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

Eduardo Sandalo Porto*
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte, Brazil
eduardo@sandalo.dev

Ana Cristina Vieira de Melo
Departamento de Ciência da Computação
Universidade de São Paulo (USP)
São Paulo, Brazil
acvm@ime.usp.br

Abstract

A significant portion of the Internet's critical infrastructure relies on the precise implementation of network protocols, yet bridging the gap between formal verification and applied network engineering remains a challenge. In this paper, we present a case study in implementing and formally verifying a parser for the MQTT 5.0 protocol using Lean 4. Our approach includes three components: (I) correct-by-construction modeling of the packet format, leveraging dependent types to preclude invalid packets and encode data-dependent structural variations; (II) a layered, proof-carrying parser library utilizing state monads to process inputs whose exact layout depends on runtime parsed values; and (III) formal verification of parser completeness (roundtrip) and soundness (reconstruction) with respect to a serializer. Ultimately, this work serves as a foundational step toward a generalized, embedded domain-specific language for generating verified network protocol parsers.

Keywords

Formal Verification, Parser Combinators, Dependent Types, Network Protocols, Domain-Specific Languages, Lean 4

1 Introduction

Formal methods (FM), including formal specification and verification, have seen rising adoption as critical services increasingly rely on complex computer networks. One area directly benefiting from FM is network protocol specification. As online services expand into novel fields [3], researchers must prioritize the correctness of network-operating programs.

The Message Queuing Telemetry Transport (MQTT) protocol [2], ubiquitous in IoT for its lightweight footprint, is an ideal non-trivial case study to explore this intersection. Its client/server publish-subscribe architecture is simple enough for verification experiments, yet sufficiently complex to avoid being a mere toy example.

Tools such as Lean 4 [10] serve as both interactive theorem provers and general-purpose programming languages. This allows developers to formalize specifications, prove properties, and write programs within a single environment. Its metaprogramming capabilities enable complex compile-time generation and automated proofs, uniting FM with practical software engineering.

Our work contributes to bridging the gap between FM and communication protocol research using Lean 4. We formalize part of the official MQTT v5.0 specification [2], implement a serializer and a parser, and prove that the parser is sound and complete with respect to the serializer. To accomplish this, we propose three components:

C1. Correct-by-construction modeling of MQTT packets,
C2. Implementation of a serialization and parsing library,
C3. Formal verification of parser *roundtrip* and *reconstruction*.

We tackle **C1** using dependent types to reject the construction of invalid packets and to encode the structural variations in data based on values known only at runtime. **C2** consists of a layered parser combinator library based on the state transformer and option monads, utilizing a set of primitives to compositionally build serializers and parsers. Finally, **C3** leverages the compositional nature of our library to prove two core properties for each data structure: *roundtrip* (completeness), ensuring that parsing a serialized structure successfully recovers the original data, and *reconstruction* (soundness), ensuring that any successfully parsed stream corresponds exactly to the serialization of the resulting data structure.

In this preliminary phase of the project, we have implemented and verified most of components **C1**, **C2**, and **C3** for the fixed and variable headers of MQTT control packets, leaving payloads as future work. The central contribution of this work is demonstrating how the structural compositionality of both the parser primitives and their corresponding proofs allows verification to scale up the protocol hierarchy. This provides a clear path toward an embedded domain-specific language (eDSL) for generating verified network protocol parsers natively in Lean 4.

2 Background: MQTT

Assuming network packets are correctly parsed, protocol semantics define the meaning of messages and state transitions. Our focus, however, is strictly on MQTT's syntax, which is the packet format defining which bytes represent each logical component of the protocol, and how they structurally depend on one another. As defined by the official specification [2], MQTT clients and servers share a packet format composed of three parts: a *fixed header*, a *variable header*, and a *payload*. Each part is built from fundamental building blocks, which we call **primitives**. Notably, the structural layout of the variable header and payload depends heavily on which of the 15 control packet types (e.g., `CONNECT`, `PUBLISH`, `SUBSCRIBE`, ...) is being transmitted. Thus, the shape of a packet is affected by the values present in the fixed header.

2.1 Primitives

Packets are constructed from a standardized set of fixed bit sequences and base data primitives. These include standard 8-bit, 16-bit, and 32-bit unsigned integers, alongside variable-byte integers (VBIs) that compress values into one to four bytes to maintain a light footprint. Dynamically sized data, such as topic names or authentication payloads, are encoded using length-prefixed UTF-8

*Work partially conducted while the author was at Universidade de São Paulo (USP).

strings, length-prefixed binary data, and string pairs. Because all higher-level packet components are composed entirely of these base types, formally specifying their exact bounds and wire layouts is the foundation of our correct-by-construction modeling.

2.2 Fixed Header

Fixed headers share a common, rigid structure (Table 1) consisting of 4 bits defining the packet type, 4 bits for type-specific flags, and a VBI encoding the remaining length of the packet. Most packet types have constant flag values. A notable exception is the PUBLISH packet, which we use as a running example in future sections. The PUBLISH fixed header contains 1-bit flags DUP and RETAIN, and the 2-bit QoS (Quality of Service) flag, which dynamically alters the required layout of the subsequent variable header.

Table 1: MQTT 5.0 fixed header structure.

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

2.3 Variable Headers

The structure of a variable header is completely dependent on the packet type parsed from the fixed header, with a total of 15 variations. Each variation is constructed from the primitives, with some containing a list of *properties*. This list is prefixed by its total byte size as a VBI, followed by individual properties consisting of an identifier and a primitive whose type depends on the identifier.

Because variable headers rely on data parsed at runtime, their verification requires context-sensitive handling. For instance, a PUBLISH variable header contains a topic name string, a property list, and a packet identifier. Crucially, the packet identifier is only present if `QoS > 0`.

3 C1: Correct-by-construction modeling

To model MQTT packets correctly-by-construction, we leverage Lean's dependent type system. By encoding invariants directly into type definitions, we make it impossible to instantiate invalid packet structures, moving the burden of structural correctness to the type checker.

3.1 Primitives

While standard types such as `UInt8` and `String` are provided by Lean, MQTT requires custom primitives. For instance, the VBI (`VarInt`) is strictly limited to 4 bytes. Since it must be a natural number less than 2^{28}, we define it as `abbrev VarInt := Fin 268435456`, ensuring no out-of-bounds value can exist.

Similarly, length-prefixed structures appear throughout the protocol (e.g., strings, binary data, property lists). To systematically reason about the physical wire footprint of any parsed data structure, we introduce a `GetByteSize` type class providing a computable `byteSize` function. We use it to abstract length-prefixed structures by defining a dependent record `WithByteSize α lenTyp`. This record pairs a value of type α with a length of type `lenTyp`, alongside a proof that the length matches the exact byte footprint of the serialized value. This prevents malformed length prefixes.

We also define combinator primitives to encode structural invariants not explicitly written in the specification. We define `OptType` for conditionally present fields, `SizedList` for complex length-prefixed structures, and `ConstVal` to enforce that a parsed value exactly matches a specification-defined constant:

```
abbrev OptType (α : Type) (b : Bool) : Type :=
  cond b α Unit
abbrev SizedList (α lenTyp : Type) [GetByteSize α] [...] :=
  WithByteSize (List α) lenTyp
abbrev ConstVal (α : Type) (expected : α) : Type :=
  { val : α // val = expected }
```

Finally, to capture highly specific conformance rules, we introduce a generalized `PredType`. This restricts a type to values that satisfy a dynamically generated list of predicates. We define these predicates as `Conditions`, which bundle a computable decision procedure with a textual description to facilitate debugging:

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

Additionally, to automate the boilerplate between protocol enumerations and Lean inductive types, we implemented a macro `enum_with_codec`. It pairs variations of a simple inductive type to a base type, automatically generating `encode` and `decode?` functions.

3.2 Fixed Header

To model the fixed header, we must guarantee that the 4-bit flag field strictly adheres to the rules dictated by the 4-bit packet type. The packet type is encoded as the inductive enumeration `PktKind`, pairing each variation with a `BitVec 4` using our `enum_with_codec` macro. For 14 out of the 15 control packets, the MQTT specification requires the flags to be fixed, reserved constants (e.g., `0b0000` for `CONNECT`, `0b0010` for `PUBREL`, ...). To model flags, we map each `PktKind` to a dependent type family `PktFlags : PktKind → Type`. For types with constant flags, this evaluates to a `ConstVal` bit vector. For PUBLISH, it evaluates to a structure containing the DUP, QoS, and RETAIN fields:

```
enum_with_codec PktKind : BitVec 4 where
  | connect => 1 | connack => 2 | publish => 3 | ...
def PktFlags : PktKind → Type
  | .connect => ConstVal (BitVec 4) (0b0000)
  | .publish => PublishFlags | ...

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

The `remaining_len` field introduces a right-to-left dependency, as its correctness relies on the byte footprint of the components that follow it. To enforce this global property, we assemble the components into a `RawPacket` type, then wrap it with a `PredType` to form the final, verified `Packet` type. We construct the constraint using our `ensure!` macro, which automatically captures the Lean expression as a string literal to provide readable debugging context:

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

3.3 Variable Headers

We extend this dependency chain to the variable headers. Because the layout of a variable header is entirely dictated by the fixed header, we define a type-level function `VarHeader.getType` that maps a packet's `kind` and `flags` to its specific Lean structure (e.g., `Var_Connect`, `Var_Publish`, ...).

This approach forces context-sensitive validation into the type signatures of the specific header structures. Returning to the `PUBLISH` example, the variable header must account for the `QoS` flag parsed in the fixed header to determine if a packet identifier is present. Using the `OptType` primitive defined earlier, we model this as:

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

We model the list of properties as `SizedList Property VarInt`, and a property itself is a dependent pair of its identifier `id` and an application of `Property.valType id`, which dependently chooses a type based on the `id`.

```
def Property.getKind : VarInt → Property.Kind
  |  1 | 23 | ... => .u8  | 19 | 34 | ... => .u16  | ...
abbrev Property.typeOfKind : Property.Kind → Type
  | .u8 => UInt8  | .u16 => UInt16  | ...
abbrev Property.valType (id : VarInt) : Type :=
  Property.typeOfKind (getKind id)
```

4 C2: Parser Library

To transform our packets to and from byte streams, we implement a layered serialization and parser combinator library. The core engine of this library is the `Parser` monad, which is a simple state transformer over lists of bytes returning optional parsed values:

```
abbrev Parser (α : Type) := StateT (List UInt8) Option α
```

We chose to parse over `Lists` instead of `ByteArrays` to greatly simplify the inductive proofs. While this trades raw execution speed for proof tractability, it allows the library to serve as a formally verified executable specification. As future work, we plan to extend our parsers to feature custom error messages and array parsing. Our strategy is to implement performant array parsers and prove their semantic equivalence to the list counterparts, leveraging the proofs already established in Section 5.

4.1 Primitives

While basic primitives such as the unsigned integers are parsed using straightforward monadic sequences, length-prefixed structures pose a specific challenge in proving the correctness of the size, as well as parser termination. This is simpler to deal with on length-prefixed strings and binary data: our library supplies a proof-carrying `bytesWithProof` function, returning a dependent sum containing the parsed value and a proof of its size.

Lists of properties are harder to deal with, as they logically contain multiple `Property` instances but are prefixed by the total byte size. We isolate a list's full sequence into a bounded sub-stream using `bytesWithProof`, then parse sub-items recursively. However, we need to guarantee that the sum of sub-items is correct and this process terminates.

We achieve this by introducing the `ChunkItem` α type class for parseable list elements, required by the `SizedList` type. The class bundles two crucial proofs: `h_pos`, which guarantees that an element's byte size is strictly greater than zero, and `h_consumed`, which proves that successfully parsing the element advances the input stream by exactly that size:

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

Using these invariants, we implement `ChunkItem.parseChunkLoop`, recursively extracting items from the isolated byte chunk until it is empty. These invariants guarantee that the sum of sub-item sizes is correct, and they are also fed to the termination checker to ensure totality. Similarly, parsing `VarInts` requires recursive functions that accumulate proofs for correctness-by-construction and termination.

Standard primitives implement a `Codec` type class to provide base serializers and parsers. To parse our combinator primitives, such as `ConstVal` and `PredType`, we extract the underlying value and evaluate its required predicates using Lean's `if` expressions. Because these conditions are backed by computable decision procedures, a successful evaluation automatically produces the proof required to instantiate the dependent type.

The takeaway is that we concentrate complexity in serializing and parsing primitives, then compose these functions to easily represent higher abstractions. Thus, the actual packet parser consists of a sequence of simple calls to parser primitives.

4.2 Fixed Header

The advantage of concentrating complexity within the primitives becomes apparent when parsing headers. For the fixed header, the parser must extract the first byte, split it into two 4-bit vectors, and instantiate the dependent types defined in Section 3.2.

```
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 size ← VarInt.parser
  return { kind, flags, size }
```

We treat `PktKind` and `PktFlags` as subcomponents, with decoders implemented by macros. Crucially, `PktFlags.decode?` explicitly depends on the runtime value of `kind`. Because we operate within the `Parser` monad, any invalid bit combination silently short-circuits the execution, safely propagating failure.

4.3 Variable Header

The library dynamically dispatches to the correct variable header parser based on the packet kind. We implement a `parserValue` function that pattern matches the fixed header's type and flags, then selects the corresponding parser whose return type is defined by `VarHeader.getType`. For our running example, the `PUBLISH` variable header parser receives the `QoS` flags parsed from the fixed header:

```
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 }
```

Note that Lean's type inference guarantees that `packet_id` is of the correct type, and that no boolean expression other than `qos.val > 0` is allowed for the optional parsing condition.

5 C3: Formal Verification

For each data structure T and byte stream BS, we define a serializer $ser : T \to BS$ and a parser $par : BS \to Option(T \times BS)$. To guarantee correctness, we formalize and prove two core properties for every component, equivalent to completeness and soundness:

Theorem 5.1 (Roundtrip). *For any value $d : T$ and remainder stream $r : BS$, parsing the serialization of d concatenated with r perfectly recovers d and leaves r untouched:*

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

Theorem 5.2 (Reconstruction). *For any value $d : T$ and byte streams $s, r : BS$, if parsing s succeeds and yields d with remainder r, then s must be exactly the serialization of d concatenated with r:*

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

Alternatively, these theorems establish that the parser is a left-inverse of the serializer, and the serializer is a right-inverse of the parser. Consequently, the proven correctness of the parser is relative to the serializer, rather than an absolute guarantee of conformance to the MQTT 5.0 specification [2]. This is mitigated by our correct-by-construction modeling: trivial parser implementations are precluded due to strict proof obligations. Still, if the serializer's implementation deviates from the standard, the parser proofs alone provide limited protocol-level guarantees. To bridge this gap between internal consistency and external protocol correctness, we must ensure that the data structures and serialization logic accurately reflect the MQTT standard. We measure and quantify this process through the specification's official *conformance statements*, as evaluated in Section 6.

5.1 Primitives

The complexity of the proofs is constrained to the primitives and monadic combinators. For each primitive, we evaluate expressions up to parsing, then perform theory-specific reasoning. To facilitate proof composition, we developed custom *tactics* (Lean metaprograms) and proved core theorems for monadic parsing.

Unsigned integer proofs are automated by the `bv_decide` tactic. String proofs require reasoning about UTF-8 alongside `ByteArray` and `List` transformations, which are only partially supported by the Lean standard library. Length-prefixed primitives introduce the challenge of ensuring the size proofs carried by the parsers remain consistent throughout execution.

VBIs require extensive proofs due to recursive modulo arithmetic. Because the VBI size is strictly bounded to 4 bytes, we verify its correctness via finite function unrolling rather than induction, which would otherwise struggle with the bounded bit-shift semantics. To handle heavy type coercions from bit-vectors and bounded numbers to the naturals, we developed custom tactics `natify_uint8` and `crush_lits` to map values to `Nat`. Specifically, `natify_uint8` traverses a proof state, rewriting values of type `UInt8` to `Nat` while adding restriction lemmas, such as an exclusive upper bound of 256. Similarly, `crush_lits` traverses expressions in the goal state that result in `Nat` (usually, transformations over `Fin`), forcing them into a closed literal, if possible. Both tactics work together to produce simpler proof states with linear modulo arithmetic. These states are then solved using the `omega` tactic, based on the *omega* test for Presburger arithmetic [11]. Notably, formalization uncovered a bug in our original VBI parser: failing to check VBI *canonicity* invalidates reconstruction.

Finally, parametric types such as `OptType` and `SizedList` depend on the proofs of their parameters. While optional types are trivial, sized lists are not, requiring the `ChunkItem` type class to guarantee totality and exact byte consumption. Ultimately, most valid primitives instantiate the `LawfulCodec` type class, which groups the roundtrip and reconstruction properties:

```
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
```

Furthermore, to satisfy the correct-by-construction size bounds discussed in Section 3.2, primitives and packet components must prove that their serialized length exactly matches their computed footprint. We guarantee this through the `LawfulByteSize` type class:

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

Composing base theorems uses two strategies: backward proofs for roundtrips (rewriting goals to match assumptions) and forward proofs for reconstruction (combining hypotheses to satisfy goals). Roundtrip proofs are straightforward simplifications utilizing primitive theorems. Reconstruction, however, requires breaking a successful monadic `bind` hypothesis into intermediate states. To automate this, we proved core `Parser` theorems (`bind_run_success` and `pure_run_success`) and developed custom tactic macros. The `step_parser` tactic automatically isolates the next parser in the sequence, extracts its success hypothesis, and extracts the intermediate input state. Finally, `finish_parser` completes the monadic unrolling by extracting the final state equivalence, eliminating the manual tracking of byte streams across the monadic chain.

5.2 Headers

As seen in Section 4.2, `FixedHeader.parser` parses a byte and a VBI, then performs bit operations to extract the type and flags. We directly use the `LawfulCodec` primitive proofs, but dealing with bit vectors requires more care. We prove roundtrip and reconstruction for `PktKind` and `PktFlags` using `bv_decide`, then compose all primitives to finalize the fixed header.

The verification of the variable headers demonstrates the primary advantage of our compositional approach. Because all logical complexity (bounded recursion, monadic state unrolling, bounds checking, etc) is isolated and proven within the base primitives, verifying the variable headers requires almost no custom logic.

Similar to our parser dispatcher, we define `roundtrip_value` and `reconstruct_value` functions that pattern match the fixed header and dispatch the proofs to the specific header variations, collapsing the proof into simple structural rewriting. Roundtrip proofs are resolved by consecutive simplifier (`simp`) calls utilizing the primitive theorems. Reconstruction proofs apply our `step_parser` tactic to deconstruct the monadic binds, sequentially substituting the reconstructed primitives into the proof state. For instance, the `PUBLISH` roundtrip and reconstruction proofs simply follow this strategy, applying the primitive proofs for `Str`, `OptType`, and `Properties`. This demonstrates that by isolating invariant proofs in the primitives, our verification architecture scales cleanly up the protocol hierarchy without proof state explosion. While this final process is still manual, we plan on fully automating the proofs for each composite type built using our primitives.

6 Evaluation

We evaluate this work on proof scalability and protocol conformance. Structurally, our architecture prevents proof state explosion. By isolating theory-specific reasoning within base primitives and custom tactics, verifying composite packets simplifies to straightforward rewriting sequences. Our preliminary results show a roughly 1:1 ratio of non-proof to proof lines, a baseline we aim to reduce in further work via complete automation of proofs regarding composite structures built with our primitives.

We measure correctness against the MQTT 5.0 specification [2] using its 251 mandatory *conformance statements*. We manually classified each statement as *syntax*-related, *semantic*-related, or *other*.[1] For this preliminary phase, we defined "syntax-related" as strictly packet-local constraints independent of external state (e.g., client/server roles or previous packets), though future work may expand this scope. Of the 68 syntax-related statements identified, our implementation formally satisfies 34, covering fixed and variable headers. The remaining statements largely include unimplemented payloads or complex edge cases postponed to future work. This baseline confirms our primitives capture real-world invariants without breaking the verification pipeline, establishing a viable trajectory toward full specification conformance.

Much of the protocol's behavior resides solely in the specification's natural language prose. Translating full English specifications into formal models remains a challenge beyond our current scope, though we view large language model (LLM) pipelines as a promising avenue for future automated formalization and validation.

[1] Our classification is available in the provided artifact.

7 Related Work

A substantial body of work exists on the verification of MQTT semantics [1, 8, 9, 15, 17]. To the best of our knowledge, there is no existing work specifically on verified parsing of MQTT's syntax. The generation of bidirectional parsers is also a widely researched field [5, 13, 14].

Considering verified binary format parsers, a prominent effort is Narcissus [6], a framework developed in the Rocq Prover (formerly Coq) to automatically derive correct-by-construction encoders and decoders from relational format specifications. Similarly, the Everparse project [12, 16] uses F* to generate formally verified binary parsers in C from high-level specifications, and Vest [4] utilizes the Verus toolkit to accomplish a similar goal in Rust.

Our work diverges from EverParse and Vest by eliminating the division between specification and implementation languages. By unifying the packet model, parser, and proofs directly within Lean 4 without code extraction, we maintain a tight coupling between syntax and semantics, enabling end-to-end protocol verification. Furthermore, our work differs from Narcissus in its target domain: while the original Narcissus paper focused on the relatively static layout of lower-level TCP/IP stack protocols, our work tackles the dynamic, application-layer protocol MQTT. While currently at an earlier stage of maturity than these established frameworks, our approach demonstrates a viable foundation for native protocol formalization in Lean 4.

8 Conclusion and Future Work

In this paper, we presented the foundational architecture of a formally verified parser for the MQTT 5.0 protocol in Lean 4 to help bring FM and network protocols closer together. By combining correct-by-construction modeling with a monadic parser combinator library, we demonstrated how runtime-dependent structural variations can be safely encoded using dependent types. Additionally, our compositional proofs of completeness and soundness show that verification can scale up the protocol hierarchy without proof state explosion. In practice, this verified parser can serve as an executable specification to validate a highly optimized implementation in another language through differential and property-based testing, following a *verification-guided development* approach [7].

While our current implementation covers most of the fixed and variable headers of MQTT control packets, future work will extend this library to payloads, custom error handling, and array parsing. Ultimately, the structural compositionality of our parser primitives serves as the foundation for an embedded domain-specific language (eDSL). This eDSL will utilize Lean 4's metaprogramming capabilities to automatically generate correct-by-construction network parsers directly from high-level specifications, bridging the gap between formal methods and applied network engineering.

Artifact Availability

The Lean 4 source code, proofs, and build instructions for the parser presented in this paper are publicly available on Zenodo with the Apache 2.0 license at https://doi.org/10.5281/zenodo.22181982.

Acknowledgements

This project was supported by the CNPq grant #134250/2025-7.

References

[1] Sabina Akhtar and Ehtesham Zahoor. 2021. Formal Specification and Verification of MQTT Protocol in PlusCal-2. *Wirel. Pers. Commun.* 119, 2 (July 2021), 1589–1606. doi:10.1007/s11277-021-08296-4

[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/mqtt-v5.0-os.html

[3] David Basin, Nate Foster, Kenneth L. McMillan, Kedar S. Namjoshi, Cristina Nita-Rotaru, Jonathan M. Smith, Pamela Zave, and Lenore D. Zuck. 2025. It Takes a Village: Bridging the Gaps between Current and Formal Specifications for Protocols. *Commun. ACM* 68, 8 (July 2025), 50–61. doi:10.1145/3706572

[4] Yi Cai, Pratap Singh, Zhengyao Lin, Jay Bosamiya, Joshua Gancher, Milijana Surbatovich, and Bryan Parno. 2025. Vest: Verified, Secure, High-Performance Parsing and Serialization for Rust. In *Proceedings of the USENIX Security Symposium.*

[5] Nils Anders Danielsson. 2010. Total parser combinators. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming (ICFP '10).* Association for Computing Machinery, New York, NY, USA, 285–296. doi:10.1145/1863543.1863585

[6] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. 2019. Narcissus: correct-by-construction derivation of decoders and encoders from binary formats. *Proc. ACM Program. Lang.* 3, ICFP (July 2019), 82:1–82:29. doi:10.1145/3341686

[7] Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, John Kastner, Anwar Mamat, Matt McCutchen, Neha Rungta, Bhakti Shah, Emina Torlak, and Andrew Wells. 2024. How We Built Cedar: A Verification-Guided Approach. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE 2024).* Association for Computing Machinery, New York, NY, USA, 351–357. doi:10.1145/3663529.3663854

[8] Amina Jandoubi, M. Bennani, Olfa Mosbahi, and Abdelaziz El Fazziki. 2024. Analyzing MQTT Attack Scenarios: A Systematic Formalization and TLC Model Checker Simulation:. In *Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering.* SCITEPRESS - Science and Technology Publications, Angers, France, 370–378. doi:10.5220/0012625600003687

[9] Wei Lin, Sini Chen, and Huibiao Zhu. 2025. Formal verification and security analysis of MQTT-SN. *International Journal on Software Tools for Technology Transfer* 27, 1 (Feb. 2025), 5–19. doi:10.1007/s10009-025-00793-2

[10] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, André Platzer and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 625–635. doi:10.1007/978-3-030-79876-5_37

[11] William Pugh. 1991. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (Albuquerque, New Mexico, USA) *(Supercomputing '91).* Association for Computing Machinery, New York, NY, USA, 4–13. doi:10.1145/125826.125848

[12] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. Everparse: verified secure zero-copy parsers for authenticated message formats. In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19).* USENIX Association, USA, 1465–1482.

[13] Tahina Ramananandro, Gabriel Ebner, Guido Martínez, and Nikhil Swamy. 2025. Secure Parsing and Serializing with Separation Logic Applied to CBOR, CDDL, and COSE. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25).* Association for Computing Machinery, New York, NY, USA, 2744–2758. doi:10.1145/3719027.3765120

[14] Tillmann Rendel and Klaus Ostermann. 2010. Invertible syntax descriptions: unifying parsing and pretty printing. In *Proceedings of the third ACM Haskell symposium on Haskell (Haskell '10).* Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/1863523.1863525

[15] Alejandro Rodríguez, Lars Michael Kristensen, and Adrian Rutle. 2019. Formal Modelling and Incremental Verification of the MQTT IoT Protocol. In *Transactions on Petri Nets and Other Models of Concurrency XIV*, Maciej Koutny, Lucia Pomello, and Lars Michael Kristensen (Eds.). Vol. 11790. Springer Berlin Heidelberg, Berlin, Heidelberg, 126–145. doi:10.1007/978-3-662-60651-3_5 Series Title: Lecture Notes in Computer Science.

[16] Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Irina Spiridonova, Haobin Ni, Dmitry Malloy, Juan Vazquez, Michael Tang, Omar Cardona, and Arti Gupta. 2022. Hardening attack surfaces with formally proven binary format parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022).* Association for Computing Machinery, New York, NY, USA, 31–45. doi:10.1145/3519939.3523708

[17] Ibtissem Talamali, Razika Lounas, and Mohamed Mezghiche. 2024. Formal Specification and Verification of MQTT Protocol Using CoQ Proof Assistant. In *2024 International Conference on Advanced Aspects of Software Engineering (ICAASE).* 1–6. doi:10.1109/ICAASE64542.2024.10850926