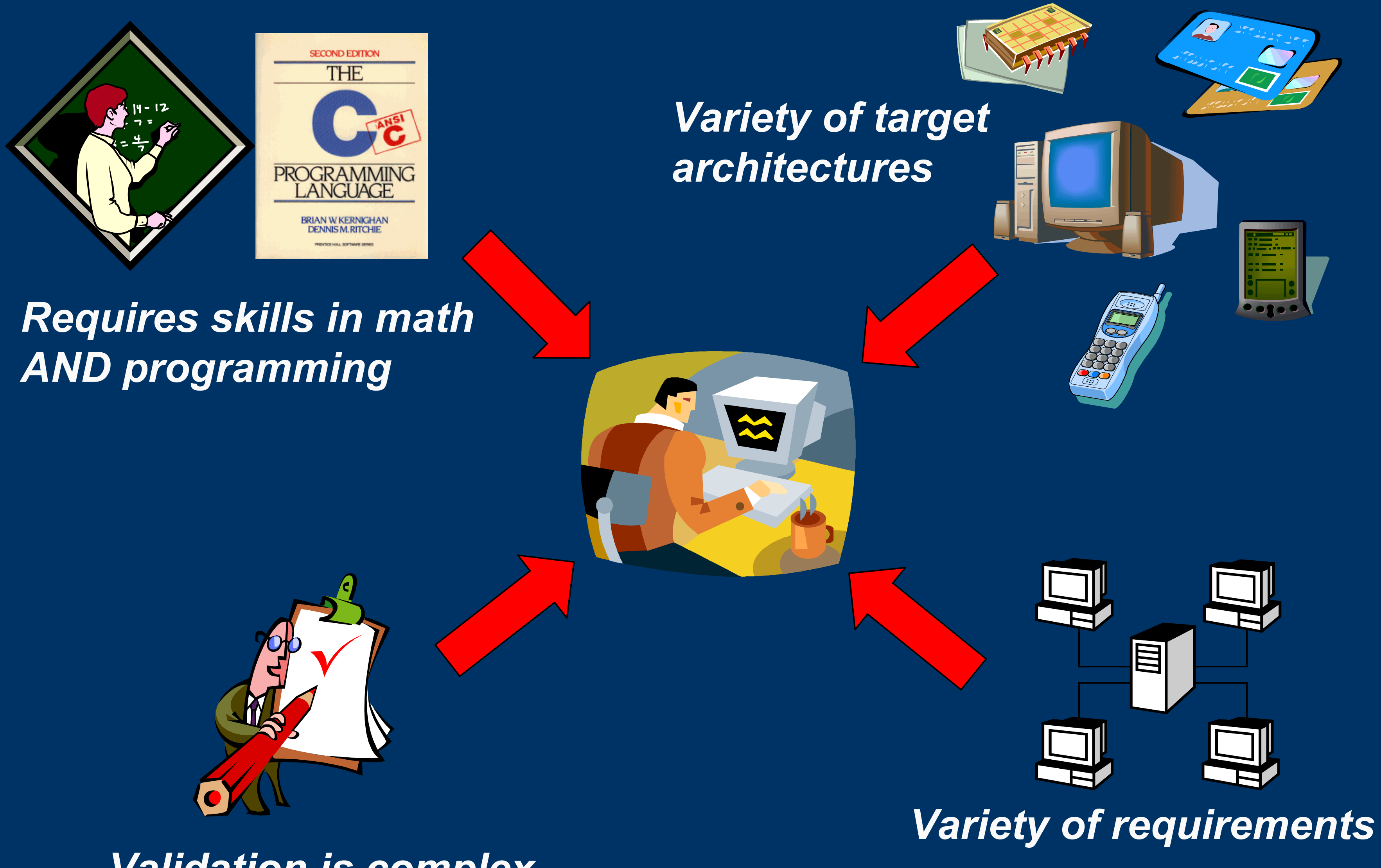
Cryptol Tutorial

| galois |

Challenge:
To Support the Correctness of Implementations of Crypto-algorithms

- Crypto-alg V&V critical in crypto-modernization programs
- Must manage assurance in face of exploding complexity and demands
- Not just the NSA / DoD
  - 25% of algorithms submitted for FIPS validation had security flaws
    - Director NIST CMVP, March 26, 2002

Contributing Factors



Variety of target architectures

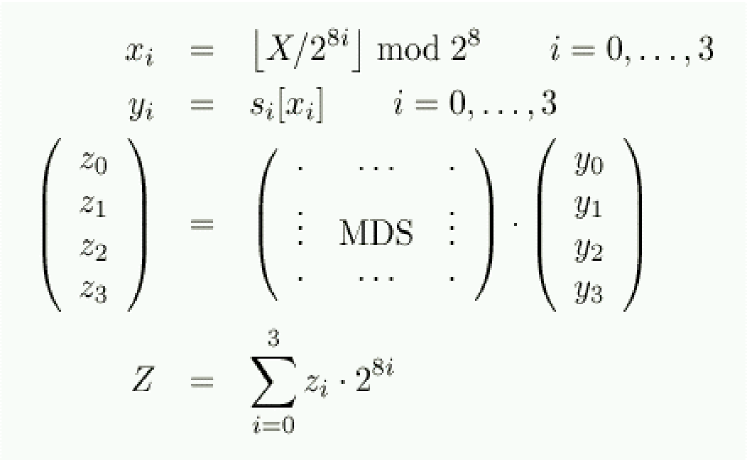
Requires skills in math AND programming

Variety of requirements

Validation is complex and tedious

Lack of clear reference implementations

```
#define MDS_GF_FDBK     0x169
#define LFSR1(x)  ( ((x) >> 1)   ^ (((x) & 0x01) ?
MDS_GF_FDBK/2 : 0))
            SR2(x)  ( ((x) >> 2)   ^ (((x) & 0x02) ?
                              MDS_GF_FDBK/2 : 0)
                              ^ (((x) & 0x01) ?
                              MDS_GF_FDBK/4 : 0))
             _1(x)  ((DWORD)   (x))
             _X(x)  ((DWORD)  ((x) ^ LFSR2(x)))
             _Y(x)  ((DWORD)  ((x) ^ LFSR1(x) ^
            )
            0                 Mul_1
#define M01                   Mul_Y
return   ((M00(b[0]) ^ M01(b[1]) ^
          M02(b[2]) ^ M03(b[3]))          ) ^
         ((M10(b[0]) ^ M11(b[1]) ^
          M12(b[2])^ M13(b[3])) << 8)^
          M22(b[2]) ^ M23(b[3])) <<16)
         ((M30(b[0]) ^ M31(b[1]) ^
          M32(b[2])^ M33(b[3])) <<24);
```



$$x_i = \lfloor X/2^{8i} \rfloor \bmod 2^8 \qquad i = 0,\ldots,3$$

$$y_i = s_i[x_i] \qquad i = 0,\ldots,3$$

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} \cdot & \cdots & \cdot \\ \vdots & \text{MDS} & \vdots \\ \cdot & \cdots & \cdot \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

$$Z = \sum_{i=0}^{3} z_i \cdot 2^{8i}$$

It’s hard to relate implementations to the underlying math

Approach:
Specifications and Formal Tools

Cryptol

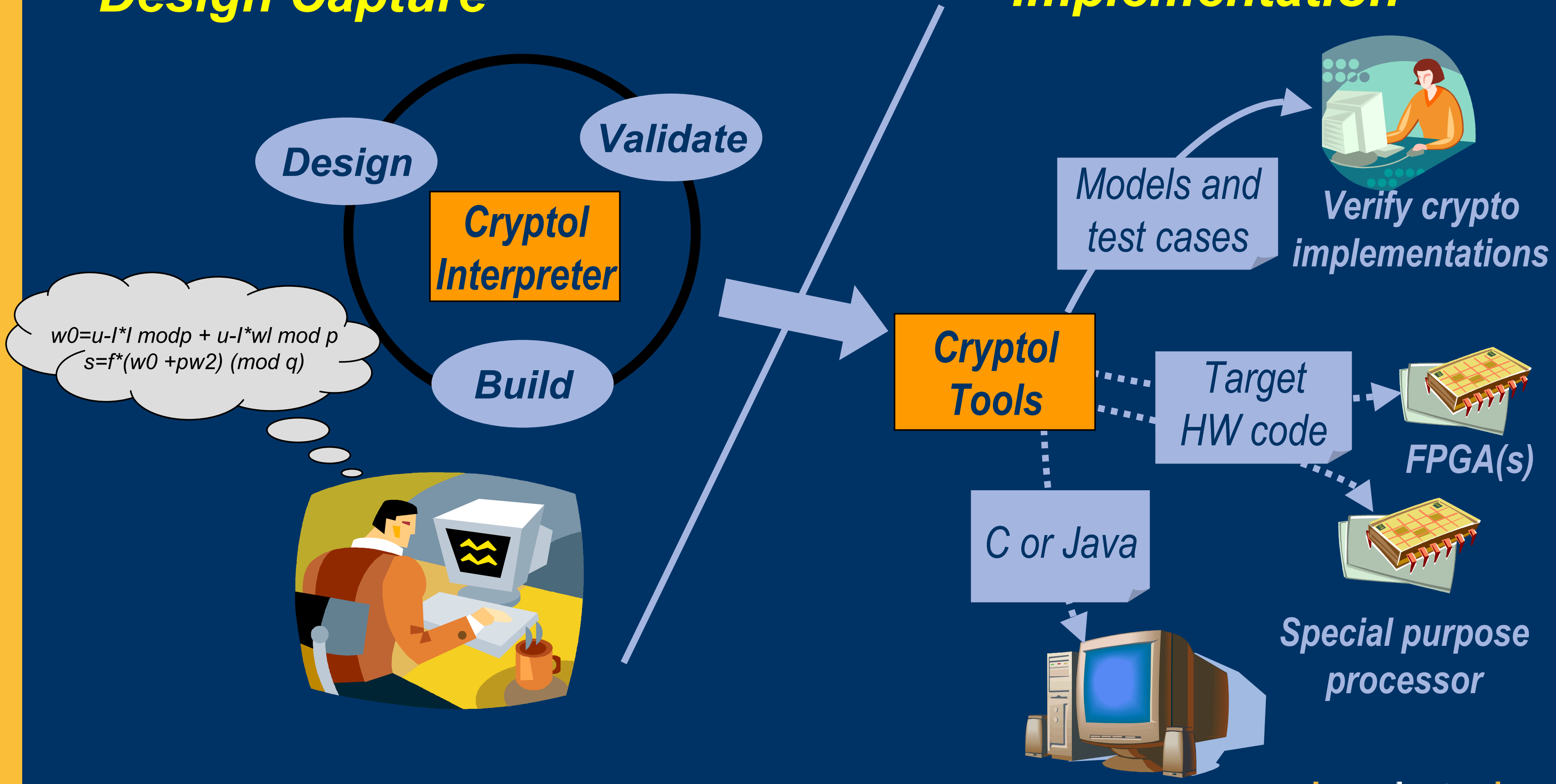
The Language of Cryptography

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

One Specification - Many Uses

Domain-Specific Design Capture

Assured Implementation



*w0=u-l*l modp + u-l*wl mod p*
s=f(w0 +pw2) (mod q)*

Design

Validate

Cryptol Interpreter

Build

Cryptol Tools

Models and test cases

Verify crypto implementations

Target HW code

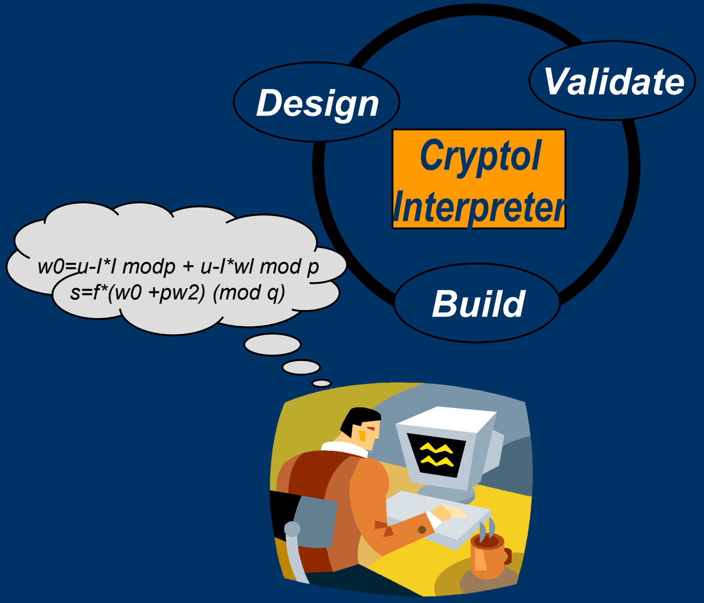
FPGA(s)

C or Java

Special purpose processor

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
- Structure and guide an implementation

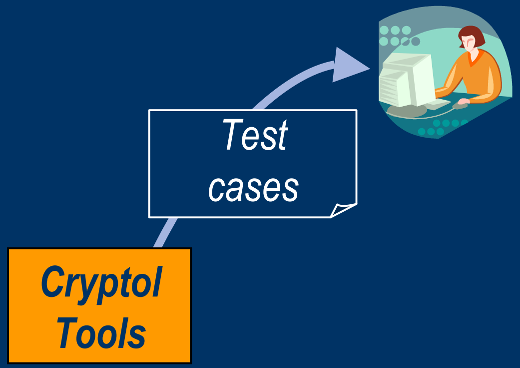
galois

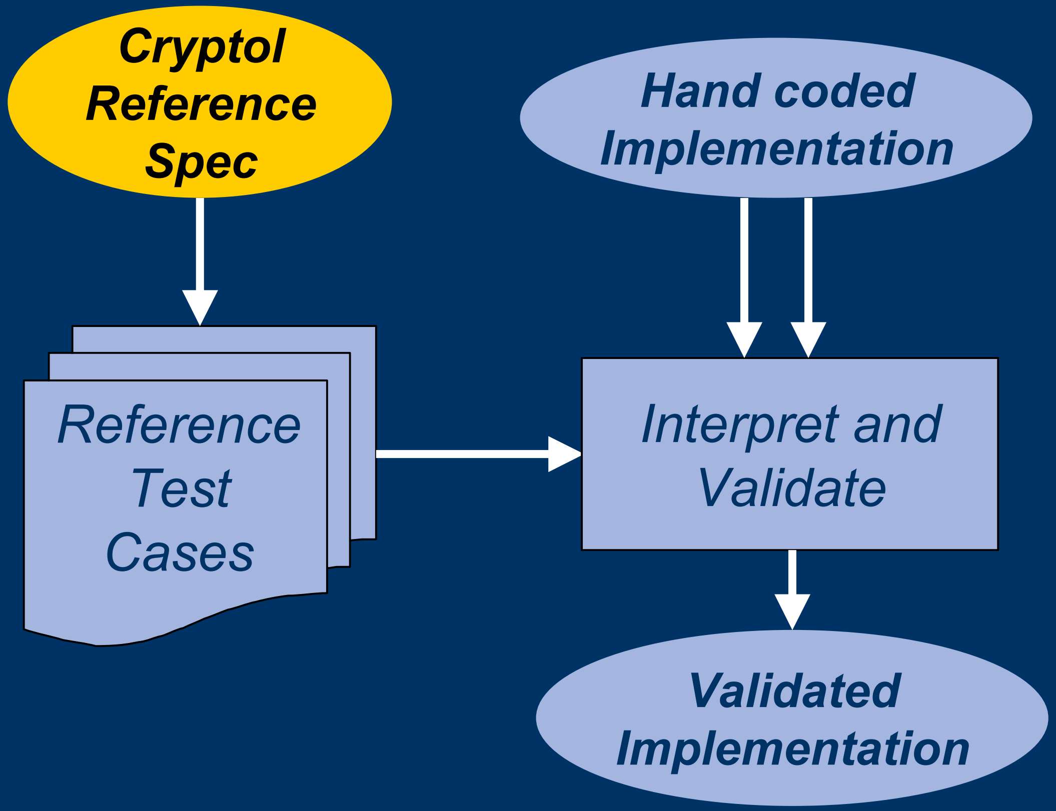
Key Ideas in Cryptol

- **Domain-specific data and control abstractions**
  - Sequences
  - Recurrence relations (not for-loops)
- **Powerful data transformations**
  - Data may be viewed in many ways
  - Machine independent
- **Flexible sizes**
  - Algorithms parameterized on size
    - Size constraints are explicit in many specs
    - Number of iterations may depend on size
  - A *Size-Type* system captures and maintains size constraints

Choosing what to leave out is critical

Usage: Testing





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

Cryptol Programs

- **File of mathematical definitions**
  - Two kinds of definitions: values and functions
  - Definitions may be accompanied by a type declarations (a *signature*)
- **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,x') = 2 * x + x';
```

Data types

- Homogeneous sequences

  ```
  [False True False True False False True]
  [[1 2 3 4] [5 6 7 8]]
  ```

- Numbers are represented as sequences of bits
  - Aka “words”
  - Decimal, octal (`0o`), hex (`0x`), binary (`0b`)

  ```
  123, 0xF4, 0b11110100
  ```

- Quoted strings are just syntactic sugar for sequences of 8-bit words

  ```
  “abc” = [0x61 0x62 0x63]
  ```

- Heterogenous data can be grouped together into *tuples*

  ```
  (13, “hello”, True)
  ```

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*
  - Like C's *a?b:c*

Sequences

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

```
[1..5] # [3 6 8]  =  [1 2 3 4 5 3 6 8]
[50 .. 99] @ 10  =  60
```

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

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

Cryptol Types

- **Types express size and shape of data**

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

- **Strong typing**
  - The types provide mathematical guarantees on interfaces
- **Type inference**
  - Use type declarations for active documentation
  - All other types computed
- **Parametric polymorphism**
  - Express size parameterization of algorithms

AES Types

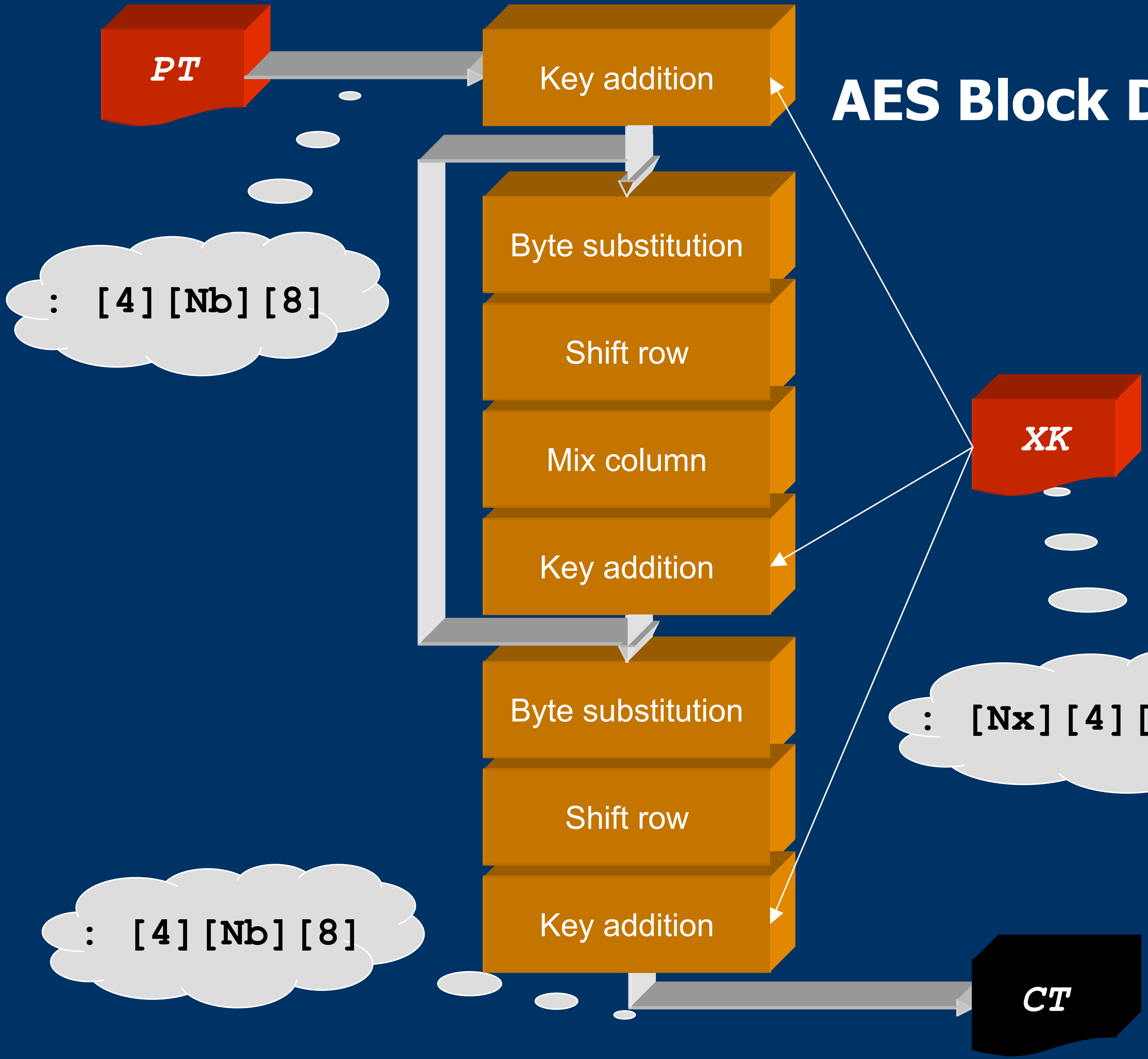
- “The State can be pictured as a rectangular array of bytes. This array has four rows, the number of columns is denoted by Nb and is equal to the block length divided by 32.”

```
state :  [4][Nb][8];
```

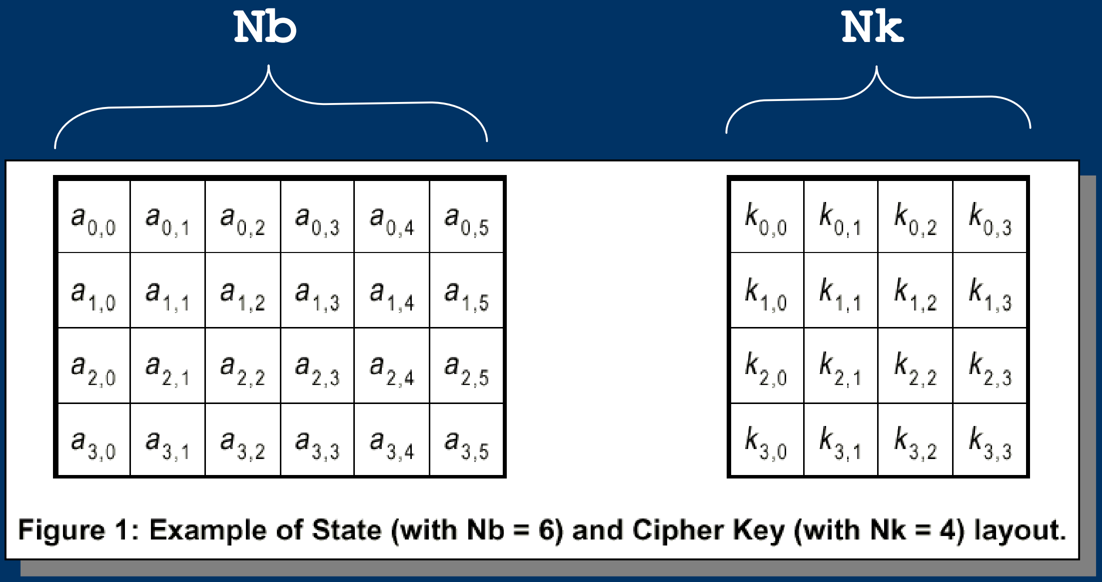
- “The input and output used by Rijndael at its external interface are considered to be one-dimensional arrays of 8-bit bytes numbered upwards from 0 to the 4*Nb-1. The Cipher Key is considered to be a one-dimensional array of 8-bit bytes numbered upwards from 0 to the 4*Nk-1.”

```
input :  [4 * Nb][8];
key   :  [4 * Nk][8];
```

AES Block Diagram



AES API



$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

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]

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

Splitting and Joining sequences

`0x99FAC6F975BABB3E`

↓ `split`

`[0x99 0xFA 0xC6 0xF9 0x75 0xBA 0xBB 0x3E]`

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

↓ `join`

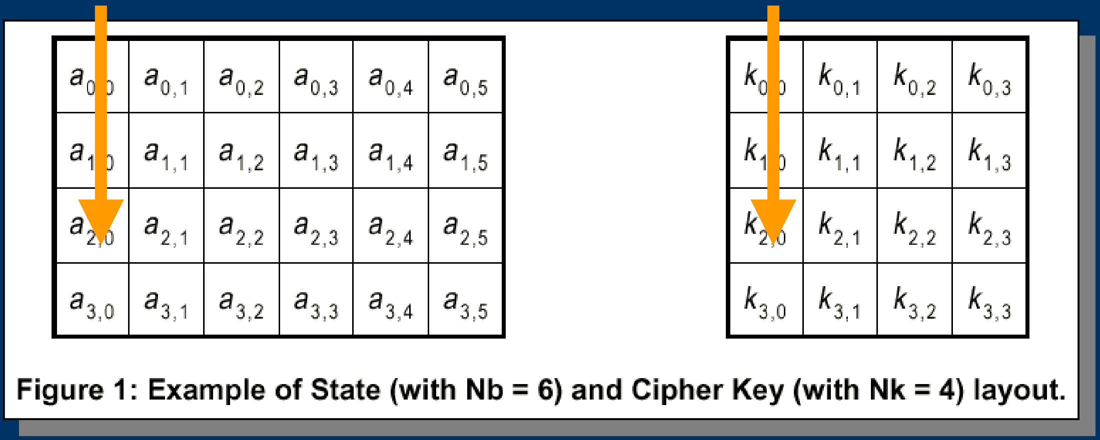
`0x99FAC6F975BABB3E`

Striping

- 2D sequences considered to be row major

```
stripe : [4*Nb][8] -> [4][Nb][8];
stripe(block) = transpose(split(block));

unstripe : [4][Nb][8] -> [4*Nb][8];
unstripe(state) = join(transpose(state));
```



$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

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

AES encryption

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

encrypt(XK,PT) = unstripe(Rounds(State,XK))
  where {
    State : [4][Nb][8];
    State = stripe(PT);
  };
```

Sequence Comprehensions

- The comprehension notion borrowed from set theory
  - $\{ a+b \mid a \in A, b \in B\}$
  - Adapted to sequences)

- Applying an operation to each element

```
   [| 2*x + 3   ||   x <- [1 2 3 4] |]
= [ 5 7 9 11 ]
```

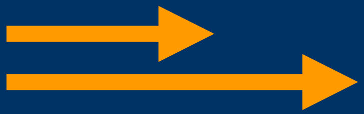
Traversals

- 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

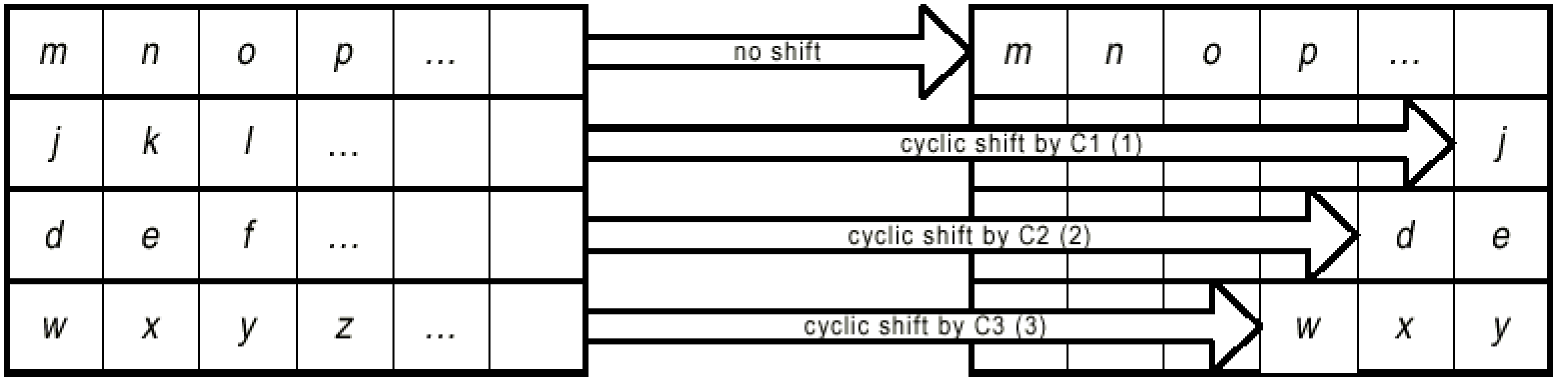


Figure 3: ShiftRow operates on the rows of the State.

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

Column traversals

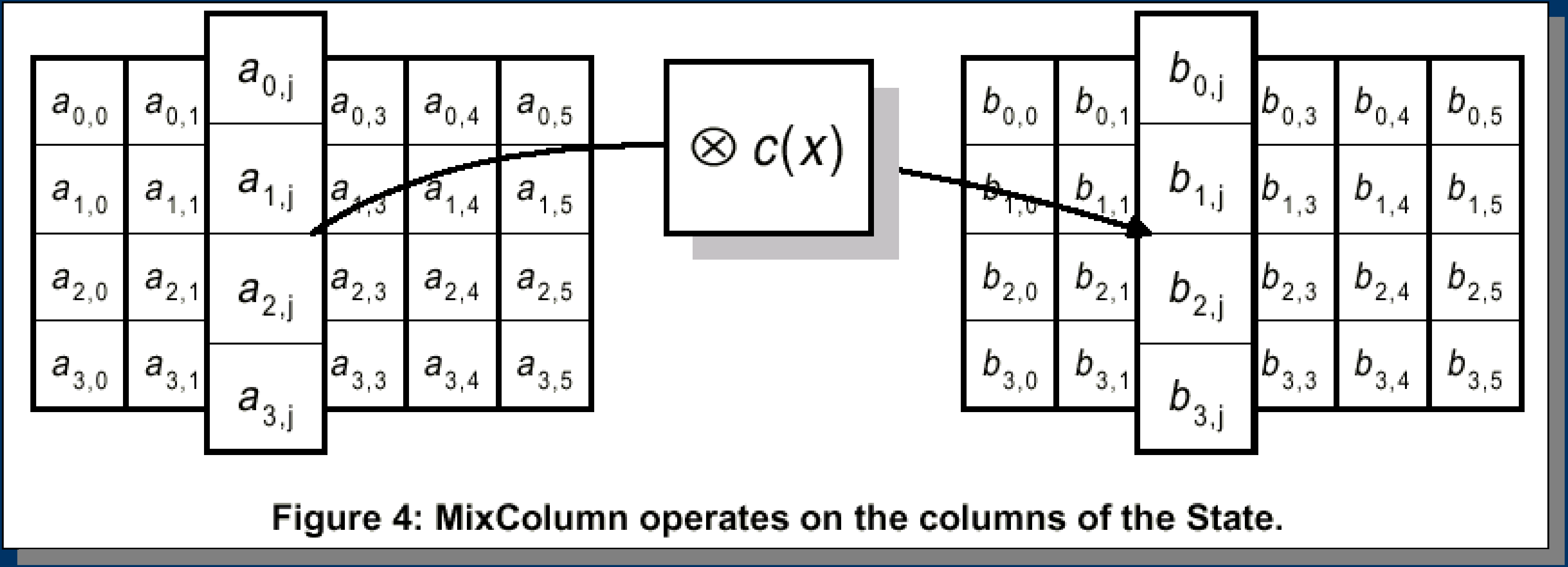


Figure 4: MixColumn operates on the columns of the State.

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

Nested traversals

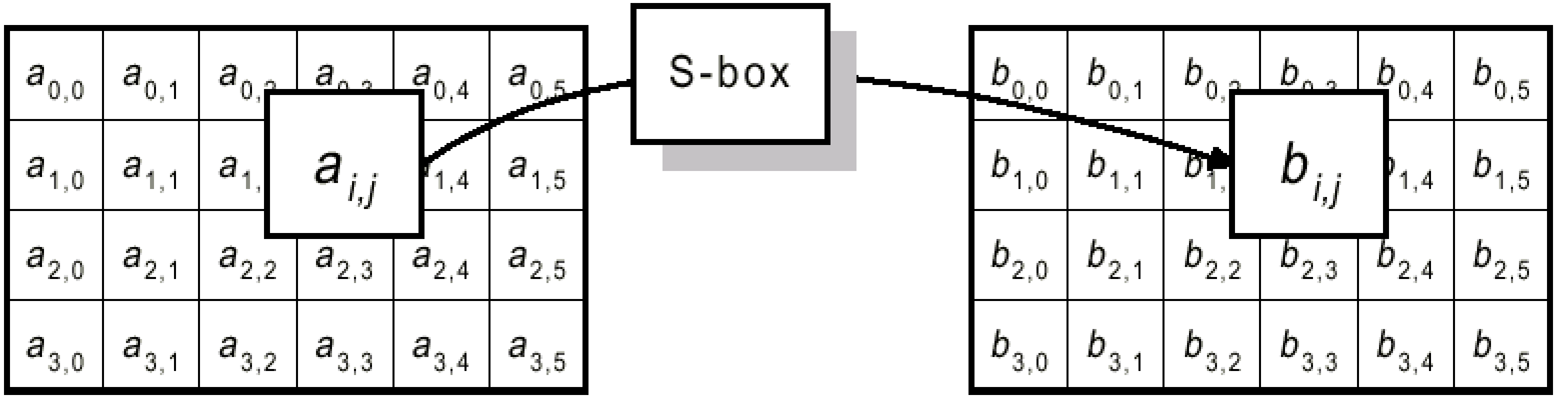


Figure 2: ByteSub acts on the individual bytes of the State.

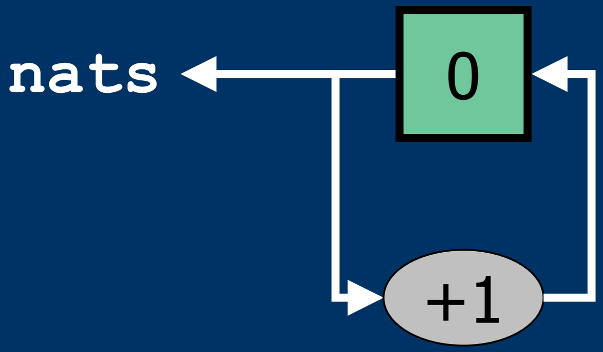
```
ByteSub : [4][Nb][8] -> [4][Nb][8];
ByteSub(state) = [| [| sbox @ a || a <- row |]
                  || row <- state |]

sbox : [256][8];
sbox = [| affine(inverse x)
        || x <- [0..255] |];
```

Recurrence

- **Textual description of shift circuits**
  - Follow mathematics: use stream-*equations*
  - Stream-definitions can be *recursive*

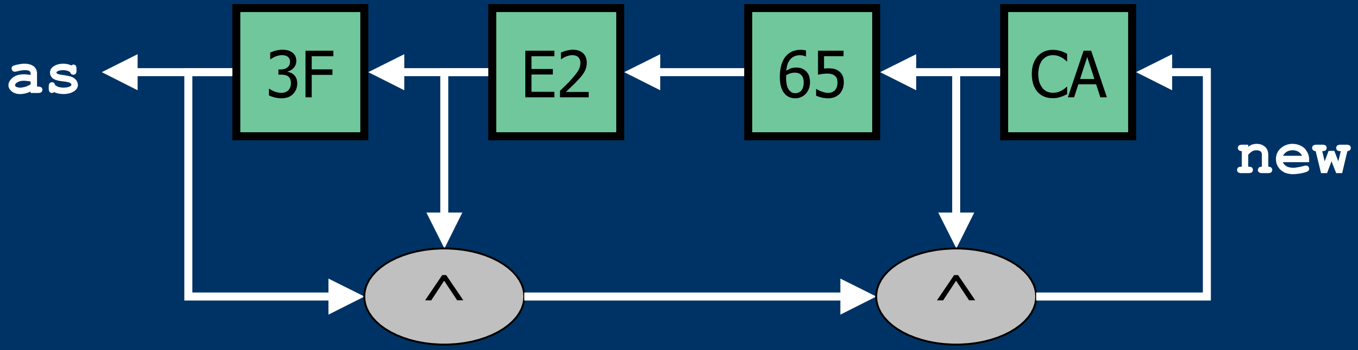
```
nats = [0] # [| y+1 || y <- nats |];
```



galois

More Complex Stream Equations

```
as  = [0x3F 0xE2 0x65 0xCA] # 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
```

AES Key Expansion

```
keyExpansion : [4*Nk][8] -> [(Nr+1)*Nb][4][8];
keyExpansion key = W
  where {
    keyCols : [Nk][4][8];
    keyCols = split key;
    W = keyCols # [| nextWord (i, old, prev)
                        || i <- [Nk..((Nr+1)*Nb-1)]
                        || old <- W
                        || prev <- drop (Nk-1, W)
                  |]; };
```

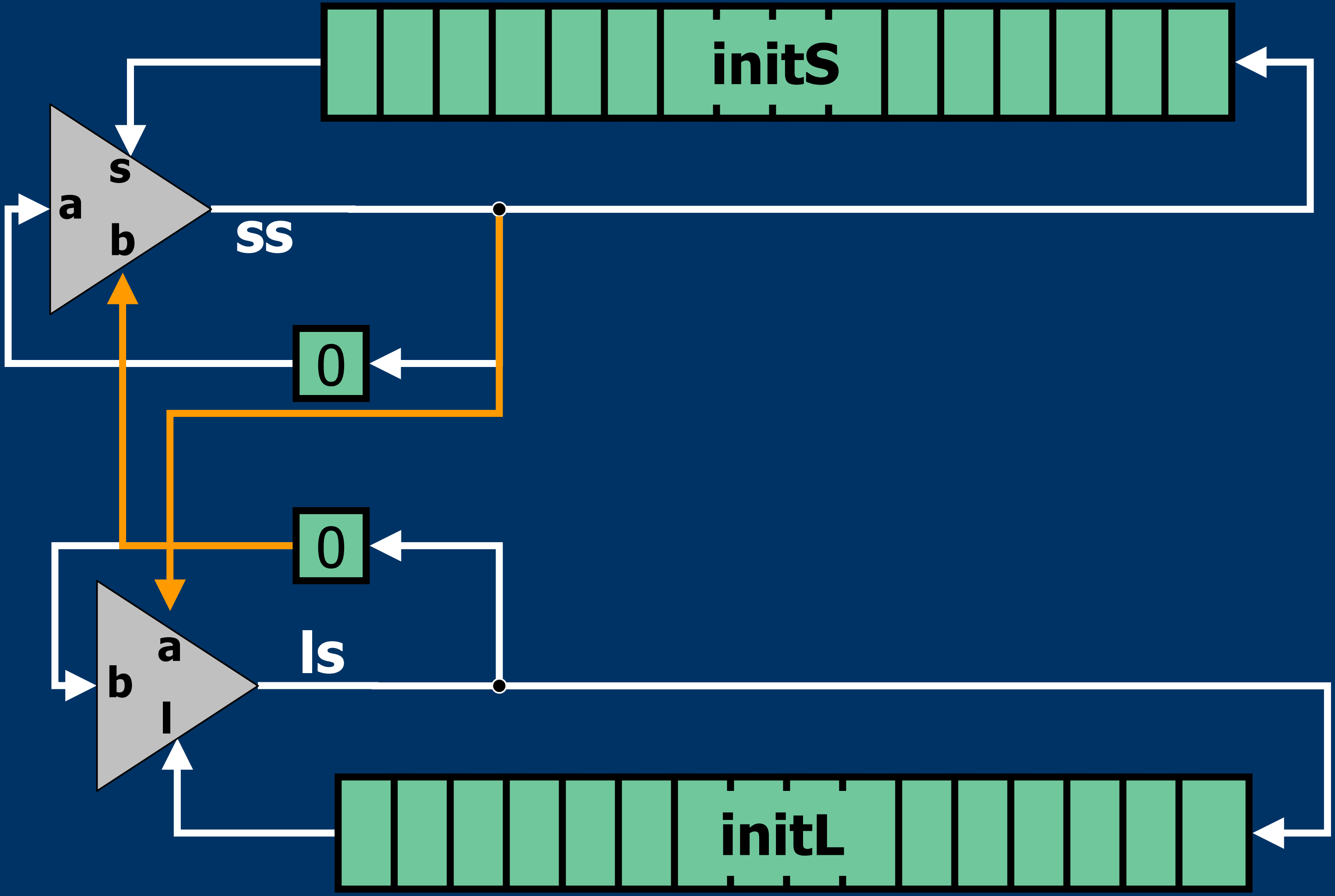
RC6 Key Expansion

- Original specification is written in terms of arrays and updates
  - Key expansion code appears entirely symmetrical
  - Cryptol exposes non-symmetry

```
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 |];
```

"Circuit" Diagram

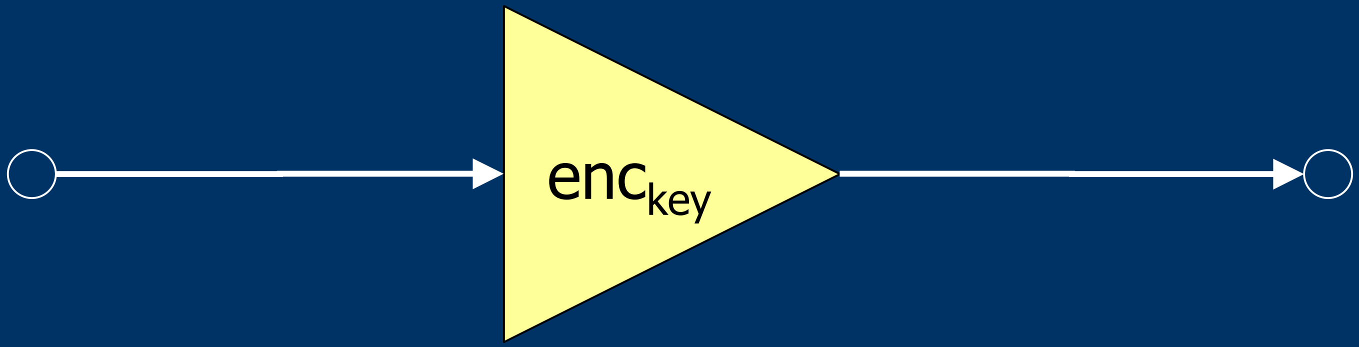


galois

Modes: Electronic code book

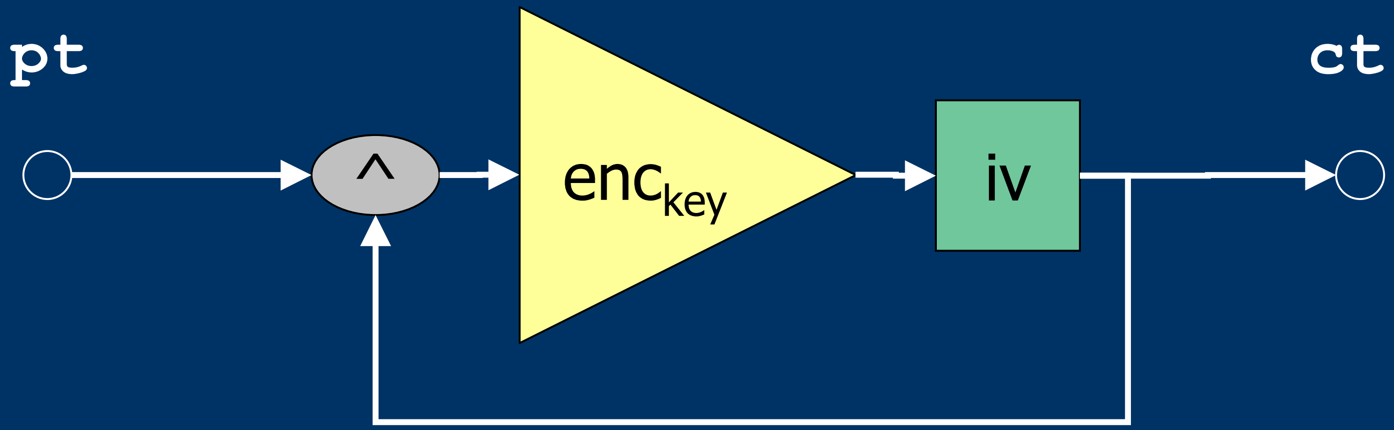
- Modes are expressed in the same way as other cycles

ct = [| encrypt (x, key) || x <- pt |]



Modes: Cipher Block Chaining

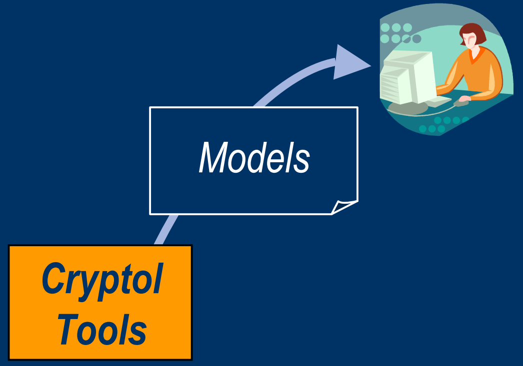
```
ct = [iv] # [| encrypt (x^y, key)
                   || x <- pt
                   || y <- ct |]
```

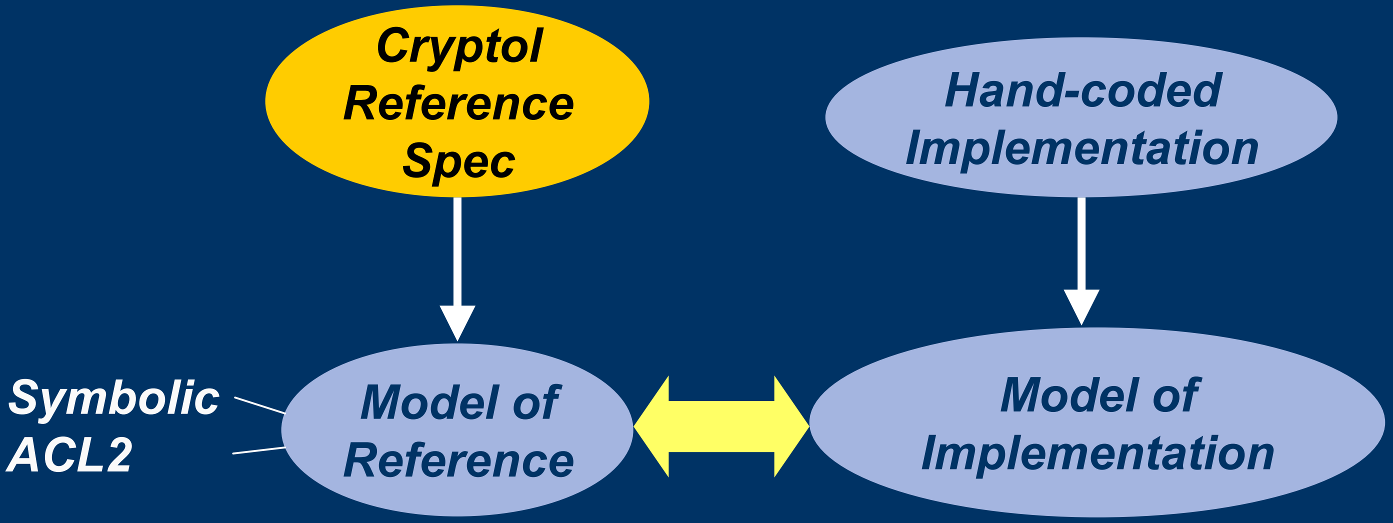


Ideal for Reference Implementations

- **Domain Specific**
  - Naturally understandable to developers
  - Simplifies expression, inspection, reuse
- **Executable**
  - Run tests and debug for correctness
  - Generate test cases
- **Declarative**
  - Not implementation-specific, concise
  - Multiple uses – test, generation, model building, etc.
  - Highly retargetable to any architecture
- **Unambiguous**
  - Formal basis
  - Precise syntax and semantics
  - Independent of underlying machine models

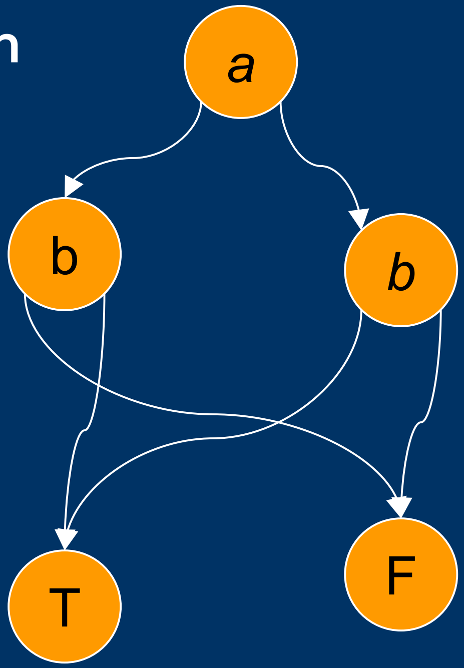
Usage: Verification





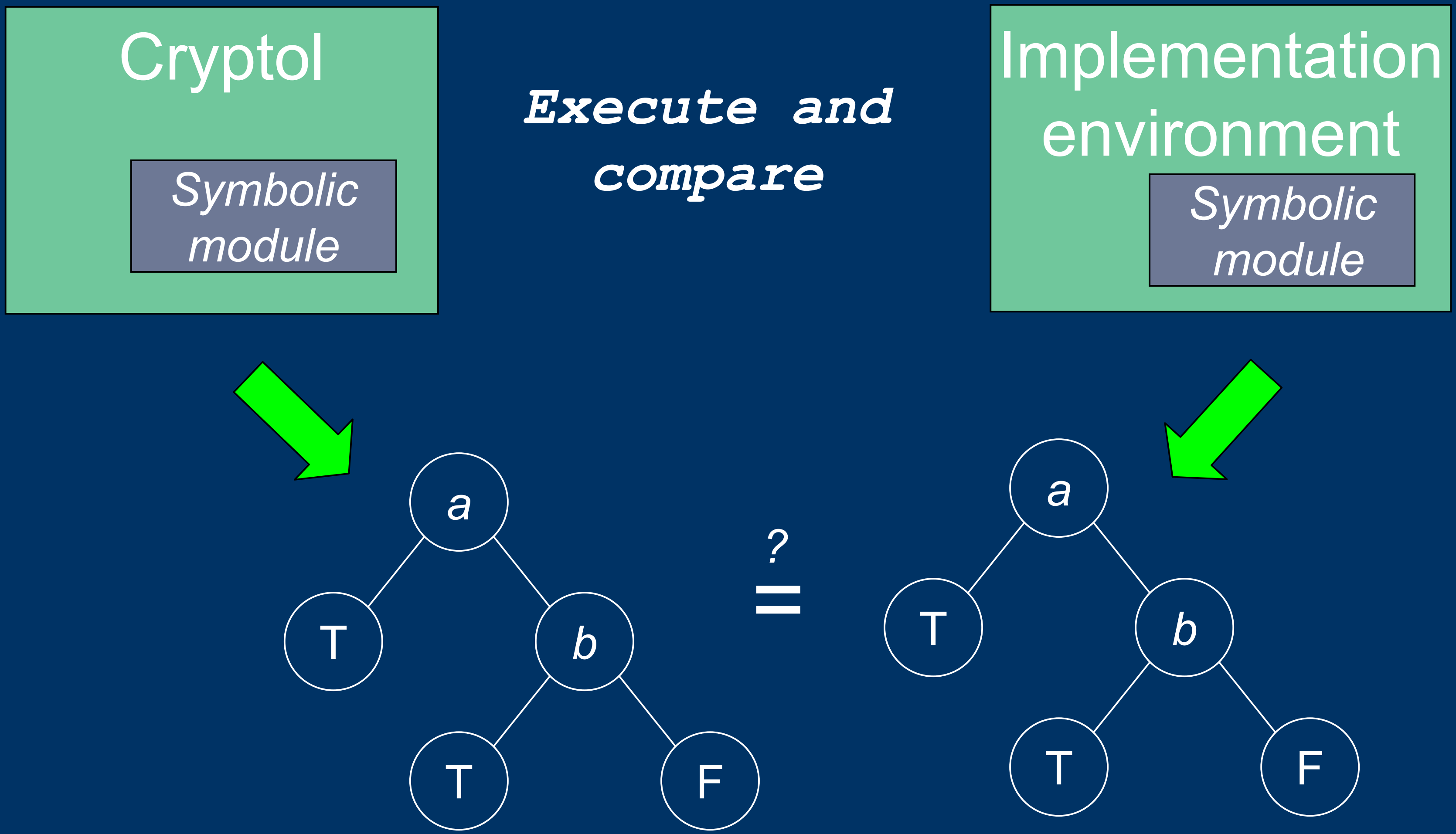
- Model checking and theorem proving now in development
- Enables formal verification between reference and implementation
- Much higher assurance of correctness

Model Checking



- **Logic formula describing input-output function**
  - "Symbolic bits"
  - Explore all possible cases concurrently
- **Binary Decision Diagrams (BDDs)**
  - Maximal sharing between cases
  - Equivalence of two BDDs is constant time
    - But, without proper care, BDDs grow exponentially
- **SAT solving**
  - Discover potential equivalences within the formulae
    - Lemmas
  - Powerful techniques for demonstrating the equivalences

Verification Architecture



Cryptol

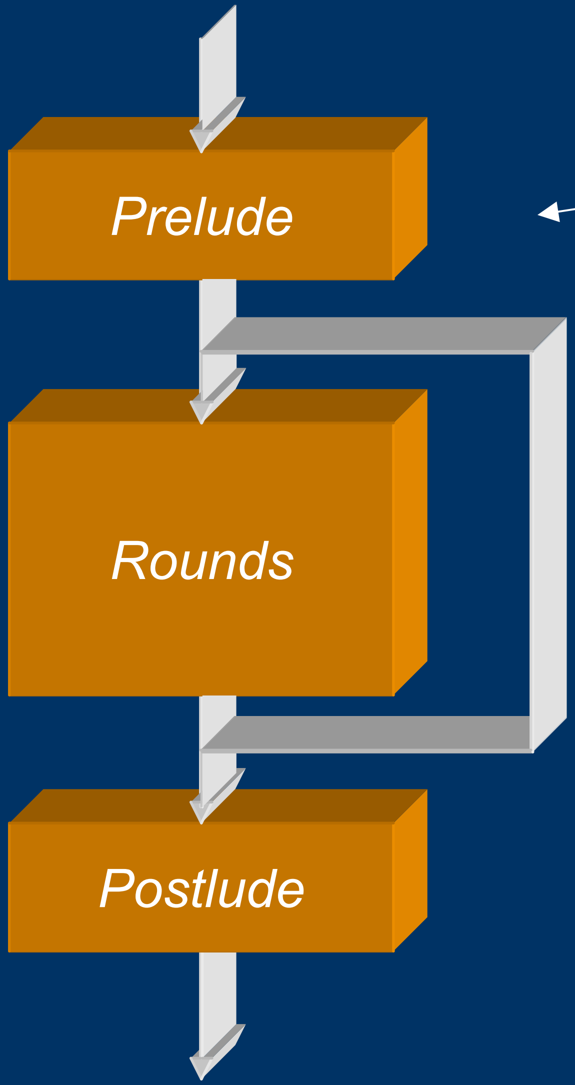
Symbolic module

Execute and compare

Implementation environment

Symbolic module

Strategies for Crypto Verification



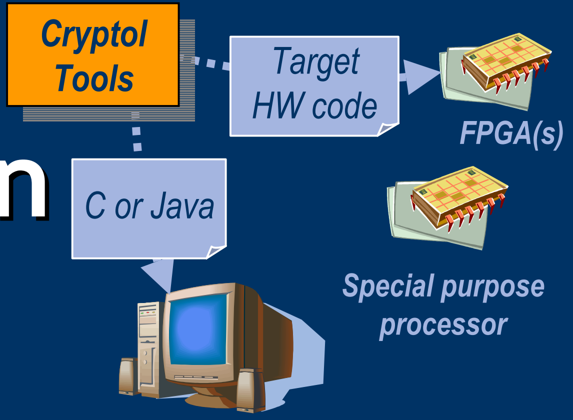
Small, relatively easy to apply brute force methods

1. Brute force: Verify a variant with a reduced number of rounds
2. Some user intervention: Isolate body of loop, verify the body, not the loop

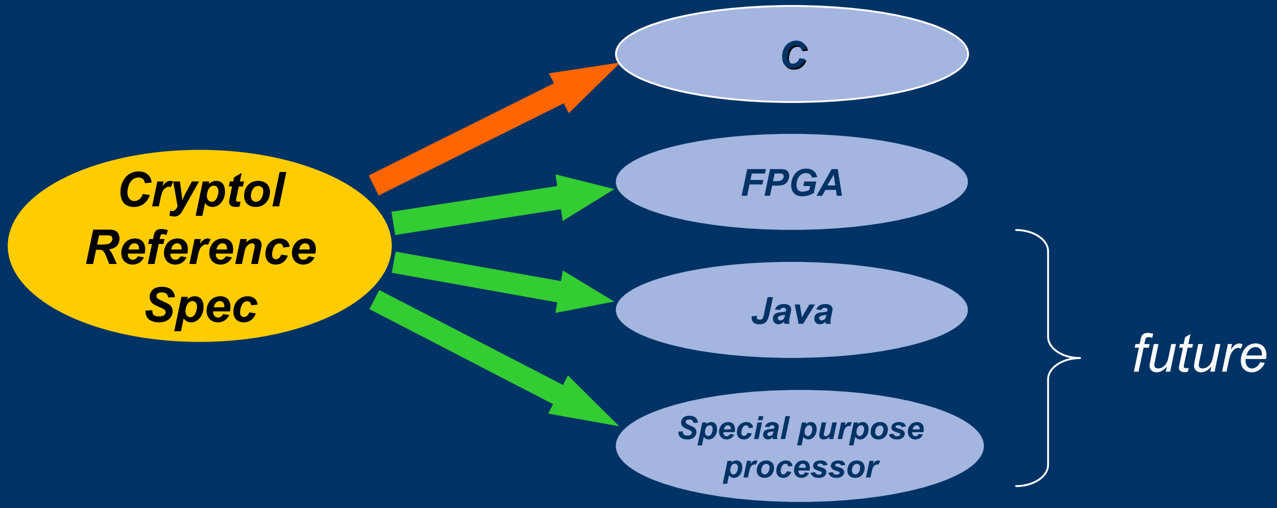
Why does this work?

- **Not trying to solve a general problem**
  - Relatively small # of iterations overall
  - Number of rounds largely independent of overall correctness
  - Simple structure of most crypto-algorithms
  - Relatively small memory footprint
- **The Role of Cryptol**
  - Scoping down to the crypto domain
    - Enables effective use of powerful verification tools
  - Authoritative model in this domain

Usage: Code Generation

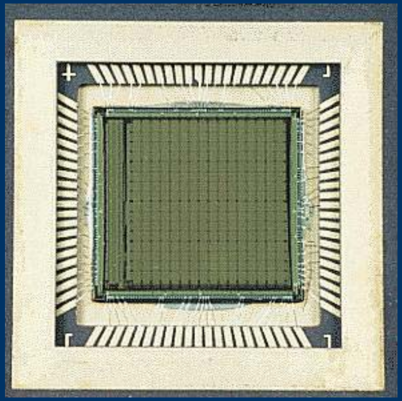


A single correct, executable Cryptol specification can be deployed to a variety of target platforms…



- One specification to 'get right'
- Many targets for use

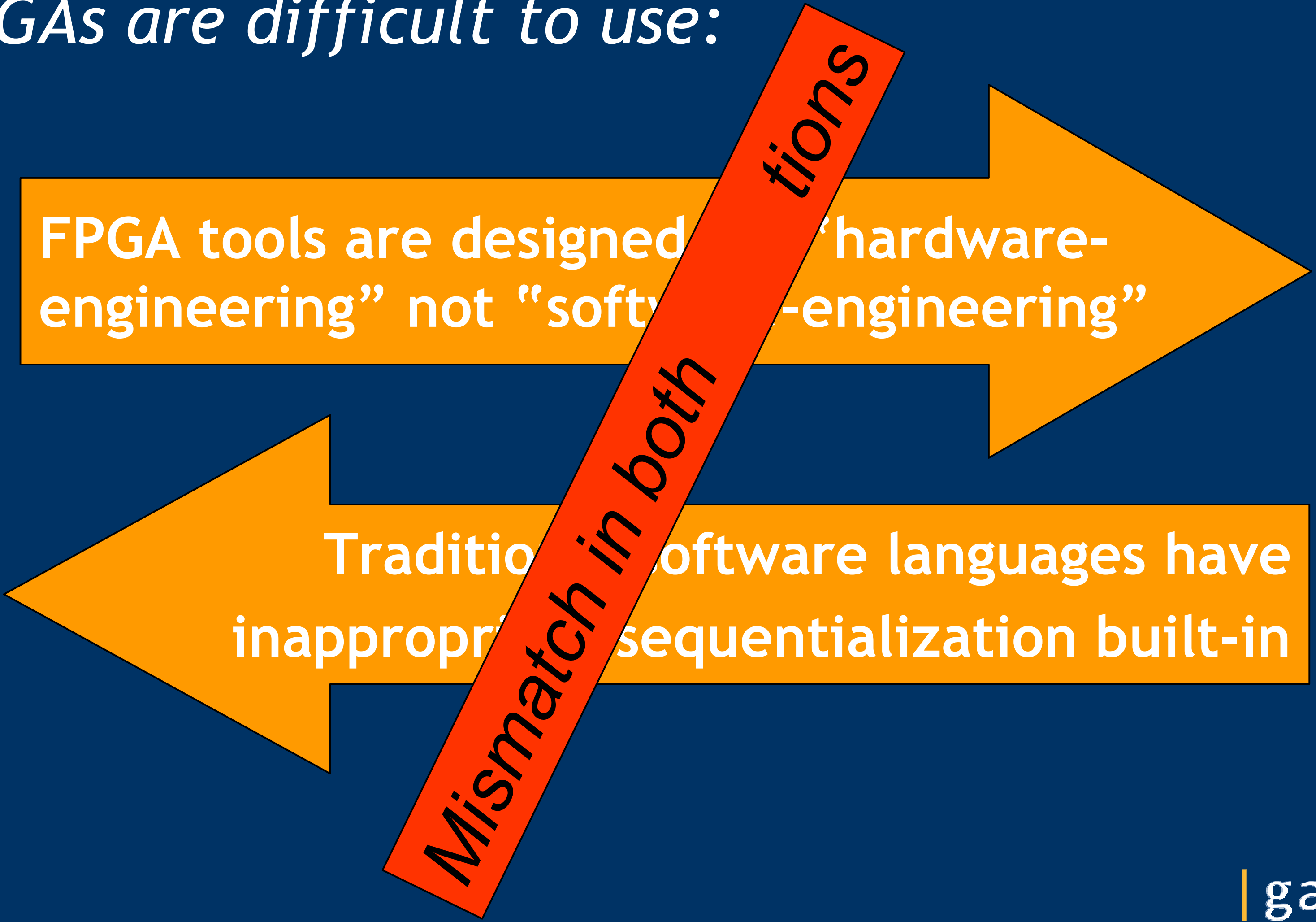
FPGAs: An Opportunity



- **Configurable hardware**
  - Very fast, hugely parallel resource
- **Ubiquitous FPGAs in the future?**
  - Large FPGA farms connected to network servers
  - General FPGA resources attached to computation engines: SGI, SRC, for example
- **The Crypto-domain**
  - Highly parallel encryption/decryption
  - Highly parallel crypto-analysis
- **Natural match between Crypto and FPGAs**
  - Manipulation of bit sequences
  - Parallelism

The Problem

FPGAs are difficult to use:



FPGA tools are designed … “hardware-engineering” not “soft… -engineering”

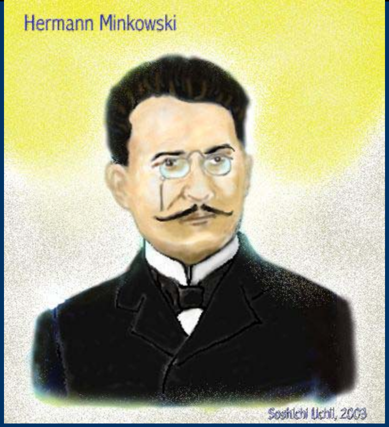
Traditio… oftware languages have inappropr… sequentialization built-in

Mismatch in both … tions

Key Observation

- Sequences are descriptions only
- Implementation of sequences can be:
  - Laid out in time
    - Loops and/or state machines
  - Laid out in space
    - Parallel and/or pipeline
  - Or a mixture of both
    - The mathematical specification is the same

> Henceforth space by itself, and time by itself, are doomed to fade away into mere shadows, and only a kind of union of the two will preserve an independent reality.
>
> *Minkowski,*
> *Space and Time, Sept. 21, 1908*

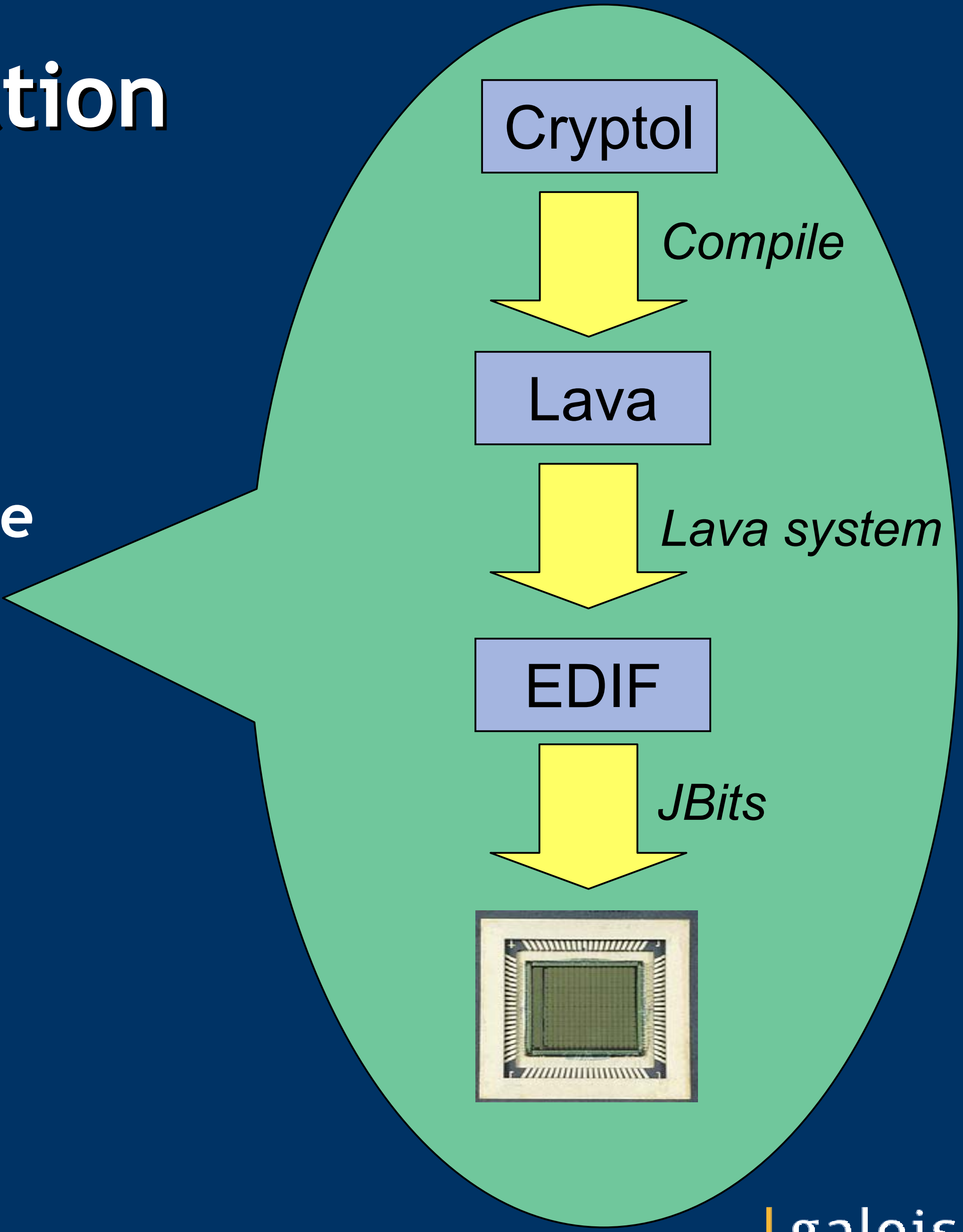


Sequentialization in Cryptol comes only from data-dependency — just like hardware

| galois |

FPGA Compilation Route

- Adapt Cyptol-to-C compiler to produce
  1. Lava (via Jbits)
  2. VHDL



Cryptol

Compile

Lava

Lava system

EDIF

JBits

Naturally Matched Technologies

- **Cryptol**
  - Language designed for crypto-mathematicians
  - Generate finite-state machine descriptions
  - Formal semantics
- **Lava**
  - Language designed for 2D FPGA specification
  - Compute placements
  - Formal semantics

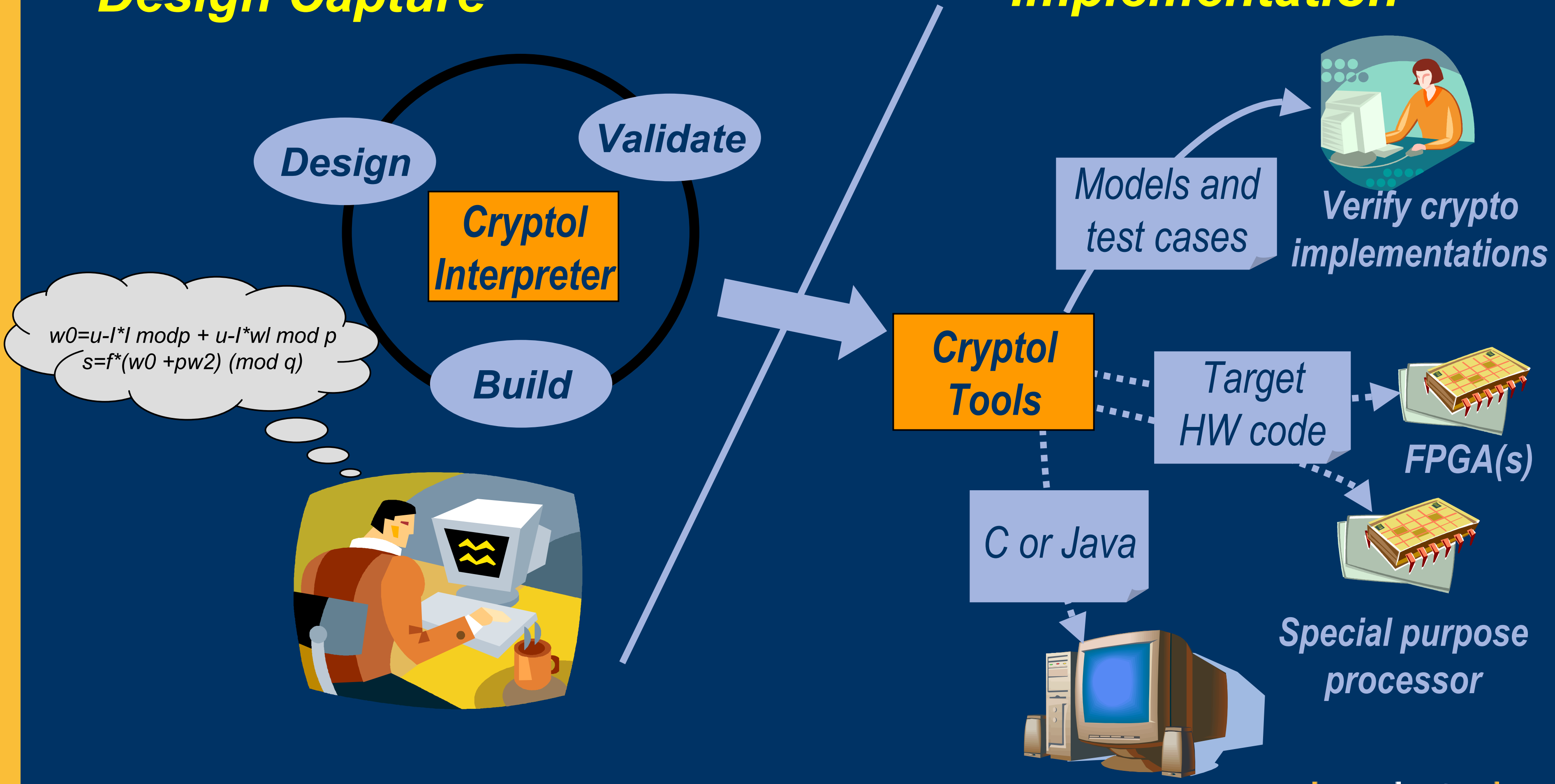
Benefits

- **FPGA resources become available to crypto-mathematicians**
  - Not just to hardware engineers
- **Low barrier-to-entry for FPGA use**
  - Cryptol spec may have been developed for other purposes
  - Standard libraries of Cryptol specifications
  - FPGA implementation is a small delta for the user
- **Cross-compilation development scenario**
  - Develop specs on conventional hardware
  - Execute on FPGA

www.cryptol.net

Domain-Specific Design Capture

Assured Implementation



Design

Validate

Cryptol Interpreter

Build

w0=u-l*l modp + u-l*wl mod p
s=f*(w0 +pw2) (mod q)

Cryptol Tools

Models and test cases

Verify crypto implementations

Target HW code

FPGA(s)

Special purpose processor

C or Java