A Coq Starter Kit to Verify TLS Packet Processing in C*

Reynald Affeldt[1] and Kiyoshi Yamada[2]

[1] National Institute of Advanced Industrial Science and Technology, Japan
[2] Lepidum Co., Ltd., Tokyo, Japan

Abstract. TLS is such a widespread security protocol that errors in its implementation can have disastrous consequences. This heavy responsibility is mostly borne by programmers who are almost left to themselves, caught between error-prone low-level programming with C and specifications with the ambiguities of natural language. Our purpose is to provide a Coq framework for the formal verification of TLS packet processing written in C. First, we provide a new library for C verification based on Separation logic. This library features a simple encoding of C types that makes for easy and faithful modeling. Second, we introduce a formalization of the RFC for TLS that improves on the original document by making prose statements palpable and even spotting errors. Last, we investigate application to an existing implementation of TLS from which we extract, specify and start verification of a parsing function, such functions being a notorious source of security bugs.

1 Introduction

TLS (Transport Layer Security) [7] is such a widespread security protocol that errors in its implementation can have disastrous consequences. This heavy responsibility is mostly borne by programmers who are almost left to themselves, caught between error-prone low-level programming with the C programming language [1] and specifications with the ambiguities of natural language.

We want to use formal verification with a proof-assistant to improve the implementations of TLS. There exist several ways to use proof-assistant technologies to improve the implementations of communication protocols in general. In [4], the authors develop an HOL specification of TCP against which they test existing implementations of the Socket API; this is effective but lets open the question of the C source code adequacy to the programmer's intent. In [14], the author proposes to use a dependently-typed programming language to specify and verify network packet processing; unfortunately such implementations continue to be developed in C for performance reasons.

* This is a revised version of a non-reviewed paper that was presented at the 28th Workshop of the Japan Society for Software Science and Technology (JSSST 2011) (`http://jssst11.kuis.kyoto-u.ac.jp/`).

Our purpose here is to provide a framework in the Coq proof-assistant [13] for the interactive verification, at the source code level, of C programs that process TLS packets.

As the main element of this framework, we provide a new library for C verification based on Separation logic [3], a variant of Hoare logic that deals with pointers, the latter being heavily used in network packet processing. The originality is a simple encoding of C aggregated types (hereafter, C structs). This is a faithful model of C so that existing code can be ported in a systematic way and so that formal models can be readily pretty-printed and compiled, thus reducing the trusted base to a minimum. Our Separation logic is equipped with the expected standard lemmas such as the frame rule and tested against the standard in-place list reversal example. We introduce our formal model of C in Sect. 2 and its Separation logic in Sect. 3.

The task of processing network packets is disciplined by various RFCs that describe in a semi-formal fashion the format of the network packets. In order not to depart from common practice, we insist on having a formalization of the RFC for TLS that can be syntactically compared with the original document [7]. This not only gives us formal grounds to lay down specifications of the C source code, but this also has the side-effect of improving the original RFC by eliminating prose-only statements and even spotting errors. This is explained in Sect. 4.

Finally, we investigate in Sect. 5 application to an existing implementation of TLS, namely PolarSSL [8]. Concretely, we port the function that parses initialization packets, specify it w.r.t. the formal RFC, and, as a first step, verify the first part of this function, that parses the packet header. It is interesting to note that even recent security bugs can be found in such well-scrutinized functions (e.g., CVE-2011-0014 for ClientHello in OpenSSL [9]). Upon completion, this verification will provide PolarSSL with advanced debugging and certification.

2 Formal Model of C

2.1 An Encoding of C Types Parametrized by a Type Context

We first define a set of integral types: unsigned and signed 32-bit integers, and unsigned 8-bit characters: `Inductive ityp : Type := uint32 | sint32 | uchar.` C types are then defined as follows:

```
Inductive stag := mkStag : string → stag. (* struct tags *)
Inductive typ : Type :=
| btyp of ityp (* basic integral types *)
| ptyp of typ (* pointer types *)
| rtyp of stag
| styp of stag & list (string * typ). (* struct types *)
```

`styp` corresponds to C structs, fields' names being encoded as strings. Like `ptyp`, `rtyp` corresponds to pointers; intuitively, `rtyp tg` is the same as `ptyp (styp tg l)` if `l` is associated with `tg` in the current context. This alternative way to write pointers allows for the definition of recursive structures. For example, singly-linked lists are defined as follows:

```
Definition C_lst_flds := ("data", btyp uint32) ::
 ("next", rtyp (mkStag "C_lst")) :: nil.
Definition C_lst := styp (mkStag "C_lst") C_lst_flds.
```

Not all types allowed by the above syntax are proper C types; the predicate `wft` forbids empty structs and structs with homonymous fields.

We define a contextual equality to cope with the double representation of pointers. The first step is a predicate `a =t= b` that holds when `a` and `b` are syntactically the same type (it treats `rtag tg` like `ptyp (styp tg l)` for any `l`). Then the desired contextual equality is the predicate `t1 =t c t= t2`, where `c` is a type context, defined as follows:

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

`cover c t` is a predicate that holds when all the tags in the type `t` appear in (the domain of) `c`.

`sizeof` is an important function for (un)marshalling data structures. As its C namesake, it computes the number of bytes needed to put data in memory. This requires to model the size of pointers. Let `ptr_size` be the number of bytes needed to encode a pointer. It is a parameter of our model and its properties allow for 32, 64, etc. architectures:

```
Parameter ptr_len : nat. Parameter ptr_size : nat.
Parameter Hptr_size : ptr_len = ptr_size * 8.
```

Ignoring padding issues, the `sizeof` function is defined as follows:

```
Definition sizeof_i 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 ⇒ sizeof_i x
   | ptyp _ ⇒ ptr_size
   | rtyp _ ⇒ ptr_size
   | styp _ l' ⇒ iplus (map (sizeof' m) (uzip2 l'))
  end end.
Definition sizeof t := sizeof' (typ_max_depth t) t.
```

`sizeof'` takes a size argument `n` that is expected to be the maximal depth of `t`: this is standard practice to show Coq that a function terminates.

2.2 C Expressions and their Evaluation

Values are made of finite-size integers. In particular, pointers are only known to be of size `ptr_len`:

```
Inductive value : Type := bval32 of int 32 | bval8 of int 8
  | pval of int ptr_len | sval of list value.
```

We model a subset of the expression language of C as follows:

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

`cst32`, `cst8` are for (signed) integral constants; they can be constructed using the function `Z2s` that builds finite-size integers (to be seen as signed). `cst_pe` is for pointer constants; they come with a proof of wellformedness of the pointed type. `cst_se` is for struct constants; they come with a proof of wellformedness of the type and a proof that the list of values `vs` is *compatible* (in the sense of `typ_val`). When computable, these proofs are hidden by notations. `fld` and `fld'` are for accessing the fields of structs: `fld` corresponds to the "." notation of C and `fld'` corresponds to the "→" notation (more precisely, `fld' p f`, noted `p ↪ f` here, would be written `&(p →f)` in C). `bop_ne` is for various binary operators. `add_pe` is for pointer arithmetic.

We now explain the evaluation of C expressions. Evaluation has to deal with types because of pointer arithmetic. Let us assume a type context of type `ctxt` and a typing environment of type `tenv`. We first define a typing function:

```
Definition tenv := list (var * typ).
Fixpoint typ_of (c : ctxt) (env : tenv) e : option typ := ...
```

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

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

Since `tstore` does not prevent ill-typed values, evaluation is actually defined w.r.t. a `store` 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 type.

The following excerpt of the evaluation function illustrates pointer arithmetic and also how to deal with the double representation of pointers. Concretely, when it runs into `rtyp tg`, evaluation looks for the set of fields/types corresponding to the tag `tg` in the type context (line 10 below).

```
0 Definition typof (s : store) e := typ_of (_ctxt s) (store_tenv s) e.
1 Fixpoint eval (e : exp) (s : store) : option value := ...
2 | add_pe e1 e2 ⇒
3  match [ e1 ]_ s, [ e2 ]_ s with
4  | Some (pval i1), Some (bval32 i2) ⇒
5   match typof s e2 with | Some (btyp sint32) ⇒
6    match typof s e1 with
```

```
    | Some (ptyp t) ⇒
        Some (pval (scalez i1 (sizeof t) (s2Z i2)))
    | Some (rtyp tg) ⇒
      match assoc_get tg (_ctxt s) with
      | Some l ⇒
        Some (pval (scalez i1 (sizeof (styp tg l)) (s2Z i2)))
      | None ⇒ None
      end
... where "'[' e ']_' s" := (eval e s).
```

`scalez p i k` means to add `k`×`i` to `p` and `s2Z` interprets a finite-size integer as a signed integer. As often when reasoning interactively about imperative programs (e.g., [10], Sect. 3.1), the evaluation of expressions does not perform read/write side-effects to the heap.

2.3 Semantics of C Commands

To produce the formal model of C commands, we use an existing Coq module [15]. Given a syntax, operational semantics, Hoare triples, and basic properties for a set of one-step commands, it generates a syntax, operational semantics and a sound Hoare logic for the corresponding WHILE-language (i.e., with structured control-flow). We use the following set of one-step commands to define our subset of C:

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

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

The important difference between C and an archetypal language such as the one of Separation logic [3] is that memory is accessed by blocks, whose length is determined by the type of read/written data. For illustration, the operational semantics of lookup is defined as follows:

```
Reserved Notation " s '–' c '⟶' t ".
Inductive exec0 : option state→cmd0→option state→Prop := ...
| exec0_lookup : ∀ s h tx x e v p,
  typof s (var_e x) =ot _ctxt s ot= tx →
  typof s e =ot _ctxt s ot= ptyp tx →
  [ e ]_ s = Some (pval p) → heap_get (u2Z p) tx h = Some v →
  Some (s, h) – x ←* e ⟶ Some (store_upd x v s, h)
```

`u2Z` interprets a finite-size integer as an unsigned integer (`Z2u` performs the converse operation); `heap_get a tx h` turns the `sizeof tx` bytes starting at address

`a` into a value; "· `=ot` · `ot=` ·" is the same as "· `=t` · `t=` ·" when the left hand-side is a `Some`, and `false` otherwise. The semantics that we define therefore enforces type checking, so that, say, lookup executes only when the type of the variable and of the dereferenced expression agree (lines 3–4 above). Programs that deviate from this behavior require adjustments to be modeled, what benefits anyway to clarity and therefore security.

We have implemented in Coq a set of pretty-printing functions to translate programs in our C model to compilable code. Since the Coq evaluation engine is not optimized for that purpose, it is important to find an efficient way of implementing pretty-printing: depth-first traversal of the abstract syntax tree together with state-passing does the trick. The string obtained by pretty-printing can be retrofitted to the original application by copy-pasting. We have experimented with a few functions (`ssl_parse_client_hello`, `asn1_get_len`) of PolarSSL [8] and confirmed by running the new program against an OpenSSL [9] client that the PolarSSL server still behaves as expected.

3 A Typed Extension of Separation Logic for C

The definition of a Separation logic for our C model essentially amounts to provide the Separation logic assertions and to derive Separation logic-specific lemmas (the frame rule and so on). We *shallow-embed* assertions, i.e., they are functions of type `Definition assert := store →hp.t →Prop`. This is a standard approach; see, e.g., [5] for an illustration of this technique. Here, we only focus on the *mapsto* connective, since it departs from textbook Separation logic because of C types.

3.1 The Typed Mapsto Connective

The mapsto connective specifies singleton heaps. In the textbook Separation logic mapsto connective, a singleton heap corresponding to address a and contents b (where both a and b are integers) is specified by $e_1 \mapsto e_2$, where e_1 and e_2 evaluate respectively to a and b. We extend the mapsto connective with types to account for the various data structures of C. This typed mapsto connective is noted $e \overset{t}{\mapsto} e'$ where 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 block that contains the encoding of v. 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 the type `t`. The notation "$\vdash^a \cdot \rightarrow$" (to be used in Sect. 5) is for a generalization of the mapsto connective to C arrays.

Once connectives are defined, lemmas must be proved so as to facilitate the task of formal verification. In the case of the typed mapsto connective, this leads to original lemmas. Consider for example reading fields' contents of heap-allocated structs. Let us suppose that `x` points to a struct of type `styp tg l`, what could 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, when performing a lookup, to derive that `(var_e x↪f ⊢ t →k ⋆ TT) s h`, where $\star$ is the separating conjunction and `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 ^ ptr_len →
 ∀ i f t t' vi k,
  assoc_get f l = Some  t' → t =t _ctxt s t= t' →
  ifind f (uzip1 l) = Some i → ith i vs = Some vi →
  val2cst (_ctxt s) t vi = Some k →
  (var_e x ↪ f ⊢ t → k ⋆ TT) s h.
```

3.2 Standard Example: In-place List Reversal

In-place list reversal operates on singly-linked lists as defined in Sect. 2.1:

```
Definition NULL : exp := cst_pe wf_C_lst (Z2u ptr_len 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`, pointed to by variable `i` before execution and pointed to by variable `ret` after. This is specified as follows:

```
0 Lemma reverse_list_verif : ∀ l,
1 {{ fun s h ⇒ (wf_tstore (_tstore s) ∧
2   s ⊢^g "C_lst" ⊣ C_lst_flds ∧ s ⊢^t rem ⊣ ptyp C_lst ∧
3   s ⊢^t i ⊣ ptyp C_lst ∧ s ⊢^t ret ⊣ ptyp C_lst) ∧
4   pointed_list l i s h }}
5  reverse_list
6 {{ pointed_list (rev l) ret }}.
```

Line 1 is a wellformedness condition on the store of variables. Line 2 means that the type of singly-linked lists belongs to the type context. Lines 2–3 give the type of the variables. We have been able to complete the formal verification without appealing to any axioms. Unsurprisingly, the proof script is complicated by intricate byte-level manipulations for which more lemmas are still to be found.

4 Formal Specification of TLS Packets

The description of packet formats in the RFC for TLS is semi-formal. A dedicated syntax (the *presentation language*) is introduced but its use is not entirely consistent throughout the document, and many conditions are still only described in natural language. On the one hand, it is necessary to formalize the description of packet formats to be able to write a formal specification for parsing functions, but, on the other hand, RFCs proved themselves useful despite their defects. Therefore, we insist on just improving the RFC with formal artifacts that can be related convincingly to their informal counterparts. Concretely, we provide a Coq encoding of a subset of the presentation language, resorting to shallow-embedding when packet formats are more naturally represented this way. The result is a formalization of packet formats that can be syntactically compared with the RFC.

4.1 An Encoding of The TLS Presentation Language

The TLS presentation language ([7], Sect. 4) consists of the following datatypes:

1. `opaque` is the type of bytes.
2. `T T'[n]` defines the type `T'` of *fixed-length vectors* made of `n` bytes, where `n` is a multiple of the size of `T`.
3. `T T'<a..b>` defines the type `T'` of *variable-length vectors*. They consist of a payload, whose length lies between `a` and `b` and that encodes data structures of type `T`, and a header (the "length field") that is large enough (but no larger) to encode the length of the payload.
4. `enum { e`$_1$`(v`$_1$`), ..., e`$_n$`(v`$_n$`) [[, (m)]] } T` defines the *enumerated* type `T`. The length of the payload must be sufficient to encode the largest value (one of `v`$_i$ or `m`). This payload is preceded by a "length field", like variable-length vectors.
5. Structure types are defined as being close to C structs but in fact they are closer to dependent records (e.g., `TLSPlaintext` in Sect. 4.2).
6. *Variants* extend structures with fields whose type depends "on some knowledge that is available within the environment" ([7], Sect. 4.6.1). This "knowledge" is the value of an enumerated that can come from preceding fields in the structure (e.g., the `body` field of `Handshake`, Sect. 7.4 of [7]) (in which case we are dealing with a dependent record) or from the (implicit) environment (e.g., the "length field" of the enclosing Handshake packet in the case of `ClientHello`, Sect. 7.4.1.2 of [7]).

Putting dependent records aside, we encode the presentation language using the `tls_typ` inductive type. Since it is important for bound-checking in parsing functions, we give `tls_typ` the minimum and maximum size of the underlying list of bytes as parameters. We use dependent types to figure out the "length field" of variable-length vectors and enumerateds, and to check divisibility constraints on fixed-length vectors: these proof obligations can be inferred automatically and hidden using notations.

```
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.
```

This formalization of the representation language led us to spot errors in the RFC. Here is a concrete example. Sect. 7.4.1.4 of [7] defines the `Extension` type has follows (using `tls_typ`s with notations):

```
Definition signature_algorithm := 13.
Definition ExtensionType :=
  \enum 2 \{ signature_algorithm :: nil \} 65535.
Definition extension_data_type := opaque \< 0 \.. 2^16 - 1 \> 2.
Definition Extension :=
  struct{ ("extension_type", ExtensionType) ;
          ("extension_data", extension_data_type) }.
```

This type is used in Sect. 7.4.1.2 of [7] to define the type of the `extensions` field of a ClientHello packet:

```
Definition extensions_type := Extension \< 0 \.. 2^16 - 1 \> 2.
```

This is ruled out automatically by type-checking because the maximum size of `extensions_type` is the same as `extension_data_type`, whereas they should be nested in a strict fashion. Another example of erroneous specification is about the length of variable-length vectors. According to Sect. 4.3 of [7], 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_type`.

4.2 Dealing with Dependent Records in Packet Formats

It is not easy to encode as an inductive type structure types that are dependent records. In such situations, we resort to shallow-embedding using Coq dependent records. For this purpose, we introduce a generic decoding function for `tls_typ`:

```
Fixpoint decode n{a b}(t:tls_typ a b) lst:bool*list byte:= ...
Definition decoder {a b}(t:tls_typ a b):=decode (depth t) t.
Definition decodep {a b} (t : tls_typ a b) lst :=
 let (ret, lst'):= decoder t lst in ret && (length lst' == 0).
```

We also introduce the type `packet p` of lists of bytes that satisfy the predicate `p`, where `p` is typically a decoding function:

```
Record packet (p : list byte → bool) :=
 { body :> list byte ; Hp : p body }.
```

As an example, let us consider the definition of `TLSPlainText` (Sect. 6.2.1 of [7]):

```
Definition change_cipher_spec := 20. Definition alert := 21.
Definition handshake := 22. Definition application_data := 23.
Definition ContentType := \enum 1 \{ change_cipher_spec ::
  alert :: handshake :: application_data :: nil \} 255.
Definition length_maxp x := S41.bytes2valueN x ≤ 2 ^ 14.
Structure TLSPlainText := {
 type : packet (decodep ContentType) ;
 version : packet (decodep ProtocolVersion) ;
 length : packet (fun x ⇒ decodep uint16 x && length_maxp x) ;
 fragment : packet
  (decodep (opaque \[[ S41.bytes2valueZ length \]])) }.
```

The dependency is between the fields `fragment` and `length`. Each field is expressed as a `packet` of some predicate; checking whether a list of bytes is a `TLSPlainText` packet consists in applying these predicates in sequence. `length_maxp` corresponds to the prose statement that "TLSPlaintext records [carry] data in chunks of 2^14 bytes or less".

Using above ideas, we formalized the packet formats of the Handshake protocol (which is a layer below the Record protocol, to which `TLSPlainText` belongs). The Handshake protocol encloses in particular ClientHello packets whose parsing in C is the topic of Sect. 5.

5 Towards Verification of PolarSSL ClientHello Parsing

5.1 The Parsing Function and its Data Structures

The central data structure in PolarSSL records the characteristics of the TLS connection: the stage of the protocol (field "state"), the version used (fields "*_ver"), the session number (field "session"), the negotiated cipher suite (field "cipher" of `ssl_session`), the session id (field "id" of `ssl_session`), cipher suites of the server (field "ciphers"), and the nonce for this session (field "randbytes"). Other fields ("in_hdr", "in_msg", "in_left") are for navigation into the buffer that stores the incoming bytes:

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

Fig. 1 displays the beginning of the PolarSSL function that parses ClientHello packets of TLS version 1.0. This part deals with the header of the encapsulating Record packet. As often done in other proof assistant-based verification projects of C code (e.g., [12]), we adapt the original code to structured control-flow by replacing the gotos with if-then-else's and by merging returns (so that the `ret` instruction in Fig. 1 is a nop). This is therefore not exactly the original code

```
0 Definition ssl_parse_client_hello : @while.cmd cmd0 bexp := (
1  ssl_fetch_input "ret" "ssl" (cst32 (Z2s _ 5)) ;
2  while.ifte (var_e "ret" ≠ cst32 (Z2s 32 0))
3  ret
4  ("buf" ←* var_e "ssl" ↪ "in_hdr" ;
5   "_buf0_" ←* var_e "buf" ;
6   while.ifte (var_e "_buf0_" && cst8 (Z2s 8 -128) ≠
7               cst8 (Z2s 8 0))
8   ("ret" ← POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ; ret)
9   (while.ifte (var_e "_buf0_" ≠ SSL_MSG_HANDSHAKE)
10   ("ret" ← POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ; ret)
11   ("_buf1_" ←* add_pe (var_e "buf") cst32_1 ;
12    while.ifte (var_e "_buf1_" ≠ SSL_MAJOR_VERSION_3)
13    ("ret" ← POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ;
14     ret)
15    ("_buf3_" ←* add_pe (var_e "buf") (cst32 (Z2s _ 3)) ;
16     "_buf4_" ←* add_pe (var_e "buf") (cst32 (Z2s _ 4)) ;
17     "n" ← ((c2i) (var_e "_buf3_") \<<e cst32 (Z2s _ 8)) \|e
18             (c2i) (var_e "_buf4_") ;
19  ...
```

Fig. 1. `ssl_parse_client_hello` (`ssl_srv.c`, v.0.14.0): formal model

that we verify, but this is the code that we retrofit in the original application; compared to the original function, it is close in structure and has the advan-

tage of being formally verifiable. C expressions are almost ported as they are thanks to our library for finite-size integers [6] to represent bit-wise operations. Yet, since expressions cannot have read side-effects, some C expressions need to be split into several commands using temporary variables (hence, the "_bufi_" variables in Fig. 1). `ssl_parse_client_hello` calls several library functions, such as `ssl_fetch_input`, a function that reads bytes from the input socket and fills a buffer with them. We do not plan to formally verify library functions for the time being and just axiomatize their correctness.

5.2 Verification Goal and Approach

We want to prove that, given an appropriate initial state and input from the network (modeled as the list of bytes `SI`, for "socket input"), `ssl_parse_client_hello` either fails (by returning a non-zero value) or succeeds in checking that the incoming ClientHello packet is valid and updating the state of the server:

```
Lemma POLAR_parse_client_hello_triple : ∀ 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, ...
 {{ (* precondition (see below) *) }}
   ssl_parse_client_hello
 {{ fun s h ⇒ (∃ i, [ var_e "ret" ]_ s = Some (bval32 0_32) ∧
                (* postcondition (see below) *))
  ∨ [ var_e "ret" ]_ s ≠ Some (bval32 0_32) }}.
```

The precondition below specifies the initial state: the type context (from line 1), the local variables (their types from line 3, their values from line 5), and the initial state of the heap. The latter is specified by a Separation logic formula (starting from line 6). It is the formalization of Fig. 2 (left part). Except for pointers and the state of the protocol (`S74.client_hello`), fields are uninitialized. The buffer `BU` is a sensitive storage space: it is for the input bytes and verification must make sure that it is not overrun.

```
0  fun s h ⇒ wf_tstore (_tstore s) ∧
1   s ⊢^g "ssl_context" ⊣ ssl_ctxt ∧ s ⊢^g "ssl_session" ⊣ ssl_sess ∧
2   ...
3   s ⊢^t "ret" ⊣ btyp sint32 ∧ s ⊢^t "ssl" ⊣ ptyp ssl_context ∧
4   s ⊢^t "buf" ⊣ ptyp (btyp uchar) ∧  ...
5   s ⊢^v "ssl" ⊣ pval vssl ∧ ...
6   ((cst_pe _ a ⊢^a uchar → map cst8 BU) ⋆
7    (cst_pe _ rb ⊢^a uchar → map cst8 RB) ⋆
8    (cst_pe _ id ⊢^a uchar → map cst8 ID) ⋆
9    (cst_pe _ ses ⊢ ssl_session →
10    (bval32 cipher0 :: bval32 length0 :: pval id :: nil)
11    CST_SE _) ⋆
12   (cst_pe _ ciphers ⊢^a uint32 → map cst32 CI) ⋆ TT ⋆
13   (var_e "ssl" ⊢ ssl_context →
14    (bval32 (Z2u 32 S74.client_hello) ::
15     bval32 majv0 :: bval32 minv0 :: bval32 mmaj0 ::
```

```
16      bval32 mmin0 :: pval ses ::
17      pval (a + Z2u ptr_len 8) :: pval (a + Z2u ptr_len 13) ::
18      bval32 0_32 :: pval ciphers :: pval rb :: nil) CST_SE _) s h
```

At the time of this writing, our verification effort has not gone further than what is displayed in Fig. 1, i.e., parsing of the Record header. We will therefore just comment about the postcondition (see [16] for a tentative complete formalization). The first part of the postcondition specifies that the incoming bytes form a valid ClientHello. This could be checked by applying the appropriate decoding function from the formal RFC to the slice of the buffer `BU` containing the incoming bytes (to be precise, from the 8th byte—PolarSSL magic number—and of length the value stored in the "in_left" field of the `ssl_context` data structure). This is our ultimate goal but, for the time being, we are in the process of checking correctness conditions one by one, as we advance through the verification. Let us illustrate how we work with the formal RFC. Regarding the Record header, we have to verify for example that it specifies a Handshake packet, i.e., that `u2Z (nth 8 BU 0_8) = S621.handshake`, or that the length that it encodes is bounded according to the RFC, i.e., that `S621.length_maxp (n SI)` holds, where `n` is that slice of `SI` that contains the encoded length. These examples show that the formal RFC is useful to replace magic numbers and ad-hoc interpretation of prose statements from [7]. The second part of the postcondition specifies that

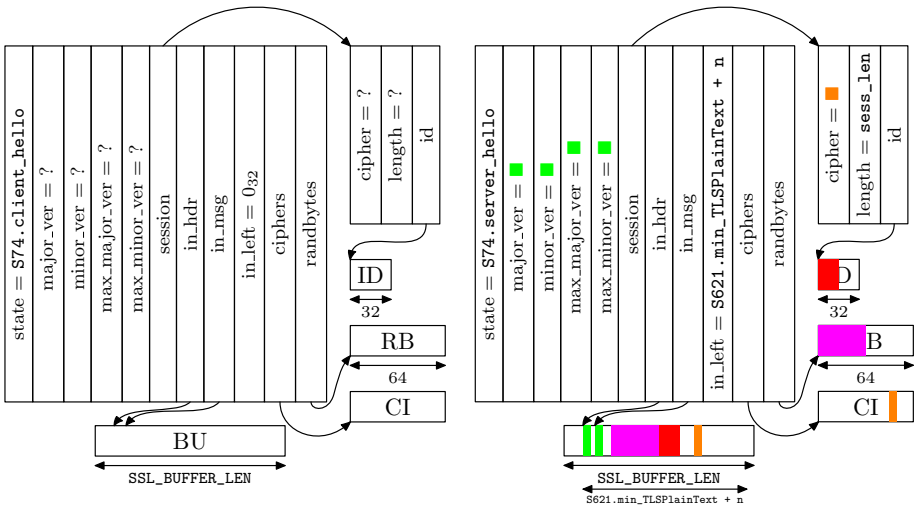

Fig. 2. State of the heap before (on the left) and after parsing (one the right)

the state of the heap after parsing has been updated correctly with the incoming data. As for the precondition, this is captured by a Separation logic formula. Fig. 2 (right part) provides a pictorial representation that can be compared with the initial heap state (on the left). The array `BU` is filled with a ClientHello

packet whose contents are duplicated in PolarSSL data structures. For example, the array for random bytes `RB` has been half-filled with the client nonce. Also, the state of the protocol is updated to `S74.server_hello`.

Note that it will not be possible to guarantee that `ssl_parse_client_hello` succeeds for any correct incoming packet because PolarSSL has several restrictions, that are either common practice (e.g., the restriction that the length of the Handshake must be larger that the length of the ClientHello that is embeds—"Theoretically, a single handshake message might span multiple records, but in practice this does not occur.", [2] p.70) or just application-related limitations (PolarSSL does not handle packets as large as what is allowed by the RFC).

6 Related Work

[10] proposes a formalization of Separation logic for C in the Isabelle proof-assistant, with an application to a memory allocator. Our use-case being different led us to work on different issues, like the formalization of the RFC for TLS. Our technical development also differs: at the level of the definition of C types ([10] only distinguishes between scalar and aggregate types, so that case-by-case lemmas are required to ensure that C types are correctly modeled—Sect. 5.3 in [10]); at the level of pointer arithmetic ([10] favors a variant of the Burstall-Bornat model for heap access whereas we stick to direct, byte-level accesses). Yet, it is interesting to compare lemmas in both formalizations (e.g., the corollary of Theorem 7.5 in [10] with the lemma `mapsto_styp_inv` in Sect. 3.1).

[11] provides a complete model of C in Coq, but without Separation logic. Again, technical developments differ in several ways. For example, to avoid dealing with a type context, [11] chooses a "structural" encoding for structs: an enclosing struct can always be referred to by using "indices" so as to enable the definition of recursive types. Originally, we did not choose this direction because it requires to rework the types from the original program.

[12] proposes a different approach to the problem of interactive verification of C programs. There is no Separation logic per se, but Hoare logic is used to establish simulations (see Sect. 5.2 in [12]). Application of this approach to PolarSSL would require the construction of a reference implementation, what would be another way to formalize the RFC for TLS.

7 Conclusion

In this paper, we introduced a framework for formal verification of TLS packet processing in C. It consists of a formal model of a subset of C in Coq. This model features a simple encoding of C types that allows for easy and faithful modeling. We equipped this model with a Separation logic, that differs from textbook Separation logic because of C types. This logic has been tested by deriving standard Separation logic-lemmas and by verifying the standard in-place list reversal. We then investigated application to a parsing function of an existing implementation of TLS. Specification required formalization of the

corresponding RFC. We did so by providing encodings of packet formats that led us to improve the original document, in particular by spotting errors.

We are also 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 [9]). At this stage, there are still many ways to improve our model of C and its Separation logic. We already worked out a model for dynamic allocation that extends [3] (Sect. 7) with C types. We plan to work on compliance with the C standard (in particular, portability issues) and on the interface with assembly (so as to verify those parts of the implementation of TLS implementations that use assembly for cryptography).

References

1. Harbison, S.P., Steele, G.L: C: A Reference Manual. 5th edition. Prentice Hall, 2002
2. Rescorla, E.: SSL and TLS: Designing and Building Secure Systems. 11th Printing. Addison Wesley, 2000.
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. Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., Wansbrough, K.: Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. Proc. of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006), pp.55–66. ACM Press, 2006.
5. Marti, N., Affeldt, R., Yonezawa, A.: Formal Verification of the Heap Manager of an Operating System using Separation Logic. Proc. of the 8th Intl. Conf. on Formal Engineering Methods (ICFEM 2006), vol. 4260 of LNCS, pp.400–419. Springer, Oct. 2006.
6. Affeldt, R., Marti, N.: An Approach to Formal Verification of Arithmetic Functions in Assembly. Proc. of the 11th Annual Asian Computing Science Conf. (ASIAN 2006), vol. 4435 of LNCS, pp.346–360. Springer, Jan. 2008.
7. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246. IETF, 2008.
8. PolarSSL. Open Source embedded SSL/TLS cryptographic library. `http://polarssl.org`.
9. OpenSSL. Open Source toolkit for SSL/TLS. `http://www.openssl.org`.
10. Tuch, H.: Formal Verification of C Systems Code. *J. Autom. Reasoning*, vol. 42(2–4), pp.125–187. Springer, 2009.
11. 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.
12. Winwood, S., Klein, G., Sewell, T., Andronick, J., Cock, D., Norrish M.: Mind the Gap. Proc. of the 22nd Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs 2009), vol. 5674 of LNCS, pp.500–515. Springer, 2009.
13. *The Coq Proof Assistant: Reference Manual.* Ver. 8.3. Available at `http://coq.inria.fr`. INRIA, 2010.
14. Brady, E.: Idris—Systems Programming meets Full Dependent Types. Proc. of the 5th ACM Wksp. Programming Languages meets Program Verification (PLPV 2011), pp.43–54. ACM Press, 2011.
15. 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`.

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