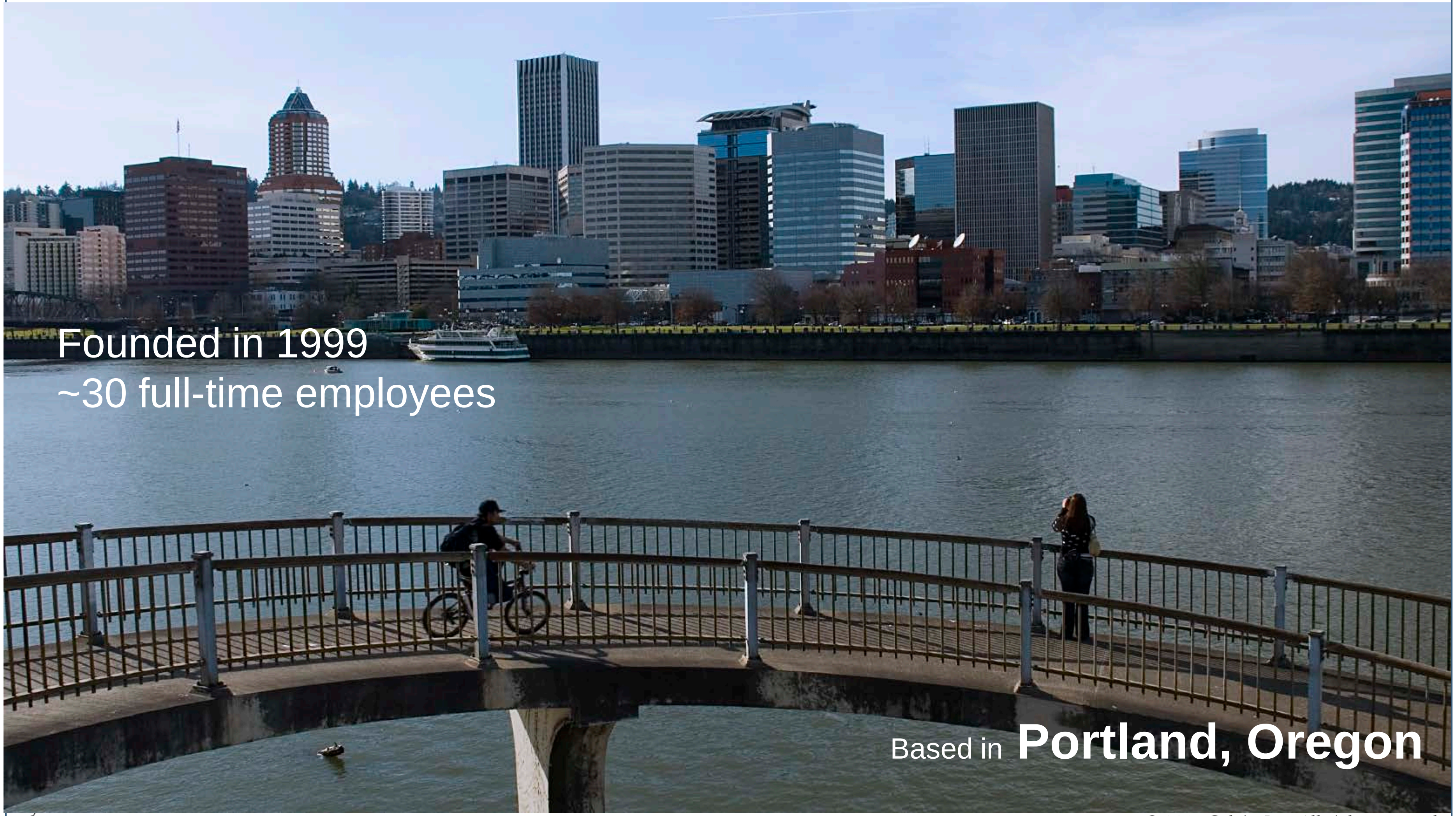
Creating trustworthiness in critical systems

Galois [gal-wah]

*Named after French mathematician Évariste Galois*

| galois |

# Company facts



Founded in 1999
~30 full-time employees

Based in **Portland, Oregon**

|galois|

# Offering overview

## CLIENT SERVICES

Trusted software development
for government and commercial clients

- Formal Methods
- Language and Tools Design
- Systems Engineering
- Trustable Software

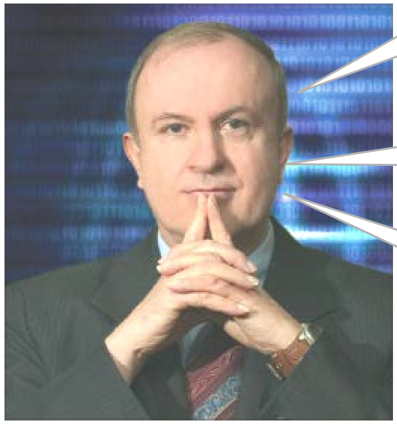
## TECHNOLOGY SOLUTIONS

Technologies that provide information
assurance to the most challenging
systems and software environments

- Cross-domain Solutions
- Trusted Collaboration
- Communications Security
- Evaluation Tools

| galois |

## The problem



“Of the 1.3 million cryptographic devices in the U.S. inventory, 73 percent will be replaced over the next 15 years ...”

“… a severe lack of diagnostic capabilities to evaluate software products to detect unintentional vulnerabilities and maliciously implanted functionality in a timely and cost-effective manner.”

“Software evaluation is very manpower intensive and doesn’t scale.”

**Daniel G. Wolf**

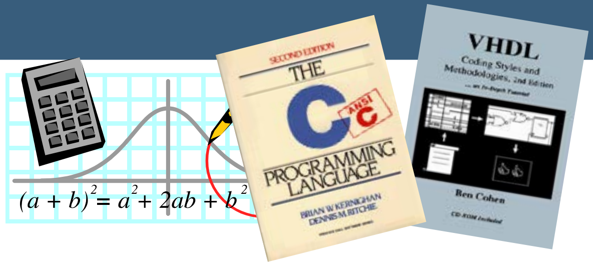
**Former director**
**Information Assurance Directorate**
**National Security Agency**
**January, 2006**

## Challenge: to support the correct of implementations of crypto-algorithms

- Algorithm V&V is critical in crypto-modernization
  - Must manage assurance in face of exploding complexity and demands
- Not just the NSA / DoD
  - 48% of crypto-modules, and 27% of crypto-algorithms had flaws.
  - Without evaluation, about 50-50 chance of buying correct crypto
    - NIST Computer Security Division, 2008 Annual report (page 15)

| galois |

# Contributing factors


$(a + b)^2 = a^2 + 2ab + b^2$
SECOND EDITION
THE C PROGRAMMING LANGUAGE
ANSI C
BRIAN W KERNIGHAN
DENNIS M RITCHIE
VHDL
Coding Styles and Methodologies, 2nd Edition
Ben Cohen
CD-ROM Included

***Creating a crypto algorithm requires skills in math AND programming***

***Variety of target architectures***

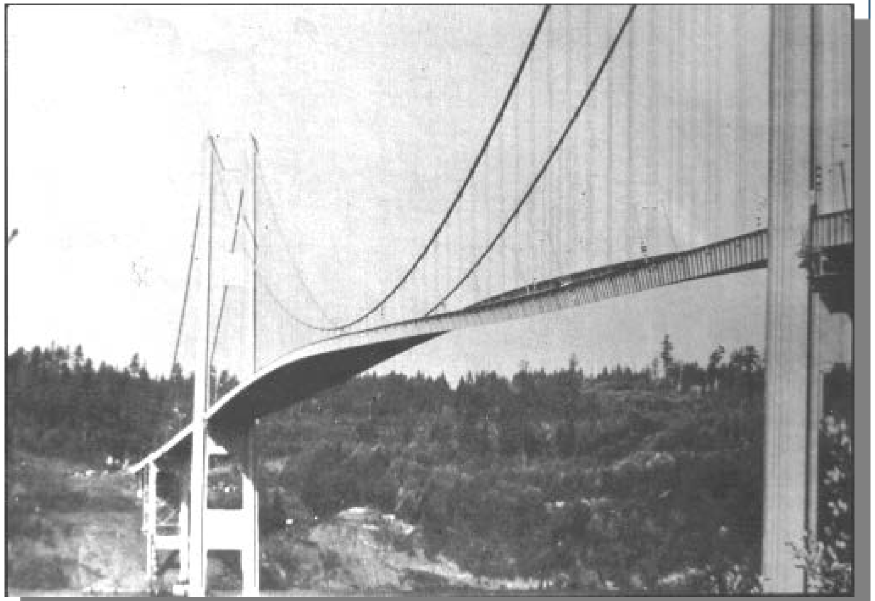
***Validation is complex and tedious***

***Variety of requirements***

| galois |

## A vision for software

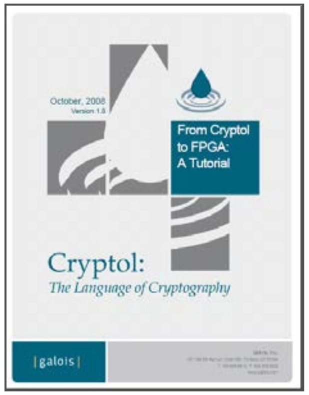
- Let the software itself be trustworthy
  - Software artifacts to speak for themselves
  - Rather than hoping to rely on the process that created them
- Use mathematical models to enable tractable analysis
  - Executable models and formal methods
  - A model is an abstraction that allows thought at a higher level
- Follow open standards
  - Build components with high internal integrity
  - Maximize interoperability

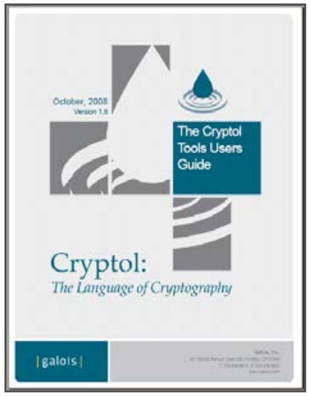


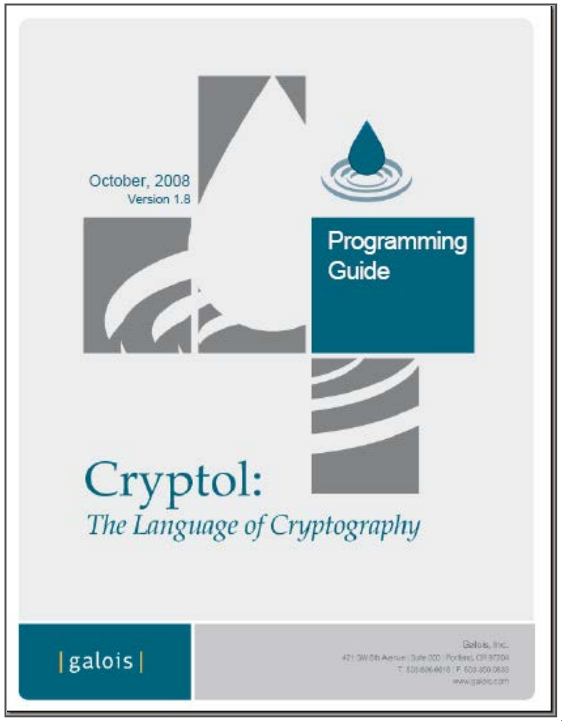
We want to see software built with the same diligence and analysis as other engineers build bridges

# Approach: Specifications and formal methods

- Declarative specification language
  - Language tailored to the crypto domain
  - Designed with feedback from NSA
- Execution and Validation Tools
  - Tool suite for different implementation and verification applications
  - In use by crypto-implementers


October, 2008
Version 1.8
From Cryptol to FPGA: A Tutorial
Cryptol:
*The Language of Cryptography*
galois


October, 2008
Version 1.8
The Cryptol Tools Users Guide
Cryptol:
*The Language of Cryptography*
galois


October, 2008
Version 1.8
Programming Guide
Cryptol:
*The Language of Cryptography*
galois
Galois, Inc.
421 SW 6th Avenue | Suite 300 | Portland, OR 97204
T 503.626.6616 | F 503.350.0833
www.galois.com

## Cryptol Project Mission: *To reduce the cost (in both time and money) of developing and certifying cryptographic applications*

Cryptol Specification

### A Domain Specific Specification Language
- Precise, Declarative Semantics
- High level design exploration

### Automated Synthesis down to FPGA and VHDL verification
- Algebraic rewrite-based compilation
- Traceability back to specification
- Verifying generated and 3rd party VHDL

Certificate of Trust

This is to certify that

Cryptol Specification

can be trusted

[illegible]

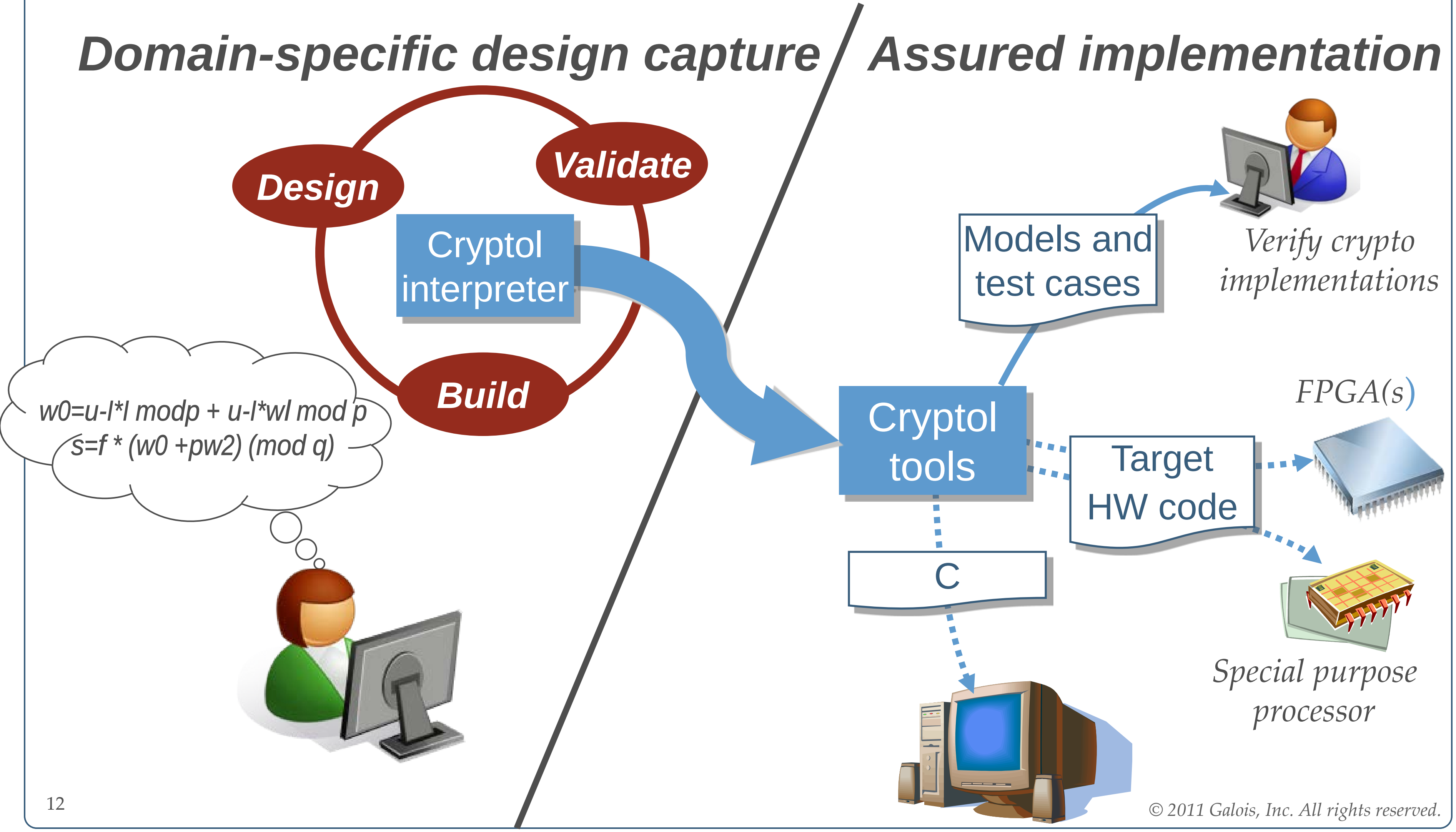
Certificate Authority

### Property specification and verification
- SAT/SMT based property verification
- Push button assurance

## Cryptol language design goals

- High-level domain-specific language (DSL) for cryptographic algorithms
  - Specify algorithms precisely and unambiguously
  - But also be executable
- Use Cryptol specifications to guide and document crypto implementations
  - And even generate them
- Be neutral as to implementation platform
  - Don’t bake in Von Neumann assumptions
  - Same Cryptol specification can be compiled to multiple architectures
- Have a clean, unambiguous semantics
  - Essential for any specification language
  - Forms the basis for verification

# One specification - Many uses



| galois |

## Cryptol programs

- File of mathematical definitions
  - Bit-precise type signatures
  - Strong static typing, with size and shape polymorphism
- Definitions are computationally neutral
  - Cryptol tools provide the computational content (interpreters, compilers, code generators, verifiers)

```
x : [4][32];
x = [23 13 1 0];

F : ([16],[16]) -> [16];
F (x,y) = 2 * x + y;
```

## Data types: Numbers and sequences

- Numbers, a.k.a. “words”
  - 123, 0xF4, 0b11110100
  - Types: 75 : [8]
- Homogeneous sequences
  - [False True False True False False True]
  - [[1 2 3 4] [5 6 7 8]]
  - Can be finite or infinite

|galois|

## Data types: Collections

- Heterogenous data can be grouped together into tuples
  - (13, “hello”, True)
- Or, into records: Tuples with named fields
  - {x : [8]; y : [8]}
- Values can be arbitrarily nested:
  - A sequence of records:
    - [{x=12; y=5} {x=16; y=8}]☐
- Functions can take/return arbitrary values
- What Cryptol doesn’t have:
  - Pointers, null or any other kind
  - Mutable objects, side effects
  - Cryptol is “purely functional”

|galois|

## Sequences

- Sequence operators
  - Concatenation (#), indexing (@)

    [1..5] # [3 6 8]  =  [1 2 3 4 5 3 6 8]

    [50 .. 99] @ 10  =  60

- Shifts and Rotations
  - Shifts (**`<<, >>`**), Rotations (**`<<<, >>>`**)

    [1 2 3 4] << 2  =  [3 4 0 0]

    [1 2 3 4] <<< 2 =  [2 3 1 2]

galois

## Standard operations

- Arithmetic operators
  - Result is modulo the word size of the arguments
  - `+ - * / % **`
- Boolean operators
  - From bits, to arbitrarily nested matrices of the same shape
  - `& | ^ ~`
- Comparison operators
  - Equality, order
  - `== != < <= > >=`
  - returns a Bit
- Conditional operator
  - Expression-level *if-then-else*

|galois|

## Cryptol types

- Types express size and shape of data

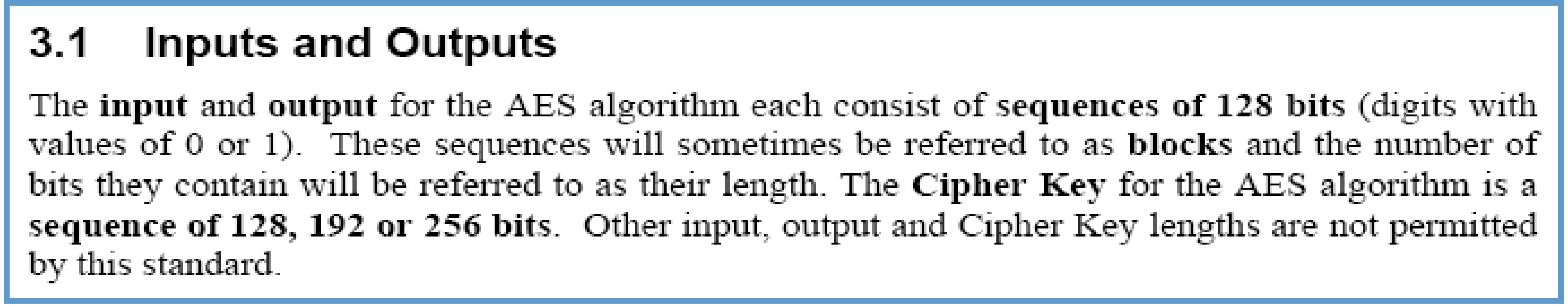
```
[[0x1FE 0x11] [0x132 0x183] [0x1B4 0x5C] [0x26  0x7A]]
                  has type      [4][2][9]
```

- Type inference
  - Use type declarations for active documentation
  - All other types computed
- Parametric polymorphism
  - Express size parameterization of algorithms

| galois |

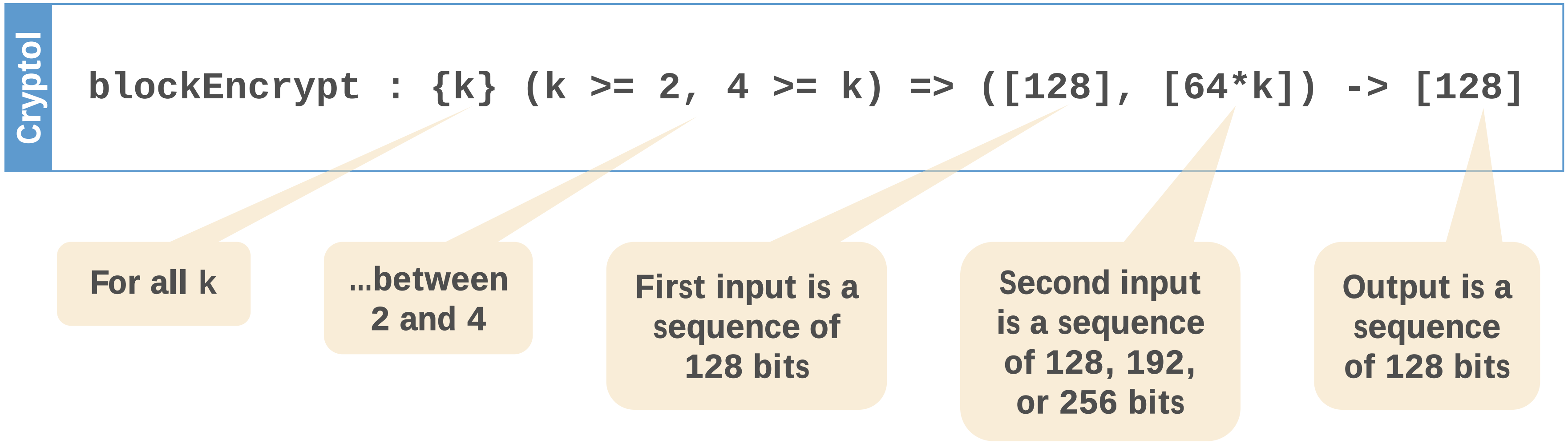
## Cryptol: Specify interfaces unambiguously

From the Advanced Encryption Standard definition†



### 3.1 Inputs and Outputs

The **input** and **output** for the AES algorithm each consist of **sequences of 128 bits** (digits with values of 0 or 1). These sequences will sometimes be referred to as **blocks** and the number of bits they contain will be referred to as their length. The **Cipher Key** for the AES algorithm is a **sequence of 128, 192 or 256 bits**. Other input, output and Cipher Key lengths are not permitted by this standard.

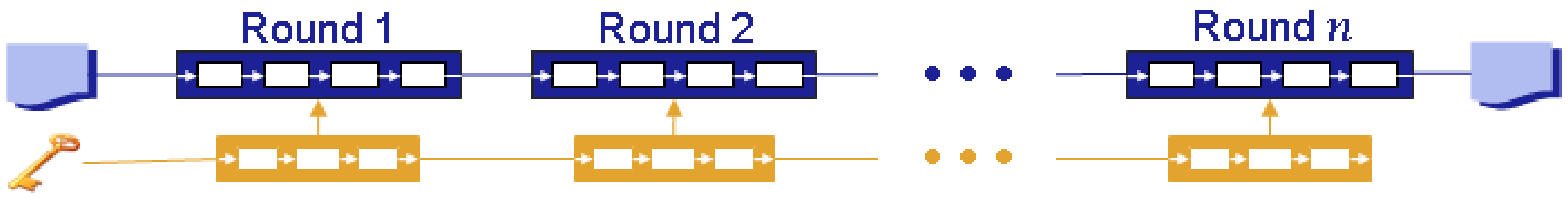


```
blockEncrypt : {k} (k >= 2, 4 >= k) => ([128], [64*k]) -> [128]
```

Cryptol
For all k
...between 2 and 4
First input is a sequence of 128 bits
Second input is a sequence of 128, 192, or 256 bits
Output is a sequence of 128 bits

†http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

# Cryptol: Expressing data-flow dependencies



Cryptol

```
encrypt128 : ([4][32],[4][4][8]) -> [4][4][8];
encrypt128 (initialKey, plainText) = cipherText where {
   roundKeys = [ initialKey ] # [| nextKey (round, prev)
                                  || round <- [1..10]
                                  || prev <- roundKeys
                                  |];
   initialState = first(roundKeys) ^ plainText;
   rounds = [ initialState ] # [| nextState (prev, roundKey, round)
                                 || round <- [1..10]
                                 || prev <- rounds
                                 || roundKey <- drop (1, roundKeys)
                                 |];
   cipherText = last(rounds);
};
```

| galois |

## AES/Rijndael API

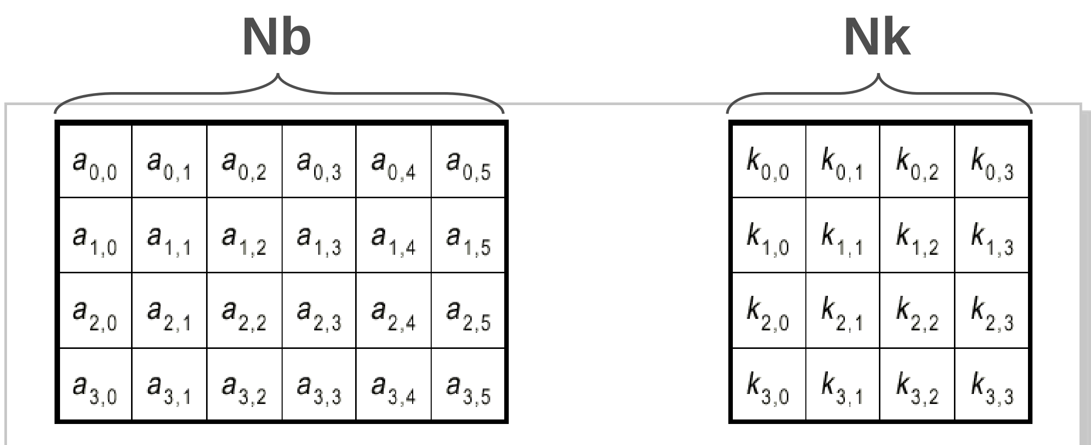


Figure 1: Example of State (with Nb = 6) and Cipher Key (with Nk = 4) layout.

keySchedule : [4*Nk][8] -> *Xkey*

encrypt : (*Xkey*, [4*Nb][8]) -> [4*Nb][8]

decrypt : (*Xkey*, [4*Nb][8]) -> [4*Nb][8]

*type Xkey* = ( [4][Nb][8]

            , [max(Nb,Nk)+5][4][Nb][8]

            , [4][Nb][8])

## Splitting and joining sequences

$$\begin{tikzcd}
\mathbf{0x99FAC6F975BABB3E} \arrow[d, "\textbf{split}"] \\
\mathbf{[0x99\ 0xFA\ 0xC6\ 0xF9\ 0x75\ 0xBA\ 0xBB\ 0x3E]} \arrow[d, "\textbf{join}"] \\
\mathbf{0x99FAC6F975BABB3E}
\end{tikzcd}$$

*Polymorphic operation: use a type to resolve how many terms in the split list*

| galois |

## Sequence comprehensions

- The comprehension notion borrowed from set theory
- Applying an operation to each element

```
[| 2*x + 3  ||  x <- [1 2 3 4] |]

 = [5 7 9 11]
```

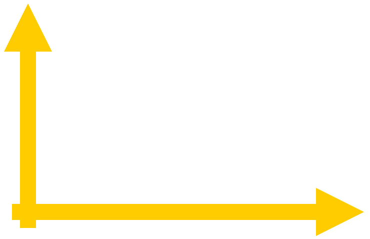
|galois|

## Traversals

- Cartesian traversal

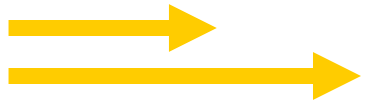
```
[| [x y] ||  x <- [0 1 2], y <- [3 4] |]

  = [[0 3] [0 4]
     [1 3] [1 4]
     [2 3] [2 4]]
```



- Parallel traversal

```
[| x + y ||  x <- [1 2 3]
         ||  y <- [3 4 5 6 7] |]
 =  [4 6 8]
```



# Row traversals in AES

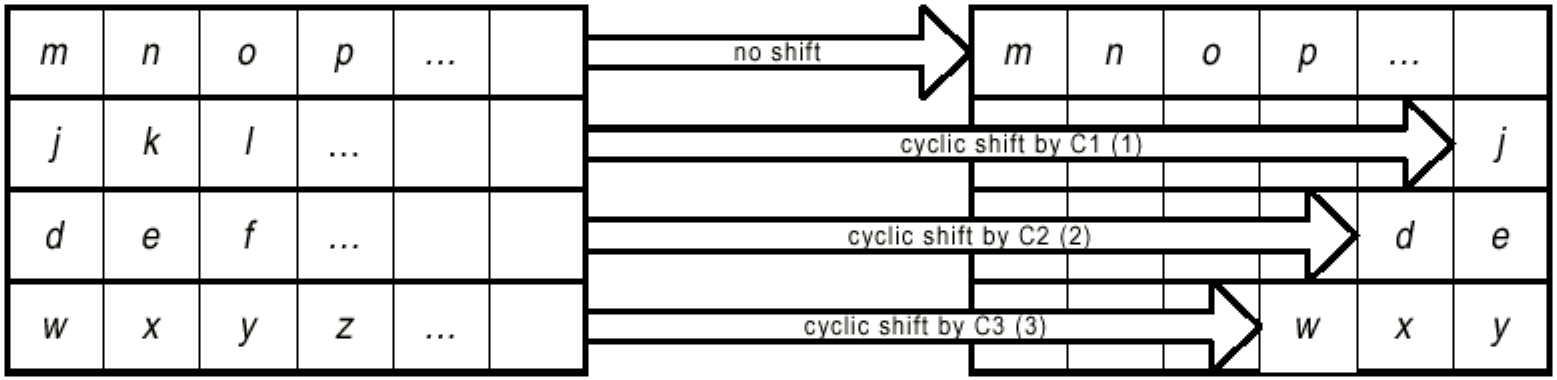


**Figure 3: ShiftRow operates on the rows of the State.**

```
ShiftRow : [4][Nb][8] -> [4][Nb][8];
ShiftRow(state)
  = [| row <<< i || row <- state
             || i <- [0 1 2 3] |];
```

# Column traversals

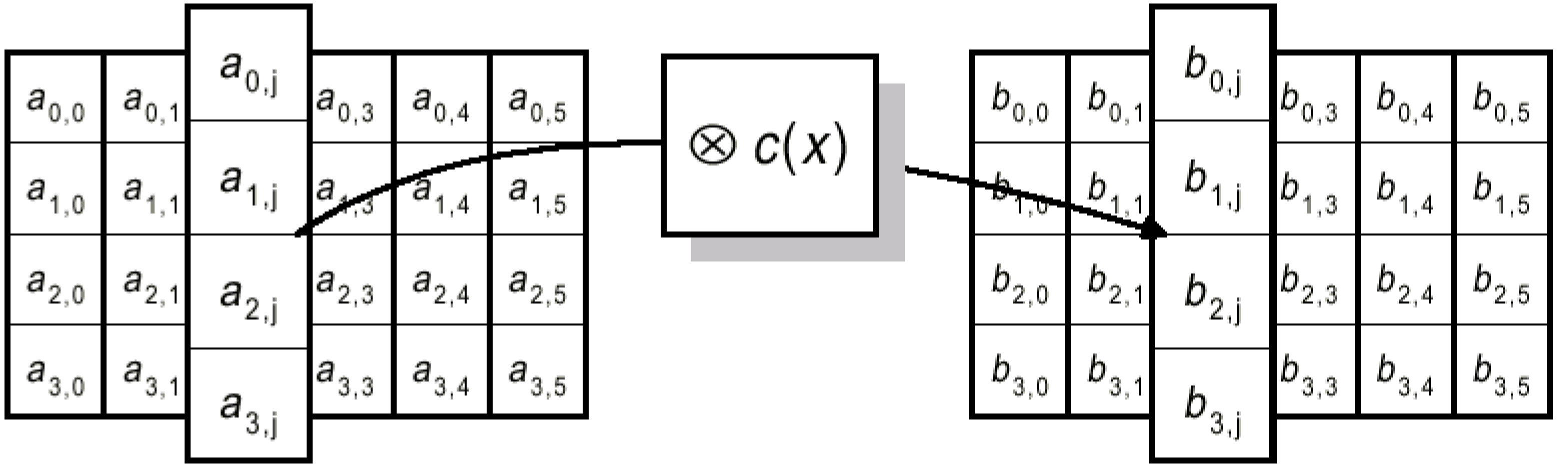
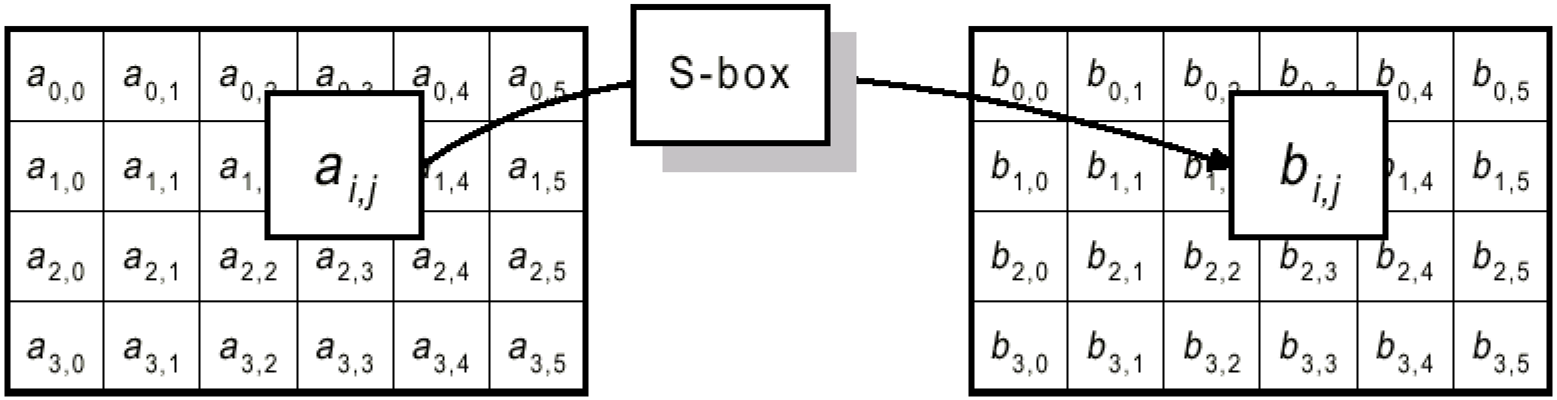

$a_{0,0}$ $a_{0,1}$ $a_{0,j}$ $a_{0,3}$ $a_{0,4}$ $a_{0,5}$
$a_{1,0}$ $a_{1,1}$ $a_{1,j}$ $a_{1,3}$ $a_{1,4}$ $a_{1,5}$
$a_{2,0}$ $a_{2,1}$ $a_{2,j}$ $a_{2,3}$ $a_{2,4}$ $a_{2,5}$
$a_{3,0}$ $a_{3,1}$ $a_{3,j}$ $a_{3,3}$ $a_{3,4}$ $a_{3,5}$
$\otimes c(x)$
$b_{0,0}$ $b_{0,1}$ $b_{0,j}$ $b_{0,3}$ $b_{0,4}$ $b_{0,5}$
$b_{1,0}$ $b_{1,1}$ $b_{1,j}$ $b_{1,3}$ $b_{1,4}$ $b_{1,5}$
$b_{2,0}$ $b_{2,1}$ $b_{2,j}$ $b_{2,3}$ $b_{2,4}$ $b_{2,5}$
$b_{3,0}$ $b_{3,1}$ $b_{3,j}$ $b_{3,3}$ $b_{3,4}$ $b_{3,5}$

**Figure 4: MixColumn operates on the columns of the State.**

```
MixColumn : [4][Nb][8] -> [4][Nb][8];
MixColumn(state)
  = transpose [| ptimes(col,cx)
            || col <- transpose(state)
            |]
```

| galois |

## Nested traversals



S-box
$a_{i,j}$
$b_{i,j}$
$a_{0,0}$ $a_{0,1}$ $a_{0,4}$ $a_{0,5}$
$a_{1,0}$ $a_{1,1}$ $a_{1,4}$ $a_{1,5}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$ $a_{2,5}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $a_{3,4}$ $a_{3,5}$
$b_{0,0}$ $b_{0,1}$ $b_{0,4}$ $b_{0,5}$
$b_{1,0}$ $b_{1,1}$ $b_{1,4}$ $b_{1,5}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$ $b_{2,4}$ $b_{2,5}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$ $b_{3,4}$ $b_{3,5}$

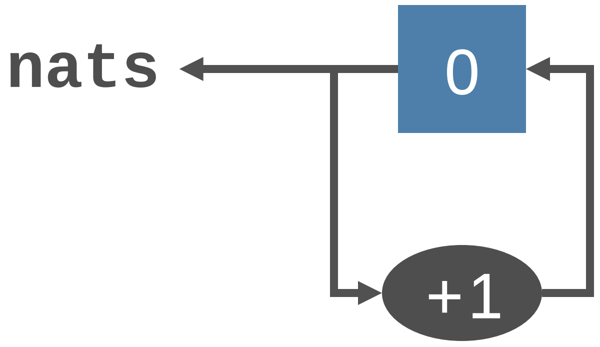
**Figure 2: ByteSub acts on the individual bytes of the State.**

```
ByteSub : [4][Nb][8] -> [4][Nb][8];
ByteSub(state) = [| [| sbox @ a || a <- row |]
                 || row <- state
                 |];
```

| galois |

## Recurrence

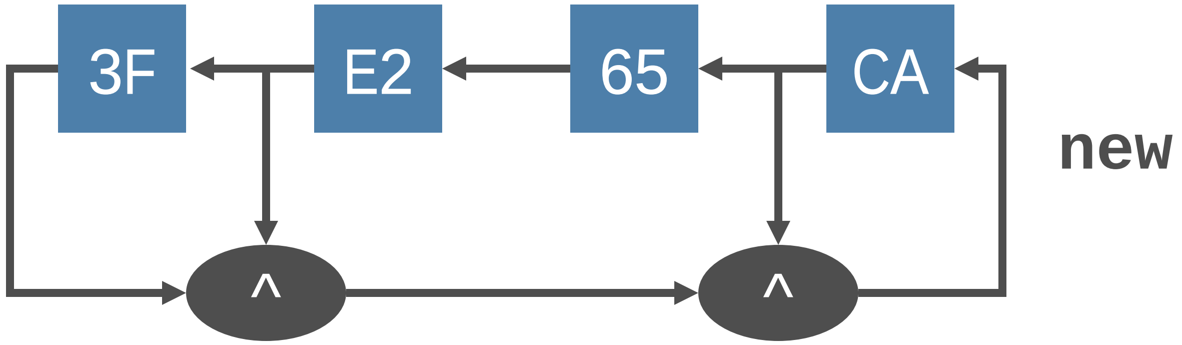
- Textual description of shift circuits
  - Follow mathematics: use *stream-equations*
  - Stream-definitions can be *recursive*
- nats = [0] # [| y+1 || y <- nats |];



| galois |

# More complex stream equations

```
as  = [Ox3F OxE2 Ox65 OxCA] # new;
new = [| a ^ b ^ c || a <- as
                   || b <- drop(1,as)
                   || c <- drop(3,as)  |];
```



## AES rounds

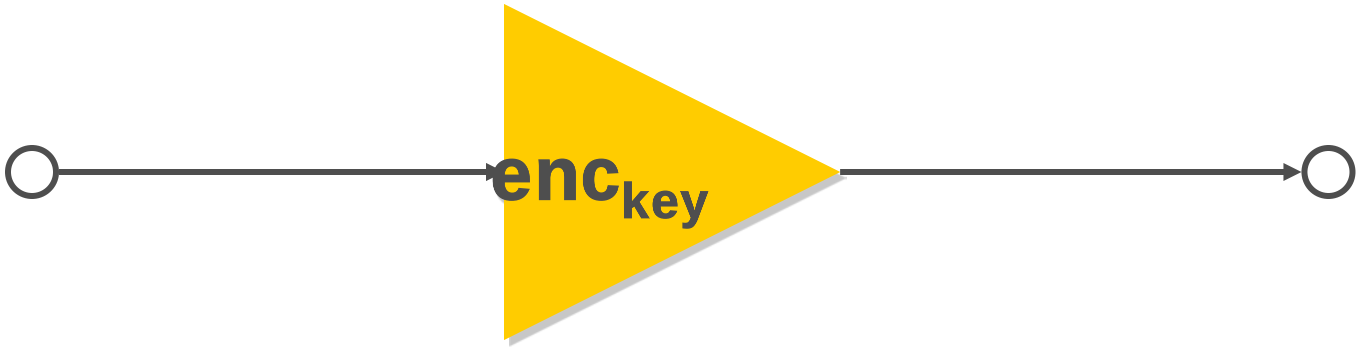
```
Rounds(State,(initialKey,rndKeys,finalKey)) = final
  where {
    istate = State ^ initialKey;
    rnds = [istate] # [| Round(state,key)
                      || state <- rnds
                      || key <- rndKeys |];
    final = FinalRound(last(rnds),finalKey);
  };

Round :([4][Nb][8],[4][Nb][8]) -> [4][Nb][8];
Round (State,RoundKey)
  = MixColumn(ShiftRow(ByteSub(State))) ^ RoundKey
```

## Modes: Electronic code book

- Modes are expressed in the same way as other cycles

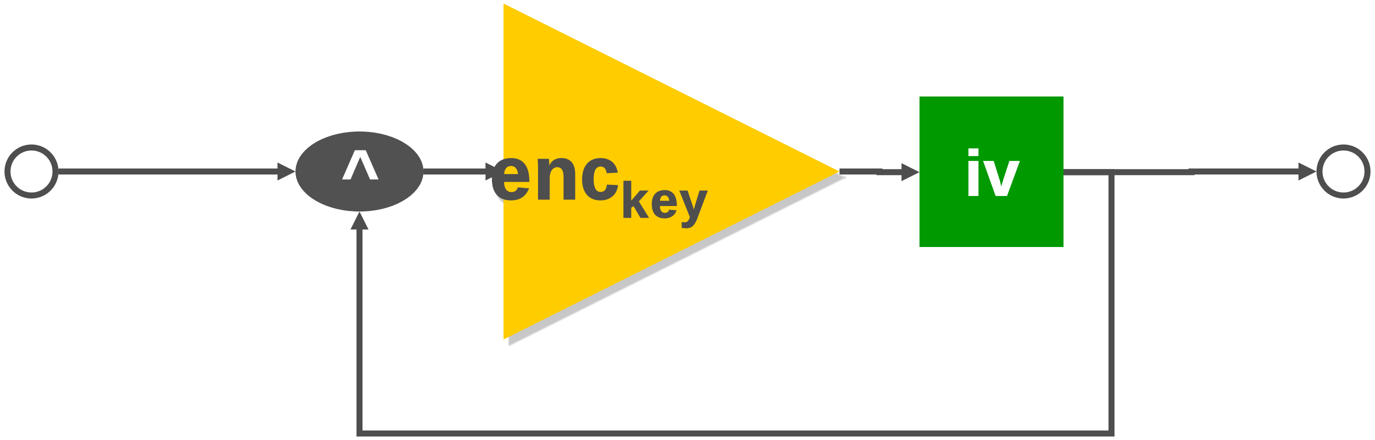
cts = [| encrypt (pt, key) || pt <- pts |]



galois

## Modes: Cipher block chaining

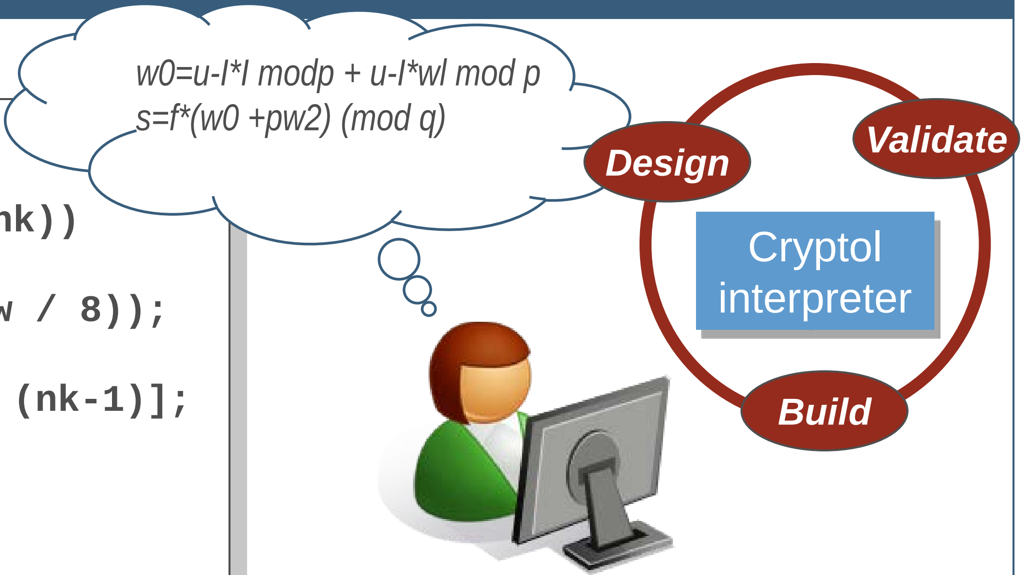
```
cts = [iv] # [| encrypt (pt^ct, key)
            || pt <- pts
            || ct <- cts
                       |]
```



| galois |

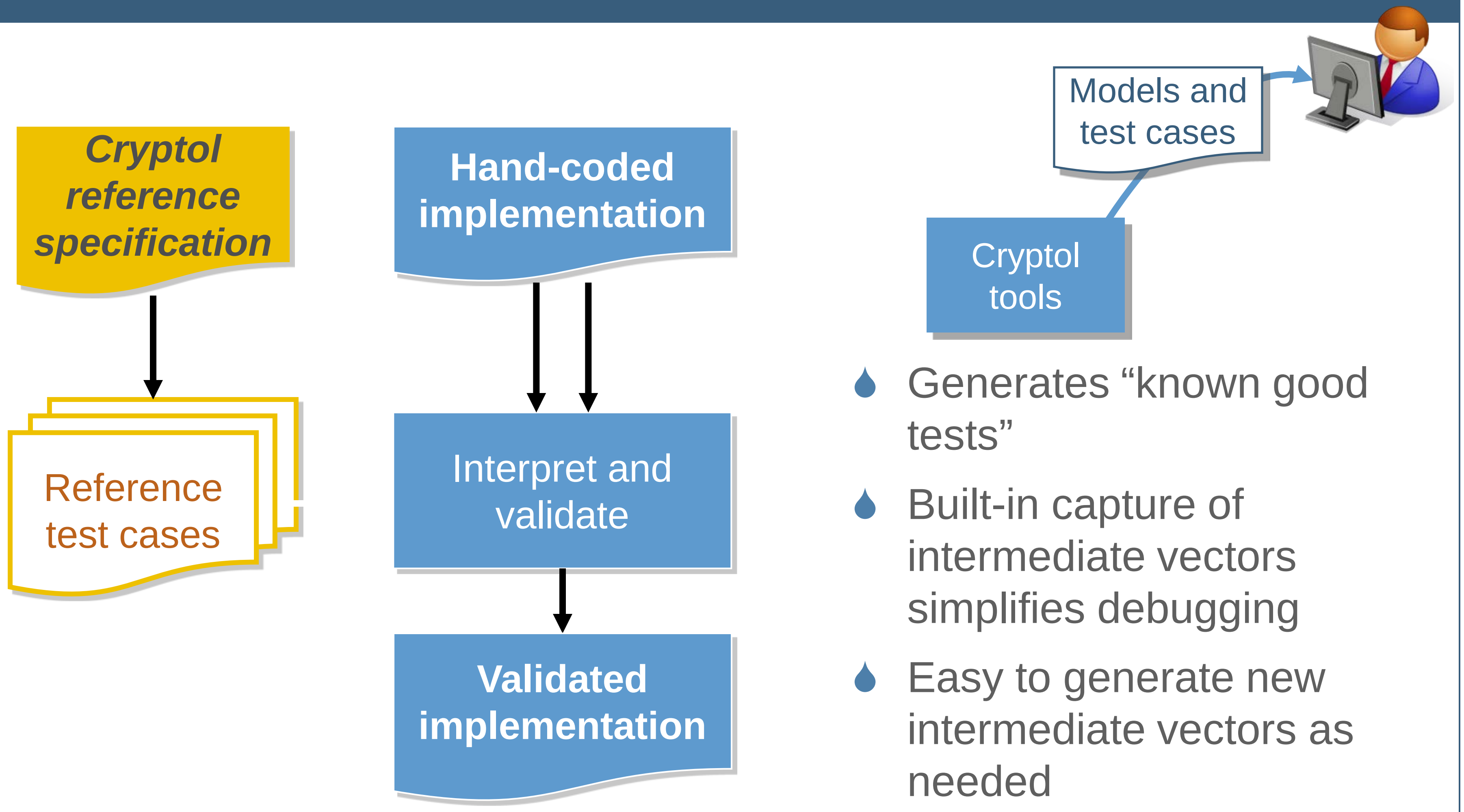
## Domain-specific design capture

```
rc6ks : {a} (w >= width a) =>
        [a][8] -> [r+2][2][w];
rc6ks key = split (rs >>> (v - 3 * nk))
  where {
    c = max (1, (width key + 3) / (w / 8));
    v = 3 * max (c, nk);
    initS =  [pw (pw+qw) ..]@@[0 .. (nk-1)];
    padKey : [4*c][8];
    padKey = key # zero;
    initL : [c][w];
    initL = split (join padKey);
    ss = [| (s+a+b) <<< 3
         || s <- initS # ss
         || a <- [0] # ss
         || b <- [0] # ls |];
    ls = [| (l+a+b) <<< (a+b)
         || l <- initL # ls
         || a <- ss
         || b <- [0] # ls |];
    rs = ss @@ [(v-nk) .. (v-1)];
  };
```



- Models crypto-algorithm
- Natural expression
- Clear and unambiguous

# Test case generation



- Generates “known good tests”
- Built-in capture of intermediate vectors simplifies debugging
- Easy to generate new intermediate vectors as needed

## Equivalence checking

- Given two Cryptol functions *f*, *g*
  - Either prove they agree on all inputs:
    - $\forall x.\ f\ x == g\ x$
  - Or, provide a counter example *x* such that
    - $f\ x\ != g\ x$
- Typically:
  - *f*: Specification, written for clarity
  - *g*: Implementation, optimized for speed/space/FPGA, etc.

| galois |

## Simple equivalence checking example

```
f, g, h : [64] -> [64];
f x = 2*x;
g x = x << 1;
h x = x >> 1;
```

```
Cryptol> :eq f g
True
```

```
Cryptol> :eq f h
False
f 2
   = 4
h 2
   = 1
```

galois

## Other equivalence checking use-cases

- Actually, *f* and *g* don't have to be written in Cryptol. Our equivalence checker also supports:
  - C code
  - Java (via JVM byte code)
  - Anything else that can be translated into And-Inverter Graphs (our formal model notation)
- And in particular,
  - Xilinx FPGA netlists
  - This is the basis of our 3rd parth VHDL verification work

| galois |

## Cryptol Project Mission: *To reduce the cost (in both time and money) of developing and certifying cryptographic applications*

**Cryptol Specification**

**A Domain Specific Specification Language**
- Precise, Declarative Semantics
- High level design exploration

**Automated Synthesis down to FPGA and VHDL verification**
- Algebraic rewrite-based compilation
- Traceability back to specification
- Verifying generated and 3rd party VHDL

Certificate of Trust

This is to certify that

Cryptol Specification

can be trusted

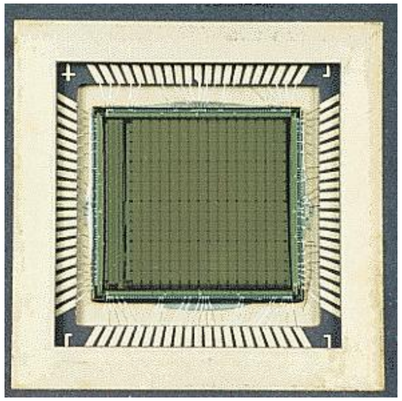
[illegible]

Certificate Authority

**Property specification and verification**
- SAT/SMT based property verification
- Push button assurance

## Why implement crypto algorithms on FPGAs?

- Lack of trust in commodity hardware
  - Evaluators can see as much of the solution as possible
  - Do not have to ship designs off-shore
  - FPGAs are more flexible than programmable custom crypto processors
- Natural match between Cryptography and FPGAs
  - Highly-parallel stream processing
- And FPGAs are *fast…*



## Why use Cryptol to produce FPGAs?

### Because Cryptol naturally models hardware

- No Von Neumann assumption
  - Sequentialization comes only from data-dependency
- Sequences can be mapped over space or time
  - Cryptol’s fixed-length sequences naturally model hardware bit-vectors
  - The user can explore space-time tradeoffs without significantly changing the Cryptol source

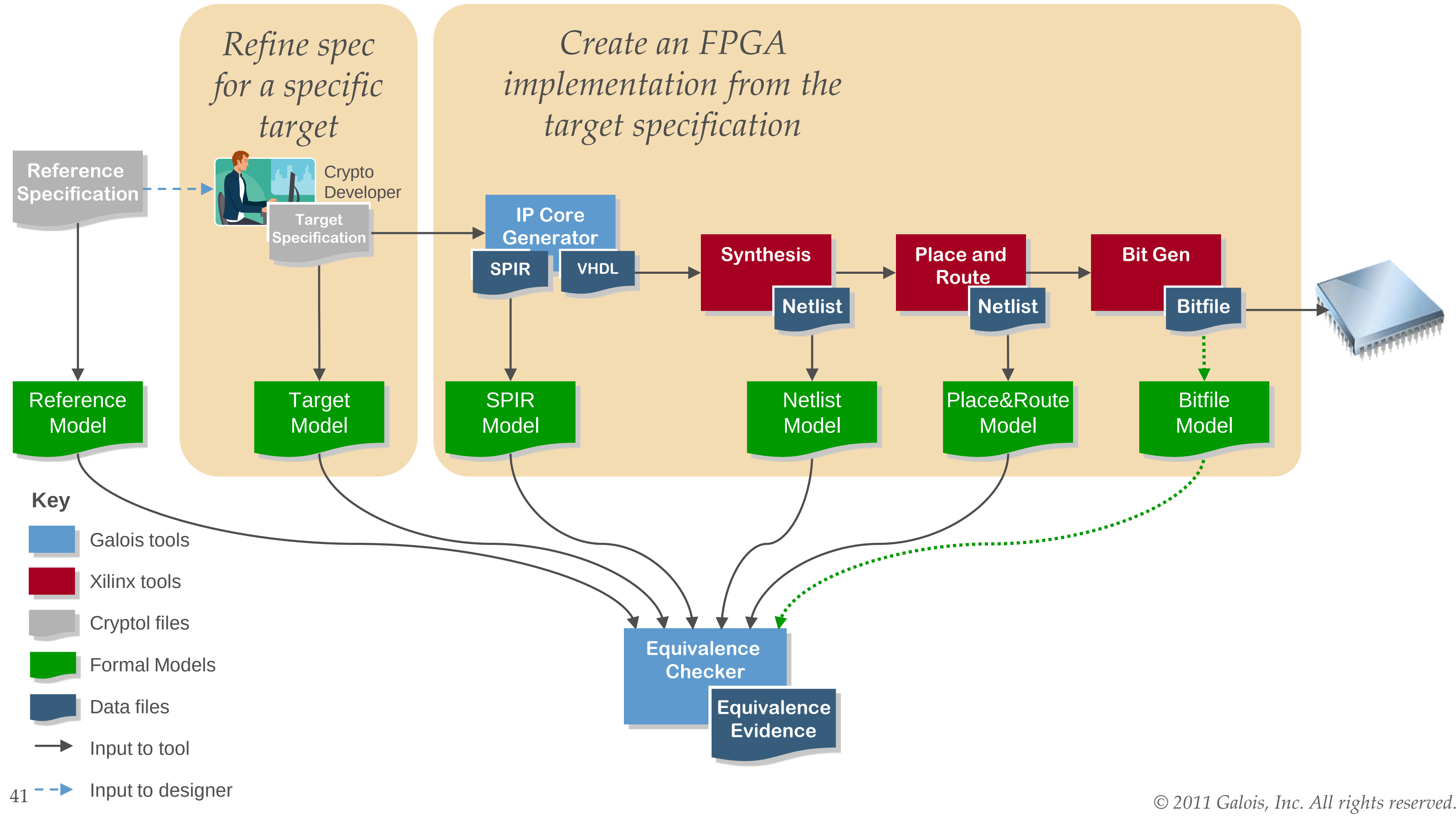
### Because Cryptol provides a low barrier-to-entry to FPGAs

- Standard libraries of Cryptol specifications
  - FPGA implementation is a small delta for the user
- Cross-compilation development
  - Develop specs on conventional hardware
  - Execute on FPGA

galois

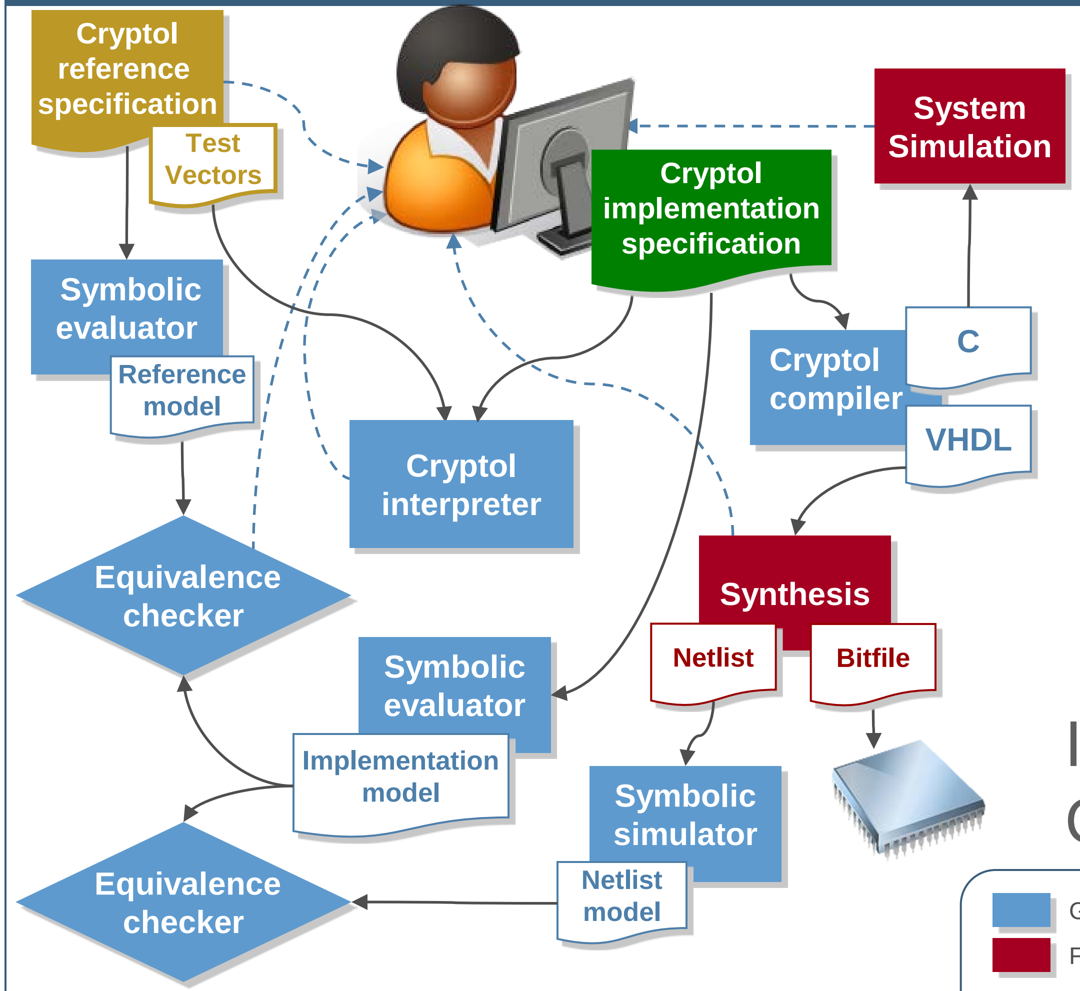
# Strategy

*Maintain functional equivalence with the reference specification throughout the tool chain*



| galois |

# Cryptol in the Development Process: Automatically-generated VHDL



A Cryptol-FPGA engineer:

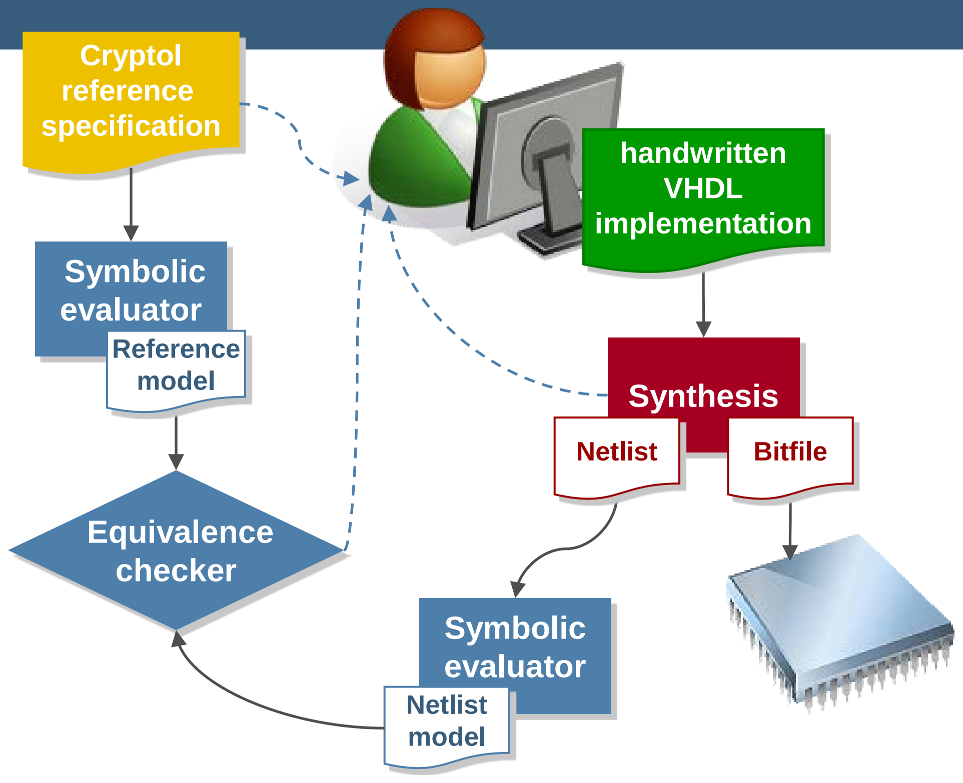
- Modifies the reference specification to emphasize particular implementation choices
- Generates the VHDL automatically
- Uses the equivalence checker to confirm that the implementation remains equivalent to the reference specification

It is easier to experiment in Cryptol than in VHDL

| | | | |
|---|---|---|---|
| 🟦 | Galois tools | 🟨 | Specification |
| 🟥 | FPGA Vendor tools | ▭ | Data files produced by Cryptol tools |
| → | Input to tool | ▭ | Data files produced by vendor tools |
| - - ▶ | Feedback to designer | 🟩 | Source files |

| galois |

# Cryptol in the Development Process: Hand-written VHDL

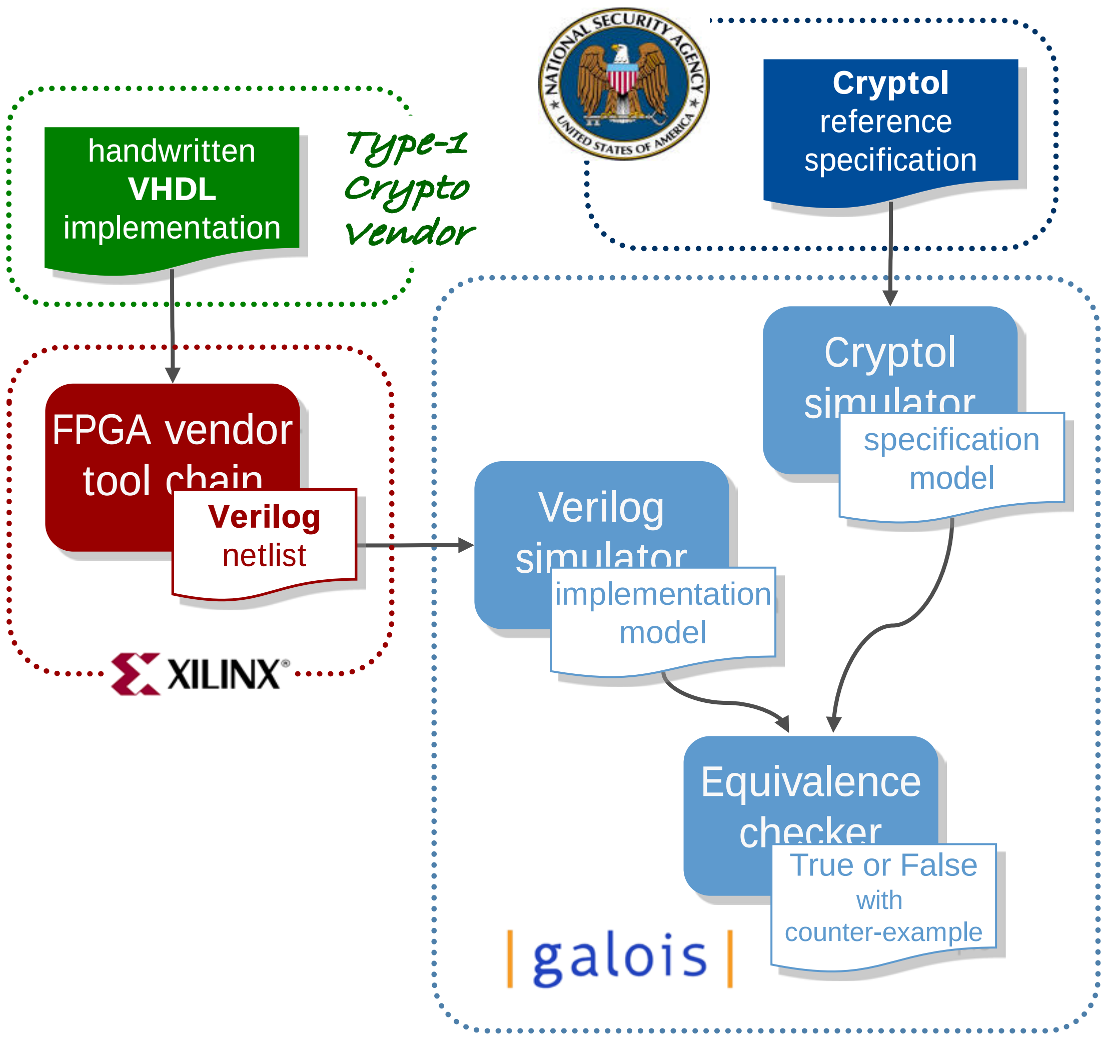


A VHDL-FPGA engineer:

- Studies the reference specification to gain understanding
- Crafts a VHDL implementation by hand
- Uses the equivalence checker to debug the implementation or confirm that it is equivalent to the reference specification

| | | | |
|---|---|---|---|
| 🟦 | Galois tools | 🟨 | Specification |
| 🟥 | FPGA Vendor tools | ▭ | Data files produced by Cryptol tools |
| → | Input to tool | ▭ | Data files produced by vendor tools |
| - - ▸ | Feedback to designer | 🟩 | Source files |

# Cryptol use-case: Verify VHDL crypto cores



## SHA3: NIST Hash competition

- "NIST has opened a public competition to develop a new cryptographic hash algorithm, which converts a variable length message into a short "message digest" that can be used for digital signatures, message authentication and other applications."
- 51 original submissions
- Recently, 5 candidates made it to the final round
- Galois has received requests to verify VHDL implementations for some
- We'll look at Skein verification in detail:

http://csrc.nist.gov/groups/ST/hash/sha-3/index.html

## Case Study: Verification process

- Develop a specification
- Understand the implementation
- Bring the implementation to “Cryptol” land
  - via Cryptol’s foreign-function-interface
- Coerce the type signature of the implementation and specification
- Use Cryptol verification tools to prove equivalence

| galois |

## Develop a specification

```
encrypt256 : ([32][8],[16][8],[32][8]) -> [4][64];
encrypt256 (key,tweak,pt) = vn + kn
  where {
   // Threefish-256 has 72 rounds:
   nr  = 72;
   nw  = 4;
   …
   key_words : [4][64];
   key_words = split(join key);
   tw_words : [2][64];
   tw_words = split(join tweak);
   pt_words : [nw][64];
   pt_words = split(join pt);
  };
```

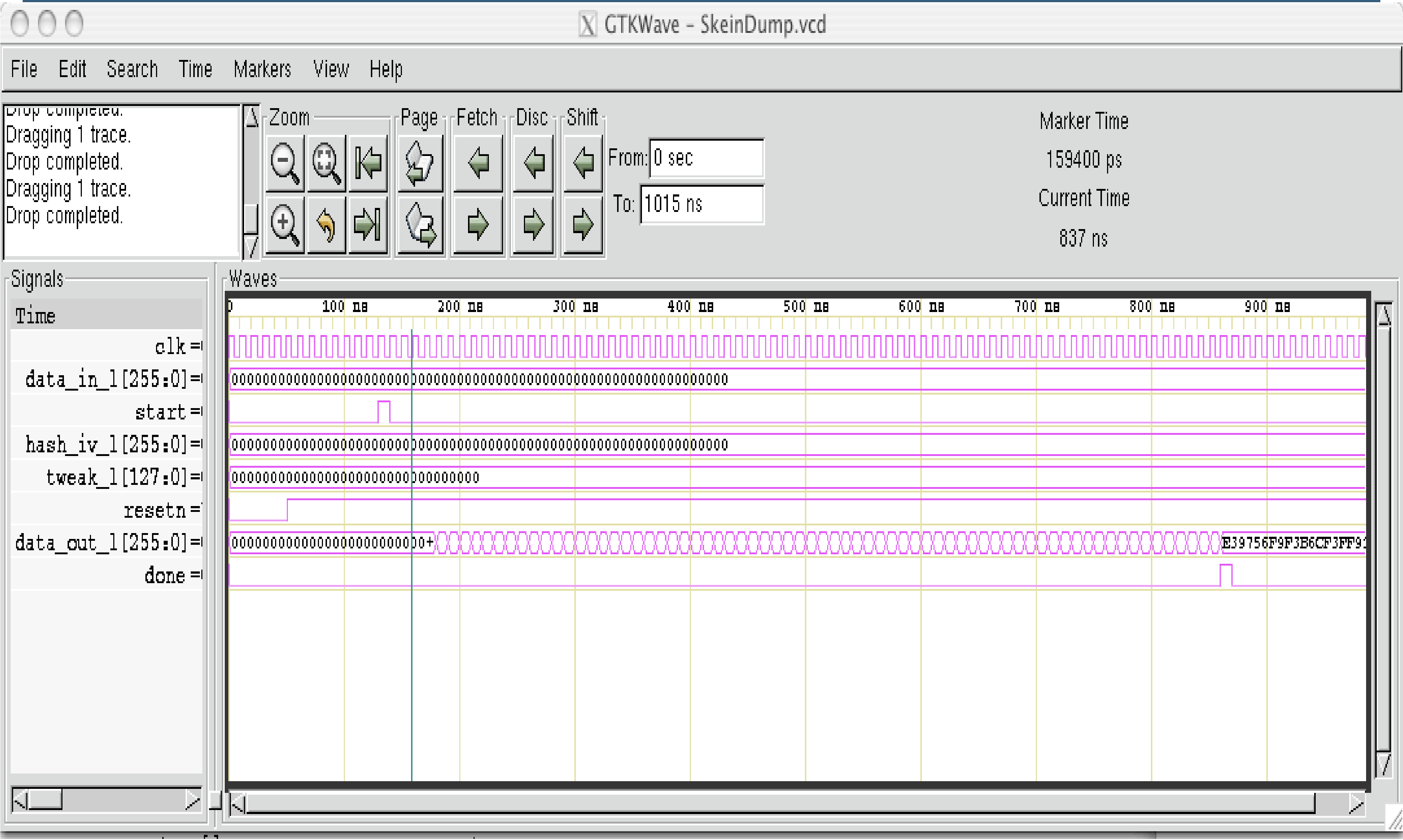
http://www.galois.com/blog/2009/01/23/a-cryptol-implementation-of-skein/

## Verification process

- Develop a specification
- Understand the implementation
- Coerce the type signature of the implementation and specification
- Use Cryptol verification tools to prove equivalence

| galois |

## Understanding the implementation



```
Drop completed.
Dragging 1 trace.
Drop completed.
Dragging 1 trace.
Drop completed.
```

From: 0 sec
To: 1015 ns

Marker Time
159400 ps
Current Time
837 ns

```
Time
clk =
data_in_l[255:0] =   0000000000000000000000000000000000000000000000000000000000000000
start =
hash_iv_l[255:0] =   0000000000000000000000000000000000000000000000000000000000000000
tweak_l[127:0] =     00000000000000000000000000000000
resetn =
data_out_l[255:0] =  000000000000000000000000000+   E39756F9F3B6CF3FF9
done =
```

## Import the VHDL to Cryptol

```
extern vhdl("datatype.vhd”, "skein_mixcolumn.vhd", "skein_round.vhd”,
            "skein_shiftrow.vhd“, "skein_add_round_key.vhd”, "skein.vhd“,
             skein, clock=clk, reset=resetn, invertreset)
extern_skein : [inf](start:Bit, data_in_L:[256], hash_iv_L:[256],
                       tweak_L:[128]) ->
                 [inf](done:Bit, data_out_L:[256]);

vhdlSkein : [256] -> [256];
vhdlSkein inp = res
  where { wait     = (False, inp, zero, zero);
          start    = (True, inp, zero, zero);
          rest     = [wait] # rest;
          (_, res) = extern_skein([wait start] # rest) @ 74;
        };
```

## Verification process

- Develop a specification
- Understand the implementation
- Coerce the type signature of the implementation and specification
- Use Cryptol verification tools to prove equivalence

## Coerce the type signatures

```
skeinRef : [256] -> [256];
skeinRef inp =
  alignOut(encrypt256(zero, zero, reverse (split inp))) ^ inp
  where { alignOut : [4][64] -> [256];
          alignOut xs = join (reverse (split (join xs) : [32][8]));
        };



vhdlSkein : [256] -> [256];
vhdlSkein inp = res
  where { wait     = (False, inp, zero, zero);
          start    = (True, inp, zero, zero);
          rest     = [wait] # rest;
          (_, res) = extern_menLong([wait start] # rest) @ 74;
        };
```

galois

## Verification Process

- Develop a specification
- Understand the implementation
- Coerce the type signature of the implementation and specification
- Use Cryptol verification tools to prove equivalence

## First verification

- Skein implementation by Men Long in VHDL
- Skein UBI Block AIG Sizes
  - Cryptol Reference, 118156 nodes
  - Men Long, 653963 nodes
- Ambiguity issue: Men Long’s concise cyclic rotation was interpreted differently by GHDL, Simili and Xilinx.  Resolved by replacement by call to standard library function.
- Used ABC (UC Berkeley) Equivalence Checker
- In ~1 hr VHDL code proved equivalent to spec

# Second verification

- Full Skein VHDL Implementation
- Skein AIG Sizes (256 bits input/output)
  - Cryptol Reference, 301342 nodes
  - Stefan Tillich, 900496 nodes
- Used ABC (UC Berkeley) Equivalence Checker
- Time: ~17.5h
- VHDL code is equivalent to Cryptol spec.

http://www.iaik.tugraz.at/content/research/

| galois |

## Cryptol Project Mission:

*To reduce the cost (in both time and money) of developing and certifying cryptographic applications*

**A Domain Specific Specification Language**
- Precise, Declarative Semantics
- High level design exploration

**Automated Synthesis down to FPGA and VHDL verification**
- Algebraic rewrite-based compilation
- Traceability back to specification
- Verifying generated and 3rd party VHDL

Certificate of Trust

This is to certify that

Cryptol Specification

can be trusted

[illegible]

Certificate Authority

**Property specification and verification**
- SAT/SMT based property verification
- Push button assurance

## Property verification

- Equivalence checking shows functional equivalence
  - The input/output behaviors are “precisely the same”
  - Or, they both have the exact same bugs..
- Property verification goes further
  - Allows “correctness” properties to be specified and proved automatically
- Traditional examples:
  - The algorithm works correctly
  - The function defined is associative and commutative
  - Value returned is the minimum..
- Classic crypto example:
  - For all values of key and plain-text, encryption followed by the decryption using the same key returns the plain-text
  - In Cryptol:

```
theorem encDec: {key, pt}. dec (key, enc(key, pt)) == pt;
```

# Example: Merge-sort

```
mergeSort : {a b} (fin a, fin b) => [a][b] -> [a][b];
mergeSort xs = take(width xs, unTag (taggedMrgSort (tag xs)));

taggedMrgSort xs = if v then combined else xs
    where {
     (_, v)     = xs@1;
     xsGrouped = groupBy (2, xs);
     strm1      = [| x || [x _] <- xsGrouped |];
     strm2      = [| y || [_ y] <- xsGrouped |];
     sorted1    = taggedMrgSort strm1;
     sorted2    = taggedMrgSort strm2;
     combined   = merge (sorted1, sorted2);
   };

merge (xs, ys) = if       ~vx   then ys
                 else if ~vy   then xs
                 else if x < y then xh # merge(xt, ys)
                 else yh # merge(xs, yt)

 where { (x, vx) = xs @ 0;
         (y, vy) = ys @ 0;
         xh = take(1, xs);
         xt = drop(1, xs);
         yh = take(1, ys);
         yt = drop(1, ys);
      };
```

## Test

```
mergeSortOK =
    (mergeSort []               == [])
  & (mergeSort [1 1]            == [1 1])
  & (mergeSort [1 0 3]          == [0 1 3])
  & (mergeSort [100 99 .. 0] == [0 .. 100])
  & (mergeSort [1 3 1 1 4 5] == [1 1 1 3 4 5]);
```

But we’d like to do better!

## Proof obligations

- To prove sorting correct, we need to show
  - Output is in non-decreasing order
  - Output is a permutation of the input
- Strategy:
  - Define these as “predicates” in Cryptol
  - Write a theorem to capture correctness
- Example: recognizing non-decreasing sequences:

```
nonDecreasing : {a b} (fin a, fin b) => [a][b] -> Bit;
nonDecreasing xs = pairComps == ~zero
  where pairComps = [| x <= x' || x  <- [0] # xs
                               || x' <- xs
                    |];
```

- Recognizing permutations is similar

| galois |

## Putting it together

- Express correctness by combining the two:

```
theorem mergeSortIsCorrect: {xs}.
     nonDecreasing(ys) & isPermutationOf(xs,
ys)
  where ys = mergeSort(xs);
```

- Theorem declarations are:
  - First class citizens of Cryptol; coexists with the code
    - No need to learn a separate “verification” language
  - Not comments, or documentation
    - Although they serve as great documentation
  - Can be quick-checked for fast feedback
  - Or, proved automatically using SAT/SMT based technologies
  - External tool usage is all transparent to the user!
- Counter-examples are priceless!

galois

## Other Cryptol assurance tools

- "Quickcheck" property-based testing
  - User gives a property, Cryptol automatically tests it on random inputs.
- Safety checking
  - Automatically checks that a Cryptol function will never raise an exception
  - Some possible exceptions: Divide-by-zero, Out-of-bounds array access, assertion failures
- Use of SMT-based property checkers
  - SAT: Checks for satisfiability of large Boolean formulas
  - SMT extends SAT with higher-level constraint solvers (linear arithmetic, arrays, functions, etc.)
- Semi-automatic theorem proving
  - Translator from Cryptol to Isabelle theorem prover
  - User can specify arbitrary Cryptol properties, but proof may need human guidance

## Verification tools use cases

- Formal Verification
  - Prove whether a low level implementation matches a high level mathematical specification
- Verifying Compiler
  - Confirm the correct functionality of generated code
- Regression verification
  - Use previous implementations to maintain the correctness of future implementations
  - cf. Regression testing

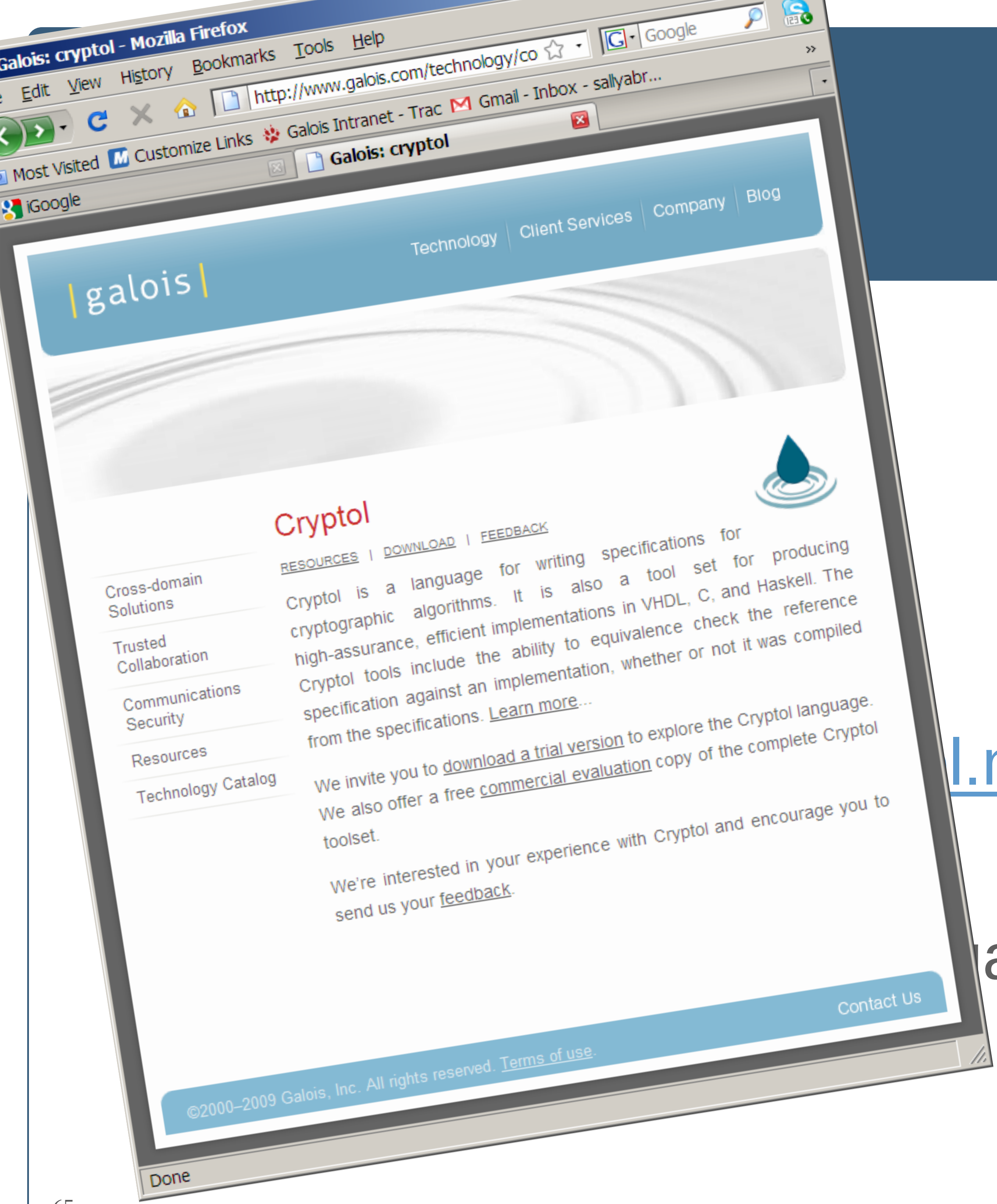
## Current Research Challenges

- Cryptol language
  - Extension into other data-flow domains (like DSP)
  - Other code generators (like ASIC, GPU, proprietary microcode)
- FPGA generation
  - Improved type constraint simplifications and error messages
  - If a Cryptol spec is not synthesizeable, communicating back to user why not
  - Synthesizing circuits across multiple FPGAs
  - Supporting vendor optimizations in Cryptol
- Equivalence checking
  - End-to-end Xilinx tool flow support
  - ECC and PKI algorithms, initial attempts quite promising
  - Large bit vector sizes, deep semantic optimizations
  - Equivalence checking for modes

# Questions?

l.net

alois.com


Galois: cryptol - Mozilla Firefox
Edit View History Bookmarks Tools Help
http://www.galois.com/technology/co
Google
Most Visited Customize Links Galois Intranet - Trac Gmail - Inbox - sallyabr...
iGoogle
Galois: cryptol
Technology Client Services Company Blog
galois
Cross-domain Solutions
Trusted Collaboration
Communications Security
Resources
Technology Catalog
Contact Us
Done

Cryptol

RESOURCES | DOWNLOAD | FEEDBACK

Cryptol is a language for writing specifications for cryptographic algorithms. It is also a tool set for producing high-assurance, efficient implementations in VHDL, C, and Haskell. The Cryptol tools include the ability to equivalence check the reference specification against an implementation, whether or not it was compiled from the specifications. Learn more...

We invite you to download a trial version to explore the Cryptol language. We also offer a free commercial evaluation copy of the complete Cryptol toolset.

We're interested in your experience with Cryptol and encourage you to send us your feedback.

©2000–2009 Galois, Inc. All rights reserved. Terms of use.