An Assessment Methodology for Learning Mission Characteristics

Joshua Meise and Stephen Taylor
Dartmouth College, Hanover NH, USA

joshua.m.meise.gr@dartmouth.edu
stephen.taylor@dartmouth.edu

Abstract: This paper presents a novel methodology for assessing the effectiveness of machine learning algorithms in deriving a protocol specification from a set of network traffic captures. Such specifications describe the data formats and mission characteristics associated with valid messages and form the basis for protecting military vehicles from malicious traffic. This paper enhances previous work, which automatically generates a hardware guard from a hand-written grammar, to produce a fully automated, end-to-end process that generates a parser, either in software or as a hardware guard, directly from mission training data sets. A hardware guard is realized as a Field Programmable Gate Array (FPGA) circuit block implementing a parser. The parser performs deep packet validation, checking that every message conforms to the associated grammar and rejecting invalid packets. Several modern machine learning algorithms and specification inference tools exist that can be leveraged to automatically infer a specification from a data capture. Unfortunately, to date, there has been a paucity of mission data sets to provide ground-truth, and the lack of a common set of metrics to assess learning algorithms. To this end, this paper introduces a parser repository that provides a ground-truth data set for a variety of formats and protocols, alongside equivalent protocol specifications expressed in several formalisms, namely BNF, Hammer, Daffodil, and Daedalus. Each set of equivalent specifications is provided with a common collection of true and false test vectors to validate their operation and benchmark learning performance using appropriate metrics. Specifications for standard formats, including JavaScript Object Notation (JSON), Universal Resource Locators (URLs), and Hypertext Transfer Protocol (HTTP), provide the basis for generic testing. The Micro Air Vehicle Link Protocol (MAVLink) and Space Packet Protocol (SPP) are used as examples of mission-oriented grammars. MAVLink is a byte-oriented protocol, while SPP is a bit-oriented protocol, leading to fundamental differences in the way that learning algorithms must operate to validate packets; the latter is also indicative of the complexities involved in parsing of SAE J1939 – used on ground vehicles – and MIL-STD-1553 – used on aircraft.

Keywords: Protocol reverse engineering, Parsing, MAVLink, Space packet protocol, Traffic validation

1. Introduction

Many military vehicles communicate with other vehicles and/or their ground control stations via wireless networks. This exposes an attack surface whereby an adversary may exploit vulnerabilities in the vehicle's onboard systems by embedding malicious implants within legitimate traffic. These implants may cause unexpected D5 effects (USAF, 2012) on the vehicle that undermine its operational mission. Consequently, it is valuable to validate both the syntax and semantics of messages in real-time, on board the vehicle before messages are acted upon. This *deep packet validation* ensures that all messages comply with both the expected format and that payloads comply with the bounds set on the vehicle's specific mission. Validation may be performed by a *software parser* on board the vehicle, or by a *hardware guard* acting as a bump-in-the-wire between the communication interface and the vehicle's onboard systems. Hardware guards can be realized with Field Programmable Gate Arrays (FPGA's) that implement a parser. This ensures that parsing occurs below the software stack, forming a hardware base-of-trust that is less susceptible to compromise. Previous work on packet validation has already resulted in a process that automatically generates a hardware guard from a protocol specification (Dahlstrom *et al.*, 2022; Taylor and Pope, 2024). The goal of this work is to enhance the process by directly generating the specification from a traffic data capture *taken during a mission training event*, allowing for the construction of mission-specific parsers. A central challenge is to produce a concise and intelligible specification, that engineers can understand, evolve, and optimize, without highly specialist skills in parser development or protocol specification writing.

There exist many formalisms for the expression of communication protocols and data formats, broadly referred to in this paper as *protocol specifications*. These formalisms extend from parser combinator libraries and data description languages to notation systems. A specification, written in such a formalism, consists of a set of rules defining the syntax and, to a certain extent, the semantics of the protocol. Parser combinator libraries, like Hammer (Riverside Research, 2026), build increasingly complex parsers by combining simple parsers, defined in a library. Data description languages, such as Daedalus (Diatchki *et al.*, 2024) and Daffodil (The Apache Software Foundation, no date), provide a succinct mechanism for expressing the syntax and semantics of a protocol in a domain-specific language. Notation systems, such as Backus-Naur Form (BNF) (Backus *et al.*, 1960), provide a means for defining a formal grammar specifying a protocol. It is possible, using pre-processors, to add syntactic notations to BNF that simplify specifications; xBNF (Taylor, Guzman and Meise, 2025) and ABNF (Crocker and Overell, 2008) both achieve such simplifications. Each of these formalisms provides the basis for the automatic

generation of a validation parser which produces a binary accept/reject answer, indicating whether a message's format complies with the specification. Validation parsers can be constructed using tools like Daffodil, Daedalus, or Hammer's native parser generators, or, for formal grammars expressed in BNF, Bison (Levine, 2009). There are also a wide variety of parsing methods (Grune and Jacobs, 2010), many of which are employed by these tools.

The automatic construction of a specification through inference, opens the door to the construction of a new generation of *mission-specific parsers*. Several protocol reverse engineering tools and techniques have already been developed to perform this inference (Cui, Kannan and Wang, 2007; Wang *et al.*, 2011; Chandler, Wick and Fisher, 2023). These tools aim to infer the syntax of a protocol from a data capture or network trace fully automatically. In contrast, the goal here is to work interactively with a systems engineer to formulate a representative specification; consequently, the requirement for a fully automated solution is less severe. The more pressing need is to incrementally build a formulation of the semantic meaning of data within message payloads. Sadly, to date, there has been a paucity of mission data sets to provide ground-truth, and a lack of a common set of metrics to assess alternative learning algorithms in this context. To this end, this paper introduces a methodology for baselining learning algorithms that is supported by an emerging parser experimentation repository (Taylor *et al.*, 2026).

2. Methodology

Figure 1 shows the end-to-end process that we envisage for designing a software parser or hardware guard directly from a data capture. Traffic data is captured from a mission training event and fed to an inference tool to infer a protocol specification. An engineer performs minor interventions to validate the output of the inference tool, ensuring correct semantic inference of protocol fields. The specification is then fed to a parser generator tool to construct either a software or hardware parser. The parser is validated against a set of true and false positive test vectors and, along with the specification itself, assessed against metrics evaluating correctness, performance, succinctness, and resource use.

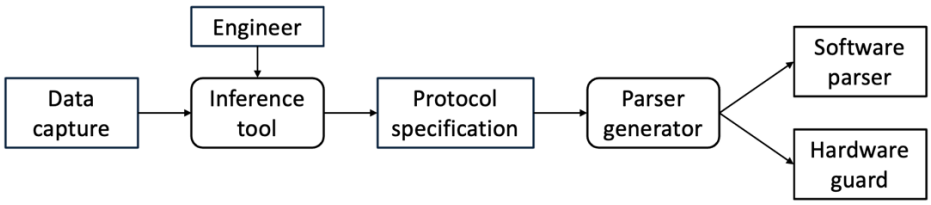


Figure 1: Parser development pipeline

There are currently a wide variety of inference tools, formalisms for expressing specifications along with corresponding parser generators, and parsing methods. Consequently, it is important to enable the selection of elements of the trade-space most suited to a designated mission based on well-formed ground-truth data and metrics. Our methodology thus involves four core steps:

1. Specifications:

- Baseline specifications to allow for comparative analysis of syntactic inference.
- Mission-specific specifications to assess learning algorithms' performance in the inference of semantics.

2. Testing framework: True positive tests form a representative selection of messages. False positive tests form a representative selection of invalid messages.
3. Implementation: Automated parser generation tools are applied to generate software parsers and/or hardware guards for each specification.
4. Assessment: Evaluating the resulting alternatives from the perspective of performance, succinctness, and resource metrics.

To date this methodology has been employed to build a parser experimentation repository that is available online as an open-source repository for evaluating parsing formalisms, parser generator tools, and learning algorithms in the Thayer School of Engineering at Dartmouth College (Taylor *et al.*, 2026). The Thayer repository includes a variety of equivalent parsers (expressed in xBNF, BNF, Hammer, and Daffodil), a simple baseline

learning algorithm (discussed in Section 5), automated implementation and validation scripts, and baseline test vectors.

3. Specifications

Protocol specifications may take the form of a *context-free grammar* (expressed in BNF or xBNF), a collection of combinators (as in Hammer), or a collection of statements in a *data description language*, such as Daedalus or Daffodil. To date, the Thayer repository provides specifications for three core standard formats: JavaScript Object Notation (JSON) (Bray, 2017), Universal Resource Locators (URLs) (Berners-Lee, Masinter and McCahill, 1994), and Hypertext Transfer Protocol (HTTP) (Nielsen *et al.*, 1999), expressed in xBNF, BNF, and Hammer. It also contains specifications for two *mission-oriented protocols*, namely the Micro Air Vehicle Link Protocol (MAVLink) (MAVLink, no date) and Space Packet Protocol (SPP) (CCSDS, 2020).

JSON, URL, and HTTP are examples of recursive textual data formats. Such formats are often used within message payloads and are conveniently expressed with context-free grammars which allow for recursive grammar rules. The Thayer repository demonstrates how the specification for JSON is developed incrementally, starting with simple strings and numbers (JSTRING, and JNUM respectively), then adding complexity by combining them within the full JSON definition. Similarly, the HTTP request for comment (RFC) (Nielsen *et al.*, 1999) largely leverages the format for URLs. The increasing complexity of these formats allows the repository to act as both a pedagogical tool for constructing specifications and permits evaluation of inference and parsing tools on different data formats.

MAVLink is an example of a byte-based tactical messaging protocol: every byte in a message is a semantic entity and no values need be analyzed as a subset of bits within a byte or across byte boundaries within a message. Most multi-byte fields in MAVLink messages represent standard types, such as floating-point numbers or fixed-width integers in little-endian byte order. The ground-truth data in the Thayer repository consists of MAVLink messages captured between an actual drone and its ground station. MAVLink allows for two message versions, MAVLink 1 and MAVLink 2. Figure 2 shows the serialization of MAVLink 1 messages which closely resembles the format of messages in various other tactical communication protocols: a fixed-length header containing a length field, followed by a variable-length payload. Mission-relevant data is generally contained in the payloads and includes information such as GPS coordinates, altitude data, and velocity values. Figure 3 shows the serialization of a sample payload for an Attitude message.

STX uint8_t	LEN uint8_t	SEQ uint8_t	SEQ ID uint8_t	COMP ID uint8_t	MSG ID uint8_t	PAYLOAD (0 - 255 bytes)	CHECKSUM uint16_t

Figure 2: MAVLink 1 message serialization

time_boot_ms uint32_t	roll float	pitch float	yaw float	rollspeed float	pitchspeed float	yawspeed float

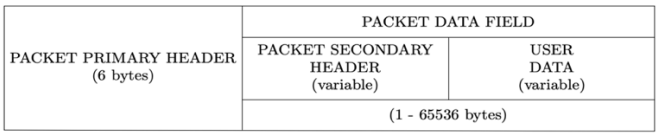
Figure 3: MAVLink Attitude message payload

Figure 4 illustrates the added *context-sensitive complexity* of MAVLink 2 messages: in addition to the fields in MAVLink 1 headers, MAVLink 2 headers contain two 1-byte flag fields, one of which is an incompatibility flag (INC FLAGS) that indicates if the message is signed with a 13-byte signature appended to the message. The payload of MAVLink 2 messages also truncates zero bytes during serialization.

STX uint8_t	LEN uint8_t	INC FLAGS uint8_t	COMP FLAGS uint8_t	SEQ uint8_t	SYS ID uint8_t	COMP ID uint8_t	MSG ID (3 bytes)	PAYLOAD (0 - 255 bytes)	CHECKSUM uint16_t	SIGNATURE (13 bytes)

Figure 4: MAVLink 2 message serialization

In contrast to MAVLink, Space Packet Protocol (SPP) is a bit-oriented protocol: SPP headers contain bit-fields, which do not align at byte boundaries. The overall structure of SPP messages is depicted in Figure 5 and contains a fixed-length packet primary header, followed by a variable-length packet data field. The Thayer repository contains a set of both telemetry and command SPP messages in binary files. These messages were captured using NASA's NOS3 satellite simulator (NASA-JSTAR, no date).



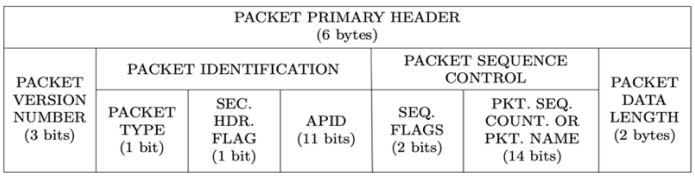


Figure 5: SPP message serialization

3.1 Baseline Specifications

To explore the performance of learning algorithms, it is useful to consider generic message formats, with well-defined syntax as a baseline. These baseline specifications provide little in terms of semantic parsing: only fields that are constant across all message types – typically found in the message headers – are enumerated, while payload fields may take the full range of values for every byte. The associated parsers *do not validate mission-specific data.* For example, Figure 6 shows a baseline specification, written in xBNF, for a MAVLink 1 Attitude message; here the wildcard symbol '*' indicates the presence of a single byte of *any* value.

```
message :  message_header message_payload checksum ;
message_header :  0xFE 0x1C 0x00 0x00 * * * 0x1E ;
message_payload :  time_boot_ms roll pitch yaw rollspeed pitchspeed yawspeed ;
time_boot_ms :  * * * * ;
roll :  * * * * ;
pitch :  * * * * ;
yaw :  * * * * ;
rollspeed :  * * * * ;
pitchspeed :  * * * * ;
yawspeed :  * * * * ;
checksum :  * * ;
```

Figure 6: Example baseline xBNF specification for MAVLink 1 Attitude message

3.2 Mission-specific Specifications

Mission-specific specifications are constructed by grouping related bytes in payload fields to form semantic entities and placing bounds on the values accepted. These specifications enable rigid semantic parsing in addition to the syntactic parsing provided by baseline parsers. Values are bounded based on what a systems engineer expects the vehicle to observe during a given mission and may include, for example, bounds on mission latitude and longitude, vehicle speed and altitude, or attributes of expected image data. For example, Figure 7 shows a rudimentary mission-specific xBNF specification for a MAVLink 1 Attitude message in a specific mission profile; the values within square brackets (i.e. [X – Y]) designate ranges of values in little-endian format.

```
message :  message_header message_payload checksum ;
message_header :  0xFE 0x1C 0x00 0x00 * * * 0x1E ;
message_payload :  time_boot_ms roll pitch yaw rollspeed pitchspeed yawspeed ;
time_boot_ms :  [ 'little_endian(0, uint32)' -'little_endian(6000, uint32)'] ;
roll :  [ 'little_endian(-3.15, float)' -'little_endian(3.15, float)'] ;
pitch :  [ 'little_endian(-3.15, float)' -'little_endian(3.15, float)'] ;
yaw :  [ 'little_endian(-3.15, float)' -'little_endian(3.15, float)'] ;
rollspeed :  [ 'little_endian(-0.57, float)' -'little_endian(0.59, float)'] ;
pitchspeed :  [ 'little_endian(-0.52, float)' -'little_endian(0.56, float)'] ;
yawspeed :  [ 'little_endian(-0.94, float)' -'little_endian(0.99, float)'] ;
checksum :  * * ;
```

Figure 7: Example mission-specific xBNF specification for MAVLink 1 Attitude message

While baseline specifications may be constructed in a once-off process, mission-specific specifications must be reconstructed and fine-tuned for each mission. The process of manually fine-tuning these specifications is time-intensive and requires expertise in specification writing, making it difficult under operational constraints. The ability to automatically infer, a-priori, a specification from a data capture on a training event, makes the construction of mission-specific specifications feasible.

4. Testing Framework

The Thayer repository contains a set of true and false positive tests for each protocol. These tests form the basis for evaluating the correctness of each specification and its associated parsers. As described in Section 3, the repository contains real data captures for both MAVLink and SPP missions. These captures form a true positive basis for testing the parsers on messages that have occurred during real or simulated missions. In addition to these data captures, the repository contains an automatic test generation framework for both protocols. Collectively, the resulting test suite ensures that the parsers generated from the baseline specifications accept all messages that are of the correct syntactic format (as outlined in Section 3.1). Similarly, true positive tests for the JSON, HTTP and URL specifications are manually constructed and test that the generated parsers accept the full range of expected formats; they also aim to test that various message attributes in valid combinations with one another are accepted by the parsers.

False positive tests ensure that the resulting parsers reject messages that have an incorrect format. For MAVLink and SPP, messages of an incorrect format are those which have invalid byte values in fields containing constant bytes, invalid enumerations in fields which only accept specific byte values, or messages whose payload length does not correspond to the payload length field in the message header. These tests are constructed through fuzzing, making minor alterations to a message with a valid format such that it no longer abides by the intended format defined in the specification. Similarly, tests for HTTP and URL check that messages with invalid character sequences or field values are rejected by the parsers. Since the JSON RFC (Bray, 2017) does place restrictive bounds on values or valid character sequences in specific fields, the false positive tests primarily test that messages with incorrect formats, such as invalid structure nesting or unbalanced parentheses, are rejected by the JSON parsers.

5. Implementation

Section 2 defines a methodology for an automated, end-to-end process for building parsers from mission-training data captures. This toolchain is implemented in the Thayer repository by a variety of automation scripts and forms the basis for evaluation.

To establish a baseline inference mechanism, Figure 8 shows a rudimentary learning algorithm for byte-based protocols, such as MAVLink. The algorithm takes as input a binary file containing messages captured during a mission-training event. The output of the algorithm is a context-free grammar expressed in xBNF, though it may easily be adapted to output any prospective specification language. The algorithm groups messages by length and creates a single non-terminal symbol to represent all messages of a particular length. The top-level production for the grammar is formed as a disjunction of all these non-terminals (i.e. the parser accepts a message that conforms to one of the individual message formats at a particular length). The non-terminal for each message length produces a rule containing a conjunction of terminals and new non-terminals, one for each byte in the message. Terminals correspond to bytes which are constant across all messages of that length. New non-terminals correspond to positions whose byte values vary across messages of the same length – these non-terminals capture byte positions that have multiple values. A new disjunctive rule is introduced for each of these new non-terminals that captures all possible byte values allowed in each position in messages of the relevant length.

Baseline Specification Inference Algorithm

Input: `messages.bin`: mission training dataset
Output: `gmr.xbnf`: inferred xBNF specification

```
read file messages.bin
group messages by length

for each message length do
    create non-terminal for length

for each message do
    if first message of length then
        initialize production for length's non-terminal with concatenation of terminals for each byte
    else
        for each byte in message do
            if terminal value in byte position and byte value differs from terminal then
                create new non-terminal for byte value
                add current byte value and former terminal to alternation for new non-terminal
            else if non-terminal value in byte position and byte value not in alternation then
                add byte value to alternation

write grammar to gmr.xbnf
```

Figure 8: Baseline specification inference algorithm

Since this rudimentary baseline algorithm does not infer the semantics of the protocol fields, the engineer's interaction with the tool is currently limited to verifying the final output specification. Currently, for a baseline, no multi-byte semantic grouping is applied, though reverse engineering tools, particularly BinaryInferno (Chandler, Wick and Fisher, 2023), are already available to produce such groupings. In accordance with standard practice for the training and testing of machine learning models, the input file is split into disjoint training and testing data sets prior to being input to the algorithm. The training set is used by the inference algorithm to infer the specification, while the testing set is used to assess this inferred specification (cf. Section 6).

To generate a parser, the specification is input to a parser generator for conversion into either a software parser or a hardware guard. Currently, the generation of software parsers is implemented by either Bison or Hammer. The input to the Bison is a context-free grammar; Hammer accepts an alternative formulation in the C programming language.

6. Assessment

As discussed in Section 2, each stage in the toolchain presented in Figure 1 must be validated and evaluated. To this end, we introduce a set of evaluation metrics for the accuracy of the inference tool, the succinctness of the protocol specification, the software and/or hardware resource utilization of the generated parser, and the performance of hardware guards.

The accuracy of the inference algorithm may be evaluated using *convergence*, *precision*, and *recall* as primary metrics. Convergence evaluates the rate at which an inferred specification covers all messages and formats contained in a mission training data capture. As outlined in the algorithm in Section 5, a new non-terminal is introduced to the specification upon the first occurrence of a message of a given length, or each time a byte field is found to have a non-constant value. Furthermore, an additional terminal is introduced each time a previously unobserved byte value is found to occur in a specific field for a given message length. The algorithm is thus said to converge once the number of inferred terminals and non-terminals remains constant. Precision and recall are metrics for the accuracy of an inference algorithm in deriving a specification that correctly incorporates critical information about a protocol. Precision is an indicator for how many of the byte values in the inferred specification correctly match with a byte value in the actual data capture. Recall quantifies how many of the byte values in the actual data capture were correctly derived in the inferred specification. Mathematically, precision is the ratio of true positive predictions to all positive predictions (the sum of true positives and false positives), and recall is the ratio of true positive predictions to all actual positives (the sum of true positives and false negatives). A true positive is a byte value in the rule for a non-terminal in the inferred specification that correctly matches with a byte value in the corresponding field in the testing set. Conversely, a false positive is a byte value in the rule for a non-terminal in the inferred specification that is outside the range of byte values deemed appropriate for the given mission. False negatives are byte values that appear in a field in the testing set that are not enumerated in the non-terminal corresponding to that field in the inferred specification. Precision is thus a metric for how "permissive" a parser is; parsers with higher precision will be less permissive

to messages with greater variation in format from the training data. Recall indicates how good a parser's coverage is; a higher recall value indicates that an inferred specification is representative of messages that may be observed in a live mission.

The succinctness of the inferred specification is assessed using the *number of lines of code, terminals, and non-terminals* in the specification as metrics. The number of lines of code provides an indication of how easy the specification is to understand. The number of terminals and non-terminals provide a preliminary indication of the parser's software and hardware resource utilization, as well as the redundancy of the specification. In addition to evaluating an inference tool's ability to construct a succinct protocol, these metrics also form the basis for comparing the expressive qualities of alternative formalisms.

The performance of software parsers is evaluated based on *the binary size* and *memory usage* of the executable software parser, as well as its *execution time*. This provides an indication of the computational resources required for the implementation of the software parsers. Since a primary application locale for many such parsers is a resource constrained computational environment onboard a military vehicle, these metrics provide valuable insight into the viability of software parsers. They also provide a basis for evaluating alternative parsing algorithms, and the parser generator tools that employ those algorithms.

Similarly, both the resource use and performance of hardware guards must be evaluated. Since hardware guards are often implemented on FPGAs, FPGA resource utilization forms a critical metric. Metrics that indicate FPGA resource utilization include *Look-up Table, Flip-Flop, and Block RAM usage*. Evaluation of these metrics provides an indication of the implementation viability of parsers on cost-effective, everyday FPGA devices. Hardware guard performance is evaluated considering *bandwidth* and *latency* as metrics. Hardware guards need to enable the processing of packets at line rate to prevent backlogs. As in the case of software parsers, these metrics provide the basis for the comparison of alternative parsing algorithms and methods.

Dahlstrom *et al.* (2022) provide an example assessment of the performance of hardware guards, while Cavanagh and Taylor (2026) provide an example analysis of software parser performance. Here we evaluate the baseline inference algorithm, introduced in Section 5, using a mission training data capture containing 33,261 MAVLink 2 messages. Messages with 48 alternative payload types are present in this data capture. The data capture was input to the inference algorithm to produce a baseline xBNF specification for MAVLink 2. Since this rudimentary inference algorithm simply enumerates values that are observed in the training data no false positives exist; consequently, precision is a meaningless metric in this baseline case since it is always 100%. Figure 9 shows the convergence of the inferred specification to the ground-truth specification when run with a 100:0 train:test split. It is clear from Figure 9 that the number of non-terminal symbols converge faster than the number of terminal symbols. Information used to derive non-terminal symbols (all message lengths and variable fields) is thus contained in approximately the first third of the data capture.

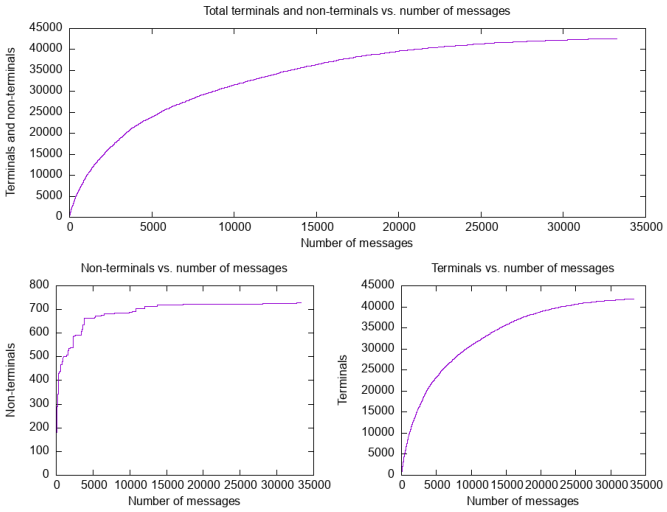


Figure 9: Convergence of inference algorithm on train:test split of 100:0

Table 1 presents the relevant metrics of this algorithm when run with four different splits of testing and training data. These metrics include the inference algorithm's recall, the number of lines of code in the inferred specification, the number of terminals and non-terminals in the inferred specification, and the number of

messages from the full mission training data capture that are rejected by the generated Bison look-ahead, left-to-right, rightmost derivation parser (LALR(1)) parser. This latter number may be reduced by pointing the engineer to areas of the resulting specification that are too tightly constrained, allowing a quick way to manually edit the grammar.

Table 1: Results from baseline inference algorithm

Train:Test	Recall (%)	Lines of code	Terminals	Non-terminals	Valid messages rejected (%)
30:70	96.1658	5,082	31,496	674	15.0
50:50	97.7469	5,821	36,750	712	5.84
80:20	99.2436	6,399	41,034	727	0.00926
100:0	100	6,488	41,814	727	0

While the inferred specifications are verbose, they achieve high levels of recall, even in the 30:70 train:test split. It is particularly notable that most of the variation in the specification after observing 30% of the messages arises due to the observance of fresh byte values (as indicated by the increasing number of terminals). To mitigate such variation and accelerate convergence, future work aims to decide on appropriate ranges for terminals as opposed to enumerating each observed byte value in the rule for a non-terminal. This will both increase recall and reduce the verbosity of the inferred specifications. It will furthermore lead to the ability to learn representative specifications from smaller training mission training data captures.

7. Conclusion

This paper lays the foundation for an automated, end-to-end process for producing a parser directly from a data capture, thereby making construction of mission-specific parsers feasible. To this end, we describe the methodology, the Thayer repository that supports it, and a baseline algorithm for the automatic construction of protocol specifications from mission training data captures. Suitable metrics for the evaluation of this end-to-end toolchain are presented with example results, thereby standardizing the basis evaluation of the various components of the toolchain. By rigorously applying this methodology, we expect it to eventually be possible for a systems engineer to rapidly select an inference algorithm, specification formalism, and parsing technique that generates an effective parser to protect a vehicle on a given mission.

8. Future Work

The algorithm proposed in Section 5 as a baseline can easily be adapted to incorporate formalized learning methodologies into the process of specification inference. The goal is to both increase the coverage of the inferred specification whilst keeping field bounds stringent enough to prevent malicious traffic from penetrating the parser. The first step in this process is to decide when to define a non-terminal as a range between two byte values, rather than an enumeration of several byte values. This will increase recall by decreasing false negatives. Ranging will, however, result in a more “permissive” parser with higher false positives and thus lower precision.

A further step is to perform semantic analysis of the fields present in a data capture. The aim is to identify not only fields that correspond to common datatypes, such as fixed-width integers and floating-point numbers, but also to infer what fields are representative of mission-relevant values, such as GPS coordinates, altitudes or velocities. This will enable an engineer to interact with the inference tool in deciding on appropriate bounds on mission-relevant fields, as opposed to byte-level fields. Reverse engineering tools, such as BinaryInferno (Chandler, Wick and Fisher, 2023), have already shown great promise in inferring semantic information about fields in MAVLink-style messages. This work can be leveraged as a comparative baseline for evaluating the semantic analysis capabilities of the inference tool.

Acknowledgements

This research is supported by the Institute for Security Technology Studies (ISTS) at Dartmouth College. The authors would like to thank Sergey Bratus for his insights into state-of-the-art parsing and reverse engineering technologies.

Ethics Declaration: No ethical clearance necessary.

AI Declaration: No AI tools were used in the development of this paper.

References

Backus, J.W. *et al.* (1960) "Report on the Algorithmic Language ALGOL 60," *Communications of the ACM*, 3(5), pp. 209–314. Available at: https://doi.org/10.1145/367236.367262.

Berners-Lee, T., Masinter, L.M. and McCahill, M.P. (1994) *Uniform Resource Locators (URL)*. Request for Comments RFC 1738. Internet Engineering Task Force. Available at: https://doi.org/10.17487/RFC1738.

Bray, T. (2017) *The JavaScript Object Notation (JSON) Data Interchange Format*. Request for Comments RFC 8259. Internet Engineering Task Force. Available at: https://doi.org/10.17487/RFC8259.

Cavanagh, B. and Taylor, S. (2026) "Apache Daffodil for Network Traffic Validation," in *European Conference on Cyber Warfare and Security* forthcoming.

CCSDS (2020) *Space Packet Protocol*. CCSDS 133.0-B-2. Washington, DC, USA: Consultative Committee for Space Data Systems.

Chandler, J., Wick, A. and Fisher, K. (2023) "BinaryInferno: A Semantic-Driven Approach to Field Inference for Binary Message Formats," *Network and Distributed System Security Symposium*, San Diego, CA, USA: Internet Society. Available at: https://doi.org/10.14722/ndss.2023.23131.

Crocker, D. and Overell, P. (2008) *Augmented BNF for Syntax Specifications: ABNF*. Request for Comments RFC 5234. Internet Engineering Task Force. Available at: https://doi.org/10.17487/RFC5234.

Cui, W., Kannan, J. and Wang, H.J. (2007) "Discoverer: Automatic Protocol Reverse Engineering from Network Traces," *16th USENIX Security Symposium (USENIX Security 07)*. Available at: https://www.usenix.org/conference/16th-usenix-security-symposium/discoverer-automatic-protocol-reverse-engineering-network.

Dahlstrom, J. *et al.* (2022) "Automatic Construction of Hardware Traffic Validators," *European Conference on Cyber Warfare and Security*, 21(1), pp. 60–69. Available at: https://doi.org/10.34190/eccws.21.1.200.

Diatchki, I.S. *et al.* (2024) "Daedalus: Safer Document Parsing," *Daedalus*, 8(PLDI), pp. 816-840. Available at: https://doi.org/10.1145/3656410.

Grune, D. and Jacobs, C.J.H. (2008) *Parsing Techniques: A Practical Guide.* 2nd edn. New York: Springer.

Levine, J. (2009) *Flex & Bison: Text Processing Tools*. Sebastopol, CA: O'Reilly Media, Inc.

MAVLink (no date) *MAVLink Developer Guide | MAVLink Guide*. Available at: https://mavlink.io/en/ (Accessed: January 23, 2026).

NASA-JSTAR (no date) *NASA Operational Simulator for Small Satellites — NOS3 documentation*. Available at: https://nos3.readthedocs.io/en/latest/ (Accessed: January 15, 2026).

Nielsen, H. *et al.* (1999) *Hypertext Transfer Protocol – HTTP/1.1*. Request for Comments RFC 2616. Internet Engineering Task Force. Available at: https://doi.org/10.17487/RFC2616.

Riverside Research (2026) "riversideresearch/hammer." Riverside Research. Available at: https://github.com/riversideresearch/hammer (Accessed: January 15, 2026).

Taylor, S. *et al.* (2026) "lvln/thayer_parsers: Parser Experimenation Repository." Available at: https://github.com/lvln/thayer_parsers (Accessed: January 15, 2026).

Taylor, S., Guzman, B. and Meise, J. (2025) "thayer_parsers/xbnf at main · lvln/thayer_parsers." Available at: https://github.com/lvln/thayer_parsers/tree/main/xbnf (Accessed: January 23, 2026).

Taylor, S. and Pope, G. (2024) "Hardware Sequence Combinators," *International Conference on Cyber Warfare and Security*, 19(1), pp. 358–365. Available at: https://doi.org/10.34190/iccws.19.1.1965.

The Apache Software Foundation (no date) *Apache Daffodil*. Available at: https://daffodil.apache.org/ (Accessed: January 3, 2026).

USAF (2012) *Cyber Vision 2025*. U.S. Air Force.

Wang, Y. *et al.* (2011) "Biprominer: Automatic Mining of Binary Protocol Features," *2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pp. 179–184. Available at: https://doi.org/10.1109/PDCAT.2011.25.