LLMs Unleashed: Generating Protocol Code from RFC Specifications

Junfeng Long[1], **Jinshu Su**[2], **Biao Han**[1*]

[1]College of Computer Science and Technology, National University of Defense Technology, Changsha, China
[2]Academy of Military Science, Beijing, China
ljf9302@nudt.edu.cn, birchsu@139.com, nudtbill@163.com

Abstract

RFC (Request for Comments) documents constitute the foundation of network protocol standardization. However, they are expressed in natural language, they tend to be lengthy and ambiguous, forcing protocol implementers to rely on extensive manual parsing and coding—a process that is both labor-intensive and prone to errors. This makes the automated parsing and comprehension of RFC documents a major challenge in network protocol research. To address this gap, we introduce large language models (LLMs) into the task of automatic network protocol code generation from RFC documents (RFC2Code) and propose a comprehensive evaluation framework to quantitatively assess LLM performance. We develop an end-to-end automated protocol generation system, APG (Automated Protocol-Generation), which supports implementations of ICMP, IGMP, NTP, and TCP. Compared to prior NLP (Natural language processing) methods, APG achieves a fully automated workflow with approximately 3.17× faster processing, 95% compile success and behavioral correctness for stateless protocols like ICMP, and 90% interoperability for complex stateful protocols such as TCP, requiring only minimal manual intervention.

Code — https://github.com/Assassin187/APG

1 Introduction

Requests for Comments (RFC), internet protocol specifications have evolved over decades, encompassing all layers from low-level network interconnection to high-level transport control. However, as internet protocol becomes increasingly complex, implementing these specifications often requires engineers to carefully study dozens of pages of text, manually translating natural language descriptions into accurate and maintainable code. Rapid prototyping of network protocols can significantly reduce the aforementioned costs.

Automatically or semi-automatically generated prototype code enables swift functional verification of protocols, ensuring the correctness of core processes such as state machine logic, field validation, and multi-phase interactions. These prototypes can also be directly employed in security testing (e.g., fuzzing), allowing researchers to efficiently uncover potential vulnerabilities and enabling downstream use in verification, simulation, and network security research.

Automated conversion of RFC specifications into compilable protocol implementations faces significant hurdles. While existing techniques (1) traffic/binary analysis (Sija et al. 2018) can extract packet formats and state machines, lacks to access to natural language information in RFCs and require extensive, labor-intensive, error-prone manual rule authoring for semantic parsing; (2) methods relying on specialized grammars (Yen et al. 2021) demand substantial handcrafted rules and frequent expert intervention, limiting scalability and full automation. and (3) recent LLM approaches (Duclos et al. 2024; Sharma and Yegneswaran 2023) show promise, they have not achieved end-to-end automation generating directly compilable code.

Challenges. Despite LLMs significantly reducing the complexity of protocol analysis and implementation through enhanced text understanding and code generation, major challenges remain: (1) The output of LLMs is known to be inherently unreliable (Ji et al. 2023) and highly sensitive to the phrasing of user inputs (Sahoo et al. 2024), making prompt engineering critical to task success; (2) real-world protocol implementations typically involve large-scale modular functionality and extensive code, raising the question of how to effectively embed domain knowledge into prompt design and guide LLMs in decomposing and composing large-scale protocol implementations.

To address the time-consuming nature of manual protocol implementation and the difficulty of balancing generality with human involvement in existing approaches, this paper makes a first effort to propose an **end-to-end code generation framework** for *RFC2Code* based on LLMs. It incorporates the domain expertise of network protocol to guide the LLM in carrying out the task. And we propose a **systematic evaluation framework** to quantify LLM performance on the RFC2Code task. We design general prompt templates targeting both stateless message protocols and stateful transport protocols , enabling LLMs to generate C/C++ implementations. We evaluated the APG (Automated Protocol-Generation) generated code in real-world network environments. Experimental results demonstrate that our approach enables fully automated code generation for stateless proto-

*Corresponding author

cols and achieves high compatibility with existing protocol stacks in complex TCP implementations with minimal human intervention. Currently, APG supports the generation of ICMP, TCP, NTP, and IGMP protocol implementations.

Contributions & organization

(1) System of RFC2Code (Section 3). We introduce large language models (LLMs) into the task of protocol-code generation from RFC documents (RFC2Code) task for the first time. Based on network domain expertise, we develop an automated protocol generation (APG) system.

(2) RFC2Code Evaluation (Section 4). For the RFC2Code scenario, we propose a systematic evaluation framework to quantify LLM performance and limitations in automated network-protocol generation.

(3) Interoperability Validation (Section 5). Experiments have demonstrated that the APG system achieves full automation with zero human intervention, a capability absent in prior work. For stateless protocols, our generation speed significantly outperforms that of previous work, and high compatibility requiring minimal human effort for TCP implementations

2 Related Work

2.1 Large Language Model

Large language models (LLMs) demonstrate remarkable capabilities in text understanding and generation (OpenAI et al. 2024)(Team et al. 2024). More recently, their applicability has extended into the domain of code, achieving strong performance across a wide range of software engineering tasks, including software development, software design (Qian et al. 2023)(Huang et al. 2023), requirements elicitation (Mu et al. 2023), and specification formalization (Luo et al. 2024). These advances are closely aligned with our work, which explores the coding capabilities and applications of LLMs.

Contemporary LLMs have acquired substantial domain knowledge in networking—for instance, integrating LLMs into fuzzing frameworks like AFL for protocol vulnerability discovery(Meng et al. 2024), or leveraging them in network configuration and routing algorithm development, as demonstrated in NetConfEval(Wang et al. 2024). Given that network protocols are published as natural language RFC documents, LLMs—pretrained on vast internet-scale corpora—are well-equipped to parse RFCs, extract key functionalities and behaviors, and thus open up new possibilities for automatically generating protocol implementation code.

2.2 Automated Network Protocol Implementation

Early studies primarily rely on formal synthesis techniques to automatically generate executable code from high-level specifications using methods such as reactive synthesis (Nir Piterman 2012)(Pnueli and Rosner 1989), inductive synthesis (Alur et al. 2015), or proof-driven synthesis From program verification to program synthesis. Another line of research focus on semantic parsing, where natural language descriptions are mapped to structured representations via intermediate logical forms, and subsequently translate into target programming languages (Berant et al. 2013)(Chen and Mooney 2011)(Liang et al. 2017)(Yin and Neubig 2017). However, these approaches often struggle with accuracy when faced with domain-specific terminology and the structural complexity of network protocols.

In recent years, with the rise of large language models (LLMs), several efforts have explored leveraging pretrained models to extract protocol specifications. For instance, PROSPER (Sharma and Yegneswaran 2023) and RFCNLP (Pacheco et al. 2022) utilize GPT-3.5 and technical language embeddings, respectively, to extract finite state machines (FSMs) from RFC documents, demonstrating the potential of LLMs in protocol understanding. However, these approaches largely remain confined to FSM extraction or attack synthesis, and have yet to achieve end-to-end generation of executable protocol code from RFCs.

Against this backdrop, Sage (Yen et al. 2021) emerges as the first representative system to explore the RFC2Code task. It extends Combinatory Categorial Grammar (CCG) CCG with domain-specific vocabulary to parse RFC texts into logical forms (LF), and completes the LF-to-code mapping via rule filtering and interactive refinement, successfully generating a Linux-compatible implementation of the ICMP protocol. Despite its contributions in semantics-driven protocol generation, Sage heavily depends on manual rule extension and expert intervention to ensure correctness, and its rule-based design presents scalability challenges for more complex protocols with extensive component interactions.

3 LLM-based System for RFC2Code

Given these limitations in automation and complex protocol support, this work introduces the first fully LLM-based RFC2Code framework, proposing a system that integrates automated extraction, generation, and validation stages, aiming to achieve higher levels of automation when handling structurally complex and highly interactive network protocols.

Before diving into each module, we define a key internal artifact used throughout APG: the **Implementation Guidebook**. This guidebook is a structured intermediate representation extracted from the RFC text. It includes: protocol component names and descriptions, functional module mappings and function signatures, etc.

3.1 Overview

The APG system follows a three-stage pipeline (Figure 1) to convert RFC specifications into executable protocol implementations: (1) RFC Analyst parses the RFC text to extract structured functional elements and generate the Implementation Guidebook; (2) Network Protocol Coder uses the guidebook to produce C/C++ implementation code aligned with protocol logic and message structures; (3) Code Reviewer evaluates the generated code using a composite framework that checks for compile success, behavioral accuracy, and conformance with protocol specifications. This modular ar-

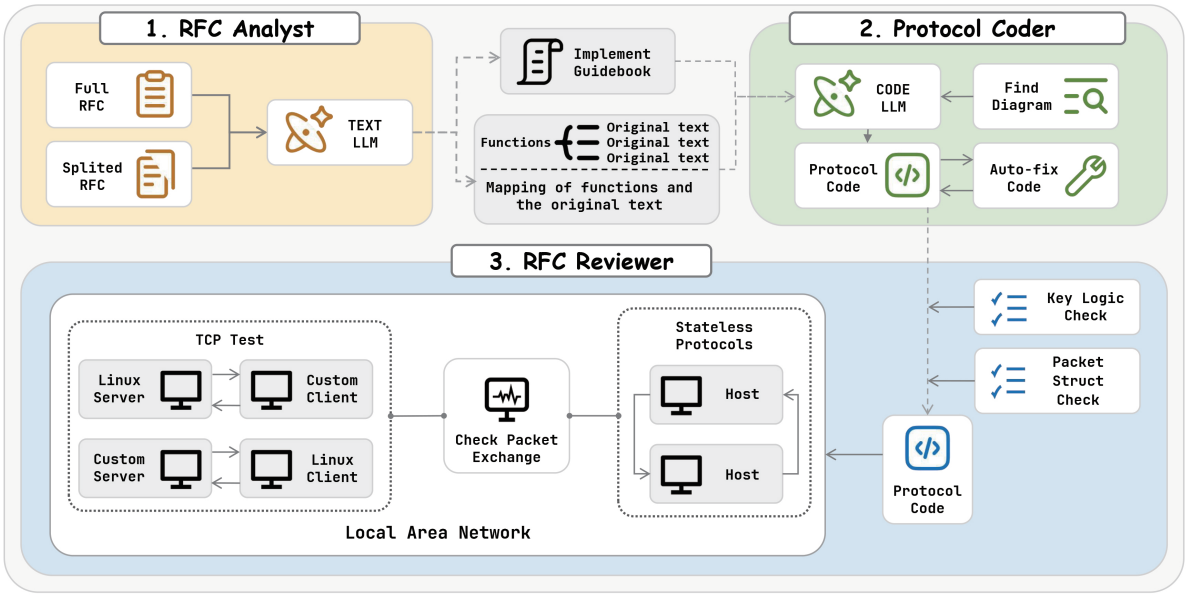

Figure 1: Overview

chitecture enables both horizontal scalability across protocols and vertical integration from specification parsing to end-to-end compilation.

3.2 RFC Analyst

RFC documents vary widely in length, format, and complexity. Some (like ICMP or NTP) describe stateless message protocols with a focus on field structures, while others (like TCP or QUIC) define intricate connection management systems and layered interactions. To efficiently process such heterogeneous documents, the RFC Analyst employs protocol-specific prompt templates that balance broad coverage with targeted information extraction (see Appendix 2). These prompts are used to guide LLMs in parsing the RFCs and assembling an accurate and implementation-ready guidebook.

We categorize protocols into two types. (1) **Stateless Protocols** (e.g., ICMP, IGMP, NTP): These involve simple message exchanges with fixed packet formats. For these protocols, we use parameterized prompts to extract message-type tables, field definitions, and payload generation rules. These prompts require minimal manual configuration and generalize well across similar protocols. (2) **Stateful Protocols** (e.g., TCP, QUIC): These protocols involve complex behaviors such as connection setup, error handling, and state transitions. Here, the prompts are tailored to extract: functional modules, corresponding RFC sections and more. Extracted elements are then aggregated into the Implementation Guidebook. Table 1 outlines a sample structure of the extracted elements.

3.3 Network Protocol Coder

The Network Protocol Coder translates the Implementation Guidebook into complete C/C++ code for the target protocol. Depending on protocol complexity, the code generation strategy varies.

For simple, stateless packet protocols such as ICMP, we design a "one-shot" prompt template that, in a single invocation, automatically generates all packet-type–specific population functions—covering header-field assignments, flag settings, and the complete logic. This method requires no multi-round interaction or manual subtask decomposition and can produce high-quality, directly compilable, and deployable protocol implementation code under one-shot conditions; with minimal parameter substitution, it can be easily extended to other stateless protocols such as NTP and IGMP.

For transport protocols with more complex logic (e.g., TCP, QUIC), we adopt a modular design, abstracting a "Connection Management Class" as the core parent module and generating separate header and source files for each functionality. Since the core operations of these protocols (reliable delivery, flow control, congestion management, error recovery, etc.) all depend on connection state, our prompts explicitly extract and centrally maintain all global state variables within the Connection Management Class. Subsequent functional modules inherit from this par-

Extracted Target	Extracted Content
class_name	Name of the Functional Module Class
component_description	Description of the functionality completed by this class
respective_sections	The section corresponding to the function
function_api	Function signature and description

Table 1: Information Extracted From the Protocol

ent class to directly access the connection context, ensuring state consistency and inter-module collaboration. Given that RFC 9293 spans nearly 270,000 characters and a single extraction cannot cover every implementation detail, our prompts dynamically reference or summarize the original document content to help the LLM "recall" pertinent knowledge, yielding outputs that more faithfully adhere to the RFC specification.

3.4 Code Reviewer

LLMs are known to produce hallucinations or syntactically valid but semantically incorrect code. Therefore, relying solely on successful compilation as a success metric would be insufficient. To address the lack of systematic evaluation standards in the RFC2Code task, the Code Reviewer module introduces a two-stage evaluation framework:

(1) **Protocol Comprehension Assessment**, this evaluates whether the generated Implementation Guidebook captures the intended structure and semantics of the RFC. Evaluation criteria differ by protocol type: For stateless protocols, completeness of packet-field tables and correct message logic; For stateful protocols: coverage of mandatory modules (e.g., connection teardown, timeout handling) and coherence across modules. (2) **Code Quality Evaluation**, which employs a multidimensional metric suite to measure structural correctness, behavioral consistency, and enforceability of the generated code. This includes static field-matching and state-management checks, as well as dynamic compilation validation and behavior-implementation-accuracy assessments. Through this composite scoring framework, we comprehensively characterize the capability boundaries of LLMs in automated protocol-code generation.

4 LLMs Facilitate Network Protocol Generation

Building upon the design principles outlined in the previous chapter, we construct an evaluation framework that integrates domain-specific knowledge from the networking field to assess the feasibility and performance of two core components: (1) the ability of LLMs to automatically parse RFC documents and extract protocol's Implement Guidebook, and (2) the effectiveness of LLMs in generating protocol implementation code based on the extracted Implement Guidebook. Through systematic experiments on these two sub-tasks, we aim to provide a comprehensive analysis of the strengths and limitations of mainstream LLMs in the context of automated network protocol generation.

ModuleName	Main Description
TCPConnection	This class manages the state and behavior of a TCP connection
TCPWindow	This class manages the send and receive windows in TCP
TCPTimeout	This class manages retransmission timeouts (RTO)
...	...

Table 2: Example of Gold Function Set

4.1 Extract Protocol Functional Components From RFC

RFC documents are often lengthy and structurally inconsistent. Extracting a comprehensive and accurate set of functional components from them is a prerequisite for reliable code generation. In this section, we evaluate the capacity of LLMs to parse and structure RFC knowledge into Implementation Guidebooks.

Expriment: To evaluate the effectiveness of large language models in protocol functionality extraction tasks, we select three representative RFCs: RFC 9293 (TCP), RFC 9000 (QUIC), and RFC 792 (ICMP). We manually construct corresponding gold-standard sets of functional components for each protocol, covering all identifiable modules and their core elements within the respective RFCs. These serve as reference benchmarks. Table 2 presents a subset of core functionalities from the standard TCP function set; the complete list is provided in the appendix (see Appendix 1). Subsequently, we conduct multiple independent extraction experiments using GPT-4o and Qwen2.5 under three input conditions: (1) full RFC text, (2) chapter-wise segmentation, and (3) fixed-length chunking. We measure the models' performance on functionality extraction in terms of precision, recall, and F1 score.

Takeaways. The results of our analysis demonstrate that:

- **Full-document input outperforms chunking strategies**: In our experiments on ICMP, TCP, and QUIC protocols, we found that providing the entire RFC document to the LLM in a single input yielded the highest extraction performance (see Figures 2(a) and 2(d)). For structurally simple and concise protocols like ICMP (RFC 792), all input strategies performed reasonably well; however, GPT-4o occasionally miss certain message types, leading to a drop in recall. For complex protocols such as TCP and QUIC, where relevant information is dispersed across sections, LLMs perform best when given access to the full RFC in a single input. Qwen2.5-Plus achieved F1 scores of 0.84 (TCP) and 0.72 (QUIC), slightly outperforming GPT-4o in TCP but trailing in QUIC.
- **Multi-turn chapter-wise chunking performs poorly**: When we segment the RFCs by chapter and fed them to the model in multiple rounds, extraction performance dropped significantly. Due to the extensive length of RFCs like those for TCP and QUIC—often exceeding hundreds of thousands of characters—chapter-wise chunking introduces an excessive number of dialogue turns. As a result, the model tends to forget earlier context and generate overly fine-grained labels based on local content, deviating from the intended standard label set. This fragmented input leads to numerous redundant or irrelevant extractions, causing a substantial decline in overall precision.
- **Fixed-length chunking balances performance and cost**: Compared to chapter-based splitting, fixed-length segmentation maintains F1 scores around a moderate level of 0.6 while significantly reducing the number of dialogue rounds and token consumption. For models with limited context windows or under computa-

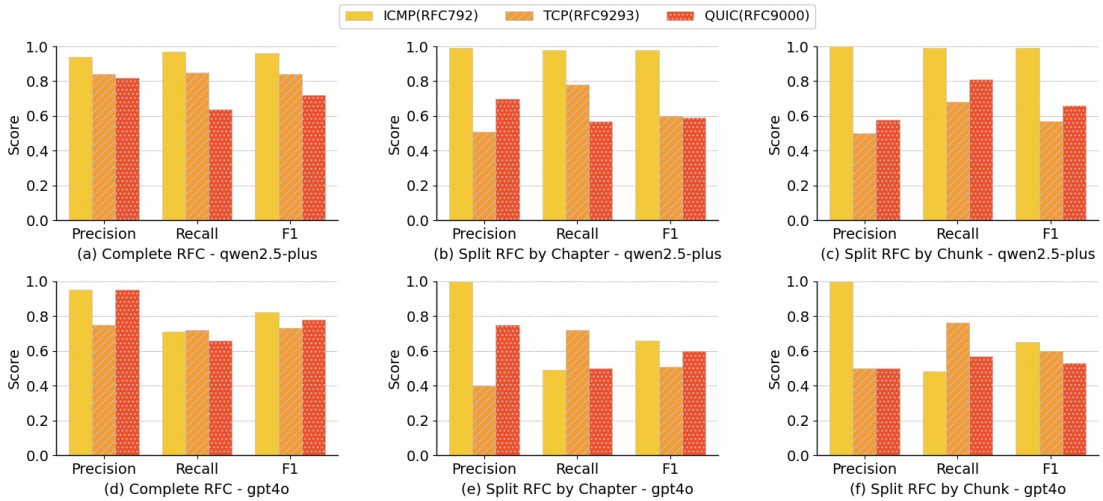

Figure 2: Extraction effect of APG under different extraction strategies

tional constraints, fixed-length chunking offers an efficient tradeoff—preserving acceptable extraction accuracy while minimizing operational cost—thus providing a cost-effective alternative.

4.2 Generate Code Based on Functional Components

Having validated the ability of LLMs to extract functional structures, now we evaluate their effectiveness in translating these into syntactically correct and semantically consistent C/C++ protocol implementations. Given that industrial-grade transport protocols (e.g., TCP, QUIC) are highly optimized and feature complex code coupling beyond the capacity of prompt engineering, we constrain our experiments to the level of minimal functional components. Given the exploratory nature and limited sample size, our reported compile rates and behavioral correctness metrics should be interpreted qualitatively rather than statistically.

Protocol	Model	CSR (%)	Packet Struct (%)	Behavior (%)
ICMP	Qwen2.5	95.0	95.9	100.0
	Qwen-Coder	60.0	100.0	100.0
IGMP	Qwen2.5	100.0	100.0	100.0
	Qwen-Coder	100.0	100.0	100.0
NTP	Qwen2.5	100.0	100.0	N/A
	Qwen-Coder	100.0	100.0	N/A

Table 3: Stateless protocol code generation results using coder model and general model[1]. Since the NTP protocol in our experiment involves only simple packet padding logic, its protocol behavior is marked as N/A.

[1]To compare the difference between the coder and general models within the same series, we selected only the Qwen models.

Expriment: We design a two-factor experimental setup combining "Model Type × Contextual Reference". Under this setup, we repeatedly generate code under three conditions(see Table 4): (1) using only implement guidebook; (2) with additional raw RFC excerpts; and (3) with additional auto-summarized textual abstracts. We began with simple message-based protocols, for which the RFCs are sufficiently concise and thus did not require additional context references. Subsequently, we conduct the above experiments on both general-purpose LLMs (Qwen2.5-Plus and GPT-4o) and a code-specific model (Qwen-coder), using previously extracted component tables that fully cover all standard modules as guidance. For each condition, we generate 10 implementations of the protocol. Evaluation metrics included: (1) Composite Success Rate (CSR)[2], used to comprehensively assess the executability of generated code; (2) Message structure conformance to RFC specifications; (3) Coverage of all protocol state transitions in the generated implementation; (4) Behavioral logic consistency checks, such as TCP handshake sequencing and ICMP payload generation. All static inspections and dynamic tests are conducted in a Linux environment.

Takeaways. The results of our analysis demonstrate that:

- **High reliability of message structure generation**: Experimental results show that current large language models can achieve 100% field definition accuracy and correct basic behavior logic in stateless message protocols (such as IGMP queries/reports, ICMP echo requests/replies, NTP timestamp filling) and TCP message construction tasks (see Table 3 & Table 4). This fully demonstrates the high precision of LLMs in the "templated" message generation scenario.

[2]CSR = DCR + (1 – DCR) × RCR, where DCR (Direct Compile Rate) denotes the percentage of code that compiles without modification, and RCR (Repairable Compile Rate) indicates the proportion of remaining code that can be successfully compiled after up to three automated fixes.

Input + Model	Packet Struct	State Mgmt.	Prot. Behavior	CSR
NoContent + Qwen2.5	1	0.6039	0.6095	0.93
FullContent + Qwen2.5	1	0.6013	0.7286	1.00
Summary + Qwen2.5	1	0.8221	0.8571	1.00
NoContent + Qwen-coder	1	0.7119	0.9524	0.93
FullContent + Qwen-coder	1	0.6061	0.5619	0.80
Summary + Qwen-coder	1	0.6700	0.9500	0.70
NoContent + GPT-4o	1	0.7204	0.6571	1.00
FullContent + GPT-4o	1	0.7508	0.7000	1.00
Summary + GPT-4o	1	0.6662	0.7143	1.00

Table 4: TCP code generation results under different combinations of Model Type and Contextual Reference

- **Feasibility of end-to-end automation**: For stateless message protocols, the APG framework can generate directly compilable protocol implementations with zero fixes—only in a few ICMP cases, due to minor errors in variable array length calculations by the specialized coder model, individual compilations failed. For complex transport protocols (such as TCP), the average compile success rate of both general-purpose LLMs and specialized code models has exceeded 90% (Figure 4(a)), indicating that our system is capable of completing the end-to-end automated process from component extraction to compilable protocol code.

- **The role of contextual prompts in enhancing complex logic generation**: During the generation of TCP state machine and flag exchange logic, introducing raw text segments or summaries significantly improves behavioral consistency. For example, the behavior consistency of Qwen2.5-Plus under no-context conditions is only 0.61, but when full paragraphs and summarized prompts are added, it increases to 0.73 and 0.86, respectively (Table 4). For the specialized coder model, its coverage is higher under the "zero prompt" condition with no context prompts, but when too many original RFC paragraphs are input, exceeding the model's optimal context length or conflicting with its pre-trained preferences, it leads to deviations in the generated logic. The coder model's performance on code executability also shows that when large amounts of text are input, the generation effect of the specialized code model is severely reduced.

	APG (Qwen2.5)	APG (Qwen-coder)	Sage
ICMP	6 min	6 min	19 min
IGMP	3 min	1 min	2 min
NTP	2 min	1 min	6 min

Table 5: Time Comparison Between APG and Sage

5 Real-World Interoperability Validation

5.1 Experiment on Stateless Protocols

To validate our system's effectiveness on real-world protocols, we follow Sage's experimental methodology, testing against the RFCs for ICMP, NTP, and IGMP. For ICMP, we perform local tests on its eight message types and interoperate with the ping utility in a live environment, using tcpdump to verify packet correctness. For IGMP and NTP, we capture the packets emitted by our generated code and validate their accuracy. Given the commonalities among these protocols, the system employs only two generic prompt templates—one for packet-format extraction and one for protocol-code generation.

Evaluation on ICMP: In our ICMP implementation, we cover all eight message types defined in RFC 792. Following Sage's testing protocol, we generate and evaluate ten code instances—each subjected to unit tests for all eight message types. Figure 3 illustrates the number of successful tests per message type: aside from "Destination Unreachable" and "Redirect", which each exhibited one failure, all other types achieve 100 % correctness, yielding an overall accuracy of 97.5 %. The observed errors stemmed from one implementation's improper allocation of buffer space for the variable payload in echo messages, which led to checksum miscalculations.Subsequently, we use the ping utility to verify interoperability in a live environment; all ten implementations correctly interact with the existing toolchain. AGP generates a ICMP implementation in approximately 6 minutes on average, representing a 3.17× speed improvement compared to the 19-minute average reported by Sage(see Table5). Compared to Sage's deterministic, rule-heavy approach necessitating extensive manual maintenance, our LLM-driven method demonstrates superior automation and adaptability while maintaining consistently high correctness across multiple samples.

Evaluation on IGMP and NTP: Thanks to our universally designed templates, we simply substitute the input with the RFC texts for IGMP and NTP to generate the corresponding protocol code—without altering any configuration. We then employ a static-testing framework to construct and send packets using our generated code to real hosts, capturing both outgoing and incoming traffic with tcpdump.For

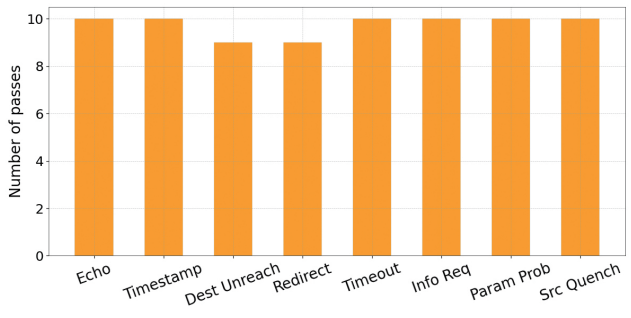

Figure 3: Number of Unit Test Passes for Generated ICMP

NTP, our initial tests against NTPv1 failed because modern servers do not support NTP version 1. After switching to NTPv4, the generated code correctly construct and transmit NTP packets. For IGMP, we use our code to build IGMPv1 packets and send them to a router, verifying that the returned multicast-query messages carries the correct group addresses.The experiments demonstrate that our generated code interoperates seamlessly with existing systems. In contrast to Sage—which requires extensive manual additions of CCG rules for each protocol—our method extends to NTP and IGMP with zero human intervention, showcasing superior automation and generality.

5.2 Interoperability Experiment on TCP

After validating our stateless-protocol generation capability, we extend our focus to complex TCP protocol. To emulate protocol functionality, we use APG on Ubuntu 24.04 to generate a user-space TCP implementation based on raw sockets, then performed end-to-end interactions with a native Linux-kernel TCP client and server, capturing the entire three-way handshake, four-way teardown, and data-transfer processes via tcpdump. Although the LLM can produce a "skeleton" implementation containing correct field declarations and state variables, real-world interoperability reveal the following limitations:

Ambiguity in RFC text vs. diagram: RFC 9293's textual description of connection teardown mentions only "send FIN", yet its illustrative diagram shows the exchange "FIN+ACK $\rightarrow$ ACK $\rightarrow$ FIN+ACK $\rightarrow$ ACK". When relying solely on the text, the model typically generates a single FIN packet and fails to trigger the peer's response. To address this, we devise a heuristic that automatically extracts the complete handshake sequence from the diagram fragment and injects it as an explicit prompt (see Appendix 3).

Model	No Figure	Figure Enhanced
summary + GPT-4o	0.7	0.73
summary + Qwen2.5	0.66	0.86
summary + Qwen-coder	0.86	0.95

Table 6: Impact of Diagram Prompts on Flag Coverage

Experimental results show that prioritizing diagram-derived prompts can improve correct teardown-signal sequencing by up to 20 % (see Table 6).

Checksum-function generation errors: The common checksum-calculation function is particularly prone to errors, likely because RFCs seldom detail the precise computation and the model focuses more on protocol-feature specifics. As a result, the LLM relies on its existing knowledge and fails to adjust to per-protocol nuances. Given the function's reuse across protocols, we borrow Sage's approach by manually implementing checksum algorithms for several protocols and importing them as library functions, thereby avoiding repeated mistakes.

Endianness-conversion omissions: When evaluating TCP flags (e.g., $flags\&(SYN \mid ACK) == (SYN \mid ACK)$) or populating SEQ/ACK fields, the model often omits necessary ntohs(), htonl(), etc., conversions. This leads to packet-inspection failures and incorrect field values on real hosts. By explicitly adding constraints in our prompts to enforce correct host-to-network and network-to-host byte-order conversions for all multi-byte fields, we substantially reduce—but do not entirely eliminate—such errors.

Sequence-number defects: Experiments consistently expose deviations in SEQ and ACK increment logic.Even after instructing the model to focus on the sequence number logic, the error still persisted. Accurate sequence-number management is critical for reliable data transfer; therefore, we further refine our prompt constraints on sequence management and apply minimal manual patches, which enable full reproduction of the kernel-stack's sequence-progression behavior.

In summary, by refining prompts, abstracting key algorithms, and applying minimal manual patches, the LLM-generated TCP implementation interoperates seamlessly with the kernel stack. Across ten tested code instances, those adjusted with the smallest patches achieve a 90 % success rate in three-way handshakes, four-way teardowns, and data-transfer scenarios; the single unfixable sample suffered a severe segmentation fault beyond the scope of network-logic errors.

6 Conclusion

In this paper, we introduce large language models into the RFC2Code task for the first time and propose a systematic evaluation framework to quantify their performance in protocol functionality extraction and code synthesis. Based on this, we develop an end-to-end prototype system, APG. Through a series of interoperability experiments on protocols, we demonstrate that APG can automatically generate correct and compilable interoperable protocol implementations with little to no human intervention. Compared to existing approaches, APG significantly improves the degree of automation and generation speed. However, challenges remain in handling ambiguous RFC content, low-level logic, and scalability to broader protocol families. Future work will focus on extending support to more protocols, enhancing prompt generalization, and incorporating verification techniques to improve robustness and correctness of the generated code.

Acknowledgements

This work is supported by the National Natural Science Foundation of China (Grant No. 62372462), the science and technology innovation Program of Hunan Province (Grant No. 2024RC1044).

References

Alur, R.; Bodik, R.; Dallal, E.; Fisman, D.; Garg, P.; Juniwal, G.; Kress-Gazit, H.; Madhusudan, P.; Martin, M.; Raghothaman, M.; Saha, S.; Seshia, S.; Singh, R.; Solar-Lezama, A.; Torlak, E.; and Udupa, A. 2015. *Syntax-Guided Synthesis*. Nato Science for Peace and Security Series D-Information and Communication Security. Netherlands: IOS Press. ISBN 9781614994947.

Berant, J.; Chou, A.; Frostig, R.; and Liang, P. 2013. Semantic Parsing on Freebase from Question-Answer Pairs. In Yarowsky, D.; Baldwin, T.; Korhonen, A.; Livescu, K.; and Bethard, S., eds., *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, 1533–1544. Seattle, Washington, USA: Association for Computational Linguistics.

Chen, D.; and Mooney, R. 2011. Learning to Interpret Natural Language Navigation Instructions from Observations. *Proceedings of the AAAI Conference on Artificial Intelligence*, 25(1): 859–865. Publisher: Association for the Advancement of Artificial Intelligence (AAAI).

Duclos, M.; Fernandez, I. A.; Moore, K.; Mittal, S.; and Zieglar, E. 2024. Utilizing Large Language Models to Translate RFC Protocol Specifications to CPSA Definitions. ArXiv:2402.00890 [cs].

Huang, Q.; Vora, J.; Liang, P.; and Leskovec, J. 2023. Mlagentbench: Evaluating language agents on machine learning experimentation. *arXiv preprint arXiv:2310.03302*.

Ji, Z.; Lee, N.; Frieske, R.; Yu, T.; Su, D.; Xu, Y.; Ishii, E.; Bang, Y. J.; Madotto, A.; and Fung, P. 2023. Survey of hallucination in natural language generation. *ACM computing surveys*, 55(12): 1–38.

Liang, C.; Berant, J.; Le, Q.; Forbus, K. D.; and Lao, N. 2017. Neural Symbolic Machines: Learning Semantic Parsers on Freebase with Weak Supervision. In Barzilay, R.; and Kan, M.-Y., eds., *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 23–33. Vancouver, Canada: Association for Computational Linguistics.

Luo, Q.; Ye, Y.; Liang, S.; Zhang, Z.; Qin, Y.; Lu, Y.; Wu, Y.; Cong, X.; Lin, Y.; Zhang, Y.; et al. 2024. Repoagent: An llm-powered open-source framework for repository-level code documentation generation. *arXiv preprint arXiv:2402.16667*.

Meng, R.; Mirchev, M.; Böhme, M.; and Roychoudhury, A. 2024. Large Language Model guided Protocol Fuzzing. In *Proceedings 2024 Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society. ISBN 978-1-891562-93-8.

Mu, F.; Shi, L.; Wang, S.; Yu, Z.; Zhang, B.; Wang, C.; Liu, S.; and Wang, Q. 2023. Clarifygpt: Empowering llm-based code generation with intention clarification. *arXiv preprint arXiv:2310.10996*.

Nir Piterman, Y. S., Amir Pnueli. 2012. Synthesis of reactive(1) designs | Proceedings of the 7th international conference on Verification, Model Checking, and Abstract Interpretation. Archive Location: world.

OpenAI; Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F. L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; Avila, R.; Babuschkin, I.; Balaji, S.; Balcom, V.; Baltescu, P.; Bao, H.; Bavarian, M.; Belgum, J.; Bello, I.; Berdine, J.; Bernadett-Shapiro, G.; Berner, C.; Bogdonoff, L.; Boiko, O.; Boyd, M.; Brakman, A.-L.; Brockman, G.; Brooks, T.; Brundage, M.; Button, K.; Cai, T.; Campbell, R.; Cann, A.; Carey, B.; Carlson, C.; Carmichael, R.; Chan, B.; Chang, C.; Chantzis, F.; Chen, D.; Chen, S.; Chen, R.; Chen, J.; Chen, M.; Chess, B.; Cho, C.; Chu, C.; Chung, H. W.; Cummings, D.; Currier, J.; Dai, Y.; Decareaux, C.; Degry, T.; Deutsch, N.; Deville, D.; Dhar, A.; Dohan, D.; Dowling, S.; Dunning, S.; Ecoffet, A.; Eleti, A.; Eloundou, T.; Farhi, D.; Fedus, L.; Felix, N.; Fishman, S. P.; Forte, J.; Fulford, I.; Gao, L.; Georges, E.; Gibson, C.; Goel, V.; Gogineni, T.; Goh, G.; Gontijo-Lopes, R.; Gordon, J.; Grafstein, M.; Gray, S.; Greene, R.; Gross, J.; Gu, S. S.; Guo, Y.; Hallacy, C.; Han, J.; Harris, J.; He, Y.; Heaton, M.; Heidecke, J.; Hesse, C.; Hickey, A.; Hickey, W.; Hoeschele, P.; Houghton, B.; Hsu, K.; Hu, S.; Hu, X.; Huizinga, J.; Jain, S.; Jain, S.; Jang, J.; Jiang, A.; Jiang, R.; Jin, H.; Jin, D.; Jomoto, S.; Jonn, B.; Jun, H.; Kaftan, T.; Kaiser, ; Kamali, A.; Kanitscheider, I.; Keskar, N. S.; Khan, T.; Kilpatrick, L.; Kim, J. W.; Kim, C.; Kim, Y.; Kirchner, J. H.; Kiros, J.; Knight, M.; Kokotajlo, D.; Kondraciuk, ; Kondrich, A.; Konstantinidis, A.; Kosic, K.; Krueger, G.; Kuo, V.; Lampe, M.; Lan, I.; Lee, T.; Leike, J.; Leung, J.; Levy, D.; Li, C. M.; Lim, R.; Lin, M.; Lin, S.; Litwin, M.; Lopez, T.; Lowe, R.; Lue, P.; Makanju, A.; Malfacini, K.; Manning, S.; Markov, T.; Markovski, Y.; Martin, B.; Mayer, K.; Mayne, A.; McGrew, B.; McKinney, S. M.; McLeavey, C.; McMillan, P.; McNeil, J.; Medina, D.; Mehta, A.; Menick, J.; Metz, L.; Mishchenko, A.; Mishkin, P.; Monaco, V.; Morikawa, E.; Mossing, D.; Mu, T.; Murati, M.; Murk, O.; Mély, D.; Nair, A.; Nakano, R.; Nayak, R.; Neelakantan, A.; Ngo, R.; Noh, H.; Ouyang, L.; O'Keefe, C.; Pachocki, J.; Paino, A.; Palermo, J.; Pantuliano, A.; Parascandolo, G.; Parish, J.; Parparita, E.; Passos, A.; Pavlov, M.; Peng, A.; Perelman, A.; Peres, F. d. A. B.; Petrov, M.; Pinto, H. P. d. O.; Michael; Pokorny; Pokrass, M.; Pong, V. H.; Powell, T.; Power, A.; Power, B.; Proehl, E.; Puri, R.; Radford, A.; Rae, J.; Ramesh, A.; Raymond, C.; Real, F.; Rimbach, K.; Ross, C.; Rotsted, B.; Roussez, H.; Ryder, N.; Saltarelli, M.; Sanders, T.; Santurkar, S.; Sastry, G.; Schmidt, H.; Schnurr, D.; Schulman, J.; Selsam, D.; Sheppard, K.; Sherbakov, T.; Shieh, J.; Shoker, S.; Shyam, P.; Sidor, S.; Sigler, E.; Simens, M.; Sitkin, J.; Slama, K.; Sohl, I.; Sokolowsky, B.; Song, Y.; Staudacher, N.; Such, F. P.; Summers, N.; Sutskever, I.; Tang, J.; Tezak, N.; Thompson, M. B.; Tillet, P.; Tootoonchian, A.; Tseng, E.; Tuggle, P.; Turley, N.; Tworek, J.; Uribe, J. F. C.; Vallone, A.; Vijayvergiya, A.; Voss, C.; Wainwright, C.; Wang, J. J.; Wang,

A.; Wang, B.; Ward, J.; Wei, J.; Weinmann, C. J.; Welihinda, A.; Welinder, P.; Weng, J.; Weng, L.; Wiethoff, M.; Willner, D.; Winter, C.; Wolrich, S.; Wong, H.; Workman, L.; Wu, S.; Wu, J.; Wu, M.; Xiao, K.; Xu, T.; Yoo, S.; Yu, K.; Yuan, Q.; Zaremba, W.; Zellers, R.; Zhang, C.; Zhang, M.; Zhao, S.; Zheng, T.; Zhuang, J.; Zhuk, W.; and Zoph, B. 2024. GPT-4 Technical Report. ArXiv:2303.08774 [cs].

Pacheco, M. L.; von Hippel, M.; Weintraub, B.; Goldwasser, D.; and Nita-Rotaru, C. 2022. Automated Attack Synthesis by Extracting Finite State Machines from Protocol Specification Documents. ArXiv:2202.09470 [cs].

Pnueli, A.; and Rosner, R. 1989. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '89*, 179–190. Austin, Texas, United States: ACM Press.

Qian, C.; Liu, W.; Liu, H.; Chen, N.; Dang, Y.; Li, J.; Yang, C.; Chen, W.; Su, Y.; Cong, X.; et al. 2023. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*.

Sahoo, P.; Singh, A. K.; Saha, S.; Jain, V.; Mondal, S.; and Chadha, A. 2024. A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint arXiv:2402.07927*.

Sharma, P.; and Yegneswaran, V. 2023. PROSPER: Extracting Protocol Specifications Using Large Language Models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, 41–47. Cambridge MA USA: ACM. ISBN 979-8-4007-0415-4.

Sija, B. D.; Goo, Y.-H.; Shim, K.-S.; Hasanova, H.; and Kim, M.-S. 2018. A Survey of Automatic Protocol Reverse Engineering Approaches, Methods, and Tools on the Inputs and Outputs View. *Security and Communication Networks*, 2018(1): 8370341.

Team, G.; Georgiev, P.; Lei, V. I.; Burnell, R.; Bai, L.; Gulati, A.; Tanzer, G.; Vincent, D.; Pan, Z.; Wang, S.; et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

Wang, C.; Scazzariello, M.; Farshin, A.; Ferlin, S.; Kostić, D.; and Chiesa, M. 2024. NetConfEval: Can LLMs Facilitate Network Configuration? *Proceedings of the ACM on Networking*, 2(CoNEXT2): 1–25.

Yen, J.; Lévai, T.; Ye, Q.; Ren, X.; Govindan, R.; and Raghavan, B. 2021. Semi-automated protocol disambiguation and code generation. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 272–286. Virtual Event USA: ACM. ISBN 978-1-4503-8383-7.

Yin, P.; and Neubig, G. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In Barzilay, R.; and Kan, M.-Y., eds., *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 440–450. Vancouver, Canada: Association for Computational Linguistics.