A Verified Protocol Buffer Compiler

Qianchuan Ye
Purdue University
USA
ye202@purdue.edu

Benjamin Delaware
Purdue University
USA
bendy@purdue.edu

Abstract

The code responsible for serializing and deserializing untrusted external data is a vital component of any software that communicates with the outside world, as any bugs in these components can compromise the entire system. This is particularly true for verified systems which rely on trusted code to process external data, as any defects in the parsing code can invalidate any formal proofs about the system. One way to reduce the trusted code base of these systems is to use interface generators like Protocol Buffer and ASN.1 to generate serializers and deserializers from data descriptors. Of course, these generators are not immune to bugs.

In this work, we formally verify a compiler for a realistic subset of the popular Protocol Buffer serialization format using the Coq proof assistant, proving once and for all the correctness of every generated serializer and deserializer. One of the challenges we had to overcome was the extreme flexibility of the Protocol Buffer format: the same source data can be encoded in an infinite number of ways, and the deserializer must faithfully recover the original source value from each. We have validated our verified system using the official conformance tests.

CCS Concepts • **Theory of computation** → **Program verification**; • **Software and its engineering** → **Software verification**; *Source code generation*;

Keywords Serialization, Program verification, Coq

ACM Reference Format:
Qianchuan Ye and Benjamin Delaware. 2019. A Verified Protocol Buffer Compiler. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '19), January 14–15, 2019, Cascais, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3293880.3294105

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CPP '19, January 14–15, 2019, Cascais, Portugal
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6222-1/19/01…$15.00
https://doi.org/10.1145/3293880.3294105

1 Introduction

Serialization is the process of encoding structured data into a binary representation according to a standardized format, usually in order to communicate with some external client or for persistent storage. In the case of distributed systems, serialized data is used to exchange messages, perform remote procedure calls, and orchestrate synchronization. Failure to produce or correctly interpret encoded data can result in lost or corrupted data, potentially placing the system into an inconsistent state. Given their importance, serializers and deserializers are particularly ripe targets for formal verification. This is especially true when serialization and deserialization code is included in the trusted code of a formally verified system, as bugs in those components can compromise the assurance case for the entire system [13].

There are no shortage of standardized serialization formats in the wild, including protocol-specific formats like TCP/IP and DNS [21, 23, 24], language-specific marshaling libraries [14, 20], human readable formats like JSON [6] and XML [7], and interface generators like ASN.1 [12] and Protocol Buffers [30]. Interface generators are particularly popular for application developers [16], because they offer a general-purpose, language-independent solution to serialization, while producing a more compact encoding than human readable formats. These systems provide a domain-specific language in which users can describe their application's data types, as well as a compiler for building a library of data type definitions, methods to manipulate those data types, and serializer and deserializers in a desired target language. Delegating the generation of this library to a compiler removes the possibility of user-introduced bugs in serialization and deserialization code, but these compilers may themselves have bugs. Numerous security vulnerabilities have been reported for various ASN.1 compilers in MITRE's Common Vulnerability Enumeration (CVE) [1–3]. Thus, removing the generated serializers and deserializers from the trusted code base is an important piece of building high-assurance systems that rely on interface generators.

In this paper, we do just that, verifying a Protocol Buffer compiler once and for all. This proof is parameterized over the data schema, such that we can derive functional correctness proofs for the serializer and deserializer generated for an arbitrary data description. One of the challenges here is that the Protocol Buffer standard was designed with flexibility, not verification, in mind. It imposes all the typical

challenges of verifying handwritten serializer and deserializer functions and more. Firstly, both the signature and the implementations of the generated functions are highly dependent on the supplied data description. Secondly, the format is highly flexible, such that the same structured data value can be encoded into many different binary strings; a correct deserializer must be able to recover this value from each of these. To overcome these challenges, we have made the following contributions:

- We have formalized a practical subset of version 3 of the Protocol Buffer standard in Coq [28], capturing its dependent data representations and its flexible encoding strategy.
- We have implemented serializer and deserializer generators, and proved them functionally correct with respect to the standard.
- We have evaluated our implementation by extracting OCaml implementations of a serializer and deserializer for the reference example in Protocol Buffer's official repository [17]. In addition, we have used the official conformance tests to validate that our formalization faithfully captures Protocol Buffer's informal specification.

Our system is available in an accompanying code supplement.

Key to our approach is a decomposition of the specification of the format into two separate layers which progressively relate a structured data value to its possible encodings. We then construct serializers and deserializers by composing together verified implementations for each of the intermediate layers, so that we are able to decompose the "end-to-end" correctness proof into proofs of correctness for each of the layers. This allows us to cleanly separate the representation of multiple encodings of a value from the specification of the bit-level representation of the encoded data, as Section 3 will discuss in more detail.

In order to concretize our discussion, we begin with an example of how our system can be used to derive serializers and deserializers from a data description. In Protocol Buffers, this description is typically called a *message descriptor*, and the structured values it describes are called *messages*. Consider the following simple descriptor for a timestamp message:

```
Definition Timestamp: Descriptor :=
  [(Singular (Base int64), "seconds", 1);
   (Singular (Base int32), "nanos", 2)].
```

Our "compilers" are simply functions which take a descriptor as an argument:

```
Definition encode_timestamp : ⟦Timestamp⟧ → Bytes :=
  encode_message Timestamp.
Definition decode_timestamp : Bytes → option ⟦Timestamp⟧ :=
  decode_message Timestamp.
```

⟦Timestamp⟧ is the Coq type of messages denoted by the descriptor Timestamp; Section 3 provides the complete details of this denotation function. We can use Coq's extraction mechanism to extract executable OCaml implementations of encode_timestamp and decode_timestamp.

We have proven soundness theorems for encode_message and decode_message too, which can be instantiated with a concrete message descriptor:

```
Theorem encode_timestamp_correct :=
  encode_message_correct Timestamp.
Theorem decode_timestamp_correct :=
  decode_message_correct Timestamp.
```

The statements of these soundness theorems are given in Section 4. These theorems can be used to prove end-to-end correctness of a larger verified system that makes use of these implementations.

The rest of the paper proceeds as follows: we begin by highlighting the flexibility of the Protocol Buffer format before giving its complete specification. We then discuss the generation of and soundness proofs for serializers and deserializers in Section 4. In Section 5, we evaluate our system by implementing a reference example, which we validate using the official conformance test. Section 6 presents related work before a discussion of future work in Section 7, which is followed by the conclusion.

2 An Introduction to Protocol Buffers

We begin with a brief introduction to Protocol Buffers, in order to give readers an intuition of the format. To define the shape of a Protocol Buffer message, a user provides a message descriptor in a ".proto" file that is fed to a Protocol Buffer compiler. The message descriptor for the timestamp example from Section 1 is as follows:

```
message Timestamp {
  int64 seconds = 1;
  int32 nanos = 2;
}
```

In this example, the first line specifies the name of the data type: Timestamp. Inside the curly brackets, each line defines a *field* of this data type, which consists of the *type*, the *name* and the *tag* of this field. The next section will discuss the types of fields in more detail. Names of fields are only used by the users to access or update these fields of parsed, structured data, and do not appear in the serialized data. The message's tags need to be unique numbers and they are used to identify the fields in the encoded binary format. From this description, a Protocol Buffer compiler will generate a data type implementation in a chosen target language, with an interface to manipulate, serialize, and deserialize messages.

2.1 Structured Data

Protocol Buffers support more types than just int32 and int64, including floating points, booleans, and strings. Types can

furthermore be either *scalar* or *repeated*. Scalar types include base types like integers, composite types like enumerations, and other user-defined types. The latter are typically called *embedded messages* or *embedded fields*. Repeated types are simply sequences of some scalar type, including embedded messages. The following data descriptor for a Person message includes examples of all these features:

```
message Person {
  int32 id = 1;
  string name = 2;
  repeated int32 advisors = 3;
  Timestamp last_updated = 4;
}
```

The id and name fields are scalar fields of type 32-bit integer and string, respectively. The advisors field is a sequence of integers. last_updated is an example of an embedded field, whose type is the Timestamp message defined above.

Importantly, every Protocol Buffer type has a *default value* that is used when a field is not included in the encoded message. The use of default values can reduce the size of messages, since it is not necessary to encode a field that has a default value. Not surprisingly, the default value of numeric types is 0, of strings is the empty string, and of repeated fields is the empty list. The default value of embedded messages, e.g., last_updated, is underspecified in the Protocol Buffer documentation. In our implementation, we choose to use option types to represent embedded messages, so that the default value of an embedded message is None. This aligns with the value of null used by the official Protocol Buffer C++ implementation.

2.2 Serialized Data

An encoded Protocol Buffer message is a binary string which is essentially a sequence of key-value pairs. Each key-value pair represents a field or a part of a field, where the key includes both the field tag and the type of a value. As an example, a timestamp message whose seconds field is 1 and nanos field is 10 can be encoded as the binary string: 08 01 10 0A. The first byte, 08, is a package of the tag and the *wire type* of a field. At a first approximation, this byte signifies that the subsequent byte, 01, is an integer and has a field tag of 1, i.e. it is the value of the seconds field. Similarly, the third byte, 10, indicates that the following byte is the value of the nanos field.

The format includes the wire type as part of the key to ensure that every pair contains enough information to determine the length of its value. In version 3 of the Protocol Buffer standard, there are only four wire types: varint, 32-bit, 64-bit and length-delimited. Section 3 discusses how wire types map to field types in more detail. Each wire type is associated with a distinct number, and determines how subsequent bytes are to be deserialized. Variable-length integers (varint) are encoded as *base 128 varints* [15]. In this format, the lower seven bits of each encoded byte represent the corresponding seven bits of the integer, while the most significant bit indicates whether subsequent bytes should be included. Somewhat counterintuitively, variable length integers are even used as the wire type for fixed-width numbers, as this allows smaller values to be encoded with fewer bytes. The length-delimited wire type is used for string-like types, repeated types, and embedded messages. Values of this wire type are serialized by first encoding the number of bytes of the encoded value as a varint, followed by the actual value. This encoding of length-delimited wire types makes the Protocol Buffer format a non-context free language.

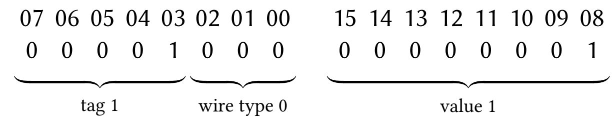

Figure 1. The encoded bits, 08 01, for seconds field

A field’s tag and its wire type are packaged together into a variable-length integer, with the three lowest bits encoding the wire type of the value and the higher bits encoding the field’s tag. An example is shown in Figure 1. We can now consider how to encode various fields of the Person message:

- A name field with a value of “Bob” is encoded as 12 03 42 6F 62. The lower three bits of the tag are 3 this time, indicating this is a length-delimited wire type, while the higher bits are the tag of name. A length-delimited wire type indicates that next varint is the number of bytes in the value, three in this case of the “Bob” value.
- An advisors field with a value of [1;2;3] is encoded as 1A 03 01 02 03. The first varint is, again, the tag of advisors and the length-delimited wire type. The second varint is the number of bytes of the value and then the value of each element follows, in order.
- A last_updated field whose embedded message has a seconds field of 1 and a nanos field of 10 is encoded as 22 04 08 01 10 0A. The first varint is the tag of last_updated and the wire type length-delimited. The second varint is the number of bytes in the embedded message, which is recursively encoded using the same process, resulting in the same string as the first example.

We noted above that 08 01 10 0A is only one of many possible encodings. Since Protocol Buffer strives for maximum flexibility, structured data may be encoded in many different ways. The standard permits many kinds of flexibility:

- Fields can be serialized in arbitrary order. A Timestamp message can be encoded by first encoding name and then id, or the other way around.
- Fields can be absent, indicating that the absent field should have the default value associated with its type.

Thus, both 08 01 and 10 0A are valid encodings of a Timestamp.

- Scalar fields can occur multiple times, with only the last one taking effect. This allows clients to update a field by simply appending more bytes to the encoded representation. Thus, 08 02 08 01 is a valid encoding of a Timestamp whose seconds field is set to 01.
- A message may include *unknown fields*, whose tags do not appear in the message descriptor. This feature is for backward-compatibility: one client may update its descriptor with additional fields which are unknown to other clients. Instead of generating errors, the other clients will simply ignore the unknown fields when deserializing.
- Repeated fields can be broken up into pieces which are encoded individually. Each piece may be a single element of the original list encoded as a scalar type, or a few elements encoded as a repeated type. The corresponding value of this field is the concatenation of all the pieces in order. The encoding may interleave other fields between each piece. As an example, the byte string 18 01 1A 02 02 03 is another valid encoding of the field advisors with its value set to [1;2;3]. Here, the first key-value pair has the advisors tag and a varint wire type, so the next byte is the first element of the list. The next key-value pair again has the advisors tag, but it has a length-delimited wire type to indicate the subsequent value is a slice of the list.
- Embedded messages can be similarly broken up into pieces, with each piece containing a few fields of the message. As an example, 22 02 08 01 22 02 10 0A encodes the last_updated field by individually encoding each of the fields of the embedded message.

In summary, there are many different ways to encode a particular message, all of which need to be captured by our specification of the Protocol Buffer format.

3 A Formalization of Protocol Buffers

This section presents a Coq formalization of a subset of the Protocol Buffer format that captures the key features of the standard. Our formalization includes a model of structured data built from a selection of base types, repeated and scalar fields, and embedded messages, as well as a precise specification of the valid binary encodings of a message. Section 7 discusses the missing features in more detail, but they represent a straightforward extension of this core. We present our formalization in pseudocode for clarity; the full implementation is included in the accompanying code supplement.

3.1 Encoding Descriptors and Messages

We begin by discussing our embedding of message descriptors and messages in Coq. Message descriptors are defined by the (mutually) inductive types shown in Figure 2.

```
Descriptor : Type := list Field
Field : Type := PBType × string × N
PBType : Type := Singular SingularType | Repeated SingularType
SingularType : Type := Base BaseType | Embedded Descriptor
BaseType : Type := int32 | int64 | fixed32 | fixed64 | string
WireType : Type := varint | 32bit | 64bit | length−delimited
```

Figure 2. Definition of Message Descriptor

A message descriptor is just a list of *field descriptors*[1]. Each field descriptor contains its *Protocol Buffer type*, denoted by PBType, its name, and its tag. Because the name is not used when encoding, a message descriptor is effectively a mapping from tags to their associated Protocol Buffer types. A Protocol Buffer type can be either a singular or repeated type. A singular type is either a *base type* or an embedded message, which takes another descriptor as its argument. Our implementation only supports a subset of Protocol Buffer base types, but it is a simple matter to add more base types. The Coq embedding of our Timestamp and Person message descriptors is straightforward:

```
Timestamp : Descriptor :=
  [(Singular (Base int64), "seconds", 1);
   (Singular (Base int32), "nanos", 2)]
Person : Descriptor :=
  [(Singular (Base int32), "id", 1);
   (Singular (Base string), "name", 2);
   (Repeated (Base int32), "advisors", 3);
   (Singular (Embedded Timestamp), "last_updated", 4)]
```

A particular message descriptor, desc, can be embedded as an inductive data type in Coq via a dependently-typed denotation function ⟦desc⟧ , so that the messages associated with desc are simply values of its denotation. Shallowly embedding messages in this way lets us leverage Coq's type checker to ensure that messages are well-formed with respect to a particular message descriptor. Figure 3 shows the definition of the denotation function for message descriptors; we overload this notation to define denotation functions for all the data types in Figure 2.

Since Coq does not allow records to be defined programmatically, we denote the descriptor into a generic Tuple type. A Tuple is essentially a fixed-length heterogeneous list [8], indexed by a list of types. Each element in a Tuple corresponds to a field in the descriptor. For example, ⟦Timestamp ⟧ = Tuple [N; N]. The first element, of type N, is the value of seconds and the second element is the value of nanos . Hence the tuple ts : ⟦Timestamp⟧ := [1, 10] is a message of Timestamp with seconds := 1 and nanos := 10. Because

[1]Our implementation actually uses length-indexed vectors; we use lists here for presentation purposes

```
(* ⟦·⟧ : Descriptor → Type *)
⟦[field_1; field_2; ... ; field_n]⟧ := Tuple [ ⟦field_1⟧ ; ⟦field_2⟧ ; ... ; ⟦field_n⟧ ]

(* ⟦·⟧ : Field → Type *)
⟦(ty, name, tag)⟧ := ⟦ty⟧

(* ⟦·⟧ : PBType → Type *)
⟦Singular (Base ty)⟧ := ⟦ty⟧
⟦Repeated (Base ty)⟧ := list ⟦ty⟧
⟦Singular (Embedded desc)⟧ := option ⟦desc⟧
⟦Repeated (Embedded desc)⟧ := list ⟦desc⟧

(* ⟦·⟧ : WireType → Type *)
⟦varint⟧ := N
⟦32bit⟧ := word 32
⟦64bit⟧ := word 64
⟦length−delimited⟧ := list (word 8)

(* ⟦·⟧ : BaseType → Type *)
⟦ty⟧ := ⟦toWireType ty⟧

(* toWireType : BaseType → WireType *)
toWireType int32 := varint
toWireType int64 := varint
toWireType fixed32 := 32bit
toWireType fixed64 := 64bit
toWireType string := length−delimited
```

Figure 3. Denotation of Message Descriptor

each Tuple type has a fixed-length, users can only access defined fields. As a convenience, we have defined special functions for accessing a particular field by name: ts!"seconds"= 1, for example. Attempting to access an unknown field will generate a type error.

The denotation of a field is just the denotation of the underlying Protocol Buffer type. The denotation of a Protocol Buffer type is also straightforward, although as we mentioned in Subsection 2.1 the denotation of a single embedded message is an option type. Base types are first mapped to a wire type via the toWireType function; these are then denoted to normal Coq types. Since all base types with the same wire type are encoded in the same way, we do not distinguish their embeddings in Coq.

```
⟦Timestamp⟧ = Tuple [N; N]
⟦Person⟧ = Tuple [N; list (word 8); list N; option (Tuple [N; N])]

Definition person : ⟦Person⟧ := [1; "Bob"; [1; 2; 3]; Some [1; 10]].
```

Figure 4 defines some operations on message descriptors and messages that will be useful later.

3.2 Specifying Binary Formats

We are now equipped to specify the valid bit-level encodings of the messages associated with a particular descriptor.

- descriptorOK : Descriptor → Prop
  This predicate asserts that the given descriptor is a well-formed descriptor: the tags are within a valid range and the names are not empty, and, most importantly, tags and names are unique. The uniqueness of tags is crucial for the soundness of serialization.
- default : ∀ desc : Descriptor, ⟦desc⟧
  Function default takes a descriptor and returns its default message. E.g., default Person = [0; ""; []; None].
- ·∈ ·: N → Descriptor → Prop
  tag ∈ desc asserts that one of the fields in desc has the given tag. Similarly, we write tag ∉ desc as the negation of this assertion.
- ·[·] : ∀ desc : Descriptor, BoundedTag desc → PBType
  desc[tag] gets the Protocol Buffer type of the field with the given tag. E.g., Person[1] = Singular (Base int32).
- ·[·] : ∀ {desc : Descriptor}, ⟦desc⟧ → ∀ tag : BoundedTag desc, ⟦desc[tag]⟧
  msg[tag] looks up the value of the field with the given tag in msg. The descriptor desc is implicit. E.g., person[1] = 1.
- ·[·↦ ·] : ∀ {desc : Descriptor}, ⟦desc⟧ → ∀ tag : BoundedTag desc, ⟦desc[tag]⟧ → ⟦desc⟧
  msg[tag ↦ val] updates the value of the field with the given tag in msg to the new value val. The descriptor desc is implicit. E.g., person[1 ↦ 2] = [2; "Bob"; [1; 2; 3]; Some [1; 10]].

Figure 4. Operations on descriptors and messages. BoundedTag desc is a tag that must appear in desc: it is essentially a numeric tag, packed with a witness of its presence. We do not have to worry about the operations accepting a non-existing tag this way.

These specifications will form the correctness criteria for serializers and deserializers. One natural way to specify such an encoding is via a dependently-typed function: format' : ∀ desc : Descriptor, ⟦desc⟧ → Bytes. Unfortunately, such a function can only capture one possible encoding of a particular message, which is insufficient for specifying the correctness of a Protocol Buffer deserializer. Instead, we chose to specify the valid encodings of a message as a *relation*: format : ∀ desc : Descriptor, ⟦desc⟧ → Bytes → Prop. We call such a relation a *format* of a particular message descriptor. More generally, given a source type S and a target type T, a format of type S → T → Prop is a relation that relates any source to its valid targets. Given a format fmt, if (s, t)∈ fmt, then t is a valid encoding for s.

Instead of building this format relation directly, we construct it as the composition of two intermediate relations, as illustrated in Figure 5. This structure allows us to decompose the proofs of correctness for serializers and deserializers, which rely on this relation, into more manageable pieces. The first of these layers covers the sources of flexibility of

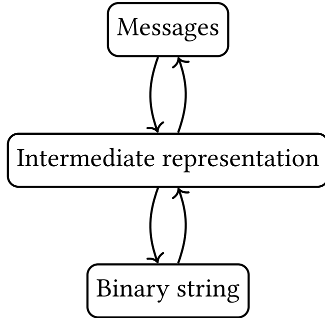

Figure 5. The 2-layer architecture

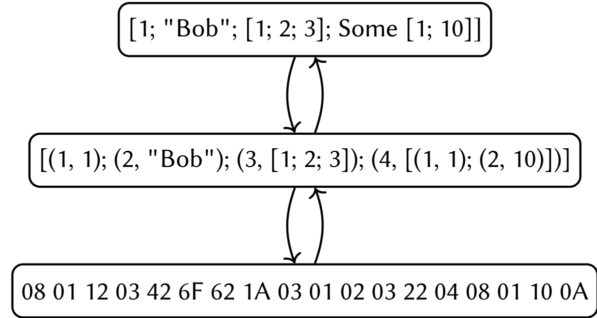

Figure 6. Relating a Person message to its encoding

the Protocol Buffer standard described in Subsection 2.2 by relating a message to a set of valid intermediate representations. The intermediate representation, which will be called IR in the remainder of this paper, is a linearized representation of a message which fixes the logical sequence of key value pairs that will be included in the serialized bitstring. This IR is one step closer to the final bitstring, but defers bit-level details to the next layer. As an example, one possible representation of the previous Timestamp example at this layer would be [(1, 1), (2, 10)]. This format is encoded as an inductively defined data type in Coq:
format_to_ir : ∀ desc : Descriptor, ⟦desc⟧ → IR → Prop.

The second layer relates IR values to binary strings. This layer does not contain any non-determinism, so the format specification is quite straightforward: we sequentially format the elements in the IR list using a library of formats for wire types. For each element in the IR, we format its tag and its corresponding wire type, and finally the actual value according to its type. The resulting binary strings are concatenated to obtain the binary encoding of the entire message. The format for this layer has type:
format_ir_to_bs : Descriptor → IR → Bytes → Prop.

Combining the intermediate formats from these two layers, we obtain an end-to-end specification format_message : ∀ desc : Descriptor, ⟦desc⟧ → Bytes → Prop, as the composition of the relations given the descriptor:

$$(s, t) \in \text{format_message} \leftrightarrow \\ \quad \exists\, t_1, (s, t_1) \in \text{format_to_ir} \wedge (t_1, t) \in \text{format_ir_to_bs}$$

Figure 6 shows an encoding of the Person message from before which is permitted by this format. We first relate the encoding of this message as an inductive data type to its IR. The non-deterministic format relation allows infinitely many IRs from this message; Figure 6 shows one such IR. Note that the value of field last_updated is itself an IR value representing the embedded Timestamp message. The next layer relates this particular IR value to its binary encoding.

Relating structured message to intermediate values To explain how the first layer relates a structured message to its possible intermediate representations, we first provide a precise type definition for IR:

$$\begin{aligned} \text{IR} &: \text{Type} \coloneqq \text{list IRElm} \\ \text{IRElm} &: \text{Type} \coloneqq \mathbb{N} \times (\Sigma\ (w : \text{WireType})\,.\,[\![w]\!] \\ &\qquad + \Sigma\ (ty : \text{BaseType})\,.\,[\![ty]\!] \\ &\qquad + \Sigma\ (ty : \text{BaseType})\,.\,\text{list}\ [\![ty]\!] \\ &\qquad + \text{IR}) \end{aligned}$$

The elements of an IR value are simply pairs of tags and a disjoint sum of values. We omit the constructors of the sum type and the first component of the dependent products when they are obvious from the context. Thus, we will write (1, 1) for an element representing the second field with value 1 of Timestamp, instead of $(1, \mathsf{in_L}(\mathsf{in_R}(\text{int64}, 1)))$.

Readers may wonder why values are not simply the denotation of tag's associated type, i.e. ⟦desc[tag]⟧ , where desc is the descriptor of the source message. One reason is that such an encoding does not allow for unknown tags. This is the motivation for including the first component of the sum type, which represents fields of arbitrary wire types ⟦w⟧ . Another reason is that fields of repeated type can be broken up into pieces, where each piece might be a single value or a list of values, so the type of this tag can be either ⟦ty⟧ or list ⟦ty⟧ . The IR for the advisors field of the previous example could be [(3, [1; 2; 3])] or [(3, 1), (3, 2), (3, 3)], for example. In addition, if a field is an embedded message, its value will be a nested IR. For example, the IR for person's last_updated field is (4, [(1, 1); (2, 10)]), whose value is the IR for the message of Timestamp.

Note that this definition allows an IR to be inconsistent with a message descriptor: if desc[tag] is Singular (Embedded desc'), for example, the associated value has to be an IR value. For this reason, we have developed a well-formedness property for IR values with respect to a message descriptor desc. This property formalizes the set of valid sequences of key-value pairs allowed by the Protocol Buffer documentation. The well-formedness property serves two purposes: it is the criterion that deserializers use to discard nonsensical sequences, and it allows the format in the second layer to assume all the sources are valid, simplifying the soundness proofs for that layer.

Definition 3.2 (Well-formedness of IR and IRElm)**.** We say an IR value is well-formed if all of its elements are well-formed, where an element (tag, v) is well-formed if it satisfies the following rules:

1. If tag $\notin$ desc, v's type is $[\![\text{w}]\!]$ for some wire type w.
2. If tag $\in$ desc and desc[tag] = Singular (Base ty), v's type is $[\![\text{ty}]\!]$.
3. If tag $\in$ desc and desc[tag] = Repeated (Base ty), v's type is either $[\![\text{ty}]\!]$ or list $[\![\text{ty}]\!]$. In the case that toWireType ty = length−delimited, only $[\![\text{ty}]\!]$ is permitted.
4. If tag $\in$ desc, and desc[tag] = Singular (Embedded d') or desc[tag] = Repeated (Embedded d') for some descriptor d', v is a well-formed IR value.

The third rule ensures that if ty has a length-delimited wire type, v's type cannot be list $[\![\text{ty}]\!]$. Otherwise, it would not be possible to tell whether the encoded value is a list of strings or a single string. For example, assume we have a field with a tag of 1, type of Repeated (Base string), and a value of ["Alice", "Bob"]. If this value can be formatted as (1, ["Alice", "Bob"]), it is impossible to tell if the on-the-wire value represents a single string or a list of strings. Hence, the standard disallows this case. Similarly, if desc[tag] = Repeated (Embedded desc'), v's type can only be IR.

Given this definition of IR, the intermediate format can be directly expressed as an inductively defined relation. We say msg' $\vdash$ ir $\simeq$ msg if ir correctly represents a "path" from msg' to msg. We call msg' the initial message and msg the result message, with ir explaining how to update the fields of msg' to arrive at msg. Figure 7 gives the definition of this relation. A complete message with descriptor desc can be related to its intermediate representations using an "default" initial message:

$$\text{ir} \simeq \text{msg} \equiv \text{default desc} \vdash \text{ir} \simeq \text{msg}$$

Intuitively, these rules spell out how to update the initial message to produce the result message by interpreting the IR as a series of updates. Seen in this manner, the IRNil rule is the base case of this process: an empty IR does not update the message, resulting in the same initial and result messages. The remaining rules explain how to perform a single update using the last element of the IR, with each rule updating the initial message according to the tag and the type of the value. Each judgment handles one aspect of the flexible encoding described in Subsection 2.2. For example, IRUnknown encodes the possibility of unknown fields, while IRSingular encodes the possibility of overwriting values. The treatment of missing fields is slightly more subtle: if a field is missing in ir, then there is no rule to update this particular field, so the field will retain the same value as the initial message. This is the reason the relation for complete messages uses a default initial message.

$$\frac{}{\text{msg} \vdash [] \simeq \text{msg}}\ (\text{IRNil})$$

$$\frac{\text{msg}' \vdash \text{ir} \simeq \text{msg} \quad \text{tag} \notin \text{desc} \quad \text{w} : \text{WireType} \quad \text{v} : [\![\text{w}]\!]}{\text{msg}' \vdash \text{ir} +\!\!+ [(\text{tag}, \text{v})] \simeq \text{msg}}\ (\text{IRUnknown})$$

$$\frac{\text{msg}' \vdash \text{ir} \simeq \text{msg} \quad \text{tag} \in \text{desc} \quad \text{desc[tag]} = \text{Singular (Base ty)} \quad \text{v} : [\![\text{ty}]\!]}{\text{msg}' \vdash \text{ir} +\!\!+ [(\text{tag}, \text{v})] \simeq \text{msg[tag} \mapsto \text{v]}}\ (\text{IRSingular})$$

$$\frac{\text{msg}' \vdash \text{ir} \simeq \text{msg} \quad \text{tag} \in \text{desc} \quad \text{desc[tag]} = \text{Repeated (Base ty)} \quad \text{v} : [\![\text{ty}]\!]}{\text{msg}' \vdash \text{ir} +\!\!+ [(\text{tag}, \text{v})] \simeq \text{msg[tag} \mapsto \text{msg[tag]} +\!\!+ \text{[v]]}}\ (\text{IRRepeatedSingle})$$

$$\frac{\text{msg}' \vdash \text{ir} \simeq \text{msg} \quad \text{tag} \in \text{desc} \quad \text{desc[tag]} = \text{Repeated (Base ty)} \quad \text{v} : \text{list } [\![\text{ty}]\!] \quad \text{toWireType ty} \neq \text{length−delimited}}{\text{msg}' \vdash \text{ir} +\!\!+ [(\text{tag}, \text{v})] \simeq \text{msg[tag} \mapsto \text{msg[tag]} +\!\!+ \text{v]}}\ (\text{IRRepeatedList})$$

$$\frac{\text{msg}' \vdash \text{ir} \simeq \text{msg} \quad \text{tag} \in \text{desc} \quad \text{desc[tag]} = \text{Singular (Embedded desc')} \quad \text{msg}'' : [\![\text{desc}']\!] \quad \text{msg[tag]} = \text{None} \quad \text{default desc}' \vdash \text{ir}' \simeq \text{msg}''}{\text{msg}' \vdash \text{ir} +\!\!+ [(\text{tag}, \text{ir}')] \simeq \text{msg[tag} \mapsto \text{Some msg}'']}\ (\text{IREmbeddedNone})$$

$$\frac{\text{msg}' \vdash \text{ir} \simeq \text{msg} \quad \text{tag} \in \text{desc} \quad \text{desc[tag]} = \text{Singular (Embedded desc')} \quad \text{msg}''\ \text{msg}''' : [\![\text{desc}']\!] \quad \text{msg[tag]} = \text{Some msg}'' \quad \text{msg}'' \vdash \text{ir}' \simeq \text{msg}'''}{\text{msg}' \vdash \text{ir} +\!\!+ [(\text{tag}, \text{ir}')] \simeq \text{msg[tag} \mapsto \text{Some msg}''']}\ (\text{IREmbeddedSome})$$

$$\frac{\text{msg}' \vdash \text{ir} \simeq \text{msg} \quad \text{tag} \in \text{desc} \quad \text{desc[tag]} = \text{Repeated (Embedded desc')} \quad \text{msg}'' : [\![\text{desc}']\!] \quad \text{default desc}' \vdash \text{ir}' \simeq \text{msg}''}{\text{msg}' \vdash \text{ir} +\!\!+ [(\text{tag}, \text{ir}')] \simeq \text{msg[tag} \mapsto \text{msg[tag]} +\!\!+ [\text{msg}'']]}\ (\text{IREmbeddedRepeated})$$

Figure 7. Inference rules for msg' $\vdash$ ir $\simeq$ msg

These rules can be used to define the relation used by the first layer:

```
format_to_ir (desc : Descriptor) (msg : ⟦desc⟧ ) (ir : IR) : Prop :=
  default desc ⊢ ir ≃ msg
```

We prove the following lemma regarding well-formedness of intermediate values, in order to allow the second layer to safely assume that its source value is always well-formed.

Lemma 3.3. *Given descriptor desc, for any message msg :* $[\![\textit{desc}]\!]$ *and ir : IR, if default desc* $\vdash$ *ir* $\simeq$ *msg, then ir is well-formed.*

- format_varint. The format for varints.
- format_word. The format for words, which takes an extra argument as its size. E.g., format_word 32 formats a 32-bit word to binary string.
- format_list. The format for list, which takes another format for its underlying types. E.g., format_list format_varint formats a list of varints.
- format_length_delimited. This format is just a composition of previous formats. It formats a list to a binary string and then prepends the size of this string as varint before it. E.g., format_length_delimited format_varint formats a list of varints with the size of the resulting string.

Figure 8. Formats for denotation types

$$\frac{\text{(ir, bs)} \in \text{format_list format_ir_elm_to_bs}}{\text{(ir, bs)} \in \text{format_ir_to_bs}} \;(\text{IRFormat})$$

$$\frac{\text{v} : [\![\text{ty}]\!] \quad \text{(v, bs)} \in \text{format_ir_val_to_bs} \quad \text{(pack tag (toWireType ty), bs')} \in \text{format_varint}}{\text{((tag, v), bs' ++ bs)} \in \text{format_ir_elm_to_bs}} \;(\text{IRElmSingular})$$

$$\frac{\text{ir : IR} \quad \text{(ir, bs)} \in \text{format_ir_val_to_bs} \quad \text{(pack tag length-delimited, bs')} \in \text{format_varint}}{\text{((tag, ir), bs' ++ bs)} \in \text{format_ir_elm_to_bs}} \;(\text{IRElmIR})$$

$$\frac{\text{v} : [\![\text{ty}]\!] \quad \text{(v, bs)} \in \text{format_singular_val}}{\text{(v, bs)} \in \text{format_ir_val_to_bs}} \;(\text{IRValSingular})$$

$$\frac{\text{ir : IR} \quad \text{(ir, bs)} \in \text{format_length_delimited format_ir_elm_to_bs}}{\text{(ir, bs)} \in \text{format_ir_val_to_bs}} \;(\text{IRValIR})$$

Figure 9. A subset of format_ir_to_bs relation

Relating intermediate values to binary strings The second layer of the relation relies on subformats that relate each of the denoted types to binary strings. The most important of these are listed in Figure 8.

A subset of this relation is shown in Figure 9. The relation format_singular_val relates a singular wire type value to a bitstring by using the formats corresponding to the value's type. The pack function makes a package of tag and wire type, as explained in Subsection 2.2. Figure 9 shows the cases when the value of an IR element is a singular type or an IR value, but other cases can be derived in a similar manner. Note that the rules in Figure 9 do not have to check if the tag matches the type in descriptor, thanks to the well-formedness guarantee provided by the first layer.

```
encode_to_ir [] [] := []
encode_to_ir ((tag, ty)::desc) (v::msg) :=
  encode_val_to_ir tag ty v ++ (encode_to_ir desc msg)

encode_val_to_ir tag (Singular (Base ty)) v := [(tag, v)]
encode_val_to_ir tag (Repeated (Base ty)) [v1; v2; ... ; vn] :=
  if ty ≠ length−delimited then [(tag, [v1; v2; ... ; vn])]
  else [(tag, v1); (tag, v2); ... ; (tag, vn)]
encode_val_to_ir tag (Singular (Embedded desc)) (Some v) :=
  [(tag, encode_to_ir desc v)]
encode_val_to_ir tag (Singular (Embedded desc)) None := []
encode_val_to_ir tag (Repeated (Embedded desc)) [v1; v2; ... ; vn] :=
  [(tag, encode_to_ir desc v1); (tag, encode_to_ir desc v2); ... ;
   (tag, encode_to_ir desc vn)]
```

Figure 10. Definition of encode_to_ir

4 Sound Generation of Serializers and Deserializers

This section first discusses how we generate serializers and deserializers from a message descriptor. We then show how the specification of the Protocol Buffer format from the previous section is used to state and prove these functions correct.

4.1 Generating Serializers

A serializer for a particular message descriptor maps messages of that descriptor type to bitstrings:

encode_message : ∀ desc : Descriptor, ⟦desc⟧ → Bytes.

Much like the Protocol Buffer format, encode_message is implemented as a composition of functions to and from intermediate values. The first of these functions: encode_to_ir: ∀ desc : Descriptor, ⟦desc⟧ → IR is straightforward, and shown in Figure 10: each field in the given message is encoded in turn. Although the format admits many possible intermediate encodings of a message, encode_to_ir chooses the "canonical" one, in that it makes the same choices as the reference implementation when possible. In particular, it encodes a repeated field as a length-delimited list, unless the field is a repeated string. The function implementing the last layer, encode_ir_to_bs: Descriptor → IR → Bytes, is identical to the format relation, which dictates a single encoding for each IR value. The end-to-end serializer is simply a composition of these functions:

```
encode_message (desc : Descriptor) : ⟦desc⟧ → Bytes :=
  (encode_ir_to_bs desc) ∘ (encode_to_ir desc)
```

Correctness of Generated Serializers A correct serializer for a message descriptor is a refinement of the format for that descriptor, in the sense that it settles on a single target value allowed by the format relation for every message:

Theorem 4.1 (Soundness of Protocol Buffer Serializer)**.** *For all well-formed descriptors, desc, encode_message desc will*

map each source message to a target encoding permitted by format_message desc:

$$\forall\ s\ t.\ \ encode_message\ desc\ s = t \longrightarrow (s,\ t) \in format_message\ desc$$

This soundness proof is decomposed into proofs of correctness for the intermediate functions. The proof of correctness for encode_to_ir is the more interesting of the two, and proceeds by nested induction on both the intermediate representation and the message descriptor. The key lemma needed for this proof is:

Lemma 4.2. *Given desc, msg msg' : $[\![desc]\!]$, tg and ty. If the following assumptions hold:*

1. *msg' ⊢ ir ≃ msg*
2. *descriptorOK (tg, ty)::desc*
3. *every tag in ir is also in desc*

we can conclude that v::msg' ⊢ ir ≃ v::msg for any v : $[\![ty]\!]$.

This lemma captures the fact that when a descriptor is augmented with a new field, and a new value is accordingly added to its message, if the tag of this new field is not present in ir, this ir will preserve the transition to the new message modulo the new value. The second assumption ensures that tg ∉ desc, since tags are unique, while the third ensures that tg is not in ir.

4.2 Generating Deserializers

A deserializer for a particular message descriptor is a partial function from bitstrings to messages:

decode_message : ∀ desc : Descriptor, Bytes ⟶ option $[\![desc]\!]$.

A decoder should signal an error via None if the input bitstring is malformed, i.e. not permitted by the format for the message descriptor. Similar to the implementation of serializers, decode_message is defined as a composition of intermediate functions. The functions are algorithmically straightforward, although ensuring their termination requires some care. The function for the first layer, decode_from_ir: ∀ desc : Descriptor, IR ⟶ option $[\![desc]\!]$ closely mirrors the inference rules in Figure 7, iteratively updating the default message using the key-value pairs in the intermediate representation. The second function, decode_ir_from_bs: Descriptor ⟶ Bytes ⟶ option IR inverts encode_ir_to_bs by decoding the fields from the bitstring using counterparts to the formats in Figure 8 and concatenating them to build the intermediate representation. The functions rely on a fuel parameter to guarantee termination.

In order to ensure that decoders have enough fuel, they rely on measure functions for binary strings and IR values, with the measurement of a binary string being its length. The measurement of an intermediate representation is the total number of elements it contains, including the elements from any embedded messages. For example, the IR in Figure 6 has six total elements: four for the outermost IR value and two for the inner IR value in last_updated. Another consideration is that decode_ir_from_bs needs to check whether a bitstring is malformed, e.g., the decoded wire type has to be consistent with the decoded tag.

The implementation of decode_message is the composition of these two functions:

```
decode_message (desc : Descriptor) (bs : Bytes) : option [[desc]] :=
  let ir := decode_ir_from_bs desc bs in
  match ir with
    | Some ir' ⇒ decode_from_ir desc ir'
    | None ⇒ None
  end.
```

Correctness of Generated Deserializers A correct decoder should recover a message from all of its possible encodings, and signal an error if the input bitstring does not encode *any* message:

Theorem 4.3 (Soundness of Protocol Buffer Deserializers)**.** *For all well-formed descriptors, desc, decode_message desc will map every bitstring in the codomain of format_message desc to a related source value, returning None otherwise:*

$$\forall\ s\ t.\ \ (s,\ t) \in format_message\ desc \longrightarrow decode_message\ t = Some\ s\ \wedge$$
$$\forall\ s\ t.\ \ decode_message\ desc\ t = Some\ s \longrightarrow (s,\ t) \in format_message\ desc$$

The proof of this theorem is derived from lemmas about the correctness of decode_ir_from_bs and decode_from_ir. The proof of correctness for decode_ir_from_bs requires a proof that it preserves the well-formedness of intermediate messages. Since its format assumes that the source IR is always well-formed, the deserializer needs to discard any invalid IRs.

Lemma 4.4. *For any descriptor desc and binary string bs, if descriptorOK desc and decode_ir_from_bs desc bs = Some ir, then ir is well-formed.*

The soundness proof for decode_from_ir is by induction on the fuel. Recall that the value of an IR element can itself be an IR value, so inducting on the fuel parameter provides a strong enough induction hypothesis to handle any embedded IR values.

5 Evaluation

To demonstrate the utility of our system, we have reimplemented an example from the Protocol Buffer official repository [2]. The descriptor used in this "address book" example describes a message containing someone's contact information, including their name, email, and phone numbers. The example comprises two programs: "add_person" prompts a user to input the contact information, serializes it, and adds it to a small database file; "list_people" reads a database file

[2]This example can be found in `/examples/{addressbook.proto,add_person.cc,list_people.cc}`.

in the Protocol Buffer format and prints all the contacts in it. The message descriptor has a straightforward embedding as a Descriptor:

```
Definition Timestamp : Descriptor :=
  [(Singular (Base int64), "seconds", 1);
   (Singular (Base int32), "nanos", 2)].

Definition PhoneNumber : Descriptor :=
  [(Singular (Base string), "number", 1);
   (Singular (Base int32), "type", 2)].

Definition Person : Descriptor :=
  [(Singular (Base string), "name", 1);
   (Singular (Base int32), "id", 2);
   (Singular (Base string), "email", 3);
   (Repeated (Embedded PhoneNumber), "phones", 4);
   (Singular (Embedded Timestamp), "last_updated", 5)].

Definition AddressBook : Descriptor :=
  [(Repeated (Embedded Person), "people", 1)].
```

To implement the serializer and deserializer for AddressBook messages, we simply concretize the parametrized descriptor:

```
Definition encode_addressbook := encode_message AddressBook.
Definition decode_addressbook := decode_message AddressBook.
```

To execute these functions, we used Coq's extraction mechanism to produce OCaml modules with message serialization and deserialization functions. We then linked these modules with OCaml implementations of "add_person" and "list_people" which handled IO operations. To show that our functions can serve as a replacement for the official implementation, we read and write to the same address book database with both our implementation and the reference implementation. Unsurprisingly, both implementations successfully process the serialized file and print out the expected information.

5.1 Conformance Tests

We have mechanically certified that our compiler meets its specification, but it is possible that our specification does not conform to Protocol Buffer's informal specification. In order to validate our specification, we tested our implementation against Protocol Buffer's official conformance test suite. These tests consist of a test runner and a client. The runner creates a test client process and sends it requests for each test case in the suite. The test client receives each request, decodes the payload, encodes the data back to the requested output format, and then sends the result back to the test runner. Clients may also respond with an error, as some test cases are intentionally malformed. The test runner accumulates all the test responses and eventually reports both the successful and failing tests.

Each request includes the message that the client should process, as well as the input and output format. Protocol Buffers support encoding not only to its binary format but also to JSON, so the test runner may ask the client to decode the data from JSON or encode the data to JSON, although clients are usually asked to use the binary format. We skipped all the JSON tests and Protocol Buffer version 2 tests, as we do not support those formats. Some tests require some features that we do not yet support, such as the oneof type. We modified and used the official Python test client as a proxy to process the requests and responses, and sent the payload to our OCaml client to perform the real tests. Our OCaml client reads the message from standard input, deserializes and reserializes the message, and then writes the result to standard output. Our OCaml client uses extracted Coq code in a similar manner to the aforementioned address book example.

Our implementation successfully passed 179 of 194 test cases. The fifteen failing test cases are not surprising: ten of these failing tests use oneof types, which we do not support. Another failing test uses version 3.5 of Protocol Buffers, which requires the unknown fields to be retained during parsing and included in the serialized output, another feature we currently do not support. Our implementation fails the final four tests because in base 128 varints format, the most significant bit can be set to 1 if the next byte is 0. For example, 0 can be encoded as 0, 80 0, and 80 80 0, but our current implementation only handles the canonical encoding. These results suggest that our specification is a correct formalization of Protocol Buffer's informal description.

6 Related Work

Formally Verified Parsers for Context-Free Languages In order to reduce the trusted code base of formally verified compilers, there have been a number of efforts to verify standalone parsers for *context-free languages*. These are not sufficient to build a Protocol Buffer compiler for two key reasons: firstly, Protocol Buffer's binary format is not context-free, due to its length-delimited wiretype. Secondly, these parsers would constitute only half a solution to deserialization, as *semantic actions* are also needed to build a message from a parse tree. Our deserializers handle both parsing the binary string and building an in-memory message from the parsed data. Barthwal and Norrish formally verified an SLR parser generator in HOL [5], showing every generated automaton is both sound and complete with respect to the grammar it was generated from. In contrast, Jourdan et. al formally verified a *validator* for LR(1) automata produced by an untrusted parser generator [18] to avoid formally verifying the generator itself. Koprowski and Binsztok [19] developed an operational semantics of partial expression grammars (PEGs) with semantic actions and proved that an interpreter was sound with respect to those semantics. In other related work, the authors of RockSalt [22], a formally verified Native Client sandbox-policy checker, developed a regular-expression DSL in order to specify and generate parsers from bitstrings into

various instruction sets. This DSL was equipped with a relational denotational semantics that the authors used to prove correctness of their parser. Subsequent work [27] extended this DSL to support bidirectional grammars in order to provide a uniform language for specifying and generating both decoders and encoders, proving a similar notion of consistency to what we present here.

Extensible Format Description Languages Alternative interface generators include XDR [26], ASN.1 [12], and Apache Avro [4], each of which provide their own domain-specific data description languages and compilers for their respective languages. There has been limited work on verified interface generators. One notable exception is the work of Collins et. al [9] to verify that the encoders and decoders generated by an untrusted ASN.1 compiler satisfy a round-trip property using Galois's SAW symbolic-analysis engine [11]. This round-trip property states that the encoder and decoder functions are inverses of each other. Notably, this specification uses functions, not relations, as their compiler uses the deterministic distinguished encoding strategy. Much of the work in that project was getting the ASN.1 [12] compiler to generate code amenable to automatic analysis by SAW, and there are no guarantees regarding other formats specified in ASN.1.

The Verdi [31] framework for building verified distributed systems originally used OCaml's (unverified) Marshal library to serialize data. In order to reduce the trusted code base, Verdi's authors are developing a verified serialization library for Coq called Cheerios [25]. The framework packages a type and its associated encoder and decoder functions into a typeclass, along with a proof that the deserializer is sound with respect to the encoder function. Typeclass resolution is used to automatically build encoders and decoders in a type-directed manner. Once again, this library strategy does not consider the possibility of noncanonical encodings.

Geest et. al [29] proposed a verified system to describe data formats in an embedded domain-specific language and to derive serializers and deserializers from these descriptions. They modeled the data schema as an universe, which is a collection of types in some structure, and defined a denotation function to map the universes to the actual types in Agda. This is similar to our definition of message descriptor as data schema, which dictates the actual types of the messages by denotation function. They also decomposed the transformation into two layers: the high-level data, such as the structured data containing natural numbers, is first converted to low-level data, the same structured data with corresponding words, and then the low-level data is serialized to strings. While our system also has a similar architecture and the intermediate representation has a smilar role as their low-level data, our first layer handles the non-determinism rather than type conversion. Unlike our system, their encoding and decoding process is canonical, thus cannot handle Protocol Buffer's flexibility.

Our implementation builds upon the Narcissus framework [10] for synthesizing serializers and deserializers from relational specifications. The framework includes a user-extensible library of format combinators and a set of tactics for deriving implementations of serializers and deserializers from these specifications. In Narcissus, serializers and deserializers are derived directly from arbitrary format specifications, and proofs of correctness are constructed alongside the functions. In contrast, our compiler takes a fixed data description language and produces serializers and deserializers that conform to the Protocol Buffer standard. By sacrificing flexibility, however, we are able to prove our compiler correct once and for all, although our proofs are mostly manual. Our statements of correctness use Narcissus' specifications for serializers and deserializers. The second layer of our compiler relies on Narcissus' definitions of common data structures and fixed-width word format in order to serialize IR values, although we had to extend the library with additional formats specific to Protocol Buffers, e.g., varints.

7 Future work

This section discusses the missing features and possible future improvements for our formalization. As noted in Section 1, the subset of Protocol Buffer version 3 we currently support is realistic enough to be used in most applications. However, there are a number of features that are needed to make our compiler fully functional.

- We do not support oneof types, which are essentially sum types. To support this feature, we have to extend the definitions of message descriptor and its denotation, and also the inference rules of the first layer to capture the behavior of oneof types. Since all the members of a oneof type have their own tags but share the same field, the main difficulty is probably that we need to "group" the tags and manipulate the message by this group, instead of a single tag. Unsurprisingly, we should not have to change the second layer at all.
- We do not support recursive and mutually recursive embedded messages. That is, the fields of a message cannot have the same type as the message itself. This feature is rarely used in practice: the official conformance suite does not include any tests for this feature.
- Our current work focuses on serializers and deserializers, so all the base types are denoted into the Coq types that are actually used in serialization. While we still provide a usable programming interface, it is not so pleasant for end-users. As one example, bool is denoted into the integer type, but users will probably expect to use booleans when manipulating a bool field. One potential solution is to have another denotation that maps the base types to more user-friendly Coq

types. We could then add another layer on top of the current architecture which relates the new denotation to the one used in this paper, and develop encoders and decoders for that layer.

- In Protocol Buffers version 3, unknown fields are discarded. However, in version 3.5, such fields are retained during parsing and included in the serialized output.
- The encoding of varints is also non-deterministic: the most significant bit can be set to 1 if the next byte is 0. To support this feature, we could extend the format to non-deterministically choose between these two cases, although this would complicate the proof of deserializer correctness.

8 Conclusion

We have presented a formally verified compiler for a realistic subset of the Protocol Buffer serialization format, which can generate provably correct serializers and deserializers for an arbitrary message descriptor. We can extract the resulting implementations to OCaml, and the soundness proofs can be used as part of verifying a larger system. We have demonstrated the usability of our system on an example drawn from Protocol Buffer's official repository, and shown that our implementation satisfies all the official conformance tests whose features we support.

Acknowledgments

We thank Robert Dickerson, and the anonymous reviewers for their valuable input. This research was supported through a faculty startup package from Purdue University.

References

[1] 2016. CVE-2016-5080. Available from MITRE, CVE-ID CVE-2016-5080.. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5080

[2] 2017. CVE-2017-9023. Available from MITRE, CVE-ID CVE-2017-9023.. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9023

[3] 2018. CVE-2018-11058. Available from MITRE, CVE-ID CVE-2018-11058.. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-11058

[4] Apache Software Foundation. 2016. Apache Avro 1.8.0 Documentation. http://avro.apache.org/docs/current/ [Accessed May 04, 2016].

[5] Aditi Barthwal and Michael Norrish. 2009. Verified, Executable Parsing. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 160–174.

[6] Tim Bray. 2017. *The JavaScript Object Notation (JSON) Data Interchange Format.* RFC 8259.

[7] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. 1997. Extensible Markup Language (XML). *World Wide Web Journal* 2, 4 (1997), 27–66.

[8] Adam Chlipala. 2013. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant.* MIT Press.

[9] Nathan Collins, Mark Tullsen, Aaron Tomb, and Lee Pike. 2017. Formal Verification of a Vehicle-to-Vehicle (V2V) Messaging System. In *Embedded Security in Cars (ESCARS).*

[10] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. [n. d.]. NARCISSUS: Deriving Correct-By-Construction Decoders and Encoders from Binary Formats. arXiv:1803.04870

[11] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. 2016. Constructing Semantic Models of Programs with the Software Analysis Workbench. In *Verified Software. Theories, Tools, and Experiments*, Sandrine Blazy and Marsha Chechik (Eds.). Springer International Publishing, Cham, 56–72.

[12] Olivier Dubuisson. 2001. *ASN. 1: communication between heterogeneous systems.* Morgan Kaufmann.

[13] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17).* ACM, New York, NY, USA, 328–343. https://doi.org/10.1145/3064176.3064183

[14] Python Software Foundation. 2018. Pickle - Python object serialization. https://docs.python.org/3/library/pickle.html

[15] Google Inc. [n. d.]. Protocol Buffers Encoding - Base 128 Varints. https://developers.google.com/protocol-buffers/docs/encoding#varints.

[16] Google Inc. [n. d.]. Protocol Buffers Frequently Asked Questions. https://developers.google.com/protocol-buffers/docs/faq.

[17] Google Inc. [n. d.]. Protocol Buffers repository. https://github.com/protocolbuffers/protobuf.

[18] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *Programming Languages and Systems*, Helmut Seidl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 397–416.

[19] Adam Koprowski and Henri Binsztok. 2011. TRX: A Formally Verified Parser Interpreter. *Logical Methods in Computer Science* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:18)2011

[20] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2018. The OCaml system release 4.07 Documentation and user's manual - Module Marshal. https://caml.inria.fr/pub/docs/manual-ocaml/libref/Marshal.html

[21] P. Mockapetris. 1987. *Domain names - implementation and specification.* RFC 1035.

[22] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: Better, Faster, Stronger SFI for the x86. *SIGPLAN Not.* 47, 6 (June 2012), 395–404. https://doi.org/10.1145/2345156.2254111

[23] Jon Postel. 1981. *Internet Protocol.* RFC 791.

[24] Jon Postel. 1981. *Transmission Control Protocol.* RFC 793.

[25] Keith Simmons. 2016. Cheerios. (2016). courses.cs.washington.edu/courses/cse599w/16sp/projects/cheerios.pdf.

[26] Raj Srinivasan. 1995. *XDR: External data representation standard.* Technical Report.

[27] Gang Tan and Greg Morrisett. 2018. Bidirectional Grammars for Machine-Code Decoding and Encoding. *Journal of Automated Reasoning* 60, 3 (01 Mar 2018), 257–277. https://doi.org/10.1007/s10817-017-9429-1

[28] The Coq Development Team. 2018. The Coq proof assistant reference manual, version 8.8.1. (2018).

[29] Marcell van Geest and Wouter Swierstra. 2017. Generic Packet Descriptions: Verified Parsing and Pretty Printing of Low-Level Data. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Type-Driven Development (TyDe 2017).* ACM, 30–40. https://doi.org/10.1145/3122975.3122979

[30] Kenton Varda. [n. d.]. Protocol Buffers. https://developers.google.com/protocol-buffers/.

[31] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15).* ACM, New York, NY, USA, 357–368. https://doi.org/10.1145/2737924.2737958