

# Cryptol Tutorial

## Part 2:

## The Second Part

Sean Weaver

HCSS 2011

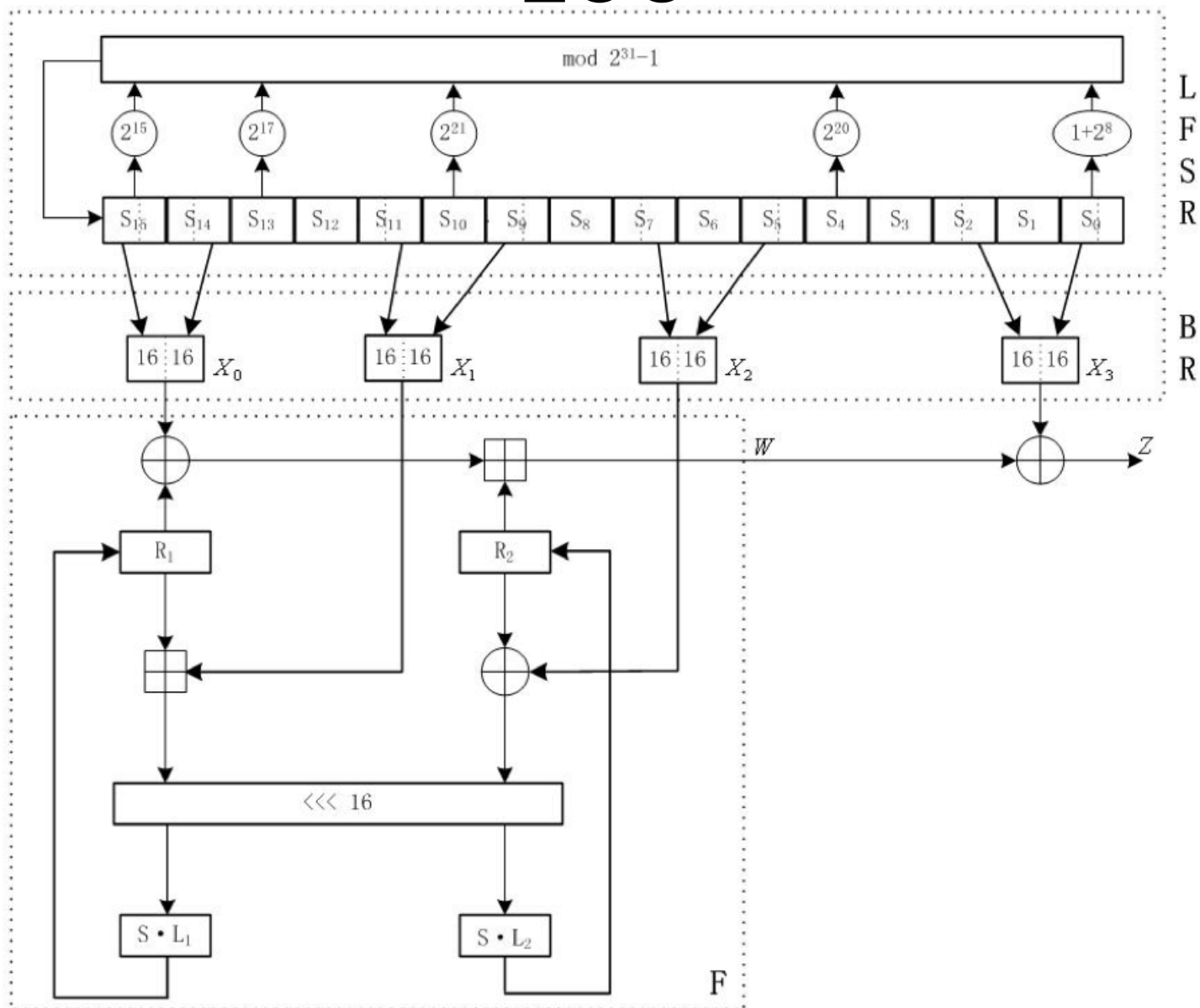
# Outline

- Cryptol Demo
  - The ZUC stream cipher
- Verification of Inferior Language Source Code
  - Java AES
  - Java MD5

# ZUC

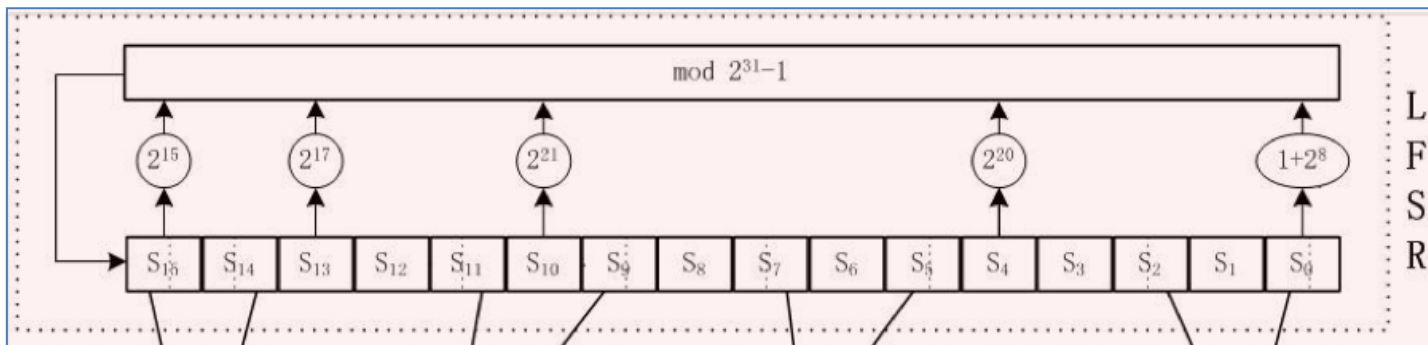
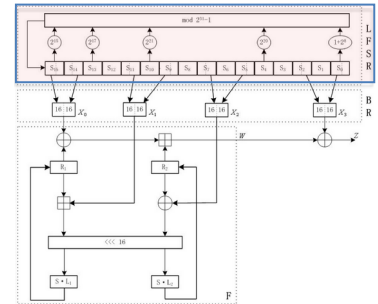
- Latest initiative by 3GPP for securing mobile networks<sup>[1]</sup>
- Word-oriented stream cipher
  - 128-bit key
  - 128-bit initialization vector
  - Generates keystream of 32-bit words
- Forms the heart of the 3GPP confidentiality algorithm 128-EEA3 and the 3GPP integrity algorithm 128-EIA3<sup>[1]</sup>

# ZUC



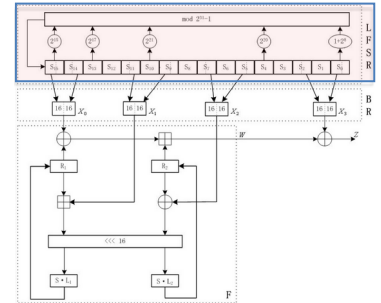
# LFSR

- “The linear feedback shift register (LFSR) has 16 of 31-bit cells ( $s_0, s_1, \dots, s_{15}$ )...” [2]
- “The LFSR has 2 modes of operations: the initialization mode and the working mode. In the initialization mode, the LFSR receives a 31-bit input word  $u$ ...” [2]



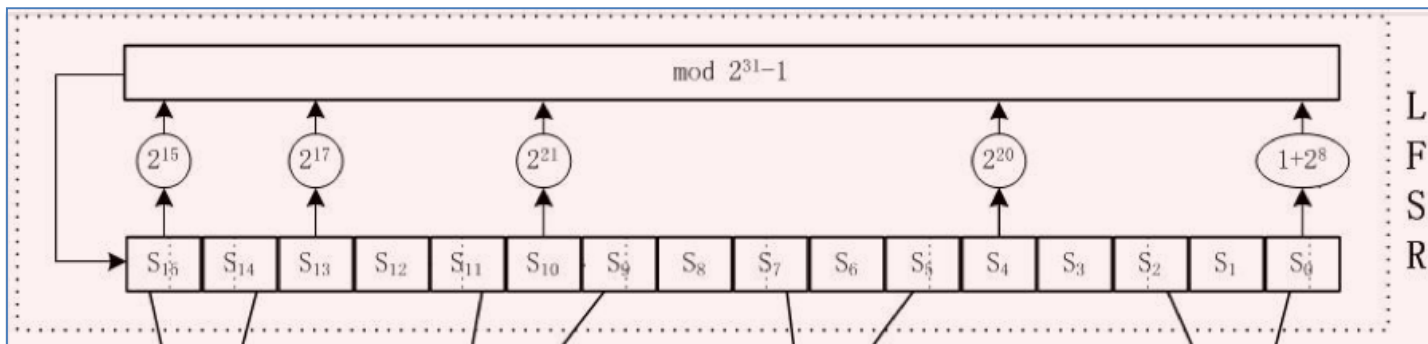
# LFSR

- “The linear feedback shift register (LFSR) has 16 of 31-bit cells ( $s_0, s_1, \dots, s_{15}$ )...” [2]
- “The LFSR has 2 modes of operations: the initialization mode and the working mode. In the initialization mode, the LFSR receives a 31-bit input word  $u$ ...” [2]



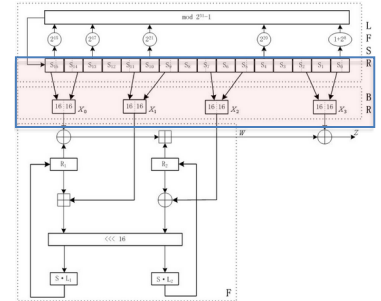
```
LFSRWithWorkMode : [16][31] -> [16][31];
```

```
LFSRWithInitialisationMode : ([31], [16][31]) -> [16][31];
```



# Bit Reorganization

- “The middle layer of the algorithm is the bit-reorganization. It extracts 128 bits from the cells of the LFSR and forms 4 of 32-bit words...” [2]



```
Bitreorganization()
```

```
{
```

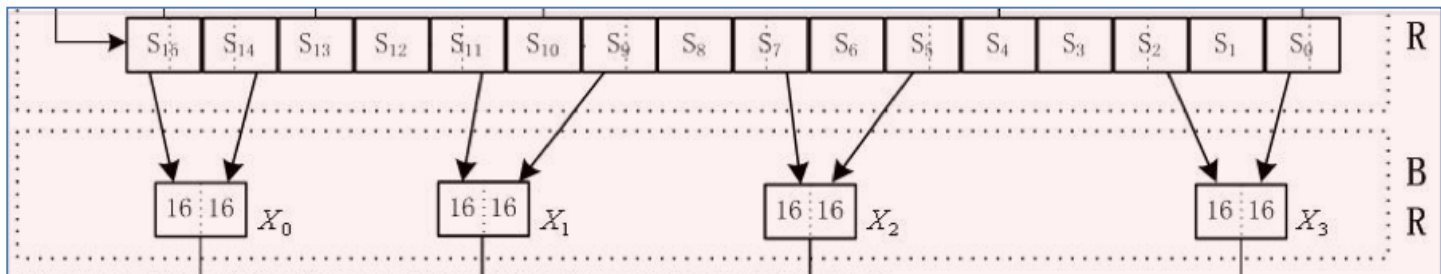
$$1. X_0 = s_{15H} \parallel s_{14L}$$

$$2. X_1 = s_{11L} \parallel s_{9H}$$

$$3. X_2 = s_{7L} \parallel s_{5H}$$

$$4. X_3 = s_{2L} \parallel s_{0H}$$

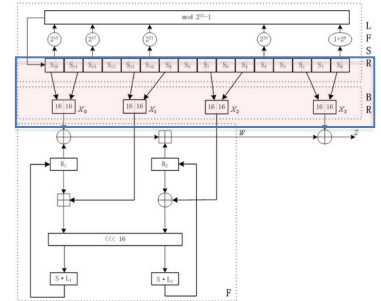
```
}
```



# H and L

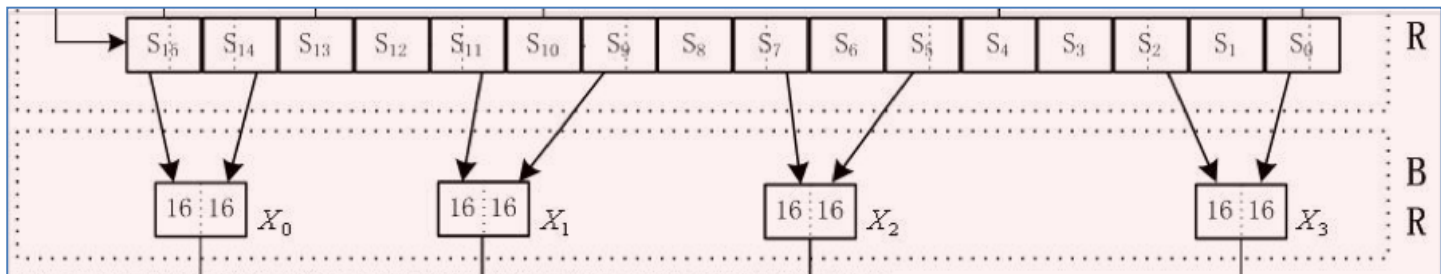
- “ $a_H$ : The leftmost 16 bits of integer  $a$ .” [2]

$H : \{b\} \text{ (fin } b, b \geq 16) \Rightarrow [b] \rightarrow [16];$   
 $H(a) = \text{drop}(\text{width}(a)-16, a);$



- “ $a_L$ : The rightmost 16 bits of integer  $a$ .” [2]

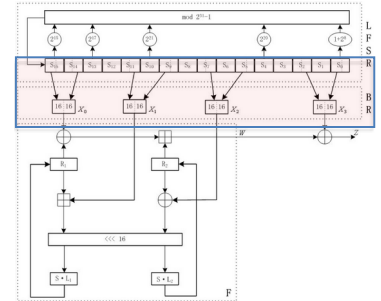
$L : \{b\} \text{ (} b \geq 16) \Rightarrow [b] \rightarrow [16];$   
 $L(a) = \text{take}(16, a);$





# Bit Reorganization

- “The middle layer of the algorithm is the bit-reorganization. It extracts 128 bits from the cells of the LFSR and forms 4 of 32-bit words...” [2]



```
Bitreorganization()
```

```
{
```

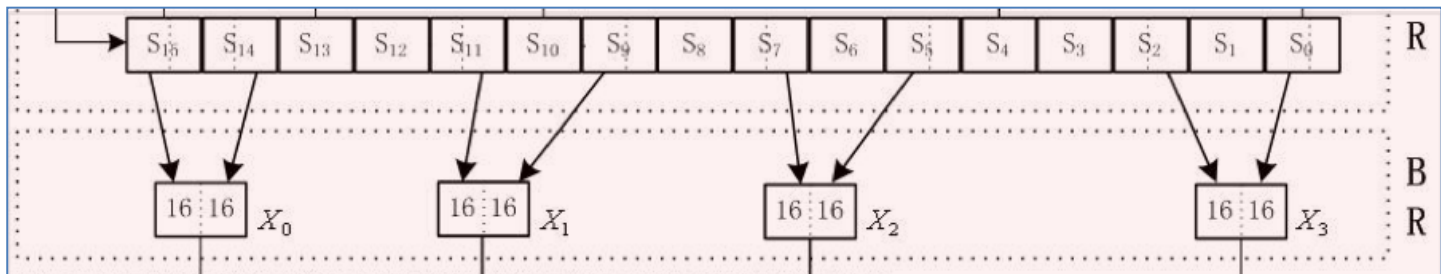
$$1. X_0 = s_{15H} \parallel s_{14L}$$

$$2. X_1 = s_{11L} \parallel s_{9H}$$

$$3. X_2 = s_{7L} \parallel s_{5H}$$

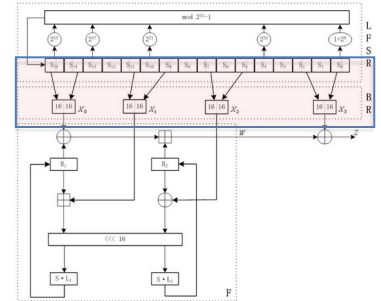
$$4. X_3 = s_{2L} \parallel s_{0H}$$

```
}
```



# Bit Reorganization

- “The middle layer of the algorithm is the bit-reorganization. It extracts 128 bits from the cells of the LFSR and forms 4 of 32-bit words...” [2]



```
Bitreorganization()
```

```
{
```

```
1.  $X_0 = s_{15H} || s_{14L}$ 
```

```
2.  $X_1 = s_{11L} || s_{9H}$ 
```

```
3.  $X_2 = s_{7L} || s_{5H}$ 
```

```
4.  $X_3 = s_{2L} || s_{0H}$ 
```

```
}
```

```
Bitreorganization : [16][31] -> [4][32];
```

```
Bitreorganization (s) = [X0 X1 X2 X3]
```

```
where {
```

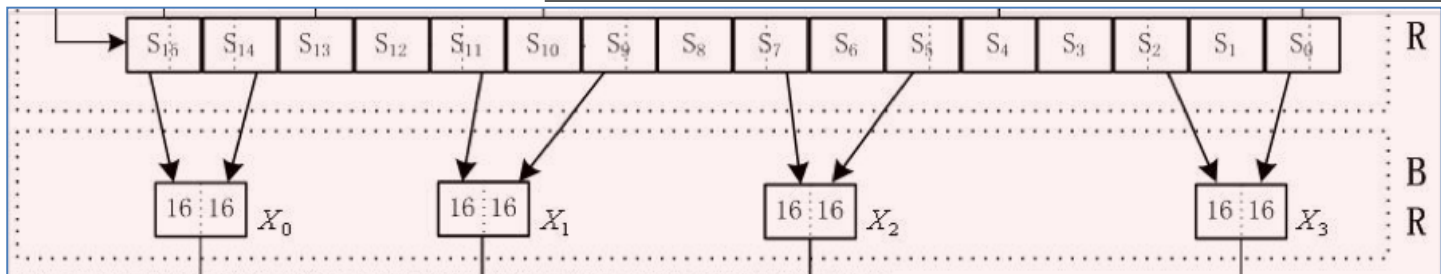
```
 $X0 = L(s@14) \# H(s@15);$ 
```

```
 $X1 = H(s@9) \# L(s@11);$ 
```

```
 $X2 = H(s@5) \# L(s@7);$ 
```

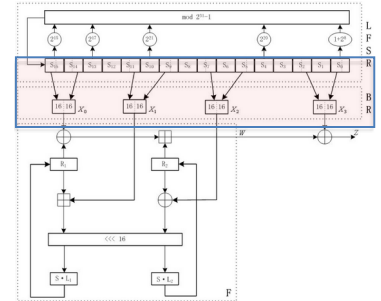
```
 $X3 = H(s@0) \# L(s@2);$ 
```

```
};
```



# Bit Reorganization

- “The middle layer of the algorithm is the bit-reorganization. It extracts 128 bits from the cells of the LFSR and forms 4 of 32-bit words...” [2]



```
Bitreorganization()
{
```

```
1. X0 = s15H || s14L
2. X1 = s11L || s9H
3. X2 = s7L || s5H
4. X3 = s3L || s1H
```

```
Bitreorganization : [16][31] -> [4][32];
Bitreorganization (s) = [X0 X1 X2 X3]
```

where {

```
X0 = L(s@14) # H(s@15);
X1 = H(s@9) # L(s@11);
X2 = H(s@5) # L(s@7);
X3 = H(s@0) # L(s@2);
```

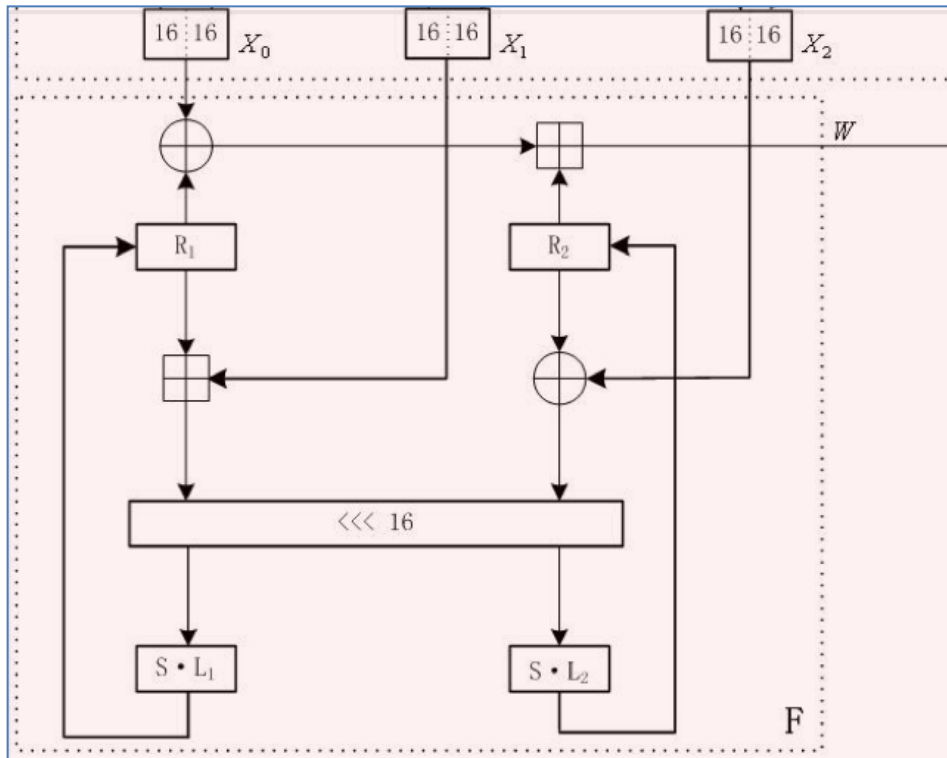
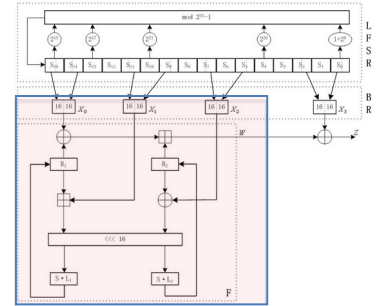
```
void Bitreorganization() {
```

```
    BRC_X0 = ((LFSR_S15 & 0x7FFF8000) << 1) | (LFSR_S14 & 0xFFFF);
    BRC_X1 = ((LFSR_S11 & 0xFFFF) << 16) | (LFSR_S9 >> 15);
    BRC_X2 = ((LFSR_S7 & 0xFFFF) << 16) | (LFSR_S5 >> 15);
    BRC_X3 = ((LFSR_S2 & 0xFFFF) << 16) | (LFSR_S0 >> 15);
```

```
}
```

# F

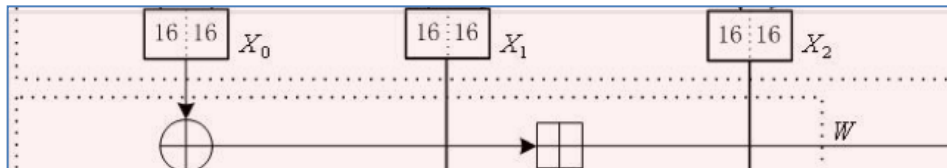
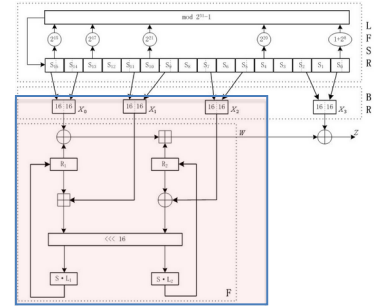
- “The nonlinear function F has 2 of 32-bit memory cells R1 and R2. Let the inputs to F be X0, X1 and X2, which come from the outputs of the bit-reorganization (see section 3.3), then the function F outputs a 32-bit word W.” [2]



$F(X_0, X_1, X_2)$   
 $\{$   
 1.  $W = (X_0 \oplus R_1) \boxplus R_2$   
 2.  $W_1 = R_1 \boxplus X_1$   
 3.  $W_2 = R_2 \oplus X_2$   
 4.  $R_1 = S(L_1(W_{1L} || W_{2H}))$   
 5.  $R_2 = S(L_2(W_{2L} || W_{1H}))$   
 $\}$

# F

- “The nonlinear function F has 2 of 32-bit memory cells R1 and R2. Let the inputs to F be X0, X1 and X2, which come from the outputs of the bit-reorganization (see section 3.3), then the function F outputs a 32-bit word W.” [2]



```

F : ([3][32], [2][32]) -> ([32], [2][32]);
F([X0 X1 X2], [R1 R2]) = (W, [R1' R2'])
  where {
    W    = (X0 ^ R1) + R2;
    W1   = R1 + X1;
    W2   = R2 ^ X2;
    R1'  = S(L1(H(W2) # L(W1)));
    R2'  = S(L2(H(W1) # L(W2)));
  };

```

```

F (X0, X1, X2)
{
  1. W = (X0 ⊕ R1) ⊞ R2
  2. W1 = R1 ⊞ X1
  3. W2 = R2 ⊕ X2
  4. R1 = S(L1(W1L || W2H))
  5. R2 = S(L2(W2L || W1H))
}

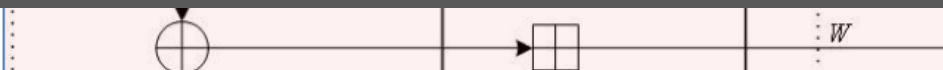
```

**C**

```

u32 F() {
    u32 W, W1, W2, u, v;
    W = (BRC_X0 ^ F_R1) + F_R2;
    W1 = F_R1 + BRC_X1;
    W2 = F_R2 ^ BRC_X2;
    u = L1((W1 << 16) | (W2 >> 16));
    v = L2((W2 << 16) | (W1 >> 16));
    F_R1 = MAKEU32(S0[u >> 24], S1[(u >> 16), S0[(u >> 8) & 0xFF], S1[u & 0xFF]);
    F_R2 = MAKEU32(S0[v >> 24], S1[(v >> 16), S0[(v >> 8) & 0xFF], S1[v & 0xFF]);
    return W;
}

```



$F : ([3][32], [2][32]) \rightarrow ([32], [2][32]);$   
 $F([X_0 \ X_1 \ X_2], [R_1 \ R_2]) = (W, [R_1' \ R_2'])$   
 where {  
      $W = (X_0 \wedge R_1) + R_2;$   
      $W_1 = R_1 + X_1;$   
      $W_2 = R_2 \wedge X_2;$   
      $R_1' = S(L_1(H(W_2) \# L(W_1)));$   
      $R_2' = S(L_2(H(W_1) \# L(W_2)));$   
 };

$F(X_0, X_1, X_2)$   
 {  
     1.  $W = (X_0 \oplus R_1) \boxplus R_2$   
     2.  $W_1 = R_1 \boxplus X_1$   
     3.  $W_2 = R_2 \oplus X_2$   
     4.  $R_1 = S(L_1(W_{1L} || W_{2H}))$   
     5.  $R_2 = S(L_2(W_{2L} || W_{1H}))$   
 }



# S-boxes

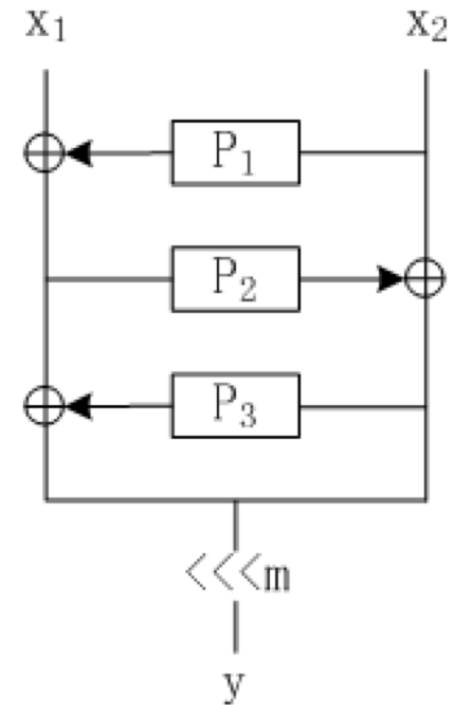
- “The 32x32 S-box S is composed of 4 juxtaposed 8x8 S-boxes, i.e.,  $S=(S_0, S_1, S_2, S_3)$ , where  $S_0=S_2$ ,  $S_1=S_3$ ” [2]

|   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | A  | B  | C  | D  | E  | F  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 3E | 72 | 5B | 47 | CA | E0 | 00 | 33 | 04 | D1 | 54 | 98 | 09 | B9 | 6D | CB |
| 1 | 7B | 1B | F9 | 32 | AF | 9D | 6A | A5 | B8 | 2D | FC | 1D | 08 | 53 | 03 | 90 |
| 2 | 4D | 4E | 84 | 99 | E4 | CE | D9 | 91 | DD | B6 | 85 | 48 | 8B | 29 | 6E | AC |
| 3 | CD | C1 | F8 | 1E | 73 | 43 | 69 | C6 | B5 | BD | FD | 39 | 63 | 20 | D4 | 38 |
| 4 | 76 | 7D | B2 | A7 | CF | ED | 57 | C5 | F3 | 2C | BB | 14 | 21 | 06 | 55 | 9B |
| 5 | E3 | EF | 5E | 31 | 4F | 7F | 5A | A4 | 0D | 82 | 51 | 49 | 5F | BA | 58 | 1C |
| 6 | 4A | 16 | D5 | 17 | A8 | 92 | 24 | 1F | 8C | FF | D8 | AE | 2E | 01 | D3 | AD |
| 7 | 3B | 4B | DA | 46 | EB | C9 | DE | 9A | 8F | 87 | D7 | 3A | 80 | 6F | 2F | C8 |
| 8 | B1 | B4 | 37 | F7 | 0A | 22 | 13 | 28 | 7C | CC | 3C | 89 | C7 | C3 | 96 | 56 |
| 9 | 07 | BF | 7E | F0 | 0B | 2B | 97 | 52 | 35 | 41 | 79 | 61 | A6 | 4C | 10 | FE |
| A | BC | 26 | 95 | 88 | 8A | B0 | A3 | FB | C0 | 18 | 94 | F2 | E1 | E5 | E9 | 5D |
| B | D0 | DC | 11 | 66 | 64 | 5C | EC | 59 | 42 | 75 | 12 | F5 | 74 | 9C | AA | 23 |
| C | 0E | 86 | AB | BE | 2A | 02 | E7 | 67 | E6 | 44 | A2 | 6C | C2 | 93 | 9F | F1 |
| D | F6 | FA | 36 | D2 | 50 | 68 | 9E | 62 | 71 | 15 | 3D | D6 | 40 | C4 | E2 | 0F |
| E | 8E | 83 | 77 | 6B | 25 | 05 | 3F | 0C | 30 | EA | 70 | B7 | A1 | E8 | A9 | 65 |
| F | 8D | 27 | 1A | DB | 81 | B3 | A0 | F4 | 45 | 7A | 19 | DF | EE | 78 | 34 | 60 |

|   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | A  | B  | C  | D  | E  | F  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 55 | C2 | 63 | 71 | 3B | C8 | 47 | 86 | 9F | 3C | DA | 5B | 29 | AA | FD | 77 |
| 1 | 8C | C5 | 94 | 0C | A6 | 1A | 13 | 00 | E3 | A8 | 16 | 72 | 40 | F9 | F8 | 42 |
| 2 | 44 | 26 | 68 | 96 | 81 | D9 | 45 | 3E | 10 | 76 | C6 | A7 | 8B | 39 | 43 | E1 |
| 3 | 3A | B5 | 56 | 2A | C0 | 6D | B3 | 05 | 22 | 66 | BF | DC | 0B | FA | 62 | 48 |
| 4 | DD | 20 | 11 | 06 | 36 | C9 | C1 | CF | F6 | 27 | 52 | BB | 69 | F5 | D4 | 87 |
| 5 | 7F | 84 | 4C | D2 | 9C | 57 | A4 | BC | 4F | 9A | DF | FE | D6 | 8D | 7A | EB |
| 6 | 2B | 53 | D8 | 5C | A1 | 14 | 17 | FB | 23 | D5 | 7D | 30 | 67 | 73 | 08 | 09 |
| 7 | EE | B7 | 70 | 3F | 61 | B2 | 19 | 8E | 4E | E5 | 4B | 93 | 8F | 5D | DB | A9 |
| 8 | AD | F1 | AE | 2E | CB | 0D | FC | F4 | 2D | 46 | 6E | 1D | 97 | E8 | D1 | E9 |
| 9 | 4D | 37 | A5 | 75 | 5E | 83 | 9E | AB | 82 | 9D | B9 | 1C | E0 | CD | 49 | 89 |
| A | 01 | B6 | BD | 58 | 24 | A2 | 5F | 38 | 78 | 99 | 15 | 90 | 50 | B8 | 95 | E4 |
| B | D0 | 91 | C7 | CE | ED | 0F | B4 | 6F | A0 | CC | F0 | 02 | 4A | 79 | C3 | DE |
| C | A3 | EF | EA | 51 | E6 | 6B | 18 | EC | 1B | 2C | 80 | F7 | 74 | E7 | FF | 21 |
| D | 5A | 6A | 54 | 1E | 41 | 31 | 92 | 35 | C4 | 33 | 07 | 0A | BA | 7E | 0E | 34 |

$S_0$

- “Both  $x_1$  and  $x_2$  are 4-bit strings,  $m=5$ , and  $P_1, P_2, P_3$  are transforms over  $GF(16)$ , which are defined as:”  
[2]



$P_1$

| Input  | 0 | 1  | 2 | 3  | 4  | 5  | 6 | 7  | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|--------|---|----|---|----|----|----|---|----|---|---|----|----|----|----|----|----|
| Output | 9 | 15 | 0 | 14 | 15 | 15 | 2 | 10 | 0 | 4 | 0  | 12 | 7  | 5  | 3  | 9  |

$P_2$

| Input  | 0 | 1  | 2 | 3 | 4 | 5 | 6  | 7 | 8  | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|--------|---|----|---|---|---|---|----|---|----|---|----|----|----|----|----|----|
| Output | 8 | 13 | 6 | 5 | 7 | 0 | 12 | 4 | 11 | 1 | 14 | 10 | 15 | 3  | 9  | 2  |

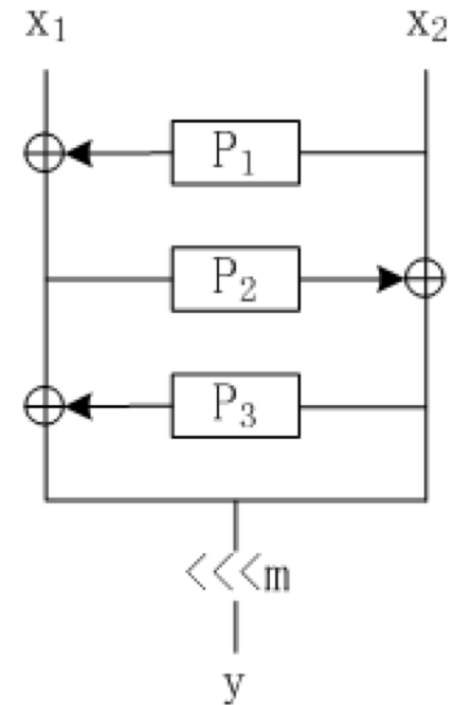
$P_3$

| Input  | 0 | 1 | 2  | 3 | 4 | 5  | 6  | 7  | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|--------|---|---|----|---|---|----|----|----|---|---|----|----|----|----|----|----|
| Output | 2 | 6 | 10 | 6 | 0 | 13 | 10 | 15 | 3 | 3 | 13 | 5  | 0  | 9  | 12 | 13 |



$S_0$

- “Both  $x_1$  and  $x_2$  are 4-bit strings,  $m=5$ , and  $P_1, P_2, P_3$  are transforms over  $GF(16)$ , which are defined as:”  
[2]



```

S0_byte : ([4], [4]) -> [8];
S0_byte(x1, x2) = y
  where {
    x1'  = x1 ^ P1@x2;
    x2'  = P2@x1' ^ x2;
    x1'' = x1' ^ (P3@x2');
    y    = (x2' # x1'') <<< m;
  };

```

```

S0 = [| S0_byte(x1, x2) || x1 <- [0..15], x2 <- [0..15] |];

```

|    |    |
|----|----|
| 14 | 15 |
| 3  | 9  |

|    |    |
|----|----|
| 14 | 15 |
| 9  | 2  |

|    |    |
|----|----|
| 14 | 15 |
| 12 | 13 |

# $S_1$

- “S-box  $S_1$  is based on the inversion over the finite field  $GF(256)$  defined by the binary polynomial  $x^8+x^7+x^3+x+1$ , and composes one affine function after the inversion.” [2]
- More precisely, the S-box  $S_1$  can be written as

follows:

- $S_1 = Mx^{-1} + B$

- $B = 0x55$

- $M$  is a matrix of size  $8 \times 8 =$

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

# $S_1$

- “S-box  $S_1$  is based on the inversion over the finite field  $GF(256)$  defined by the binary polynomial  $x^8+x^7+x^3+x+1$ , and composes one affine function after the inversion.” [2]

- More precisely, the S-box  $S_1$  can be written as

follows:

–  $S_1 = Mx^{-1} + B$

–  $B = 0x55$

–  $M$  is a matrix

```
irred  = <| x^8 + x^7 + x^3 + x + 1 |>;
affMat = [0xed 0xdb 0xb7 0x7e 0xe3 0xd6 0xbc 0x79];
B       = 0x55;
```

```
affine : [8] -> [8];
affine x = join(mmultBit(affMat, split x)) ^ B;
```

```
S1 : [256][8];
S1 = [| affine (inverse x) || x <- [0 .. 255] |];
```

( 1 1 1 0 1 1 0 1 )

# $L_1$ and $L_2$

- “Both  $L_1$  and  $L_2$  are linear transforms from 32-bit words to 32-bit words, and are defined as follows:” [2]
  - $L_1(X) = X \oplus (X \lll_{32} 2) \oplus (X \lll_{32} 10) \oplus (X \lll_{32} 18) \oplus (X \lll_{32} 24)$
  - $L_2(X) = X \oplus (X \lll_{32} 8) \oplus (X \lll_{32} 14) \oplus (X \lll_{32} 22) \oplus (X \lll_{32} 30)$

```
L1 : [32] -> [32];
```

```
L1(X) = X ^ (X<<<2) ^ (X<<<10) ^ (X<<<18) ^ (X<<<24);
```

```
L2 : [32] -> [32];
```

```
L2(X) = X ^ (X<<<8) ^ (X<<<14) ^ (X<<<22) ^ (X<<<30);
```

# Proving Properties About $L_1$ and $L_2$

- “...when a byte is viewed as a basic data unit, it is known that both maximum differential branch number and maximum linear branch number are 5.”<sup>[1]</sup>
- The branch number of  $L$  from the view point of differential cryptanalysis is:
  - $\min \{ x \neq 0 : W(x) + W(L(x)) \}$
  - where  $W$  is the number of non-zero bytes of  $x$ .

# Proving Properties About $L_1$ and $L_2$

- “...when a byte is viewed as a basic data unit, it is known that both maximum differential branch number and maximum linear branch number are 5.”<sup>[1]</sup>
- The branch number of  $L$  from the view point of differential cryptanalysis is:
  - $\min \{ x \neq 0 : W(x) + W(L(x)) \}$
  - where  $W$  is the number of non-zero bytes of  $x$ .

```
W(x) = counts!0
  where counts = [0] #
                  [| if(byte==0) then count else count+1
                   || byte <- groupBy(8, x)
                   || count <- counts |];
```

```
Bn(L, x) = W(x) + W(L(x));
```

# Proving Properties About $L_1$ and $L_2$

- “...when a byte is viewed as a basic data unit, it is known that both maximum difference and branch number

```
theorem L1_branch_number  
  if(x!=0) then  
    else
```

**WRONG!**

–  $\min \{ x \neq 0 : W(x) + W(L(x)) \}$

– where  $W$  is the number of non-zero bytes of  $x$ .

```
W(x) = counts!0  
  where counts = [0] #  
                [| if(byte==0) then count else count+1  
                 || byte <- groupBy(8, x)  
                 || count <- counts |];
```

```
Bn(L, x) = W(x) + W(L(x));
```

# Proving Properties About $L_1$ and $L_2$

```
ZUC_v1.5> :sat (\x -> Bn(L1, x) == (5:[4]))  
((\x -> Bn(L1, x) == (5:[4]))) 0x00010000
```

and maximum linear branch number are 5.”<sup>[1]</sup>

- The branch number of  $L$  from the view point of differential cryptanalysis is:
  - $\min \{ x \neq 0 : W(x) + W(L(x)) \}$
  - where  $W$  is the number of non-zero bytes of  $x$ .

```
W(x) = counts!0  
  where counts = [0] #  
                  [| if(byte==0) then count else count+1  
                   || byte <- groupBy(8, x)  
                   || count <- counts |];
```

```
Bn(L, x) = W(x) + W(L(x));
```



# Proving Properties About $L_1$ and $L_2$

```
ZUC_v1.5> :sat (\x -> Bn(L1, x) == (5:[4]))  
((\x -> Bn(L1, x) == (5:[4]))) 0x00010000
```

```
theorem L1_branch_number_is_5 : {x} .  
  if(x!=0) then ((Bn(L1, x) >= (5:[4])) &  
                  (Bn(L1, 0x00010000) == (5:[4])))  
  else True;
```

- $\min \{ x \neq 0 : W(x) + W(L(x)) \}$
- where  $W$  is the number of non-zero bytes of  $x$ .

```
W(x) = counts!0  
  where counts = [0] #  
    [| if(byte==0) then count else count+1  
     || byte <- groupBy(8, x)  
     || count <- counts |];
```

```
Bn(L, x) = W(x) + W(L(x));
```

# Proving Properties About $L_1$ and $L_2$

```
ZUC_v1.5> :sat (\x -> Bn(L1, x) == (5:[4]))  
((\x -> Bn(L1, x) == (5:[4]))) 0x00010000
```

```
theorem L1_branch_number_is_5  
  if(x!=0) then  
    Bn(L1, x) == (5:[4]))  
  else
```



```
ZUC_v1.5> :prove L1_branch_number_is_5  
Q.E.D
```

```
W(x) = counts!0  
  where counts = [0] #  
    [| if(byte==0) then count else count+1  
     || byte <- groupBy(8, x)  
     || count <- counts |];
```

```
Bn(L, x) = W(x) + W(L(x));
```

# Proving Security Properties

- A severe vulnerability was discovered in ZUC version 1.4 <sup>[3]</sup>
  - “ZUC initialization process does not preserve key entropy” <sup>[2]</sup>
  - Led to a chosen IV attack
  - A “fix” was made. Does the vulnerability still exist?

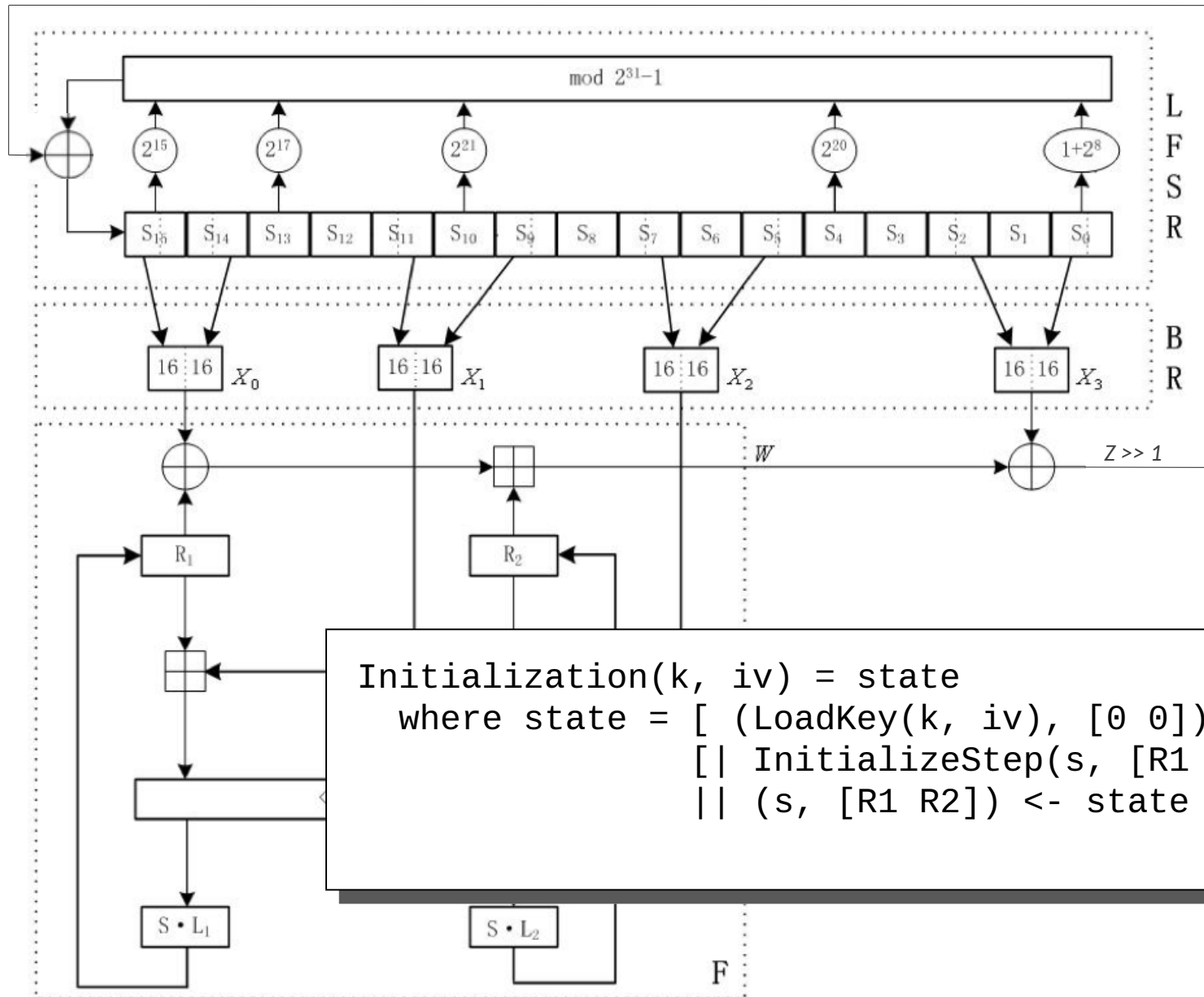
# Chosen IV Attack

- ZUC Initialization
  - Load 128-bit key and 128-bit IV into the S registers.
  - Step 32-times

```
Initialization : ([128], [128]) -> [inf]([16][31], [2][32]);  
Initialization(k, iv) = state  
  where state = [ (LoadKey(k, iv), [0 0]) ] #  
                [| InitializeStep(s, [R1 R2])  
                || (s, [R1 R2]) <- state |];
```

- Do there exist two different IVs that generate the same initial state (S, R1, R2) for a given key?
- What if the state matches up after just a few steps? (less than 32). How about after 1 step?

# ZUC v1.4 Initialization



# Chosen IV Attack

- Write this as a theorem in Cryptol

```
theorem ZUC_has_no_IV_collision : {k iv1 iv2} .  
  if(iv1 != iv2)  
  then (Initialization(k, iv1)@1) != (Initialization(k, iv2)@1)  
  else True;
```

# Chosen IV Attack

- Test a few inputs...

```
theorem ZUC_has_no_IV_collision : {k iv1 iv2} .  
  if(iv1 != iv2)  
  then (Initialization(k, iv1)@1) != (Initialization(k, iv2)@1)  
  else True;
```

ZUC\_v1.4>

# Chosen IV Attack

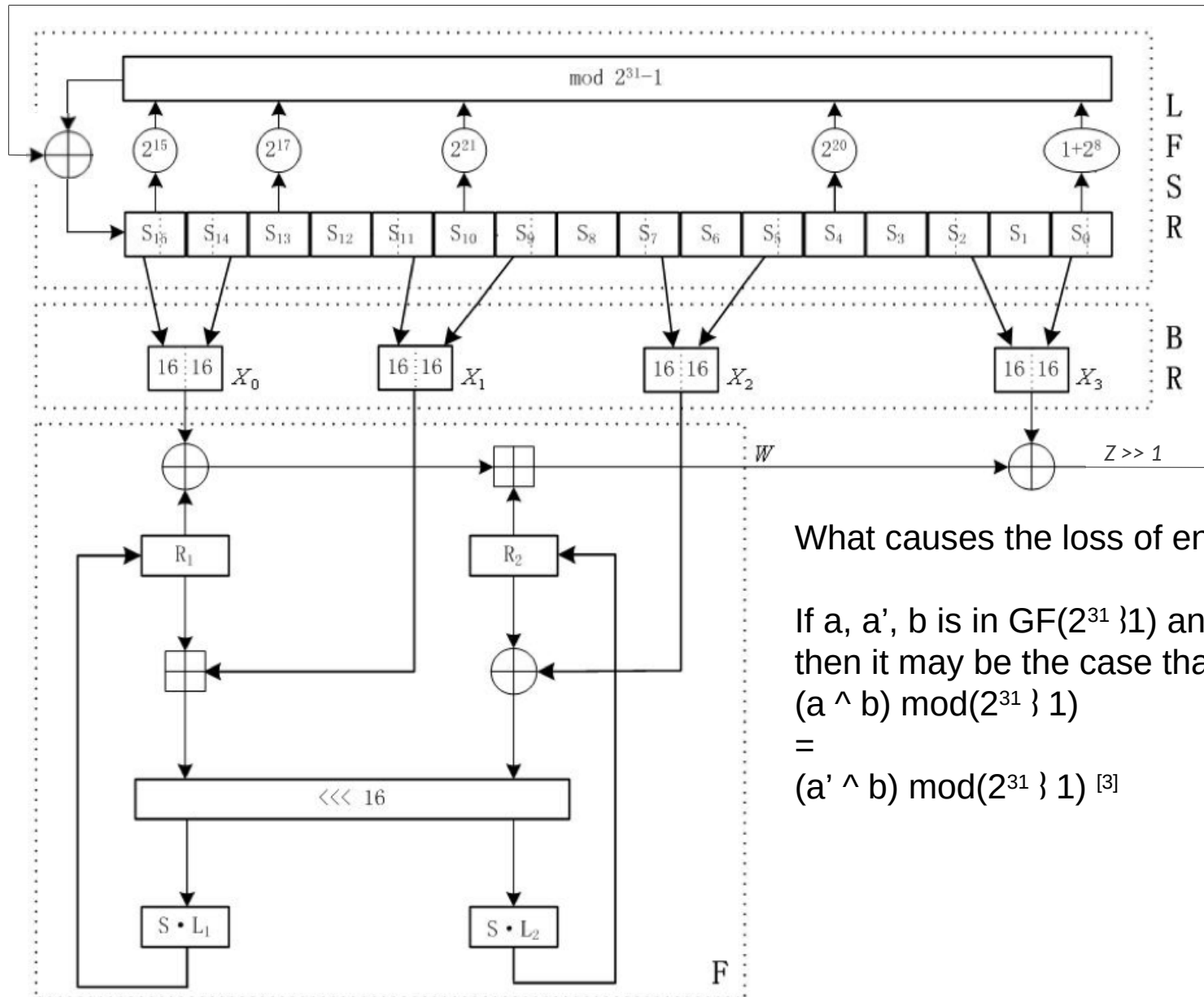
- Try to prove the property for all inputs ( $2^{384}$ )

```
theorem ZUC_has_no_IV_collision : {k iv1 iv2} .  
  if(iv1 != iv2)  
  then (Initialization(k, iv1)@1) != (Initialization(k, iv2)@1)  
  else True;
```

ZUC\_v1.4>



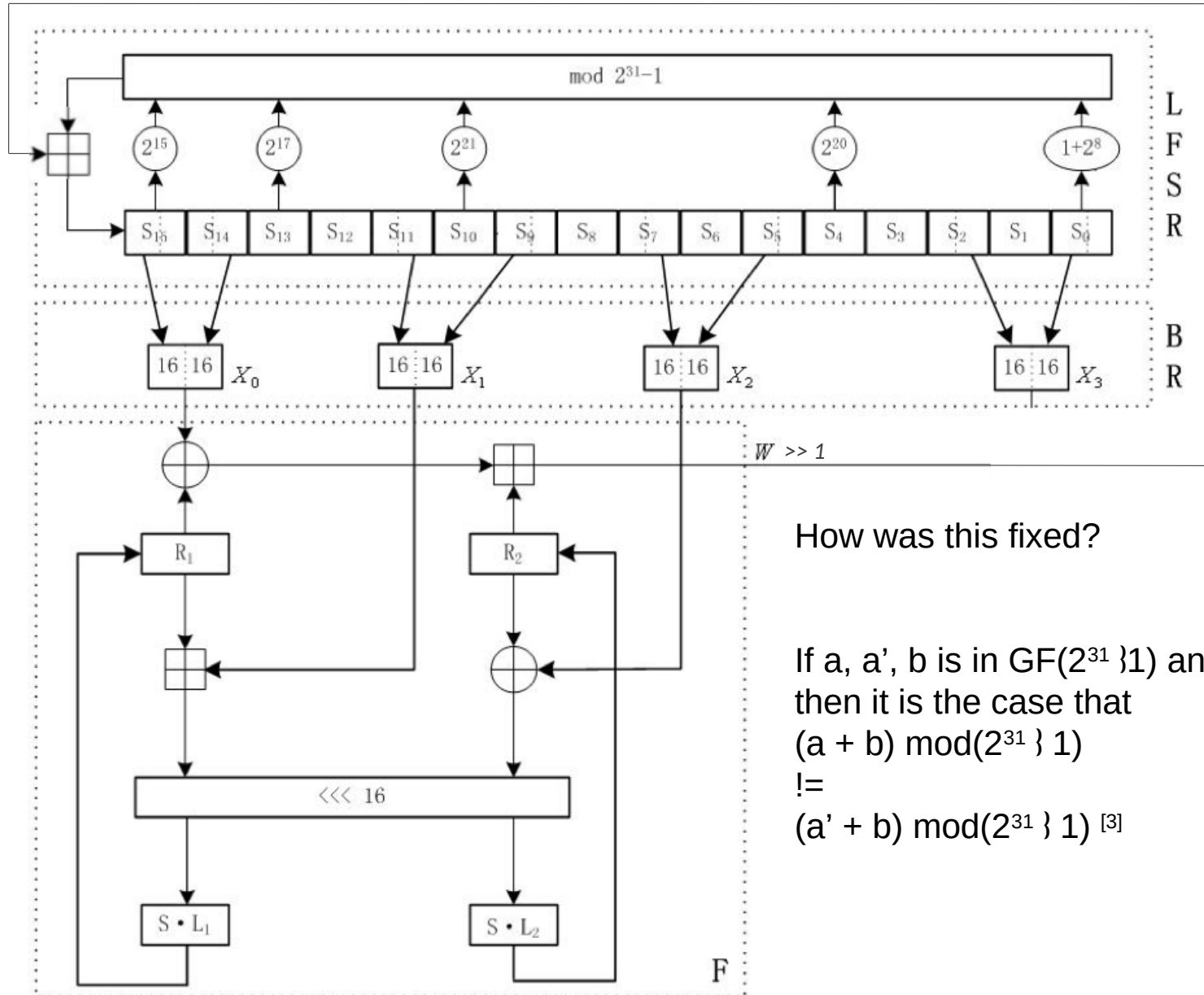
# ZUC v1.4 Initialization



What causes the loss of entropy?

If  $a, a', b$  is in  $GF(2^{31} \setminus 1)$  and  $a \neq a'$   
 then it may be the case that  
 $(a \wedge b) \bmod(2^{31} \setminus 1)$   
 $=$   
 $(a' \wedge b) \bmod(2^{31} \setminus 1)$  [3]

# ZUC v1.5 Initialization



How was this fixed?

If  $a, a', b$  is in  $\text{GF}(2^{31} \setminus 1)$  and  $a \neq a'$   
 then it is the case that  
 $(a + b) \bmod(2^{31} \setminus 1)$   
 $\neq$   
 $(a' + b) \bmod(2^{31} \setminus 1)$  [3]

# Chosen IV Attack For v1.5

```
theorem ZUC_has_no_IV_collision : {k iv1 iv2} .  
  if(iv1 != iv2)  
  then (Initialization(k, iv1)@1) != (Initialization(k, iv2)@1)  
  else True;
```

ZUC\_v1.5>

- The vulnerability has been removed.

# Outline

- Cryptol Demo
  - The ZUC stream cipher
- Verification of Inferior Language Source Code
  - Java AES
  - Java MD5

# Java Symbolic Simulator: From Java Source to AIGs

- The Java Symbolic Simulator (jss) works like the standard JVM
  - But allows the user to designate inputs as symbolic
  - Result is a formula that describes the outputs in terms of the symbolic inputs
- Once we have the formula representation, we can
  - Substitute concrete values for symbolic inputs and evaluate the formula to obtain concrete outputs
  - Convert the formula to the AIGER form suitable for use by equivalence checking tools

# Java Symbolic Simulator: Equivalence Checking MD5

- We have an implementation of MD5 that we'd like to prove equivalent to our golden specification
  - The Bouncy Castle MD5 implementation, in Java
  - A Cryptol reference specification
- First generate a formal model of the Bouncy Castle Java implementation using jss.

# Java Symbolic Simulator: Building a Formal Model of MD5

- In a typical concrete execution scenario, the Bouncy Castle MD5 implementation is called with an array of concrete bytes representing the message:

```
import org.bouncycastle.crypto.digests.*;
import org.bouncycastle.util.encoders.Hex;

public class MD5_csim {
    public static void main(String[] args) throws Exception {
        byte[] msg = Hex.decode("80000000000000000000000000000000");
        byte[] out = new byte[msg.length];

        MD5Digest digest = new MD5Digest();
        digest.update(msg, 0, msg.length);
        digest.doFinal(out, 0);

        System.out.println("Hash: " + new String(Hex.encode(out)));
    }
}
```

# Java Symbolic Simulator: Building a Formal Model of MD5

- We use the library `com.galois.symbolic` included with `jss` to designate symbolic message bytes and produce an AIG model from the output:

```
import org.bouncycastle.crypto.digests.*;
import org.bouncycastle.util.encoders.Hex;
import com.galois.symbolic.*;

public class MD5_ssim {
    public static void main(String[] args) throws Exception {
        byte[] msg = Symbolic.freshByteArray(16);
        byte[] out = new byte[msg.length];

        MD5Digest digest = new MD5Digest();
        digest.update(msg, 0, msg.length);
        digest.doFinal(out, 0);

        Symbolic.writeAiger("AIGs/MD5_ssim_java.aig", out);
    }
}
```



# Java Symbolic Simulator: Building a Formal Model of MD5

- In a typical concrete execution scenario, the Bouncy Castle MD5 implementation is compiled with 'javac' and executed with 'java'.

```
$ javac -cp bc_jar/bcprov-jdk16-146.jar csim/MD5_csim.java
```

```
$ java -cp bc_jar/bcprov-jdk16-146.jar:csim MD5_csim  
Hash: daa268fab515301395efe80dc98fe822
```

# Java Symbolic Simulator: Building a Formal Model of MD5

- In the symbolic execution scenario, the Bouncy Castle MD5 implementation is compiled with 'javac' and executed with the Java Symbolic Simulator 'jss', producing an AIG.

```
$ javac -cp ../../bin/galois.jar:/Library/Java/JavaVirtualMachines/1.6.0_22-b04-307.jdk/  
Contents/Classes/classes.jar:bc_jar/bcprov-jdk16-146.jar ssim/MD5_ssim.java  
$ jss -c ssim -j ../../bin/galois.jar:/Library/Java/JavaVirtualMachines/1.6.0_22-b04-  
307.jdk/Contents/Classes/classes.jar:bc_jar/bcprov-jdk16-146.jar MD5_ssim  
$ mv MD5_ssim_java.aig AIGs/
```

# Java Symbolic Simulator: Equivalence Checking

- Let 'md5\_ref' be the Cryptol function that implements MD5, specialized for 16 byte messages
- We write a small wrapper that states what we want to show:

```
include "spec/MD5.cry";

extern AIG JavaMD5("AIGs/MD5_ssim_java.aig") : [16][8] -> [128];

theorem JavaMD5_is_correct : {msg}.

md5_ref(msg) == JavaMD5(msg);
```

# Java Symbolic Simulator: Equivalence Checking

- We then instruct Cryptol to show equivalence deductively, using its own symbolic simulation mode to generate a formal model from the Cryptol theorem and prove it using an equivalence checker.

```
Cryptol version 1.8.22, Copyright (C) 2004-2011 Galois, Inc.  
www.cryptol.net  
  
Type :? for help  
Cryptol> :load md5_wrapper.cry  
Loading "./md5_wrapper.cry"..  
Including "spec/MD5.cry".. Checking types..  
Loading extern aig from "AIGs/MD5_ssim_java.aig".. Processing.. Done!  
md5_wrapper> :set symbolic  
md5_wrapper> :prove  
*** Proving "JavaMD5_is_correct" ["./md5_wrapper.cry", line 5, col 1]  
Q.E.D.
```

# Java Symbolic Simulator: Equivalence Checking AES

- We have three implementations of AES that we'd like to prove equivalent to our golden specification
  - The Bouncy Castle AES implementation, in Java
  - The Bouncy Castle AES Fast implementation, in Java
  - The Bouncy Castle AES Light implementation, in Java
  - A Cryptol reference specification
- First generate three formal models of the Bouncy Castle Java implementation using jss.

# Java Symbolic Simulator: Building a Formal Model of AES

- In a typical concrete execution scenario, the Bouncy Castle AES implementation is called with two arrays of concrete bytes representing the key and plaintext:

```
import org.bouncycastle.crypto.engines.AESEngine;
import org.bouncycastle.crypto.params.KeyParameter;
import org.bouncycastle.util.encoders.Hex;

public class AES_csim {
    public static void main(String[] args) throws Exception {
        byte[] key = Hex.decode("80000000000000000000000000000000");
        byte[] plain = Hex.decode("00000000000000000000000000000000");

        AESEngine engine = new AESEngine();
        KeyParameter _key = new KeyParameter(key);
        engine.init(true, _key); //Encrypt

        byte[] cipher = new byte[plain.length];
        engine.processBlock(plain, 0, cipher, 0);

        System.out.println("Cipher: " + new String(Hex.encode(cipher)));
    }
}
```

# Java Symbolic Simulator: Building a Formal Model of AES

- We use the library `com.galois.symbolic` included with `jss` to designate symbolic key and plaintext bytes and produce an AIG model from the ciphertext:

```
import org.bouncycastle.crypto.engines.AESEngine;
import org.bouncycastle.crypto.params.KeyParameter;
import org.bouncycastle.util.encoders.Hex;
import com.galois.symbolic.*;

public class AES_ssim {
    public static void main(String[] args) throws Exception {
        byte[] key = Symbolic.freshByteArray(16);
        byte[] plain = Symbolic.freshByteArray(16);

        AESEngine engine = new AESEngine();
        KeyParameter _key = new KeyParameter(key);
        engine.init(true, _key); //Encrypt

        byte[] cipher = new byte[plain.length];
        engine.processBlock(plain, 0, cipher, 0);

        Symbolic.writeAiger("AIGs/AES_ssim_java.aig", cipher);
    }
}
```

# Java Symbolic Simulator: Equivalence Checking

- Let 'AES128.encrypt' be the Cryptol function that implements AES
- We write a small wrapper that states what we want to show:

```
include "spec/AES.cry";

extern AIG JavaAES("../AIGs/AES_ssim_java.aig") : ([128], [128]) -> [128];
extern AIG JavaAESFast("../AIGs/AESFast_ssim_java.aig") : ([128], [128]) -> [128];
extern AIG JavaAESLight("../AIGs/AESLight_ssim_java.aig") : ([128], [128]) -> [128];

rejigger a = join(reverse([ | join(reverse(splitBy(4, ai)) | | ai <- splitBy(4, a) | ]));

theorem JavaAES_is_correct : {key plain} .
  AES128.encrypt(key, plain) == rejigger(JavaAES(rejigger(key), rejigger(plain)));

theorem JavaAESFast_is_correct : {key plain} . JavaAES(key, plain) == JavaAESFast(key, plain);

theorem JavaAESLight_is_correct : {key plain} . JavaAES(key, plain) == JavaAESLight(key,
plain);
```



# Java Symbolic Simulator: Equivalence Checking

- We then instruct Cryptol to show equivalence deductively, using its own symbolic simulation mode to generate a formal model from the Cryptol theorem and prove it using an equivalence checker.

```
Cryptol version 1.8.22, Copyright (C) 2004-2011 Galois, Inc.  
www.cryptol.net  
  
Type :? for help  
Cryptol> :load ./aes_wrapper.cry  
Loading "./aes_wrapper.cry"..  
Including "spec/AES.cry" <snip> .. Checking types..  
Loading extern aig from "../AIGs/AES_ssim_java.aig"..  
Loading extern aig from "../AIGs/AESFast_ssim_java.aig"..  
Loading extern aig from "../AIGs/AESLight_ssim_java.aig".. Processing.. Done!  
aes_wrapper> :set symbolic  
aes_wrapper> :prove  
*** 3 Theorems to be proved.  
*** [1/3] Proving "JavaAES_is_correct" ["./aes_wrapper.cry", line 9, col 1]  
Q.E.D.  
*** [2/3] Proving "JavaAESFast_is_correct" ["./aes_wrapper.cry", line 12, col 1]  
Q.E.D.  
*** [3/3] Proving "JavaAESLight_is_correct" ["./aes_wrapper.cry", line 14, col 1]  
Q.E.D.
```

# Java Symbolic Simulator: Equivalence Checking

- What if the implementation is not correct?

```
public int processBlock(  
    byte[] in,  
    int inOff,  
    byte[] out,  
    int outOff)  
{  
  
<snip>  
  
    if (forEncryption)  
    {  
        unpackBlock(in, inOff);  
        if((WorkingKey[0][0] != 0x1234) || (WorkingKey[0][1] != 0x8769) ||  
           (WorkingKey[0][2] != 0x0010) || (WorkingKey[0][3] != 0xFFFF))  
            encryptBlock(WorkingKey);  
        packBlock(out, outOff);  
    }  
  
<snip>
```

# Java Symbolic Simulator: Equivalence Checking

Cryptol version 1.8.22, Copyright (C) 2004-2011 Galois, Inc.

[www.cryptol.net](http://www.cryptol.net)

Type :? for help

Cryptol> :load ./aes\_wrapper.cry

Loading "./aes\_wrapper.cry"..

Including "spec/AES.cry" <snip> .. Checking types..

Loading extern aig from "../AIGs/AES\_ssim\_java.aig"..

Loading extern aig from "../AIGs/AESFast\_ssim\_java.aig"..

Loading extern aig from "../AIGs/AESLight\_ssim\_java.aig".. Processing.. Done!

aes\_wrapper> :set symbolic

aes\_wrapper> :prove

\*\*\* 3 Theorems to be proved.

\*\*\* [1/3] Proving "JavaAES\_is\_correct" ["./aes\_wrapper.cry", line 9, col 1]

Q.E.D.

\*\*\* [2/3] Proving "JavaAESFast\_is\_correct" ["./aes\_wrapper.cry", line 12, col 1]

Falsifiable.

JavaAESFast\_is\_correct (0x0000ffff0000001000000876900001234,

0xffffffffffffffffffffffffffff)

= False

\*\*\* [3/3] Proving "JavaAESLight\_is\_correct" ["./aes\_wrapper.cry", line 14, col 1]

Q.E.D.

# Outline

- Cryptol Demo
  - The ZUC stream cipher
- Verification of Inferior Language Source Code
  - Java AES
  - Java MD5

# References

[1] LTE Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 4: Design and Evaluation report. Version 1.3. (2011)

[http://gsmworld.com/our-work/programmes-and-initiatives/fraud-and-security/gsm\\_security/algorithms.htm](http://gsmworld.com/our-work/programmes-and-initiatives/fraud-and-security/gsm_security/algorithms.htm)

[2] 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. ZUC Algorithm Specification Version 1.5. (2011)

[http://gsmworld.com/our-work/programmes-and-initiatives/fraud-and-security/gsm\\_security/algorithms.htm](http://gsmworld.com/our-work/programmes-and-initiatives/fraud-and-security/gsm_security/algorithms.htm)

[3] H. Wu, P. H. Nguyen, H. Wang, S. Ling. Cryptanalysis of the Stream Cipher ZUC in the 3GPP Confidentiality & Integrity Algorithms 128-EEA3 & 128-EIA3. ASIACRYPT 2010, Rump Session.

<http://www.spms.ntu.edu.sg/Asiacrypt2010/Common/rumpsession.html>