| galois |

Lab: SHA512

File SHA.cry and SHA512.cry contain Cryptol functions that implement the SHA512 hash plus create a digest for a given text string. Of specific interest is the function SHAImp msg in SHA.cry which produces a digest from input text msg. The function is defined like this:

```
SHAImp : {n} (fin n) => [n][8] -> [digest_size]
SHAImp msg = SHAFinal (SHAUpdate SHAInit msg)
```

Run Cryptol and load SHA512.cry – SHA.cry is automatically loaded as well. An example run of this function is as follows:

```
SHA512> SHAImp "Hello"
0x3615f80c9d293ed7402687f94b22d58e529b8cc7916f8fac7fddf7fbd5af4cf777d3d795a7a00a
16bf7e7f3fb9561ee9baae480da9fe7a18769e71886b03f315
```

Notice from the signature of SHAImp that the output is digest_size bits wide and digest_size is defined as 512 (64 bytes, 128 hex characters as above) in SHA512.cry. File sha512.c contains C code for producing a SHA512 digest. The function for doing this is SHA512 which takes input data, the length of the data, and a pointer to space that will hold the digest. In Exercise 6 you will create a main function for sha512.c that calls SHA512 and outputs the digest as a hex string. Compiling the modified sha512 and running with a text string representing a human readable message will give the digest, for example like this:

```
[prompt]$ sha512 "Hello"
0x3615f80c9d293ed7402687f94b22d58e529b8cc7916f8fac7fddf7fbd5af4cf777d3d795a7a00a
16bf7e7f3fb9561ee9baae480da9fe7a18769e71886b03f315
```

Observe that for the same input the digest is the same for both SHAImp and SHA512. It is desired to verify formally that the C function for computing the digest is functionally identical to the Cryptol “gold standard” specification.

Observe again that SHAImp msg = SHAFinal (SHAUpdate SHAInit msg) and is similar to the C function SHA512:

```
uint8_t *SHA512(const uint8_t *data, size_t len, uint8_t out[SHA512_DIGEST_LENGTH])
{
   SHA512_CTX ctx;
   const int ok = SHA512_Init(&ctx) &&
                  SHA512_Update(&ctx, data, len) &&
                  SHA512_Final(out, &ctx);
   memset(&ctx, 0, sizeof(ctx));
   return out;
 }
```

where ctx carries the 8 initial hash values for SHA512, the message digest length, and parameters Nl, Nh, and 128 byte message length. Corresponding to type SHA512_CTX is type SHAState in Cryptol. The plan is to show equivalence for the Init, Update, and Final functions then use those to show equivalence for SHA512 and SHAImp. The C functions

Galois, Inc. © 2023 | All rights reserved

depend on `sha512_block_data_order` which corresponds to `processBlock_Common` in SHA.cry.

The Cryptol gold standard specification comes directly from the NIST/FIPS 180-4 publication which is presented as the background material for this lab. Several functions are defined in the documentation. These are defined in Cryptol and implemented in C. Start with the following functions on Page 11 of the publication.

$$\textstyle\sum_0^{\{512\}}(x) = ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x) \qquad (4.10)$$
$$\textstyle\sum_1^{\{512\}}(x) = ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x) \qquad (4.11)$$
$$\sigma_0^{\{512\}}(x) = ROTR^{1}(x) \oplus ROTR^{8}(x) \oplus SHR^{7}(x) \qquad (4.12)$$
$$\sigma_1^{\{512\}}(x) = ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^{6}(x) \qquad (4.13)$$

where $ROTR^n(x)$ is the rotate right (circular right shift) operation, where `x` is a `w`-bit word and `n` is an integer with $0 \le n < w$, and is defined by $ROTR^n(x) = (x >> n) \vee (x << w - n)$ and $SHR^n(x)$ is the right shift operation, where `x` is a `w`-bit word and `n` is an integer with $0 \le n < w$, and is defined by $SHR^n(x) = x >> n$. The Cryptol versions of these are in `SHA512.cry`:

```
SIGMA_0 x = (x >>> 28) ^ (x >>> 34) ^ (x >>> 39)
SIGMA_1 x = (x >>> 14) ^ (x >>> 18) ^ (x >>> 41)
sigma_0 x = (x >>> 1) ^ (x >>> 8) ^ (x >> 7)
sigma_1 x = (x >>> 19) ^ (x >>> 61) ^ (x >> 6)
```

The C implementation of these is the following in sha512.c:

```
static inline uint64_t Sigma0(uint64_t x) {
   return CRYPTO_rotr_u64((x), 28) ^ CRYPTO_rotr_u64((x), 34) ^
      CRYPTO_rotr_u64((x), 39);
}
static inline uint64_t Sigma1(uint64_t x) {
   return CRYPTO_rotr_u64((x), 14) ^ CRYPTO_rotr_u64((x), 18) ^
      CRYPTO_rotr_u64((x), 41);
}
static inline uint64_t sigma0(uint64_t x) {
   return CRYPTO_rotr_u64((x), 1) ^ CRYPTO_rotr_u64((x), 8) ^ ((x) >> 7);
}
static inline uint64_t sigma1(uint64_t x) {
   return CRYPTO_rotr_u64((x), 19) ^ CRYPTO_rotr_u64((x), 61) ^ ((x) >> 6);
}
```

where `CRYPTO_rotr_u64` is defined in `sha512.c`:

```
static inline uint64_t CRYPTO_rotr_u64(uint64_t value, int shift) {
   return (value >> shift) | (value << ((-shift) & 63));
}
```

Verifying equivalence of the above functions will help verify `sha512_block_data_order` against `processBlock_Common` so these are considered first.

Exercise 1:
Develop a SAW file for proving that the `sigma` and `SIGMA` functions above are equivalent to their corresponding Crypto specifications. Run `saw` on the file and report the result. ■

The next target is sha512_block_data_order because SHA512 in sha512.c, the end goal in proving correctness for this example, depends on SHA512_Init, SHA512_Update, and SHA512_Final, all of which depend on sha512_block_data_order and the proof for sha512_block_data_order runs faster with the above functions proved first. The specification given in the FIPS publication starts on Page 20. The Cryptol specification is processBlock_Common in SHA.cry. In sha512.c sha512_block_data_order is used in SHA512_Init, SHA512_Update, and SHA512_Final like this:

```
sha512_block_data_order (c->h, p, 1)
```

where, from sha512.h, c->h is an array of 8 64 bit integers, p is an array of 128 8 bit integers and from the declaration of SHA512_block_data_order, 1 is of type size_t which is a 64 bit integer. Therefore, for the SHA512_block_data_order setup in a SAW file, say s2.saw, the llvm_execute_func line looks like this:

```
llvm_execute_func [state_ptr, data_ptr, llvm_term {{ 1:[64]}}];
```

where state_ptr is part of a pair (state,state_ptr) obtained from

```
(state,state_ptr) <- pointer_to_fresh "state" (llvm_array 8 (llvm_int 64));
```

and data_ptr is part of a pair (data,data_ptr) obtained from

```
(data,data_ptr) <- pointer_to_fresh "data" (llvm_array 128 (llvm_int 8));
```

The state and data variables are used by the Cryptol specification in

```
llvm_points_to state_ptr
   (llvm_term {{ processBlock_Common state (split (join data)) }});
```

(see SHA.cry for why (split (join date)) is used in the above). The llvm_verify line, partially written, looks like this:

```
sha512_bdo_ov <- llvm_verify m "sha512_block_data_order"
                 [Sigma0_ov, Sigma1_ov, sigma0_ov, sigma1_ov] false
                 sha512_block_data_order_setup ...;
```

If ... is replaced by, say z3, the SAW file, with all the sigma functions, declarations of pointer_to_fresh, and the completed setup function for SHA512_block_data_order, the proof will not finish, at least not in a reasonable time. For this particular case the only way to get a reasonable result is to use an uninterpreted function library. An example, which works in this case is to replace ... with

```
(w4_unint_z3 [“SIGMA_0”,”SIGMA_1”,”sigma_0”,”sigma_1”])
```

The w4_unint_z3 command is used to specify that some Cryptol functions should be considered uninterpreted. You will have to study SMT solvers, such as Z3, to understand precisely what that means for a solver and you likely do not want to do this, expecting that SAW should take care of this. Perhaps SAW will eventually be able to do so. For now, a rough explanation is offered. Suppose some function f is given and a proof of f(x) == f(y) is needed. Normally, SAW will expand f and pass the whole thing to the SMT solver. However, by calling (w4_unint_z3 ["f"]), then SAW will leave f uninterpreted and the solver will only be able to prove f(x) == f(y) if x == y. This works because Cryptol functions are pure and so if the arguments are equal then the return values must be equal. So, the downside of declaring functions uninterpreted is the desired proof may be missed but, if the above is enough, a fast proof will be generated. The problem is knowing which functions can be deemed uninterpreted. Here is some advice. When a SAW proof hangs there are a couple things to try first:

1. If the C code has some complexity that you think might be slowing the proof down, prove that bit separately and use it as an override.
2. If the Cryptol specification has some complexity that you think might be slowing the proof down and the same exact Cryptol function appears on both sides of the equality, then leave that function uninterpreted. It's important that the arguments for this function on both sides of the equality are themselves equal for the uninterpreted trick to work. We will call this idea “equals for equals”.

For this problem, compare the sigma functions in `sha512.c` and `SHA512.cry`. These are very simple and obey “equals for equals”. The next question is why were these picked for being uninterpreted? Sorry, the answer is it was tried and it worked. Other functions could have been deemed uninterpreted but they were unnecessary. By the way, `(w4_unint_z3 [])` is the same as `z3`.

Exercise 2:
Verify C function `SHA512_block_data_order` is functionally equivalent to the Cryptol specification `processBlock_Common`. Write the SAW file, `s2.saw`, and run it. ■

Next consider the equivalence of the `SHA512_Update` function in C and the `SHAUpdate` specification in Cryptol. The goal is to add the following `llvm_verify` line to s2.saw which will become s3.saw:

```
update_ov <-llvm_verify m "SHA512_Update"
   [sha512_block_data_order_ov] false SHA512_Update_setup
   (w4_unint_z3 ["processBlock_Common"]);
```

The above makes use of `sha512_block_data_order_ov`, found earlier, and recognizes that `processBlock_Common` should be treated as uninterpreted. These are hints intended to save the reader time figuring these out on their own.

The next step is to complete `SHA512_Update_setup`. Start with

```
llvm_execute_func [sha_ptr, data_ptr, llvm_term {{ `127 : [64] }}];
```

These parameters correspond to the `SHA512_Update` function of `sha512.c`:

```
int SHA512_Update(SHA512_CTX *c, const void *in_data, size_t len)
```

The `sha_ptr` will be the `SHA512_CTX` struct pointer `c`. This may be obtained from the following:

```
(sha512_ctx, sha_ptr) <- pointer_to_fresh_sha512_state_st 0;
```

to be added to the setup function (note: the 0 will be important later but even 127 is fine for now) and

```
let pointer_to_fresh_sha512_state_st n = do {
   h <- llvm_fresh_var "sha512_ctx.h" (llvm_array 8 (llvm_int 64));
   block <- if eval_bool {{ `n == 0 }} then do {
      return {{ [] : [0][8] }};
   } else do {
      llvm_fresh_var "sha512_ctx.block" (llvm_array n (llvm_int 8));
   };
   sz <- llvm_fresh_var "sha512_ctx.sz" (llvm_int 128);
   let state = {{ { h = h, block = (block # zero) : [128][8], n = `n : [32], sz = sz } }};
   ptr <- llvm_alloc (llvm_struct "struct.sha512_state_st");
   points_to_sha512_state_st_common ptr (h, sz, block, {{ `n : [32]}}) n;
   return (state, ptr);
};
```

To be added to the SAW file as a function block.

The `data_ptr` will be the `in_data` pointer. The `llvm_term` that is 127 will be the `len` parameter: block size is 128 bytes and 127 will be a message length one short of that (you can try other values). The in_data parameter is an array of 127 bytes, which can be cast in the `SHA512_Update_setup` as

```
(data, data_ptr) <- pointer_to_fresh "data" (llvm_array 127 (llvm_int 8));
```

where `pointer_to_fresh` has been completed earlier. The return value of the `SHA512_Update` function is 1 which, as a SAW directive in the setup function, becomes this:

```
llvm_return (llvm_term {{ 1 : [32] }});
```

Needed later will be a post-condition that ties the action of the Cryptol `SHAUpdate` function with the C `sha512_Update` function. This can be managed by the following SAW functions:

```
let points_to_sha512_state_st ptr state num = do {
  points_to_sha512_state_st_common
    ptr ({{ state.h }}, {{ state.sz }}, {{ take`{num} state.block }}, {{ state.n }}) num;
};

let points_to_sha512_state_st_common ptr (h, sz, block, n) num = do {
  llvm_points_to (llvm_field ptr "h") (llvm_term h);
  llvm_points_to_at_type (llvm_field ptr "Nl") (llvm_int 128) (llvm_term sz);
  if eval_bool {{ `num == 0 }} then do {
    return ();
  } else do {
    llvm_points_to_untyped (llvm_field ptr "p") (llvm_term block);
  };
  llvm_points_to (llvm_field ptr "num") (llvm_term n);
  llvm_points_to (llvm_field ptr "md_len") (llvm_term {{ `64 : [32] }});
};
```

So, the following post-condition, to be used later, can be added to the setup function as:

```
points_to_sha512_state_st sha_ptr {{ SHAUpdate sha512_ctx data }} 127;
```

Exercise 3:
Verify C function `SHA512_Update` is functionally equivalent to the Cryptol specification `SHAUpdate`. Write the SAW file, `s3.saw`, and run it. ■

Galois, Inc. © 2023 | All rights reserved

The next step is to verify the equivalence of the Cryptol `SHAFinal` function and the C `SHA512_Final` function. The llvm_verify in main will look like this:

```
final_ov <- llvm_verify m "SHA512_Final"
            [sha512_bdo_ov] false SHA512_Final_setup
            (w4_unint_z3 ["processBlock_Common"]);
```

For `SHA512_Final_setup` we need

```
llvm_execute_func [out_ptr, sha_ptr];
```

for the C function `SHA512_Final` which has the prototype

```
int SHA512_Final (uint8_t out[SHA512_DIGEST_LENGTH], SHA512_CTX *sha)
```

Parameter `out_ptr` is just a 64 byte array for the digest that may be defined as

```
out_ptr <- llvm_alloc (llvm_array 64 (llvm_int 8));
```

The `sha_ptr` is obtained, as before, like this:

```
(sha512_ctx,sha_ptr) <- pointer_to_fresh_sha512_state_st 127;
```

Finally, we need a post-condition that ensures the value of `out_ptr` from the C function `SHA512_Final` given state `sha512_ctx` matches the output of the Cryptol function `SHA512Final` given the same state. This post-condition looks like this:

```
llvm_points_to out_ptr (llvm_term {{ split`{64} (SHAFinal sha512_ctx) }});
```

The reason for the ``split`{64}`` is to get an array (or sequence) of 64 bytes from the 512 bit number that `SHAFinal` produces so as to match the type of the 64 byte array that `out_ptr` is referencing.

Exercise 4:
Verify C function `SHA512_Final` is functionally equivalent to the Cryptol specification `SHAFinal`. Write the SAW file, `s4.saw`, and run it. ■

Now it is time to verify the equivalence of Cryptol specification SHAImp and the C function SHA512 given, in this case, 127 byte data. In `main` of the SAW file add the following:

```
llvm_verify m "SHA512"
   [update_ov, final_ov] false SHA512_setup
   (w4_unint_z3 ["processBlock_Common"]);
```

It remains to write `SHA512_setup`. This is straightforward as it is similar to previous setup blocks and discussion is not needed. The C function call in the setup block looks like this:

```
llvm_execute_func [ data_ptr, llvm_term {{ `127 : [64] }}, out_ptr];
```

Observer in `sha512.c` the prototype of function `SHA512` is:

```
uint8_t *SHA512(const uint8_t *data, size_t len, uint8_t out[SHA512_DIGEST_LENGTH])
```

The `data_ptr` term is created from this:

```
(data, data_ptr) <- pointer_to_fresh "data" (llvm_array 127 (llvm_int 8));
```

The `out_ptr` term is created from this:

```
out_ptr <- llvm_alloc (llvm_array 64 (llvm_int 8));
```

The post-condition looks like this:

```
llvm_points_to out_ptr (llvm_term {{ split`{64} (SHAImp data) }});
```

and the C function outputs out_ptr so the following becomes the return value in the setup block:

```
   llvm_return out_ptr;
```

Exercise 5:
Verify C function `SHA512` is functionally equivalent to the Cryptol specification `SHAImp`. Write the SAW file, `s5.saw`, and run it. ■

Exercise 6:
The digest function of Cryptol does not require the length of the input message as input. The C function should not either. Modify sha512.c to take as input from the command line a message up to 127 characters and to output the 32 byte digest array associated with that input. Run `sha512 “Hello World Folks”` and verify the output is as above using the Cryptol function `SHAImp`. ■

Galois, Inc. © 2023 | All rights reserved