Exploring Zero-Shot Prompting for Generating Data Format Descriptions

Prashant Anantharaman*
Narf Industries

Vishnupriya Varadharaju
Narf Industries

***Abstract*—Parsers validate and process untrusted user input and transform it into data structures that provide easier access. Software engineers either build these parsers for data formats from scratch or leverage libraries targeting specific formats. The recent surge in data description languages (DDLs) and parser combinator libraries for parsing data formats has aided developers in producing parsers using standardized tools. However, producing a parser for an unfamiliar data format in an unfamiliar DDL can be daunting, given the learning curve of understanding two specifications or manuals well.**

As more researchers adopt tools such as GitHub Copilot [39] to simplify their programming tasks, we ask whether LLMs already hold sufficient knowledge to produce valid DDL specifications for popular data formats. To explore this, we systematically prompt LLMs to provide specifications in valid DDL syntax and evaluate whether these specifications are *syntactically valid* and *correct*.

We found that while some LLMs, such as GPT 4 Turbo, Claude 3.5 Sonnet, and Deepseek V3, can produce valid Kaitai Struct YAML files, Hammer C files, and Rust Nom files, they struggle to produce valid specifications in complex DDLs, such as DaeDaLus, Spicy, and DFDL. In general, all LLMs fare much better at producing syntactically valid C code using the Hammer library and Rust code using the Nom library, given the large corpora of valid C and Rust code available. None of the LLMs in our test were able to produce a valid DFDL file or DaeDaLus file. We also found that while providing the specification manuals for the DDLs did not help in producing more syntactically valid specifications, providing sample specification files modestly increased the number of successful compilations.

I. Introduction

Large Language Models (LLMs) have been effectively used to tackle complex security tasks, such as reverse engineering stripped binaries by labeling functions and variables in them [41], [49]. Similarly, programmers have been extensively using AI assistants, such as GitHub Copilot, to aid in bug fixes and routine programming tasks [36]. While these results demonstrate that programmer effort is significantly reduced by using LLMs for these targeted tasks, parsers for data formats are still written mainly by hand today. Figure 1 shows the iterative approach taken to build parsers for data formats.

File formats such as PDFs are incredibly complex, spanning over 1000 pages, and do not contain any grammar descriptions [48] (the machine-readable PDF specification, the Arlington DOM, is primarily to check types on dictionary and array objects). While researchers have used natural language processing to aid with these parser generations, they have been error-prone and have required extensive human-assisted testing [25].

*Corresponding author: prashant.anantharaman@narfindustries.com

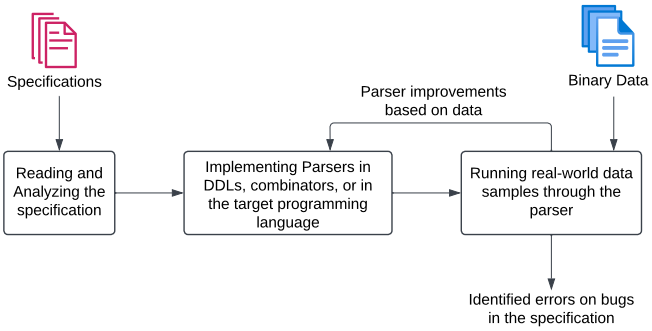

Fig. 1. The workflow for how developers implement specifications in parsers.

Vendors also often create *dialects* of formats to suit their specific needs. These dialects may leverage bytes reserved for future implementations or violate the specification entirely by implementing new features. These patterns have been observed extensively in the evolution of the Modbus protocol and the PDF file format. Researchers building these parsers often encounter these *specification drifts* and must decide whether to throw errors on these changes or allow them by modifying the specification.

This paper hypothesizes that given LLMs have been trained on publicly available data (including web pages), they may already include sufficient context on the syntax of data description languages (DDLs) and data format specifications. These DDLs are often used to describe various formats to provide systematic frameworks to parse unsanitized data into parsed objects. There are currently many popular DDLs at various stages of development: Kaitai Struct [50], Apache DFDL [26], DaeDaLus [14], Spicy [43], and Parsley [30]. Other popular DDLs, such as the format analyzers on Wireshark and EverParse [37], are designed to cover only network protocols.

A natural question a reader may ask is that if LLMs understand format specifications, why not directly generate code in the target programming language? DDLs use a paradigm where developers can easily understand format descriptions and make changes and bug fixes as necessary [20], adhering to LangSec principles [40]. Parser combinator code (such as Nom for Rust and Hammer for C/C++/Python) is directly in the target programming language.

Researchers have already demonstrated that LLMs can be provided with domain-specific language (DSL) grammar in Backus-Naur Normal Form (BNF), and they can be prompted

to produce data that adheres to that grammar or is a subset of it [44], [47]. However, producing valid parsers for data formats is cumbersome and requires specific expertise. Fakhoury et al. [15] presented 3DGen to tackle these problems to automate the producing specifications in DSLs with the assistance of LLMs.

While they primarily focused on the EverParse [37] parsing tool, we go beyond by demonstrating how these LLMs can generate specifications in a wide range of DDLs. In addition, while EverParse was designed for network protocols, we also studied specification generation for file formats. Finally, instead of relying on generating inputs from the specification using tools such as 3DTestGen, we constructed a test corpus using publicly available data via GovDocs, CommonCrawl, and packet captures.

We explore how accurate parser production can be made accessible to a larger audience with minimal domain expertise in DDL technologies and the targeted data formats. We seek to answer the following research questions as part of this investigation:

RQ1: Can off-the-shelf LLMs produce DDL code that is *syntactically valid*?

RQ2: Does the generated DDL code cover 100% of the format specification?

RQ3: Does the generated DDL code reject malformed inputs?

RQ4: Can LLMs learn the syntax of DDLs they do not know using example specifications and manuals?

We frame the following hypotheses on the above research questions.

Hypothesis 1: LLMs can produce some syntactically valid DDLs, but their success rate depends on the complexity of the data formats.
Hypothesis 2: LLMs tend to miss strict specification requirements and constraints.
Hypothesis 3: LLM-generated DDL code is permissive and accepts some malformed inputs.
Hypothesis 4: After being provided with manuals and examples, LLMs can produce *some* syntactically valid DDL code that was previously invalid.

We created a list of 20 data formats (network protocols and file formats) and collected a corpus of at least 100 representative samples for each of these data formats. We then evaluated the LLMs on a collection of evaluation scenarios designed to answer the above questions. To this end, our contributions are as follows:

- We created the most extensive corpus of validated specifications in several DDLs for the wider community (Section III).
- We demonstrate that without much context provided via prompting, some LLMs can provide accurate, syntactically valid specifications in a DDL syntax. We validated these claims using a corpus of publicly available files and packet captures (Section IV and V).
- Finally, we show that by providing example specifications, we are able to teach LLMs DDL syntax. We show that Few-Shot Learning performs better at providing compiling DDL files than Zero-Shot Learning (Section VI).

a) Organization: Section II provides the necessary background for the paper. Section III describes the experimental setup and provides an overview of our LLM-based DDL Generation system. Sections IV, V, and VI tackle the research questions posed. Finally, Section VIII discusses future directions and the impact of this work on the LangSec field.

b) Availability: Our experimental setup was built using around 1500 lines of Python code. The source code and the DDL files generated by our experiments are available on our GitHub repository [31]. The repository also contains the scripts and SQLite database files used to generate all the graphs and tables included in this paper.

II. Background

A. Parser Security

Parsers are a program's first line of defense. They must reject invalid data while converting valid data into a data structure for the rest of the program. Language-theoretic security (LangSec) proposes that parsers for data formats must be separated from the rest of the code and fully validate any inputs before the rest of the code operates on them.

Given the critical nature of parsers, it is vital to ensure that the parsers can be easily audited and missing checks can be added. *Shotgun parsing* approaches, where parsing and processing logic are interspersed, make it harder to realize these LangSec goals [7]. To tackle the challenge of shotgun parsing, two broad paradigms have been proposed: (1) using parser combinators to describe data formats—making it easier to write code that looks like formal grammars, and (2) describing data formats in the syntax of formal grammars in Data Description Languages (DDLs) and using compilers to generate parser code from them. In our study, we incorporated four DDLs and two parser combinators, as shown in Table I. Additionally, Figure 2 demonstrates the syntax used by each DDL and parser combinator in our study for a portion of the IPv4 packet.

B. Data Description Languages

In DDLs, we describe grammar in particular syntax and run a parser generator to generate code in any target language. DDLs are, hence, essential to standardization efforts since standardization bodies can define the syntax and semantics of a data format in a DDL and rely on the parser generation tools to generate parsers for most languages.

Kaitai Struct is a data description language and a parser generation toolkit that supports runtimes in several languages [50]. It is one of the most used DDLs and supports various formats in its gallery. Kaitai uses YAML syntax and unconventionally handles offsets using a syntax where locations and types associated with those locations are defined. Additionally, they also support a web IDE to build specifications and visualize them with data.

Data Format Description Language (DFDL) is an industry-standard often used to model text-based and binary data [26]. The Apache Daffodil runtime uses the DFDL

TABLE I
A COMPLETE LIST OF FORMATS, PROTOCOLS, DDLS, AND LLMS WE INCLUDE IN OUR STUDY.

File Formats	Network Protocols	Parsing Technologies	Large Language Models
PNG	DNS	Kaitai Struct	Claude 3.5 Sonnet
JPG	ICMP	DFDL	Claude 3.5 Haiku
GIF	Bitcoin Transactions	DaeDaLus	GPT-4o
TIFF	Modbus	Zeek Spicy	GPT-4-Turbo
NITF	NTP	Hammer	Llama-3.3-70B
DICOM	TLS Client Hello	Rust Nom	DeepSeek Coder V3
ELF	HTTP/1.1		Gemini 1.5 Flash
ZIP	MQTT		
GZIP	ARP		
SQLITE3	HL7 v2		

Hammer Parser

```
h_sequence(
    ...
    h_uint16(), // total_length
    h_uint16(), // identification
    h_uint16(), // b67
    h_uint8(),  // ttl
    h_uint8(),  // protocol
    h_uint16(), // header_checksum
    h_repeat_n(h_uint8(), 4), //
    ↪  src_ip_addr
    h_repeat_n(h_uint8(), 4), //
    ↪  dst_ip_addr
    NULL
);
```

Rust Nom

```
...
let (input, length) =
↪  number::streaming::be_u16(input)?;
let (input, id) =
↪  number::streaming::be_u16(input)?;
let (input, flag_frag_offset) =
↪  flag_frag_offset(input)?;
let (input, ttl) =
↪  number::streaming::be_u8(input)?;
let (input, protocol) =
↪  ip::protocol(input)?;
let (input, chksum) =
↪  number::streaming::be_u16(input)?;
let (input, source_addr) =
↪  address(input)?;
let (input, dest_addr) =
↪  address(input)?;
...
```

DaeDaLus

```
def IPv4_header_s =
  struct
    ...
    total_length    : uint 16
    identification  : uint 16
    b67             : uint 16
    ttl             : uint 8
    protocol        : uint 8
    header_checksum : uint 16
    src_ip_addr     : [uint 8; 4]
    dst_ip_addr     : [uint 8; 4]
```

Kaitai Struct

```
seq:
    ...
  - id: total_length
    type: u2be
  - id: identification
    type: u2be
  - id: b67
    type: u2be
  - id: ttl
    type: u1
  - id: protocol
    type: u1
  - id: header_checksum
    type: u2be
  - id: src_ip_addr
    size: 4
  - id: dst_ip_addr
    size: 4
```

DFDL

```
<xs:sequence>
    ...
    <xs:element name="Length"
    ↪  type="b:bit" dfdl:length="16"/>
    <xs:element name="Identification"
    ↪  type="b:bit" dfdl:length="16"/>
    <xs:element name="Flags"
    ↪  type="b:bit" dfdl:length="3"/>
    <xs:element name="FragmentOffset"
    ↪  type="b:bit" dfdl:length="13"/>
    <xs:element name="TTL" type="b:bit"
    ↪  dfdl:length="8"/>
    <xs:element name="Protocol"
    ↪  type="b:bit" dfdl:length="8"/>
    <xs:element name="Checksum"
    ↪  type="chksum:IPv4Checksum"/>
    <xs:element name="IPSrc"
    ↪  type="ip:IPAddress"/>
    <xs:element name="IPDest"
    ↪  type="ip:IPAddress"/>
</xs:sequence>
```

Spicy

```
type ip4_hdr: record {
    ...
    len: count;
    id: count;
    DF: bool;
    MF: bool;
    offset: count;
    ttl: count;
    p: count;
    sum: count;
    src: addr;
    dst: addr;
};
```

Fig. 2. Sample Files in DDL syntax for a portion of the IPv4 header packet. We implemented the same portion of the packet in Hammer and DaeDaLus syntax, whereas Rust Nom [6], Kaitai [23], DFDL [5], and Spicy [51] were sourced from publicly available repositories.

specifications or schemas to parse data and converts the data to an information set (Infoset). DFDL schemas are described in XML files. Unlike other DDLs, DFDL also includes a serializer to restore these infosets into binary data.

The **DaeDaLus** DDL [14] defines data formats in a syntax that closely resembles Nail [4]—while implementing it as a library within Haskell. Additionally, DaeDaLus also includes a parser generator for C++. At its core, DaeDaLus relies on various Haskell features, for example, offsets and other complex constructs.[1]

[1]The authors have used DaeDaLus and Daedalus interchangeably.

Zeek Spicy parser generator is a part of the Zeek Intrusion Detection system project [43]. It uses a domain-specific language to describe file formats and network protocols. The *spicyc* compiler can generate a compiled parser or produce C++ code that can be imported into existing applications.

C. Parser Combinators

The idea of parser combinators originates from functional programming [21]. Parser combinators make composing larger parsers using these combinators a lot easier by allowing combining parsers that may perform partial parsing operations

TABLE II
PROMPT TEMPLATES WE USED IN OUR STUDY TO QUERY LLMS.

Prompt Type	Template
First Prompt	You are a software developer who has read the {specification} for the {format}. Can you list all the fields in the specification along with all the values each field can take?
Generate Scripts	Can you use this knowledge to generate a {ddl} specification for the {format} in {output} format? Make sure to cover the entire specification, including any optional fields. Do not provide any text response other than the format specification. Show only the complete response. Do not wrap the response in any markdown.
Fixing Errors	The previous response gave me an error. Can you use this error message: "{message}" to improve the specification and give me an improved, complete, and fixed {ddl} specification in {output} format. Give me only the complete generated code and no text with it. Ensure that the previous requirements are still met.
From Samples	You are a software developer familiar with the {specification} for the {format}. Given various sample specifications for different formats in {output}, use the insights from these samples to generate a comprehensive {ddl} specification for {format} in {output} format. Ensure that the entire specification is covered, including all optional fields. Provide only the complete {format} specification without any additional text or markdown formatting. Here are sample specifications for reference {sample_text}.
Using Manuals	Study the attached documentation carefully for {ddl} to answer my upcoming questions. You are a software developer who has read the {specification} for the {format}. Can you use this knowledge and the {ddl} documentation shared previously to generate a {ddl} specification for the {format} in {output} format? Make sure to cover the entire specification, including any optional fields. Return the response containing only the specification without any explanation.

using higher-order functions like Kleene star, choices, and sequences.

A parser-combinator tool is a library that provides these combinator functions—where each of these functions returns a parser object that can be called on input. On a successful parse, these parsers return an abstract syntax tree (AST) to provide access to these parsed objects.

Although parser combinators are standard in the functional programming world, **Hammer** [35] and **Nom** [10] were among the first parser combinators to be introduced in systems programming. Hammer is a C-based parsing library that provides bindings in several languages. Whereas Nom is a Rust-based zero-copy parsing library.

D. Data Formats

Table I lists the file formats, DDLs, and LLMs we leverage in this paper. The data formats selected are widely used and often straightforward (unlike PDF or Microsoft Office formats). For example, network protocols such as Network Time Protocol (NTP), Address Resolution Protocol (ARP), and Message Queuing Telemetry Transport (MQTT) protocol are all limited in the number of fields and the values each field can take.

III. EXPERIMENTAL SETUP

To study the research questions posed, we built a system that sends a sequence of prompts (Table II) to the set of LLMs. The first three prompts are provided sequentially, building on previous context. The prompt to fix errors is tried up to three times with previously encountered error messages. Figure 3 provides an overview of our approach. We selected 20 popular data formats spanning network protocols and file formats. For each of these data formats, we identified the specification versions, RFC numbers, and URLs when RFCs were not available. We selected six popular DDLs that are capable of describing a wide range of formats.

We prompted each LLM to generate a file in a specific data format using a particular DDL syntax. We stored these files and ran a compiler or syntax checker to ensure the produced file was syntactically valid. We prompted the LLMs to correct the DDL file with an error message in the event of syntax errors. We attempt to resolve the DDL in three subsequent prompts, in addition to the original query.

While the same DDL file is overwritten when an LLM produces an improved specification, we store intermediate results in an SQLite database. The database stores the LLM response and other metadata, such as the number of attempts and whether the response was compiled successfully.

Since parser-combinator libraries are essentially in the target programming language (C for Hammer and Rust for Nom), the parsing function needs to be invoked in the main function. As part of the prompts, we asked the LLMs to produce code that can take a binary input file as a command-line input to ensure that we can easily compare implementations. Additionally, to evaluate these DDLs, working implementations of each DDL compiler and execution engine were needed. To aid future researchers, we created a Docker image with working implementations of each compiler.

A. Data Collection

We provided the LLMs with the specification or RFC version numbers and the full name of the format to provide them with sufficient information to produce DDL files. For RQ4, we also provided the LLMs with five sample specifications in DDL syntax, which we collected from the DDLs' GitHub repository.

To evaluate the generated parsers, we collected files from the GovDocs1 corpus, which contains a total of 986,278 files across PDFs, images, source code, and various Microsoft Office files [16]. We leveraged this corpus to collect PNG (4,125), JPEG (109,283), and GIF (36,302) files. We sourced various packet captures from the Wireshark GitHub repository and website for various common network protocols, like NTP. We relied on packet captures from NYPA's AGILe lab to test Modbus and ARP implementations [32].

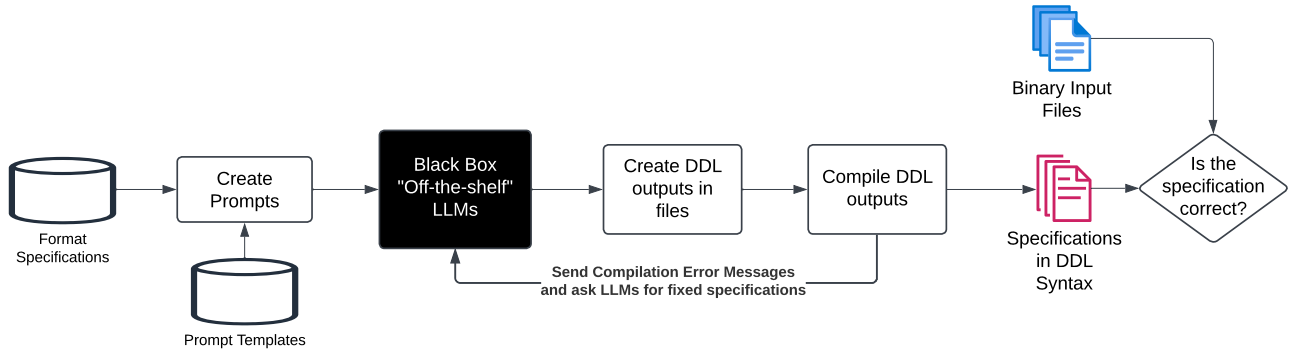

Fig. 3. Overall workflow of the experimental system. The overall goal of the project is to construct a library of parsers for each DDL.

B. Large Language Models

Many LLMs are available today, each catering to different needs and offering various features, such as improved reasoning, faster performance, higher intelligence, and different sizes (based on context lengths).

For our research, we selected seven different models from various companies, including proprietary and open-source options. The models chosen are from leading companies known for offering high-performing yet cost-effective models in the industry. We selected OpenAI's GPT-4 Turbo and GPT-4o [33], Anthropic's Claude Sonnet 3.5 and Claude 3.5 Haiku [2], and Google's Gemini 1.5 Flash [17]. Additionally, we experimented with open-source models from DeepSeek Chat V3 [12] and Meta's latest Llama 3.3 [28]. Few of these models reported improved context-retention capabilities over long context windows, when compared to their predecessors. Models in the Gemini 1.5 family even demonstrated strong performance in long-context benchmarks like the Needle-in-a-Haystack task [11].

For all models except Llama 3.3, we obtained direct access to the API from the respective organizations for a fee based on the model type and the number of tokens used. An API key is required to interact with a model through an API request. For Llama 3.3, we used Together AI's hosting infrastructure, which offers cloud-based access to open-source models for a nominal token-based fee. This approach is advantageous for users lacking the computational resources to deploy LLMs locally.

Among the myriad variables influencing LLM performance, temperature is one such parameter governing output variability. Temperature modulates the randomness of model-generated responses: at a setting of 0, the model operates deterministically, yielding identical outputs for identical prompts. Conversely, as the temperature increases, response diversity and creative expression become more pronounced, introducing greater variability in output. API-based models grant users explicit control over this parameter, offering a crucial degree of flexibility in experimentation.

Following prior work, we varied the LLM temperature setting from 0, 0.25, 0.5, 0.75, and 1 for all our LLMs to test variability in the outputs [36].

Another critical parameter is the *max_tokens* setting, which dictates the maximum number of tokens an LLM can generate in response to a single request. We standardized this parameter across all models at 2048 tokens to ensure uniformity in evaluation. Notably, the total number of tokens, comprising both input and generated tokens, must remain within the model's designated context length. By capping *max_tokens*, we aimed to mitigate discrepancies arising from differing default token limits across various LLM architectures.

a) Understanding and Retrieving Specifications: The first query asked the LLM whether it recognized and understood the specifications for a particular standard. If it did, it was prompted to share the full specification. This step was crucial in determining whether the model had prior knowledge of the topic and could provide accurate details. Studies have shown that explicitly establishing context in initial prompts leads to improved coherence in LLM responses [53].

b) Generating Structured Specifications: Once the model confirmed its understanding, it was instructed to use that knowledge to generate specifications in specific structured formats. For instance, if the task required defining a file format or network protocol, the model was asked to produce the specification in a DDL syntax. The generated output was then compiled and checked for correctness.

c) Error Handling and Iterative Correction: If errors were encountered on compilation, they were extracted and used to generate a follow-up prompt. The model was then asked to refine its response based on the error message and return a corrected version of the specification. This process was repeated thrice, with each iteration evaluating whether the generated code had been compiled successfully.

To ensure context retention, each subsequent prompt included all prior interactions. This method maintained continuity in the conversation, preventing the model from losing track of prior interactions. However, one inherent limitation of this approach is the constraint imposed by the model's context window. If the conversation exceeds the model's context limit, earlier messages are truncated, leading to the potential loss of critical details. This constraint is why we restricted the iterative correction process to three cycles.

IV. RQ1: Zero-Shot DSL code generation

To explore this research question, we studied the responses provided by various LLMs when we prompted them to provide a specification in the DDL syntax. We sent each LLM between 2000 and 4000 queries.

Figure 4 shows heatmaps denoting all six DDLs: Kaitai Struct, DaeDaLus, Zeek Spicy, Rust Nom, DFDL, and Hammer. The specifications produced by LLMs for DFDL resembled valid XML but contained syntactic errors that the LLMs were not able to fix when provided with the error messages. As a result, we were unable to construct a single DFDL specification that could compile. Appendix C shows the commands we used on all the DDLs to check for syntactic validity.

a) Kaitai Struct: We used Kaitai Struct's compiler to generate Python files. These files contain a class that can be invoked with a byte array passed as an argument. These classes throw an exception when the parsing fails at any point. During the compilation stage, the compiler throws syntax errors that we can provide back to the LLM to improve the results. Figure 4 shows that many LLMs could produce valid files since Kaitai Struct uses a standard YAML syntax. However, Gemini could not produce any valid Kaitai Struct YAML files, whereas Llama only produced one.

b) Hammer: We prompted the LLMs to produce a C program that takes a binary file as a command-line argument and invokes the Hammer parser on it. In the case of Hammer parsers, the GPT models are not very successful at producing valid files. In comparison, Claude 3.5 Sonnet and Deepseek V3 produce many valid files.

c) DaeDaLus: We invoke the DaeDaLus compilation command on the *.ddl* files to check for syntactic validity. Unfortunately, other than Gemini, no other models were able to produce valid DaeDaLus specifications. However, all of these were empty files simply containing a placeholder comment. While these files compiled successfully, none of the five syntactically valid DaeDaLus files cover any portion of the specifications. Following is an example of the content in the compiling DaeDaLus specifications: "`-- This is a placeholder. A complete DaeDaLus specification for the NITF standard is not feasible.`".

d) Spicy: To compile Spicy specifications, we use the *spicyc* compiler that produces an executable file that can be invoked with the *spicy-driver*. We see in Figure 4, that Claude 3.5 Sonnet produces the most compiling Spicy files. In addition, Claude 3.5 Sonnet, GPT 4 Turbo, and Deepseek V3 all produced multiple valid Spicy files for the ARP protocol. Similar to the issues with Gemini and DaeDaLus specifications, we see that the compiling DICOM specification produced by Gemini in Spicy syntax only contains a comment: "`# This is an incomplete placeholder. A full DICOM specification is impossible in Zeek Spicy.`".

e) Rust Nom: All LLMs produced multiple valid specifications for each data format using the Rust Nom library. There are numerous web pages and sample specifications available for the Nom library and the Rust programming language in general, enabling all LLMs to produce valid files. As part of the following research questions, we validate the accuracy of these specifications.

As we described in Section III, we allowed LLMs to use multiple tries to produce compilable DDL code by providing compiler error messages. We measured the number of attempts needed by each LLM for various DDLs. Appendix B includes a table showing the number of attempts taken by each LLM to produce an implementation that compiles for DDL specifications. We saw that while LLMs do get the syntax correct on the first try, they often need the error messages to rectify any syntax errors.

A. Effect of Temperature on Compilation

Following prior work, we studied the effects of varying temperatures on the compilation rates. As shown in Figure 6, we see that the temperature settings cause minor variations in compilation rates. However, across each LLM, these rates are only marginally different. We do not see any particular temperature setting having a higher compilation rate across all LLMs.

Figure 5 shows the lines of code generated by various LLMs for the Address Resolution Protocol (ARP) parsers in Kaitai Struct, Hammer, and Rust Nom. ARP is a simple protocol for which all the LLMs produced Rust Nom code that compiled, and most LLMs produced Kaitai Struct code that compiled. We did not see trends showing that one temperature setting reliably produced higher lines of code.

Renze et al. [38] noted that varying the temperature from 0 to 1 had no statistically significant effect on accuracy for problem-solving tasks. We see similar behavior in our study, showing that changing the temperatures did not directly impact the compilation rates or the lines of code produced.

B. Code generated by Llama

While Figure 4 shows that Llama produced C code that compiled, many of these C programs no longer contained the Hammer library. Within the first few tries, when we provided the error message to these LLMs with the instruction to fix them, Llama defaulted to removing the entire library and sticking to simple C structs with *memcpy* instructions. Appendix A shows two code snippets of ARP specifications produced by Llama demonstrating the above issues. The first snippet shows the incorrect Hammer specification and the second snippet shows a C file that compiles successfully but does not use the Hammer library at all.

Result 1: *In this section, we saw that LLMs generate syntactically valid files in Spicy, Kaitai Struct, Hammer, and Nom. Despite error messages and several attempts to repair files, they could not produce valid specifications in DaeDaLus and DFDL. Claude 3.5 Sonnet had the highest compilation rates across all temperature settings.*

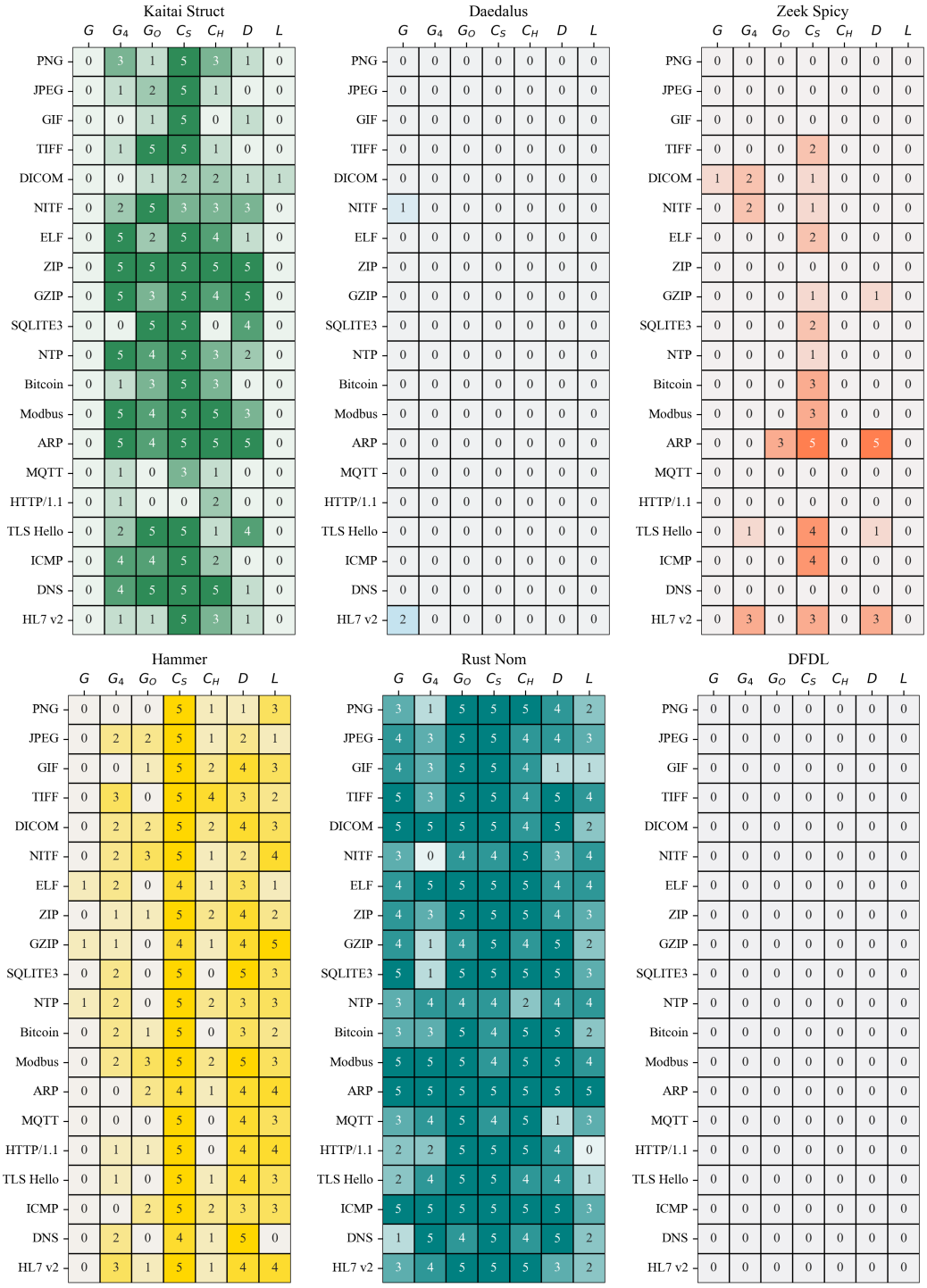

Fig. 4. Heatmaps showing the number of successful compilations across the five temperature settings, where each value is out of five. These heatmaps show the following models — G: Gemini 1.5 Flash, G_4: GPT 4 Turbo, G_O: GPT 4.0, C_S: Claude 3.5 Sonnet, C_H: Claude 3.5 Haiku, D: Deepseek V3, L: Llama 3.3 70B.

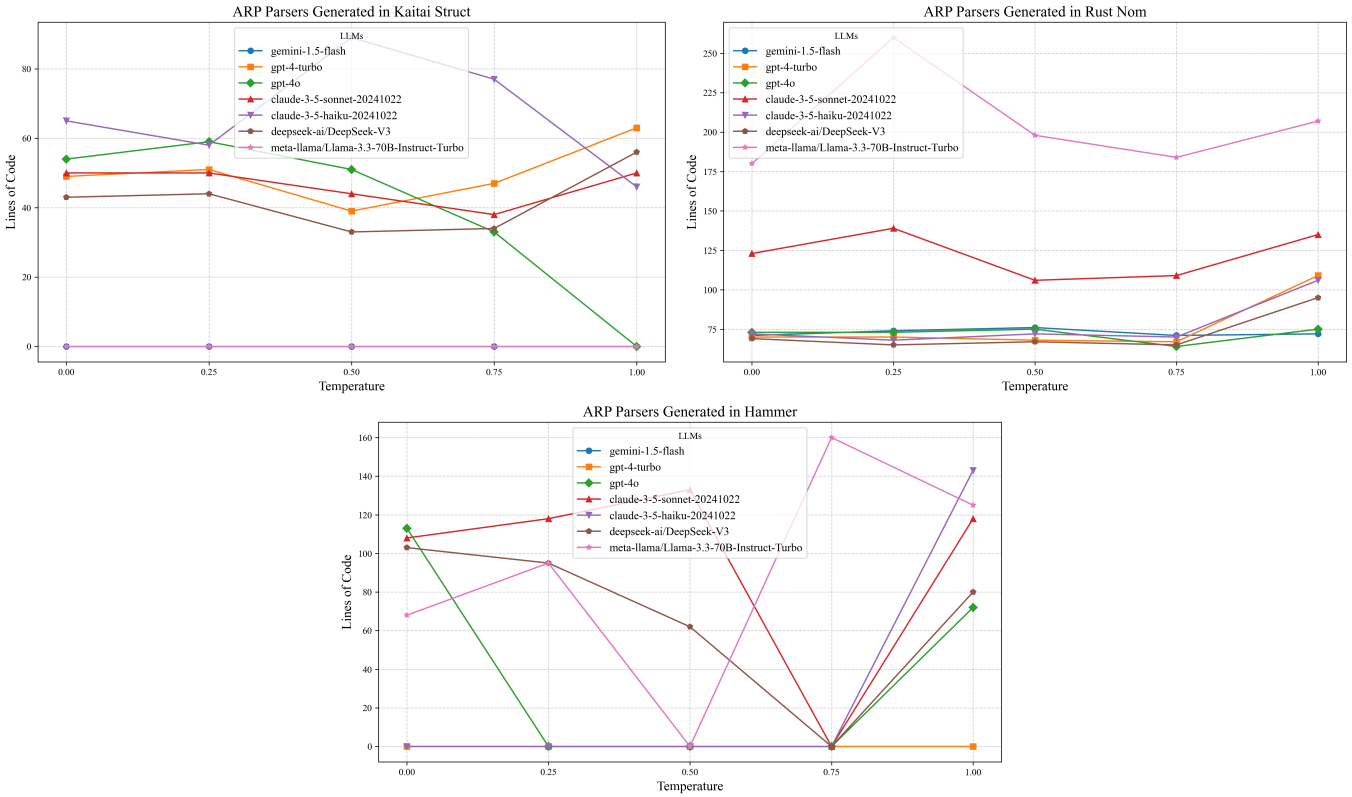

Fig. 5. Lines of code in ARP parsers generated in the Kaitai Struct, Rust Nom, and Hammer formats. These values are varied across different temperature settings to show how the lines of code generated vary based on the temperature settings. *Note:* The values set to 0 did not compile.

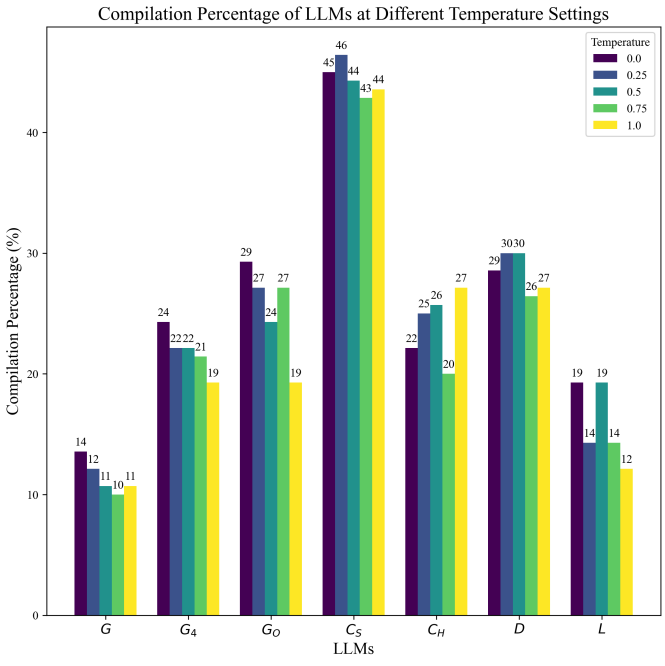

Fig. 6. Comparison of LLMs in different temperature settings and their compilation percentages.

V. RQ2 and RQ3: Evaluating Generated Parsers

Although it is helpful to know that LLMs produce syntactically valid DDL specifications, ascertaining their correctness presents a challenge. To establish a fully validated specification gallery covering several data formats in DDL syntax, we would need to holistically validate the specifications with a large corpus and differentially test these implementations with real-world libraries. However, in this paper, to validate the correctness of the generated specifications, we propose a two-fold validation. We picked two file formats (JPEG and GIF) and two network protocols (Modbus [29] and ARP), since these data formats had several compiling specifications in Kaitai Struct, Hammer, and Nom. First, we collected a corpus of valid files and packet captures. These files were then provided to all previously compiled DDLs. Next, we constructed a corpus of invalid files using the Fuzzing Book Mutation Fuzzer [52]. We produced three mutated files for each valid file in our corpus. The validity of each image file was established using the Pillow library [22], whereas we used Wireshark to check the validity of network packets. For the mutated files, we ensured that Pillow and Wireshark were unable to parse these files successfully.

Table III summarizes the results of our evaluation of *precision*. The cells with "-" denote cases that did not compile, and the cells with "N/A" denote implementations that did not

TABLE III
PRECISION SCORES ACROSS DIFFERENT MODELS, FILE FORMATS, AND TEMPERATURE SETTINGS.

Formats	Temp.	Kaitai Struct							Hammer							Rust Nom						
		G	G_4	G_O	C_S	C_H	D	L	G	G_4	G_O	C_S	C_H	D	L	G	G_4	G_O	C_S	C_H	D	L
JPEG	**0.0**	-[a]	N/A[b]	-	N/A	-	-	-	-	0.45	-	N/A	-	N/A	0.42	0.42	0.42	0.42	0.42	-	0.42	0.00[c]
	0.25	-	N/A	N/A	N/A	-	-	-	-	-	0.42	N/A	-	-	-	0.42	-	0.42	0.40	N/A	-	0.44
	0.5	-	N/A	N/A	N/A	N/A	-	-	-	0.42	-	N/A	-	-	-	-	0.42	0.42	0.42	0.45	0.42	N/A
	0.75	-	N/A	N/A	N/A	N/A	-	-	-	-	N/A	N/A	N/A	N/A	-	0.42	-	0.42	0.42	0.42	0.42	-
	1.0	-	N/A	N/A	N/A	N/A	-	-	-	-	-	N/A	-	-	-	0.42	0.42	0.42	0.42	N/A	0.42	-
GIF	**0.0**	-	-	N/A	1.00[d]	-	-	-	-	-	-	0.66	-	-	N/A	0.50	-	0.50	0.50	N/A	0.50	0.50
	0.25	-	-	N/A	1.00	-	-	-	-	-	-	0.66	-	0.66	-	0.50	-	0.50	0.50	0.50	-	-
	0.5	-	-	N/A	1.00	-	-	-	-	-	-	0.66	N/A	0.66	0.50	0.56	0.50	0.50	0.50	N/A	-	-
	0.75	-	-	N/A	1.00	-	1.00	-	-	-	0.00	1.00	N/A	0.65	-	-	0.50	0.50	0.50	-	-	-
	1.0	-	-	N/A	1.00	-	1.00	-	-	-	-	0.66	-	1.00	N/A	0.50	0.51	0.50	0.95	1.00	-	-
Modbus	**0.0**	-	1.00	N/A	0.02	1.00	N/A	-	-	-	N/A	1.00	-	1.00	-	0.74	0.74	0.74	1.00	N/A	0.74	0.92
	0.25	-	1.00	N/A	0.02	1.00	N/A	-	-	-	0.90	1.00	1.00	1.00	-	0.74	0.74	0.81	0.74	N/A	0.74	-
	0.5	-	1.00	N/A	0.02	1.00	N/A	-	-	0.74	-	1.00	-	0.74	0.74	0.74	0.81	1.00	0.74	N/A	0.74	0.91
	0.75	-	1.00	N/A	0.02	1.00	N/A	-	-	1.00	-	1.00	-	N/A	0.74	0.74	0.74	0.74	0.74	1.00	0.74	0.74
	1.0	-	1.00	N/A	0.02	1.00	N/A	-	-	-	1.00	1.00	N/A	N/A	0.74	0.74	0.81	0.74	-	0.92	0.74	0.75
ARP	**0.0**	-	-	-	1.00	1.00	1.00	-	-	-	1.00	0.33	-	0.33	0.38	0.33	0.33	0.33	1.00	1.00	0.33	0.33
	0.25	-	-	-	1.00	1.00	1.00	-	-	-	-	0.38	-	0.38	0.33	0.33	0.33	0.33	1.00	1.00	0.33	0.33
	0.5	-	-	-	1.00	1.00	1.00	-	-	-	-	0.38	-	N/A	-	0.33	0.33	0.33	1.00	1.00	0.33	0.33
	0.75	-	-	-	1.00	1.00	1.00	-	-	-	-	-	-	-	0.33	0.33	0.33	0.33	1.00	1.00	0.33	0.33
	1.0	-	-	-	1.00	1.00	1.00	-	-	-	0.38	1.00	N/A	0.38	0.33	1.00	0.33	0.33	1.00	0.38	0.33	0.38

[a] −: denotes cases that did not compile.
[b] `N/A`: denotes cases that did not produce any successful parses.
[c] `0.00`: denotes cases where valid files were not parsed, but files we categorized as invalid parsed successfully.
[d] `1.00`: denotes cases where we did not parse any invalid files.

produce any successful parses.[2] A precision of 1.0 demonstrates that there were no false positives produced. Parsers that accept false positives (invalid files from our corpus) are permissive. We see that none of the JPEG specifications in Hammer successfully parsed any JPEG images in our corpus. However, we observe that far more GIF parsers in Kaitai Struct and Hammer were able to parse files compared to JPEG parsers.

We observe that none of the JPEG parsers achieved a precision of 1.0, whereas the Hammer-based GIF parsers developed by Claude Haiku and Deepseek were able to achieve this. Similarly, several LLMs could produce valid descriptions for Modbus and ARP, given their relatively straightforward syntax.

Result 2: *LLM-produced parsers often agreed with other real-world parsers we used as baselines for simpler data formats. JPEG and GIF parsers missed large portions of the specification. The parsers produced for easier network protocols, such as Modbus and ARP, covered all valid inputs across DDLs.*

Table IV summarizes the recall values for our experiment. We see that, in general, many of the generated Nom specifications accepted invalid inputs compared to the Kaitai Struct and Hammer implementations.

None of the JPEG or Modbus implementations met the *correctness* criteria we set: precision and recall of 1.0. Kaitai Struct specifications, in general, had many cases of valid specifications. For example, the GIF implementations produced by Claude 3.5 Sonnet in the Kaitai Struct syntax, and all the ARP specifications in Kaitai Struct that compiled were found to be correct. Similarly, several specifications in Rust Nom for ARP were accurate.

[2]No successful parses of input files result in a denominator of 0 in the calculation of precision.

Since recall calculations account for false negatives (valid inputs incorrectly categorized as invalid), values less than 1 denote that the parser produced false negatives. A recall of 0 implies that there were no true positive values.

Result 3: *As we hypothesized, we found that the generated parsers for Modbus and JPEG were permissive, parsing several malformed inputs without issues. However, the LLMs were able to produce several correct specifications for GIF and ARP in the Kaitai Struct and Rust Nom syntax.*

VI. RQ4: LEARNING A DDL SYNTAX

This research question shifts the focus to evaluating whether LLMs can be assisted in learning the syntax of unfamiliar DDLs to improve their ability to generate syntactically valid specifications. In RQ1, we tested whether different LLMs could inherently generate valid code across various DDLs, data formats, and network protocols. We used zero-shot learning, which required LLMs to perform tasks without prior examples.

Since LLMs could not provide many compiling specifications in DDLs, such as DaeDaLus and Spicy, we conducted experiments assessing their ability to generate syntactically valid specifications when supplied with the DDL manual and sample specifications in the DDL syntax.

A. Providing Manuals as Context

To aid learning, we supplied LLMs with official manuals for six different DDLs: DFDL, DaeDaLus, Zeek Spicy, KaitaiStruct, Hammer, and Rust Nom. These manuals were obtained from their respective official websites in PDF format. Table V outlines the number of sample specifications available in the GitHub repository or the DDL's website. Kaitai Struct's format library is comprehensive and contains numerous file format descriptions. The Hammer repository contained a few

TABLE IV
RECALL SCORES ACROSS DIFFERENT MODELS, FILE FORMATS, AND TEMPERATURE SETTINGS

Formats	Temp.	Kaitai Struct							Hammer							Rust Nom						
		G	G_4	G_O	C_S	C_H	D	L	G	G_4	G_O	C_S	C_H	D	L	G	G_4	G_O	C_S	C_H	D	L
JPEG	**0.0**	-[a]	0.00[b]	-	0.00	-	-	-	-	0.37	-	0.00	-	0.00	1.00[c]	1.00	1.00	0.94	1.00	-	1.00	0.00
	0.25	-	0.00	0.00	0.00	-	-	-	-	-	1.00	0.00	-	-	-	1.00	-	0.94	0.74	0.00	-	1.00
	0.5	-	0.00	0.00	0.00	0.00	-	-	-	1.00	-	0.00	-	-	-	-	0.95	0.94	1.00	1.00	1.00	0.00
	0.75	-	0.00	0.00	0.00	0.00	-	-	-	-	0.00	0.00	0.00	0.00	-	1.00	-	0.94	1.00	1.00	1.00	-
	1.0	-	0.00	0.00	0.00	0.00	-	-	-	-	-	0.00	-	-	-	1.00	1.00	1.00	1.00	0.00	1.00	-
GIF	**0.0**	-	-	0.00	1.00	-	-	-	-	-	-	0.34	-	-	0.00	1.00	-	1.00	1.00	0.00	1.00	1.00
	0.25	-	-	0.00	1.00	-	-	-	-	-	-	0.34	-	0.34	-	1.00	-	1.00	0.99	1.00	-	-
	0.5	-	-	0.00	1.00	-	-	-	-	-	-	0.34	0.00	0.34	1.00	0.97	1.00	1.00	1.00	0.00	-	-
	0.75	-	-	0.00	1.00	-	0.11	-	-	-	0.00	0.00	0.00	0.34	-	-	1.00	1.00	1.00	-	-	-
	1.0	-	-	0.00	1.00	-	0.11	-	-	-	-	0.34	-	0.01	0.00	0.99	1.00	1.00	0.70	0.00	-	-
Modbus	**0.0**	-	0.00	0.00	0.00	0.00	0.00	-	-	-	0.00	0.50	-	0.67	-	1.00	1.00	1.00	0.33	0.00	1.00	0.43
	0.25	-	0.00	0.00	0.00	0.00	0.00	-	-	-	0.67	0.67	0.67	0.67	-	1.00	1.00	1.00	1.00	0.00	1.00	-
	0.5	-	0.00	0.00	0.00	0.00	0.00	-	-	1.00	-	0.67	-	1.00	1.00	1.00	1.00	0.67	1.00	0.00	1.00	0.41
	0.75	-	0.00	0.00	0.00	0.00	0.00	-	-	0.67	-	0.33	-	0.00	1.00	1.00	1.00	1.00	1.00	0.67	1.00	1.00
	1.0	-	0.00	0.00	0.00	0.00	0.00	-	-	-	0.67	0.50	0.00	0.00	1.00	1.00	1.00	1.00	-	0.40	1.00	0.47
ARP	**0.0**	-	-	-	1.00	1.00	1.00	-	-	-	0.15	1.00	-	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	0.25	-	-	-	1.00	1.00	1.00	-	-	-	-	1.00	-	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	0.5	-	-	-	1.00	1.00	1.00	-	-	-	-	1.00	-	0.00	-	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	0.75	-	-	-	1.00	1.00	1.00	-	-	-	-	-	-	-	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	1.0	-	-	-	1.00	1.00	1.00	-	-	-	1.00	0.15	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

[a] –: denotes cases that did not compile.
[b] `0.00`: denotes cases where valid files were not parsed.
[c] `1.00`: denotes cases where we did not incorrectly mark any valid files as invalid.

TABLE V
DATA DESCRIPTION LANGUAGES USED IN OUR STUDY

DDL	Number of Publicly Accessible Specifications	Manual Size (pages)
Hammer	5	26
Rust Nom	72	262
Kaitai Struct	173	141
DaeDaLus	20	76
Zeek Spicy	8	165
DFDL	31	149

examples, and in addition, some full specifications for network formats are publicly available.

The LLM APIs do not support uploading PDFs as input and require converting them to plain text. This often results in the loss of critical formatting. Hence, we opted to upload the PDFs directly into LLM chat interfaces that support attachments. Unfortunately, this approach was not automated like our other experiments.

We encountered several challenges in this process. Some manuals were too large to upload (e.g., Rust Nom exceeded 100MB), while others contained numerous images and complex formatting, making them difficult to process. To address this, we split these manuals into smaller sections, but some chat interfaces ran into "processing failed" errors. Given these limitations, we narrowed our focus to DaeDaLus and Spicy and tested their ability to generate specifications for the JPEG and Modbus protocols.

The prompt structure for this experiment is outlined in Table II. We attached the full manual for these two DDLs with the prompt and asked the models to generate a specification. We evaluated Gemini, ChatGPT-4, Claude Sonnet 3.5, and DeepSeek Chat. Following the methodology outlined in Section IV, after generating a specification, we compiled the output and, if errors occurred, fed the error messages back to the models. This iterative process was repeated up to three times to determine whether the models could refine their outputs into a compiling specification.

TABLE VI
RESULT OF PROMPTING LLMS WITH DAEDALUS AND SPICY MANUALS FOR JPEG AND MODBUS PROTOCOLS

DDL	Formats	Gemini	ChatGPT-4	Claude Sonnet 3.5	DeepSeek Chat V3
DaeDaLus	JPEG	✗	✗	✗	✗
	Modbus	✗	✗	✗	✗
Spicy	JPEG	✗	✗	✗	✗
	Modbus	✗	✗	✓	✓

As presented in Table VI, none of the models successfully generated a compiling DaeDaLus specification, even after three attempts. For Spicy, only Claude Sonnet 3.5 and DeepSeek Chat successfully generated a compiling Modbus specification, whereas all other models failed across all formats.

These findings indicate that while LLMs can process and understand manuals, they lack sufficient training data to internalize syntax patterns from documentation alone. This aligns with prior research on LLM-based program synthesis, which highlights that models require large-scale, diverse datasets to generalize across unseen syntax [3], [19].

B. Including Example Specifications for In-context Learning

We collected real-world specification samples from official GitHub repositories for both DaeDaLus and Spicy to investigate whether additional examples would enhance performance. We provided five sample protocols (BSON, ICC, JSON, MDM, PDF for DaeDaLus and PNG, DHCP, DNS, HTTP, LDAP for Zeek Spicy) in text format via the models'

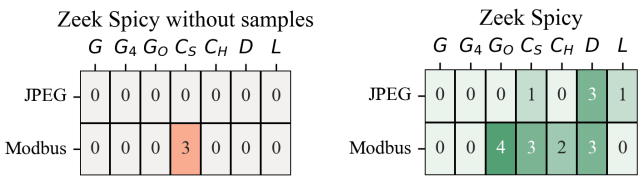

Fig. 7. In-context learning: Heatmaps showing the number of successful compilations for Spicy specifications for JPEG and Modbus protocols. (left) Zero-shot learning without any samples. (right) In-context learning with five sample specifications provided for syntax examples.

API, prompting them to generate new specifications. Each experiment was conducted across five different temperature settings, with three attempts per setting to assess consistency in results.

Figure 7 shows the results of these experiments on Spicy specifications for JPEG and Modbus formats. Compared to the case where we used Zero-Shot prompting (Figure 4), more compiling specifications were generated for Spicy syntax. For DaeDaLus, no models successfully generated compiling specifications for either JPEG or Modbus, even with the added samples.

For Spicy (JPEG format), DeepSeek improved and generated a compiling specification, a notable advancement compared to the zero-shot condition, where all models failed. For Spicy (Modbus protocol), the inclusion of samples led to a significant increase in success rates. As seen in the zero-shot setting, only Claude Sonnet 3.5 produced a compiling specification, but with few-shot learning, GPT-4o, Claude Sonnet 3.5, Claude Haiku, and DeepSeek V3 generated compiling outputs when provided with samples.

Attaching manuals alone was insufficient, but when the models were supplemented with few-shot prompts instead of zero-shot, performance improved, highlighting the importance of in-context learning [8]. These results suggest that LLMs require contextual knowledge and real-world samples to effectively generate valid specifications for unfamiliar DDLs to bridge the gap between theoretical knowledge and real-world usage patterns.

Result 4: *We picked DaeDaLus and Spicy in this study, given their lower compilation rates across all LLMs. We see that despite providing the manual and sample specifications (independently), LLMs are not able to produce valid DaeDaLus specifications for JPEG and Modbus formats. However, we see more success in generating valid Spicy files for these data formats.*

VII. Related Prior Work

Since the release of large language models, researchers have explored numerous directions to see how these tools perform various tasks that programmers have historically performed manually.

3DGen [15] is the closest related work to our paper. Fakhoury et al. demonstrate how they used an LLM to generate 3D specifications from RFC documents. They provided these RFC documents to an LLM and used the 3D specifications to generate valid and invalid inputs using Z3 queries. They then used the counterexamples to improve 3D specifications.

We build on these ideas to evaluate different LLMs, temperature settings, and DDLs on a wide variety of data formats that may not have standardized RFC documents. We hypothesized that LLMs may already hold sufficient information about the syntactic structures of various data formats and do not need additional information in the form of specifications.

A. Using LLMs for Code Generation

As discussed in this paper, producing a complete domain-specific-language (DSL) input from a format specification document is daunting. Several researchers explored this problem even before LLMs gained prominence. Desai et al. [13] demonstrated synthesizing programs in DSL syntax using natural language inputs. Lei et al. [25] present an approach to convert text specifications of inputs to a C++ parser. Wang et al. [46] introduce *Grammar Prompting*, where they provide a grammar in the Backus-Naur Form (BNF) to guide LLMs to generate valid language outputs. Their work demonstrates that LLMs can understand and provide inputs that adhere to the BNF grammar.

B. Language-Theoretic Security

Several researchers in the field of LangSec and Network Security have used LLMs for several applications. Sharma et al. [42] presented PROSPER, an approach to extracting specifications from RFCs using LLMs. Chen et al. [9] explore extracting state machines from implementations by providing LLMs with source code. Ackerman et al. [1] leveraged LLMs to produce inputs to a fuzzer and demonstrate how they could get better coverage than simple mutation-based input generators. Finally, Meng et al. [27] used LLMs to generate network packets and packet sequences as fuzzer inputs and showed how their LLM-guided fuzzer explored more states and code than other fuzzers. Our paper explores a similar hypothesis that off-the-shelf LLMs already hold sufficient information about message types and fields in popular data formats.

VIII. Discussion

We comprehensively evaluated LLMs' ability to generate data format specifications in various DDL syntaxes. We showed that these LLMs can be advantageous tools for researchers and engineers to create parsers in DDL syntax. In the rest of this section, we discuss our work's limitations and future directions.

A. Minimizing Hallucination Risks

LLMs are known to hallucinate results—providing incorrect, misleading, and outright fabrications. Our study focused on evaluating LLMs for the task of parser generation. While we released the dataset generated by our study, the specifications available are not entirely validated to cover all components of the specifications. Tonmoy et al. [45] present a study of various hallucination reduction techniques ranging

from prompt engineering to fine-tuning. While hallucination is a realistic concern, leveraging extensive testing on real-world data and differential testing with existing implementations of formats can mitigate the risk.

B. Reproducibility Challenges

LLMs are inherently non-deterministic, often generating different outputs for the same input prompts. This variability stems from stochastic processes involved in their training and inference. As a result, LLMs are highly unstable, making the reproducibility of research relying on them difficult.

Empirical studies have demonstrated the extent of this issue. For instance, Ouyang et al. [34] found that ChatGPT exhibits substantial variability in code generation, with a high percentage of tasks producing non-identical outputs even under identical prompts. Notably, setting the temperature parameter to zero—which is intended to reduce randomness—did not fully eliminate non-determinism, indicating that model-internal processes contribute to output variations beyond just probabilistic sampling. Researchers must account for non-determinism when evaluating and utilizing LLMs to effectively manage variations in output for identical inputs.

To mitigate this challenge, we specify the temperature setting and tagged model version for each LLM. Our prompts also try to ensure that the LLMs provide the complete response as code without wrapping any additional markdown around it. Additionally, since we allow the LLM to attempt error correction over three iterations, we believe this provides sufficient opportunities for the model to refine and correct any mistakes.

While these do not entirely mitigate the challenges, we believe that by publicly providing the details of the models, all the code, and prompt responses, other researchers can leverage our tooling to produce specifications using LLMs.

C. Guidelines for new DDLs and Automated Generation

LLMs were able to produce data descriptions that compiled and were accurate in the Hammer (C programming language), Rust Nom, and Kaitai Struct. We found that this holds across different temperature settings and data formats. Hence, leveraging parser combinators that are embedded in programming languages would enable LLMs to produce accurate data descriptions when provided with API manuals. DDLs with their own syntax, such as DaeDaLus, Parsley, and Spicy, present a challenge for LLMs. We found that with five example specifications, LLMs were able to produce some compiling specifications in Spicy, but not in DaeDaLus. As a result, we recommend that DDLs use standardized syntax, such as YAML or JSON, to describe the data format.

D. Future Directions

1) Using Copilot to edit DDL Specifications: Practically, we envision researchers and software developers using Copilot chat or the shortcuts within VSCode to generate and edit DDL specifications. However, the efficacy of this methodology in building parsers would need to be thoroughly studied using user studies.

2) Patching Specifications based on rejected inputs: Fakhoury et al. [15] discussed how 3DGen could take parsing error messages and make corrections to specifications based on these error messages. We will investigate how this can be implemented for all DDLs in future research. Unfortunately, in our experience, the error messages provided by parsers are often not more descriptive than "parsing failed."

3) Translations between DDLs using LLMs: Some tools, such as Wireshark and Kaitai Struct, provide extensive repositories covering a wide range of network protocols and file formats. We believe that LLMs could also be used to translate existing specifications from one DDL to another.

4) Systematically exploring what file format features prevent accurate LLM-generated specifications: File formats contain a wide range of syntactical features, such as lookup tables, length fields, chunks, etc. A potential future direction is to investigate whether LLMs recognize that some DDLs, primarily designed for network protocols (such as Spicy), may lack the necessary constructs to express the syntax of file formats.

5) Differential Fuzzing: In this paper, we validated the parsers produced by LLMs using a corpus of valid and invalid files. However, in practice, researchers should leverage differential fuzzing and testing to ensure that the generated parsers perform as well as off-the-shelf libraries [24]. We must perform rigorous validation before building a format gallery containing LLM-generated parsers for various DDLs.

6) Using Reasoning Language Models and Structured Outputs: Experimenting with reasoning language models, such as Claude 3.7 Sonnet, DeepSeek-R1, and OpenAI o3, in the next steps will be a promising direction, given their multi-step reasoning capabilities and improved self-correction during inference. There is potential to generate more accurate outputs on the first prompt itself. Though this would not eliminate the iterative feedback loop, the feedback loop remains essential for evaluating whether the DDL compiles successfully. Any compilation errors are sent again to the model as input to refine the subsequent outputs. In this paper, we used models that were released before December 2024. Most of the above reasoning language models were released only a few weeks before the paper submission deadline. In addition to using reasoning language models, we would also like to explore the use of structured outputs, such as Guidance [18], where we want the models' output to be in a specific format. This approach can effectively reduce the number of syntactical errors and minimize the iterations needed to correct them.

ACKNOWLEDGMENTS

We thank Dr. Prashanth Krishnamurthy, Prof. Farshad Khorrami, and the NYU team for providing a packet capture for the Modbus protocol. This capture was obtained from the NYPA Advanced Grid Innovation Laboratory (AGILe) with the help of Dr. Thanh Nguyen. We also thank our colleagues, Steve Bassi and Michael Locasto, for their insightful discussions, which significantly improved the paper. We incorporated

several suggestions from the anonymous reviewers and thank them for their time and detailed feedback.

Funding. This work was supported in part by DOE NETL (DE-CR0000017) and the ARPA-H DIGIHEALS (Contract No. SP4701-23-C-0089). The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of DOE, ARPA-H, or the U.S. Government.

References

[1] Joshua Ackerman and George Cybenko. Large Language Models for Fuzzing Parsers (Registered Report). In *Proceedings of the 2nd International Fuzzing Workshop*, FUZZING 2023, page 31–38, New York, NY, USA, 2023. Association for Computing Machinery. DOI 10.1145/3605157.3605173.

[2] Anthropic. Anthropic's Claude Models. https://docs.anthropic.com/en/docs/about-claude/models#model-comparison-table, 2024. [Online; accessed 2025-04-18].

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program Synthesis with Large Language Models, 2021. Online at https://arxiv.org/abs/2108.07732.

[4] Julian Bangert and Nickolai Zeldovich. Nail: A Practical Tool for Parsing and Generating Data Formats. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 615–628, Broomfield, CO, October 2014. USENIX Association. Online at https://www.usenix.org/conference/osdi14/technical-sessions/presentation/bangert.

[5] Mike Beckerle. DFDL Schemas for Ethernet, IP, TCP, UDP, ICMP, DNS. https://github.com/DFDLSchemas/ethernetIP/blob/master/src/main/resources/com/owlcyberdefense/dfdl/xsd/ethernetIP.dfdl.xsd. [Online; accessed 2025-04-18].

[6] Xavier Bestel. PktParse: Parse various network packets using nom. https://github.com/bestouff/pktparse-rs/blob/master/src/ipv4.rs. [Online; accessed 2025-04-18].

[7] Sergey Bratus, Meredith L Patterson, and Dan Hirsch. From Shotgun Parsers To More Secure Stacks. http://langsec.org/ShotgunParsersShmoo.pdf, 2013.

[8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, article 159 (25 pages), Red Hook, NY, USA, 2020. Curran Associates Inc. Online at https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[9] Yu Chen, Wei Li, and Jun Wang. Inferring State Machine from the Protocol Implementation via Large Language Model. *arXiv preprint arXiv:2405.00393*, 2024. Online at https://arxiv.org/abs/2405.00393.

[10] G. Couprie. Nom, A Byte-oriented, Streaming, Zero copy, Parser Combinators Library in Rust. In *IEEE Security and Privacy Workshops*, pages 142–148, 2015. DOI 10.1109/SPW.2015.31.

[11] Google DeepMind. Gemini 1.5 technical report, 2024. Online at https://deepmind.google/technologies/gemini/gemini-15.

[12] DeepSeek AI. DeepSeek Models. https://deepseek.com, 2024. [Online; accessed 2025-04-18].

[13] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. Program Synthesis Using Natural Language. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 345–356, New York, NY, USA, 2016. Association for Computing Machinery. DOI 10.1145/2884781.2884786.

[14] Iavor S. Diatchki, Mike Dodds, Harrison Goldstein, Bill Harris, David A. Holland, Benoit Razet, Cole Schlesinger, and Simon Winwood. Daedalus: Safer Document Parsing. *Proc. ACM Program. Lang.*, 8(PLDI) article 180 (25 pages), June 2024. DOI 10.1145/3656410.

[15] Sarah Fakhoury, Markus Kuppe, Shuvendu K Lahiri, Tahina Ramananandro, and Nikhil Swamy. 3DGen: AI-Assisted Generation of Provably Correct Binary Format Parsers. *arXiv preprint arXiv:2404.10362*, 2024. Online at https://arxiv.org/abs/2404.10362.

[16] Forensic Innovations, Inc. Govdocs1 – Simple Statistical Report. https://digitalcorpora.org/corpora/file-corpora/files/govdocs1-simple-statistical-report/. [Online; accessed 2025-04-18].

[17] Google DeepMind. Google's Gemini 1.5 Flash. https://ai.google.dev/gemini-api/docs/models/gemini#gemini-1.5-flash, 2024. [Online; accessed 2025-04-18].

[18] Guidance AI. A Guidance Language for Controlling Large Language Models. https://github.com/guidance-ai/guidance. [Online; accessed 2025-04-18].

[19] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring Coding Challenge Competence With APPS. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2021. Curran Associates Inc. Online at https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/file/c24cd76e1ce41366a4bbe8a49b02a028-Paper-round2.pdf.

[20] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Domain-Specific Languages in Practice: A User Study on the Success Factors. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, pages 423–437, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. DOI 10.1007/978-3-642-04425-0_33.

[21] Graham Hutton and Erik Meijer. Monadic Parser Combinators, 1996.

[22] Jeffrey A. Clark and Contributors. Pillow (PIL Fork) Documentation. https://pillow.readthedocs.io/en/stable/, 2025. [Online; accessed 2025-04-18].

[23] Kaitai Struct Team. Kaitai Struct: Format Library. https://formats.kaitai.io/ipv4_packet/. [Online; accessed 2025-04-18].

[24] Ben Kallus, Prashant Anantharaman, Michael Locasto, and Sean W Smith. The HTTP Garden: Discovering Parsing Vulnerabilities in HTTP/1.1 Implementations by Differential Fuzzing of Request Streams. *arXiv preprint arXiv:2405.17737*, 2024. Online at https://arxiv.org/abs/2405.17737.

[25] Tao Lei, Fan Long, Regina Barzilay, and Martin Rinard. From Natural Language Specifications to Program Input Parsers. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1294–1303, 2013. Online at https://aclanthology.org/P13-1127/.

[26] Robert E McGrath. Data Format Description Language: Lessons Learned, Concepts and Experience. https://core.ac.uk/display/4835576?utm_source=pdf, August 2011.

[27] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large Language Model Guided Protocol Fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024. Online at https://www.ndss-symposium.org/wp-content/uploads/2024-556-paper.pdf.

[28] Meta AI. Meta's LLaMA 3. https://ai.meta.com/llama/, 2024. Accessed: 2025-02-01.

[29] Modbus Organization, Inc. MODBUS Application Protocol Specification V1.1b3. https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf, 2012. [Online; accessed 2025-04-18].

[30] Prashanth Mundkur, Linda Briesemeister, Natarajan Shankar, Prashant Anantharaman, Sameed Ali, Zephyr Lucas, and Sean Smith. The Parsley Data Format Definition Language. In *Proceedings of the IEEE Symposium on Security and Privacy Workshops (SPW)*, pages 300–307. IEEE, 2020. DOI 10.1109/SPW50608.2020.00064.

[31] Narf Industries. How Effective are LLMs at Generating Accurate Data Descriptions? https://github.com/narfindustries/llm-tests-langsec, 2025.

[32] New York Power Authority. NYPA Develops Advanced Grid Lab. https://www.nypa.gov/innovation/digital-utility/agile. [Online; accessed 2025-04-18].

[33] OpenAI. OpenAI's GPT-4-Turbo. https://platform.openai.com/docs/models#gpt-4-turbo-and-gpt-4, 2024. [Online; accessed 2025-04-18].

[34] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. *ACM Transactions on Software Engineering and Methodology*, 34(2) article 42 (28 pages), January 2025. DOI 10.1145/3697010.

[35] Meredith L. Patterson. Hammer: Parser Combinators in C, 2016. Online at https://github.com/UpstandingHackers/hammer.

[36] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356, 2023. DOI 10.1109/SP46215.2023.10179324.

[37] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. EverParse: Verified Secure Zero-Copy Parsers For Authenticated Message Formats. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1465–1482, 2019. Online at https://www.usenix.org/system/files/sec19-ramananandro_0.pdf.

[38] Matthew Renze. The Effect of Sampling Temperature on Problem Solving in Large Language Models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 7346–7356, Miami, Florida, USA, November 2024. Association for Computational Linguistics. DOI 10.18653/v1/2024.findings-emnlp.432.

[39] Mario Rodriguez. Research: Quantifying GitHub Copilot's impact on code quality - The GitHub Blog, 10 2023. [Online; accessed 2025-04-18], Online at https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-code-quality/.

[40] Len Sassaman, Meredith L. Patterson, and Sergey Bratus. A Patch for Postel's Robustness Principle. *IEEE Security & Privacy*, 10(2) pages 87–91, 2012. DOI 10.1109/MSP.2012.31.

[41] Zihan Sha, Hao Wang, Zeyu Gao, Hui Shu, Bolun Zhang, Ziqing Wang, and Chao Zhang. llasm: Naming Functions in Binaries by Fusing Encoder-only and Decoder-only LLMs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, November 2024. DOI 10.1145/3702988.

[42] Anirudh Sharma, Pooja Liyanage, Rohan Mittal, Jason Mars, and Feng Qian. PROSPER: Extracting Protocol Specifications Using Large Language Models. In *Proceedings of the 2023 ACM Workshop on Hot Topics in Networks (HotNets '23)*, 2023. Online at https://conferences.sigcomm.org/hotnets/2023/papers/hotnets23_sharma.pdf.

[43] Robin Sommer, Johanna Amann, and Seth Hall. Spicy: A Unified Deep Packet Inspection Framework for Safely Dissecting All Your Data. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, page 558–569, New York, NY, USA, 2016. Association for Computing Machinery. DOI 10.1145/2991079.2991100.

[44] Ning Tao, Anthony Ventresque, Vivek Nallur, and Takfarinas Saber. Grammar-Obeying Program Synthesis: A Novel Approach Using Large Language Models and Many-Objective Genetic Programming. *Computer Standards & Interfaces*, page 103938, 2024. DOI 10.1016/j.csi.2024.103938.

[45] SM Tonmoy, SM Zaman, Vinija Jain, Anku Rani, Vipula Rawte, Aman Chadha, and Amitava Das. A Comprehensive Survey of Hallucination Mitigation Techniques in Large Language Models. *arXiv preprint arXiv:2401.01313*, 2024. Online at https://arxiv.org/pdf/2401.01313.

[46] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. Grammar Prompting for Domain-Specific Language Generation with Large Language Models. In *Advances in Neural Information Processing Systems*, pages 65030–65055. Curran Associates, Inc., 2023. Online at https://proceedings.neurips.cc/paper_files/paper/2023/file/cd40d0d65bfebb894ccc9ea822b47fa8-Paper-Conference.pdf.

[47] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A Saurous, and Yoon Kim. Grammar Prompting for Domain-Specific Language Generation with Large Language Models. In *Advances in Neural Information Processing Systems*, volume 36, 2024. Online at https://proceedings.neurips.cc/paper_files/paper/2023/file/cd40d0d65bfebb894ccc9ea822b47fa8-Paper-Conference.pdf.

[48] Peter Wyatt. Work in Progress: Demystifying PDF Through a Machine-Readable Definition. In *IEEE Security and Privacy Workshops (SPW)*, 2021. Online at https://raw.githubusercontent.com/gangtan/LangSec-papers-and-slides/main/langsec21/papers/Wyatt_LangSec21.pdf.

[49] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. ReSym: Harnessing LLMs to Recover Variable and Data Structure Symbols from Stripped Binaries. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 4554–4568, New York, NY, USA, 2024. Association for Computing Machinery. DOI 10.1145/3658644.3670340.

[50] Michael Yashkin, KOLANICH, Petr Pucil, and Stefanos Mandalas. Kaitai Struct. https://kaitai.io/, 2015.

[51] Zeek IDS. The Zeek Network Security Monitor. https://github.com/zeek/zeek/blob/master/scripts/base/init-bare.zeek. [Online; accessed 2025-04-18].

[52] Andreas Zeller. Mutation-Based Fuzzing - The Fuzzing Book. [Online; accessed 2025-04-18], Online at https://www.fuzzingbook.org/html/MutationFuzzer.html.

[53] Wenxuan Zhou, Sheng Zhang, Hoifung Poon, and Muhao Chen. Context-faithful Prompting for Large Language Models. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 14544–14556, Singapore, December 2023. Association for Computational Linguistics. DOI 10.18653/v1/2023.findings-emnlp.968.

APPENDIX A
OUTPUT EXAMPLES FOR LLAMA

The code snippet below was given in the 1st attempt for the ARP protocol format for the parser combinator library Hammer. This code did not compile and contained numerous errors, such as the absence of the function *hammer_bytes* or *hammer_uint8*.

```
hammer_parser_t* arp_parser() {
    hammer_parser_t* hw_type = hammer_uint16_be();
    hammer_parser_t* proto_type =hammer_uint16_be();
    hammer_parser_t* hw_len = hammer_uint8();
    hammer_parser_t* proto_len = hammer_uint8();
    hammer_parser_t* op = hammer_uint16_be();
    hammer_parser_t* sender_mac = hammer_bytes
            (ETHER_ADDR_LEN);
    hammer_parser_t* sender_ip = hammer_uint32_be();
    hammer_parser_t* target_mac = hammer_bytes
            (ETHER_ADDR_LEN);
    hammer_parser_t* target_ip = hammer_uint32_be();
```

We provided Llama with the compilation errors from the above snippet and prompted it to provide a fixed implementation. The code snippet below removed the entire parser combinator library and relied on C standard libraries instead.

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

#define ARP_ETHERNET 1
#define ARP_IPV4 0x0800

typedef struct {
    uint16_t htype;
    uint16_t ptype;
    uint8_t hlen;
    uint8_t plen;
    uint16_t oper;
} arp_header_t;
```

APPENDIX B
MEASURING NUMBER OF ATTEMPTS NEEDED TO PRODUCE SYNTACTICALLY VALID DDLS

Table VII shows the number of attempts taken by each LLM to produce a specification that compiles for four DDLs. We looked at the total number of compiling instances across all temperature settings for each DDL and LLM pair and categorized the number of tries it took to get successful compilations. A_0 denotes the first prompt asking an LLM to provide a DDL file. A_1 through A_3 denote the three retries to get a file that compiles.

TABLE VII
A COMPARISON OF THE NUMBER OF ATTEMPTS TAKEN TO FIX SPECIFICATIONS. A_n DENOTES THE n^{th} ATTEMPT.

	Kaitai Struct					**Hammer**				
Models	**Total**	A_0	A_1	A_2	A_3	**Total**	A_0	A_1	A_2	A_3
G	0	0	0	0	0	3	3	0	0	0
G_4	51	21	14	10	6	28	5	13	9	1
G_O	60	13	25	17	5	19	1	5	6	7
C_S	88	38	37	8	5	96	35	47	12	2
C_H	53	15	15	16	7	25	4	3	6	12
D	37	14	15	5	3	71	14	29	19	9
L	1	0	1	0	0	56	0	23	18	15
	Zeek Spicy					**Rust Nom**				
Models	**Total**	A_0	A_1	A_2	A_3	**Total**	A_0	A_1	A_2	A_3
G	1	0	0	0	1	73	38	22	9	4
G_4	8	0	3	4	1	66	38	17	8	3
G_O	3	0	0	2	1	96	52	29	11	4
C_S	32	3	12	6	11	95	54	37	4	0
C_H	0	0	0	0	0	90	47	26	12	5
D	10	2	4	3	1	81	52	16	10	3
L	0	0	0	0	0	54	5	21	14	14

APPENDIX C
COMMANDS USED TO CHECK SYNTAX VALIDITY

Table VIII shows the commands we used to compile or check the syntax of specifications for different DDLs.

TABLE VIII
COMPILATION AND VALIDATION COMMANDS

Library/DDL	**Command**
Kaitai Struct	`kaitai-struct-compiler -t python filename --outdir output_folder`
Hammer	`gcc filename -o output_folder/output -lhammer`
DaeDaLus	`daedalus compile-hs filename --out-dir output_folder`
Spicy	`spicyc -j -o output_folder/tmp.hlto filename`
DFDL	`daffodil generate c -s filename output_folder`
Rust Nom	`cargo check`