CBSOFT'26
CONGRESSO BRASILEIRO DE SOFTWARE

Foundations for Formally Verified Network Protocol Parsers in Lean 4: A Case Study with MQTT

Eduardo Sandalo Porto
eduardo@sandalo.dev
UFMG

Ana Cristina Vieira de Melo
acvm@ime.usp.br
USP

30th Brazilian Symposium on Programming Languages (SBLP 2026)

Motivation

- Network protocols are the backbone of computer communication
  - Many bugs in the past related to protocol parsers. (Heartbleed, EternalBlue, ...)
  - We need more and better tools for formally verified network stacks [1]
- A network protocol can be seen as a language:

Syntax (*our focus*)

- Wire layout and bit representations
- Structural & runtime dependencies
- Attack surface: invalid data, buffer overflows, out-of-bounds access, ...
- ***Goal***: sound and complete parsing & serialization

Semantics

- Meaning of messages and protocol state machines
- Delivery rules, message order, ...
- Widely researched in formal methods
- *Blind spot*: assumes packets are parsed

Network Protocol Parsers (Binary Formats)

- Input: a stream of bytes
- Output: structured, typed data (AST)
- **Challenges**:
  - Tricky bitwise transformations
  - Highly data-dependent formats

tag: u8	**var**: (**tag** == 1 ? u16 : str)	**payload**: {**x** : T(**var**) | pred(**x**)}

- How can we:
  - 1) Formally specify data variations?
  - 2) Guarantee properties written in the specification?
  - 3) Write verified parsers?

The Approach

Lean 4 as a unified platform for *specification* & *implementation* & *verification*

***Goals**:*

1. Formally **specify** syntactic constraints in the protocol
2. **Implement** parsers for the protocol
3. Prove the parsers are **correct**

***Lean 4**:*

- Dependent types are expressive enough to encode data-dependencies
- Unified language for specification, implementation, proofs
- Metaprogramming with macros enables parser and proof synthesis

- Current scope: Work-in-progress foundational case study on *MQTT 5.0*
- Future: Native eDSL generating verified parsers via macros

Our Contributions

C1: Modeling (*Correct-by-construction*)

- **Dependent types:** Encode protocol invariants directly into types
- **Compile-time safety:** Invalid packets are rejected by the type checker

C2: Parser Library (*Proof-carrying combinators)*

- **Monadic state**: Layered parser primitives and combinators over byte sequences
- **Proof-carrying:** Primitives and combinators handling dynamic layouts and termination

C3: Verification (*Soundness & completeness)*

- **Compositionality:** Proofs for primitives and combinators scale the protocol hierarchy
- **Core theorems:** *Roundtrip* (completeness) and *reconstruction* (soundness)

MQTT 5.0

- We demonstrate our strategy through a *case study* with MQTT 5.0
  - Pub/sub client/server protocol, ubiquitous Internet of Things protocol
  - Official OASIS standard (informal specification) [2]
  - **Verification sweet-spot**: compact enough for formalization, but not a toy example
    - 15 packet types, dynamic headers and payloads, 251 *conformance statements*
- The **Packet Format**
  - **Fixed Header ↦ Variable Header ↦ Payload**

Fixed Header

Byte	Bits 7–4	Bits 3–0
1	Packet Type	Type-Specific Flags
2–5	Remaining Length	

Variable Header

- 15 variations depending on the packet type
- Built using 7 protocol primitives
- May contain list of *properties*

Payload

- May vary or be optional within a packet type
- Future work (we currently focus on headers)

C1: Correct-by-Construction Modeling

- Use **dependent types** to preclude invalid packets
- Library of protocol **primitives** and **combinators**

Primitives

- Unsigned integers (`UInt8`, `UInt16`, `UInt32`)
- Variable Byte Integers (`VarInt`)
- Length-prefixed strings, binary data (`Str`, `BinaryData`)

```
abbrev VarInt := Fin (2^28)
abbrev Str := WithByteSize String UInt16
```

Combinators

- Length-prefixed generic (`WithByteSize`)
- Length-prefixed list (`SizedList`)
- Constant value (`ConstVal`)
- Optional value (`OptType`)
- Predicate constrained (`PredType`)

```
structure WithByteSize (α lenTyp)
    [s : GetByteSize α] [Coe lenTyp Nat] where
  val : α
  len : { n : lenTyp // s.byteSize val = n }

abbrev OptType (α : Type) (b : Bool) : Type :=
  cond b α Unit

abbrev PredType (α : Type) (gen : α → List Condition) : Type :=
  { val : α // ∀ c ∈ gen val, c.prop }
```

C1: Correct-by-Construction Modeling

Fixed Header

Flag shape determined by packet type

```
def PktFlags : PktKind → Type
  | .connect     => ConstVal (BitVec 4) (0b0000)
  | .connack     => ConstVal (BitVec 4) (0b0000)
  | .publish     => PublishFlags
  | ...

structure FixedHeader where
  kind  : PktKind
  flags : PktFlags kind
  remaining_len : VarInt
```

Variable Header

Layout depends on packet type and flags

```
def VarHeader.getType : (k : PktKind) → PktFlags k → Type
  | .connect, _     => Var_Connect
  | .publish, f     => Var_Publish f.val.qos
  | ...

structure Var_Publish (qos : QoSBits) where
  topic_name : TopicName
  packet_id  : OptType UInt16 (qos.val > 0)
  props      : Properties
```

Global Properties: Right-to-left dependencies can be modeled globally

```
structure RawPacket where
  fh : FixedHeader
  vh : VarHeader fh
  pl : Payload fh vh
```

```
abbrev Packet :=
  PredType RawPacket fun p =>
    [ensure! p.vh.byteSize + p.pl.byteSize =
             p.fh.remaining_len.val]
```

C2: Parser Combinator Library

- **Parsing engine**: `abbrev Parser (α : Type) := StateT (List UInt8) Option α`
  - *State Transformer* + *Option* for monadic byte-stream processing with short-circuiting
  - *List* over *ByteArray* for easier proofs; we plan to prove *List* ↔ *ByteArray* parser equivalence
- **Proof-carrying**: `def Parser.bytesWithProof (n : Nat) : Parser { l : List UInt8 // l.length = n }`
  - Carries compile-time properties about the parsed values

- **Codecs**
  - Serialization and parsing combinators

```
class Codec (α : Type) where
  parser : Parser α
  serialize : α → List UInt8
```

- **Byte-sized lists**
  - Size correctness and termination

```
class ChunkItem (α : Type) [GetByteSize α] [c : Codec α] where
  h_pos : ∀ (a : α), 0 < GetByteSize.byteSize a
  h_consumed : ∀ {a : α} {input rest : List UInt8},
    c.parser.run input = some (a, rest) →
    input.length = GetByteSize.byteSize a + rest.length
```

C2: Parser Combinator Library

Fixed Header

Bit extraction & dynamic flag decoding

```
def FixedHeader.parser : Parser FixedHeader := do
  let byte ← UInt8.parser
  let bv := byte.toBitVec
  let kind ← PktKind.decode? (bv.extractLsb 7 4)
  let flags ← PktFlags.decode? kind (bv.extractLsb 3 0)
  let remaining_len ← VarInt.parser
  return { kind, flags, remaining_len }
```

Takeaway:

Complexity is concentrated in the primitives and combinators

Variable Header

Dispatch to the appropriate variation

```
def VarHeader.parserValue :
    (k : PktKind) → (f : PktFlags k) →
    Parser (VarHeader.getType k f)
  | .connect, _      => Var_Connect.parser
  | .publish, f      => Var_Publish.parser f.val.qos
  | ...

def Var_Publish.parser (qos : QoSBits) :
    Parser (Var_Publish qos) := do
  let topic_name ← Str.parser
  let packet_id ← OptType.parser (qos.val > 0)
  let props ← Properties.parser
  return { topic_name, packet_id, props }
```

C3: Formal Verification

Roundtrip (completeness)

$$par(ser(d) \oplus r) = some(d, r)$$

"Parsing serialized data perfectly recovers the original value and leaves the remainder stream untouched".

Reconstruction (soundness)

$$par(s) = some(d, r) \longrightarrow s = ser(d) \oplus r$$

"Any successfully parsed byte stream corresponds strictly to the serialization of the parsed data".

- Proofs guarantee consistency between serializer/parser
  - Protocol conformance validated against the specification - see evaluation

Round-trip and reconstruction

```
class LawfulCodec (α : Type) [c : Codec α] where
  roundtrip : ∀ (a : α) {rest : List UInt8},
    c.parser.run (c.serialize a ++ rest) = some (a, rest)
  reconstruct : ∀ {a : α} {input rest : List UInt8},
    c.parser.run input = some (a, rest) →
    input = c.serialize a ++ rest
```

Correctness of byte sizes

```
class LawfulByteSize (α : Type)
    [c : Mqtt.Codec α] [b : GetByteSize α] where
  serialize_len : ∀ (a : α),
    (c.serialize a).length = b.byteSize a
```

C3: Formal Verification (proof engineering)

Primitives

- Unsigned integers: solved by `bv_decide`
- VBIs: finite unrolling across 4 bytes
  - Custom tactics: `natify_uint8;` `crush_lits;`
  - Then, automated via `omega`
  - **Notably**, uncovered a bug in the parser when missing canonicity check
- Strings: custom proofs for List ↔ Arrays
  - Tricky cases: proving correct sizes

Composition

- Combinator proofs usually by construction
- Parametric types call subproofs via type class
- **Roundtrip** proofs**:**
  - Rewriting through `simp`
- **Reconstruction** proofs:
  - Harder to prove
  - Monadic bind unrolling via custom `step_parser` `finish_parser`
- Headers:
  - **Mechanically call** the primitive and combinator proofs

Takeaway: Concentrating complexity in the bottom layers allows proofs to scale easily

Evaluation

Protocol Conformance

- 251 total **conformance statements** in the official OASIS specification
- We classified 68 statements as **syntax**
  - Wire layouts, packet encoding
- We cover 34 statements
  - Exceptions are the unimplemented payloads and a few tricky cases
- Translating informal specifications to formal takes effort
  - Could we use LLMs?

Proof Effort and Scaling

- ~1:1 proof-to-code ratio
- Proof burden concentrated at the leaves
  - Some primitives took a lot of effort, but composition is effortless
- The composition strategy is easily mechanized
  - **Enables the future automated synthesis through an embedded domain-specific language (eDSL)**

Conclusion & Further Work

Core Contributions

- **C1: Correct-by-Construction Modelling**: dependent types eliminate invalid states at compile time
- **C2: Proof-Carrying Combinators**: Monadic combinators with functional and termination guarantees
- **C3: Formal Verification**: Scalable guarantees of roundtrip and reconstruction

Future directions

- Expand coverage to the full specification
- Verify equivalence with a faster, ByteArray-based parser
- Parser and proof synthesis through an eDSL in Lean
- Investigate ways to validate specification formalization
- **Towards end-to-end protocol verification in Lean**

Acknowledgements

Thank you!
Questions?

- The paper and slides are available at https://sandalo.dev
- This project was supported by CNPq

CNPq
Conselho Nacional de Desenvolvimento Científico e Tecnológico

References

[1] Basin, David, Nate Foster, Kenneth L. McMillan, et al. “It Takes a Village: Bridging the Gaps between Current and Formal Specifications for Protocols.” Commun. ACM 68, no. 8 (2025): 50–61. https://doi.org/10.1145/3706572.
[2] Andrew Banks, Ed Hurlburt, and Rahul Gupta. 2019. MQTT Version 5.0. Technical Report. OASIS Standard. https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqttv5.0-os.html