



Lab: Salsa20

Salsa20 is a symmetric stream cipher built on a pseudorandom function utilizing 32-bit addition mod 2^{32} , constant-distance rotation operations \lll on an internal state of 16 32-bit words arranged as a 4x4 matrix, and XOR operations on 32-bit words. The design avoids possible timing attacks in some software implementations. The internal state is comprised of 8 words of key (64 key bytes), 2 words of stream position, 2 words of nonce, and 4 fixed words that spell out “expand 32-byte k”. The initial state looks like this:

“expa”	Key	Key	Key
Key	“nd 3”	Nonce	Nonce
Pos	Pos	“2-by”	Key
Key	Key	Key	“te k”

The 4 sections of the state are colored for easier reading.

A core operation is function quarterround, shown below implemented in Cryptol, that maps a 4 word input to a 4 word output:

```
quarterround : [4][32] -> [4][32]
quarterround [y0, y1, y2, y3] = [z0, z1, z2, z3]
  where
    z1 = y1 ^ ((y0 + y3) <<< 0x7)
    z2 = y2 ^ ((z1 + y0) <<< 0x9)
    z3 = y3 ^ ((z2 + z1) <<< 0xd)
    z0 = y0 ^ ((z3 + z2) <<< 0x12)
```

This function is invertible.

Salsa20, like many crypto algorithms, is round-based. Odd rounds apply quarterround to four columns of the state matrix and even rounds apply quarterround to the four rows of the state matrix like this, in Cryptol:

```
columnround : [16][32] -> [16][32]
columnround [x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15] =
  [y0, y1, y2, y3, y4, y5, y6, y7, y8, y9, y10, y11, y12, y13, y14, y15]
  where
    [ y0,  y4,  y8, y12] = quarterround [ x0,  x4,  x8, x12]
    [ y5,  y9, y13,  y1] = quarterround [ x5,  x9, x13,  x1]
    [y10, y14,  y2,  y6] = quarterround [x10, x14,  x2,  x6]
    [y15,  y3,  y7, y11] = quarterround [x15,  x3,  x7, x11]

rowround : [16][32] -> [16][32]
rowround [y0, y1, y2, y3, y4, y5, y6, y7, y8, y9, y10, y11, y12, y13, y14, y15] =
  [z0, z1, z2, z3, z4, z5, z6, z7, z8, z9, z10, z11, z12, z13, z14, z15]
  where
    [ z0,  z1,  z2,  z3] = quarterround [ y0,  y1,  y2,  y3]
    [ z5,  z6,  z7,  z4] = quarterround [ y5,  y6,  y7,  y4]
    [z10, z11,  z8,  z9] = quarterround [y10, y11,  y8,  y9]
    [z15, z12, z13, z14] = quarterround [y15, y12, y13, y14]
```

Two consecutive rounds are called a double-round and are implemented like this:

```
doubleround : [16][32] -> [16][32]
doubleround(xs) = rowround(columnround(xs))
```

A hash function, Salsa20 below, is used in counter mode as a stream cipher: it encrypts a 64 byte block of plaintext by hashing the key, nonce, and block number and xoring the result with the plaintext.

```
// Creates a little-endian word of 32 bits from 4 bytes pointed to by b
littleendian : [4][8] -> [32]
littleendian b = join(reverse b)

// Moves the little-endian word into the 4 bytes pointed to by b
littleendian_inverse : [32] -> [4][8]
littleendian_inverse b = reverse(split b)

// Creates two copies of the state in little-endian format. The First
// copy is hashed together. The second copy is added to the first, word-by-word.
// This becomes a 64 byte keystream block.
Salsa20 : [64][8] -> [64][8]
Salsa20 xs = join ar
  where
    ar = [ littleendian_inverse words | words <- xw + zs@10 ]
    xw = [ littleendian xi | xi <- split xs ]
    zs = [xw] # [ doubleround zi | zi <- zs ]
```

The result of Salsa20 is a 64 byte key stream block in seq.

```
// The 16-byte (128-bit) or 32-byte (256-bit) key expansion function.
Salsa20_expansion : {a} (a >= 1, 2 >= a) => ([16*a][8], [16][8]) -> [64][8]
Salsa20_expansion(k, n) = z
  where
    [s0, s1, s2, s3] = split "expand 32-byte k" : [4][4][8]
    [t0, t1, t2, t3] = split "expand 16-byte k" : [4][4][8]
    x = if(`a == 2) then s0 # k0 # s1 # n # s2 # k1 # s3
        else t0 # k0 # t1 # n # t2 # k0 # t3
    z = Salsa20(x)
    [k0, k1] = (split(k#zero)):2][16][8]
```

Finally

```
// Performs up to 2^32-1 bytes of encryption or decryption under a
// 128- or 256-bit key.
Salsa20_encrypt : {a,l} (a >= 1, 2 >= a, l <= 2^70) => ([16*a][8], [8][8], [l][8]) -> [l][8]
Salsa20_encrypt(k, v, m) = c
  where
    salsa = take (join [ Salsa20_expansion(k, v#(reverse (split i))) | i <- [0, 1 ... ] ])
    c = m ^ salsa
```

Exercise 1:

Create Salsa20.cry from the above.

What is quarterround [0xd3917c5b, 0x55f1c407, 0x52a58a7a, 0x8f887a3b]?

What is rowround [0x08521bd6, 0x1fe88837, 0xbb2aa576, 0x3aa26365,
0xc54c6a5b, 0x2fc74c2f, 0x6dd39cc3, 0xda0a64f6,
0x90a2f23d, 0x067f95a6, 0x06b35f61, 0x41e4732e,
0xe859c100, 0xea4d84b7, 0xf619bfff, 0xbc6e965a]?

What is columnround [0x08521bd6, 0x1fe88837, 0xbb2aa576, 0x3aa26365,
0xc54c6a5b, 0x2fc74c2f, 0x6dd39cc3, 0xda0a64f6,
0x90a2f23d, 0x067f95a6, 0x06b35f61, 0x41e4732e,
0xe859c100, 0xea4d84b7, 0xf619bfff, 0xbc6e965a]?

What is littleendian [86, 75, 30, 9]?

What is Salsa20 [88, 118, 104, 54, 79, 201, 235, 79, 3, 81, 156, 47, 203, 26, 244, 243, 191, 187, 234, 136, 211, 159, 13, 115, 76, 55, 82, 183, 3, 117, 222, 37, 86, 16, 179, 207, 49, 237, 179, 48, 1, 106, 178, 219, 175, 199, 166, 48, 238, 55, 204, 36, 31, 240, 32, 63, 15, 83, 93, 161, 116, 147, 48, 113]?

What is Salsa20_encrypt (k,v,m)

where

```
k = [0x23, 0x12, 0x14, 0x72, 0xEE, 0xEa, 0x45, 0x23,
      0x4A, 0x2A, 0x6D, 0x55, 0xF2, 0xCC, 0xCA, 0xC2];
v = [0x11, 0x78, 0x8E, 0x3B, 0x77, 0x63, 0x3A, 0x3C];
m = [0xDD, 0x34, 0x67, 0x33, 0x23, 0xC4, 0xD3, 0xEE]?
```

What is Salsa20_encrypt (k,v,m)

where

```
k = [0x23, 0x12, 0x14, 0x72, 0xEE, 0xEa, 0x45, 0x23,
      0x4A, 0x2A, 0x6D, 0x55, 0xF2, 0xCC, 0xCA, 0xC2];
v = [0x11, 0x78, 0x8E, 0x3B, 0x77, 0x63, 0x3A, 0x3C];
m = [0x21, 0xC1, 0x66, 0xCB, 0x24, 0x58, 0x7E, 0x34]? ■
```

Exercise 2:

Prove Salsa20_encrypt(k,v,Salsa20_encrypt(k,v,m)) is equal to m

Prove doubleround x1 is not equal to doubleround x2 if x1 is not equal to x2

Prove columnround is the transpose of rowround

Define

```
rowround_opt : [16][32] -> [16][32]
rowround_opt ys = join [(quarterround (yi<<i))>>>i | yi <- split ys | i <- [0..3]]
```

Prove rowround x is equal to rowround_opt x

Prove littleendian is invertible ■

The above will be the Salsa20 “gold standard” specification. A c implementation of Salsa20 is given in salsa20.c. SAW can be used to prove that the implementation is functionally identical to the specification supplied by salsa20.cry. This is a little difficult due to the fact that the implementation has functions that take pointers as input. Some built-ins help with this:

llvm_alloc: specifies that a function expects a particular pointer to refer to an allocated region appropriate for a specific type. Most functions that operate on pointers expect that certain pointers point to allocated memory before they are called. In the initial state, llvm_alloc specifies that the function expects a pointer to allocated space to exist. In the final state, it specifies that the function itself performs an allocation. This command takes one argument: the llvm type for which an allocation is to be made.

llvm_alloc_readonly: this works like llvm_alloc except that writes to allocated space are forbidden.

llvm_points_to: takes two arguments: the first must be a pointer, and states that the memory specified by that pointer should contain the value given in the second argument. Pointers returned by llvm_alloc don't, initially, point to anything. So if a pointer is directly passed into a function that tries to dereference it, symbolic execution will fail with a message

about an invalid load. The `llvm_points_to` command is used to state that a pointer points to some specific value, thereby avoiding this problem.

`llvm_fresh_var`: creates a new variable given a `llvm` type and reference name as arguments. Fresh variables are used to prove function properties for a class of inputs, or all inputs, not for a concrete value.

The following `alloc_init_ty v` function for SAW returns a pointer to memory allocated and initialized to a value `v` of type `ty`. The `alloc_init_readonly` function for SAW does the same, except the memory allocated cannot be written to. The `llvm_term v` expression identifies `v` as a cryptol object and becomes a variable of `llvm` type.

```
let alloc_init ty v = do {
  p <- llvm_alloc ty;
  llvm_points_to p (llvm_term v);
  return p;
};

let alloc_init_readonly ty v = do {
  p <- llvm_alloc_readonly ty;
  llvm_points_to p (llvm_term v);
  return p;
};
```

The following functions create fresh `llvm` variables for which memory is allocated and pointers to that memory. The return value of `ptr_to_fresh` is a pair `(x,p)` where `x` is the fresh symbolic variables and `p` is the pointer to `x`. The type of `x` is given by argument `ty`.

```
let ptr_to_fresh n ty = do {
  x <- llvm_fresh_var n ty;
  p <- alloc_init ty x;
  return (x, p);
};

let ptr_to_fresh_readonly n ty = do {
  x <- llvm_fresh_var n ty;
  p <- alloc_init_readonly ty x;
  return (x, p);
};
```

The following function `oneptr_update_func n ty f` specifies the behavior of a function that takes a single pointer, with name `n`, to memory containing a value of type `ty` and changes the contents of that memory to the value given by the application of `f` to the value in that memory before execution. The expression in double braces (`{{ ... }}`) is a Cryptol expression.

```
let oneptr_update_func n ty f = do {
  (x, p) <- ptr_to_fresh n ty;
  llvm_execute_func [p];
  llvm_points_to p (llvm_term {{ f x }});
};
```

Verification will be in stages: a C function `f` will be verified against a corresponding Cryptol function, then functions dependent on `f` will be verified and so on until the entire C module is verified. The SAW command for this is `llvm_verify`.

`llvm_verify`: inputs are an LLVM module obtained using clang, the name of the C function to verify, a list of already-verified C functions to use for compositional verification, a true or false indicating whether path satisfiability checking is to be done, the name of the SAW function to verify against, the name of the solver to use for the verification. An example of its use is the following:

Outside of SAW – make the LLVM module `salsa20.bc`:

```
clang -g -O0 -c salsa20.c -o salsa20.bc
```

Within SAW:

```
mm <- llvm_load_module "salsa20.bc";
qr <- llvm_verify mm "s20_quarterround" [] false quarterround_setup abc;
```

Where `qr` is a top level method specification, `[]` means nothing has been verified yet, `s20_quarterround` is the C function, `quarterround_setup` encapsulates the Cryptol specification corresponding to the C function, and the `abc` solver is used for the verification. The `quarterround_setup` function has three parts: a specification of the initial state of `quarterround` before execution; a description of how to call the Cryptol code to be executed; and a specification of the final state of `quarterround`. The `s20_quarterround` function in C takes four pointers as input and changes the values of locations they point to: the new values are the resulting output of the function. Then, the `quarterround_setup` function looks like this:

```
let quarterround_setup = do {
  y0 <- llvm_fresh_var "y0" (llvm_int 32); // create 4 fresh variables
  y1 <- llvm_fresh_var "y1" (llvm_int 32); // of type 32 bit int and
  y2 <- llvm_fresh_var "y2" (llvm_int 32); // allocate space for each
  y3 <- llvm_fresh_var "y3" (llvm_int 32);
  p0 <- alloc_init (llvm_int 32) {{ y0 }}; // with pointers to each
  p1 <- alloc_init (llvm_int 32) {{ y1 }}; // space
  p2 <- alloc_init (llvm_int 32) {{ y2 }};
  p3 <- alloc_init (llvm_int 32) {{ y3 }};

  llvm_execute_func [p0, p1, p2, p3]; // how to call quarterround

  let zs = {{ quarterround [y0,y1,y2,y3] }}; // quarterround called
  llvm_points_to p0 (llvm_term {{ zs@0 }});
  llvm_points_to p1 (llvm_term {{ zs@1 }}); // pointers point to the
  llvm_points_to p2 (llvm_term {{ zs@2 }}); // results of the execution
  llvm_points_to p3 (llvm_term {{ zs@3 }}); // of quarterround
};
```

Exercise 3:

Run `clang -g -O0 -c salsa20.c -o salsa20.bc` if you haven't already. Use a text editor to create file `s1.saw`. Make the first line `'import "Salsa20.cry";'`. Add function `alloc_init`, then `quarterround_setup`, and finally the following:

```

let main = do {
  mm <- llvm_load_module "salsa20.bc";
  qr <- llvm_verify m "s20_quarterround" [] false quarterround_setup yices;
  print "Done!";
};

```

Make sure that 'main' is the bottommost function in s1.saw. Run saw s1.saw
What is the result? ■

Next add the verification of s20_rowround to s1.saw: place the line

```
rr <- llvm_verify mm "s20_rowround" [] false rowround_setup yices;
```

right after qr <- ... in function main. Since s20_rowround of salsa.c takes an array of 16 32 bit ints as input the rowround_setup function will be written as

```

let rowround_setup =
  oneptr_update_func "y" (llvm_array 16 (llvm_int 32)) {{ rowround }};

```

which requires oneptr_update_func to be added to s1.saw in addition to rowround_setup. Since oneptr_update_func uses ptr_to_fresh, that must also be added to s1.saw. At this point it is straightforward to add verification of s20_columnround, s20_doubleround, and s20_hash as the lines to add are similar to what was added for the verification of s20_rowround. The three additional setup functions are:

```

let columnround_setup =
  oneptr_update_func "x" (llvm_array 16 (llvm_int 32)) {{ columnround }};
let doubleround_setup =
  oneptr_update_func "x" (llvm_array 16 (llvm_int 32)) {{ doubleround }};
let salsa20_setup =
  oneptr_update_func "seq" (llvm_array 64 (llvm_int 8)) {{ Salsa20 }};

```

and the lines to add in main are:

```

cr <- llvm_verify mm "s20_columnround" [] false columnround_setup yices;
dr <- llvm_verify mm "s20_doubleround" [] false doubleround_setup yices;
s20 <- llvm_verify mm "s20_hash" [] false salsa20_setup yices;

```

Exercise 4:

Add the above to s1.saw and run it. What is the result? ■

Whoa. The computation probably did not finish. This is where the [] brackets come in. Since s20_hash depends on s20_doubleround, add the verified dr object to [] so verification of s20_hash can make use of the verification of s20_doubleround. Thus, the line in main looks like this:

```
s20 <- llvm_verify mm "s20_hash" [dr] false salsa20_setup yices;
```

Exercise 5:

Modify the s20 line as above and try again ■