Верификация алгоритма SHA-256 с помощью Coq и VST

Статья посвящена формальной верификации C-программы OpenSSL SHA-256 с помощью доказательного помощника Coq и модуля VST. В документе описывается спецификация SHA-256, её формализация на языке Coq и применение сепарационной логики для верификации C-кода.

Верификация алгоритма SHA-256 состоит из нескольких этапов:

1. Написание функциональной спецификации алгоритма на языке Coq, которая является формализацией стандарта FIPS 180-4 Secure Hash Standard [FIPS 2012].
2. Написание спецификаций, использующих сепарационную логику и тройки Хоара для связи функциональной спецификации на языке Coq с реализацией на языке Си.
3. Доказательство того, что код на Си удовлетворяет написанным спецификациям.

Функциональная спецификация

Функциональная спецификация представляет из себя описание текста стандарта в Coq. Результатом данного этапа является файл на языке Coq с функциональной спецификацией стандарта стандарта FIPS 180-4.
Пример функциональной спецификации требования к вычислению расписания сообщений, которое используются для итеративной генерации хэша:

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

Данное требование транслируется в Coq следующим образом:

```
Function W (M: Z → int) (t: Z) {measure Z.to_nat t} : int :=
  if zlt t 16
  then M t
  else (Int.add (Int.add (sigma_1 (W M (t-2))) (W M (t-7)))
          (Int.add (sigma_0 (W M (t-15))) (W M (t-16)))).
```

В процессе написания функциональной спецификации также возникает вопрос относительно её корректности. Проблема в том, что Coq является функциональным языком, а следовательно менее производительным, чем язык Си. То есть вычислить функцию и проверить её результат даже на мощном железе не всегда представляется возможным. Автор статьи решил данную проблему, реализовав функции хеширования с помощью более оптимизированного алгоритма, который отличается от алгоритма,

описанного в ГОСТе. Далее, проверив корректность быстрой функции на разных входах, можно доказать её эквивалентность функции SHA-256, описанной в стандарте, воспользовавшись аксиомой функциональной экстенсиональности.

Исходный код:
source code 1 SHA256.v
source code 2 spec_sha.v

Формализация условий корректности программы на Си

На этом этапе автор статьи для каждой функции в языке Си должны составить формальную спецификацию с помощью логики и тройки Флойда-Хоара.

Specification of how the C code corresponds function-for-function to the functional spec.

Реализация функции SHA256 на C имеет особенности (стр.10) :

- Инкрементальное хэширование (Поддерживает функции `SHA256_Init`, `SHA256_Update`, `SHA256_Final` для обработки сообщений по частям, даже если они не выровнены по блокам)
  - One calls SHA256_Init to initialize a context,
  - SHA256_Update with each si in turn,
  - then SHA256_Final to add the padding and length and hash the last block.
- Within the 64-round computation, it store only the the most recent 16 elements of Wt, in a buffer accessed modulo 16 using bitwise-and in the array subscript.

Пример:

```
void SHA256(const unsigned char *d, size_t n, unsigned char *md) {
    SHA256_CTX c;
    SHA256_Init(&c);
    SHA256_Update(&c,d,n);
    SHA256_Final(md,&c);
}
```

source code

```
Definition SHA256_spec :=
 DECLARE _SHA256
 WITH d: val, len: Z, dsh: share, msh: share, data: list Z, md: val; kv: val
 PRE [ _d OF tptr tuchar, _n OF tuint, _md OF tptr tuchar ]
   PROP (readable_share dsh; writable_share msh;     Zlength data * 8 < two_p 64;
         Zlength data ≤ Int.max_unsigned)
   LOCAL (temp _d d; temp _n (Int.repr (Zlength data)); temp _md md; gvar _K256 kv)
   SEP(K_vector kv; (data_block dsh data d); memory_block msh (Int.repr 32) md)
 POST [ tvoid ]
   SEP(K_vector kv; (data_block dsh data d); data_block msh (SHA_256 data) md).
```

Доказательство корректности кода на Си

Доказательство каждой из вызываемых функций на языке Coq было выделено в отдельный модуль.
Структура проекта следующая:
verif_sha_bdo.v
verif_sha_final.v
verif_sha_init.v
verif_sha_update.v
verif_sha.v
verif_SHA256.v
spec_sha.v

Пример доказательства корректности функции **SHA256:**

```
Lemma body_SHA256: semax_body Vprog Gtot f_SHA256 SHA256_spec.
Proof.
start_function.
rewrite data_at__isptr; Intros.
forward_call (* SHA256_Init(&c); *)
   (v_c, Tsh).
forward_call (* SHA256_Update(&c,d,n); *)
  (@nil byte, data,v_c,Tsh, d,dsh, Zlength data, gv).
 simpl app.

forward_call (* SHA256_Final(md,&c); *)
    (sublist 0 (Zlength data) data, md, v_c, Tsh, msh, gv).

forward. (* return; *)
change (Tstruct _SHA256state_st noattr) with t_struct_SHA256state_st.
autorewrite with sublist.
entailer!.
Qed.
```

Исходный код: [source]

Внутри доказательства **SHA256** вызывается функция **SHA256_Init**, которая была доказана в отдельном модуле:

```
void SHA256_Init (SHA256_CTX *c)
  {
  c->h[0]=0x6a09e667UL; c->h[1]=0xbb67ae85UL;
  c->h[2]=0x3c6ef372UL; c->h[3]=0xa54ff53aUL;
  c->h[4]=0x510e527fUL; c->h[5]=0x9b05688cUL;
  c->h[6]=0x1f83d9abUL; c->h[7]=0x5be0cd19UL;
  c->Nl=0;  c->Nh=0;
  c->num=0;
  }
```

```
Lemma body_SHA256_Init: semax_body Vprog Gtot f_SHA256_Init SHA256_Init_spec.
Proof.
start_function.
name c_ _c.
unfold data_at_.
(* BEGIN: without these lines, the "do 8 forward" takes 40 times as long. *)
unfold field_at_.
unfold_data_at (field_at _ _ _ _ _).
simpl fst; simpl snd.
(* END: without these lines *)
Time do 8 (forward; unfold upd_Znth; if_tac;
  unfold Zlength in *; simpl Zlength_aux in *; try lia;
  unfold sublist; simpl app).
Time repeat forward. (* 14 sec *)
unfold sha256state_.
Exists (map Vint init_registers,
      (Vint Int.zero, (Vint Int.zero, (repeat Vundef (Z.to_nat 64), Vint Int.zero)))).
unfold_data_at (data_at _ _ _ _).
Time entailer!. (* 5.2 sec *)
repeat split; auto.
unfold s256_h, fst, s256a_regs.
rewrite hash_blocks_equation. reflexivity.
unfold data_at. apply derives_refl'; f_equal.
f_equal.
simpl.
repeat (apply f_equal2; [f_equal; apply int_eq_e; compute; reflexivity | ]); auto.
Time Qed. (* 33.6 sec *)
```

Исходный код: [source]

Выводы

Что нам может быть полезна статья/код:

- спецификация типов данных, описанных в стандарте
- трюки для работы с контекстом и памятью
- есть похожие на наши функции для хэширования, можно посмотреть как сопоставить функциональные <--> относительные спецификации для их алгоритма

Tips для спецификации

- Для проверки функциональной спецификации автор написал оптимизированный алгоритм, использующий кэширование, проверил его корректность на различных тестах, а затем доказал эквивалентность исходной функции
- Хранит только 16 последних элементов Wt в буфере, доступ к которому осуществляется по модулю 16

Интересные факты:

- Функциональная спецификация алгоритма - 202 строки, Доказательство корректности программы на C - 6539 строк