An Intrinsic Encoding of a Subset of C and its Application to TLS Network Packet Processing

Reynald AFFELDT and Kazuhiko SAKAGUCHI
National Institute of Advanced Industrial Science and Technology

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 report here on the construction in the Coq proof-assistant of libraries to model, specify, and verify C programs to process TLS packets. We provide in particular an encoding of the core subset of C whose originality lies in its use of dependent types to statically guarantee 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 (namely, PolarSSL) of which we specify and verify a parsing function for network packets. Thanks to this experiment, we were able to spot ambiguities in the RFC and to correct bugs in the C source code.

1. INTRODUCTION

TLS (Transport Layer Security) [DR08] is such a widespread security protocol that errors in its implementation can have disastrous consequences. For illustration, the Heartbleed bug found in OpenSSL [OPE] (CVE-2014-0160) allowed theft of servers' private keys, thus compromising completely any security guarantee. This responsibility for security 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 [BFN+06]; 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 [Bra11]; yet, such implementations continue to be developed in C for performance reasons.

Our purpose is to provide a framework in the Coq proof-assistant [CDT14] 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. The originality of our encoding of C is the use of dependent types to provide what is called an "intrinsic" encoding [BHKM12], i.e., an encoding such that only correctly-typed C expressions and C commands can be represented (see Section 4). An intrinsic encoding helps its user by detecting modeling errors as soon as the target C program is modeled. It also helps during formal proof. Indeed, since the proofs that the C expressions and the C commands are correctly-typed are part of the syntax, they can be naturally hidden using Coq notations, and therefore they do not clutter the display during formal proof. An intrinsic encoding also helps when developing the verification framework because having to deal with only correctly-typed programs reduces the number of error cases to be treated when developing the semantics and the related lemmas. We do not think that there are any significant drawback to the use of an intrinsic encoding, except maybe less obvious reporting of errors when developing the framework in Coq. In order to be able to develop our intrinsic encoding of the core subset of C, we start by providing an encoding of C types parameterized with a type context (Section 2) together with functions to calculate alignment and sizeof information (Section 3).

Our verification framework is based on Separation logic [Rey02], a variant of Hoare logic that deals with pointers, the latter being pervasively used in network packet processing. We have adapted and encoded the standard Separation logic for our model of C, and equipped it with the expected lemmas such as the frame rule. Because direct manipulation of memory in terms of bytes complicates formal specification by revealing details such as padding, we also provide reasoning rules that treat C data structures in a "logical" way. This proves useful even for a simple example such as the mandatory in-place list reversal (Section 5).

The task of processing network packets is disciplined by various standards. RFCs (Requests For Comments) are published by the IETF (Internet Engineering Task Force) to document Internet-related innovations. Strictly speaking, RFCs are not standards, but many of them are authoritative documents. In practice, the RFC for TLS [DR08] acts as a de facto standard. It describes in particular the format of network packets, but in a semi-formal fashion. 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. 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 (Section 6).

Finally, we apply the above framework to the formal verification of a parsing function from an existing implementation of TLS, namely PolarSSL [POL] (Section 7). This experiment was originally motivated by the observation that many security vulnerabilities of mainstream implementations of TLS are caused by incomplete parsing of network packets. It seems difficult for a programmer to implement all the checks that are specified in the RFC, in particular those regarding the length of payloads, that are partly implicit. Recently, this observation has been confirmed by security vulnerabilities with much media coverage such as the OpenSSL Hearbleed bug (CVE-2014-0160) or the GnuTLS buggy parsing of session IDs (CVE-2014-3466). In fact, we discovered similar implementation errors while performing formal verification of a parsing function of PolarSSL. Concretely, we formalize the function

from PolarSSL [POL] that parses initialization packets[1], specify it w.r.t. the formal RFC, and verify it. Section 7.5 comments more specifically on the implementation errors we found.

Outline. In Section 2, we introduce a non-structural encoding of the core subset of C types. In Section 3, we explain how we formalize alignment and sizeof calculations that conform with the C standard. In Section 4, we use the encoding developed in the previous sections to build a dependently-typed encoding of the core subset of C. In Section 5, we provide an encoding of Separation logic for reasoning about C programs. In Section 6, we provide an encoding of the RFC for TLS [DR08] that helped us find ambiguities in the original prose document. In Section 7, we apply the above framework to the formal verification of a parsing function of PolarSSL [POL], in the source code of which we managed to find implementation errors. We review related work in Section 8 and conclude in Section 9.

About notations in this paper. We display the Coq formalization almost as it is, using only a few obvious non-ascii symbols to ease reading. Since we work with the SSReflect extension [GM10] of Coq, we explain, when they appear, SSReflect's idiosyncrasies, as well as notations and definitions that are specific to SSReflect.

2. AN ENCODING OF C TYPES USING TYPE CONTEXTS

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 [Ler12], but at the price of an indirect encoding of mutually recursive types. We choose to refer to C structures by names, using a *type context*. The result is a direct type representation, but also a more involved mechanization because termination of type traversal is not structural. The next section (Section 3) completes our model of C types by providing alignment and size calculations.

2.1 Encoding of Type Contexts

We define our subset of the C types as an inductive type[2]:

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

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

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

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

[1]Even recent security bugs can be found in such apparently well-scrutinized functions (e.g., CVE-2011-0014 for ClientHello in OpenSSL).
[2]We have also extended `typ` to deal with arrays of structures (see [AMS14]) but since we do not rely on this extension in our case study we skip this explanation.

([eqType of t] is the type t with a decidable equality; we borrow this definition from the SSReflect library.)

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 included in the domain of the type context (otherwise it is "incomplete" in C parlance). Put formally:

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

(The function tags collects the tags 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 to be complete when all the types in its codomain are covered. Completeness can therefore be decided by checking whether the set of tags in the codomain of the context is included in the domain:

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

(The notation ⌊ f ⌋ stands for Some f; the function unzip2 retrieves the second projections of a list of pairs.)

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

```
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.
```

(The construct if t1 is p then t2 else t3 matches the term t1 against the pattern p and reduces accordingly to t2 or t3; it is an extension of Coq provided by SSReflect. The notation x.2 (resp. x.1) is for the second (resp. first) projection of the pair x. The notation == is for the boolean equality of types with a decidable equality.) This computable relation states that the tag tg_1 refers to a structure with at least one field whose type is a structure with tag tg_2. Using this relation, we build the set of all paths of nested tags:

```
Module PathNested.
```

```
Record t g n : Type := mkt {
  p :> {:n+1.-tuple tag} ;
  Hp1 : thead p ∈ Γ.dom g ;
  Hp : path (nested g) (thead p) (behead p) }.
End PathNested.
```

(The notation `{:n+1.-tuple tag}` is for the type of lists of `tag`s of size `n+1`. The head (resp. tail) of such a non-empty list can be taken using the function `thead` (resp. `behead`). A `path` is a non-empty sequence that obeys a progression relation. We borrow these datatypes from the library of SSReflect.)

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 `uniq` predicate from the SSReflect library):

```
Definition no_cycle g := ∀ n (p : PathNested.t g n), uniq p.
```

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

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

2.3 Deciding Well-formedness of Type Contexts

The well-formedness property in the previous section is a proposition of type `Prop`. We also provide a computational version to make Coq automatically enforce well-formedness of type contexts.

Providing a computational version is a bit involved for **(3) No cycle**, essentially because of the universal quantification over the paths' size in the predicate `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 : Γ.t) n : seq {:n+1.-tuple tag} := ...
Definition all_nestedpaths_in g n (s : seq {:n+1.-tuple tag}) :=
  ∀ (p : PathNested.t g n), PathNested.p _ _ p ∈ s.
Lemma compute_paths_complete g n :
  all_nestedpaths_in g n (compute_paths g n).
```

We can now build a boolean predicate `no_cycleb` that checks that there is no path as long as the type context, which implies the `uniq`ness of all the paths and thus the absence of cycles:

```
Definition no_cycleb g := compute_paths g (size (Γ.dom g)) == nil.
Lemma no_cycleb_sound g : no_cycleb g → no_cycle g.
```

(Note that `no_cycleb g` should actually be read as `no_cycleb g = true`. In fact, SSReflect systematically injects booleans into propositions using a coercion.) Similarly, we provide computational versions of **(1)** and **(2)** as the boolean predicates `completeb` and `no_emptyb`, and arrive at a provably sound boolean predicate to check well-formedness of type contexts:

```
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 the above soundness lemma, we even provide a notation that automatically enforces well-formedness:

```
Record wfctxt := mkWfctxt {
  wfctxtg :> Γ.t ;
  Hwfctxtg : wf_ctxt wfctxtg }.
Notation "\wfctxt{ g }" := (mkWfctxt g (wf_ctxtb_sound g erefl)).
```

2.4 Formalization of C Types using Well-formed Type Contexts

We formalized C types as dependent pairs of a type and a proof that the type is covered by a well-formed type context (i.e., the type is a "complete" type in C parlance):

```
Module Ctyp.
Record t (g : wfctxt) : predArgType := mk {
  ty :> typ ;
  Hty : cover g ty }.
End Ctyp.
Notation "g '.-typ:' ty " := (Ctyp.mk g ty erefl).
```

We also note "`ityp: t`" the construction of an integral arithmetic C type and "`:*t`" the construction of a pointer C type (provided the type context can be automatically inferred).

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;};}
```

To model these C types, we first define two structure tags:

```
Definition cell_tg := mkTag "cell".
Definition header_tg := mkTag "header".
```

For each of these structure tags, we define the associated lists of fields together with their types:

```
Definition cell_flds :=
  ("data", ityp uchar) :: ("head", ptyp (styp header_tg)) :: nil.
Definition header_flds := ("first", ptyp (styp cell_tg)) :: nil.
```

These definitions provide us with enough information to define a type context `g` that consists of the `cell` and `header` structures and to automatically check that it is well-formed:

```
Definition g :=
  \wfctxt{ "cell" ▹ cell_flds, "header" ▹ header_flds, ∅ }.
```

(The notation ▹ is for `Γ.add`, that adds a pair tag/fields to a type context, and ∅ is for `Γ.emp`, the empty type context.)

Eventually, we can define out of the type context `g` the model of the `cell` and `header` types (using the notation for the constructor `Ctyp.mk` introduced just above):

```
Definition cell := g.-typ: styp cell_tg.
Definition header := g.-typ: styp header_tg.
```

We will pursue this example in Section 3.2 by providing sizeof calculations.

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. For this purpose, we program a generic traversal function for C types. This function will also later be used as a part of a pretty-printer for C programs (Section 4.4).

3.1 A Generic Traversal Function for C Types

Our goal is to produce a function `typ_traversal` that traverses objects of type `g.-typ` to compute some result, such as alignment or sizeof. For 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 since it is not structural because of the use of a context, termination must be formally established.

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

```
Variable g : wfctxt.
Record config {Res Accu : Type} := mkConfig {
  f_ityp : integral → Res ;
  f_ptyp : typ → Res ;
  f_styp_iter : Accu → string * g.-typ * Res → Accu ;
  f_styp_fin : tag * g.-typ → (Accu → Accu) → Res }.
```

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

Although Coq natively accepts only structurally-recursive functions, its standard library provides well-founded recursion from a well-founded ordering by the `Fix` construct from the module `Coq.Init.Wf`.

Hereafter, let us assume that `Res` is the return type of the traversal function `typ_traversal`, that `Accu` is the auxiliary type used by the fold on structures, and that `c` is a computation configuration:

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

We now produce a function `styp_frec0` such that recursive traversal for structures will be obtained by using `Fix`.

```
Record Trace : Type := mkTrace {
  trace_size : nat ;
  trace :> PathNested.t g trace_size }.
```

```
Definition remains t := size (Γ.dom g) - trace_size t.
Program Definition styp_frec0 (t : Trace)
  (f : ∀ t', remains t' < remains t → Res) : Res := ...
```

The function `styp_frec0` has its argument packed in a dependent record of type `Trace`: the field `trace` is the path (of size `trace_size`) representing the previous nested calls of the function (with the next structure `tag` to proceed as the last element of the path). As observed in Section 2.2, such a path is bounded by the size of the type context. By defining `styp_frec0`'s argument measure as the difference between the current path and its bound (definition `remains` above), we can prove that any recursive call will terminate. This gives us the recursive function `styp_frec` for structure traversal, that we use to define the traversal function `typ_traversal` for any `typ` traversal:

```
Definition styp_frec :=
  Fix well_founded_remains (fun _ ⇒ Res) styp_frec0.
Program Definition typ_traversal (ty : g.-typ) : Res :=
  match Ctyp.ty _ ty with
    | ityp i ⇒ c.(f_ityp) i
    | ptyp p ⇒ c.(f_ptyp) p
    | styp tg ⇒ styp_frec (mkTrace 0 _)
end.
```

3.2 Standard-compliant Calculations of Alignment and Size for C Types

Here follows the instantiation of `typ_traversal` to compute alignment:

```
0 Definition align_ptr := 4.
1 Definition align_integral t :=
2   match t with uint ⇒ 4 | sint ⇒ 4 | uchar ⇒ 1 | ulong ⇒ 8 end.
3 Definition align_config g := mkConfig g
4   align_integral
5   (fun _ ⇒ align_ptr)
6   (fun a x ⇒ maxn x.2 a)
7   (fun _ x ⇒ x 1).
8 Definition align {g} := typ_traversal g (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" [IJ02, p. 158] is encoded by taking the maximal alignment of the structure's fields types (line 6 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 `typ_traversal` to compute the size of datatypes:

```
0 Definition sizeof_ptr : nat := 4.
1 Definition ptr_len := sizeof_ptr * 8.
```

```
2 Definition sizeof_integral t :=
3   match t with uint ⇒ 4 | sint ⇒ 4 | uchar ⇒ 1 | ulong ⇒ 8 end.
4 Definition sizeof_config g := mkConfig g
5   sizeof_integral
6   (fun _ ⇒ sizeof_ptr)
7   (fun a x ⇒ a + padd a (align x.1.2) + x.2)
8   (fun ty a ⇒ a 0 + padd (a 0) (align ty.2)).
9 Definition sizeof {g} := typ_traversal 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" [IJ02, 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, for example the fact that the size is never zero or that alignment always divides the size:

```
Lemma sizeof_gt0 (t : g.-typ) : 0 < sizeof t.
Lemma align_sizeof (t : g.-typ) : align t | sizeof t.
```

Lemmas about alignment and size are used in particular to prove the properties of pointer arithmetic (whose evaluation is defined using `sizeof`—see Section 4.2.3) and to reason about access to the fields of structures (field access, as defined in Section 4.2.1, returns addresses computed using `sizeof`, see [AMS14] for details). The formal verification of our case study (Section 7) requires reasoning using such properties because the target source code features accesses to fields of structures and to arrays via pointer arithmetic.

Examples of Alignment and Sizeof Calculations. Let us illustrate with the example of Section 2.4 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.
```

The efficiency of the computation of the size of datatypes seems reasonable in the context of interactive formal verification. Above computations (`sizeof cell`, `sizeof header`) take about 0.1 second. In our case study (Section 7), the computation of the size of the main data structure (13 fields including two nested structure with 5 fields each—see Section 7.1) takes 2 seconds.

4. A DEPENDENTLY-TYPED ENCODING OF THE CORE SUBSET C

The salient feature of our encoding of the core subset of C is the use of a dependently-typed syntax to ensure that only well-typed C programs can be modeled. This is made possible by the encoding of C types that we introduced in Sections 2 and 3. This encoding strategy comes in contrast to the standard approach of developing Separation logics for typed programming languages (e.g., [AB07]) in which values usually are a tagged union of integers, pointers, etc. In our setting, values are lists of bytes whose size corresponds to the `sizeof` of some C type (Section 4.1) and

expressions are dependently-typed with C types (Section 4.2). This dependently-typed syntax rejects C programs that are ill-typed, and thus expression evaluation need not fail because of type mismatch. We take advantage of this fact by modeling expression evaluation as a total function, without resorting to option types that usually model typing errors (see Section 4.2.3). This dependently-typed encoding of expressions naturally extends to C commands. We demonstrate this in Section 4.3 by extending previous work on formal verification of an assembly language [ANY12]. Last, we equip our formalization with pretty-printing functions developed in Coq so that verified programs can be compiled with a standard C compiler (Section 4.4).

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. We therefore define *physical values* as any list of bytes (machine integers of size 8 bits, type `int 8`) whose size corresponds to a valid C 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 `t.-phy` the Coq type of physical values of C type `t`.

It is nevertheless necessary to switch between physical values and list of bytes (e.g., to perform memory updates, see Section 4.3) or machine integers (to perform arithmetic operations, casts, etc.). For that purpose, we provide functions for conversions between lists of bytes and machine integers (e.g., `ptr◃i8` turns a list of bytes to a machine integer of pointer size, `i32◃i8` turns a list of bytes to a machine integer of size 32, etc.) as well as functions for conversions between machine integers and mathematical integers ($\mathbb{Z}$`◃u` interprets a machine integer as an unsigned (positive) integer, $\mathbb{Z}$`◃s` interprets a machine integer as a signed integer, $\mathbb{N}$`◃u` converts a machine integer to a natural number, etc.).

The following notations allow for switching directly between physical values and machine integers. We denote by `[ i ]p` the physical value corresponding to a machine integer `i` that represents an integral type (C type automatically inferred by Coq). For pointers, `phy◂ptr i` is the physical value corresponding to a machine integer `i` of pointer size (`int ptr_len`) and `ptr◂phy` converts a physical value back to a pointer.

We will use physical values in particular to define the evaluation of C expressions and the execution of C commands (Sections 4.2.3 and 4.3).

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 typing rules into the definition of the syntax, so that terms are well-typed by construction.

Before giving the complete formal definition of C expressions, let us explain the idea of this "intrinsic" encoding [BHKM12] using an example. In our formal

development, the constant `1` seen as a signed integer (the default type in C) is an expression of type `exp (ityp: sint)`. Let us assume that we are given a variable `buf`, noted `%"buf"` when considered as an expression, of type `exp (:*(ityp: uchar))`. First, we define addition, multiplication, etc. for arithmetic types using the following constructor:

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

For example, the addition is "`bop_n add_e`" (noted $+$), the multiplication "`bop_n mul_e`" (noted $*$), etc. Then, we define pointer arithmetic using the following constructor:

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

(The notation $+$ is overloaded to represent both addition of integral types and pointer arithmetic, see below.)

Then, `%"buf" * [ 1 ]sc` is forbidden by typing, but `%"buf" + [ 1 ]sc` is allowed, and moreover deemed to have type `exp (:*(ityp: uchar))`, as desired.

Here follows a more exhaustive definition of C arithmetic and pointer expressions (to be completed in Sections 4.2.1 and 4.2.2 with structure field access and casts[3]) (`g` is a type context and σ is an environment that associates variables to types):

```
Variables (g : wfctxt) (σ : g.-env).
Inductive exp : g.-typ → Type :=
| var_e : ∀ str t, env_get str σ = ⌊ t ⌋ → exp t
| cst_e : ∀ t, t.-phy → exp t
| bop_n : ∀ t, binop_ne (* numerical operators *) →
  exp (ityp: t) → exp (ityp: t) → exp (ityp: t)
| bop_r : ∀ t, binop_re (* relational operators *) →
  exp (ityp: t) → exp (ityp: t) → exp (g.-ityp: uint)
| add_p : ∀ t, exp (:* t) → exp (ityp: sint) → exp (:* t)
| eq_p : ∀ t, exp (:* t) → exp (:* t) → exp (g.-ityp: uint)
| ifte_e : ∀ t, exp (ityp: uint) → exp t → exp t → exp t.
...
```

The constructor `var_e` is the injection from strings to expressions for variables; the type of variables is fixed by the environment σ; we note `%str` for a variable with string identifier `str`. `cst_e` is for constants. We note `[ pv ]c` for the constant built from the physical value `pv`. We can build constants directly from a mathematical integer `z` (type `Z` in Coq): `[ z ]sc` for a signed integer (in two's complement notation), `[ z ]uc` for an unsigned character, `[ z ]8sc` for a signed character, etc. (the bit-level representation of `z` is silently truncated if `z` is out of bounds). `bop_n` is for numerical operations over integral types: addition (noted $+$), subtraction ($-$), multiplication ($*$), bitwise-and (&), bitwise-or ($|$), left-shift ($\ll$). `bop_r` is for relational operators for arithmetic types: testing for equality (noted $=$), inequality ($\neq$), less-than ($<$), less-than-equal ($\leq$), greater-than ($>$), greater-than-equal ($\geq$), logical or ($\|\|$), logical and (&&). Relational operators return an (unsigned) integer (0 or 1). `add_p` is for pointer arithmetic (also noted $+$ using overloading, see below). `eq_p` is for

[3]We do not model pointer subtraction and floating-point arithmetic, essentially because it is not needed by our case study (Section 7).

testing pointer equality (also noted = using overloading). `ifte_e` is for conditional expressions (notation: `e ? f : g`).

Overloading of Notations using Canonical Structures. The C language overloads several notations: `+` is used to add integral types but also to increment pointers, `==` is used to compare integral types but also pointers, etc. This overloading can be disambiguated by looking at the type of operands. Since our Coq model of C expressions is typed, we can also provide overloading: this helps limiting the number of Coq notations we introduce for the various syntactic constructs of C we model.

We illustrate overloading of notations using the example of addition. To represent both arithmetic and pointer addition as a single construct, we use Coq's canonical structures. We first provide a dependent record with a projection that is suitable to represent the several additions one wants to overload. The notation `+` is actually a shortcut for this projection.

```
Structure Cadd g (σ : g.-env) :=
  { Cadd_t1 : g.-typ ;
    Cadd_t2 : g.-typ ;
    Cadd_add : exp σ Cadd_t1 → exp σ Cadd_t2 → exp σ Cadd_t1 }.
Definition Cadd_add_nosimpl g σ := nosimpl (Cadd_add g σ).
Notation "a '+' b" := (Cadd_add_nosimpl _ _ _ a b).
```

For the Coq type inference to be able to decide between the several instantiations of the above parametric addition, we declare one canonical instance for arithmetic addition, and another for pointer addition:

```
Canonical Structure Cadd_i g σ t :=
  Build_Cadd g σ (ityp: t) (ityp: t) (bop_n σ t add_e).
Canonical Structure Cadd_p g σ t :=
  Build_Cadd g σ (:* t) (ityp: sint) (add_p σ t).
```

This way, Coq can find out, when typing, which of the canonical structures actually fits the notation `+`. The same idea can be applied to other syntactic constructs such as equality.

We now complete the presentation of the syntax of C expressions of Section 4.2 with structure field access and type conversions.

4.2.1 *Structure Field Access.* The constructor `fldp` of `exp` is for field access:

```
0 | fldp : ∀ f tg (t : g.-typ) (e : exp (:* t)) (H : styp tg = t) t',
1   assoc_get f (get_fields g tg) = ⌊ t' ⌋ →
2   exp (:* t')
```

We use the type of the dereferenced pointer (structure of type `t`, with tag `tg`) to check whether the field `f` is indeed valid: at line 1, `get_fields g tg` returns the list of fields of the C structure tagged `tg` so that one can check whether the accessed field `f` has the right type `t'`. This equality can in fact be automatically checked by Coq when involved expressions are ground. Checking such equalities is the purpose of the `erefl` proofs hidden in the notation for field access:

```
Notation "e '&→' f" := (@fldp _ _ f _ _ e erefl _ erefl).
```

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. This is a customary simplification in mechanizations of Separation logic [Jen13, §3.4.4].

4.2.2 *Type Conversions.* In C, type conversions between arithmetic 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. We provide three boolean tests to decide safety. `UnConv.safe t t'` holds when conversion from `t` to `t'` is safe (for example, when a small unsigned integer is converted to a larger unsigned integer). Safe type conversions are modeled by the constructor `safe_cast` below. The constructor `unsafe_cast` is for casts that may result in data loss or misinterpretation. The boolean test `UnConv.data_loss t t'` checks whether data can be lost when converting from `t` to `t'` (for example, when a large unsigned integer is converted to a smaller integer). `UnConv.misinterpret` checks whether data can be misinterpreted as a result of conversion (for example, when unsigned integers are converted to the corresponding signed integers—the higher-order bit becomes the sign bit):

```
| safe_cast : ∀ t t', exp (ityp: t) →
  UnConv.safe t t' → exp (ityp: t')
| unsafe_cast : ∀ t t', exp (ityp: t) →
  UnConv.data_loss t t' || UnConv.misinterpret  t t' →
  exp (ityp: t')
```

See the Coq formalization [AMS14] or standard literature (e.g., [Sea06, p. 162–163]) for the precise definitions of safe and unsafe casts.

We have chosen to make visible implicit type conversions. We write casts using notations that hide the boolean tests, the latter being automatically checked by Coq since they are part of our model of the C syntax (similarly to what we do for field accesses—see Section 4.2.1). We write unsafe casts with uppercases, e.g., "`(UINT) e`". In particular, "`(int) e`" is for safe casts to (signed) integers. 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, their evaluation need not fail because of type mismatch. We take this opportunity to model evaluation as a total function, thus avoiding `option` types and proofs that expressions are well-typed that might clutter formal proofs. Evaluation of an expression of type `t` always succeeds by returning a physical value of type `t.-phy`. Regarding undefined evaluation results, we rely on the underlying formalization of machine integers. Concretely, well-typed arithmetic expressions are evaluated according to the semantics of the type `int` of machine integers. Operations with this type are formalized as a module interface with a closed implementation that does not reveal the meaning of offending operations. For example, the interface reveals the overflow result of unsigned addition but not the one of signed addition, which is undefined according to the C standard. As a consequence, nothing can

be formally proved about the evaluation of well-typed arithmetic expressions that ought to result in an undefined behavior.

Evaluation of an expression is performed w.r.t. a `store`, which in essence extends the type environment by adding to each variable a physical value compatible with its type. We note `[ e ]_s` the evaluation of `e` w.r.t. store `s`. The complete definition is a bit long, so for the sake of clarity we just content ourselves with the example of pointer arithmetic (the complete definition can be found online [AMS14]). Suppose that e_1 + 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; in other words, e_1 + e_2 is actually `add_p t` e_1 e_2. First, we evaluate e_1 and convert the result to a machine integer of pointer size (conversion `ptr◃i8`). Second, we evaluate e_2 and convert the result to a machine integer of 32 bits (conversion `i32◃i8`). The latter is further interpreted as a signed integer (conversion $\mathbb{Z}$`◃s`). Finally, the pointer is added the product of the size of the type `t` and `k` and converted back to a physical value (conversion `phy◂ptr`):

```
Fixpoint eval {g σ t} (s : store σ) (e : exp σ 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
      phy◂ptr t (add_prod p (sizeof t) (ℤ◃s k))
  end ...
```

4.2.4 *Boolean Expressions.* Boolean expressions are formalized using arithmetic expressions by the following inductive type:

```
Inductive bexp {g} (σ : g.-env) :=
| exp2bexp of exp σ (g.-ityp: uint)
| bneg of bexp σ.
```

The intent is that an arithmetic expression is interpreted as "true" when its evaluation is not 0. `exp2bexp e` (notation: `\b e`) injects an arithmetic expression into boolean expressions and `bneg b` (notation: ¬ `b`) is the logical negation of `b`.

4.3 Syntax and Semantics of The 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). It is defined in such a way that accesses to uninitialized memory are treated as failures. This is important for our case study (Section 7) because such dangerous memory accesses are the main source of bugs one can expect from a C implementation of network packet parsing (see Section 7.5 for concrete examples). We do not provide a malloc command because it was not required by our case study. Yet, we see no reason why its addition to our intrinsic encoding could cause any problem (except the anticipated technicalities of the related Separation logic rules). Also, we do not provide goto commands and function calls because it is reasonable to encode them in our case study: there is only one goto command that can be encoded using an additional local variable and structured control-flow, and function calls can be

```
Inductive exec_cmd₀ : option state → cmd₀ → option state → Prop :=
| exec_skip : ∀ s, ⌊ s ⌋ ≻ skip ⇝ ⌊ s ⌋
| exec_assign : ∀ s h t str Hstr e,
  ⌊ s, h ⌋ ≻ @assign t str Hstr e ⇝ ⌊ store_upd Hstr [ e ]_s s, h ⌋
| exec_lookup : ∀ s h t str Hstr e v,
  let a := ℕ◃u (ptr◂phy [ e ]_s) in
  heap_get t a h = ⌊ v ⌋ →
  ⌊ s, h ⌋ ≻ @lookup _ str Hstr e ⇝ ⌊ store_upd Hstr v s, h ⌋
| exec_lookup_err : ∀ s h t str Hstr e,
  let a := ℕ◃u (ptr◂phy [ e ]_s) in
  heap_get t a h = ⊥ →
  ⌊ s, h ⌋ ≻ @lookup t str Hstr e ⇝ ⊥
| exec_mutation : ∀ s h t e₁ e₂ v,
  let a := ℕ◃u (ptr◂phy [ e₁ ]_s) in
  heap_get t a h = ⌊ v ⌋ →
  ⌊ s, h ⌋ ≻ @mutation t e₁ e₂ ⇝ ⌊ s, heap_upd t a [ e₂ ]_s h ⌋
| exec_mutation_err : ∀ s h t e₁ e₂,
  let a := ℕ◃u (ptr◂phy [ e₁ ]_s) in
  heap_get t a h = ⊥ →
  ⌊ s, h ⌋ ≻ @mutation t e₁ e₂ ⇝ ⊥
where "s ≻ c ⇝ t" := (exec_cmd₀ s c t).
```

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

inlined (provided some care about scoping rules) since they are not recursive. We nevertheless plan to address as future work the addition of function calls to our intrinsic encoding.

We derive the syntax and the semantics (as well as various lemmas) of the core subset of C by instantiating a parameterized module adapted from previous work [ANY12]. 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 at line 3 requires that the expression to be dereferenced is of pointer type `:*t` and that the destination variable is of the type `t`:

```
0 Inductive cmd₀ : Type :=
1 | skip : cmd₀
2 | assign : ∀ t str, env_get str σ = ⌊ t ⌋ → exp σ t → cmd₀
3 | lookup : ∀ t str, env_get str σ = ⌊ t ⌋ → exp σ (:* t) → cmd₀
4 | mutation : ∀ t, exp σ (:* t) → exp σ t → cmd₀.
```

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

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

The operational semantics for the basic commands is given in Figure 1. It is a relation between an optional *state* (a pair of a store `s` and a heap `h`) ($\bot$ represents an execution error). The effect of an assignment (constructor `exec_assign`) is to update the store. It never fails. In particular, it cannot fail as a result of a type mismatch because the constructor `assign` has been defined so that the types of the assigned variable and of the expression agree. Also, assignment cannot fail as a result of a memory access since expressions have no side-effect. 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 (function `heap_get`); eventually, the value obtained is saved in the store (function `store_upd`). Since lookup accesses the memory, it fails when the accessed address is not initialized (constructor `exec_lookup_err`). The constructors for mutation should now be self-explanatory.

4.4 Example of Formalization of a C Program: In-place Reverse-list

In this section, we model the mandatory example of Separation logic: in-place reverse-list. We also take this opportunity to introduce pretty-printing functions to display Coq models of C programs in their usual concrete syntax.

First, we model the following C type of singly-linked lists:

```
struct Clst {
  unsigned int data;
  struct Clst (*next); };
```

This is achieved in the same way as we modeled the self-referential C structures in Section 2.4. We define a structure tag, then the list of typed fields, that we wrap up in a context, to be used as the parameter of models of C types:

```
Definition Clst_tg := mkTag "Clst".
Definition Clst_flds :=
  ("data", ityp uint) :: ("next", ptyp (styp Clst_tg)) :: nil.
Definition g := \wfctxt{ "Clst" ▹ Clst_flds , ∅ }.
Definition Clst := g.-typ: (styp Clst_tg).
```

One can check that `Clst` indeed models the intended type declaration by pretty-printing:

```
Eval compute in (typ_to_string_rec g Clst "" "")%string.
  = "struct Clst { unsigned int data; struct Clst (*next); } "
  : string
```

Now that the type of singly-linked lists is modeled, we set up an environment of typed variables for the in-place reverse-list program. More precisely, we declare three variables `"i"`, `"ret"`, and `"rem"` that are pointers to singly-linked lists:

```
Definition σ : g.-env :=
  ("i", :* Clst) :: ("ret", :* Clst) :: ("rem", :* Clst) :: nil.
```

One can again check by pretty-printing that the modeled environment is the right one:

```
Eval compute in (foldl (fun s p ⇒ s ++ typ_to_string
  (C_types.Ctyp.ty _ (snd p)) (fst p) (line ";")) "" σ).
```

```
= "struct Clst (*i);
   struct Clst (*ret);
   struct Clst (*rem);" : string
```

Given this environment σ, we instantiate a parameterized module that provides us with a syntax and a semantics (and also a Separation logic, see Section 5) to write programs using this environment. Eventually, we can use the syntax introduced in the previous sections to write the in-place reverse-list program:

```
Definition reverse_list :=
  "ret" := NULL ;
  While (¬ (\b %"i" = NULL)) {
    "rem" :=* (%"i" &→ "next") ;
    (%"i" &→ "next") *:= %"ret" ;
    "ret" := %"i" ;
    "i" := %"rem" }.
```

Coq notations arguably provide us with a convenient syntax. The dependently-typed encoding does not clutter Coq's output with uninformative proofs (checking the validity of field accesses, the fact that the type of variables and the type of dereferenced/assigned expressions agree) because they are hidden in Coq notations. As a matter of fact, we have successfully used this syntax to carry out the case study of Section 7. We can again check the adequacy of the Coq model by pretty-printing to C's concrete syntax. The script

```
Goal PrintAxiom _ (pp_cmd 0 reverse_list "").
  compute -[pp_Z append Z.add Z.sub Z.mul].
  rewrite !Z2uK //=.
```

outputs the proof goal

```
  ============================
   PrintAxiom string "ret = NULL;
                      while (!(((i) == (NULL)))) {
                          rem = *(i)→next;
                          *(i)→next = ret;
                          ret = i;
                          i = rem; }"
```

from which the C program can be for example copy-pasted for compilation.

We pursue the in-place reverse-list example in Section 5.3.2 by providing the specification of singly-linked lists in Separation logic.

5. SEPARATION LOGIC FOR THE CORE SUBSET OF C

In this section, we introduce a Separation logic for the core subset of C that we formalized in the previous section. We start by providing a Hoare logic (Section 5.1). Then, we propose a formalization of the mapsto connective of Separation logic and illustrate how we formalize the accompanying Separation logic rules (Section 5.2). This mapsto connective uses physical values and, therefore, when it deals with structures, it exposes their padding. To hide padding, we introduce a notion of "logical view" of C values (duly illustrated with the example of singly-linked lists) for which we also provide Separation logic rules connecting with the physical values

(Section 5.3). We conclude this section with an overview of the formalization of the core subset of C (Section 5.4).

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 parameterized modules from previous work [ANY12]. To instantiate these modules, we essentially need to provide the Hoare triples for the basic commands (`assign`, `lookup`, `mutation`), a proof that they are sound w.r.t. the operational semantics given in Section 4.3 (so that the module can provide a soundness proof for the whole Hoare logic with structured control-flow), and a proof that the weakest liberal precondition is a valid precondition for basic commands (so that the module can provide a (relative) completeness proof for the whole Hoare logic). See the Coq formalization [AMS14] for details about soundness and completeness. Let us briefly comment on the Hoare triples for the basic commands (inductive relation `hoare0`, Figure 2). For example, the assignment rule

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

is encoded by the constructor `hoare0_assign`. Its precondition is expressed using the predicate transformer `wp_assign`. Assertions are shallow-encoded: they have type `assert` defined as `store → heap → Prop`. Because of the shallow encoding, substitution is encoded by updating the store using `store_upd`. Other Hoare rules should be self-explanatory since they follow the operational semantics explained in Section 4.3.

5.2 The Mapsto Connective and Separation-logic Triples

5.2.1 *The Mapsto Connective of Separation Logic for C.* Regarding the encoding of Separation logic, what is new is not the separating conjunction $\star$ or the separating implication $\mathrel{-\!\!\star}$ (their encoding is as usual, see for example [ANY12]), but rather the primitive mapsto connective. The mapsto connective is usually 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. For C, it would not be practical to make every memory byte a cell. A cell ought rather be a C (physical) value and the address on the left-hand side of the mapsto be the starting address of this physical value. This amounts to provide a version of mapsto parameterized with the accessed type (like the "chunks" in Appel and Blazy's Separation logic [AB07]).

The physical values of Section 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 for example to abide by alignment rules and to guarantee the absence of the null pointer in allocated areas. Before defining the mapsto connective for C, we first provide a relation `phy_mapsto` (between a physical value `v`, an address `a`, and a heap `h`) that specifies what it means for a physical value to be correctly stored in memory. The following inductive predicate specifies that the heap `h` contains exactly the bytes of the physical value `v` (line 3), that it maps the sequence of addresses `a`, `a + 1`, ..., `a + sizeof t - 1` (written `iota a (sizeof t)` in SSReflect) where `t` is the type of the physical value `v` (line 4), that this area

```
Inductive wp_assign {str t} Hstr e P : assert :=
| wp_assign_c : ∀ s h, P (@store_upd _ _ str t Hstr [ e ]_s s) h →
  wp_assign Hstr e P s h.

Inductive wp_lookup {str t} Hstr (e : exp σ (:* t)) P : assert :=
| wp_lookup_c : ∀ s h pv,
  heap_get t (ℕ◃u (ptr◂phy [ e ]_ s)) h = ⌊ pv ⌋ →
  P (@store_upd _ _ str t Hstr pv s) h →
  wp_lookup Hstr e P s h.

Inductive wp_mutation {t} (e₁ : exp σ (:* t)) e₂ P : assert :=
| wp_mutation_c : ∀ s h v, let a := ℕ◃u (ptr◂phy [ e₁ ]_s) in
  heap_get t a h = ⌊ v ⌋ →
  P s (heap_upd t a [ e₂ ]_s h) →
  wp_mutation e₁ e₂ P s h.

Inductive hoare0 : assert → cmd₀ → assert → Prop :=
| hoare0_skip : ∀ P, hoare0 P skip P
| hoare0_assign : ∀ P t str Hstr e,
  hoare0 (wp_assign Hstr e P) (@assign t str Hstr e) P
| hoare0_lookup : ∀ P t str Hstr e,
  hoare0 (wp_lookup Hstr e P) (@lookup t str Hstr e) P
| hoare0_mutation : ∀ P t e₁ e₂,
  hoare0 (wp_mutation e₁ e₂ P) (@mutation t e₁ e₂) P.
```

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

does not overrun the memory (as guaranteed by C's malloc) (line 5), and that the physical value v is aligned at the address a (line 6):

```
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 →
4   hp.dom h = iota a (sizeof t) →
5   ℤ◃ℕ a + ℤ◃ℕ (sizeof t) < 2^ptr_len →
6   align t | a →
7   phy_mapsto v a h.
```

($\mathbb{Z} \triangleleft \mathbb{N}$ injects a natural number into relative integers.) `phy_mapsto` gives us the basis to define (a raw form of) the mapsto connective of Separation logic, $\mathtt{a} \mapsto_{\mathrm{p}} \mathtt{v}$ (subscript "p" for "physical") that associates an address a to a (typed) physical value v:

```
Notation "a '↦ₚ' v" := (fun _ ⇒ phy_mapsto v (ℕ◃u (ptr◂phy a))).
```

5.2.2 *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 [Rey08, p.88]:

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

Recall that $P \mathrel{-\!\!*} Q$ holds when the heap can be extended with a disjoint part for which P holds so that Q holds for the extended heap.

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

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

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

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

5.3 Separation Logic with Logical C Values

5.3.1 *The Logical View of C Values.* In the previous section, we explained how to formalize Separation logic using a physical view of C values that exposes byte-level details such as padding. The advantage of this approach is that it makes for a simple and convincing semantics. Yet, when it comes to formal verification of portable C programs, the contents of padding are often irrelevant and it would be an inconvenience to have to deal with it in the specification or during the formal proof. 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. The logical view comes as an add-on to our Separation logic; it provides an alternate way to specify data structures and Separation logic reasoning rules, and its cost is just a side-condition (the correspondence between the physical and the logical value) that trivially holds for data structures with no padding.

Logical values are defined by the (mutually) inductive type `log`, indexed by a C type `g.-typ` (Figure 3). According to this definition, a logical value of an integral type is a machine integer of the appropriate size (constructors `log_of_uint`, `log_of_sint`, `log_of_uchar`, `log_of_ulong`). For any pointer type, it is a machine integer of size `ptr_len` (constructor `log_of_ptr`). For a structure, it is an association list of strings (for the fields' names) and logical values (of type `g.-env`). Hereafter, we note `t.-log` for the type of logical values of type `t`.

We write $pv \mathbin{©} lv$ when the physical value pv and the logical value lv correspond to the same list of bytes. In particular, $pv \mathbin{©} lv$ holds when pv and lv are respectively a physical and a logical value of integral types or pointers with the same underlying byte string, but, with its current definition, it may not be decidable when both values refer to structures.

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. This relation is defined by the (mutually) inductive predicate `log_mapsto` (Figure 4). Logical and physical values for integral types and pointers are the same (`phy_of_log` just performed a type cast). The difference between logical and physical values is that logical values leave undefined the contents of the interleaving padding, if any. This can be observed in the constructors for structure types (`log_of_styp_mapsto` and `cons_logs_mapsto`) where that

```
Inductive log : g.-typ → Type :=
| log_of_uint : int 32 → log (ityp: uint)
| log_of_sint : int 32 → log (ityp: sint)
| log_of_uchar : int 8 → log (ityp: uchar)
| log_of_ulong : int 64 → log (ityp: ulong)
| log_of_ptr : ∀ t', int ptr_len → log (:* t')
| log_of_styp : ∀ tg, logs (get_fields g tg) → log (styp tg)
with logs : g.-env → Type :=
| nil_logs : logs nil
| cons_logs : ∀ hd tl, log hd.2 → logs tl → logs (hd :: tl).
```

Fig. 3. Logical view of C values

part of the heap that corresponds to padding (`pad`) is universally quantified and only constrained by its size (`pad_sz`) (the notation ⊎ is for concatenation of disjoint heaps).

Like its physical counterpart `phy_mapsto`, the predicate `log_mapsto` gives rise to a "logical mapsto" connective for Separation logic:

```
Notation "a '↦ₗ' v" := (fun _ ⇒ log_mapsto v (ℕ◃u (ptr◂phy a))).
```

5.3.2 *Example: Singly-linked Lists.* We saw in Section 4.4 the formal model `Clst` of singly-linked lists, a data structure containing two fields: `"data"` of type `uint` and `"next"` of type "pointer to a `Clst` structure". Using this type, we now define a Separation-logic assertion that relates a Coq list of `int 32` (for the contents of the `"data"` fields) with a pointer of type `:*Clst` (that points to a singly-linked list):

```
0 Inductive sepClst : seq (int 32) → (:* Clst).-phy → assert :=
1 | lnil : ∀ s, sepClst nil pv0 s hp.emp
2 | lcons : ∀ hd tl a p, a ≠ pv0 →
3   ((a ↦ₗ mk_cell hd (ptr◂phy p)) ⋆ sepClst tl p) ⟹
4   sepClst (hd :: tl) a.
```

The constructor `lnil` relates an empty list `nil` with the NULL pointer `pv0`. The constructor `lcons` relates the list `hd::tl` with the pointer `a` (which is non-NULL, as specified at line 2), provided the latter points to a `Clst` containing `hd` as `"data"` and whose `"next"` pointer is related to `tl` (line 3). Above, `mk_cell` constructs a logical value of type `log Clst`, and ⟹ is notation for the entailment relation between `assert`s, defined as follows:

```
Definition entails (P Q : assert) : Prop := ∀ s h, P s h → Q s h.
```

The complete example of in-place reverse-list example can be found online [AMS14].

5.3.3 *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 (i.e., $pv \circledcirc lv$) for the substitution:

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

```
Inductive log_mapsto {g} : ∀ {t : g.-typ},
  t.-log → nat → hp.t → Prop :=
| log_of_uint_mapsto : ∀ (v : (g.-ityp: uint).-log) a h,
  phy_mapsto (phy◃log v) a h → log_mapsto v a h
| log_of_sint_mapsto : ∀ (v : (g.-ityp: sint).-log) a h,
  phy_mapsto (phy◃log v) a h → log_mapsto v a h
| log_of_uchar_mapsto : ∀ (v : (g.-ityp: uchar).-log) a h,
  phy_mapsto (phy◃log v) a h → log_mapsto v a h
| log_of_ulong_mapsto : ∀ (v : (g.-ityp: ulong).-log) a h,
  phy_mapsto (phy◃log v) a h → log_mapsto v a h
| log_of_ptr_mapsto : ∀ t (v : (:* t).-log) a h,
  phy_mapsto (phy◃log v) a h → log_mapsto v a h
| log_of_styp_mapsto : ∀ t tg H vs a h pad pad_sz,
  align t | a →
  ℤ◃ℕ (a + size (hp.dom h)) + ℤ◃ℕ pad_sz < 2 ^ ptr_len →
  logs_mapsto (get_fields g tg) vs a h →
  pad_sz = padd (a + size (hp.dom h)) (align t) →
  hp.dom pad = iota (a + size (hp.dom h)) pad_sz →
  log_mapsto (log_of_styp 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 →
  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).
```

Fig. 4. Logical mapsto connective

The definition of the relation © is currently such that the above rule does not make it possible to read/write complete padded structures. C programs that manipulate structures as first-class objects would require us to extend the definition of convertibility. Fortunately, the current definition is sufficient to already treat many programs, such as the non-trivial case study we present in Section 7.

Here follows the formalization of the Separation logic rule above:

```
Inductive wp_lookup_back_conv {t} x H (e : exp σ (:* t)) P : assert :=
| wp_lkbr1_conv : ∀ s h (pv : t.-phy) (lv : t.-log), pv © lv →
  ([ e ]_ s ↦₁ lv ⋆
    ([ e ]_ s ↦₁ 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 σ (:* t)) P :
  { wp_lookup_back_conv x H e P } lookup x e H { P }.
```

5.4 Overview of Our Separation Logic for the Core Subset of C

For reference and also to help navigation in the Coq scripts, Table I summarizes in terms of files and lines of code (as given by the standard coqwc command) our

file	contents	spec	proof
C_types.v	Sect. 2	284	821
C_types_fp.v	Sect. 3	352	743
C_value.v	Sect. 4.1, 5.3.1	671	1287
C_expr.v	Sect. 4.2	594	770
C_expr_equiv.v	Lemmas about expression rewriting	142	434
C_expr_ground.v	Lemmas about ground expressions	125	306
C_seplog.v	Sect. 4.3, 5 (except 5.3.1, 5.3.2)	558	1102
C_contrib.v	Lemmas about Separation logic [Aff14]	583	866
C_tactics.v	Ltac-based automation [Aff14]	1569	458
C_pp.v	Sect. 4.4	244	21
C_examples.v	Sect. 2.4, 3.2	57	2
C_swap.v	Swap of two memory cells	35	56
C_reverse_list_header.v	Sect. 4.4, 5.3.2	95	51
C_reverse_list_triple.v	Verification with lemmas only	13	152
C_reverse_list_tactics.v	Verification with Ltac tactics	30	200
	Total	5352	7269

Table I. Overview of the formalization of Separation logic for C (in terms of l.o.c.)

formalization of Separation Logic for the core subset of C. The contents of most files have actually been explained in the previous sections. In addition, we have also developed about 30 tactics to automate mundane reasoning steps about entailments between Separation logic formulas and about Hoare triples. We will comment a bit more about automation in Section 7.4 when reporting on the application of our Separation logic to the verification of a network packet parsing function (which is the purpose of Section 7).

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 [DR08] 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 Figures 5, 6 and 7). We use this formal RFC in Section 7.3 to specify a parsing function taken from an existing implementation of TLS. Since such a specification is mostly about the format of network packets, we believe that other parsing functions taken from different implementations can be specified using the same formal RFC.

6.1 Encoding the TLS Presentation Language

The presentation language [DR08, §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`. This explicit number of bytes is an original feature of the presentation language used in the RFC for TLS.

(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. Again, the explicit mention of the number of bytes used by data structures is peculiar among the various RFCs.

(4) `enum { e1(v1), . . . , en(vn) [[, (m)]] } T` defines the *enumerated* type `T`. The size of the payload must be sufficient to encode the largest value: one of the v_i's or `m` (`m` is optional, hence the notation `[[ . . . ]]`). This payload is preceded by a "length field", like the 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 below).

(6) *Variants* extend structures with fields whose type depends "on some knowledge that is available within the environment" [DR08, §4.6.1]. This "knowledge" can be the (implicit) environment (e.g., the "length field" of the enclosing Handshake packet in the case of `ClientHello` [DR08, §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` [DR08, §7.4]) (in which case we are in fact dealing with a dependent record).

Putting dependent records aside (see Section 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 its arity. We use dependent types to automatically check divisibility constraints on fixed-length vectors (line 3), the "length field" (variable `k` below) of variable-size vectors (line 6) and enumerateds (line 9):

```
0  Inductive tls_typ : Z → Z → Type :=
1  | opaque : tls_typ 1 1
2  | arr : ∀ n, tls_typ n n → ∀ m, 0 ≤ m →
3    m mod n == 0 →
4    tls_typ m m
5  | varr : ∀ n m (t : tls_typ n m) (k : nat) a b, a ≤ b →
6    k ≠ 0 → b < 2^(k * 8) → 2^((k - 1) * 8) ≤ b → m ≤ ℤ◃ℕ k + b →
7    tls_typ (ℤ◃ℕ k + a) (ℤ◃ℕ k + b)
8  | enum : ∀ k l n, uniq l →
9    Zmax(l, n) < 2^(k * 8) → 2^((k - 1) * 8) ≤ Zmax(l, n) →
10   tls_typ (ℤ◃ℕ k) (ℤ◃ℕ k)
11 | pair : ∀ {n1 m1 n2 m2},
12   string * tls_typ n1 m1 → tls_typ n2 m2 →
13   tls_typ (n1 + n2) (m1 + m2)
14 | typ_nil : tls_typ 0 0.
```

The arithmetic proofs that are necessary to craft a value of type `tls_typ` can be automatically inferred by Coq. We can therefore hide them using Coq notations that mimic those used in the RFC. The only difference between our notations and those of the RFC is that we make explicit the "length field" of datatypes.

```
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) }.
```

Fig. 5. Formalization of the `Extension` type (left: TLS RFC; right: Coq formalization)

For example, the second field of the structure type `Extension` appears as `opaque extension_data<0..2^16-1>` in the RFC; the fact that its "length field" is 2 is implicit. In the Coq formalization, we need to make it explicit, but still its correctness is ensured by type-checking:

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

See Figures 5 and 6 for examples and the Coq scripts [AMS14] for definitions.

Consistency of Definitions in the RFC. The type `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. Figure 5 shows on the left the `Extension` type [DR08, §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 Figure 5 displays the Coq counterpart using `tls_typ`s: the syntactic match is obvious (the main difference with the informal syntax is that we make the length field of variable-size vectors and enumerateds explicit, as explained above). The problem with the `Extension` type defined as such is that it is used to define the type of the `extensions` field of ClientHello packets using the following declaration ([DR08, §7.4.1.2], or line 13 in Figure 7):

```
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 Figure 5:

```
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 the RFC, it "must be an even multiple of the length of a single element" [DR08, §4.3] which is not possible in general when variable-size vectors are nested such as in `extensions`.

6.2 Dealing with Dependent Records in the Presentation Language

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

```
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;
```

```
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 extensions_type :=
  Extension < 0 .. (2 ^ 16 - 1) > 2.
```

Fig. 6. Formal specification for the types of the fields of ClientHello packets (left: TLS RFC; right: Coq) (see Figure 5 for the type `Extension` and Figure 7 for the ClientHello packet itself)

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

The function `decode'` takes as arguments a `tls_typ` (parameterized by `n` and `m`, the minimum and maximum size of the list of bytes it represents, see Section 6.1) and a list of bytes, and returns a boolean indicating whether the first bytes of the list conform to this `tls_typ`, together with the rest of the bytes. The parameter `k` is a bound for the recursion. The function `decode` instantiate this bound to the depth of the `tls_typ` being matched. Finally, the boolean predicate `decodep` decides whether a list of bytes conforms to a `tls_typ`. Second, 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 ([DR08, § 6.2.1], reproduced on the left of Figure 7). First, observe that the structure `ClientHellop_packet` is parameterized by `m` which is a list of bytes coming from an outer packet[4]. Each field of `ClientHellop_packet` is a `packet` that conforms to the appropriate `tls_typ`. For example, the field `cipher_suites` is for list of bytes that conforms to `cipher_suites_type` (defined in Figure 6) according to the boolean predicate `decodep`. There is a dependency between the field `extensions`

[4]ClientHello packets belong to the Handshake protocol (whose packets are themselves encapsulated into the Record protocol). This is illustrated by Figure 8.

```
0  struct {
1   ProtocolVersion client_version;
2
3   Random random;
4   SessionID session_id;
5   CipherSuite
6    cipher_suites<2..2^16-2>;
7   CompressionMethod
8    compression_methods<1..2^8-1>;
9   select (extensions_present) {
10   case false:
11    struct {};
12   case true:
13    Extension extensions<0..2^16-1>;
14  };
15 } ClientHello;
```

```
Structure ClientHello_packet
 {m} (H : decodep uint24 m) := {
 client_version : packet (fun x ⇒ decodep
  ProtocolVersion x && proverp x) ;
 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 ( dselectb(
  client_extensions_present (ℕ◃i8 m)
  (var_sz session_id) (var_sz cipher_suites)
  (var_sz compression_methods) \)  {
    (false, decodep struct{}) ;
    (true, decodep extensions_type) }) }.
```

Fig. 7. Formal specification of ClientHello packets (left: TLS RFC; right: Coq) (see Figure 6 for the definition of the types of the fields)

(see line 13 on the left) and its predecessors: the predicate `extensions_present` decides the presence of extensions depending on the sizes of `session_id`, `cipher_suites`, and `compression_methods` (and also the value of `m`). Yet, this relation is only expressed in prose in the RFC. In our Coq formalization, each field is represented by a `packet` of some predicate. Checking whether `extensions_present` is true can therefore be formally expressed by the following boolean predicate (see Figure 7 on the right where it is used for the meaning of parameters):

```
Definition client_extensions_present m sid cys cpm :=
  ClientHello_sz sid cys cpm < ℤ◃ℕ m.
```

Contrary to `tls_typ`s, we have no generic decoding function such as `decode` for TLS packets formalized as Coq dependent records but we can write case-by-case decoders in a systematic way using the generic decoding function for `tls_typ`s. For example, the following function provably decodes the sequence of the fields of ClientHello packets formalized as in Figure 7:

```
Definition ClientHello_decode m l : bool * seq byte :=
if ¬ decodep uint24 m then (false, l) else
let (a₁, l₁) := let (a₁', l₁) := decode ProtocolVersion l in
 (a₁' && proverp (take (ℕ◃ℤ (fixed_sz ProtocolVersion)) l), l₁) in
let (a₂, l₂) := decode Random l₁ in
let (a₃, l₃) := decode SessionID l₂ in
let (a₄, l₄) := decode cipher_suites_type l₃ in
let (a₅, l₅) := decode compression_methods_type l₄ in
if client_extensions_present (ℕ◃i8 m)
  (size l₂ - size l₃ - ℕ◃ℤ (fixed_sz SessionID))
  (size l₃ - size l₄ - ℕ◃ℤ (fixed_sz cipher_suites_type))
  (size l₄ - size l₅ - ℕ◃ℤ (fixed_sz compression_methods_type)) then
 let (a₆, l₆) := decode extensions_type l₅ in
 ([&& a₁, a₂, a₃, a₄, a₅ & a₆], l₆)
```

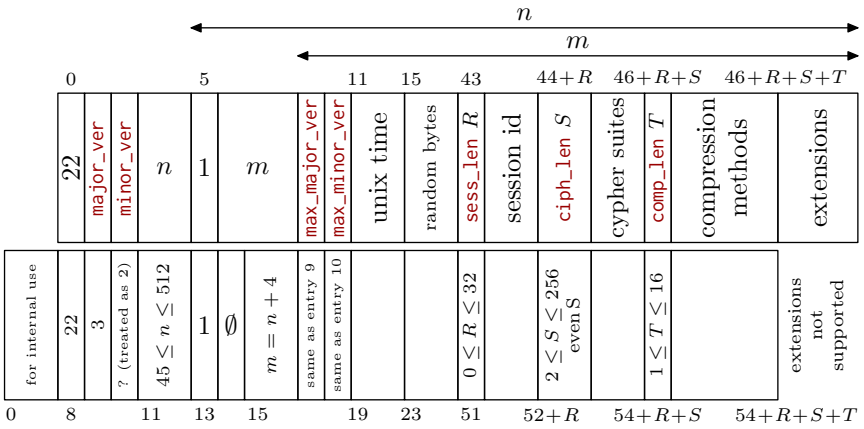

Fig. 8. Format of a Record packet containing a Handshake packet of size n containing a ClientHello packet of size m. RFC [DR08] definition on the top; internal representation used by PolarSSL [POL] at the bottom.

```
else
 ([&& a1, a2, a3, a4 & a5], l5).
```

($\mathbb{N} \triangleleft \mathbb{Z}$ denotes the absolute value.)

Using above ideas, we have formalized many of the packet formats of the TLS protocol. This gives us all the definitions we need to perform the formal verification of source code implementations of TLS, as we see in the next section.

7. VERIFICATION OF POLARSSL CLIENTHELLO PARSING

We experiment our framework with the verification of the source code of a parsing function from PolarSSL [POL], an implementation of the TLS protocol. As already explained in Section 1, this experiment is motivated by the fact that many security vulnerabilities arise from parsing functions that do not implement all the checks required by the RFC.

Precisely, We verify the function `ssl_parse_client_hello` (`library/ssl_srv.c`, version 0.14.0) that parses initialization ClientHello packets. Figure 8 displays a byte-level representation of the packets that `ssl_parse_client_hello` is intended to parse. They are Record protocol packets, containing a Handshake protocol packet starting at byte index 5 (as indicated by the magic number 22 at byte index 0), the latter supposed to be a ClientHello packet (as indicated by the magic number 1 at byte index 5). We also provide in Figure 8 the byte representation used internally by PolarSSL. In particular, PolarSSL is a bit more restrictive than the RFC; we indicate for example some of the restrictions it imposes on incoming ClientHello packets (see Sections 7.3 and 7.5 for a more precise discussion about these restrictions).

7.1 The Main Data Structure

The central data structure in a PolarSSL server is of C 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 Figure 8). Here follows the corresponding typ definition (Figure 9 provides a pictorial representation):

```
Definition ssl_sess :=
 ("cipher", ityp sint) ::
 ("length", ityp sint) ::
 ("id",     ptyp (ityp uchar)) :: nil.
Definition ssl_session := styp (mkTag "ssl_session").
Definition ssl_ctxt :=
 ("state",         ityp sint) ::
 ("major_ver",     ityp sint) ::
 ("minor_ver",     ityp sint) ::
 ("max_major_ver", ityp sint) ::
 ("max_minor_ver", ityp sint) ::
 ("session",       ptyp ssl_session) ::
 ("in_hdr",        ptyp (ityp uchar)) ::
 ("in_msg",        ptyp (ityp uchar)) ::
 ("in_left",       ityp sint) ::
 ("fin_md5",       md5_context) ::
 ("fin_sha₁",      sha₁_context) ::
 ("ciphers",       ptyp (ityp sint)) ::
 ("randbytes",     ptyp (ityp uchar)) :: nil.
Definition ssl_context := styp (mkTag "ssl_context").
```

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 Formalization of The ClientHello Parsing Function

Roughly, the original C function ssl_parse_client_hello is 161 lines long, about 85 lines if we remove comments (many of them are important because they explain the magic numbers coming from the RFC of TLS) and debugging information. Once converted to Coq, it is 132 lines long (this includes about 12 lines that had to be added afterwards to correct implementation errors in the original source code, see Section 7.5). The language of arithmetic expressions of our subset of C is sufficiently rich so that C expressions can be almost ported as they are. Yet, we need to adapt the original code to structured control-flow by replacing one goto with if-then-else's and by merging returns. Moreover, since the expression language is side-effect free, some C expressions need to be split into several commands using temporary variables. This makes the target program longer in terms of lines of code but reasoning on individual lines is actually simpler. The main drawback is maybe the program transformation in itself that needs to be trusted. 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 [WKS+09].

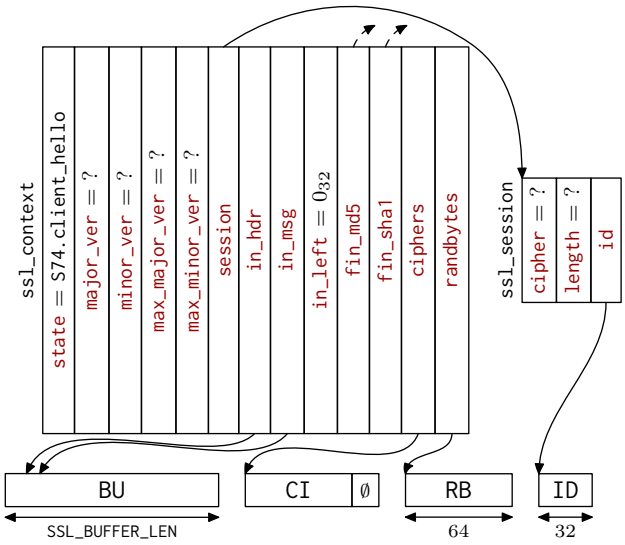

Fig. 9. Pictorial representation of PolarSSL's memory state before parsing a ClientHello packet (see Figure 10 for the state after)

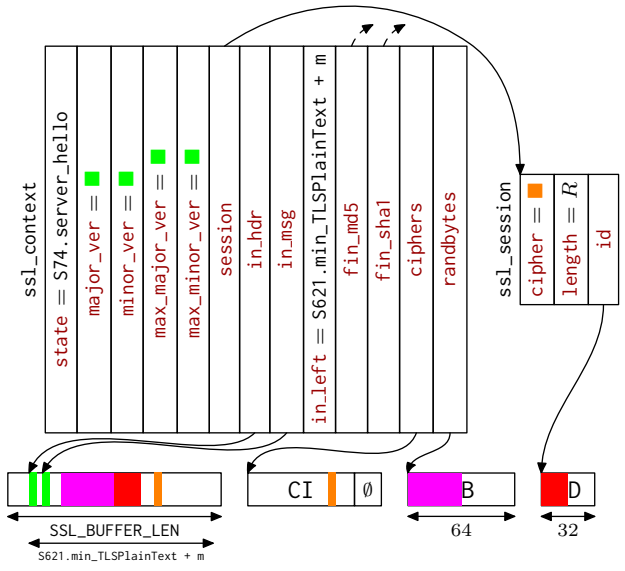

Fig. 10. Pictorial representation of PolarSSL's memory state after parsing a ClientHello packet (see Figure 9 for the state before)

The formal model of the PolarSSL function that parses ClientHello packets of TLS can be found online [AMS14]. Its variables are typed according to the following environment σ:

```
Definition σ : g.-env :=
 ("ret", ityp: sint) ::
 ("ssl", :* (g.-typ: ssl_context)) ::
 ("buf", :* (ityp: uchar)) ::
 ("buf0", ityp: uchar) ::
 ("buf1", ityp: uchar) ::
 ("buf2", ityp: uchar) :: ...
```

Equipped with the environment σ, we use the syntax defined in Section 4 to translate the original C source code into its Coq model, whose first lines follow for illustration (recall that the notation `&→` is for field access—see Section 4.2.1; see [AMS14] for the complete model):

```
Definition ssl_parse_client_hello1 cont :=
  "ret" :=ssl_fetch_input(%"ssl", [ 5 ]sc) ;
  If \b %"ret" ≠ [ 0 ]sc Then
    Return
  Else (
  "buf" :=* %"ssl" &→ "in_hdr" ;
  "buf0" :=* %"buf" ;
  If \b (%"buf0" & [ 128 ]uc) ≠ [ 0 ]uc Then
    "ret" := [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c;
    Return
  Else ( ...
```

Dealing with Calls to 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.). We axiomatize their correctness in the form of Separation-logic triples.

Since we do not have yet an encoding of function calls, we formalize external functions by augmenting the `cmd` type with axioms. To respect the scoping rules of C, we axiomatize the fact that no local variable is modified by calls to external functions. Here follows the example of the standard `memcpy` function that copies the contents of the array pointed to by `src` to the array of the same length pointed to by `dest`:

```
(* MEMCPY(3): void *memcpy(void *d, const void *s, size_t l); *)
Definition size_t := g.-ityp: uint.
Definition void_p := g.-typ: ptyp (ityp uchar).
Axiom memcpy : ∀ ret, env_get ret σ = ⌊ void_p ⌋ →
  exp σ void_p → exp σ void_p → exp σ size_t → cmd.

Lemma memcpy_triple ret H e dest src len DEST SRC :
  ℤ◃ℕ (size SRC) = ℤ◃u len → ℤ◃ℕ (size DEST) = ℤ◃u len →
  { ‘! \b e = [ len ]pc ⋆ src ↦ SRC ⋆ dest ↦ DEST }
  memcpy ret H dest src e
  { ‘! \b e = [ len ]pc ⋆ src ↦ SRC ⋆ dest ↦ SRC }.
```

```
Admitted.

Lemma memcpy_input_inde ret H dest src len :
  modified_vars (memcpy ret H dest src len) = nil.
Admitted.
```

(The notation ‘! ... is for Separation logic assertions that do not depend on the heap.)

7.3 Formal Specification of the ClientHello Parsing Function

We want to prove that, given any 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` below) or succeeds (by returning 0, assertion `success`) in checking that the incoming ClientHello packet is valid. As pictured in Figure 8, a ClientHello packet is contained in a Handshake protocol packet that is itself contained in a Record protocol packet. In Section 6.2, we explained how we specified a ClientHello packet and how we write a decoding function (`ClientHello_decode`) to test whether a list of bytes represents a ClientHello packet. Similarly, we provide decoding functions (`TLSPlainText_header_decode` and `Handshake_header_decode`) for headers of Handshake and Record protocol packets. We use above functions to form a decoding function for complete ClientHello packets, which are the packets that the function `ssl_parse_client_hello` expects:

```
Definition RecordHandshakeClientHello_decode l :=
  let (a1, l1) := S621.TLSPlainText_header_decode l in
  let '(a2, m, l2) := S74.Handshake_header_decode l1 in
  let (a3, l3) := S7412.ClientHello_decode m l2 in
  (a1 && a2 && a3, l3).
```

We are now in a position to formally state the correctness of the parsing function `ssl_parse_client_hello`. The most important point of the specification is the use of the formal specification of ClientHello packets explained just above. It appears in the postcondition of the following Hoare triple:

```
0 Lemma POLAR_parse_client_hello_triple (SI : seq (int 8)) ...
1   PolarSSLAssumptions SI →
2   { init_ssl_var ⋆ init_bu ⋆ init_rb ⋆ init_id ⋆ init_ses ⋆
3     init_ciphers ⋆ init_ssl_context }
4   ssl_parse_client_hello
5   { error ∨ (success ⋆
6     !!((RecordHandshakeClientHello_decode SI).1)) ⋆
7     final_bu ⋆ final_rb ⋆ final_id ⋆ final_ses ⋆ init_ciphers ⋆
8     final_ssl_context }.
```

(The notation `!!( ... )` is for Separation logic assertions that do not depend on the state of the program.)

About the Precondition. The precondition (lines 2–3) specifies the initial state: the initial value of the variable `"ssl"` (assertion `init_ssl_var`) and the initial state of the heap. The latter is captured by a Separation logic formula that, intuitively, formalizes Figure 9. `init_bu` specifies 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_ciphers`, `init_ssl_context`) is space for the nonce of the TLS connection (resp. space for the session id, for the session id data structure, for the ciphers known by the server, and for the `ssl_context` data structure).

By way of example, let us look at the `init_ssl_context` assertion. It formalizes in terms of a mapsto formula the central data structure of Figures 9 and 10. Initially, the stage of the protocol is `S74.client_hello`[5], version fields are uninitialized (variables `majv0`, `minv0`, `mmaj0`, `mmin0`), `in_left` is set to 0 (no byte read so far), and other fields are pointers to other data structures:

```
Definition Ssl_context server_status majv minv mmaj mmin ses bu
  inleft md5s sha1s ciphers rb :=
  %"ssl" ↦ₗ mk_ssl_context server_status majv minv mmaj mmin
    (ptr◂phy ses) (ptr◂phy bu `+ `( 8 )_ptr_len)
    (ptr◂phy bu `+ `( 13 )_ ptr_len) inleft md5s sha1s
    (ptr◂phy ciphers) (ptr◂phy rb).
...
let init_ssl_context := Ssl_context (zext 24 S74.client_hello)
  majv0 minv0 mmaj0 mmin0 ses bu `( 0 )_32 md5s sha1s ciphers rb in
...
```

(`` `( i )_ n `` converts the mathematical integer `i` to a machine integer of `n` bits; `` `+ `` is for the addition of machine integers; `zext` performs zero-extension of machine integers.) Parameters `md5s` and `sha1s` are for cryptographic functions but we are not concerned with them in parsing.

About the Postcondition. The postcondition (lines 5–8) specifies in particular 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: Figure 10 provides a pictorial representation that can be compared with the initial heap state (Figure 9). `final_bu` says that the buffer array is filled with the incoming bytes (note that it is also proved during the verification that this buffer contains a prefix of the socket input). `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_ses` says that the client and the server have agreed on a common cipher, that the server indeed knows (i.e., it appears in the array `CI`), and that is recorded in the session id data structure. `final_ssl_context` says that the state of the protocol has moved to `S74.server_hello` state, that the version fields have been initialized, etc. For illustration, we display below the definition of `final_ses`:

```
Definition Final_ses SI CI ses id := ∃ i, ∃ k, ∃ chosen_cipher,
  let j := 2 * k in
  !!(chosen_cipher = zext 16 (SI `_ (compmeth + sess_len SI + j))
                `|| (SI `_ (compmeth + sess_len SI + j + 1))) *
  !!(chosen_cipher = CI `32_ i) *
  (ses ↦ₗ mk_ssl_session
    chosen_cipher `(ℤ◂ℕ (sess_len SI))_32 (ptr◂phy id)).
...
```

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

```
let final_ses := Final_ses SI CI ses id in
...
```

The first assertion gives the name `chosen_cipher` to two consecutive bytes sent by the client in the socket input `SI` and this value can also be found at index `i` in the `CI` array that contains the ciphers known by the server (`‘_` and `‘32_` are notations for indexing arrays of resp. 8-bit characters and 32-bit integers). `compmeth` is the starting index of the compression method field and `sess_len` is the length of the session id. They are defined according to the RFC for TLS by using the formal definitions introduced in Section 6 (i.e., not as magic numbers).

About the Assumptions made by the Programmer. The last part of the specification is the assumptions (line 1) made by the programmer of PolarSSL. First, it turns out that PolarSSL ignores the minor version fields and treats them as `S621.TLSv11_min` (i.e., “2” for “TLS version 1.1”) (actually, we already anticipated on this discussion when providing Figure 8). The original source code does not perform any check at all, so that we think it is deliberate and indicates an assumption rather than an error. Second, the original source code considers that the only compression method is the “null” compression method [DR08, §6.1]. This is a customary assumption because this is indeed the only compression method defined in the RFC, but this must be clearly stated to complete formal verification:

```
Definition PolarSSLAssumptions l :=
  l ‘_ min_ver = S621.TLSv11_min ∧
  l ‘_ min_req = S621.TLSv11_min ∧
  l |{ compmeth+1 + sess_len l + ciph_len l, comp_len l) =
    nseq (comp_len l) ‘( 0 )_8.
```

(Like `compmeth` and `sess_len` seen above, `min_ver`, `min_req`, `ciph_len`, and `comp_len` are indices and lengths of payloads defined using the formalization of the RFC; `nseq n a` is for the list of length `n` that contains only `a`’s.)

The formal specification we just explained is the soundness of parsing. One may also think about checking its completeness but it would not be possible to guarantee that parsing succeeds for any correct packet because of PolarSSL’s restrictions. For example, PolarSSL assumes that Record packets are larger that the Handshake packets they embed. This is nevertheless a reasonable restriction because “theoretically, a single handshake message might span multiple records, but in practice this does not occur” [Res00, p.70]. See Section 7.5 for more such restrictions.

7.4 Technical Overview of the Formal Verification

We have completed the formal verification of the lemma presented in the previous section (`POLAR_parse_client_hello_triple`). This shows that the C function that parses ClientHello packets in PolarSSL, once properly patched (see Section 7.5), is sound, i.e., that it only succeeds for packets correct w.r.t. the RFC for TLS. This claim has to be moderated by the trusted computing base of our experiment. Besides the soundness of Coq, we assume here that we have faithfully formalized the operational semantics of C, the RFC for TLS, and the target C function. Our formalization of the operational semantics of C makes assumptions about the size of integral types and of pointers; we do not think that our case study depends on

file	contents	spec	proof
rc5246.v	Sect. 6	924	334
POLAR_ssl_ctxt.v	Sect. 7.1	287	163
POLAR_library_functions.v	Sect. 7.2	31	0
POLAR_library_functions_pp.v	Pretty-printing	32	0
POLAR_library_functions_triple.v	Sect. 7.2	100	24
POLAR_parse_client_hello.v	Sect. 7.2	203	0
POLAR_parse_client_hello_pp.v	Pretty-printing	29	65
POLAR_parse_client_hello_header.v	Sect. 7.3 (definitions)	117	275
POLAR_parse_client_hello_triple1.v	Sect. 7.3 (spec.+proof)	95	731
POLAR_parse_client_hello_triple2.v	Proof	90	706
POLAR_parse_client_hello_triple3.v	Proof	118	1213
POLAR_parse_client_hello_triple4.v	Proof	134	642
	Total	2160	4153

Table II. Overview of the verification of ClientHello parsing (in terms of l.o.c.)

these assumptions but, generally speaking, C programs verified in this framework may not be portable. One may also fear errors when manually modeling the target C function; automation would mitigate this issue but also become a non-trivial part of the trusted computing base. Our opinion is that the verified function should be pretty-printed (like in Section 4.4) to be used instead of the original one, in which case the pretty-printer would fall into the trusted computing base. Observe that the Hoare logic is not part of the trusted computing base since it is proved sound w.r.t. the operational semantics.

Table II provides a line count for the various files involved in this experiment. This completes the quantitative overview of our formalization (see also Section 5.4). Regarding our case study, it turns out at the end that we needed to write about 24 lines of Coq proof script per line of C code. As we already pointed at in Section 5.4, we developed several tactics to reduce the size of proof scripts. Yet, this effort was limited by the performance of Coq's type-checking: as it often happens, automation using tactics leads to bigger proof terms, which slows down type-checking, and in our case this became a real hindrance. The type-checking performance bottleneck seems to be the formal verification of the (large) loop invariant of the nested loop in which the TLS server looks for a cipher matching the client's request. The size of proof scripts also owes much to technical subgoals dealing with integer overflow and pointer overrun checking for which we have not developed any specific automation yet.

7.5 Errors Found in the Source Code and Clarifications of the Specification

We found several bugs in the course of verification that led us to patch the original C source code. In particular, checks performed by the `ssl_parse_client_hello` function are not sufficient to ensure the packets are well-formed. For a concrete example, at the beginning of the execution, `ssl_parse_client_hello` 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 checks 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 is actually contained in the Handshake message (in other words, that `sess_len` is not too large w.r.t. `n`). Formal verification stumbles here because we later 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. The solution is to augment the check of `sess_len`'s value as follows:

```
n_old = n;
...
sess_len = buf[38];
if( (sess_len < 0 || sess_len > 32) || 45 + sess_len >= 5 + n_old)
{ return(POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO); }
```

We have to record the value of `n` in `n_old` because `n` is modified before the test we want to perform. `5` is the size of the header of the ClientHello packet. `45 + sess_len` is the length of the packet up to the session id. This magic number can actually be obtained in a systematic way by summing up all the fixed parts (i.e., removing the payloads of sub-packets) of the different packet sections up to session id (see Figure 8):

```
Definition csuites :=
  ℕ◃ℤ (S621.TLSPlainText_hd + S74.Handshake_hd +
  fixed_sz S621.ProtocolVersion + fixed_sz S7412.Random +
  fixed_sz S7412.SessionID).
```

(`csuites+1` is actually 45).

There are similar implementation errors for the checks for `ciph_len` and `comp_len`. Besides the obvious problem of potentially addressing unintented memory, there is another reason why the check regarding `comp_len` is important. Its value is used to check whether the incoming packet has a TLS extension (boolean predicate `client_extensions_present` in Section 6.2), and PolarSSL is supposed not to treat TLS extensions.

It should be noted that above implementation errors and their fixes are similar to the Heartbleed bug found in OpenSSL (CVE-2014-0160) or the GnuTLS buggy parsing of session IDs (CVE-2014-3466). The programmers simply forgot to exhaustively check the size of payloads because their specifications in the RFC is not explicit enough.

In addition to fix implementation errors, the formal verification of the parsing function `ssl_parse_client_hello` also helped clarify some of the restrictions imposed by PolarSSL. We already mentioned, for example, the fact that minor version numbers are treated as 2 (see Section 7.3). Also, PolarSSL requires the payload of the Handshake to be such that $45 \leq n \leq 512$ whereas the upper-bound indicated in the RFC is actually 2^{14} (see Section 6.2.1). We have informally indicated these restrictions in Figure 8 and they are captured in the formal verification by an intermediate step (see [AMS14]).

8. RELATED WORK

Formalization of the Semantics of C. Since we are aiming at concrete experiments of verification of C programs, we need not only an encoding of C but also to equip it with reasoning facilities, in particular Separation logic rules. We restricted ourselves to the core subset of C partly to keep our work manageable. There are several pieces of work that focus in greater depth on the sole formalization of C.

CompCert is a comprehensive formalization of C that supports most of ANSI C. In CompCert, recursive structures must be encoded "structurally": inner fields can only refer to enclosing fields [BL09, § 2.1]. This obviates the need to carry a typing environment. We did not choose 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 (Section 2.2) but it becomes more natural to write mutually recursive structures (Section 2.4). 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 in the formalization (Section 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 a simplification that we made and that is also made in related work on mechanization of Separation logic, e.g., [WKS+09]).

Work on the formalization of C now focuses on the more recent C11 standard [Kre13].

Ellison et al. [ER12] proposed a formalization of C that consists of an executable semantics built on top of the Maude rewriting system. It 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 [NGC08]. 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. In comparison, our work is oriented towards verification of functional properties, which are more general than safety. Our model of C instantiates some platform-dependent values (such as pointer size) to ease the type-checking of dependently-typed syntax. While this is a must-have for verification of embedded software, one may also want to verify portable source code. We believe that this can be achieved with a reasonable amount of work by making the size of pointers a parameter of our library.

Mechanization of Hoare Logics. Tuch proposed a formalization in the Isabelle proof-assistant of Hoare logic for C and applied it to a memory allocator [Tuc09]. A trusted C-to-HOL translation is responsible for encoding C types as Isabelle/HOL records accompanied with lemmas [Tuc09, §5.3]; padding is encoded in the form of extra fields [Tuc09, p.140]. Proofs do not fail when types are correct but "this is fragile and does not scale well" [Tuc09, 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 (Section 5.3). Like us, Tuch disallows access to the address of local variables, but this is a common restriction in Hoare logics for C (e.g., [AB07]).

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 [NMS+08] 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 [Chl11] is a framework that emphasizes "mostly automated verification". It uses an "idealized machine language" with arbitrary-precision words and infinite memory, without notion of alignment or padding. In contrast, our formalization of C takes into account realistic hardware constraints. In order to perform verification (Section 7.3), we use semi-automation with tactics similar to the one developed by Appel [App06].

Winwood et al. propose a different approach to interactive verification of C programs [WKS+09]. There is no Separation logic per se, but Hoare logic is used to establish simulations [WKS+09, § 5.2]. Application of this approach to PolarSSL verification would require the construction of a reference implementation, what would be a different way to formalize the RFC for TLS.

In CompCert, Separation logic was originally a side-project for the intermediate language Cminor only [AB07]. A Separation logic for the C encoding of CompCert has recently been formalized in Coq [DA13]. The absence of an intrinsic encoding like ours translates into additional type-checking side-conditions in reasoning rules that can nevertheless by discharged by programming a type-checker.

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 with parsing combinators based on invertible syntax descriptions to simplify the programming of reference implementations [ANO12]. The formalization introduced in Section 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 here and there in implementations. Producing formal specifications of packet formats has been a long-standing issue for which types has long been seen as a promising solution [MC00].

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 by using a wide set of automated theorem provers, or by generating Coq goals as a last resort. The whole stack has been used for example to verify C functions for numerical analysis [BCF+13]. 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 [BFCZ12].

9. CONCLUSION

TLS is a pervasive security protocol whose implementation errors can be disastrous, as exemplified by the recent Heartbleed bug (CVE-2014-0160). This indicates a real need for the formal verification of an implementation of TLS and our purpose here was to perform a concrete experiment with the Coq proof-assistant in that direction. For this purpose, we first introduced a new encoding of the core subset of the C language that only allows for correctly-typed programs to be modeled. The dependent types of Coq's language were instrumental in providing this intrinsic encoding. Though addressing only a subset of C, we took great care in providing a byte-level encoding that conforms to the C standard. We then went on providing a Separation logic to reason about C programs. In order to apply the above framework to a concrete implementation, we also formalized the packet formats from the standard RFC for TLS. Here again, dependent types came in handy to capture succinctly the length of payloads embedded in packet headers. In the process, we found some ambiguities in the original RFC document. Finally, we applied the above formalizations to the formal verification of a parsing function taken from an existing implementation of TLS, namely PolarSSL. We were able to discover and fix implementation errors and to provide a formal specification that exactly sorts out what this function is really doing.

Future Work. We would like to extend our subset of C with functions and local variables to avoid the axiomatic encoding of calls to external functions. We developed many tactics to shorten proof scripts but automation appears to slow down proof-checking. It seems important to us to optimize the formal verification task for larger experiments to be possible in practice. Once scalability issues are addressed, we plan to verify more parsing functions, hopefully leading to the construction of a formally-verified API for TLS implementations in C.

ACKNOWLEDGMENTS

This work is based on joint work with Nicolas Marti. A progress report was presented at the 7th Workshop on Programming Languages meets Program Verification (PLPV 2013) [AM13]. We are grateful to the anonymous reviewers whose comments have improved our work.

References

[AB07] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step Cminor. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10–13, 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 5–21. Springer, 2007.

[Aff14] Reynald Affeldt. Overview of the seplogC library. Available at `http://staff.aist.go.jp/reynald.affeldt/seplogC`, 2014. Implementation Notes.

[AM13] Reynald Affeldt and Nicolas Marti. Towards formal verification of TLS network packet processing written in C. In *7th Workshop on Program-*

ming Languages meets Program Verification, PLPV 2013, Rome, Italy, January 22, 2013, pages 35–46. ACM, 2013.

[AMS14] Reynald Affeldt, Nicolas Marti, and Kazuhiko Sakaguchi. An intrinsic encoding of a subset of C and its application to TLS network packet processing. Coq scripts available at `http://staff.aist.go.jp/reynald.affeldt/seplogC`, 2014. Coq documentation available at `http://staff.aist.go.jp/reynald.affeldt/coqdev`.

[ANO12] Reynald Affeldt, David Nowak, and Yutaka Oiwa. Formal network packet processing with minimal fuss: invertible syntax descriptions at work. In *6th Workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012*, pages 27–36. ACM, 2012.

[ANY12] Reynald Affeldt, David Nowak, and Kiyoshi Yamada. Certifying assembly with formal security proofs: the case of BBS. *Sci. Comput. Program.*, 77(10–11):1058–1074, 2012.

[App06] Andrew W. Appel. Tactics for separation logic. Available at `http://www.cs.princeton.edu/~appel/papers/septacs.pdf`, Jan. 2006. Early draft.

[BCF+13] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave equation numerical resolution: A comprehensive mechanized proof of a C program. *J. Autom. Reasoning*, 50(4):423–456, 2013.

[BFCZ12] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zalinescu. Verified cryptographic implementations for TLS. *ACM Trans. Inf. Syst. Secur.*, 15(1):3, 2012.

[BFN+06] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11–13, 2006*, pages 55–66. ACM, 2006.

[BHKM12] Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in Coq. *J. Autom. Reasoning*, 49(2):141–159, 2012.

[BL09] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *J. Autom. Reasoning*, 43(3):263–288, 2009.

[Bra11] Edwin Brady. Idris — systems programming meets full dependent types. In *5th ACM Workshop Programming Languages meets Program Verification, PLPV 2011, Austin, TX, USA, January 29, 2011*, pages 43–54. ACM, 2011.

[CDT14] The Coq Development Team. *The Coq Proof Assistant: Reference Manual*. INRIA, 2014. Version 8.4pl3. Available at `http://coq.inria.fr`.

[Chl11] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *32nd ACM SIGPLAN Conference*

on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011, pages 234–245. ACM, 2011.

[DA13] Josiah Dodds and Andrew W. Appel. Mostly sound type system improves a foundational program verifier. In *3rd International Conference on Certified Programs and Proofs, CPP 2013, Melbourne, VIC, Australia, December 11–13, 2013*, volume 8307 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2013.

[DR08] Tim Dierks and Eric Rescorla. The transport layer security (TLS) protocol version 1.2. Available at `http://tools.ietf.org/html/rfc5246`, Aug. 2008. IETF RFC 5246.

[ER12] Chucky Ellison and Grigore Rosu. An executable formal semantics of C with applications. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22–28, 2012*, pages 534–544. ACM, 2012.

[GM10] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *J. Formalized Reasoning*, 3(2):95–152, 2010.

[IJ02] Samuel P. Harbison III and Guy L. Steele Jr. *C: A Reference Manual*. Prentice Hall, 2002. 5th edition.

[Jen13] Jonas B. Jensen. Techniques for model construction in separation logic. Available at `http://itu.dk/~jobr/research/sl-model.pdf`, Sep. 2013.

[Kre13] Robbert Krebbers. Aliasing restriction of C11 formalized in Coq. In *3rd International Conference on Certified Programs and Proofs, CPP 2013, Melbourne, VIC, Australia, December 11–13, 2013*, volume 8307 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2013.

[Ler12] Xavier Leroy. The Compcert verified compiler, commented Coq development. Available at `http://compcert.inria.fr/doc/`, 2012. Version 1.11, 2012-07-13.

[MC00] Peter J. McCann and Satish Chandra. Packet types: Abstract specifications of network protocol messages. In *SIGCOMM*, pages 321–333, 2000.

[NGC08] Marius Nita, Dan Grossman, and Craig Chambers. A theory of platform-dependent low-level software. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7–12, 2008*, pages 209–220. ACM, 2008.

[NMS+08] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In *13th ACM SIGPLAN International Conference on Functional Programming, ICFP 2008, Victoria, BC, Canada, September 20–28, 2008*, pages 229–240. ACM, 2008.

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

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

[Res00] Eric Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison Wesley, 2000. 11th Printing.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structure. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22–25 July 2002, Copenhagen, Denmark*, pages 55–74. IEEE Computer Society, 2002.

[Rey08] John C. Reynolds. An introduction to separation logic (preliminary draft). Available at `http://www.cs.cmu.edu/~jcr/copenhagen08.pdf`, Oct. 2008.

[Sea06] Robert C. Seacord. *Secure Coding in C and C++*. Addison Wesley, 2006.

[Tuc09] Harvey Tuch. Formal verification of C systems code. *J. Autom. Reasoning*, 42(2–4):125–187, 2009.

[WKS+09] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. Mind the gap: A verification framework for low-level C. In *22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009, Munich, Germany, August 17–20, 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 500–515, 2009.