Towards Formal Verification of TLS Network Packet Processing Written in C [*] [†]

Reynald Affeldt Nicolas Marti

Research Institute for Secure Systems
National Institute of Advanced Industrial Science and Technology
{reynald.affeldt,nicolas.marti}@aist.go.jp

Abstract

TLS is such a widespread security protocol that errors in its implementation can have disastrous consequences. This responsibility is mostly borne by programmers, caught between specifications with the ambiguities of natural language and error-prone low-level parsing of network packets. We provide new Coq libraries for the formal verification of TLS packet processing written in C. The originality of our encoding of the core subset of C is its use of dependent types to guarantee statically well-formedness of datatypes and correct typing. We further equip this encoding with a Separation logic that enables byte-level reasoning and also provide a logical view of data structures. We also formalize a significant part of the RFC for TLS, again using dependent types to capture succinctly constraints that are left implicit in the prose document. Finally, we apply the above framework to an existing implementation of TLS of which we specify and verify a parsing function for network packets. Though not yet completed, this experiment already led us to spot correctness issues with the RFC and the C source code.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Correctness proofs; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical Verification

Keywords Dependent types, Coq, C, Separation logic, TLS

1. Introduction

TLS (Transport Layer Security) [14] is such a widespread security protocol that errors in its implementation can have disastrous consequences. This responsibility is mostly borne by programmers, caught between error-prone low-level programming with C and specifications with the ambiguities of natural language.

We want to use verification with a proof-assistant to improve the implementations of TLS. One can think of several ways to use proof-assistants to improve the implementations of communication protocols in general. For example, Sewell et al. developed an HOL specification of TCP to test implementations of the Socket API [8]; this proves effective but lets open the question of the source code adequacy to the programmer's intent. Brady proposed to use a dependently-typed programming language to specify and verify network packet processing [11]; yet, such implementations continue to be developed in C for performance reasons.

Our purpose is to provide a framework in the Coq proof-assistant [12] for the interactive verification of C programs that process TLS packets. Our viewpoint is the following. Programmers "almost" always get it right when they write a program. The problem is that when it comes to security, "almost" is everything. We believe that it should be possible to use proof-assistants to develop programs that are correct-by-construction by adding just a little overhead at programming-time. This is a long-term goal but this is our motivation to work on interactive theorem proving rather than aiming at full automation.

The main element of our framework is a new library for the verification of the core subset of C. It is based on Separation logic [25], a variant of Hoare logic that deals with pointers, the latter being pervasively used in network packet processing. The originality of our encoding is the use of dependent types. Technically, we parametrize the encoding of C types with a type context and provide functions to check well-formedness (Sect. 2 and 3); we then use this encoding of C types in an "intrinsic" encoding [7] of C expressions and C commands that enforces correct typing "by construction" (Sect. 4). We have encoded standard Separation logic, equipped with the expected lemmas such as the frame rule. In addition to direct manipulation of memory in terms of bytes, we provide reasoning rules that treat C data structures in a "logical" way, abstracting low-level details such as padding. This proves useful even for a simple example such as the mandatory in-place list reversal (Sect. 5).

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 [14]. This not only gives us formal grounds to lay down specifications of the C source code, but also has the side-effect of improving the original RFC by making precise prose-only statements (Sect. 6).

Finally, we investigate (Sect. 7) application to an existing implementation of TLS, namely PolarSSL [23]. Concretely, we formalize the function that parses initialization packets, specify it w.r.t. the formal RFC, and verify it. 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 [22]). At the time of this writing, the experiment is not yet completed but progress is significant, and we already found implementation errors.

[*] This work is supported by the National Institute of Information and Communications Technology of Japan.

PLPV'13 January 22, 2013, Rome, Italy.
Copyright © 2013 ACM 978-1-4503-1860-1/13/01. . . $15.00

2. A Standard-Compliant Encoding of C Types

In C, recursive references in types can only appear as pointers, so as to ensure that all types have a finite size. This can be modeled with a structurally recursive definition of types [18], but at the price of a tedious encoding of mutually recursive types. We choose to refer to C structures by names, using a *type context*. The result is a more natural type representation, but also a more involved mechanization because of non-trivial termination issues.

2.1 Encoding of C Types and of Type Contexts

We define our subset of the C types as an inductive type[1]:

```
Inductive typ : Set :=
  | btyp of ityp | ptyp of typ | styp of tag
Inductive ityp : Set :=
  | uint | sint | uchar | ulong.
Inductive tag := mkTag : string → tag.
```

The type `typ` models: basic arithmetic types (defined in `ityp`: unsigned and signed integers, unsigned characters, and unsigned long integers), pointers types, and structure types (identified by a `tag`).

To each structure `tag`, we want to associate a list of pairs of a string and a `typ` that model the fields of C structures:

```
Module Fields <: finmap.EQTYPE.
Definition A := [eqType of seq (string * typ)].
End Fields.
```

Type contexts are finally obtained by instantiating a module for finite maps: `Module Γ := compmap TagOrder Fields` (`TagOrder` is a module that equips `tag` with the lexicographical order).

We say that a type is *covered* when all the tags it contains are in the domain of the type context (otherwise it is “incomplete” in C parlance). Put formally:

```
Definition cover g t := inc (tags t) (Γ.dom g).
```

(The function `tags` collects the `tag`s in a `typ`.)

2.2 Well-formedness of Type Contexts

In C, a type context is well-formed when (1) it is *complete*, (2) it has no empty structure, and (3) recursion only goes through pointers.

(1) Completeness A context is said complete when all the types in its codomain are covered:

```
Definition complete g :=
  ∀ tg flds, Γ.get tg g = ⌊ flds ⌋ →
    ∀ t, t ∈ unzip2 flds →
      ∀ tg', tg' ∈ tags t →
        ∃ flds', Γ.get tg' g = ⌊ flds' ⌋.
```

Completeness can be decided by checking whether the set of tags in the codomain of the context is included in the domain.

(2) Non-emptiness Contrary to C++, C forbids empty structure:

```
Definition no_empty g := ∀ flds,
  flds ∈ Γ.cdom g → size flds ≠ 0.
```

(3) No cycle No structure can be defined in terms of itself, even indirectly, unless recursion goes through a pointer. To define this property, we introduce the notion of nesting of tags:

```
Definition nested g tg₁ tg₂ :=
  if Γ.get tg₁ g is ⌊ l ⌋ then
    has (fun x ⇒ match x.2 with
                  | styp tg ⇒ tg₂ = tg | _ ⇒ false
                  end) l
  else false.
```

[1] We have also extended `typ` to deal with arrays of structures but since we do not rely on this extension in our use-case we skip this explanation.

This computable relation states that the tag tg_1 refers to a structure with at least one field whose type contains a structure with tag tg_2. Using this relation, we build the set of all paths of nested tags (`{:n.+1.-tuple tag}` is the type of lists of `tag`s of size `n.+1`):

```
Definition path_nested g n : Set :=
{ p : {:n.+1.-tuple tag} | (thead p ∈ Γ.dom g)
  && path (nested g) (thead p) (behead p) }.
```

Finally, there is no cycle in a type context when all possible paths (of any size) do not contain twice the same tag (this is the meaning of the predicate `uniq`, `sval` is the dependent pair projection):

```
Definition no_cycle g :=
  ∀ n (p : path_nested g n), uniq (sval p).
```

So, formally, a well-formed context is defined as follows:

```
Definition wf_ctxt g :=
  no_cycle g ∧ complete g ∧ no_empty g.
```

We reflect the well-formedness property to make Coq automatically enforce it. Reflection is a bit involved for (3), essentially because of the universal quantification over the paths' size in `no_cycle`. We observe that if a path has no cycle then its size is bounded by the size of the type context. Therefore, to decide the absence of cycles, one only needs to check a finite number of paths, as provided by the following function, which provably enumerates all the paths of a given size:

```
Fixpoint compute_paths g n :
  seq {:n.+1.-tuple tag} := ...
Lemma compute_paths_spec g n (p:path_nested g n):
  sval p ∈ (compute_paths g n).
```

We can now build a function that checks the `uniq`ness of all the paths and prove that it implies the absence of cycles (`iota a b` is the list of the `b` integers following `a`; `tval` is the dependent pair projection):

```
Definition no_cycleb g := all
  (fun n ⇒ all (fun x ⇒ uniq (tval x))
   (compute_paths g n))
  (iota 0 (size (Γ.dom g))) &&
 (compute_paths g (size (Γ.dom g)) = nil).
Lemma no_cycleb_sound g:no_cycleb g → no_cycle g.
```

Similarly, we reflect (1) and (2) as the functions `completeb` and `no_emptyb`, and arrive at a provably sound boolean definition:

```
Definition wf_ctxtb g :=
  no_cycleb g && completeb g && no_emptyb g.
Lemma wf_ctxtb_sound g : wf_ctxtb g → wf_ctxt g.
```

We finally encode well-formed contexts as dependent pairs; by combining Coq lazy-parsing rules and reflection-proofs, we even provide a notation that automatically enforces well-formedness:

```
Definition wfctxt := {g | wf_ctxt g}.
Notation "\wfctxt{ g }" :=
  (exist _ g (wf_ctxtb_sound g (eq_refl _))).
```

Similarly, covered types (or “complete” types in C parlance) are built as dependent pairs of a type and a proof that the type is covered by a well-formed type context:

```
Definition covered (g : wfctxt) : Type :=
  {t : typ | cover (sval g) t }.
Definition mkCovered g t H : covered g :=
  exist (cover (sval g)) t H.
Notation "g '.-typ:' t " :=
  (mkCovered g t (eq_refl _)).
```

We also note “`btyp: t`” the construction of a covered basic arithmetic type and “`:*t`” the construction of a covered pointer type.

2.3 Example of Type Declaration

Let us consider two self-referential C structures:

```
{struct cell ;
 struct header {struct cell *first;};
 struct cell   {char data; struct header *head;};}
```

The Coq script below defines the context g that consists of the cell and header structures, automatically checking that it is well-formed, and eventually defines the cell and header types, automatically checking that they are covered by g. We will pursue this example in Sect. 3.2 by providing sizeof calculations.

```
Definition cell_tg := mkTag "cell".
Definition header_tg := mkTag "header".
Definition cell_flds := ("data", btyp uchar) ::
  ("head", ptyp (styp header_tg)) :: nil.
Definition header_flds :=
  ("first", ptyp (styp cell_tg)) :: nil.
Definition g := \wfctxt{ "cell" ▹ cell_flds,
  "header" ▹ header_flds, ∅ }.
Definition cell := g.-typ: styp cell_tg.
Definition header := g.-typ: styp header_tg.
```

3. Alignment and Size of C Types

In hardware, a memory access is faster when the address is a multiple of the alignment of the data. This fact has triggered particular attention for aligned memory footprints for C types in the C standard. As a consequence, in the case of structures, preserving alignment of the data often requires to add padding bytes between fields. For our encoding of C to be realistic, we need to compute alignment and correct size information for all types.

3.1 A Generic Traversal Function for C Types

Our goal is to produce a function fp_typ that traverses objects of type g.-typ to compute some result, such as alignment or sizeof. For basic arithmetic types or pointers, this is simple: alignments and sizes are given by definition. But for structures, one needs to recursively go through all the fields, and termination is non-trivial.

Before all, let us parametrize fp_typ with a record so that it can be later instantiated to perform different computations:

```
Variable g : wfctxt.
Record config {Res Tmp : Type} := mkConfig {
  f_ityp : ityp → Res ;
  f_ptyp : typ → Res ;
  f_styp_iter : Tmp → g.-typ * Res → Tmp ;
  f_styp_acc : Tmp ;
  f_styp_fin : g.-typ → Tmp → Res}.
```

The functions f_ityp and f_ptyp treat the cases of the basic arithmetic types and of the pointer types. For structures, we will perform a (left-)fold over the fields with f_styp_acc as the initial accumulator, f_styp_iter as the iterator function, and f_styp_fin as a "finalizer function" whose first argument is the styp currently treated.

Although Coq accepts natively only structurally-recursive functions, its standard library allows functions to be built with termination based on a measure of their arguments. This is enough to ensure the termination of the following fix-point combinator:

```
frec : ∀ (Arg Res : Type) (metric : Arg → nat),
 (∀ a : Arg,
  (∀ a':Arg, metric a' < metric a → Res) → Res) →
    Arg → Res
```

Hereafter, let us assume that Res is the return type of the traversal function fp_typ, Tmp is the auxiliary type used by the fold on structures, and c is an arbitrary computation configuration:

```
Variables Res Tmp : Type.
Variable c : @config Res Tmp.
```

We now produce a function fp_styp_body such that recursive traversal for structures will be obtained by using the fix-point combinator.

```
Record Trace : Type := mkTrace {
  next : {tg : tag | cover (sval g) (styp tg)} ;
  prev_sz : nat ;
  prev : {p : path_nested (sval g) prev_sz |
          tlast (sval p) = sval next} }.

Definition remains tr :=
  size (Γ.dom (sval g)) - prev_sz tr.

Program Definition fp_styp_body (tr : Trace)
 (f : ∀ tr', remains tr' < remains tr → Res) : Res
 := ...
```

The function fp_styp_body has its arguments packed in a dependent record of type Trace: next is the next structure tag to proceed, and prev is the path (of size prev_sz) representing the previous nested calls of the function (with next as its last argument). As observed in Sect. 2.2, such a path is bounded by the size of the type context. By defining fp_styp_body's argument measure as the difference between the current path and its bound, we can prove that any recursive call will terminate. This gives us the recursive function fp_styp for structure traversal, that we use to define the traversal function fp_typ for any typ traversal:

```
Definition fp_styp :=
  frec Trace Res remains fp_styp_body.

Program Definition fp_typ (t : g.-typ) : Res :=
  match sval t with
    | btyp i ⇒ c.(f_ityp) i
    | ptyp p ⇒ c.(f_ptyp) p
    | styp tg ⇒ fp_styp (mkTrace tg 0 _)
  end.
```

Alignment and Sizeof Here follows the instantiation of fp_typ to compute alignment:

```
0 Definition align_ityp it :=
1   match it with uint ⇒ 4 | sint ⇒ 4 |
2   uchar ⇒ 1 | ulong ⇒ 8 end.
3 Definition align_config g := mkConfig g
4   align_ityp (fun _ ⇒ 4)
5   (fun a x ⇒ maxn x.2 a)
6   1 (fun _ x ⇒ x).
7 Definition align {g} := fp_typ (align_config g).
```

This construction respects the C standard. For example, the fact that the "alignment requirement for a structure type will be at least as stringent as for the component having the most stringent requirements" [17, p. 158] is encoded by taking the maximal alignment of the structure's fields types (line 5 above).

To compute the size of a structure, one needs to compute the padding, as the C standard requires to have all the fields aligned. We therefore need to use the previous align function together with a padding function:

```
Definition padd addr ali :=
  let r := addr % ali in
    if r = 0 then 0 else ali - r.
```

Here follows the instantiation of fp_typ to compute the size of datatypes:

```
0 Definition sizeof_ptr : nat := 4.
1 Definition ptr_len := sizeof_ptr * 8.
2 Definition sizeof_ityp it :=
3   match it with uint ⇒ 4 | sint ⇒ 4
4   | uchar ⇒ 1 | ulong ⇒ 8 end.
5 Definition sizeof_config g := mkConfig g
6   sizeof_ityp (fun _ ⇒ sizeof_ptr)
7   (fun a x ⇒ a + padd a (align x.1) + x.2) 0
8   (fun t a ⇒ a + padd a (align t)).
9 Definition sizeof{g}:= fp_typ g (sizeof_config g).
```

This rigorously models the C standard. For example, for structures, the "rule is that the structure will be padded out to the size the type would occupy as an element of an array of such types" [17, p.158]. This is achieved by the finalizer function at line 8.

The above definitions let us prove in Coq the expected properties of alignment and sizeof, e.g.:

```
Lemma align_sizeof (t:g.-typ) : align t | sizeof t.
```

3.2 Examples of Alignment and Sizeof

Let us illustrate with the example of Sect. 2.3 the results of sizeof. For example, it will correctly compute the 3 bytes of padding between the "data" and the "head" fields of cell structures, whereas header structures have the same size as the pointer they carry:

```
Goal (sizeof cell = 1 + 3 + 4). by []. Qed.
Goal (sizeof header = 4). by []. Qed.
```

4. A Dependently-typed Encoding of Core C

4.1 The Physical View of C Values

The semantics of C exposes details at the byte-level. A value for C is therefore essentially a list of bytes, whose size ought to correspond to some type, as defined by the following dependent record:

```
Record phy {g} (t : g.-typ): Type := mkPhy {
  phy2seq :> seq (int 8) ;
  Hphy : size phy2seq = sizeof t }.
```

We note ty.-phy the type of physical values of type ty. We will use physical values in the next sections (Sect. 4.2 and 4.3) to define the evaluation of C expressions and the execution of C commands.

4.2 Dependently-typed C Expressions

Our encoding of C expressions is an inductive type exp *indexed* by a C type that varies in the result types of the different constructors. This makes it possible to build the type rules into the definition of the syntax, so that terms are well-typed by construction. For example, the constant 1 seen as a signed integer (the default in C) is an expression of type exp (btyp: sint). Assume now that we are given a variable buf (noted %"buf" when considered as an expression) of type exp (:*(btyp: uchar)). If we define addition, multiplication, etc. only for basic arithmetic types:

```
| bop_n : ∀t, binop_ne (*numerical operators*) →
  exp (btyp: t) → exp (btyp: t) → exp (btyp: t)
```

and pointer arithmetic (noted $+_p$) as follows:

```
| add_p : ∀ t,
  exp (:* t) → exp (btyp: sint) → exp (:* t)
```

then %"buf" ∗ [1]$_{sc}$ is forbidden but %"buf" $+_p$ [1]$_{sc}$ is allowed, and moreover deemed to have type exp (:∗(btyp: uchar)), as desired. Here follows a more exhaustive definition of C expressions (g is a type context and σ is an environment that associates free variables to types):

```
Inductive exp {g σ} : g.-typ → Type :=
| var_e : ∀ str t, env_get str σ = ⌊ t ⌋ → exp t
| cst_e : ∀ t (v : t.-phy), exp t
| bop_n : ∀t, binop_ne (*numerical operators*) →
  exp (btyp: t) → exp (btyp: t) → exp (btyp: t)
| bop_r : ∀t, binop_re (*relational operators*) →
  exp(btyp:t) → exp(btyp:t) → exp(g.-btyp: uint)
| add_p : ∀ t,
  exp (:* t) → exp (btyp: sint) → exp (:* t)
| bop_p : ∀ t, exp (:* t) → exp (:* t) →
  exp (g.-btyp: uint)
| ifte_e : ∀ t, exp (btyp: uint) →
  exp t → exp t → exp t.
(* to continue in Sect. 4.2.2 and 4.2.1 *)
```

var_e is for variables (noted %"x", etc.); the type of variables is fixed by the environment σ. cst_e is for constants (noted [x]$_c$, etc.; [x]$_{sc}$ (resp. [x]$_{uc}$) is for signed (resp. unsigned) integers); any list of bytes whose size corresponds to a valid C type can be a constant. bop_n is for numerical operations over integral types (such as addition +, left shift ≪, bitwise or |, etc.). bop_r is for relational operators for arithmetic types (comparisons, etc.); it returns an integer (0 or 1). add_p is for pointer arithmetic (noted $+_p$). bop_p is for pointer comparisons. ifte_e is for conditional expressions.

Boolean expressions are implemented similarly (see [2]).

4.2.1 Field Accesses

The constructor fldp of exp is for field access (notation ⇒):

```
0 (* continued from Sect. 4.2 *)
1 | fldp : ∀ f tg t (e : exp (:* t)) H t',
2   assoc_get f (get_fields t tg H) = ⌊ t' ⌋ →
3   exp (:* t')
```

We use the type of the dereferenced pointer to check whether the field is indeed valid (line 2; get_fields t tg H returns the list of fields of the C structure tagged tg). Note that a field access returns a pointer to the field instead of the field itself; this is because we do not support read side-effects from the heap in expressions.

4.2.2 Type Conversions

In C, type conversions between basic integral types may occur implicitly when necessary for execution. For example, when adding a character to an integer, the character is promoted to an integer beforehand. These conversions can lead to data loss and misinterpretations. For example, when signed integers are used in place of unsigned integers, a type conversion silently occurs that is in general unsafe when the signed integer is strictly negative.

Our encoding of C expressions supports (safe and potentially unsafe) type conversions. Safety can be decided based on the types of the conversion (boolean conditions safe, data_loss, and misinterpret below; see the implementation [2] or standard literature, e.g. [27, p. 162–163], for the precise definitions):

```
(* continued from Sect. 4.2 *)
| safe_cast : ∀ t t', exp (btyp: t) →
  UnConv.safe t t' → exp (btyp: t')
| unsafe_cast : ∀ t t', exp (btyp: t) →
  UnConv.data_loss t t' ||
  UnConv.misinterpret  t t' → exp (btyp: t')
```

Hereafter, "(int) e" is syntax for safe casts to (signed) integers (automatic checking of boolean conditions can be hidden in notations). We have chosen to make implicit type conversions visible (and we write the unsafe casts with uppercases, e.g., "(UINT) e"). To illustrate, the addition of a character and an integer is written (int) ([5]$_{8sc}$) +[5]$_{sc}$.

4.2.3 Evaluation of Expressions

Since expressions are well-typed by construction, the evaluation of an expression of type t always succeeds and returns a physical value of type t.-phy. Evaluation is performed w.r.t. a store, which is essentially a list of identifiers and physical values. We note

`[ e ]_s` the evaluation of `e` w.r.t. store `s`. The complete definition is a bit long (see [2]) so we just content ourselves with the example of pointer arithmetic. $e_1 +_p e_2$ is of type pointer to `t`. Dependent types impose that e_1 is also a pointer to `t` and that e_2 is an integer. First, we evaluate e_1 and convert the result to the value of a pointer (conversion ptr◃i8 below[2]). Second, we evaluate e_2 and convert the result to a finite-size integer (conversion i32◃i8); the latter is further interpreted as signed (conversion Z◃s). Finally, the pointer is scaled by the size of the type `t`:

```
Fixpoint eval {g σ t} (s : store σ) (e : exp t) :
  t.-phy := match e with (* ... *)
| add_p t' e₁ e₂ ⇒
  match [ e₁ ]_s, [ e₂ ]_s with
    | mkPhy l₁ H₁, mkPhy l₂ H₂ ⇒
      let p := ptr◃i8 l₁ H₁ in
      let k := i32◃i8 l₂ H₂ in
      ptyp◃ptr t' (scalez p (sizeof t') (Z◃s k))
  end (* ... *)
```

4.3 Syntax and Semantics of Core Subset of C

The core subset of C that we formalize is a while-language (the control-flow is expressed using sequences, while-loops, and structured branching) with assignment, lookup (memory dereference), and mutation (destructive update). We derive the syntax and semantics (as well as various lemmas) by instantiating a parametrized module adapted from previous work [3]. We only need to provide the syntax and semantics for the basic commands: assignment, lookup, and mutation.

Like for expressions, the encoding of commands exploits dependent types to enforce well-typed programs by construction. For example, the `lookup` constructor defined below requires that the expression to be dereferenced is of pointer type `:*t` and that the destination variable is of the type `t` (constraint line 5):

```
0 Inductive cmd₀ : Type :=
1 | skip
2 | assign : ∀ {t : g.-typ} x, @exp _ σ t →
3   env_get x σ = ⌊ t ⌋ → cmd₀
4 | lookup : ∀ {t : g.-typ} x, @exp _ σ (:* t) →
5   env_get x σ = ⌊ t ⌋ → cmd₀
6 | mutation : ∀ {t : g.-typ}, @exp _ σ (:* t) →
7   @exp _ σ t → cmd₀.
```

We refer to the complete language (with control-flow) as the type `cmd`. The constraints can be checked automatically when we write down concrete programs, so that we can hide them in user-friendly notations:

```
Notation "x '←*' e" := (@lookup _ x e (eq_refl _)).
Notation "x '←' e" := (@assign _ x e (eq_refl _)).
Notation "e₁ '*←' e₂" := (@mutation _ e₁ e₂).
```

The operational semantics for the basic commands is given in Fig. 1. It is a relation between an optional *state* (a pair of a store `s` and a heap `h`) (⊥ represents an execution error). The effect of an assignment (constructor `exec_assign`) is to update the store. A lookup (`exec_lookup`) evaluates the expression to be dereferenced, turns the resulting physical value into an address, then uses this address to get a chunk of memory of the appropriate size from the heap; finally the value obtained is stored back in the store (function `store_upd`). Lookup fails when the addressed memory is not initialized (constructor `exec_lookup_err`). The constructors for mutation should now be self-explanatory.

[2] We use many such conversions in this paper. They are displayed for the sake of completeness but can be ignored on a first reading.

5. Separation Logic for the Core Subset of C

5.1 Hoare Logic for the Core Subset of C

Like for the syntax and the semantics of the core subset of C, the Hoare logic and its properties are obtained using a parametrized module from previous work [3]. The type of assertions is shallow-encoded: `Definition assert :=store →heap →Prop`. Fig. 2 displays the inductive relation `hoare0` of the Hoare triples for the basic commands. For example, the assignment rule

$$\overline{\{P\{e/x\}\}x \leftarrow e\{P\}}$$

is encoded by the constructor `hoare_assign`. Its precondition is expressed using the predicate transformer `wp_assign`; since assertions are shallow-encoded, substitution is encoded by updating the store. Other Hoare rules should be self-explanatory since they follow the operational semantics explained in Sect. 4.3.

5.2 The Mapsto Connective of Separation Logic for C

Regarding the encoding of Separation logic, what is new is not the separating conjunction $\star$ or implication $-\!\!\star$ (their encoding is as usual, see for example [3]), but rather the primitive mapsto connective. The mapsto connective is noted $e \mapsto e'$ and holds for a heap containing one cell with contents e' at address e. For the archetypal language of textbook Separation logic, a cell consists of an (arbitrary-precision) integer. It would not be practical for C to make every memory byte a cell. A cell ought rather be a C (physical) value. This amounts to provide a version of mapsto parametrized with the accessed type (like the "chunks" in Appel and Blazy's Separation logic [5]).

The physical values of Sect. 4.1 bear no relation with the memory of the computer, but C has strict requirements regarding storage. In order to define a meaningful mapsto connective for C, one needs to abide by alignment rules and the absence of the null pointer in allocated areas. To define such a relation between a physical value, an address, and a heap, it suffices, as encoded below, to prevent the area to overrun the memory (line 4) and to impose alignment of the physical value (line 5):

```
0 Inductive phy_mapsto {g} {t : g.-typ} :
1   t.-phy → nat → hp.t → Prop :=
2 | mkPhy_mapsto : ∀ a (v : t.-phy) h,
3   hp.cdom h = v → hp.dom h = iota a (sizeof t) →
4   Z◃nat a + Z◃nat (sizeof t) < 2^ptr_len →
5   align t | a →
6   phy_mapsto v a h.
```

`phy_mapsto` gives us the basis to define (a raw form of) the mapsto connective of Separation logic, $a \mapsto_p v$ (subscript "p" for "physical") that associates an address `a` to a (typed) physical value `v`:

```
Notation "a '↦ₚ' v" := (fun _ ⇒
  phy_mapsto v (nat◃Z (Z◃u (ptr◃ptyp a)))).
```

5.3 Example of Derived Separation-logic Triple

As an example of reasoning involving the physical mapsto connective, let us consider the first backward-reasoning form for lookup [26, p.88]:

$$\overline{\{\exists v.(e \mapsto v) \star (e \mapsto v -\!\!\star P\{v/x\})\}x \leftarrow\!\!\star e\{P\}}$$

First, we provide a definition for the precondition (`wp_assign` has been explained in Sect. 5.1):

```
Inductive wp_lookup_back {t} x H
  (e : @exp g σ (:* t)) P : assert :=
| wp_lkbr1 : ∀ s h (v : t.-phy),
  ([ e ]_s ↦ₚ v ✶ ([ e ]_ s ↦ₚ v -✶
    wp_assign x H [ v ]꜀ P)) s h →
  wp_lookup_back x H e P s h.
```

```
Inductive exec0 : option state → cmd₀ → option state → Prop :=
| exec_skip : ∀ s, ⌊ s ⌋ ≻ skip ⇝ ⌊ s ⌋
| exec_assign : ∀ s h t x H (e : exp t), ⌊ s, h ⌋ ≻ assign x e H ⇝ ⌊ store_upd s x H [ e ]_s, h ⌋
| exec_lookup : ∀ s h t x H e v, let a := nat◃u (ptr◃ptyp [ e ]_s) in
  heap_get t a h = ⌊ v ⌋ → ⌊ s, h ⌋ ≻ lookup x e H ⇝ ⌊ store_upd s x H v, h ⌋
| exec_lookup_err : ∀ s h t x H (e : exp (:* t)), let a := nat◃u (ptr◃ptyp [ e ]_s) in
  heap_get t a h = ⊥ → ⌊ s, h ⌋ ≻ lookup x e H ⇝ ⊥
| exec_mutation : ∀ s h t e₁ e₂ v, let a := nat◃u (ptr◃ptyp [ e₁ ]_s) in
  heap_get t a h = ⌊ v ⌋ → ⌊ s, h ⌋ ≻ e₁ *← e₂ ⇝ ⌊ s, heap_upd t a [ e₂ ]_s h ⌋
| exec_mutation_err : ∀ s h t e₁ (e₂ : exp t), let a := nat◃u (ptr◃ptyp [ e₁ ]_s) in
  heap_get t a h = ⊥ → ⌊ s, h ⌋ ≻ e₁ *← e₂ ⇝ ⊥.
```

Figure 1. Big-step operational semantics of basic commands of the core subset of C

```
Inductive wp_assign {t : g.-typ} x H (e : @exp g σ t) (P : assert) : assert :=
| wp_assign_c : ∀ s h, P (store_upd s x H [ e ]_s) h → wp_assign x H e P s h.

Inductive wp_lookup {t : g.-typ} x H (e : @exp g σ (:* t)) (P : assert) : assert :=
| wp_lookup_c : ∀ s h v, heap_get t (nat◃Z (Z◃u (ptr◃ptyp [ e ]_ s))) h = ⌊ v ⌋ →
  P (store_upd s x H v) h → wp_lookup x H e P s h.

Inductive wp_mutation {t : g.-typ} (e₁ : @exp g σ (:* t)) e₂ (P : assert) : assert :=
| wp_mutation_c : ∀ s h v, let a := ptr◃ptyp [ e₁ ]_s in
  heap_get t (nat◃Z (Z◃u a)) h = ⌊ v ⌋ →
  P s (heap_upd t (nat◃Z (Z◃u a)) [ e₂ ]_s h) → wp_mutation e₁ e₂ P s h.

Inductive hoare0 : assert → cmd₀ → assert → Prop :=
| hoare_skip : ∀ P, hoare0 P skip P
| hoare_assign : ∀ P t x H (e : exp t), hoare0 (wp_assign x H e P) (assign x e H) P
| hoare_lookup : ∀ P t x H (e : exp (:* t)), hoare0 (wp_lookup x H e P) (lookup x e H) P
| hoare_mutation : ∀ P t e₁ (e₂ : exp t), hoare0 (wp_mutation e₁ e₂ P) (mutation e₁ e₂) P.
```

Figure 2. Hoare logic for basic commands of the core subset of C

Then, the above rule becomes provable using the Hoare triples seen in Sect. 5.1:

```
Lemma hoare_lookup_back {t} x H (e : exp (:* t)) P :
  { wp_lookup_back x H e P } lookup x e H { P }.
```

5.4 The Logical View of C Values

It was simple to define the physical view of C values. Yet, it is not practical to deal with it when it comes to formal verification of programs. For example, when accessing a structure field, one does not want to deal with the interleaving padding. To overcome this issue, we provide a "logical view" of C values. In this view, C structures are decomposed into the list of their fields, all the way down to basic datatypes.

Logical values are defined by the inductive type `log` (Fig. 3, on the left). We note `t.-log` for the type of logical values of type `t`.

Like for physical values, we define a relation between a logical value, an address, and a heap, stating that the heap contains an encoding of the logical value and that this encoding is correctly stored. The difference with the physical mapsto is that it leaves undefined the contents of the interleaving padding, if any. This relation is defined by the predicate `log_mapsto` (Fig. 3, on the right), and like its physical counterpart, this predicate gives rise to a "logical mapsto" connective for Separation logic:

```
Notation "a '↦ₗ' v" := (fun _ ⇒
  log_mapsto v (nat◃Z (Z◃u (ptr◃ptyp a)))).
```

Example: Singly-linked Lists We first define the C type `lst` of structures containing two fields: `"data"` of type `uint` and `"next"` of type pointer to a `lst` structure:

```
Definition lst_tg := mkTag "lst".
```

```
Definition flds := ("data", btyp uint) ::
  ("next", ptyp (styp lst_tg)) :: nil.
Definition g := \wfctxt{ "lst" ▹ flds, ∅ }.
Definition lst := g.-typ: (styp lst_tg).
```

Using this type, one can define a Separation-logic assertion that relates a Coq list of `int 32` (the contents of the `"data"` fields) with a pointer of type `:*lst` (that points to the singly-linked list):

```
Inductive list_seplog :
  seq (int 32) → (:* lst).-phy → assert :=
| list_nil : ∀ s, list_seplog nil phy₀ s hp.emp
| list_cons : ∀ s hd tl h₁ h₂, h₁ ⊥ h₂ →
  ∀ a p, a ≠ phy₀ →
    (a ↦ₗ mk_cell hd (ptr◃ptyp p)) s h₁ →
    list_seplog tl p s h₂ →
    list_seplog (hd :: tl) a s (h₁ ⊎ h₂).
```

where `mk_cell` constructs a logical value of `log lst`. The complete example of in-place reverse-list example can be found online [2].

5.5 Example of Derived Hoare Triple using Logical Values

We formalize a variant of the first backward-reasoning form for lookup that uses a logical value lv in the mapsto formula and a "convertible" physical value pv for substitution:

$$\frac{pv \bowtie lv}{\{\exists lv\, pv.(e \mapsto lv) \star ((e \mapsto lv) \mathrel{-\!\!\star} P\{pv/x\})\}x \leftarrow\!\!\star\, e\{P\}}$$

The relation $pv \bowtie lv$ identifies pv and lv as two different views of the same value, in the absence of any padding:

```
Definition phylog_conv {g} {t : g.-typ}
  (pv : t.-phy) (lv : t.-log) :=
  ∀ h a, phy_mapsto pv a h ⟷ log_mapsto lv a h.
```

```
Inductive log : g.-typ → Type :=
| uint_log :
  int 32 → log (btyp: uint)
| sint_log :
  int 32 → log (btyp: sint)
| uchar_log :
  int 8 → log (btyp: uchar)
| ulong_log :
  int 64 → log (btyp: ulong)
| ptyp_log :
  ∀ t, int ptr_len → log (:* t)
| styp_log : ∀ t tg H,
    logs (get_fields t tg H) → log t
with logs : g.-env → Type :=
| nil_logs : logs nil
| cons_logs : ∀ h t,
  log h.2 → logs t → logs (h :: t).
```

```
Inductive log_mapsto {g} : ∀ {t : g.-typ},
    t.-log → nat → hp.t → Prop :=
| uint_log_mapsto : ∀ a h (v : (g.-btyp: uint).-log),
  hp.dom h = iota a (sizeof (g.-btyp: uint)) →
  Z◃nat a + Z◃nat (sizeof (g.-btyp: uint)) < 2^ptr_len →
  oi32◃i8 (hp.cdom h) = ⌊ log_to_uint v ⌋ →
  align (g.-btyp: uint) | a →
  log_mapsto v a h
| sint_log_mapsto : ... (* similar to uint_log_mapsto *)
| uchar_log_mapsto : ... (* similar to uint_log_mapsto *)
| ulong_log_mapsto : ... (* similar to uint_log_mapsto *)
| ptyp_log_mapsto : ... (* similar to uint_log_mapsto *)
| styp_log_mapsto : ∀ t tg H a vs h pad_sz pad,
  align t | a → logs_mapsto (get_fields t tg H) vs a h →
  pad_sz = padd (a + size (hp.dom h)) (align t) →
  hp.dom pad = iota (a + size (hp.dom h)) pad_sz →
  Z◃nat (a + size (hp.dom h)) + Z◃nat pad_sz < 2^ptr_len →
  log_mapsto (styp_log t tg H vs) a (h ⊎ pad)
with logs_mapsto {g} : ∀ (l : g.-env),
    logs l → nat → hp.t → Prop :=
| nil_logs_mapsto : ∀ a, logs_mapsto nil nil_logs a hp.emp
| cons_logs_mapsto : ∀ hd tl v vs a pad pad_sz h₁ h₂,
  pad_sz = padd a (align hd.2) → hp.dom pad = iota a pad_sz →
  Z◃nat a + Z◃nat pad_sz < 2^ptr_len →
  log_mapsto v (a + pad_sz) h₁ →
  logs_mapsto tl vs (a + pad_sz + sizeof hd.2) h₂ →
  logs_mapsto (hd :: tl) (cons_logs hd tl v vs) a (pad ⊎ h₁ ⊎ h₂).
```

Figure 3. Logical view of C values (on the left) and the corresponding mapsto connective (on the right)

This lets us read/write basic arithmetic or pointer types to/from stored structures in memory. Yet, the relation above does not make it possible to read/write complete padded structures; in other words, C programs that manipulate structures as first-class objects would require a more permissive definition of convertibility. Fortunately, the above definition is sufficient to already treat many programs, such as the non-trivial use-case we present in Sect. 7.

Here follows the formalization of the Separation logic rule above:

```
Inductive wp_lookup_back_conv {t} x H
  (e : @exp g σ (:* t)) P : assert :=
| wp_lkbr1_conv : ∀ s h
  (pv : t.-phy) (lv : t.-log), pv ⋈ lv →
  ([ e ]_ s ↦ₗ lv ∗ ([ e ]_ s ↦ₗ lv −∗
    wp_assign x H [ pv ]c P)) s h →
  wp_lookup_back_conv x H e P s h.
```

```
Lemma hoare_lookup_back_conv {t} x H
   (e : exp (:* t)) P :
 { wp_lookup_back_conv x H e P } lookup x e H { P }.
```

6. Formalization of the RFC of TLS

TLS enhances network applications by providing, on top of TCP, a cryptographic layer consisting of four protocols: packets from the Record protocol carry packets from the Handshake, Alert, or Change Cipher Spec protocols. The description of all these packet formats in the RFC [14] is semi-formal: a dedicated syntax (the *presentation language*) is introduced, but its use is not entirely consistent and many conditions remain described in prose. Despite these defects, the RFC is still a useful document. Our purpose is therefore not to provide a formal alternative to the RFC for TLS but more modestly to improve it by providing formal definitions that can be used to verify programs, while still being convincingly mapped to their informal counterparts (as will be illustrated in Fig. 5).

6.1 An Encoding of The TLS Presentation Language

The presentation language [14, §4] consists of six datatypes:

1. `opaque` is the type of bytes.
2. `T T'[n]` defines the type `T'` of *fixed-size 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-size vectors*. They comprise a payload, whose size 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 size of the payload.
4. `enum {` $e_1(v_1)$, ..., $e_n(v_n)$ `[[, (m)]] } T` defines the *enumerated* type `T`. The size 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-size vectors above.
5. Structure types are defined as being close to C structures but are in fact often used as dependent records (see Sect. 6.2).
6. *Variants* extend structures with fields whose type depends "on some knowledge that is available within the environment" [14, §4.6.1]. This "knowledge" is the (implicit) environment (e.g., the "length field" of the enclosing Handshake packet in the case of `ClientHello`, [14, §7.4.1.2]) or the value of an enumerated that can come from preceding fields in the structure (e.g., the `body` field of `Handshake`, [14, §7.4]) (in which case we are in fact dealing with a dependent record).

Putting dependent records aside for a moment (see Sect. 6.2), we encode the presentation language using the `tls_typ` inductive type below. 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 check automatically the "length field" of variable-size vectors (line 6) and enumerateds (lines 10–11), and to check divisibility constraints on fixed-length vectors. These proof obligations can be inferred automatically and hidden using notations that mimic those used in the RFC (see [2] for definitions).

```
Inductive tls_typ : Z → Z → Type :=
| opaque : tls_typ 1 1
| arr : ∀ n, tls_typ n n → ∀ m, 0 ≤ m →
  m mod n = 0 → tls_typ m m
| varr : ∀ n m (t : tls_typ n m) (k : nat) a b,
  k ≠ 0 → a ≤ b →
  b < 2^(k * 8) → 2^((k - 1) * 8) ≤ b →
  m ≤ (Z◃nat k) + b →
  tls_typ (Z◃nat k + a) (Z◃nat k + b)
| enum : ∀ k l n, uniq l →
  Zmax_lst l n < 2^(k * 8) →
  2^((k - 1) * 8) ≤ Zmax_lst l n →
  tls_typ (Z◃nat k) (Z◃nat k)
| 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.
```

Consistency of Definitions in the RFC tls_typ gives a syntax to formalize many of the packet formats, and its use of dependent types led us to spot inconsistencies in the RFC. Here is a concrete example. Fig. 4 shows on the left the Extension type [14, §7.4.1.4]: by definition, a value of type Extension is represented by at most $2+2+2^{16}$ bytes. The right part of Fig. 4 displays the Coq counterpart using tls_typs. The problem is that this type is used to define the type of the extensions field of ClientHello packets using the following declaration ([14, §7.4.1.2]. or line 27 in Fig. 5):

```
Extension extensions<0..2^16-1>;
```

The field extensions is therefore limited to $2+2^{16}$ bytes, which is not consistent with the definition of Extension that accepts values 2 bytes larger. tls_typ definitions cannot type because of this inconsistency. A fix is to restrict a bit more the definition of extension_data_type in Fig 4:

```
Definition extension_data_type :=
  opaque < 0.. 2^16 - 1 - 2> 2.
```

Another example of dubious specification is about the size of variable-size vectors. According to [14, §4.3], it "must be an even multiple of the length of a single element" which is not possible in general when variable-size vectors are nested such as in extensions_type.

6.2 Dealing with Dependent Records in Packet Formats

tls_typ does not give a syntax for the type of structure types that are in fact dependent records. When this occurs, we resort to shallow-embedding using Coq dependent records. For this purpose, we first introduce a generic decoding function for tls_typ.

```
Fixpoint decode k {n m} (t : tls_typ n m)
  (l : seq byte) : bool * seq byte := ...
Definition decoder {n m} (t : tls_typ n m) :=
  decode (depth t) t.
Definition decodep {n m} (t : tls_typ n m) l :=
  let: (a, l'):= decoder t l in
  a && (size l' = 0).
```

decoder matches a list l of bytes against a tls_typ t (recursion is on the depth of t). decodep decides whether a list of bytes conforms to a tls_typ. 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 : seq byte → bool) : Type := {
  body :> seq byte ;
  decodable : p body }.
```

As an example of dependent record from TLS, let us consider the specification of ClientHello packets ([14, § 6.2.1], reproduced on the left of Fig. 5). There is a dependency between the field extensions and its predecessors: the predicate extensions_present decides the presence of extensions depending on the contents of session_id, cipher_suites, and compression_methods. This relation is only expressed in prose in the RFC. In the Coq formalization, each field is expressed as a packet of some predicate. Checking whether extensions_present is true can be expressed naturally. Contrary to tls_typ, we have no generic decoder for dependent records but we can write systematically decoders as follows:

```
Definition ClientHello_decoder {n}
  (_ : decodep uint24 n) l : bool * seq byte :=
let (a1, l1) := decoder ProtocolVersion l in
let (a2, l2) := decoder Random l1 in
let (a3, l3) := decoder SessionID l2 in
let (a4, l4) := decoder cipher_suites_type l3 in
let (a5, l5) := decoder compression_methods_type l4 in
if extensions_present ( nat◃ bytes n)
    (size l2 - size l3) (size l3 - size l4)
    (size l4 - size l5) then
  let (a6, l6) := decoder extensions_type l5 in
  ([&& a1, a2, a3, a4, a5 & a6], l6)
else
  ([&& a1, a2, a3, a4 & a5], l5).

Definition ClientHellop {n} (H : decodep uint24 n)
    (l : seq byte) : bool :=
  let (a, l') := ClientHello_decoder H l in
  (a && (size l' = 0)).
```

ClientHellop_decoder is parametrized by n which is a list of bytes coming from an outer packet[3]. Using above ideas, we formalized many of the packet formats of the TLS protocol; this gives us all the definitions we need for formal verification of source code.

7. Verification of PolarSSL ClientHello Parsing

We experiment our framework to the verification of the source code of a parsing function from PolarSSL [23], an implementation of the TLS protocol. We verify the function ssl_parse_client_hello (library/ssl_srv.c, version 0.14.0) that parses initialization ClientHello packets. Fig. 6 displays a byte-level presentation of the packets that ssl_parse_client_hello is intended to parse. It is a Record protocol packet, containing a Handshake protocol packet (starting at byte index 5), itself being a ClientHello packet.

7.1 The Main Data Structures

The central data structure in PolarSSL is of type ssl_context. It 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, the field "length" is the length of the session id), cipher suites of the server (field "ciphers"), and the nonce for this session (field "randbytes"). The other fields ("in_hdr", "in_msg", "in_left") are for navigation into the buffer that stores the bytes coming from the network (bottom part of Fig. 6). Here follows the corresponding typ definition (Fig. 7 provides a pictorial representation):

```
Definition ssl_ctxt :=
 ("state",         btyp sint) ::
 ("major_ver",     btyp sint) ::
 ("minor_ver",     btyp sint) ::
 ("max_major_ver", btyp sint) ::
 ("max_minor_ver", btyp sint) ::
 ("session",       ptyp ssl_session) ::
 ("in_hdr",        ptyp (btyp uchar)) ::
```

[3] ClientHello packets belong to the Handshake protocol, whose packets are encapsulated into the Record protocol.

```
enum {
    signature_algorithms(13), (65535)
} ExtensionType;

struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;
```

```
Definition signature_algorithms := 13.
Definition ExtensionType :=
   enum 2 { [:: signature_algorithms] } 65535.

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

Figure 4. Formalization of the `Extension` type, TLS RFC on the left, Coq formalization on the right

```
struct {
 uint8 major;
 uint8 minor;
} ProtocolVersion;

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

opaque SessionID<0..32>;

uint8 CipherSuite[2];

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

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;
```

```
Definition ProtocolVersion :=
  struct{ ("major", uint8) ; ("minor", uint8) }.
Definition Random :=
  struct{ ("gmt_unix_time", uint32) ;
          ("random_bytes", opaque [ 28 ])}.
Definition SessionID := opaque < 0 .. 32 > 1.
Definition CipherSuite := uint8 [ 2 ].
Definition cipher_suites_type :=
  CipherSuite < 2 .. 2^16 - 2 > 2.
Definition CompressionMethod := enum 1 { [:: null] } 255.
Definition compression_methods_type :=
  CompressionMethod < 1 .. (2 ^ 8 - 1) > 1

Definition Hello_sz sid := fixed_sz ProtocolVersion +
 fixed_sz Random + fixed_sz SessionID + Z◁nat sid.
Definition ClientHello_sz sid cys cpm := Hello_sz sid +
 fixed_sz cipher_suites_type + Z◁nat cys +
 fixed_sz compression_methods_type + Z◁nat cpm.
Definition extensions_present n sid cys cpm :=
 ClientHello_sz sid cys cpm < Z◁nat n.

Structure ClientHello_packet {n} (H: decodep uint24 n) := {
 client_version : packet (decodep ProtocolVersion);
 random : packet (decodep Random) ;
 session_id : packet (decodep SessionID) ;
 cipher_suites : packet (decodep cipher_suites_type) ;
 compression_methods :
   packet (decodep compression_methods_type) ;
 extensions : packet ( select(
   extensions_present ( nat◁bytes  n) (size session_id)
   (size cipher_suites) (size compression_methods) \) {
    (false, decodep struct{}) ;
    (true, decodep extensions_type) } ) }.
```

Figure 5. Formal specification of a ClientHello packet, TLS RFC on the left, Coq on the right. See Fig. 4 for the `Extension` type.

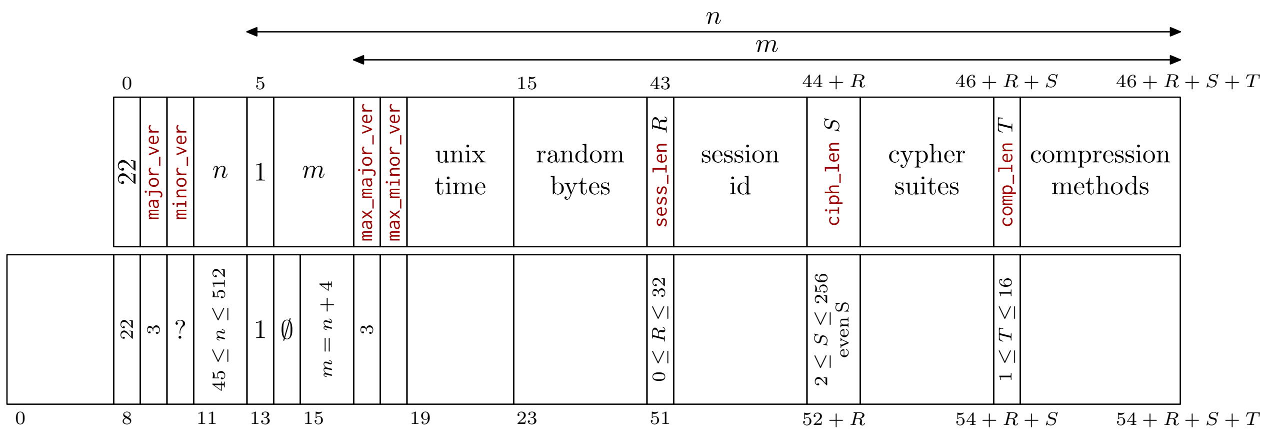

Figure 6. Packet format as described by the RFC [14] on the top, and as required by PolarSSL [23] at the bottom.

```
 ("in_msg",        ptyp (btyp uchar)) ::
 ("in_left",       btyp sint) ::
 ("fin_md5",       md5_context) ::
 ("fin_sha₁",      sha₁_context) ::
 ("ciphers",       ptyp (btyp sint)) ::
 ("randbytes",     ptyp (btyp uchar)) :: nil.
Definition ssl_context := styp (mkTag "ssl_context").
Definition ssl_sess :=
 ("cipher", btyp sint) ::
 ("length", btyp sint) ::
 ("id",     ptyp (btyp uchar)) :: nil.
Definition ssl_session := styp (mkTag "ssl_session").
```

We can now define the type context g containing the definitions of PolarSSL for `ssl_context`, `ssl_session`, etc.:

```
Definition g := \wfctxt{"ssl_context" ▹ ssl_ctxt,
 "ssl_session" ▹ ssl_sess, "md5_context" ▹ md5_cont,
 "sha₁_context" ▹ sha₁_cont, ∅ }.
```

7.2 The Parsing Function

Roughly, the original C function `ssl_parse_client_hello` is around 160 lines long, around 85 lines if we remove comments and debugging information; once converted to Coq, it is around 115 lines. C expressions are almost ported as they are. Yet, we need to adapt the original code to structured control-flow by replacing gotos with if-then-else's and by merging returns, and, since the expression language is side-effect free, some C expressions need to be split into several commands using temporary variables. This is systematic enough to be automated, and indeed this is actually what is done in other proof assistant-based verification projects such as seL4 [29].

The formal model of the PolarSSL function that parses ClientHello packets of TLS version 1.0 can be found online [2] or in Appendix A (Fig. 8 and 9). Its variables are typed according to the following environment σ:

```
Definition σ : g.-env := ("ret", btyp: sint) ::
("ssl", :* (g.-typ: ssl_context)) ::
("buf", :* (btyp: uchar)) :: ("n", btyp: sint) ::
("sess_len", btyp: sint) ::
("ciph_len", btyp: sint) ::
("comp_len", btyp: sint) :: ("i", btyp: sint) ::
("j", btyp: sint) :: ("p", :* (btyp: uchar)) :: ...
```

Dealing with External Functions `ssl_parse_client_hello` calls several library functions, such as PolarSSL-specific functions (e.g., `ssl_fetch_input`, a function that reads bytes from the input socket to fill a buffer), or standard C functions (`memset`, `memcpy`, etc.). For the time being, we just axiomatize their correctness in the form of Separation-logic triples.

Since we do not have yet a proper encoding of function calls, we formalize external functions by augmenting the `cmd` type with axioms. To encode call-by-value semantics, we also axiomatize the fact that no variable is modified by an external function. Here follows the example of `memcpy`:

```
(* void *memcpy(void *d, const void *s, size_t l); *)
Definition size_t := g.-btyp: uint.
Definition void_p := g.-typ: ptyp (btyp uchar).
Axiom memcpy : ∀ x, env_get x σ = ⌊ void_p ⌋ →
  @exp g σ void_p → @exp g σ void_p →
  @exp g σ size_t → cmd.

Lemma memcpy_triple x H e l D S
 (ret src : exp (:* (btyp: uchar))) :
 size S = nat◃u l → size D = nat◃u l →
 { !b b[ e = [uint◃i32 l]c ] ⋆ src ↦ S ⋆ ret ↦ D }
 memcpy x H ret src e
 { !b b[ e = [uint◃i32 l]c ] ⋆ src ↦ S ⋆ ret ↦ S }.
```

7.3 Verification Goal, Approach and First Results

We want to prove that, given some input from the network (modeled as a list of bytes `SI`, for “socket input”), `ssl_parse_client_hello` either fails (by returning a non-zero value, assertion `error`) or succeeds (by returning 0, assertion `success`) in checking that the incoming packet is valid:

```
Lemma POLAR_parse_client_hello_triple ...
  { init_ssl_var ⋆ init_bu ⋆ init_rb ⋆ init_id ⋆
     init_ses ⋆ init_ssl ⋆ init_ciphers }
  ssl_parse_client_hello
  { error ∨ (success ⋆ final_bu ⋆ final_rb ⋆
     final_id ⋆ final_ssl ⋆ final_ciphers ⋆
     PolarSSLRecordClientHellop SI) }.
```

The precondition specifies the initial state: the values of the variables and the state of the heap. This is captured by a Separation logic formula that formalizes the left part of Fig. 7. `init_ssl_var` gives the initial value of the `"ssl"` variable. `init_bu` defines the existence of a buffer for the incoming bytes; it is a sensitive storage space and verification must make sure that it is not overrun. Similarly, `init_rb` (resp. `init_id`, `init_ses`, `init_ssl`, `init_ciphers`) is space for the nonce of the TLS connection (resp. the initial state of the session id data structure, of the session id, of the `ssl_context` data structure, and the ciphers known by the server).

By way of example, let us look at the `init_ssl` predicate. It formalizes in terms of a mapsto formula the central data structure of Fig. 7. Initially, the stage of the protocol is `S74.client_hello`[4] and `in_left` is 0 (no byte read so far), version fields are uninitialized, and pointer fields are set to point to other data structures:

```
let init_ssl := %"ssl" ↦le mk_ssl_context
     (Z2u 32 S74.client_hello)
     majv0 minv0 mmaj0 mmin0
     (ptr◃ptyp ses)
     (ptr◃ptyp bu +i Z2u ptr_len 8)
     (ptr◃ptyp bu +i Z2u ptr_len 13)
     0₃₂ md5s sha1s
     (ptr◃ptyp ciphers) (ptr◃ptyp rb) in
```

(`Z2u` converts integers to binary representations, $+_i$ is hardware addition). Parameters `md5s` and `sha1s` are for cryptographic functions but we are not concerned with them in parsing.

The postcondition first specifies that 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. 7 (right part) provides a pictorial representation that can be compared with the initial heap state (on the left). `final_bu` says that the buffer array is filled with the incoming bytes. `final_rb` says that the array for random bytes `RB` has been half-filled with the client nonce. `final_id` says that the session id has been saved. `final_ssl` says that the state of the protocol has moved to `S74.server_hello` state, etc. `final_ciphers` says that there is a two-bytes index of a cipher in `SI` on which the server has agreed.

Most importantly, `PolarSSLRecordClientHellop` specifies that the incoming bytes form a valid ClientHello packet:

```
Definition PolarSSLRecordClientHellop l :=
 (* the Record packet is a Handshake *)
 !(Z◃u (l_0) = S621.handshake) ⋆
 (* the Handshake packet is a ClientHello *)
 !(Z◃u (l_(nat◃Z (S74.Handshake_hd + 1))) =
   S74.client_hello) ⋆
 (* requested protocol version is 3 *)
 !(Z◃u (l_req_maj) = S621.SSLv30_maj) ⋆ ...
```

[4] S74 is a Coq module named after the Section 7.4 of the RFC. We use this nomenclature to keep track of all the magic numbers from the RFC.

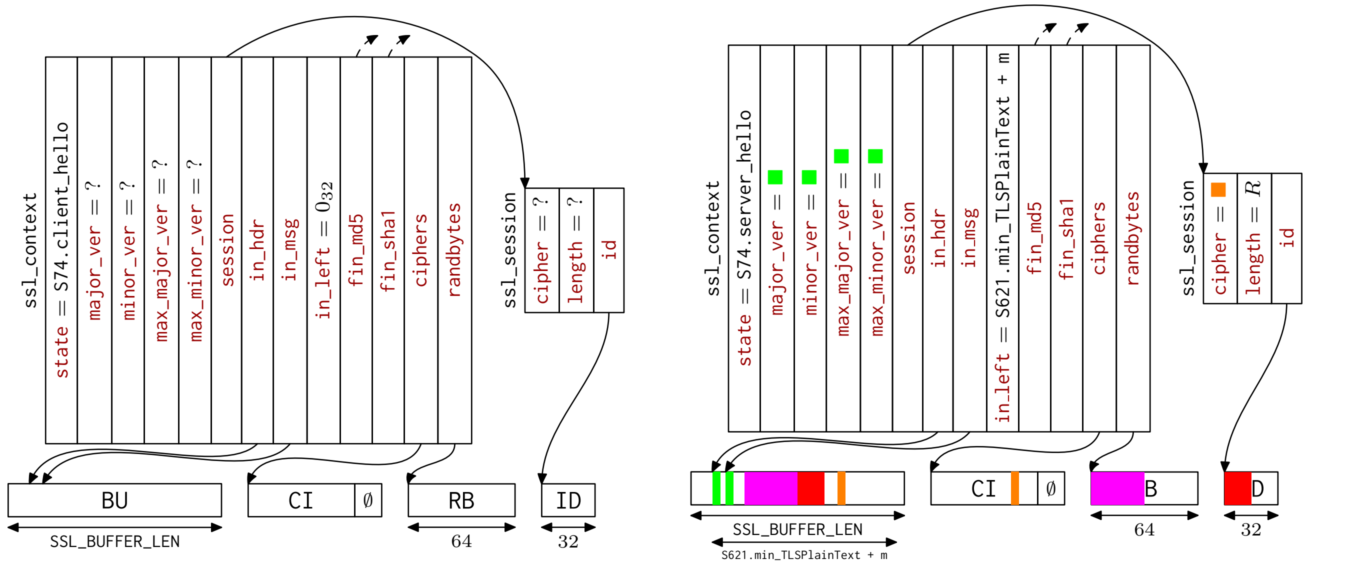

Figure 7. Pictorial representation of PolarSSL's memory state before and after parsing a ClientHello packet

Current Status At the time of this writing, verification is not yet completed but we made significant progress. We passed the point where the function parses the compression methods header, that makes for more than two-thirds of the code. What is left is a search to find out whether the cipher asked by the client is known by the server. So, we already checked most conditions in `PolarSSLRecordClientHellop`, for example that the incoming packet is a Handshake, that it contains a ClientHello packet, etc. Ultimately, we should be able to show that the predicate `PolarSSLRecordClientHellop` above implies the more general specification `ClientHellop`[5] Sect. 6.2.

Errors found so far We already found bugs in the course of verification. Checks are indeed not sufficient to ensure well-formedness of packets. Concretely, at the beginning of the function, PolarSSL retrieves the length of the Handshake packet and checks its value:

```
n = ( buf[3] << 8 ) | buf[4];
if( n < 45 || n > 512 )
{ return(POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO); }
```

It later retrieves the length of the session id and also check its value:

```
sess_len = buf[38];
if( sess_len < 0 || sess_len > 32 )
{ return(POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO); }
```

But it does not check that the session id can actually be contained into the Handshake message (in other words, that `sess_len` is not too large w.r.t. `n`). Formal verification stumbles because we try to `memcpy` the contents of the session id to a n bytes buffer that may not be large enough to welcome `sess_len` more bytes.

8. Related Work

Mechanization of Separation Logic Tuch proposes a formalization in the Isabelle proof-assistant of Separation logic for C with an application to a memory allocator [28]. A trusted C-to-HOL translation is responsible for encoding C types as Isabelle/HOL records together with lemmas [28, §5.3]; padding is encoded in the form of extra fields [28, p.140]. Proofs do not fail when types are correct but "this is fragile and does not scale well" [28, p.146]. In contrast, we formalize the alignment/sizeof functions completely in Coq and therefore avoid external trusted machinery. Tuch favors a variant of Burstall-Bornat memory model for heap access, but this causes problems with C structures: What is the type of the start address? The type of the whole structure or of the first field? In comparison, we favor direct, byte-level heap access annotated with types. It is in our Separation logic that we accommodate a logical view of datatypes to hide low-level details such as padding (Sect. 5.4).

There are several other mechanizations of Separation logic in proof-assistants but they address archetypal languages, so that encoding of types and typed expressions is not as important as in our case. Ynot [20] is a Coq axiomatization of Hoare Type Theory allowing for Separation logic-like reasoning for an imperative language with advanced features such as strong updates. Bedrock [13] is a framework that emphasizes "mostly automated verification". It uses an "idealized machine language" with arbitrary-precision words and infinite memory (this limitation seems to be addressed in the latest implementation), without notion of alignment or padding. In contrast, our formalization of C takes into account realistic hardware constraints. In order to perform verification (Sect. 7.3), we use semi-automation using tactics like Appel [4].

Winwood et al. propose a different approach to interactive verification of C programs [29]. There is no Separation logic per se, but Hoare logic is used to establish simulations [29, Sect. 5.2]. Application of this approach to PolarSSL verification would require the construction of a reference implementation, what would be another way to formalize the RFC for TLS.

Formalization of the Semantics of C CompCert is a comprehensive formalization of C that supports most of ANSI C. We restrict ourselves to the core subset of C because, since we are aiming at verification of C programs, we need to equip instructions with reasoning facilities, in particular Separation logic rules. In CompCert, Separation logic was a side-project for the intermediate language only [5]. In CompCert, recursive structures must be encoded "structurally": inner fields can only refer to enclosing fields [9, Sect. 2.1].

[5] It will not be possible to guarantee that parsing succeeds for any correct packet because PolarSSL has several restrictions (e.g., Record packets are assumed to be larger that the Handshake packets they embed—"Theoretically, a single handshake message might span multiple records, but in practice this does not occur.", [24, p.70]; also, PolarSSL does not handle packets as large as what is allowed by the RFC).

This obviates the need to carry a typing environment. We did not chose this direction because it requires to rework the types from the original program. We do carry around a typing environment: this makes the writing of the sizeof function more subtle (Sect. 2.2) but it becomes more natural to write mutually recursive structures (Sect. 2.3). Also, we use dependent types so that the Coq type-checker guarantees that programs are well-typed, thus obviating the need for explicit type-checking functions (Sect. 4). There are side-effects in CompCert expressions but when it comes to formal reasoning this is something that is difficult to handle (this is also a simplification made in [29]).

One of the last formalization of C semantics is provided by Ellison et al. [15]. It consists of an executable semantics built on top of the Maude rewriting system, that allows for the derivation of an interpreter and a debugger, thus paving the road for a formal runtime analysis system. However, it has not been designed for formal verification so that it remains elusive if, and how, one might build a reasoning system on top of it.

Nita et al. formally explore the platform dependency of the C semantics [21]. By collecting the platform-dependent parts of a program, they build a logical formula encoding memory layout conditions under which the program is memory-safe. The theory is wrapped up into a static safety analysis tool. While our model of C instantiates some platform dependent values (such as pointer size), our library should become parametrizable with a reasonable amount of work. Still,our work is oriented towards verification of functional properties, which are more general than safety.

Specification of Network Packets The formal verification of a parsing function naturally calls for a formalization of network packet formats. This is an issue that we have already tackled [1] with parsing combinators based on invertible syntax descriptions to simplify the programming of reference implementations. The formalization introduced in Sect. 6 can be seen as a stripped-down version with more emphasis on producing a formalization that can be convincingly compared with the RFC. This turned out to be important to handle the magic numbers that pop up in implementations. Producing formal specifications of packet formats has been a long-standing issue for which types emerged as a promising solution [19].

Automated Source Code Verification The software stack Frama-C/Jessie/Why3 proposes a pragmatic approach for verification by relaxing the minimal trusted base constraint. Frama-C is a plugin-based framework for analysis of C source code. Hoare-style annotations can be processed by the Jessie plugin to generate verification conditions. These goals are generated for Why3 (a framework for expressing multi-sorted first-order theories) that can discharge them using a wide set of automated theorem provers, or generate Coq goals as a last resort. The whole stack has been used for example to verify C functions for numerical analysis [10]. No experiment about communication protocols seems to have been carried out yet. Regarding automated verification of TLS implementations, Bhargavan et al. successfully verified a small reference implementation but written with a functional language [6].

References

[1] R. Affeldt, D. Nowak, Y. Oiwa. Formal network packet processing with minimal fuss: invertible syntax descriptions at work. Proc. of the 6th Wksp. Programming Languages meets Program Verification, pp.27–36. ACM, 2012

[2] R. Affeldt, N. Marti. Towards Formal Verification of TLS Network Packet Processing in C. Coq documentation. `http://staff.aist.go.jp/reynald.affeldt/coqdev`

[3] R. Affeldt, D. Nowak, K. Yamada. Certifying Assembly with Formal Security Proofs: the Case of BBS. Sci. Comput. Program. 77(10–11):1058–1074 (2012)

[4] A. W. Appel. Tactics for Separation Logic. Early draft. Jan. 13, 2006. `http://www.cs.princeton.edu/~appel/papers/septacs.pdf`

[5] A. W. Appel, S. Blazy. Separation Logic for Small-Step Cminor. Proc. of the 20th Intl. Conf. on Theorem Proving in Higher Order Logics, vol. 4732 of LNCS, pp.5–21. Springer, 2007

[6] K. Bhargavan, C. Fournet, R. Corin, E. Zalinescu. Verified Cryptographic Implementations for TLS. ACM Trans. Inf. Syst. Secur. 15(1):3 (2012)

[7] N. Benton, C.-K. Hur, A. Kennedy, C. McBride. Strongly Typed Term Representations in Coq. J. Autom. Reasoning 49(2):141–159 (2012)

[8] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, K. Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. Proc. of the 33rd SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp.55–66. ACM, 2006

[9] S. Blazy, X. Leroy. Mechanized semantics for the Clight subset of the C language. J. Autom. Reasoning 43(3):263–288 (2009)

[10] S. Boldo, F. Clément, J. Filliâtre, M. Mayero, G. Melquiond, P. Weis. Wave Equation Numerical Resolution: a Comprehensive Mechanized Proof of a C Program. J. Autom. Reasoning. To appear

[11] E. Brady. IDRIS—Systems Programming meets Full Dependent Types. Proc. of the 5th Wksp. Programming Languages meets Program Verification, pp.43–54. ACM, 2011

[12] The Coq Proof Assistant: Reference Manual. Ver. 8.4. INRIA, 2012

[13] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. Proc. of the 32nd SIGPLAN Conf. on Programming Language Design and Implementation, pp.234–245. ACM, 2011

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

[15] C. Ellison, G. Rosu. An Executable Formal Semantics of C with Applications. Proc. of the 39th SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp.533–544. ACM, 2012

[16] J.-C. Filliâtre. Verifying Two Lines of C with Why3: An Exercise in Program Verification. Proc. of the 4th Intl. Conf. on Verified Software: Theories, Tools, Experiments, vol. 7152 of LNCS, pp.83–97. Springer, 2012

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

[18] X. Leroy. The Compcert verified compiler, Commented Coq development. Version 1.11, 2012-07-13 `http://compcert.inria.fr/doc/`

[19] P. J. McCann, S. Chandra. Packet Types: Abstract specifications of network protocol messages. Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication, pp.321–333. ACM, 2000

[20] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, L. Birkedal. Ynot: dependent types for imperative programs. Proc. of the 13th SIGPLAN Intl. Conf. on Functional Programming, pp.229–240. ACM, 2008

[21] M. Nita, D. Grossman, C. Chambers. A theory of platform-dependent low-level software. Proc. of the 35th SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp.209–220. ACM, 2008

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

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

[24] E. Rescorla. SSL and TLS: Designing and Building Secure Systems. 11th Printing. Addison Wesley, 2000

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

[26] J. C. Reynolds. An Introduction to Separation Logic (Preliminary Draft). `http://www.cs.cmu.edu/~jcr/copenhagen08.pdf`. Oct. 23, 2008

[27] R. C. Seacord. Secure Coding in C and C++. Addison Wesley, 2006

[28] H. Tuch. Formal Verification of C Systems Code. J. Autom. Reasoning 42(2-4):125–187 (2009)

[29] S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, M. Norrish. Mind the Gap: A Verification Framework for Low-Level C. Proc. of the 22nd Intl. Conf. on Theorem Proving in Higher Order Logics, vol. 5674 of LNCS, pp.500–515. Springer, 2009

A. Formal Model of the `ssl_parse_client_hello` Function

```
Definition ssl_parse_client_hello :=
  "ret" ←ssl_fetch_input(%"ssl", [ 5 ]sc) ;
  If b[ %"ret" ≠ [ 0 ]sc ] Then
    ret
  Else (
  "buf"  ←∗  %"ssl" ⇒ in_hdr ;
  "_buf0_"  ←∗  %"buf" ;
  If b[ (%"_buf0_" & [ -128 ]8sc) ≠ [ 0 ]8sc ] Then
    "ret" ← [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c ; ret
  Else (
  If b[ %"_buf0_" ≠ [ SSL_MSG_HANDSHAKE ]c ] Then
    "ret" ← [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c ; ret
  Else (
  "_buf1_"  ←∗  %"buf" +p [ 1 ]sc ;
  If b[ %"_buf1_" ≠ [ SSL_MAJOR_VERSION_3 ]c ] Then
    "ret" ← [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c ; ret
  Else (
  "_buf3_"  ←∗  %"buf" +p [ 3 ]sc ;
  "_buf4_"  ←∗  %"buf" +p [ 4 ]sc ;
  "n" ← (( (int) %"_buf3_") ≪ [ 8 ]sc) | (int) %"_buf4_" ;
  If b[ %"n" < [ 45 ]sc ] Then
    "ret" ← [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c ; ret
  Else (
  If b[ %"n" > [ 512 ]sc ] Then
    "ret" ← [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c ; ret
  Else ((
  "ret" ←ssl_fetch_input(%"ssl", [ 5 ]sc + %"n") ;
  If b[ %"ret" ≠ [ 0 ]sc ] Then
    ret
  Else (
  "buf"  ←∗  %"ssl" ⇒ in_msg ;
  "_n0_"  ←∗  %"ssl" ⇒ in_left ;
  "n" ← %"_n0_" − [ 5 ]sc ;
  md5_update(%"ssl" ⇒ fin_md5, %"buf", %"n") ;
  sha1_update(%"ssl" ⇒ fin_sha1, %"buf", %"n") ;
  "_buf0_"  ←∗  %"buf" ;
  If b[ %"_buf0_" ≠ [ SSL_HS_CLIENT_HELLO ]c ] Then
    "ret" ← [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c ; ret
  Else (
  "_buf4_"  ←∗  %"buf" +p [ 4 ]sc ;
  If b[ %"_buf4_" ≠ [ SSL_MAJOR_VERSION_3 ]c ] Then
    "ret" ← [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c ; ret
  Else (
  %"ssl" ⇒ major_ver  ∗←  (int) ([ SSL_MAJOR_VERSION_3 ]c) ;
  "_buf5_"  ←∗  %"buf" +p [ 5 ]sc ;
  %"ssl" ⇒ minor_ver  ∗←  (int) (%"_buf5_"≤[SSL_MINOR_VERSION_2]c ? %"_buf5_" : [SSL_MINOR_VERSION_2]c) ;
  %"ssl" ⇒ max_major_ver  ∗←  (int) %"_buf4_" ;
  %"ssl" ⇒ max_minor_ver  ∗←  (int) %"_buf5_" ; (
  "_it_"  ←∗  %"ssl" ⇒ randbytes ;
  "_it_" ←memcpy(%"_it_", %"buf" +p [ 6 ]sc, [ 32 ]uc) ;
  "_buf1_"  ←∗  %"buf" +p [ 1 ]sc ;
  (* continue in Fig. 9 *)
```

Figure 8. Formal model of the function `ssl_parse_client_hello` (first half, see Fig. 9 for the second half)

```
(* continued from Fig. 8 *)
If b[ %"_buf1_" ≠ [ 0 ]8sc ] Then
  "ret" ← [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c ; ret
Else (
"_buf2_"  ←∗  %"buf" +p [ 2 ]sc ;
"_buf3_"  ←∗  %"buf" +p [ 3 ]sc ;
If b[ %"n" ≠ [ 4 ]sc + (( (int) %"_buf2_" ≪ [ 8 ]sc) | (int) %"_buf3_") ] Then
  "ret" ← [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c ; ret
Else (
"_buf38_"  ←∗  %"buf" +p [ 38 ]sc ;
"sess_len" ← (int) %"_buf38_" ;
If b[ %"sess_len" < [ 0 ]sc || %"sess_len" > [ 32 ]sc ] Then
  "ret" ← [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c; ret
Else
"_ssl_session_0_"  ←∗  %"ssl" ⇒ session ; (
%"_ssl_session_0_" ⇒ length  ∗← %"sess_len";
"_it_"  ←∗  %"_ssl_session_0_" ⇒ id ;
"_it_" ←memset(%"_it_", [ 0 ]sc, [ 32 ]uc) ;
"_ssl_session_0_length_"  ←∗  %"_ssl_session_0_" ⇒ length ;
"_it_"  ←∗  %"_ssl_session_0_" ⇒ id ;
"_it_" ←memcpy(%"_it_", %"buf" +p [ 39 ]sc, ((UINT) %"_ssl_session_0_length_") ) ;
"_buf39_plus_sess_len_"  ←∗  %"buf" +p ([ 39 ]sc + %"sess_len") ;
"_buf40_plus_sess_len_"  ←∗  %"buf" +p ([ 40 ]sc + %"sess_len") ;
"ciph_len" ← ( (int) %"_buf39_plus_sess_len_" ≪ [ 8 ]sc) | ( (int) %"_buf40_plus_sess_len_") ; (
If b[ %"ciph_len" < [ 2 ]sc || %"ciph_len" > [ 256 ]sc || %"ciph_len" \% 1 ≠ [ 0 ]sc ] Then
  "ret" ← [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c ; ret
Else (
"comp_len'"  ←∗  %"buf" +p ([ 41 ]sc + %"sess_len" + %"ciph_len") ;
"comp_len" ← (int) %"comp_len'" ;
If b[ %"comp_len" < [ 1 ]sc || %"comp_len" > [ 16 ]sc ] Then
  "ret" ← [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c ; ret
Else (
"_goto_have_cipher_" ← [ 0 ]sc ;
For("i" ← [ 0 ]sc ;
    "_ssl_ciphers_"  ←∗  %"ssl" ⇒ ciphers; "_ssl_ciphers_i_"  ←∗  %"_ssl_ciphers_" +p %"i" ,
    %"_goto_have_cipher_" = [ 0 ]sc && %"_ssl_ciphers_i_" ≠ [ 0 ]sc ;
    "i" ++ ){
  For("j" ← [ 0 ]sc ; "p" ← %"buf" +p [ 41 ]sc +p %"sess_len" ;
      %"_goto_have_cipher_" = [ 0 ]sc && %"j" < %"ciph_len" ;
      nop ){
    "_p0_"  ←∗  %"p" ;
    If b[ %"_p0_" = [ 0 ]8uc ] Then
      "_p1_"  ←∗  %"p" +p [ 1 ]sc ;
      If b[ (int) %"_p1_" = %"_ssl_ciphers_i_" ] Then
        "_goto_have_cipher_" ← [ 1 ]sc
      Else
        "j" +← [ 2 ]sc ; "p" +p← [ 2 ]sc
    Else
      "j" +← [ 2 ]sc ; "p" +p← [ 2 ]sc
    }
} ;
If b[ %"_goto_have_cipher_" ≠ [ 0 ]sc ] Then
  %"_ssl_session_0_" ⇒ cipher  ∗←  %"_ssl_ciphers_i_" ;
  %"ssl" ⇒ in_left  ∗←  [ 0 ]sc ;
  "_ssl_state_"  ←∗  %"ssl" ⇒ _state ;
  %"ssl" ⇒ _state  ∗←  %"_ssl_state_" + [ 1 ]sc ;
  "ret" ← [ 0 ]sc ; ret
Else
  "ret" ← [ POLARSSL_ERR_SSL_NO_CIPHER_CHOSEN ]c ; ret)))))))))))))))).
```

Figure 9. Formal model of the function `ssl_parse_client_hello` (second half, see Fig. 8 for the first half)