Formal Verification of C Programs for Implementations of Communication Protocols*

Reynald Affeldt† Kiyoshi Yamada‡

Abstract

In order to improve the security of the implementations of communication protocols, we have been developing a Coq library to formally verify C programs. Our approach is based on an extension of Separation logic that accommodates C types. The resulting formal model is faithful enough so that formal programs can almost be compiled as they are. So far, we have formalized standard lemmas of Separation logic and successfully applied our library to examples. We are now tackling the verification of parsing functions taken from an existing implementation of TLS (Transport Layer Security), such functions being a notorious source of security bugs. Because we can retrofit verified programs automatically, this would improve the original TLS implementation with both debugging and certification. In this presentation, we introduce our library and comment on preliminary experiments.

1 Introduction

In network communication infrastructures, it is too often the case that security issues arise from low-level implementation bugs. This is because the C programming language [2], the de facto standard for implementation of communication protocols, exposes many low-level details, so that bugs can have unforeseen consequences. Though C reconciles performance with productivity requirements, it also turns the seemingly mundane task of implementation into a weak link of network communication infrastructures. It is our purpose in this paper to devise an implementation approach that allows for better security of communication protocols.

The main idea of our approach is to use formal verification to fix the critical parts of the implementation of communication protocols, thus improving the security of the implementation as a whole. The history of security bugs in the implementation of communication protocols is now long enough to spot accurately potentially defective functions. This is for example the case of parsing functions, that decode network packets for application processing. Being also performance critical, parsing functions are typically written in C, and our first task for formal verification is the design of a formal model for C, with reasoning facilities.

Fortunately, the task of processing network packets is disciplined by various standards, the most famous being IETF RFCs. For communication protocols, they describe in a semi-formal fashion the format of the network packets and their processing. As such, they provide the starting point for implementations and, of course, any verification effort must be explained w.r.t. RFCs' contents.

To sum up, our approach to improve the security of implementations of communication protocols consists in:

1. Creating a faithful model of C for formal verification,

*This is a non-reviewed manuscript. To be presented at the 28th Workshop of the Japan Society for Software Science and Technology (JSSST 2011) (`http://jssst11.kuis.kyoto-u.ac.jp/`). Not submitted for publication.

†National Institute of Advanced Industrial Science and Technology

‡Lepidum Co., Ltd.

2. Formally verifying critical parts of existing implementations, and

3. Retrofitting the verified code into the original implementation.

Formal verification in this paper is performed within the Coq proof-assistant [1]. We use the latter to define formally a model of the C programming language as well as reasoning facilities, in the form of a variant of Separation logic [3], a Hoare logic extended with pointers. The important aspect of this formal model is to make it as faithful as possible so that existing code can be encoded almost as it is and, more importantly, so that runnable code can be extracted easily from the formal model. This reduces the trusted base to a bare minimum.

We apply our approach to an existing implementation of the TLS protocol [6], namely PolarSSL [7]. Concretely, we extract the function that parses ClientHello packets (that initiate TLS sessions), specify it w.r.t. the relevant RFC [6], and verify the function. It is interesting to observe that even recent security bugs can be found in such a well-scrutinized function (e.g., CVE-2011-0014). Although we have not yet completed this verification, we observe that, in spite of the many low-level details, the verification effort remains manageable providing adequate reasoning infrastructure. As a side-effect of formal verification, we are led to formalize parts of the TLS RFC. This already led to improvements such as patching (minor) errors (see Sect. 5.2). At this stage, we would identify our contributions as (1) a Coq library to write C programs and extract from them runnable code, (2) a Separation logic for C equipped with various lemmas, and (3) improved parsing functions for an existing implementation of TLS.

Outline The first part of this paper deals with formalization of C and Separation logic. In Sect. 2, we introduce our formal model of C. In Sect. 3, we introduce a Separation logic for our formal model of C. In Sect. 4, we illustrate the resulting formal library with the standard in-place list reversal example. The second part of this paper discusses application to the implementation of communication protocols. In Sect. 5, we explain how we formalize definitions from the TLS RFC so as to verify parsing functions. In Sect. 6, we show how we tackle verification of a parsing function taken from PolarSSL [7]. In Sect. 7, we show that formal C programs can be retrofitted easily to their original application. We discuss related work in Sect. 8 and conclude in Sect. 9, where we also discuss current and future work.

2 Formal Model of C

2.1 C Types

There are three basic types: unsigned and signed 32-bit integers, and unsigned 8-bit characters:

```
Inductive ityp : Set :=
        uint32 | sint32 | uchar.
```

The types of C are defined by the following inductive types:

```
Inductive stag := mkStag : string → stag.
Inductive typ : Set :=
| btyp : ityp → typ
| ptyp : typ → typ
| rtyp : stag → typ
| styp : stag → list (string * typ) → typ.
```

`stag` corresponds to structs' tags. The constructor `btyp` wraps basic types, `ptyp` corresponds to types for pointers, `styp` corresponds to C structs (fields are encoded as strings). `rtyp` also corresponds to types for pointers. Intuitively, `rtyp tg` is the same as `ptyp (styp tg flds)`; the existence of this alternative way to write pointers to structures allows for the definition of recursive data structures. For example, singly-linked lists are defined as follows:

```
Definition fields_typs :=
  ("data", btyp uint32) ::
  ("next", rtyp (mkStag "pair")) :: nil.
Definition pair_data_next : typ :=
  styp (mkStag "pair") fields_typs.
```

`sizeof` is a function that computes the number of bytes needed to put data in memory. It is defined using the following functions:

```
Definition size_ityp t : nat :=
 match t with uint32 ⇒ 4 | sint32 ⇒ 4
 | uchar ⇒ 1 end.
Fixpoint sizeof' (n : nat) (t : typ) : nat
 := match n with ... | S m ⇒
```

```
  match t with
   | btyp x ⇒ size_ityp x
   | ptyp _ ⇒ ptr_div_8
   | rtyp _ ⇒ ptr_div_8
   | styp _ l' ⇒
   iplus (map (sizeof' m) (uzip2 l'))
  end
 end.
Definition sizeof t :=
 sizeof' (typ_max_depth t) t.
```

`ptr_div_8` is the number of bytes needed to encode a pointer. It is a parameter of our model that is assumed to have the following properties.

```
Parameter size_ptr : nat.
Parameter ptr_div_8 : nat.
Parameter Hptr_div_8 :
 size_ptr = ptr_div_8 * 8.
Parameter ptr_div_8_neq0 : ptr_div_8 ≠ 0.
Parameter size_ptr_32 : 32 ≤ size_ptr.
```

This therefore allows with 32, 64, etc. architectures). `sizeof'` takes a size argument `n` that is expected to be the depth of the `t`: this is standard practice to show Coq that the function terminates.

The `sizeof` function does not assume any padding; padding can be added in an explicit way (using unaccessed fields) so as to formalize, e.g., non-portable programs (such as programs with assembly code).

Not all types allowed by the above syntax are acceptable. `wft t` ("wellformed type") is a predicate that holds when `t` does not defined nor an empty struct neither a struct with homonymous fields.

We now define (in two steps) equality between types. The first step is a predicate `a =t= b` that holds when `a` and `b` are syntactically the same type (but it treats `rtag tg` as the same as `ptyp (styp tg flds)` for any `flds`). The second step is a predicate `t1 =t g t= t2` parametrized by a type context `g`:

```
Definition ctxt :=
 list (stag * list (string * typ)).
Definition eqtm (g : ctxt) (t1 t2 : typ)
 := t1 =t= t2 ∧ cover g t1 ∧ cover g t2.
Notation "t1 '=t' g 't=' t2" :=
 (eqtm g t1 t2).
```

`cover g t` (definition omitted here) means that all the tags in `t` appear in (the domain of) `g`. `a =t g t= b` is an equality relation (reflexivity requires `cover g a`).

2.2 C Expressions

Values of the expression language are made of finite-size integers. In particular, the size of pointers is not fully specified: they are only known to be finite-size integers of size `size_ptr` (as explained in Sect. 2.1):

```
Inductive value : Type :=
| bval_int : int 32 → value
| bval_char : int 8 → value
| pval : int size_ptr → value
| stval : list value → value.
```

We model a subset of the expression language of C by an inductive type:

```
Inductive exp : Type :=
| var_e : var → exp
| cst32 : int 32 → exp
| cst8 : int 8 → exp
| cst_pe : ∀ t : typ,
  wft t → int size_ptr → exp
| cst_se : ∀ tg l vs, wft (styp tg l) →
  length l = length vs →
  ∀ (fun x ⇒ typ_val (fst x) (snd x))
  (combine (uzip2 l) vs) → exp.
| fld : exp → string → exp
| fld' : exp → string → exp
| bop_ne : binop_e → exp → exp → exp
| add_pe : exp → exp → exp
...
```

`var_e` is for variables (the type `var` is synonym for Coq `strings`). `cst32, cst8` are for (signed) integral constants; they are represented by finite-size integers. For example,

```
Definition cst32_0 := cst32 (Z2s 32 0).
```

is the expression for the constant 0 (`Z2s` converts an integer into the equivalent finite-size integer, interpreted as signed). `cst_pe` is for pointer constants; they come with a proof of wellformedness of the pointed type. `cst_se` is for constant structs; they come with a proof of wellformedness of the type of the struct and a proof that the provided list of values is "compatible" (e.g., a pointer of type `ptyp t` is associated to a value of type `int size_ptr`, not, say, a value of type `int 32`; this is captured by the relation `typ_val`). When computable, these details are hidden by the following notation:

```
Notation "v 'CST_SE' Hv" := (cst_se _
 _ v Hv (refl_equal _) (refl_equal _)).
```

`fld` and `fld'` are for accessing the fields of a struct: `fld` corresponds to the "`.`" C notation and `fld'` corresponds to the "`->`" C notation (more precisely, `fld' p f` would be written `&(p -> f)` in C). `bop_ne` is for various binary operators (on finite-size integers). `add_pe` is for pointer arithmetic. See [15] for the complete list of constructors.

2.3 Typing of Expressions

The typing of C expressions is defined w.r.t. a typing environment:

```
Definition tenv := list (var * typ).
```

For the types in the typing environment to make sense, typing environments are defined w.r.t. a type context (`ctxt` defined in Sect. 2.1).

The definition of the typing function has to cope with the issue of the double representation of pointers to structure. Concretely, when typing runs into `rtyp tg`, it has to look for the set of fields/types corresponding to the tag `tg` in the type context. This is exemplified by the following excerpt of the typing function:

```
Fixpoint typ_of (g : ctxt) (env : tenv) e
:= ...
| fld' e' f ⇒
  match typ_of g env e' with
    | Some (ptyp (styp s ss_ts)) ⇒
      match assoc_get f ss_ts with
        | Some t' ⇒ Some (ptyp t')
        | None ⇒ None
      end
    | Some (rtyp tg) ⇒
      match assoc_get tg g with
        | Some lst ⇒
          match assoc_get f lst with
            | Some t' ⇒ Some (ptyp t')
            | None ⇒ None
          end
        | None ⇒ None
      end
    | _ ⇒ None
  end
...
```

The above typing function also performs some type promotion when, e.g., different basic types are involved in an arithmetic expression (see [15] for additional details).

2.4 Evaluation of Expressions

The evaluation of expressions requires typing information to deal with pointer arithmetic. In C, `int* p; p++` does not mean to add one but to add one time the size of an `int` (typically, 4 on a 32-bit architecture) to the value of `p`.

Evaluation of expressions is defined w.r.t. a store where variables are associated with a value and a type:

```
Definition tstore :=
  list (var * (value * typ)).
```

Since `tstore` as such does not prevent ill-typed values, evaluation is actually defined w.r.t. a `cstore` that associates `tstore` with a type context (obtainable via the `_ctxt` projection) and we guarantee that any value in the store is associated with a compatible (in the sense of the relation `typ_val` seen in Sect. 2.2). Note that the evaluation of expressions does not perform read/write side-effect w.r.t. the heap; this is common practice in formalizing imperative programs (e.g., [11] or [12]—still, Compcert version 1.8 (09/2010) has recently improved on that).

The following excerpt of the evaluation function illustrates pointer arithmetic:

```
Definition typof (s : cstore) e :=
 typ_of (_ctxt s) (cstore_tenv s) e.
Fixpoint eval (e : exp) (s : cstore)
  : option value := ...
| add_pe e1 e2 ⇒
 match eval e1 s, eval e2 s with
 | Some (pval i1), Some (bval_int i2) ⇒
  match typof s e2 with
  | Some (btyp sint32) ⇒
   match typof s e1 with
  | Some (ptyp t) ⇒
    Some (pval (int_scale i1 (sizeof t)
    (Zabs_nat (s2Z i2))))
  | Some (rtyp tg) ⇒
   match assoc_get tg (_ctxt s) with
   | Some lst ⇒
     Some (pval (int_scale i1 (sizeof
     (styp tg lst)) (Zabs_nat (s2Z i2))))
   | None ⇒ None
   end
  | ...
   end
  | ...
  end
 | ...
 end
```

```
...
Notation "'[' e ']_' s" := (eval e s).
```

(`s2Z` turns a finite-size integer into its signed decimal interpretation. `int_scale p i k` means to add $k \times i$ to the value of `p`.)

2.5 C Commands

To produce a subset of the C language, we use a parametrized module for Hoare logic from previous work [14]: given a syntax, operational semantics, and axiomatic semantics for a set of one-step commands (assignment, etc.), it generates a syntax, operational semantics and Hoare logic for the corresponding WHILE-language (i.e., with structured control-flow). See [14] or file `while.v` in [15] for this module definition. Here follows the set of one-step commands that we use to define our subset of C. The names of constructors are self-explanatory.

```
Inductive cmd0 : Type :=
| skip : cmd0
| assign : var → exp → cmd0
| lookup : var → exp → cmd0
| mutation : exp → exp → cmd0
| malloc : var → exp → cmd0
| free : exp → cmd0.
Notation "x ← e" := (assign x e).
Notation "x '←*' e" := (lookup x e).
Notation "e '*←' f" := (mutation e f).
Notation "x '←malloc' e" := (malloc x e).
```

In the operational semantics, a state is a pair of a store of typed variables (the `cstore` type from Sect. 2.2) and a heap (type `hp.t`), that is map from naturals (that represent addresses) to individual bytes:

```
Definition state := cstore * hp.t.
```

An important difference between the language that we are defining and an archetypal language such as the one of Separation logic [3] is that memory accesses are done by chunks, whose length is given by the type of read/written data. For illustration, the excerpt below shows the operational semantics of lookup. The function `heap_get` is parametrized by the type `tx` of the read value so that enough bytes (actually, `sizeof tx` bytes) are retrieved:

```
0 Reserved Notation " s '--' c '--->' t ".
```

```
1 Inductive exec0 : option state → cmd0 →
2   option state → Prop := ...
3 | exec0_lookup : ∀ s h tx x e v p,
4  typof s (var_e x) =ot _ctxt s ot= tx →
5  typof s e =ot _ctxt s ot= ptyp tx →
6  [ e ]_ s = Some (pval p) →
7  heap_get (u2Z p) tx h = Some v →
8  Some (s, h) -- x ←* e --->
9   Some (cstore_upd x v s, h)
10 ...
```

(“· `=ot` · `ot=` ·” is a notation akin to “· `=t` · `t=` ·” that deals with option types.) The semantics that we define is not very permissive when it comes to type. For example, lookup executes only when the type of the variable and of the dereferenced expression agree (lines 4–5 above). Programs that deviate from this behavior require small adjustments, what benefits anyway to clarity and therefore security.

2.5.1 Dynamic Allocation

The model for dynamic allocation extends the archetypal model of [3] (Sect. 7) with C types and a notion of “control block”. The control block `ctrl a` corresponds to the allocated block starting at address `a`, the length of this allocated block is given by `block_len`, the length of the control block itself is given by `ctrl_len`.

```
Parameter ctrl : nat → hp.t.
Parameter ctrl_len :
 ∀ a h, h = ctrl a → nat.
Parameter block_len :
 ∀ a h, h = ctrl a → nat.
```

As depicted in Fig. 1, dynamic allocation creates a newly allocated block of the requested (non-zero) length together with a control block.

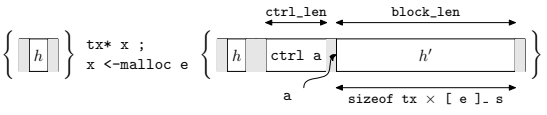

Figure 1: Effect of Dynamic Allocation

The operational semantics of dynamic allocation is formalized as follows:

```
| exec0_malloc : ∀ s h t x e adr v l
 (lst : list (int 8)),
 typof s (var_e x) =ot _ctxt s ot=
```

```
 ptyp t →
wft t →
[ e ]_s = Some (bval_int v) →
u2Z v ≠ 0 →
l = sizeof t → length lst = u2Z v * l →
(0 ≤ adr + u2Z v * l < 2 ^ size_ptr) →
h ⊥ chars2heap adr lst →
h ⊥ ctrl adr →
ctrl adr ⊥ chars2heap adr lst →
Block_len (ctrl adr) = length lst →
Some (s, h) -- x ←malloc e ---->
Some (cstore_upd x
 (pval (Z2u size_ptr adr)) s,
 h ⊎ ctrl adr ⊎ chars2heap adr lst)
```

(`chars2heap a l` turns the list `l` of bytes into a heap of contiguous cells starting at address `a`; `Z2u` converts an integer into the equivalent finite-size integer, interpreted as unsigned.) See [15] for the corresponding free function.

3 Separation Logic for C

3.1 Connectives

We formalize Separation logic [3] using a shallow encoding. This means that logical predicates are represented by Coq functions that end in the type `Prop` of Coq propositions:

```
Definition assert := cstore → hp.t → Prop.
```

This is a standard encoding approach that we already used in previous work [4, 5, 10, 14]. The separating conjunction is the most important connective of Separation logic. Intuitively, $P \star Q$ holds for a store s and a heap h when h can be split into two disjoint heaps h_1 and h_2 such that P holds for s and h_1 and Q holds for s and h_2. Put formally:

```
Definition con (P Q : assert) : assert :=
 fun s h ⇒ ∃ h1, ∃ h2, h1 ⊥ h2 ∧
  h = h1 ⊎ h2 ∧ P s h1 ∧ Q s h2.
Notation "P ⋆ Q" := (con P Q).
```

A singleton heap is represented by a *mapsto* formula. Because of types, the mapsto formula for C differs from the one of textbook Separation logic [3]. In the latter, a singleton heap corresponding to address a and contents b (where both a and b are integers) is represented by the mapsto formula $e_1 \mapsto e_2$, where e_1 and e_2 evaluate respectively to a and b.

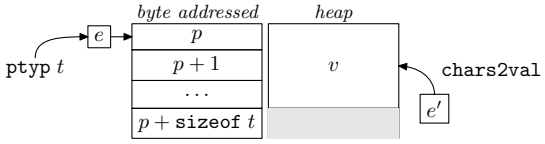

Figure 2: The heap specified by $e \overset{t}{\mapsto} e'$

For C, we extend the mapsto formula with type information to account for the various data structures. More precisely, the mapsto formula is written $e \overset{t}{\mapsto} e'$: e is an expression of type $*t$ that evaluates to some pointer p, e' is an expression of type t that evaluates to some value v, and p points to a memory chunk that contains the encoding of v. Figure 2 depicts this situation graphically. Put formally:

```
Definition mapsto t e e' s h :=
 ∃ p, [ e ]_ s = Some (pval p) ∧
 typof s e =ot _ctxt s ot= ptyp t ∧
 ∃ v, [ e' ]_ s = Some v ∧
  sizeof t = length (hp.cdom h) ∧
  chars2val t (hp.cdom h) = (v, nil) ∧
  hp.dom h = seq (u2Z p) (sizeof t).
Notation "e1 '⊢' t '↦' e2" :=
 (mapsto t e1 e2).
```

(`chars2val t l` turns the list of characters `l` into a value according to type `t`; `seq a n` is the list `a :: (a + 1) :: ... :: (a + n)`.)

3.2 Properties of Connectives

Once connectives are defined, several standard lemmas must be proved to facilitate the task of formal verification. Thanks to shallow encoding, the properties of classical predicate logic are directly derived from Coq primitives (see [4]). Shallow encoding also facilitates proofs of standard Separation logic lemmas, such as *monotony* (the property of Separation logic that allows for a logical account of destructive update [3]).

Yet, the introduction of types in connectives leads to original lemmas. For illustration, let us suppose that variable `x` points to a struct of type `styp tg l`, what would be specified as follows:

```
(var_e x ⊢ styp tg l → cst_se
   tg l vs wf_tg l_vs l_vs2) s h
```

Suppose that the ith field of `l` is named `f` and has type `t`, and that the ith value in the heap is `k`. Then, we would like, for example when performing a lookup, to derive that:

```
(var_e x .→ f ⊢ t → k ⋆ TT) s h
```

where `TT` is a formula that holds for any state. This is captured by the following lemma:

```
Lemma mapsto_styp_inv : ∀ x tg l vs s h p,
 assoc_get tg (_ctxt s) = Some l →
 ∀ (wf_tg : wft (styp tg l))
 (l_vs : length l = length vs)
 (l_vs2 : ∀ (fun x ⇒ typ_val (fst x)
  (snd x)) (combine (uzip2 l) vs)),
 (var_e x ⊢ styp tg l → cst_se tg l vs
   wf_tg l_vs l_vs2) s h →
 [var_e x]_ s = Some (pval p) →
 u2Z p + sizeof (styp tg l)
  < 2 ^ size_ptr →
 ∀ i f t t' vi k,
  ifind f (uzip1 l) = Some i →
  ith i vs = Some vi →
  assoc_get f l = Some t' →
  t =t _ctxt s t= t' →
  val2cst (_ctxt s) t vi = Some k →
  (var_e x .→ f ⊢ t → k ⋆ TT) s h.
```

(`val2cst` is a function that turns a value into a constant expression.) This lemma is used in the verification discussed in Sect. 6.3.

3.3 Triples

The language to derive Hoare triples, as well as standard Hoare logic lemmas (soundness, etc.), are obtained by the instantiation of a parametrized module (as already explained in Sect. 2.5). Of course, this requires providing basic triples for one-step commands; since they mostly mimic the operational semantics (such as `exec0_lookup` seen in Sect. 2.5 or `exec0_malloc` seen in Sect. 2.5.1), we omit them here. We were also able to derive Separation logic-specific lemmas such as the *frame rule* [3], an important lemma that enables code composition and also the proof of further lemmas.

4 Example: In-place List Reversal

In-place list reversal is the standard example for Separation logic. This program operates on singly-linked lists as (formally) defined in Sect. 2.1:

```
Definition NULL : exp :=
 cst_pe wf_pair (Z2u size_ptr 0).
Definition reverse_list :=
 ret ← NULL ;
 while.while (¬ (var_e i = NULL)) (
  rem ←* (var_e i .→ "next") ;
  (var_e i .→ "next") *← var_e ret ;
  ret ← var_e i ;
  i ← var_e rem).
```

Formal verification amounts to prove that the program `reverse_list` reverses the list `l` (not the list in the C program but its equivalent in the Coq language), pointed to by variable `i` before execution and pointed to by variable `ret` after:

```
0  Lemma reverse_list_verif : ∀ l,
1  {{ fun s h ⇒ (wf_tstore (_tstore s) ∧
2     map_tstore (_tstore s) ∧
3     wf_tenv (cstore_tenv s) ∧
4     s ⊢^g "pair" ⊣ fields_typs ∧
5     s ⊢^t rem ⊣ ptyp pair_data_next ∧
6     s ⊢^t i ⊣ ptyp pair_data_next ∧
7     s ⊢^t ret ⊣ ptyp pair_data_next) ∧
8   reverse_condition l i s h }}
9  reverse_list
10 {{ reverse_condition (rev l) ret }}.
11 Proof. ... Qed.
```

Type information appears in the precondition of the triple: line 4 means that the type of singly-linked lists belongs to the type context, lines 5 to 7 give the type of the variables (they are all pointers to singly-linked lists). Lines 1 to 3 are wellformedness conditions on the store of variables. We have been able to complete the formal verification without appealing to any axioms, but the proof script is several hundreds of lines long. This is because of intricate byte-level manipulations for which better lemmas are still to be found.

5 Formal Specification of TLS Packets

In this section, we explain how we formalize in Coq the standard definition of TLS packets [6]. The objective is application to the formal verification of parsing functions from existing TLS implementations.

5.1 Packet Description

The TLS protocol provides communications security over the Internet. It exchanges "records". Each record carries a "content type" field (e.g., 22 stands for Handshake protocol in Fig. 3), version fields and a length.

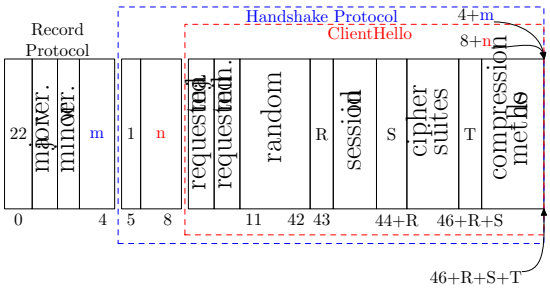

Figure 3: A ClientHello Packet

When a connection is initiated, records carry Handshake protocol packets. Each handshake protocol packet carries a(nother) "content type" field (e.g., 1 stands for "ClientHello" in Fig. 3) and a length. The negotiation phase starts with a "ClientHello" by which the client specified the version it supports, provides a random number, potentially provides a session id (to resume a previous handshake), suggests cipher suites and compression methods (Fig. 3 is a concrete illustration).

5.2 The Representation Language

In the TLS standard, the formatting of data is presented using a language (the "presentation language"): this is the topic of Sect. 4 in the RFC [6]. It defines various data structures. For example, it defines data structures called "variable-length vectors". A variable-length vector of type `T T'<floor..ceiling>` is a string of bytes (whose length is comprised between `floor` and `ceiling`) that encodes data structures of type `T` and that starts with a header of bytes (the "length field") that is big enough (but no larger) to contain the length of the payload. For example: `opaque mandatory<300..400>` has a length field of two bytes (so that any number between 300 and 400 can be encoded).

The formalization of the presentation language is a prerequisite to a trustful verification of a TLS implementation.

We encode in Coq the presentation language using the `tls_typ` inductive type. It has two parameters: the minimum and the maximum size in bytes of the underlying data. We make use of dependent types to figure out the length field for variable-length vectors (`varr` constructor below) (but also to figure out the length field of enumerateds and check divisibility constraints on arrays, see Sect. 4 of [6] for these additional data structures):

```
Inductive tls_typ : Z → Z → Type :=
| opaque : tls_typ 1 1
| arr : ∀ n, tls_typ n n →
  ∀ m, 0 ≤ m → Zmod m n = 0 →
  tls_typ m m
| enum : ∀ k l n, nodup l →
  Zmax_lst_opt l n < 2 ^ (k * 8) →
  2 ^ ((k − 1) * 8) ≤ Zmax_lst_opt l n →
  tls_typ k k
| varr : ∀ n m (t : tls_typ n m) k a b,
  k != 0 →
  b < 2 ^ (k * 8) →
  2 ^ ((k − 1) * 8) ≤ b →
  a ≤ b →
  m ≤ k + b →
  tls_typ (k + a) (k + b)
| pair : ∀ {n1 m1 n2 m2}, string →
  tls_typ n1 m1 → tls_typ n2 m2 →
  tls_typ (n1 + n2) (m1 + m2)
| typ_nil : tls_typ 0 0.
```

Given adequate notations, the above inductive type is a convenient way to produce a formal version of the RFC. For example, the previous `mandatory` type, can be written in Coq syntax as follows (the length field is explicit but uniquely determined):

```
Notation "T \< a \.. b \> n " :=
```

```
  (varr _ _ T n a b (refl_equal _)
    (refl_equal _) (refl_equal _)
    (refl_equal _) (refl_equal _))
  (at level 50).

Definition mandatory :=
  opaque \< 300 \.. 400 \> 2.
```

This seemingly simple-minded encoding of the representation language already allowed us to correct errors in the RFC. For example, the `Extension` type defined in Sect. 7.4.1.4 of [6] is actually not compatible with the type of `extensions` defined in Sect. 7.4.1.2 (both can be as large as $2^{16}-1$ bytes but they are nested in a strict fashion). Another example is about the length of encoded vectors. According to Sect. 4.3, it "must be an even multiple of the length of a single element" which is not possible in general when variable-length vectors are nested, such as in extensions.

It should be noted that this inductive type does not encode all the representation language: it does not encode so-called "variants" that truly are dependent records. To encode them, we resort to shallow encoding using Coq dependent records (see Record `ClientHello` in Fig. 4, on the left).

5.3 Modules to Organize the Specification

Using the inductive type of the representation language, we have turned that part regarding ClientHello (and the parts on which it depends) of the original RFC [6] into a Coq file.

The formal RFC is organized in modules that match the sections of the original RFC. For example, Fig. 4 compares that data structures defined in Sect. 7.4.1.2 of [6] (corresponding to the definition of the ClientHello packet) with the corresponding Coq module (named purposely `S7412`).

Coq definitions are only slightly more verbose than the original RFC. The difference in size has a simple explanation: Coq requires to make everything explicit.

6 Application: Verification of ClientHello Parsing

In this section, we apply the formal library developed in the previous sections to the verification of the function that parses ClientHello packet in PolarSSL [7]. We first formalize relevant data structures, port the source code to the formal model of Sect. 2, write down a triple of pre/post-conditions using the Separation logic of Sect. 3 using formal definitions from the TLS RFC developed in Sect. 5. At the time of this writing, formal verification is in progress.

6.1 PolarSSL Data Structures

There is a central data structure in PolarSSL that records the characteristics of the TLS connection. It records in particular the characteristics of the established connection as negotiated by the client and the server: the stage of the TLS protocol in which the server is (field "state" below), the version of TLS used (fields "∗_ver"), the session number used in case of resumption of the connection (field "session"), cipher suites known by the server (field "ciphers"), as well as the nonce for this session (field "randbytes", half of it being brought by the client, the other half by the server). Other fields ("in_hdr", "in_msg", "in_left") are for navigation into an array that stores the incoming packet. (This will be more precisely illustrated in Sect. 6.3.)

```
Definition ssl_context_typs :=
 ("state",          btyp sint32) ::
 ("major_ver",      btyp sint32) ::
 ("minor_ver",      btyp sint32) ::
 ("max_major_ver", btyp sint32) ::
 ("max_minor_ver", btyp sint32) ::
 ("session",        ptyp ssl_session) ::
 ("in_hdr",         ptyp (btyp uchar)) ::
 ("in_msg",         ptyp (btyp uchar)) ::
 ("in_left",        btyp sint32) ::
 ("ciphers",        ptyp (btyp sint32)) ::
 ("randbytes",      ptyp (btyp uchar)) ::
 nil.
Definition ssl_context := styp
 (mkStag "ssl_context") ssl_context_typs.
```

The definition of `ssl_session` follows. It is intended for storing session ids (see `SessionID` in Fig. 4). The

```
Module S7412.
Import S44 S61 S7414 S621.
Definition Random :=
  pair "gmt_unix_time" uint32
 (pair "random_bytes" (opaque \[ 28 \])
  typ_nil).
Definition SessionID :=
  opaque \< 0 \.. 32 \> 1.
Definition CipherSuite := uint8 \[ 2 \].
Definition cipher_suites_type :=
  CipherSuite \< 2 \.. (2^16 - 2) \> 2.

Definition compression_methods_type :=
CompressionMethod \< 1 \.. (2^8 - 1) \> 1.
Definition extensions_type :=
  Extension_type \< 0 \.. (2^16 - 1) \> 2.
Definition Hello_size (sid : nat) := ...
Definition ClientHello_size
  (sid cys cpm : nat) := ...
Definition client_extensions_present
  n (sid cys cpm : nat) :=
  (ClientHello_size sid cys cpm) <
   (Z_of_nat n).
Record ClientHello {n}
    (Hn : decodep uint24 n) := {
  client_version : list byte;
  _ : decodep ProtocolVersion
      client_version ;
  random : list byte ;
  _ : decodep Random random ;
  session_id : list byte ;
  _ : decodep SessionID session_id ;
  cipher_suites : list byte ;
  _ : decodep cipher_suites_type
      cipher_suites ;
  compression_methods : list byte ;
  _ : decodep compression_methods_type
      compression_methods ;
  extensions : list byte ;
  _ : (selectb( client_extensions_present
      (S41.bytes2valueN n)
      (length session_id)
      (length cipher_suites)
      (length compression_methods) \)  \{
        (true, decodep extensions_type) ;
        (false, fun x ⇒ length x == 0) \})
      extensions
}.
End S7412.
```

```
7.4.1.2.  Client Hello

struct { uint32 gmt_unix_time;
        opaque random_bytes[28]; } Random;


opaque SessionID<0..32>;

uint8 CipherSuite[2];


enum { null(0), (255) } CompressionMethod;



/* Extension_type comes from Sect. 7.4.1.4 */









struct {

    ProtocolVersion client_version;


    Random random;

    SessionID session_id;

    CipherSuite cipher_suites<2..2^16-2>;


    CompressionMethod compression_methods<1..2^8-1>;



    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };



} ClientHello;
```

Figure 4: Formalization of data structures in the RFC 5246, Coq on the left, RFC on the right

field “cipher” records the cipher suite elected by the server as the ClientHello packet is processed.

```
Definition ssl_session_typs :=
  ("cipher", btyp sint32) ::
  ("length", btyp sint32) ::
  ("id",     ptyp (btyp uchar)) :: nil.
Definition ssl_session := styp
  (mkStag "ssl_session") ssl_session_typs.
```

6.2 The ClientHello Parsing Program

We now comment on porting the PolarSSL function that parses ClientHello packets (function `ssl_parse_client_hello` from file `ssl_srv.c`, PolarSSL v.0.14.0) to our formal model of C.

The code in Fig. 5 illustrates porting. At the time of this writing, it is still performed manually: we adapt the original code to structured control-flow by replacing gotos with if-then-else's and by merging returns. Gotos can be handled using [9] (formalized in [14]) but, as a first step, structured control-flow is easier to verify because Hoare logic proofs are less verbose; this is also the choice made in other verification projects of C code (such as [13]). Strictly speaking, this is therefore not the original code that we are verifying, but yet, as it will be explained in Sect. 7, this is the code that we retrofit in the original application. Compared to the original function, it is close in structure and has the advantage of being formally verifiable.

C expressions are almost ported as they are. In particular, we benefit from our library about finite-size integers [5] to represent bit-wise operations. Yet, since expressions cannot have side-effects (as explained in Sect. 2.4), some of them need to be split into several commands using temporary variables.

6.3 Verification Goal

We want to prove that, given an appropriate initial state (as specified by the precondition, see Sect. 6.3.1 below for details), the program `ssl_parse_client_hello` either fails (by returning a non-zero value) or otherwise succeeds in (1) checking that the incoming ClientHello packet is valid, and (2) updating the state of the TLS server appropriately (this is detailed in Sect. 6.3.2 below). Here follows the skeleton for the formal statement:

```
Lemma POLAR_parse_client_hello_verif :
 ∀ SI BU RB ID CI,
 length BU = SSL_BUFFER_LEN →
 length RB = 64 → length ID = 32 →
 ∀ majv0 minv0 mmaj0 mmin0 cipher0
 length0 a rb ses id ciphers vssl,
 u2Z vssl + sizeof (styp (mkStag
  "ssl_context") ssl_context_typs) <
  2 ^ size_ptr →
 ...
 {{ (* precondition: see Sect. 6.3.1 *) }}
   ssl_parse_client_hello
 {{ fun s h ⇒ (∃ i,
  [ var_e "ret" ]_ s = Some (bval_int0_32)
    ∧ (* postcondition: see Sect. 6.3.2 *))
  ∨
  [ var_e "ret" ]_ s ≠ Some (bval_int0_32)
 }}.
```

The meaning of variables will be explained in the next sections. Among them, `SI` deserves earlier explanations because it is part of the mode. `SI` (for “socket input”) is a list of bytes that models the input stream; since there is no network interface in Separation logic, we resort to ad-hoc modeling using native Coq lists. We now explain the rest of the above statement.

6.3.1 Precondition

The precondition specifies the state of the program before executing `ssl_parse_client_hello`: the type context (from line 2 below), the initial state of the local variables (their types from line 5, their values from line 10), and the initial state of the memory.

In particular, the initial state of the memory is specified by a Separation logic formula (starting from line 12). It is really just the formalization of Fig. 6, that represents graphically the PolarSSL data structures introduced in Sect. 6.1. Besides pointers, most fields are uninitialized except for the state of the protocol in which the server is: here, it is waiting for a ClientHello packet, hence the initialization to `S74.client_hello`. Among the various storage spaces set up by the server, the array `BU` is the most sensitive: it is a buffer large enough to welcome that part of the input stream `SI` corresponding to the ClientHello packet. In the following, the notation

```
Definition ssl_parse_client_hello :
 @while.cmd cmd0 bexp := (
 ssl_fetch_input "ret" "ssl"
   (cst32 (Z2s _ 5)) ;
 while.ifte (exp2bexp
   (var_e "ret" ≠ cst32_0))
 ret
 (

 "buf" ←* var_e "ssl" .→ "in_hdr" ;

 "_buf0_" ←* var_e "buf" ;
 while.ifte (exp2bexp
  ((var_e "_buf0_" && cst8 (Z2s 8 (-128)))
   ≠ cst8 (Z2s 8 0)))
 ( "ret" ←
    POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ;
  ret
 )
 (
 ... )).
```

```
static int ssl_parse_client_hello( ssl_context *ssl )
{
    int ret, i, j, n;
    int ciph_len, sess_len;
    int chal_len, comp_len;
    unsigned char *buf, *p;

    SSL_DEBUG_MSG( 2, ( "=> parse client hello" ) );

    if( ( ret = ssl_fetch_input( ssl, 5 ) ) != 0 )
    {
        SSL_DEBUG_RET( 1, "ssl_fetch_input", ret );
        return( ret );
    }

    buf = ssl->in_hdr;

    if( ( buf[0] & 0x80 ) != 0 )
    {


...
```

Figure 5: ClientHello Parsing (formal model on the left, original code on the right)

"$\vdash^a \cdot \rightarrow$" is a generalization for arrays of the map-sto formula (Sect. 3.1), and we have omitted the `wft` proofs to ease reading.

```
fun s h ⇒ wf_tstore (_tstore s) ∧
 wf_tenv (cstore_tenv s) ∧
 s ⊢^g "ssl_context" ⊣ ssl_context_typs ∧
 s ⊢^g "ssl_session" ⊣ ssl_session_typs ∧
 ...
 s ⊢^t "_buf0_" ⊣ btyp uchar ∧
 s ⊢^t "buf" ⊣ ptyp (btyp uchar) ∧
 s ⊢^t "ret" ⊣ btyp sint32 ∧
 s ⊢^t "ssl" ⊣ ptyp ssl_context ∧
 ...
 s ⊢^v "ssl" ⊣ pval vssl ∧
 ...
 ((cst_pe _ a ⊢^a uchar → map cst8 BU) ⋆
  (cst_pe _ rb ⊢^a uchar → map cst8 RB) ⋆
  (cst_pe _ id ⊢^a uchar → map cst8 ID) ⋆
  (cst_pe _ ses ⊢ ssl_session →
   (bval_int cipher0 :: bval_int length0
    :: pval id :: nil) CST_SE _) ⋆
  (cst_pe _ ciphers ⊢^a uint32 →
   map cst32 CI) ⋆ TT ⋆
 (var_e "ssl" ⊢ ssl_context →
  (bval_int (Z2u 32 S74.client_hello) ::
  bval_int majv0 ::
  bval_int minv0 ::
  bval_int mmaj0 ::
  bval_int mmin0 ::
  pval ses ::
  pval (a [.+] Z2u size_ptr 8) ::
  pval (a [.+] Z2u size_ptr 13) ::
  bval_int 0_32 ::
  pval ciphers ::
  pval rb :: nil) CST_SE _) s h
```

6.3.2 Postcondition

We now explain the postcondition in the case where `ssl_parse_client_hello` returns 0, meaning it has correctly parsed the incoming ClientHello packet. For the purpose of explanation, we split the predicate in two: one part that specifies that the incoming ClientHello packet is valid (Sect. 6.3.2) and one part that specifies the state of the memory after parsing (Sect. 6.3.2).

ClientHello Packet Validity Checking the validity of the incoming ClientHello packet amounts to check that:

- the various lengths that are embedded into the packet (the length of the Handshake, the length of the ClientHello) are consistent between each

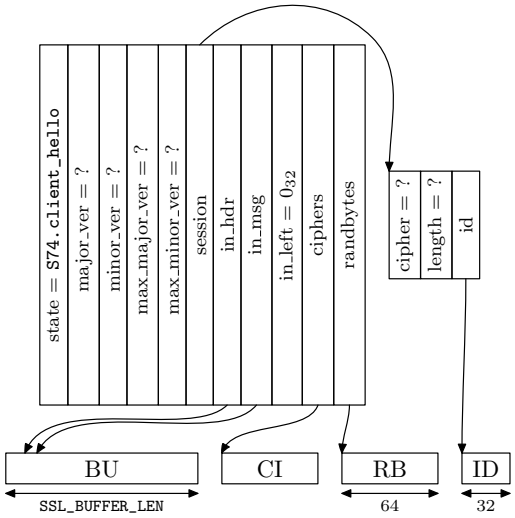

Figure 6: State of the Memory before Parsing

other (e.g., the length of the Handshake must be larger that the length of the ClientHello that is embeds—line 2 below), and

- the various encoded values conform to the RFC (e.g., the length of the Handshake packet is bounded as defined in the RFC—line 3 below).

Comments should suffice to understand the validity constraints. The identifiers (`n`, `m`, etc.) are the same as Fig. 3. Definitions from the RFC are preceded by their section number, what makes it easy to find them in [6].

```
1 (* constraints about encoded lengths *)
2 S74.min_Handshake + n SI ≤ m SI ∧
3 m SI ≤ S621.length_max ∧
4 R SI ≤ tls_max S7412.SessionID ∧
5 tls_min S7412.cipher_suites_type ≤ S SI
6  ≤ tls_max S7412.cipher_suites_type ∧
7 tls_min S7412.compression_methods_type ≤
8 T SI ≤
9 tls_max S7412.compression_methods_type ∧
10 (* the null compression is proposed *)
11 In (Z2u 8 S61.null) (SI
12 |[ comp_idx + 1 + R SI + S SI, T SI) ) ∧
13 (* no client extension *)
14 ¬ (S7412.client_extensions_present
15     (n SI) (R SI) (S SI) (T SI)) ∧
16 (* major/minor ver. are supported *)
17 u2Z (nth maj_ver SI 0₈) =
18  S621.SSLv30_maj ∧
19 u2Z (nth req_maj SI 0₈) =
20  S621.SSLv30_maj ∧
21 In (u2Z (nth min_ver SI 0₈))
22  (S621.SSLv30_min :: S621.TLSv10_min ::
23  nil) ∧
24 In (u2Z (nth req_min SI 0₈))
25  (S621.SSLv30_min :: S621.TLSv10_min ::
26  nil) ∧
27 (* there is a candidate ciper suite *)
28 even (S SI) ∧
29 i < div2 (S SI) ∧
30 (∃ k, zext 16
31 (nth (comp_idx + R SI + 2 * i) SI 0₈ ||
32  nth (comp_idx + R SI + 2 * i + 1) SI 0₈)
33  = nth k CI0₃₂) ∧ (* continues in Sect. 6.3.2 *)
```

The notation "`A` |[`a`, `n`)" represents the sub-list of `A` that start at index `a` index and consists of `n` elements. `zext` stands for zero-extension of finite-size integers.

Memory after Parsing After parsing, the memory has been initialized with the result of the TLS negotiation. Like the precondition, this is also captured by a Separation logic formula (displayed below). Fig. 7 provides an equivalent pictorial representation that can be compared with the initial memory state (Fig. 6).

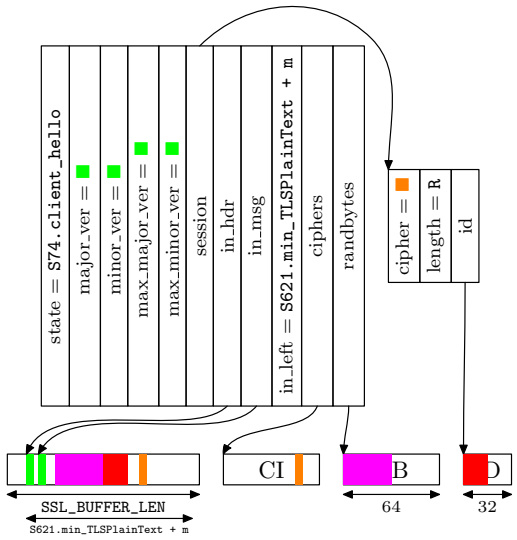

Figure 7: State of the Memory after Parsing

For example, one can observe that the `ssl_context`

data structure now specifies that the server is in the `S74.server_hello` state of the TLS protocol, meaning it is about to send back to the client a ServerHello packet. The array `BU` is now filled with a ClientHello packet (like the one of Fig. 3) and most of its contents are duplicated in PolarSSL data structures. For example, the array for random bytes `RB` has been half-filled with that part of the incoming ClientHello packet supposed to serve as the client nonce.

The following Separation logic formula formalizes Fig. 7. To draw a parallel with the informal explanations above, one can look at the specification for the array for randombytes `RB` at line 5.

```
1 (* ClientHello is in the buffer *)
2 (BU |[8, S621.min_TLSPlainText + m SI)) =
3 (SI |[0, S621.min_TLSPlainText + m SI)) ∧
4 (* the client nonce is stored *)
5 ((cst_pe _ rb ⊢a uchar →
6   map cst8
7    (SI |[ rand_idx, tls_max S7412.Random))
8  ++ map cst8
9    (skipn (tls_max S7412.Random) RB)) ⋆
10 (* the session id is stored *)
11 (cst_pe _ id ⊢a uchar →
12   map cst8 (SI |[ idx_sid, R SI) ))  ⋆
13 (* the cipher suite is fixed *)
14 (cst_pe _ ses ⊢ ssl_session →
15  (bval_int (zext 16
16   (nth (comp_idx + R SI + 2 ⋆ i) SI 08 ||
17    nth (comp_idx + R SI + 2 ⋆ i + 1) SI
18    08)) :: bval_int (Z2u 32 (R SI)) ::
19  pval id :: nil) CST_SE _) ⋆ TT ⋆
20 (* the ssl_context is initialized *)
21 (var_e "ssl" ⊢ ssl_context →
22  (bval_int (Z2u 32 S74.server_hello) ::
23  bval_int (zext 24 (nth maj_ver SI 08)) ::
24  bval_int (zext 24 (nth min_ver SI 08)) ::
25  bval_int (zext 24 (nth req_maj SI 08)) ::
26  bval_int (zext 24 (nth req_min SI 08)) ::
27  pval ses ::
28  pval (a [.+] Z2u size_ptr 8) ::
29  pval (a [.+] Z2u size_ptr 13) ::
30  bval_int (Z2u 32
31   (S621.min_TLSPlainText + m SI)) ::
32  pval ciphers ::
33  pval rb :: nil) CST_SE _) s h)
```

6.3.3 Current Status

At the time of this writing, we have verified the first few lines of `ssl_parse_client_hello` (see [15]). This can be seen as a small start but it is already quite substantial because we have spent much time producing reusable lemmas for verification while adjusting the formal model. We expect the effort to reduce as verification progresses.

`ssl_parse_client_hello` calls other library functions that do, for example, cryptographic computations. We do not plan at first to formally verified them; for the time being, we axiomatize their correctness. This is for example the case of `ssl_fetch_input` (see Fig. 5), a function that reads bytes from the input stream (modeled by the list `SI`) and fill the buffer `BU` with them.

7 From the C Model to Runnable Code

7.1 Pretty-printing

Translation from the C model to compilable code amounts to pretty-printing of the abstract syntax tree of formal programs. There is a pretty-printing function for each inductive type: variables `var_e x` embed a string `x` that is pretty-printed as such, the function `pp_exp` pretty-prints C expressions (Sect. 2.2), and so on. For example, here follows the function `pp_cmd` that pretty-prints C commands (Sect. 2.5):

```
Fixpoint pp_cmd c acc : string :=
match c with
| while.seq c1 c2 ⇒
  pp_cmd c1 (" " ++  pp_cmd c2 acc)
| while.ifte e c1 c2 ⇒
  "if (" ++ pp_bexp e (") { " ++
    pp_cmd c1 (
  " } else { " ++ (
    pp_cmd c2 (
  " }" ++ acc))))
| while.while e c ⇒
  "while (" ++ pp_bexp e (") { "
    ++ pp_cmd c (
  " }" ++ acc))
| skip ⇒ ";" ++ acc
| assign v e ⇒ pp_var v (" = " ++
  (pp_exp e (";" ++ acc)))
| lookup v e ⇒ pp_var v (" = *(" ++
  (pp_exp e (");" ++ acc)))
| mutation e1 e2 ⇒
  "*(" ++ pp_exp e1 (") = " ++
    (pp_exp e2 (";" ++ acc)))
| malloc v e ⇒ pp_var v (" = malloc("
```

```
  ++ pp_exp e (");" ++ acc))
| free e ⇒
  "free(" ++ pp_exp e (");" ++ acc)
end.
```

The `acc` parameter is an accumulator to which strings are prepended as they are pretty-printed. This choice of implementation comes from the fact that append (++) on Coq `string`s (that is in fact the type `list ascii`, where `ascii` represents characters) is costly: depth-first traversal of the abstract syntax tree together with prepending at least makes it possible to perform pretty-printing inside Coq.

Here is how we apply pretty-printing to `ssl_parse_client_hello` (the program of Sect. 6):

```
Goal pp_cmd ssl_parse_client_hello "" =
  "TRANSLATED C CODE STRING".
Proof.
rewrite //= !pp_cmd_ssl_fetch_input
 !pp_cmd_md5_update
 !pp_cmd_sha1_update
 !pp_exp_uchar_to_int
 !pp_cmd_memcpy
 !pp_cmd_memset
 //= /pp_sint /pp_binop /zero32 /one32
 !s2Z_Z2s //=
Qed.
```

Pretty-printing is achieved by rewriting instead of evaluation because of performance issues. Note that axiomatized functions (`ssl_fetch_input`, etc., discussed in Sect. 6.3.3) are pretty-printed in an ad-hoc manner.

7.2 Retrofitting

The string obtained by pretty-printing can be retrofitted to the original source code simply by copy-pasting. We have experimented with a few functions (`ssl_fetch_input`, `asn1_get_len`) of PolarSSL [7] and confirmed by recompiling and running the new program against an OpenSSL [8] client that the PolarSSL server still behaves as expected. This kind of experiment is by no way a verification but helps to confirm that the formal model of a C function has not been altered by porting to Coq (as explained in Sect. 6, porting is manual and calls for a few modifications of the source code).

8 Related Work

[11] proposes another formalization of Separation logic for C, with an application to a memory allocator. The work is done in the Isabelle proof-assistant and is arguably more complete than ours. Our use-case (the PolarSSL [7] implementation of the TLS protocol) is different and led us to work on the issue of formalization of Internet RFCs. Our technical development differs also in several ways. For example, it differs at the level of the definition of C types ([11] only distinguishes between scalar and aggregate types whereas we reflect basic types) or at the level of pointer arithmetic ([11] favors a variant of the Burstall-Bornat model for heap access whereas we stick to direct, byte-level accesses). Thus saying, we end up proving similar results (compare for example the corollary of Theorem 7.5 in [11] with the lemma `mapsto_styp_inv` in Sect. 3.2). We believe that there is value in providing another Separation logic based on different premises but the original reason why we worked out our own version is because of our application: besides parsing functions, cryptography is another sensitive point of the implementation of communication protocols and we plan in the mid-term to go on verifying cryptography, for which we plan to reuse our previous work in Coq [5, 14].

[12] provides another model of C in the Coq proof-assistant, again more complete than ours but without Separation logic. Again, technical developments differ in several ways. To avoid dealing with a type context, [12] chooses a "structural" encoding for struct: an enclosing struct can always be referred to by using "indices" so as to enable the definition of recursive types.

[13] proposes a different approach to the problem of formal verification of C programs, based on simulation with a functional program. There is no Separation logic per se, but Hoare logic is used to establish simulations (see Sect. 5.2 in [13]). Application of this approach to PolarSSL would require the construction of a reference implementation, what would be another way to formalize TLS RFC, to be contrasted with our formalization in Sect. 5.

9 Conclusion

We have introduced a formal model for a subset of C in the Coq proof-assistant. It is faithful enough so that formal C programs can be retrofitted to their original application by pretty-printing. Our formal model of C is equipped with a Separation logic, that differs from the original Separation logic mainly because of the presence of C types. We have applied this model to the formal verification of the standard in-place list reversal and we are now tackling the formal verification of a parsing function taken from an existing implementation of the TLS protocol. The formal specification relies of a formalization of a part of the corresponding IETF RFC.

Current and Future Work Formal verification of the `ssl_parse_client_hello` function from PolarSSL [7] is still in progress. In parallel, we are preparing for formal verification of basic functions from the ASN.1 parser of PolarSSL (the ASN.1 parser turned out to be a recurrent source of bugs for OpenSSL [8]). At this stage of our project, there are still many ways to improve our model of C and its Separation logic. We plan in priority to work on compliance with the C standard (in particular, portability issues) and to work on interface with assembly (so as to verify those parts of the implementation of communication protocols that use assembly for cryptography).

Acknowledgments Masashi Kikuchi contributed to improve the C model and to the formal verification. This work is supported by the project "Security Evaluation for Communication Protocols and their Implementations" (National Institute for Information and Communications Technology). The first author also acknowledges support from Kakenhi 21700048.

References

[1] The Coq Proof Assistant. INRIA. 1984–2011. `http://coq.inria.fr`.

[2] Harbison, S.P., Steele, G.L: C: A Reference Manual. 5th edition. Prentice Hall, 2002

[3] Reynolds, J.C.: A Logic for Shared Mutable Data Structures. Proc. of the 17th IEEE Symp. on Logic in Computer Science (LICS 2002), pp.55–74. IEEE, 2002.

[4] Marti, N., Affeldt, R., Yonezawa, A.: Formal Verification of the Heap Manager of an Operating System using Separation Logic. Proc. of the 8th International Conference on Formal Engineering Methods (ICFEM 2006), vol. 4260 of Lecture Notes in Computer Science, pp.400–419. Springer, October 2006.

[5] Affeldt, R., Marti, N.: An Approach to Formal Verification of Arithmetic Functions in Assembly. Proc. of the 11th Annual Asian Computing Science Conference (ASIAN 2006), vol. 4435 of Lecture Notes in Computer Science, pp.346–360. Springer, January 2008.

[6] Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246. IETF, 2008.

[7] PolarSSL. Open Source embedded SSL/TLS cryptographic library. `http://polarssl.org`.

[8] OpenSSL. Open Source toolkit for SSL/TLS. `http://www.openssl.org`.

[9] Saabas, A., Uustalu, T.: A compositional natural semantics and Hoare logic for low-level languages. *Theor. Comput. Sci.*, vol. 373(3), pp.273–302. Elsevier, 2007.

[10] Marti, N., Affeldt, R.: A Certified Verifier for a Fragment of Separation Logic. *Computer Software*, vol. 25(3), pp.135–147. Iwanami Shoten, 2008.

[11] Tuch, H.: Formal Verification of C Systems Code. *J. Autom. Reasoning*, vol. 42(2–4), pp.125–187. Springer, 2009.

[12] Blazy, S., Leroy, X.: Mechanized Semantics for the Clight Subset of the C Language. *J. Autom. Reasoning*, vol. 43(3), pp.263–288. Springer, 2009.

[13] Winwood, S., Klein, G., Sewell, T., Andronick, J., Cock, D., Norrish M.: Mind the Gap. Proc. of

the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009), vol. 5674 of Lecture Notes in Computer Science, pp.500–515. Springer, 2009.

[14] Affeldt, R., Nowak. D., Yamada, K.:Certifying Assembly with Formal Cryptographic Proofs: the Case of BBS. *Science of Computer Programming*, in press. Elsevier, 2011. `http://dx.doi.org/10.1016/j.scico.2011.07.003`.

[15] Affeldt, R. A Library for Formal Verification of Low-level Programs. Coq documentation. `http://staff.aist.go.jp/reynald.affeldt/coqdev`.