GALOISCONNECTIONS

purely functional

Cryptol: A Domain Specific Language for Cryptography

`www.cryptol.net`

Cryptol Goals

- Specification correspondence

  "Cryptol programs should be able to look like their specifications"

- Freedom from data entry

  "There shall be no barrier to the programmer specifying a lookup table via a calculation"

- Abstraction Conduction

  "Cryptol should provide a path towards higher-level specifications of Cryptographic algorithms"

Crypto-algorithm domain analysis

- Spoke with crypto-algorithm designers
  - What are the important elements of algorithm specification?
- Studied five AES finalists and DES
  - What do these algorithms have in common?
  - What differences occur between them?
- Embody the domain analysis within a language
  - Obtain feedback from crypto specialists

Relevant Concepts and Abstractions

- Block ciphers
- Vectors and matrices
- Permutations
- Lookup tables
- Various Finite Element arithmetics
- Multiple views of data
- Iteration and recurrence

Block Ciphers

- Interface

```
encrypt : (Xkey,PT) -> CT
decrypt : (Xkey,CT) -> PT
keySchedule : Key -> Xkey
```

- Chained together to operate on streams
- Simple standard stream modes:
  - Electronic Code Book (ECB)
  - Cipher Block Chaining (CBC)

Bit Vectors

- Sizes ranging from 4 bits to 128 bits (8 and 32 most common)
- All the usual boolean ops
  - Exclusive-or prevalent
- Simple modulo arithmetic (+, -, *, /)
- Permutations
  - Mostly just rotations of bit vectors
  - More general permutation used in DES

Bit Vector Operations in RC6

$a + b$	integer addition modulo 2^w
$a - b$	integer subtraction modulo 2^w
$a \oplus b$	bitwise exclusive-or of w-bit words
$a \times b$	integer multiplication modulo 2^w
$a \lll b$	rotate the w-bit word a to the left by the amount given by the least significant $\lg w$ bits of b
$a \ggg b$	rotate the w-bit word a to the right by the amount given by the least significant $\lg w$ bits of b

Lookup Tables

```
WORD Sbox[ ] = {
  0x09d0c479, 0x28c8ffe0, 0x84aa6c39, 0x9dad7287, 0x7dff9be3, 0xd4268361,
  0xc96da1d4, 0x7974cc93, 0x85d0582e, 0x2a4b5705, 0x1ca16a62, 0xc3bd279d,
  0x0f1f25e5, 0x5160372f, 0xc695c1fb, 0x4d7ff1e4, 0xae5f6bf4, 0x0d72ee46,
  0xff23de8a, 0xb1cf8e83, 0xf14902e2, 0x3e981e42, 0x8bf53eb6, 0x7f4bf8ac,
  0x83631f83, 0x25970205, 0x76afe784, 0x3a7931d4, 0x4f846450, 0x5c64c3f6,
  0x210a5f18, 0xc6986a26, 0x28f4e826, 0x3a60a81c, 0xd340a664, 0x7ea820c4,
  0x526687c5, 0x7eddd12b, 0x32a11d1d, 0x9c9ef086, 0x80f6e831, 0xab6f04ad,
  0x56fb9b53, 0x8b2e095c, 0xb68556ae, 0xd2250b0d, 0x294a7721, 0xe21fb253,
  0xae136749, 0xe82aae86, 0x93365104, 0x99404a66, 0x78a784dc, 0xb69ba84b,
  0x04046793, 0x23db5c1e, 0x46cae1d6, 0x2fe28134, 0x5a223942, 0x1863cd5b,
  0xc190c6e3, 0x07dfb846, 0x6eb88816, 0x2d0dcc4a, 0xa4ccae59, 0x3798670d,
  0xcbfa9493, 0x4f481d45, 0xeafc8ca8, 0xdb1129d6, 0xb0449e20, 0x0f5407fb,
  0x6167d9a8, 0xd1f45763, 0x4daa96c3, 0x3bec5958, 0xababa014, 0xb6ccd201,
  0x38d6279f, 0x02682215, 0x8f376cd5, 0x092c237e, 0xbfc56593, 0x32889d2c,
  0x854b3e95, 0x05bb9b43, 0x7dcd5dcd, 0xa02e926c, 0xfae527e5, 0x36a1c330,
  0x3412e1ae, 0xf257f462, 0x3c4f1d71, 0x30a2e809, 0x68e5f551, 0x9c61ba44,
  0x5ded0ab8, 0x75ce09c8, 0x9654f93e, 0x698c0cca, 0x243cb3e4, 0x2b062b97,
  0x0f3b8d9e, 0x00e050df, 0xfc5d6166, 0xe35f9288, 0xc079550d, 0x0591aee8,
  0x8e531e74, 0x75fe3578, 0x2f6d829a, 0xf60b21ae, 0x95e8eb8d, 0x6699486b,
  0x901d7d9b, 0xfd6d6e31, 0x1090acef, 0xe0670dd8, 0xdab2e692, 0xcd6d4365,
  0xe5393514, 0x3af345f0, 0x6241fc4d, 0x460da3a3, 0x7bcf3729, 0x8bf1d1e0,
  0x14aac070, 0x1587ed55, 0x3afd7d3e, 0xd2f29e01, 0x29a9d1f6, 0xefb10c53,
```

Lookup Tables

- AKA S-boxes
  - n-bit by m-bit lookup tables
- Both fixed and data-dependent (TwoFish)
- Fixed S-boxes are often calculated

- Intent is to capture notion of pre-computed values

Generating SBoxes

In the design of the S-box S, we generated the entries of S in a "pseudorandom fashion" and tested that the resulting S-box has good differential and linear properties. The "pseudorandom" S-boxes were generated by setting for $i = 0 \ldots 102$, $j = 0 \ldots 4$, $S[5i+j] = \text{SHA-1}(5i \mid c1 \mid c2 \mid c3)_j$ (where $\text{SHA-1}(\cdot)_j$ is the j'th word in the output of SHA-1). Here we view i as a 32-bit unsigned integer, and $c1, c2, c3$ are some fixed constants. In our implementation we set $c1 =$ `0xb7e15162`, $c2 =$ `0x243f6a88` (which are the binary expansions of the fractional parts in e, π, respectively) and we varied $c3$ until we found an S-box with good properties. We view SHA-1 as an operation on byte-streams, and use little-endian convention to translate between words and bytes.

Matrices

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

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

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

Multiple views

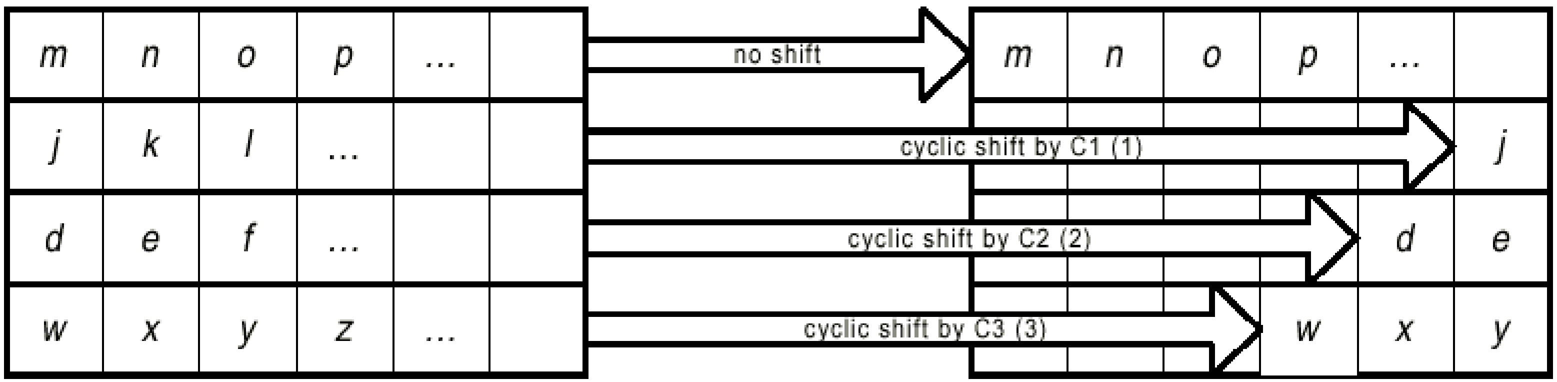


Figure 3: ShiftRow operates on the rows of the State.

Multiple views

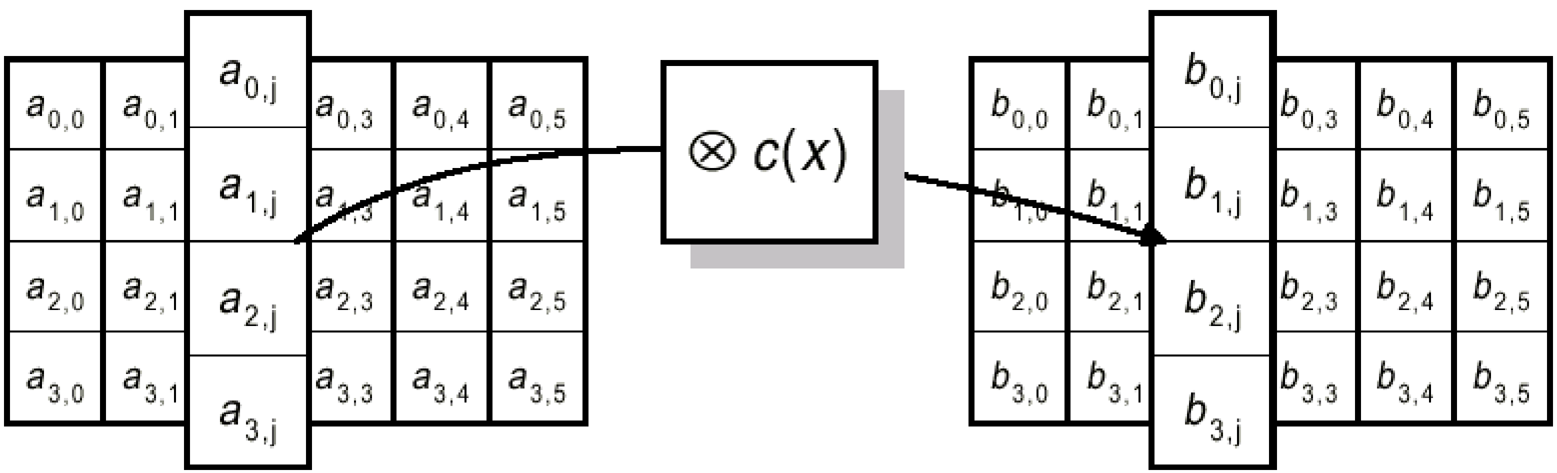


Figure 4: MixColumn operates on the columns of the State.

Matrix Arithmetic

- Matrix/vector multiplication arises on paper

$$
\begin{aligned}
x_i &= \lfloor X/2^{8i} \rfloor \bmod 2^8 \qquad i = 0, \ldots, 3 \\
y_i &= s_i[x_i] \qquad i = 0, \ldots, 3 \\
\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} &= \begin{pmatrix} \cdot & \cdots & \cdot \\ \vdots & \text{MDS} & \vdots \\ \cdot & \cdots & \cdot \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} \\
Z &= \sum_{i=0}^{3} z_i \cdot 2^{8i}
\end{aligned}
$$

- But rarely makes it into the reference code at that level of abstraction

Other Arithmetic

- Polynomials
  - often with bit coefficients
  - different interp of a bit vector

A byte b, consisting of bits $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$, is considered as a polynomial with coefficient in {0,1}:

$$b_7\,x^7 + b_6\,x^6 + b_5\,x^5 + b_4\,x^4 + b_3\,x^3 + b_2\,x^2 + b_1\,x + b_0$$

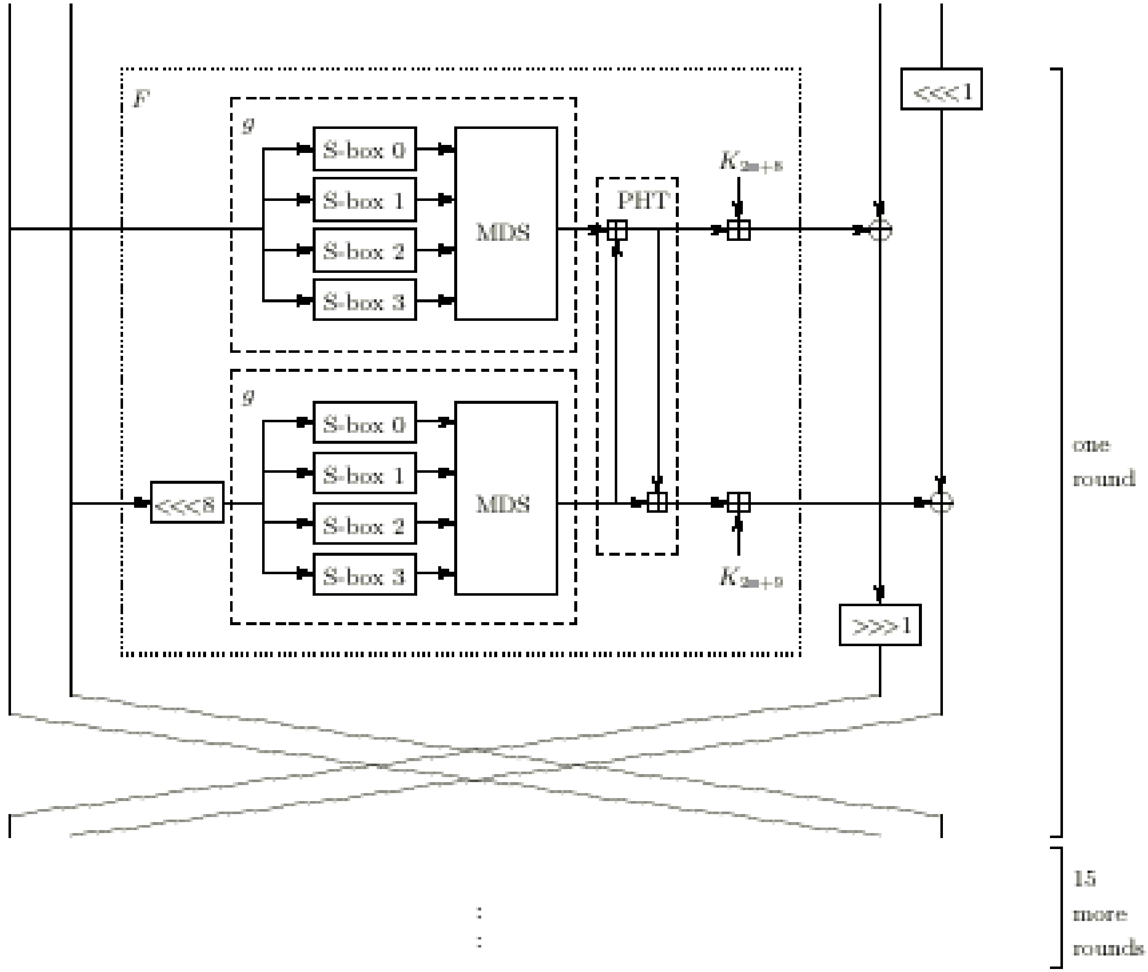
- Galois Fields (TwoFish, Rijndael)

Bounded Iteration

- Crypto tends to avoid interesting control
- For loops
  - Over fixed counts

$$
\begin{array}{l}
\textbf{for } i = 1 \textbf{ to } r \textbf{ do} \\
\quad \{ \\
\qquad t = (B \times (2B + 1)) \lll \lg w \\
\qquad u = (D \times (2D + 1)) \lll \lg w \\
\qquad A = ((A \oplus t) \lll u) + S[2i] \\
\qquad C = ((C \oplus u) \lll t) + S[2i + 1] \\
\qquad (A, B, C, D) = (B, C, D, A) \\
\quad \}
\end{array}
$$

Feistel Network (TwoFish)



Recurrence

- Found in key expansion

$$v = 3 \times \max\{c, 2r+4\}$$

$$\textbf{for } s = 1 \textbf{ to } v \textbf{ do}$$

$$\{$$

$$A = S[i] = (S[i] + A + B) \lll 3$$

$$B = L[j] = (L[j] + A + B) \lll (A + B)$$

$$i = (i+1) \bmod (2r+4)$$

$$j = (j+1) \bmod c$$

$$\}$$

Parameters

- Most algorithms operate on a range of sizes
  - Key
  - Block
- Sizes may be constrained
- Number of iterations may depend on size

Like RC5, RC6 is a fully parameterized family of encryption algorithms. A version of RC6 is more accurately specified as RC6-$w/r/b$ where the word size is w bits, encryption consists of a nonnegative number of rounds r, and b denotes the length of the encryption key in bytes. Since the AES submission is targeted at $w = 32$ and $r = 20$, we shall use RC6 as shorthand to refer to such versions. When any other value of w or r is intended in the text, the parameter values will be specified as RC6-w/r. Of particular relevance to the AES effort will be the versions of RC6 with 16-, 24-, and 32-byte keys.

From Domain Analysis to a Language

Cryptol

Domain-specific language for cryptoalgorithms

Data in Cryptol

- The smallest elements: Bits
- Everything else is a homogeneous matrix

```
[False True False True False False True]

0x4A

[0x3F 0x02 0x41 0xD8]

[[1 2 3 4] [5 6 7 8]]

[1 .. 10]
```

7 single bits

7 (or more) bits

4 elements, each 8 (or more) bits

2 elements, each having 4 elements, each 4 (or more) bits

10 elements, each of 4 (or more) bits

Numbers

- Numbers are matrices of bits
- Decimal, octal (`0o`), hex (`0x`), binary (`0b`)
- Compile-time switch chooses between

Little endian:

```
0xC5 ==
[True False True False False False True True]
```

Big endian:

```
0xC5 ==
[True True False False False True False 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*

Matrices

- Matrix operators
  - Concatenation, indexing, size
  - **`# @ @@ width`**

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

- Zero-based indexing from the left

```
[50 .. 99] @ 10  =  60
```

Shifts and Rotations

- Shifts `<<` `>>`
- Rotations `<<<` `>>>`
- Operate over top-level of a matrix

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

- For words, corresponds to usual notion

```
0xF381 >>> 4
0x1F38
```

Splitting and Joining matrices

0x99FAC6F975BABB3E

↓ split

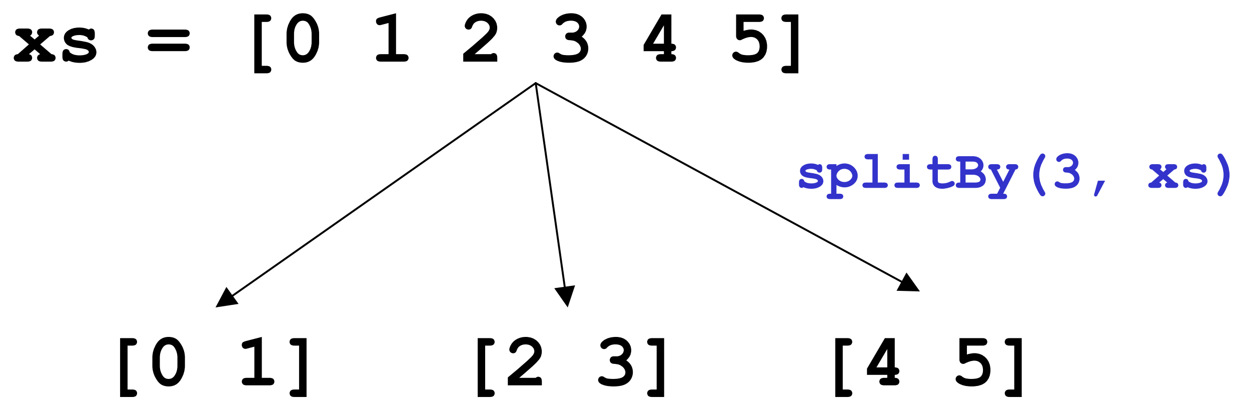
[0x99 0xFA 0xC6 0xF9 0x75 0xBA 0xBB 0x3E]

↓ join

0x99FAC6F975BABB3E

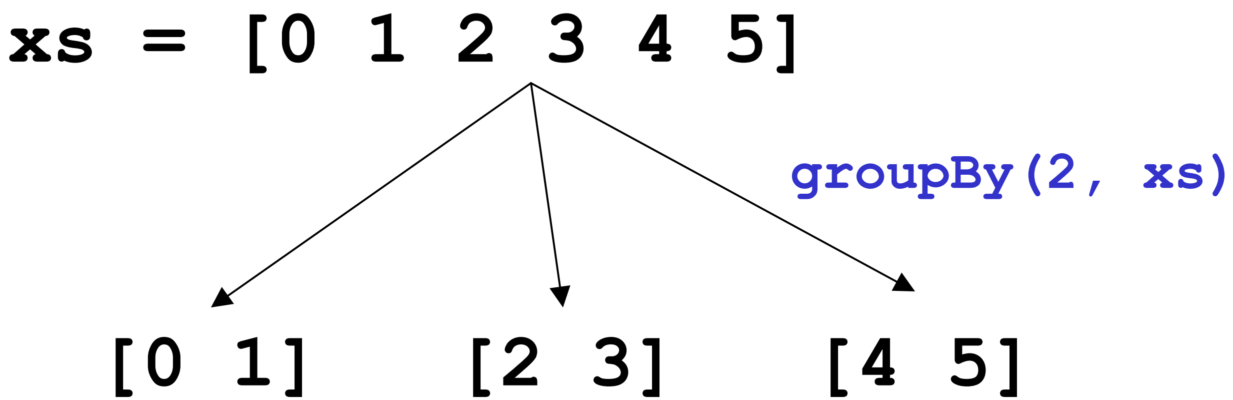
splitBy

- how shall we: **`split [0 1 2 3 4 5]`**?



groupBy

- how shall we: **`split [0 1 2 3 4 5]`**?



Matrix Comprehensions

- Borrowed the comprehension notion from set theory
  - **{ a+b | a** $\in$ **A, b** $\in$ **B}**
  - Adapted to matrices (i.e. 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..2], y <- [3..4] |]
   =  [[0 3] [0 4] [1 3] [1 4] [2 3] [2 4]]
```

- Parallel traversal

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

Row traversals

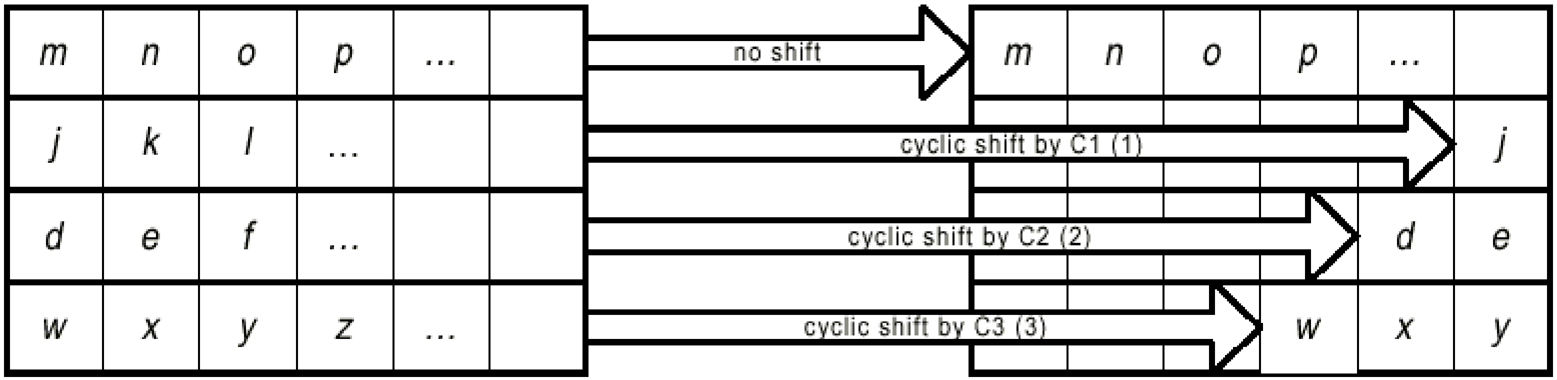


Figure 3: ShiftRow operates on the rows of the State.

```
[| row >>> i || row <- state || i <- shifts |]
```

Column traversals

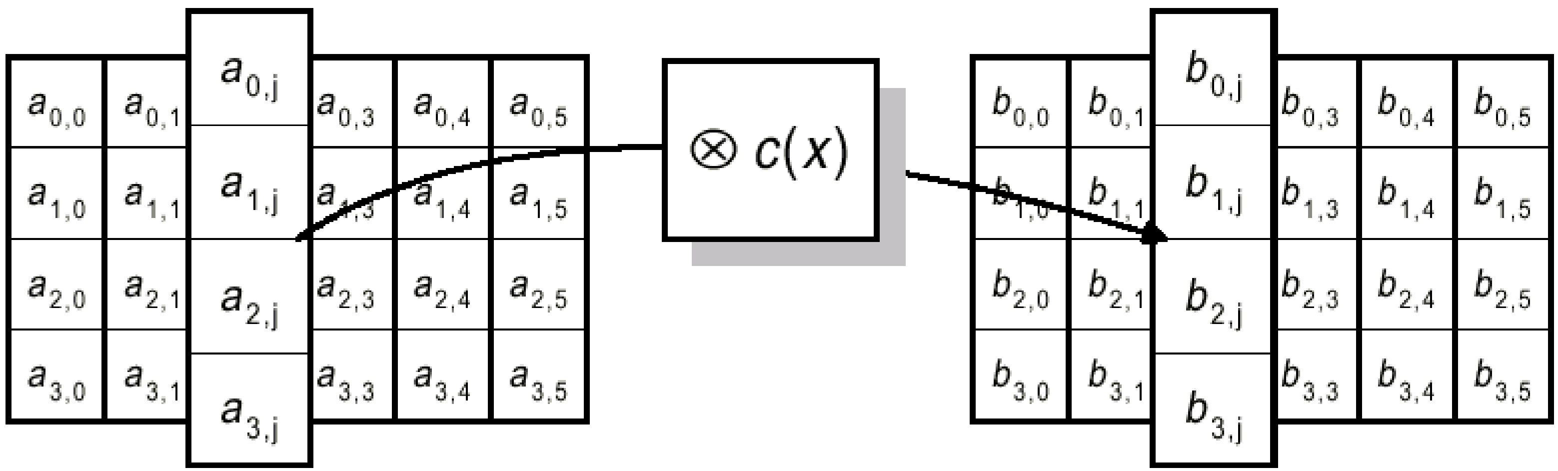


Figure 4: MixColumn operates on the columns of the State.

```
transpose [| ptimes (col, cx) || col <- transpose state |]
```

Nested traversals

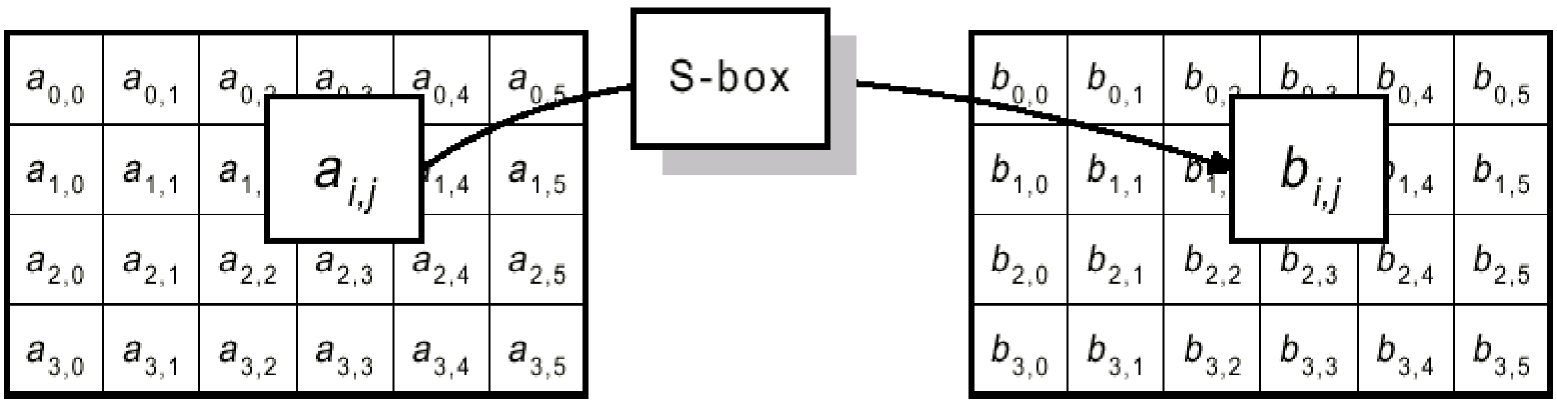


Figure 2: ByteSub acts on the individual bytes of the State.

```
[| [| sbox a || a <- row |] || row <- state |]
```

Cryptol Types

- Capture the size and dimensions of matrices
- Written as a sequence of bracketed dimensions outermost to innermost:

`213`

has type: **`[8]Bit`**

`[ [ 0 1 ] [ 2 3 ] [ 4 5 ] [ 6 7 ] ]`

has type: **`[4][2][8]Bit`**

Cryptol Types

- Capture the size and dimensions of matrices
- Written as a sequence of bracketed dimensions outermost to innermost:

```
213
```

has type: `[8]`

```
[ [ 0 1 ] [ 2 3 ] [ 4 5 ] [ 6 7 ] ]
```

has type: `[4][2][8]`

Cryptol 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];`**

Cryptol Types

- “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];`

Size Polymorphism


```
add32 (0xB4, 0x3A)
```

Size Polymorphism

How many bits am I?

At least 6 bits ...

```
x = 0x3A
```

```
x : {a} (a >= 6) => [a]
```

Shape Polymorphism

What types do I handle?

Four of something to four of the same thing...

```
swab [a b c d] = [d c b a]
```

```
swab : {a} [4]a -> [4]a
```

Syntax of Types

```
P ::= {a1 .. ai} (P1, ..., Pj) =>
        (T1, ..., Tk) -> T

T ::= a                S ::= a
    | Bit                  | Nat
    | [S]T                 | f (T1, ..., Tn)
                           | T1 (+) T2
                           | inf
 f   ::= width, lg2, min, max
(+)  ::= +, -, *, /, %, **
```

Type Inference

- Literals

```
17
```

```
($46 >= 5) => [$46]
```

- Variables

```
X = 7;
X : {a} (a >= 3) [a];

... blatz_X  ^ frobulate X ....
```

```
$23 >= (3) [$23]
```

```
($98 >= 3) => [$98]
```

Definitions are Polymorphic

```
X = 7;
X : {a} (a >= 3) [a];

blatz     : [8] -> [16];
frobulate : [4] -> [32];

... blatz X  ^  frobulate X    ...
```

Parameters are not!

```
blatz     : [8] -> [16];
frobulate : [4] -> [32];


cryptofun X =


... blatz X  ^  frobulate X    ...
```

Type Inference

- Matrices

```
[1 23 45 x]
```

```
[4]$21
```

```
($17 >= 1) => [$17]
($18 >= 5) => [$18]
($19 >= 6) => [$19]
      () => $20
```

```
($17 >= 1,
 $18 >= 5,
 $19 >= 6,
[$17] == $21,
[$18] == $21,
[$19] == $21,
 $20 == $21)
  => [4]$21
```

Type Constraints

- Constraints are dealt with in one of 4 ways:
  - Become part of a type in a binding

```
x = 13;
```

```
x : {a} (a >= 4) => [a]
```

```
a >= 4
```

  - Resolved

```
y : [6];
z = 13 + y;
```

`a` must be `6`!

  - Unresolvable

```
q : [7];
w = q + y;
```

`6` is not equal to `7`!

Unresolved Constraints

- Failure to match a signature

```
x : {a} [a];
x = 13;
```

a >= 4

The signature is more general than the inferred type!

- Ambiguous

```
xs : [100][16];
z = xs @ 4;
zs = xs @@ [1 .. ];
```

a >= 3

z : [16];

Defaulting

- Ambiguous constraints are subject to defaulting

  `a >= 4` becomes **`a == 4`**

- Defaulting is not always desirable

  1 + 1 = ...

User Feedback – the Positives

- arbitrary bit widths
- ease of rearranging data
- streams
- interactive development
- declarative nature
- ease of extracting substructures
- no need to worry about space allocation
- bulk operations
- feedback from types

User Feedback – the Negatives

- no emacs mode
- interference from types
- no control over defaulting
- not enough higher-level abstractions
- no facility to format data
- want better debugging support
- type errors can be difficult to understand
- want more control over endianness

The Endian Problem

- Cryptol made the following design choices:
  - matrices indexed from the left

    $[\ x_0\ x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ ]$

    consistent with streams

  - literals indexed from the right

    $x_7\ x_6\ x_5\ x_4\ x_3\ x_2\ x_1\ x_0$

    consistent with little endian

  - e.g.: `0b10100110 ==`

    ```
    [False True True False
     False True False True]
    ```

The Endian Problem

- But there are consequences:
  - the swap: `0xab # 0xcd == 0xcdab`
    `split 0xcdab == [0xab 0xcd]`
  - in a left shift, whose "left" is it anyway?

```
0b0111 << 1
     == 0b1110

[True True True False] << 1
     == [True True False False]
     == 0b0011
```

The Endian Problem

- Could chose differently:
  - matrices still indexed from the left
  - literals also indexed from the left
- Avoids swap and shift problems
- But less natural for encodings of numbers
- any fixed choice will lose:
  specifications feel free to use different conventions, thus we set up a road block to *specification correspondence*

Endianess

- Design space:
  - (syntactic) Bit 0 on the left or on the right
  - (semantic) Bit 0 least or most significant

	MSB	LSB
right	1000	0001
left	0001	1000

$$a = \sum_{i=0}^{23} a_i \cdot 2^{32i},$$

where each a_i is a 32–bit integer. If we write a and the a_i's as bit streams then a is just the concatenation of all the a_i's. That is, let $||$ denote concatenation. Then we write

$$a = (\, a_{23} \,||\, a_{22} \,||\, \ldots \,||\, a_1 \,||\, a_0).$$

The expression for $a \mod p$ turns out to be

$$a = t + 2s_1 + s_2 + s_3 + s_4 + s_5 + s_6 - d_1 - d_2 - d_3 \mod p.$$

Here the s_i's and d_i's are 384–bit numbers defined by:

$$\begin{array}{lcl}
t & = & (\, a_{11} \,||\, a_{10} \,||\, a_9 \,||\, a_8 \,||\, a_7 \,||\, a_6 \,||\, a_5 \,||\, a_4 \,||\, a_3 \,||\, a_2 \,||\, a_1 \,||\, a_0 \,) \\
s_1 & = & (\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, a_{23} \,||\, a_{22} \,||\, a_{21} \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,) \\
s_2 & = & (\, a_{23} \,||\, a_{22} \,||\, a_{21} \,||\, a_{20} \,||\, a_{19} \,||\, a_{18} \,||\, a_{17} \,||\, a_{16} \,||\, a_{15} \,||\, a_{14} \,||\, a_{13} \,||\, a_{12} \,) \\
s_3 & = & (\, a_{20} \,||\, a_{19} \,||\, a_{18} \,||\, a_{17} \,||\, a_{16} \,||\, a_{15} \,||\, a_{14} \,||\, a_{13} \,||\, a_{12} \,||\, a_{23} \,||\, a_{22} \,||\, a_{21} \,) \\
s_4 & = & (\, a_{19} \,||\, a_{18} \,||\, a_{17} \,||\, a_{16} \,||\, a_{15} \,||\, a_{14} \,||\, a_{13} \,||\, a_{12} \,||\, a_{20} \,||\, 0 \,||\, a_{23} \,||\, 0 \,) \\
s_5 & = & (\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, a_{23} \,||\, a_{22} \,||\, a_{21} \,||\, a_{20} \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,) \\
s_6 & = & (\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, a_{23} \,||\, a_{22} \,||\, a_{21} \,||\, 0 \,||\, 0 \,||\, a_{20} \,) \\
d_1 & = & (\, a_{22} \,||\, a_{21} \,||\, a_{20} \,||\, a_{19} \,||\, a_{18} \,||\, a_{17} \,||\, a_{16} \,||\, a_{15} \,||\, a_{14} \,||\, a_{13} \,||\, a_{12} \,||\, a_{23} \,) \\
d_2 & = & (\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, a_{23} \,||\, a_{22} \,||\, a_{21} \,||\, a_{20} \,||\, 0 \,) \\
d_3 & = & (\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, a_{23} \,||\, a_{23} \,||\, 0 \,||\, 0 \,||\, 0 \,)
\end{array}$$

The Endian Problem: Exploring the Solution Space

- Use a declarative approach
  - associate endianness with the type
- Use an operator-based approach
  - Special versions of operators:
    - **`splitBE`, `splitLE`, `joinBE`, `joinLE`**
    - what to do about **`#`**, `@`?

Sample Key Expansion Fragment

```
keyX key = ss @@ [ 0 .. n ]
  where {
    initS, initL : [1][32];
    initS = [0];
    initL = split (join key);
    ss = [|(s + a + b) <<< 3 || s <- initS # ss
                             || a <- [1] # ss
                             || b <- [0] # ls |];
    ls = [| (l + a + b) <<< (a + b)
                             || l <- initL # ls
                             || a <- ss
                             || b <- [0] # ls |];
  };
```

Generated C code

```
extern uint32* keyX32(uint32*);
uint32* keyX32 (uint32* key_keyX32)
{    static uint32 arr0[2];
     uint32 v6, v5, v4, v3, v0;
     v0=joinWord(4, 8, key_keyX32);
     v3=(v0)+(8);
     v4=ROL(v3, 8);
     v5=(16)+(v4);
     v6=ROL(v5, 3);
     arr0[0]=8;
     arr0[1]=v6;
     return (arr0); }
```

Sample Key Expansion Fragment

```
keyX key = ss @@ [ 0 .. n ]
  where {
    initS, initL : [1][36];
    initS = [0];
    initL = split (join key);
    ss = [|(s + a + b) <<< 3 || s <- initS # ss
                             || a <- [1] # ss
                             || b <- [0] # ls |];
    ls = [| (l + a + b) <<< (a + b)
                             || l <- initL # ls
                             || a <- ss
                             || b <- [0] # ls |];
  };
```

```
extern uint32** keyX36(uint32*);
uint32** keyX36 (uint32* key_keyX36)
{   static uint32* arr48[2];
    static uint32 vec49[2]={8UL, 0UL};
    ...
    copyOuter(2, v1, splitMatrix(1, 36, v0, arr38));
    copyOuter(2, v3,
      plusMatrix(0, 1, arrShape40, v1, vec39, arr41));
    copyOuter(2, v4, rolMatrix(36, v3, vec42, arr43));
    copyOuter(2, v5,
      plusMatrix(0, 1, arrShape45, vec44, v4, arr46));
    copyOuter(2, v6, rolMatrix(36, v5, 3, arr47));
    arr48[0]=vec49;
    arr48[1]=v6;
    return (arr48); }
```

The Future of Cryptol

- Crypto was in the mud, and now we've at least got it on dry land.
- Now we're headed towards higher ground:

machine words

↓

abstract crypto machine

↓

executable mathematics

Future Directions

- User-defined operators
- Extended matrix comprehensions
- Support for more arithmetics:
  - Flexible Precision arithmetic
  - Polynomial arithmetic
  - Arbitrary modulus arithmetic
- Support for 1-based and other indexing
- Flexible endian-ness

Flexible Precision Arithmetic

- Free ourselves from the bonds of power-of-two modulus arithmetic
- Not arbitrary precision arithmetic
- New operators:

  +′ : ([a], [b]) -> [max(a, b) + 1]

  *′ : ([a], [b]) -> [a + b]

  %′ : ([a], [b]) -> [b]

Polynomial Arithmetic

- **0x1a3** ≡ $x^8 + x^7 + x^5 + x^1 + 1$
- New operators:

```
padd :  {a b} ([a], [b]) -> [max(a, b)]
pmult : {a b} ([a],[b]) -> [a+b-1]
pdiv :  {a b} ([a],[b]) -> [1+a-b]
pmod :  {a b} ([a],[1+b]) -> [b]
```

Arbitrary Modulus Arithmetic

- New operators:

```
+% : (Mod n, a == width (n-1)) =>
       ([a], [a]) -> [a]
*% : (Mod n, a == width (n-1)) =>
       ([a], [a]) -> [a]
...
```

- New binding form:

```
(x + f y) withModulus 17
```

Future Directions

- Enhanced tracing/debugging
  - dump out internal registers from a run
  - format as LaTeX, HTML, ...
- Interface to C
- Interface to HCDSA
- Generate optimized, low-level code
  - FPGA
  - commercial crypto chip (e.g. AIM)

The End

www.cryptol.net