3DGen: AI-Assisted Generation of Provably Correct Binary Format Parsers

Sarah Fakhoury, Markus Kuppe, Shuvendu K. Lahiri, Tahina Ramananandro and Nikhil Swamy
Microsoft Research, Redmond, USA
{sfakhoury, makuppe, shuvendu, taramana, nswamy}@microsoft.com

***Abstract*—Improper parsing of attacker-controlled input is a leading source of software security vulnerabilities, especially when programmers transcribe informal format descriptions in RFCs into efficient parsing logic in low-level, memory unsafe languages. Several researchers have proposed formal specification languages for data formats from which efficient code can be extracted. However, distilling informal requirements into formal specifications is challenging and, despite their benefits, new, formal languages are hard for people to learn and use.**

In this work, we present 3DGen, a framework that makes use of AI agents to transform mixed informal input, including natural language documents (i.e., RFCs) and example inputs into format specifications in a language called 3D. To support humans in understanding and trusting the generated specifications, 3DGen uses symbolic methods to also synthesize test inputs that can be validated against an external oracle. Symbolic test generation also helps in distinguishing multiple plausible solutions. Through a process of repeated refinement, 3DGen produces a 3D specification that conforms to a test suite, and which yields safe, efficient, provably correct, parsing code in C.

We have evaluated 3DGen on 20 Internet standard formats, demonstrating the potential for AI-agents to produce formally verified C code at a non-trivial scale. A key enabler is the use of a domain-specific language to limit AI outputs to a class for which automated, symbolic analysis is tractable.

***Index Terms*—Code Generation, Agentic AI Systems, Trustworthy AI programming**

I. Introduction

Improper parsing of attacker-controlled input is a leading source of software security vulnerabilities,[1][2] especially when programmers transcribe informal format descriptions into efficient parsing logic in low-level, memory unsafe languages. For example, the format of TCP headers is specified in natural language and packet diagrams in the classic RFCs 793 and 9293; meanwhile, *tcp_input.c*, the TCP header parser in the Linux kernel was patched to prevent an out of bounds access in 2019, after being in the kernel for nearly 20 years.

In response, researchers have proposed languages for describing low-level binary message formats backed by code generators that yield parsing and serialization tools, e.g., Nail [1] and EverParse [2], [3]. EverParse is notable in that it produces formally verified C code from a format description language (called 3D), guaranteeing memory safety, functional correctness, and double-fetch freedom.

In an ideal world, one might hope for specifications to always be written in domain-specific languages (DSLs) like 3D that yield trustworthy executable code. However, more commonly, specifications are not entirely formal and come from a variety of sources, ranging from natural language documents, diagrams, example code snippets, sample input/output pairs, etc. Extracting a formal specification from such a variety of sources requires a significant human effort, typically requiring a process that involves:

1) Learning a new DSL;
2) Understanding the informal specification;
3) Expressing one's understanding of the informal specification in the DSL;
4) Iterating to refine intent, revisiting the previous steps to arrive at a desired specification.

This is challenging enough that developers often directly transcribe informal specifications into executable code in general purpose programming languages, leaving the door open to low-level coding errors that lead to security vulnerabilities.

A. 3DGen*: A Framework for AI-assisted DSL Programming*

In this work, we present 3DGen, a framework that uses AI agents to assist a human in translating an informal specification to executable code via a DSL, grounded specifically in generating binary format parsers using 3D. Our framework is agnostic to the AI model used, though for our experiments we use GPT-4 [4]. The core of 3DGen is an automated intent-refinement loop which assists a user in constructing a 3D specification that matches an oracle's behavior on a set of test inputs. Figure 1 sketches the high-level workflow, whose main elements mirror the steps outlined above.

1. Teaching a DSL to an agent: 3D is a small language whose syntax is based on C's syntax for typedefs, structures, and unions. Its manual is relatively compact, consisting of around 2,000 lines of text and around 20 examples. We "teach" 3DGen about 3D by giving it access to the manual, and the ability to query parts of the manual based on techniques that we describe in §III-B.

2 & 3. Digesting informal specifications into 3D: A user gathers a collection of informal specifications, including natural language documents that describe message formats and sample test inputs, and presents it to 3DGen. In turn, our framework, prompts the underlying agents to generate a 3D specification.

4. Refining intent The 3D compiler analyzes candidate 3D specifications, providing syntax and type errors that we feed back to the agents to repair their code until the produced 3D

[1] https://cwe.mitre.org/data/definitions/20.html
[2] https://cwe.mitre.org/data/definitions/502.html

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00173

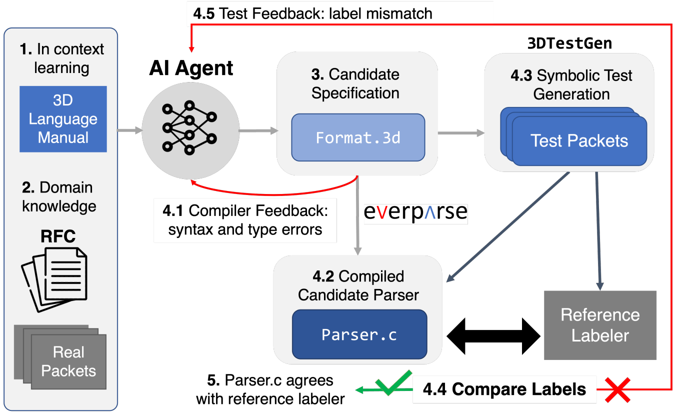

4.5 Test Feedback: label mismatch
1. In context learning
3D Language Manual
2. Domain knowledge
RFC
Real Packets
AI Agent
3. Candidate Specification
Format.3d
3DTestGen
4.3 Symbolic Test Generation
Test Packets
4.1 Compiler Feedback: syntax and type errors
everparse
4.2 Compiled Candidate Parser
Parser.c
Reference Labeler
5. Parser.c agrees with reference labeler
4.4 Compare Labels

Fig. 1: Workflow of 3DGEN

specifications are at least well defined. Next, exploiting the fact that 3D is a small language with a well-designed formal semantics, we develop 3DTESTGEN a new symbolic test-case generator for 3D. This allows us to automatically generate new test inputs for candidate specifications, and we rely on various external oracles to decide the intended classification of an input. Enriching the input with the new test cases, we repeat the loop.

5. Final Parser: Having converged with 3DGEN's help on a given 3D specification, the user relies on EverParse to generate verified C code that is guaranteed to parse only all messages that are well-formed according to the specification.

We evaluate 3DGEN on 20 Internet standard formats, starting from their specification in RFCs. While 3DTESTGEN can generate tests from candidate specifications, we still require oracles to label those tests with their desired outcome; specifically, we use the Wireshark[3] network packet analyzer to decide if a packet should be accepted or not. Interestingly, in 11 cases, 3DGEN discovers constraints specified in RFCs that Wireshark does not enforce. Additionally, for 7 protocol formats, we also evaluated 3DGEN's ability to produce 3D specification that match the behavior of prior, handwritten specifications for various formats provided as samples by the authors of EverParse. In this setting, 3DGEN uncovers 3 cases in which the human authored specifications were incorrect. That said, the specifications 3DGEN produces are only as good as the tests on which it is evaluated. In 2 cases, 3DGEN produces specifications that agree with Wireshark, but 3DTESTGEN detects, via symbolic differential testing, that the generated specification is semantically distinct from a human-authored EverParse sample—the 3DGEN-produced specification does not enforce a constraint that it should. As such, we caution that 3DGEN should not be used to blindly match the behavior of a legacy tool. Instead, we envision 3DGEN and its symbolic tools to be used by humans to iteratively refine a natural language document into a formal specification, while also growing a carefully curated test suite.

A key enabler of our technique is the use of an effectively analyzable DSL, coupled with a verified code generator, as a medium of interaction between a user's informal intent and AI-generated output. In contrast, directly prompting AI agents to produce C code from informal specifications would leave open the question of analyzing ad hoc C code for safety and security, and with an unclear formal basis against which to assess code correctness. Further, targeting a DSL enables us to integrate powerful, fully automated tools like symbolic test-case generation and differential analysis that are usually intractable for large, general-purpose languages. We conjecture that future AI-assisted programming techniques might also benefit from the use of effectively analyzable DSLs as intermediate languages.

In summary, we make the following contributions:

1) An architecture for AI-assisted programming using DSLs coupled with symbolic analysis tools to refine informal user intent into formal specifications, grounded in the scenario of binary format parsers.
2) A new symbolic analysis and test generation tool for the 3D format language, integrated in 3DGEN's intent refinement loop.
3) An evaluation of 3DGEN on a suite of 20 binary format parsers specified in Internet standard RFCs, yielding safe and secure C code.

II. PROBLEM FORMULATION

Ramananandro et al. [2] present EverParse, a library of parser and serializer combinators in the $F^\star$ programming language. They prove that every well-typed program assembled from their parser combinators produces a parser that is the inverse of the corresponding serializer, i.e., $\forall s.\ \mathsf{parse}\ (\mathsf{serialize}\ s) == \mathsf{Some}\ s$ and $\forall b\ v.\ \mathsf{parse}\ b == \mathsf{Some}\ v \implies \mathsf{serialize}\ v == b$. Parsers for formats that satisfy this mutual inverse property are particularly relevant in security-critical settings. EverParse combinators are themselves embedded within a fragment of $F^\star$ called Low* [5], which supports transpilation to C via a tool called Karamel.

Swamy et al. [3] present a DSL built on top of EverParse combinators called 3D, a language similar to C's language of type definitions, with typedefs, structures, and unions. 3D allows users to express a variety of "tag-length-value" style formats, which are commonly used in many networking protocols and other variable-length formats.

Internet RFCs often specify tag-length-value formats in natural language in an ad hoc way. For example, the TCP RFC 793 specifies the format of TCP options as follows:

```
There are two cases for the format of an option:

Case 1:  A single octet of option-kind.
Case 2:  An octet of option-kind, an octet of
option-length, and the actual option-data octets.
...
Currently defined options include
(kind indicated in octal):

    Kind     Length    Meaning
    ----     ------    -------
```

[3] https://www.wireshark.org/docs/man-pages/tshark.html

```
     0          -        End of option list.
     1          -        No-Operation.
     2          4        Maximum Segment Size.
...

 Maximum Segment Size

          +--------+--------+---------+--------+
          |00000010|00000100|   max seg size   |
          +--------+--------+---------+--------+
           Kind=2   Length=4

Maximum Segment Size Option Data:  16 bits
...
```

To produce a parser for a TCP option in 3D, one starts by specifying the format declaratively. Here's one way to do it, defining an `OPTION` as a structure with two fields: a byte field `Kind` with a constraint that restricts its values to 0, 1, and 2; and a `payload` field of type `OPTION_OF_KIND(Kind)`, a type that *depends on* the value of the `Kind` field.

```
typedef struct _OPTION {
    UINT8 Kind {
        Kind == 0x00 ||
        Kind == 0x01 ||
        Kind == 0x02
    };
    OPTION_OF_KIND(Kind) payload;
} OPTION;
```

The type `OPTION_OF_KIND`, a `casetype` in 3D, represents a form of *union* type in C, where the parameter `Kind` discriminates the case of the union. When `Kind` is 0 or 1, the payload is empty, and when the `Kind` is 2, the payload has type `MAX_SEG_SIZE`.

```
casetype _OPTION_OF_KIND(UINT8 Kind) {
    switch (Kind) {
        case 0x00: unit case0; /*unit: empty payload*/
        case 0x01: unit case1; /*unit: empty payload*/
        case 0x02: MAX_SEG_SIZE case2;
    }
} OPTION_OF_KIND;
```

Finally, the type `MAX_SEG_SIZE` is a structure, with one byte for the `Length` and 2-bytes for an unsigned 16-bit big-endian integer for the `MaxSegSize`.

```
typedef struct _MAX_SEG_SIZE {
   UINT8 Length;
   UINT16BE MaxSegSize;
} MAX_SEG_SIZE;
```

From this specification, EverParse (in its simplest mode) generates a C program with the following signature, a function `CheckOption` which, when called with a byte buffer `Input` containing at least `Length` bytes, checks that `Input` contains a valid representation of `OPTION`, returning an error code recording success, or details about where and why validation failed.

```
EVERPARSE_ERROR_CODE CheckOption(
    UINT8* Input,
    UINT64 Length)
```

A 3D user turning an ad hoc description from an RFC to a specification must convince themselves that they have captured the intent of the RFC—a process that typically involves careful review combined with testing. While 3D was designed to be used by C programmers and benefits from its resemblance to C, the constructs it offers, including type dependency, value constraints, parameterization, case analysis, etc. take effort to learn and use correctly. Further, while EverParse guarantees that the generated C code is memory safe, free from bugs that trigger undefined behaviors, and faithfully parses exactly the specified format, the source specification is still subject to audit. For example, one could easily have specified `UINT16 MaxSegSize`, forgetting a convention that networking protocols like TCP typically use big-endian integers—users need assistance in specification testing and validation.

Swamy et al. [3] report that 3D has been used to specify a suite of networking protocols used in production software at Microsoft, including in the kernel of the Windows 11 release. They report using 3D to specify "137 structs, 22 casetypes, and 30 enum type definitions" in around 5,000 lines of 3D specifications, stating that "describing those message formats required careful specification engineering and discovery" over a period of 18 months. With 3DGEN, we seek to lower this overhead, making 3D accessible to non-experts by directly synthesizing specifications from informal intent, and to offer systematic testing tools to assist with validation.

III. THE 3DGEN APPROACH

In this Section, we make precise the workflow in Figure 1, showing how we derive a 3D specification from RFCs and tests. We start by describing the algorithm abstractly, parameterized by several non-deterministic choices, AI-based components, and a test generator for 3D. We then describe an *agent-based* implementation of the AI-based components, and 3DTESTGEN, a new symbolic test generator for 3D.

A. 3DGEN*: An Abstract Algorithm*

We assume the 3D DSL is equipped with the following functions:

- 3DSYNCHK: A syntax and type checker function, given a specification p checks for syntax as well as type constraints imposed by the 3D language.
- 3DEXEC: An execution function; for any 3D specification p that satisfies 3DSYNCHK(p), given a packet i, 3DEXEC(p, i) returns true iff the specification p accepts the packet i. Concretely, we use EverParse to compile p to C code and execute it on i.
- *3DDoc*: Natural language documentation about the 3D language and examples, provided as a manual.

Algorithm 1 takes as input an *RFC* document, function *LblImpl* that is used to classify packets, as well as a (possibly empty) seed sets of positive and negative packets, I_0^+ and I_0^-. The desired 3D specification p should accept the positive packets I_0^+ and reject I_0^-. The algorithm returns a set of possible candidate 3D specifications *CandProgs*, along with an augmented set of positive (I^+) and negative (I^-) packets, generated by 3DTESTGEN and labeled using *LblImpl*, ensuring that every specification in *CandProgs* is consistent with the augmented set of packet inputs. Formally, $I_0^+ \subseteq I^+$, $I_0^- \subseteq I^-$, and for each $p \in$ *CandProgs*, $\text{3DE}\textsc{xec}(p, i^+) =$ true for each $i^+ \in I^+$, and $\text{3DE}\textsc{xec}(p, i^-) =$ false for each $i^- \in I^-$.

The algorithm iterates non-deterministically, accumulating state which records all relevant information to be fed to an LLM in st, initialized to the *3DDoc*, the *RFC*, and the seed tests. At each iteration, it performs one of the following two actions non-deterministically: (i) augment *CandProgs* with a new well-formed 3D specification p by querying an LLM (lines 5–13); or, (ii) augment the labeled packets in I^+ and I^- using 3dTestGen and LabelInputs (lines 14–17). Finally, at lines 18–26, candidates in *CandProgs* that are not consistent with the labeled packets are pruned, and any failing candidate/test pairs are added to the state (line 20).

To generate a candidate program p, at line 6 we QueryLLM with the accumulated state—this step is implemented using agents, as described in Section III-B. If p fails the 3DSynChk, we update the state with the error, and retry.

The 3D symbolic test generator 3dTestGen takes as input a set of well-formed candidate programs that satisfy the current $I^+ \cup I^-$, and outputs new (unlabeled) packets by symbolically analyzing the programs in *CandProgs*; we describe the precise implementation in Section III-C. We then use *LblImpl* to label each packet as positive or negative—concretely, we use Wireshark as an implementation of *LblImpl*. Details of LabelInputs, and the use of Wireshark as a reference labeler, is present in Section IV-C.

Throughout this procedure the generated candidate program p is only used as input to 3dTestGen. The set of tests generated are labeled by a reference implementation, Wireshark. If the candidate program does not match the behavior implemented by Wireshark, the program must be discarded or modified. For example, when the candidate program accepts packets that Wireshark labels as negative, or discards packets that Wireshark labels as positive.

B. Agent Based Implementation

Constructing a prompt for an LLM from the diverse information in Algorithm 1's accumulated state is non-trivial. It involves choosing relevant sections of natural language documentations from several pages of *RFC*, *3DDoc*, relevant examples, failing tests to focus on, etc. Composing a single monolithic prompt with all relevant context needed to solve a task is often impossible, given the restricted token context-window for LLMs.

Instead, research shows that LLM-based *agents* significantly extend the capabilities of standalone LLMs by equipping them with the abilities needed to solve tasks in a self-directed fashion, such as long-term planning, reasoning [6], conversing with other LLMs, using tools, and retrieving information critical to task resolution [7]. Agents demonstrate improved performance and generalization of task resolution abilities for a number of increasingly complex and real-world tasks [8], [9]. Furthermore, orchestrating multiple agents that are instructed to cooperate together, can scale up the capabilities of a single agent by decomposing tasks, improving factuality and reasoning [10], and validation [11].

Algorithm 1 3DGen Algorithm

```
Input: $RFC, LblImpl, I_0^+, I_0^-$
Output: $CandProgs, I^+, I^-$
 1: $(I^+, I^-) \leftarrow (I_0^+, I_0^-)$
 2: $st \leftarrow \{3DDoc, RFC, I^+, I^-\}$
 3: $CandProgs \leftarrow \{\}$
 4: for $*$ do
 5:     if $*$ then
 6:         $p \leftarrow \textsc{QueryLLM}(st)$
 7:         $se \leftarrow \text{3D}\textsc{SynChk}(p)$
 8:         if $se \neq$ SUCCESS then
 9:             $st \leftarrow st \cup \{(p, se)\}$
10:             continue
11:         end if
12:         $CandProgs \leftarrow CandProgs \cup \{p\}$
13:     end if
14:     if $*$ then
15:         $I' \leftarrow \text{3d}\textsc{TestGen}(CandProgs, I^+ \cup I^-)$
16:         $(I^+, I^-) \leftarrow (I^+, I^-) \cup \textsc{LabelInputs}(I', LblImpl)$
17:     end if
18:     for $q \in CandProgs$ do
19:         for all $i \in I^+ \cup I^-$ do
20:             if $\bigvee \begin{pmatrix} i \in I^+ \wedge \neg \text{3DE}\textsc{xec}(q, i) \\ i \in I^- \wedge \text{3DE}\textsc{xec}(q, i) \end{pmatrix}$ then
21:                 $st \leftarrow st \cup \{(q, i)\}$
22:                 $CandProgs \leftarrow CandProgs \setminus \{q\}$
23:                 break
24:             end if
25:         end for
26:     end for
27: end for
28: return $CandProgs, I^+, I^-$
```

Motivated by these findings, we design an agent system based on the AutoGen [12] multi-agent framework. AutoGen allows the instantiation of multiple agents, each unique in their task description, access to tools and inputs, and instructed to cooperate together to achieve a solution. We choose to use a multi-agent framework over a single agent, to decompose tasks and reduce overall input in the context window, shielding other agents from unrelated intermediate reasoning steps involved in distinct task refinement loops. In the AutoGen multi-agent setup agents converse in a *group chat* setting, critiquing and reflecting on task progress based on conversation history, and adapting from feedback. AutoGen provides the multi-agent conversation framework as a high level abstraction, requiring only meta prompts and tool customization from the user. The agent framework designed for 3dGen is not reliant on one particular LLM, however we use GPT4-32k across all experiments. The 3dGen multi-agent framework is implemented concretely as three distinct agents:

1) **Planner Agent:** the planner agent is a tool-backed agent that orchestrates the multi-agent conversation. In AutoGen it is instantiated as the *group chat manager*. It has access to the meta task prompt, descriptions of the other two agents, and the ability to invoke tools 3DSynChk and 3DExec, and communicate the results to the other agents.
2) **3D Developer Agent:** the 3D developer agent is tasked with generating 3D code based on instructions communicated from the other two agents. This agent has access to the full 3D language manual *3DDoc*, a set of task examples, and high level tips about generating syntactically correct 3D code.

3) **Domain Expert Agent:** the domain expert agent is tasked with communicating specifications to the 3D Developer Agent, and critiquing generated 3D code. It has access to all domain-relevant documents needed to solve the problem. In this work, the domain expert agent has access to an RFC, the network protocol specification document needed to solve the task, and examples of the task.

All three agents communicate via an inter-agent group chat, until one of two termination conditions are met: either the output of 3DExec indicates that the generated 3D specification passes on the test set, or the maximum number of iterations, as set by the user, have been completed. The control flow of task resolution follows the paradigm provided in AutoGen, and is entirely conversation-driven, i.e. the participating agents' decisions on which agents to send messages to and the procedure of computation are functions of the inter-agent conversation. An example of control flow is as follows: 1) the Domain Expert agent communicates the relevant parts of the RFC specification to the 3D Developer Agent 2) the 3D Agent generates a candidate specification 3) the Planner Agent makes a call to 3DSynChk and communicates the result to the group 4) the 3D Agent reflects on a syntax error reported by the Planner and refines the specification. One example of the control flow is later shown in Figure 3.

C. 3dTestGen*: Symbolic Test Case Generation*

In this section, we discuss our implementation of the 3dTestGen sub-routine of Algorithm 1. Our test (packet) generator, called 3dTestGen, is implemented as an extension of the EverParse toolchain, and is grounded in the formal semantics of 3D. 3dTestGen encodes 3D programs into the SMT-LIB version 2 language (a.k.a. SMT2) [13], relying on SMT-solver Z3 [14] to produce test cases. Given a 3D program p, to obtain positive (resp. negative) test cases, 3dTestGen encodes the semantics of p to Z3 and asks for models of the existential predicate: "does there exist a sequence of bytes that makes p succeed (resp. fail)". Z3 returns with one of the following answers:

- SAT: Z3 finds a model to satisfy predicate, including a concrete sequence of bytes that makes the parser succeed (resp. fail)
- UNSAT: the predicate is *unsatisfiable*, which means that p always fails (resp. succeeds)
- UNKNOWN: Z3 times out. While there may be multiple causes to Z3 timeout; in our case, this happens rarely.

As one of our contributions in this paper, we give a new "logical semantics" of 3D by encoding it to the SMT2 first-order logic with quantifiers, uninterpreted functions, linear integer arithmetic, and maybe additional theories as requested by the user: for instance, a 3D program with non-linear integer constraints would also have a logical semantics relying on non-linear arithmetic in SMT2.

The structure of our logical semantics follows the structure of the denotational semantics of 3D. Figure 2 provides a brief overview of these semantics, presenting some of its characteristic features—we refer the reader to Swamy et al.'s [3] complete denotational semantics for full details.

The formal syntax of a 3D program builds on the syntax and semantics of pure F* expressions e and types t. Expressions include variables, boolean and integer constants, arithmetic and boolean operators, decidable equality comparison, and can be extended with any pure F* expression. Encoding pure expressions to SMT is relatively straightforward, e.g., F* already provides such an encoding, however we implement a simpler encoding for the subset of pure expressions that appear in 3D programs. 3D also include atomic types, including base types for bounded integer types like $\mathbb{U}_8, \mathbb{U}_{16}$ etc. as well as refinements of these types $x : t\{e\}$, the restriction of t to elements x for which e evaluates to true.

The main feature of 3D is to introduce a notion of composite types p, including a variety of constructs, such as support for variable-length arrays and zero-terminated strings. To give a flavor of the denotational semantics, we focus here on three simple elements of 3D: atomic types t; structures $(x : t; p)$, whose first field is x of type t and the rest of the structure is described by p, dependent on x; and unions $(\underline{\text{if}}\ e\ \underline{\text{then}}\ p_1\ \underline{\text{else}}\ p_2)$, a union whose case is determined by the boolean expression e—in §II such unions were presented using the `switch`/`case` notation, which is represented in the formal semantics by a cascade of conditionals.

The denotational semantics of a 3D program p is split into two parts: a *type semantics* $\llparenthesis p \rrparenthesis$, which represents the type of the values read by the parser, and a *parser semantics* $\llbracket p \rrbracket$, a partial function representing a parser of type $\mathbb{U}_8^* \rightharpoonup \llparenthesis p \rrparenthesis \times \mathbb{N}$ (where $\mathbb{U}_8 = [0..255]$ denotes the type of bytes as 8-bit unsigned integers) that takes as argument a finite sequence σ of input bytes and, if parsing succeeds, returns the value read by the parser and the number of input bytes consumed (which is used to compose parsers together.) Swamy et al. also give a third semantics to 3D programs as an imperative C program for an executable parser, proving it equivalent to the parser semantics, which is their main compiler correctness theorem. We do not rely on this third semantics for 3dTestGen.

In Figure 2, the type semantics of atomic types $\llparenthesis t \rrparenthesis$ is just t. For a structure, the type semantics $\llparenthesis x : t; p \rrparenthesis$ is a dependent pair type. Conditionals are interpreted as conditional types. The parser semantics of a base type $\mathbb{U}_8$ returns the first element of the input byte sequence and consumes one byte (failing if the input sequence is empty). The semantics of a refinement $\llbracket x : t\{e\} \rrbracket$ parses first according to $\llbracket t \rrbracket$, and additionally checks that the refinement e is valid on the parsed result. For structures, $\llbracket x : t; p \rrbracket$, we first parse $\llbracket t \rrbracket$, obtaining a value v_1 of type t and consuming a prefix σ' of the input, and then parse $p[x := v_1]$, on the suffix σ'' of the input, obtaining v_2 and returning the pair (v_1, v_2) together with the total number of bytes consumed. Finally, parsing a conditional involves branching on e and parsing according to the appropriate branch.

Our goal is to give an SMT2 encoding for 3D that mirrors the parser semantics. There are three main features of our encoding. First, we model partial functions by adding an

Syntax *Expressions* $e ::= x \mid \text{true} \mid \text{false} \mid 0 \mid 1 \mid \dots \mid e_1 + e_2 \mid e_1 \;\&\; \&e_2 \mid e_1 = e_2 \mid \dots$

Atomic types $t ::= \mathbb{U}_8 \mid \cdots \mid x\!:\!t\{e\}$ *Composite types* $p ::= t \mid x\!:\!t; p \mid \underline{\text{if}}\ e\ \underline{\text{then}}\ p_1\ \underline{\text{else}}\ p_2$

Type semantics $(\!|\cdot|\!) : \text{Type}$ $(\!|t|\!) = t$ $(\!|x\!:\!t;p|\!) = \{x\!:\!t \;\&\; (\!|p|\!)\}$ $(\!|\underline{\text{if}}\ e\ \underline{\text{then}}\ p_1\ \underline{\text{else}}\ p_2|\!) = \text{if } e \text{ then } (\!|p_1|\!) \text{ else } (\!|p_2|\!)$

Parser semantics $[\![p]\!] : \mathbb{U}_8^* \rightharpoonup (\!|p|\!) \times \mathbb{N}$

$[\![\mathbb{U}_8]\!]\sigma = (\sigma_0, 1) \text{ if } |\sigma| \geq 1$ $[\![x\!:\!t\{e\}]\!]\sigma = (v, n) \text{ if } [\![t]\!]\sigma = (v, n) \text{ and } e[x := v] \Downarrow \text{true}$

$[\![x\!:\!t;p]\!]\sigma = ((v_1, v_2), n_1 + n_2) \text{ if } [\![t]\!]\sigma = (v_1, n_1) \text{ and } \exists \sigma', \sigma''.\ \sigma = \sigma'\sigma'' \text{ and } |\sigma'| = n_1 \text{ and } [\![p[x := v_1]]\!]\sigma'' = (v_2, n_2)$

$[\![\underline{\text{if}}\ e\ \underline{\text{then}}\ p_1\ \underline{\text{else}}\ p_2]\!] = \text{if } e \text{ then } [\![p_1]\!] \text{ else } [\![p_2]\!]$

Fig. 2: Syntax and semantics of a simplified fragment of 3D. Functions are undefined if their conditions do not hold; Expressions e, values v, and atomic types t have the standard semantics of pure mathematical terms including arithmetic and boolean operators.

additional element to the range, and denoting a parser that returns a negative number for the number of bytes consumed as a failed parser. Next, rather than explicitly passing a byte sequence as an input to a parser, we declare an uninterpreted function σ to represent the full input byte sequence, and we make parsers take two integers as arguments, the position and length of a slice of σ. This allows us to use Z3 to find a model that constrains the single, global input σ. position and length of a slice of σ. Finally, we keep our SMT2 encoding structurally similar to the F* denotational semantics, to enable the logical semantics to be easily audited for correspondence. However, the F* denotational semantics is higher-order, whereas SMT2 is first-order. For example, the F* semantics for $[\![x : t; p]\!]$ is a higher-order function, parse_pair $[\![t]\!]$ $(\lambda x.[\![p]\!])$. To address this mismatch, our encoding provides a logical semantics of a given 3D program, rather than providing a semantics for *all* 3D programs at once. This allows us to specialize all applications of higher-order functions yielding SMT2-compliant first-order definitions. The end result is full semantics of a given 3D program encoded as a first-order formula to Z3, in contrast with symbolic execution tools like Klee [15] or Sage [16], which repeatedly encode individual paths to solvers. With our encoding, we can generate test cases by asking Z3 to find inputs that are accepted by the encoded parser or not.

Our encoding includes one further feature: To ensure diversity of test cases and coverage of all possible branches of a 3D program, we instrument our encoding by labeling each branch with an integer and, for each generated test case, having Z3 produce a trace of traversed branch labels, which we can reason about, instead of accumulating the logical conditions during branch traversal.

We provide more details on our SMT2 encoding in supplementary materials. Leveraging the existing semantics of 3D and its compact, structured language of parser combinators, our implementation of 3DTESTGEN took less than 3 person-weeks and around 2000 lines of F*, OCaml, and SMT2.

D. Equivalence Checking, Decidability, and Trust

Another benefit of our logical semantics of 3D is that it generalizes naturally to *differential* symbolic analysis. In particular, given two 3D programs p_1 and p_2, we encode them both as first-order formulas and ask Z3 to find input byte sequences that are accepted by p_1 and rejected by p_2, or vice versa.

If Z3 returns UNSAT, it has found a proof of equivalence of the logical semantics p_1 and p_2, i.e., they both accept or reject the same input byte sequences and represent the same partial function.

Z3 can also return UNKNOWN. This is because the equisatisfiabliity of first-order formulas is an undecidable problem. Further, 3D programs that use theories like non-linear arithmetic also contribute to the undecidabilty. Even satisfiability checking for first-order logic is undecidable, so, in principle, even generating test input for a single program could fail. However, in practice, for the formats we have worked with, Z3 is effectively able to find test inputs for single programs, differentiating inputs for pairs of programs, and even to prove equivalence.

In general, one needs to trust that our logical encoding matches the denotational semantics of 3D. As such, our logical encoding is part of the trusted computing base. It being structured in close correspondence with the official 3D semantics makes it easy to audit for correctness. Additionally, when Z3 is able to find a concrete test case, we can confirm that it behaves as expected by checking the behavior against the behavior of the verified C code, which is proven to conform to the 3D semantics.

In the future, it is possible that more complex formats could require other approaches to enable test-case generation and equivalence checking, e.g., systematic path enumeration for test-case generation, or relational logics and rewrite rules for equivalence checking.

IV. EXPERIMENTAL SETUP

A. Network Protocols

We evaluate 3DGEN on 20 network protocols specified in IETF standards. Table I lists each protocol and a short description and the specific RFC number. We include the length of the RFC in pages, as a rough measure of complexity, though RFCs contain a lot of information beyond the description of the format—the number in parenthesis, when present, shows the number of pages in the RFC concerned with header formats.

B. Generating Specifications with 3DGEN

To evaluate 3DGEN's ability to translate natural language specifications into 3D format specifications, we use it to generate 5 candidate specifications for the protocols in Table I. We deem a generation successful if the produced specification

#	Protocol	RFC (Version)	Length (Pages)	Description
1	UDP*	768	3	User Datagram Protocol
2	ICMPv4 *	792	21	Internet Control Message Protocol
3	VXLAN*	7348	22	Virtual eXtensible Local Area Network
4	IPV6*	2460	39 (24)	Internet Protocol version 6
5	IPV4*	791	45 (12)	Internet Protocol version 4
6	TCP*	793	85 (10)	Transmission Control Protocol
7	Ethernet*	7348	22	Ethernet II Frames in VXLAN
8	GRE	2784	9	Generic Routing Encapsulation
9	IGMPv2	2236	24	Internet Group Managment Protocol
10	DHCP	2131	45 (4)	Dynamic Host Configuration Protocol
11	DCCP	4340	129 (14)	Datagram Congestion Control Protocol
12	ARP	826	10	Address Resolution Protocol
13	NTP	5905	110 (4)	Network Time Protocol
14	NBNS	1002	84 (6)	NetBIOS Name Service
15	NSH	8300	40 (8)	Network Service Header
16	TFTP	1350	11	Trivial File Transfer Protocol
17	RTP	3550	104 (3)	Transport Protocol for Real-Time Applications
18	PPP	1661	52 (11)	Point-to-Point Protocol
19	TPKT	2126	25	ISO Transport Service on top of TCP
20	OSPF	5340	94 (13)	Internet Official Protocol Standards

TABLE I: Dataset of protocols and corresponding RFCs. * denotes protocols for which there is a human written 3D specification. Page numbers in ( ) indicate the length of the extracted RFC.

correctly classifies a test set of generated packets labeled by Wireshark. We report a *pass@5* metric, which counts the number of successful generations out of 5 runs. To evaluate how well agents in the 3DGen multi-agent framework are able to understand the natural language specifications contained in the network protocol RFC, as well as its ability to learn 3D syntax, we explore the number of refinement loops needed to generate 1) a syntax- and type-correct solution as determined by 3DSynChk and 2) a semantically correct solution with respect to the test set, as determined by 3DExec. In addition, we highlight common mistakes made by the agents, as well as several instances where the agent is able to learn constraints from the RFC that are not enforced by the Wireshark.

For each protocol we instantiate 3DGen, using GPT-4-32k as the underlying LLM, at temperature 1.0. We set the number of specifications to generate to 5 and the max number of syntax or packet refinements to 15 per attempt. If the agents are able to produce a specification that passes before 15 refinements, the agent loop is terminated. We observe that allowing more refinement loops within the 3DGen agent conversation flow does not always lead to a successful attempt at generating a specification. On the other hand, reducing the number of refinement loops often does not give the agents enough attempts at solving the problem. This is especially true when the protocol is more complex, requiring the agents to produce a longer specification, for example in the case of ICMP, where there are 8 distinct message types for which the agent must generate a 3D specification.

For each protocol, we download the RFC from the IETF Data Tracker[4] as a text file to provide as input in the 3D Developer agent prompt. In some cases, the full RFC is prohibitively long, and would exhaust the GPT-4-32k token window. Since we are only interested in the part of the specification related to the header data format, in such cases we manually extract the pages of the RFC related to the data format specification, (usually labeled in a section called "Header Specification"). The length of the extract pages is denoted in ( ) in Table I.In the future, we plan to explore building a retrieval tool specific to RFC extraction.

C. Generating a Labeled Test Set for each Protocol

We build a test suite consisting of a mixture of real-world packet captures and synthetically generated tests by 3dTestGen, where we use Wireshark[5], a popular network protocol analyzer, as a reference labler to decide if a packet should be considered valid (i.e. accepted) or not. Wireshark exposes several parsers which can be used to attempt to parse packet contents, and determine if the packet is a valid instance of a particular format.

For each protocol, we collected a small number of real world packets from various sources on the Internet and retain only those that Wireshark considers valid—we discard the negative cases, since they are sometimes arbitrarily malformed. For synthetic tests, we run one iteration of the 3DGen loop seeded with the real-world packets. This produces a single candidate specification on which we run 3dTestGen to generate 200 test cases. We configure 3dTestGen to fully explore a trace with 100 branch points, and observed that while many examples have a large number of branches (e.g., ICMP has 82 branches), none of them have more than 100 branches. We then collect at least two examples from each branch point, until we reach 200 examples. This gives us some confidence that the generated test suite is diverse, though its completeness is limited by the quality of the specification on which 3dTestGen is executed. We then use Wireshark to label the tests, obtaining both positive and negative test cases.

In the ideal scenario, one could use a trusted reference labeler to label the test generated by 3dTestGen without any modification to the labels. However, using Wireshark as a reference implementation to label packets comes with its own challenges. For starters, Wireshark is a protocol analyzer typically used for diagnostics and experimentation and, by design, does not always enforce all constraints when validating a packet. Wireshark does not implement a dissector for a single RFC, but rather for a family of RFCs. Thus, a dissector may be more permissive compared to a given RFC, perhaps because a related RFC mandates such a behavior—our experiments in Section V uncover many such unenforced constraints. To label a test case produced by 3dTestGen, we rely on a Wireshark feature called *Export PDU*, which allows validating a given packet header without encapsulating it within outer protocol headers. Two exceptions were Ethernet, which does not require outer encapsulation; and TFTP which is not supported by Export PDU. We wrapped TFTP headers generated by 3dTestGen with a dummy UDP header. For Ethernet, IPv4, IPv6, VXLan, TCP, and UDP, we also had to generate dummy payloads to prevent Wireshark from raising trivial errors. We also had to disable checksums, since this is not enforced at the level of the formats. As such, using Wireshark as a labeler for 3dTestGen outputs involved a non-trivial effort.

[4] https://datatracker.ietf.org

[5] https://www.wireshark.org/docs/man-pages/tshark.html

Furthermore, the chosen reference labeler may not always be an ideal labeler with respect to the natural language specifications. For example, in the case of some protocols, the Wireshark dissector implementation deviates from the specification outlined in the RFC. Wireshark is permissive by design, and as a result, does not emit warnings for packets that violate some specifications in the RFC. In such cases, a user of 3DGen may choose to intervene and align certain labels for the generated tests. We discuss concrete examples of label alignment, as a result of a mismatch between the Wireshark implementation and RFC specifications in the Results Section (V-A).

D. Handwritten 3D Specifications

The EverParse GitHub[6] repository contains 3D specifications for seven network protocols, written by the authors of EverParse. Protocols with an existing handwritten 3D specification are marked with a * in Table I. In Section V-C we use 3DTestGen to compare the specifications generated by 3DGen to the seven handwritten specifications. We report if any of the specifications produced by 3DGen are semantically equivalent to the handwritten specifications, and otherwise use the generated set of tests to identify the cause(s) of differences.

V. Results

A. Capabilities of 3DGen

In this section, we present experimental evidence to answer the following central question underpinning our work:
RQ1: Can 3DGen produce format specifications for networks protocols standardized in RFCs?

Table II details results of 3DGen on our dataset of 20 network protocols, We report the pass@5 metric, the average number of syntax refinement, and packet feedback refinement loops across all attempts.

At a first glance, 3DGen is able to generate a passing specification for 9/20 protocols, with a pass@5 $= 45\%$. For two protocols, such as UDP and IGMP, 3DGen generates a passing specification in all 5 attempts, requiring an average of 0.3 syntax refinements and 0 packet refinements for UDP, and 3.8 and 0 for IGMP respectively. For the other seven, IPV4, DHCP, DCCP, ARP, NTP, NBNS, and NSH, 3DGen can generate at least one passing specification, but is not successful in all 5 attempts.

For the remaining 11 protocols, 3DGen is unable to generate a specification that passes on the test set labeled by Wireshark. We investigate the cause of the errors and find in all cases that 3DGen generates a specification that is consistent with the header format described in the RFC, but that Wireshark labels packets as valid despite there being constraint violations, i.e., Wireshark does not fully enforce the RFC. In some cases, Wireshark emits a warning about these violations, which can be filtered on a case-by-case basis. However, this is not a straightforward task as some warnings do not impact the data format specification, e.g., Wireshark emits a warning when a packet indicates the TCP connection is reset. Besides, in most cases a warning is not reported.

Using the RFC as a guide, two authors manually identified tests that Wireshark labels too permissively and corrected the labels. Using these corrected labels, we checked if any of the 3DGen generated specifications already pass on this modified test set, or else restart the 3DGen loop with the corrected tests. Protocols for which 3DGen is able to generate a specification that passes on the corrected test set are denoted as (1*) in Table II—3DGen is able to generate at least one candidate specification that is consistent with the labels in *all* cases.

1) Labeler and RFC disagreement

We look more closely at examples where Wireshark is too permissive: Table III details the cause(s) of disagreement between the Wireshark labels and the constraints specified by the RFC in each case.

For example, for RTP, RFC 3550[7] states that the `version` field is 2 bits and *"The version defined by this specification is two (2)"*. Although the RFC does not indicate a strong constraint that the version field must be set to 2, a large number of negative packets in the RTP test set include version numbers other than 2. Wireshark labels these packets as acceptable, however the 3DGen agent consistently generates a specification with `UINT16BE Version:2 {Version == 2}`. In this case, the agent is unable to learn from feedback about why a test with a different version value should pass, because Wireshark does not provide any warning. Enforcing this constraint by changing the Wireshark label to negative allows the specification to pass all tests.

Similarly for TFTP, RFC 1350[8] states that a TFTP header contains a 2 byte `opcode` field and enumerates 5 possible opcode values. 3DGen always produces a specification constraining the value of the opcode field to one of the 5 values in the RFC. Wireshark does not label packets with other opcode values as malformed. When we manually label tests with incorrect opcode fields as negative, the specification generated by 3DGen passes on the test set.

In the case of GRE, RFC 2784[9] states *"The Version Number field MUST contain the value zero"*. However, Wireshark does not enforce version constraints on GRE, likely because the for the PPTP variant of GRE, the version field is set to 1, and Wireshark uses the same dissector for all GRE versions.

In the case of IPV6, the generated specification consistently fails to reject packets labeled as malformed by Wireshark. The IPV6 `NextHeader` field indicates the value of the encapsulated packet, however the RFC focuses only on the constraints of a single layer, not of any encapsulated packets. In contrast, Wireshark validates packet contents across layers and rejects packets that don't encapsulate the next layer correctly.

2) Agent Mistakes

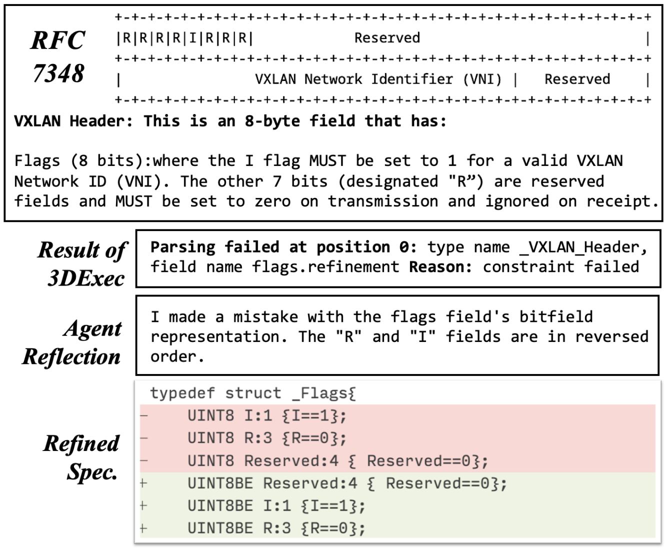
From Table II, we observe that the agents frequently make syntax mistakes, despite having access to the language manual

[6] https://github.com/project-everest/everparse

[7] https://www.ietf.org/rfc/rfc3550.txt

[8] https://datatracker.ietf.org/doc/html/rfc1350

[9] https://datatracker.ietf.org/doc/html/rfc2784



RFC 7348

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|R|R|R|R|I|R|R|R|            Reserved                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                VXLAN Network Identifier (VNI) |   Reserved    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

VXLAN Header: This is an 8-byte field that has:

Flags (8 bits):where the I flag MUST be set to 1 for a valid VXLAN Network ID (VNI). The other 7 bits (designated "R”) are reserved fields and MUST be set to zero on transmission and ignored on receipt.

Result of 3DExec

Parsing failed at position 0: type name _VXLAN_Header, field name flags.refinement **Reason:** constraint failed

Agent Reflection

I made a mistake with the flags field's bitfield representation. The "R" and "I" fields are in reversed order.

Refined Spec.

```
 typedef struct _Flags{
-    UINT8 I:1 {I==1};
-    UINT8 R:3 {R==0};
-    UINT8 Reserved:4 { Reserved==0};
+    UINT8BE Reserved:4 { Reserved==0};
+    UINT8BE I:1 {I==1};
+    UINT8BE R:3 {R==0};
```

Fig. 3: Example of a packet refinement loop by the 3DGEN agent for VXLAN RFC 7348

for the 3D language. However, given that 3D is not yet a widely used DSL, syntax mistakes from a model like GPT-4 are anticipated. We observe that the agents struggle most with using correct 3D bitfield notation. Using only the primitives that 3D supports (`UINT8`, `UINT16`, `UINT32`, `UINT64` and their BE counterparts) and refraining from using reserved keywords like ‘type’ as identifier names also require refinement steps. On the other hand, the agents are able to easily learn some other 3D specific constructs, such as the `consume-all` notation. We also observe that semantic mistakes, i.e., incorrect specifications, can stem from the agent’s difficulty in understanding the natural language in the RFC document. Figure 3 shows one such example for VXLAN. The RFC describes the fields in the header in natural language, and also provides an ASCII diagram. However, the natural language description of the fields do not indicate the order in which the different values should occur in the VXLAN header, and without the ASCII diagram a reader would not be able to correctly interpret the RFC. The agent first produces a specification which is then executed using 3DEXEC, resulting in a parsing error on the flags field, for one of the tests in the test set. Then the agent reflects on the result of 3DEXEC and determines that the parsing failed due to the mis-ordering of the `Reserved` and `I` fields. It then refined the specification by flipping the order of the fields (4) and produces a candidate that passes on all tests. Interestingly, the language in the RFC first describes the `I` flag and then says specifies the values of the other 7 bits, without indicating that there are 4 reserved bits, followed by an I bit, followed by the remaining 3 bits. Ambiguities in the RFC language may cause the agent to misinterpret the true intent.

In addition, results from Table II indicate that, without a feedback loop, a single call to GPT-4 would always result in a syntactically incorrect specification. Such a refinement loop is imperative to generating valid specifications and is likely to outperform a potential baseline approach using GPT-4 implemented outside of the 3DGEN agent refinement framework

> **Result 1:** 3DGEN *is able to generate syntactically correct 3D code, and learn from mistakes to refine specifications. Even in the presence of a noisy labeler and non-exhaustive tests, 3DGen enables users to leverage the generated specifications to align the test set with the RFC, yielding specifications that pass all aligned tests for all 20 network protocols.*

Protocol	**Accepted (x/5)**	**Avg. Syntax Refinements**	**Avg. Packet Refinements**
UDP	5	0.3	0
ICMP	0 (1*)	7.5	6
VXLAN	0 (2*)	7.6	7.4
IPV6	0 (1*)	7.8	2.0
IPV4	2	4	11
Ethernet	0 (1∗)	11.4	4.6
TCP	0 (2∗)	10.2	2.5
GRE	0 (1∗)	10.4	4.6
DHCP	2	8.2	0
DCCP	1	14.25	0.75
TPKT	0 (1*)	5.0	6.6
ARP	3	4.8	1.8
NTP	3	7.4	3
NBNS	1	4.6	4
IGMP	5	3.8	0
NSH	1	11.0	2.0
TFTP	0 (1*)	11	1
RTP	0 (1*)	3	12
PPP	0 (1*)	7.6	5
OSPFv3	0 (1*)	13.6	2
	pass@5: (45%)	pass*@5: (100%)	

TABLE II: Results of 3DGEN for 20 network protocols. * denotes protocols for which the test labels were adjusted to be consistent with the RFC.

B. Distinguishing Candidates with Differential Testing

In many cases, 3DGEN produces multiple specifications that are compatible with the test set. In this section, we aim to answer the following question:

RQ2: How do candidate format specifications generated for the same protocol differ?

We make use of 3DTESTGEN to help us answer this question, in particular its *differential testing* feature, to find tests that distinguish specifications or prove them semantically equivalent. Distinguishing tests, if any, can be surfaced to the user, along with feedback from 3DTESTGEN localizing semantic differences in 3D specifications, to help the user identify their desired specification.

Table IV shows protocols for which 3DGEN generates multiple candidate specifications, and the results of 3DTESTGEN’s differential testing between every pair of candidates, with a description of the differences found, if any. Out of 8 protocols, 7 have at least two semantically distinct specifications, whereas for VXLAN the two candidates are semantically equivalent.

For example, for ARP, 3DGEN generates 3 candidate specifications, two which are semantically equivalent, whereas the third mistakenly adds a field to the end of the ARP header `UINT8 remainder[:consume-all]`. This field consumes the rest of the data in the packet and is often used for optional variable

Protocol	Detected RFC vs Wireshark Disagreement	Wireshark Message
ICMP	Header length constraints	None
Ethernet	`Ethertype` payload length	None
VXLAN	`Reserved` bits must be 0 `I flag` must be 1 Header must be 8 bytes	None None None
IPV6	`Payload Length` exceeds framing length	Warning
GRE	`Reserved0` must be zero `Version` number must be zero	None None
TFTP	`Opcode` fields must be between 0-5	None
TCP	`Window` fields must not be zero `ACK number` must be consistent with `ACK` flag	Warning Warning
TPKT	`Version` field must be 3 `Reserved` field must be 8 bits	None None
PPP	`Code` field must be between 1-11 `Length` field must be 1 octet, at least size 4 `Data` must be constrained by `Length`	None None None
RTP	`Version` must be 2	None
OSPF	`Version` field must not be 3 `Reserved` field must be 0 Header length must be 16 bytes	None None None

TABLE III: Constraints specified in the RFC, that wireshark does not enforce.

Protocol	# Candidates	# Distinct Candidates	Divergent Fields
UDP	5	2	Optional `Data[:consume-all]` field
IPV4	2	2	Additional constraints on `Flag` values
VXLAN	2	1	None
DHCP	2	2	`options` field length Additional constraints on `Flags` field
ARP	3	2	Incorrect additional `remainder[:consume-all]` field
NTP	3	3	Additional constraints on LeapIndicator, Status, Type fields
IGMP	5	3	Optional `OtherFields[:consume-all]`
TCP	2	2	Constraints on `Options` field

TABLE IV: Semantically distinct candidate specifications

length fields. The ARP RFC 826 does not describe such a field, and the specification is incorrect but passes the test set because there is no negative-label test for which there is additional data at the end of the ARP header.

3DGen generates two candidate specifications for IPV4, and 3DTestGen find a test that distinguishes them. One specification has a field: `UINT16BE flags:3 { flags == Reserved0 || flags == DF || flags == MF }`, where `Reserved0`, `DF`, `MF` are equal to `0`, `1`, and `2`, which is consistent with the RFC. The second specification has the same field as `UINT16BE Flags:3` and does not enforce constraints on the value. While these constraints should be added to be consistent with the RFC, the test set did not contain a test violating this constraint, thus both specifications were able to pass the test set.

In both the cases of ARP and IPV4, 3DTestGen labels the one specification as more permissive than the other, e.g. for IPV4 every test that passes on the first specification also passes on the second, but not the other way around. For both of these cases, the stricter specification correctly implements the RFC. Thus, a user of 3DGen could decide to always accept the stricter specification as a pruning heuristic between candidates.

Result 2: *Multiple distinct specifications may be produced by* 3DGen *for a single protocol, and the degree to which they diverge is dependent on the quality and coverage of the test suite on which they are evaluated.* 3DTestGen *helps by finding differentiating tests or by grouping equivalent candidates, allowing users to focus on a semantic differences exhibited by concrete test cases.*

C. 3DGen *vs. Human Written Specs*

The authors of EverParse provide specifications for 7 out of the 20 protocols we ran 3DGen on. In this section, we ask:

RQ3: How do format specifications generated by 3DGen compare to handwritten specifications?

As before, we use 3DTestGen's differential testing to semantically compare 3DGen's specifications to the handwritten ones, with the results in Table V. For IPV6 and Ethernet, 3DTestGen proves that the specifications are equivalent, though syntactically distinct.

For UDP, ICMP, and VXLAN, we use 3DTestGen to identify tests that distinguish the handwritten and generated specifications. In all three cases, the root cause is incorrect or missing constraints in the handwritten specifications, demonstrating that even experts make mistakes when interpreting RFCs as 3D, and that 3DGen can help in ensuring consistency with RFCs. For UDP, the handwritten specification is missing a constraint on the `Length` field that exists in the 3DGen specification. For ICMP, the `Unused Bytes` field is too short, misinterpreting the 32 bytes as 32 bits. Similarly, in VXLAN, the `VXLanID` is two bytes short. After correcting the handwritten specification, 3DTestGen proves them equivalent to the 3DGen generated specifications. Pull requests with the revised specifications were merged into EverParse for all three protocols.

On the other hand, for TCP and IPV4, the 3DGen specification is missing constraints that exist in the handwritten specification. For TCP, although the produced specification passes on the set of tests, it is underconstrained and does not include a condition checking if the `SYN` flag is set to 1 and it does not implement all possible constraints for different `Max Segment Size` payloads. Instead it includes size constraints that are general to all options.

For IPV4, there are missing constraints on the `IHL`, `TotalLength`, and `Options` fields, which indicates that tests labeled by Wireshark do not capture these constraints. We explore whether 3DGen would be able to generate an equivalent specification, if it had access to a set of tests that can capture the missing constraints. To do this, we use 3DTestGen to generate positive and negative tests from the handwritten specification, guaranteeing that tests exercising the constraints will be included in the test set. With these tests, 3DGen is able to produce two passing specifications that contain the missing constraints, after an average of 11 syntax and 4 packet refinements. The generated specifications are both semantically equivalent to the handwritten specification.

Result 3: 3DGen *is able to produce 3D specifications semantically equivalent to human written 3D. In addition, using our framework we were able to uncover three bugs in existing handwritten 3D code for UDP, ICMP, and VXLAN, highlighting the difficulties in translating RFCs into correct implementations.*

Protocol	Equivalent?	Root Cause Divergence	After H.S. Fix
UDP	✗	H.S. Missing constraint on Length field	✓
ICMP	✗	H.S. UNUSED_BYTES type too short	✓
VXLAN	✗	H.S. VXLanID field too short	✓
IPV6	✓	None	n/a
IPV4	✗	G.S Missing value constraints on IHL, TotalLength	n/a
Ethernet	✓	None	n/a
TCP	✗	G.S. Missing constraints on options payload	n/a

TABLE V: Comparison of 3DGen generated specifications to handwritten specifications (denoted H.S.). We list the cause(s) of divergence, and where applicable, we correct the handwritten specification.

VI. Related Work

LLMs have enabled generating code from informal natural language requirements, and have shown ability to generate human like code on benchmark problems [17], [18] – however, they come with no guarantees, and have been known to contain bugs and security errors [19]. AlphaCode [20] and CodeT [21] have used tests to cluster and rank generated code to improve the empirical accuracy on benchmarks; however they do not add trust to the generated code as the natural language does not impose any correctness checks.

On the other hand, classical program synthesis [22] formulates the problem of generating code that meets a formal specification. However, these techniques are limited due to lack of availability of formal specifications, along with the intractable theoretical complexity of the synthesis. Lately, program synthesis has been applied in restricted domains with input-output examples as specifications [23], constrained by restrictions on syntax (e.g., SyGuS [24]). These restrictions make it difficult to apply them for new domains with formal guarantees. Office Domain Specific Language (ODSL) [25] has been proposed as an intermediate layer for LLMs to translate natural language user commands to programs over Office APIs. Although it shares our motivation for using 3D as a DSL, generated programs from ODSL do not have any formal guarantees since the generated programs lack a formal notion of correctness and there is no symbolic encoding of programs in ODSL into a logical formula.

Closer to our setting, Ticoder [26] uses LLMs to partially formalize user-intent as tests. Unlike 3DGen, TiCoder requires a user in the loop to validate each test, relies on an LLM-based test generation that cannot be as exhaustive as our symbolic technique and cannot provide any formal guarantees on the generated code. Endres et al. [27] generate declarative postconditions in Java and Python using LLMs and evaluate the quality of specifications offline using validation tests, but do not generate tests or verified code. Misu et al. [28] generate formal specifications and code that satisfies such specifications in Dafny programming language using LLMs, but there is no automation in helping the user establish the correctness of the specifications.

All these approaches are evaluated for a simple setup where the requirements are present as a few line docstrings, and do not require problem decomposition or the translation of requirements from complex documents such as RFCs. SAGE [29] uses natural language processing (NLP) techniques to translate informal requirements in RFCs into protocol implementations semi-automatically. SAGE extracts and surfaces ambiguities in RFCs through an intermediate logical form, that are resolved by the user, before generating code. Unlike 3DGen, SAGE can generate protocol implementation in addition to the parser; however, the generated code may have functional and security bugs, as it lacks formal specifications. Extending 3DGen and the 3D language to support full protocol implementation would be interesting avenue for future work.

The problem of inferring input grammars has been studied extensively using a variety of symbolic techniques. Stevenson and Cordy [30] provide a survey of grammar-learning techniques, particularly as used in software engineering. Other recent works include Arvada [31], for learning context-free languages from inputs; Mimid [32], which in addition to inputs also uses code analysis of a given implementation; and the work of Shi et al. [33] which uses static analysis to lifts network protocol implementation code into an "abstract format graph" that can be used for systematic testing. Another related work is ISLa [34], a specification language for "input invariants" that, like 3D, can capture some context-sensitive constraints on inputs.

Our usage of 3DGen for differential reasoning is inspired by prior works on differential symbolic testing [35], [36] and verification [37]. Our contribution lies in a succinct encoding of 3D denotational semantics that avoids costly path exploration outside of an SMT solver, and in integrating such a symbolic analysis to distinguish between multiple candidate solutions. Also related is ParDiff [38], which infers format descriptions from multiple network protocol parsers and can compare the inferred formats relationally to find differences in behavior.

VII. Conclusion

Programming in natural language using AI is a powerful new capability. However, for AI-based program synthesizers to be truly useful, they must also be trustworthy. We believe coupling AI programming assistants with symbolic tools to support intent formalization and refinement, as well guarantees about generated outputs, is a key step towards fully realizing their potential. In this work we explore this idea, showing it is possible to synthesize verified binary format parsers from specification documents using AI agents, while providing symbolic test-case generators to help both humans and AIs confirm and refine intent, and verification tools to ensure that intent is preserved down to executable C code. As a next step, we plan to evaluate our approach through user studies to assess whether tools like 3DGen more easily enable humans to author correct-by-construction programs in new DSLs.

References

[1] J. Bangert and N. Zeldovich, "Nail: A practical tool for parsing and generating data formats," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 615–628. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/bangert

[2] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko, "Everparse: Verified secure zero-copy parsers for authenticated message formats," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, p. 1465–1482.

[3] N. Swamy, T. Ramananandro, A. Rastogi, I. Spiridonova, H. Ni, D. Malloy, J. Vazquez, M. Tang, O. Cardona, and A. Gupta, "Hardening attack surfaces with formally proven binary format parsers," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA*, 2022. [Online]. Available: https://www.fstar-lang.org/papers/EverParse3D.pdf

[4] OpenAI, "Gpt-4 technical report," 2023.

[5] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, and N. Swamy, "Verified low-level programming embedded in F*," *PACMPL*, vol. 1, no. ICFP, pp. 17:1–17:29, Sep. 2017. [Online]. Available: http://arxiv.org/abs/1703.00053

[6] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2022.

[7] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, and H. Wang, "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, 2023.

[8] Z. Xi, W. Chen, X. Guo, W. He, Y. Ding, B. Hong, M. Zhang, J. Wang, S. Jin, E. Zhou *et al.*, "The rise and potential of large language model based agents: A survey," *arXiv preprint arXiv:2309.07864*, 2023.

[9] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin *et al.*, "A survey on large language model based autonomous agents," *arXiv preprint arXiv:2308.11432*, 2023.

[10] Y. Du, S. Li, A. Torralba, J. B. Tenenbaum, and I. Mordatch, "Improving factuality and reasoning in language models through multiagent debate," *arXiv preprint arXiv:2305.14325*, 2023.

[11] C. Qian, X. Cong, C. Yang, W. Chen, Y. Su, J. Xu, Z. Liu, and M. Sun, "Communicative agents for software development," *arXiv preprint arXiv:2307.07924*, 2023.

[12] Q. Wu, G. Bansal, J. Zhang, Y. Wu, S. Zhang, E. Zhu, B. Li, L. Jiang, X. Zhang, and C. Wang, "Autogen: Enabling next-gen llm applications via multi-agent conversation framework," *arXiv preprint arXiv:2308.08155*, 2023.

[13] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," https://smtlib.cs.uiowa.edu/, 2016.

[14] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer, 2008, pp. 337–340.

[15] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[16] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, p. 40–44, mar 2012. [Online]. Available: https://doi.org/10.1145/2093548.2093564

[17] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[18] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[19] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 754–768.

[20] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. d. M. d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," 2022. [Online]. Available: https://arxiv.org/abs/2203.07814

[21] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," 2022. [Online]. Available: https://arxiv.org/abs/2207.10397

[22] Z. Manna and R. Waldinger, "A deductive approach to program synthesis," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 1, p. 90–121, jan 1980. [Online]. Available: https://doi.org/10.1145/357084.357090

[23] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Found. Trends Program. Lang.*, vol. 4, no. 1-2, pp. 1–119, 2017. [Online]. Available: https://doi.org/10.1561/2500000010

[24] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/6679385/

[25] A. Gandhi, T. Q. Nguyen, H. Jiao, R. Steen, and A. Bhatawdekar, "Natural language commanding via program synthesis," 2023.

[26] S. K. Lahiri, S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, M. Musuvathi, P. Choudhury, C. von Veh, J. P. Inala, C. Wang, and J. Gao, "Interactive code generation via test-driven user-intent formalization," *CoRR*, vol. abs/2208.05950, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2208.05950

[27] M. Endres, S. Fakhoury, S. Chakraborty, and S. K. Lahiri, "Formalizing natural language intent into program specifications via large language models," *CoRR*, vol. abs/2310.01831, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.01831

[28] M. Md Rakib Hossain Misu, C. V. Lopes, I. Ma, and J. Noble, "Towards ai-assisted synthesis of verified dafny methods," 2024.

[29] J. Yen, T. Lévai, Q. Ye, X. Ren, R. Govindan, and B. Raghavan, "Semi-automated protocol disambiguation and code generation," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 272–286. [Online]. Available: https://doi.org/10.1145/3452296.3472910

[30] A. Stevenson and J. R. Cordy, "A survey of grammatical inference in software engineering," *Science of Computer Programming*, vol. 96, pp. 444–459, 2014, selected Papers from the Fifth International Conference on Software Language Engineering (SLE 2012). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642314002469

[31] N. Kulkarni, C. Lemieux, and K. Sen, "Learning highly recursive input grammars," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '21. IEEE Press, 2022, p. 456–467. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678879

[32] R. Gopinath, B. Mathis, and A. Zeller, "Mining input grammars from dynamic control flow," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 172–183. [Online]. Available: https://doi.org/10.1145/3368089.3409679

[33] Q. Shi, J. Shao, Y. Ye, M. Zheng, and X. Zhang, "Lifting network protocol implementation to precise format specification with security applications," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1287–1301. [Online]. Available: https://doi.org/10.1145/3576915.3616614

[34] D. Steinhöfel and A. Zeller, "Input invariants," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 583–594. [Online]. Available: https://doi.org/10.1145/3540250.3549139

[35] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Pundefinedsundefinedreanu, "Differential symbolic execution," in

Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. SIGSOFT '08/FSE-16. New York, NY, USA: Association for Computing Machinery, 2008, p. 226–237. [Online]. Available: https://doi.org/10.1145/1453101.1453131

[36] R. Rutledge and A. Orso, "Automating differential testing with overapproximate symbolic execution," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2022, pp. 256–266.

[37] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "Symdiff: A language-agnostic semantic diff tool for imperative programs," in *Computer Aided Verification*, P. Madhusudan and S. A. Seshia, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 712–717.

[38] M. Zheng, Q. Shi, X. Liu, X. Xu, L. Yu, C. Liu, G. Wei, and X. Zhang, "Pardiff: Practical static differential analysis of network protocol parsers," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, apr 2024. [Online]. Available: https://doi.org/10.1145/3649854