Tools for Disambiguating RFCs

Jane Yen
University of Southern California
yeny@usc.edu

Ramesh Govindan
University of Southern California
ramesh@usc.edu

Barath Raghavan
University of Southern California
barathra@usc.edu

Abstract

For decades, drafting Internet protocols has taken significant amounts of human supervision due to the fundamental ambiguity of natural language. Given such ambiguity, it is also not surprising that protocol implementations have long exhibited bugs. This pain and overhead can be significantly reduced with the help of natural language processing (NLP).

We recently applied NLP to identify ambiguous or underspecified sentences in RFCs, and to generate protocol implementations automatically when the ambiguity is clarified. However this system is far from general or deployable. To further reduce the overhead and errors due to ambiguous sentences, and to improve the generality of this system, much work remains to be done. In this paper, we consider what it would take to produce a fully-general and useful system for easing the natural-language challenges in the RFC process.

CCS Concepts

• **Networks** → **Formal specifications**.

Keywords

natural language, protocol specifications

ACM Reference Format:
Jane Yen, Ramesh Govindan, and Barath Raghavan. 2021. Tools for Disambiguating RFCs. In *Applied Networking Research Workshop (ANRW '21), July 24–30, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3472305.3472314

1 Introduction

It has long been the case that protocol behaviors are discussed and debated in the networking community for years before they are codified. The Internet Architecture Board (IAB) [9], Internet Engineering Task Force (IETF), and Internet Research Task Force (IRTF) [12] enable groups of independent networking professionals to draft Request for Comments (RFCs) to explain their networking ideas in English. Their aim is to publish their ideas globally and, typically, to codify those ideas as a networking standard. Draft RFCs are carefully evaluated and reviewed by members of working groups and editors; these individuals manually consider editorial and technical perspectives. Though the process by which RFCs are developed is effective, and can identify significant vague, incorrect, or partial descriptions and get authors to their drafts, the overall overhead of this human supervision is significant.

RFC editors follow a number of guidelines to review RFC drafts and provide feedback to the authors; authors typically aim to satisfy the guidelines as quickly as possible so that the back-and-forth review process can be minimized. The guidelines mostly address what sections are required to be present, what order the sections should be in, what minimal information should be stated clearly, and so forth. While these guidelines help drafts across time and space to maintain a uniform style, these guidelines are mostly writing advice and do not effectively help authors or reviewers to identify technical issues. To tackle technical concerns in any RFC draft requires participants' professional background knowledge.

RFC authors have many ways to describe technical details, including natural language, pseudocode, formal specifications, real code, state machine diagrams, and more. While pseudocode, formal specifications, and real code can provide precise execution steps, natural language is still preferable due to its flexibility and readability. However, natural language can also be fuzzy, which sometimes leads to a fundamental technical problem. For example, *"the type code changed to 0"* is a verbatim sentence in Internet Control Message Protocol (ICMP) RFC [23]. The noun phrase "type code" can confuse a reader as to whether it refers to as a type named "code" or a code named "type" or a specific protocol header field named "type code". If any reader interprets the noun phrase incorrectly, a packet may be generated incorrectly and get dropped by a receiver.

To strike the right balance when using natural language, we might ask the following question: **Can we systematically identify natural language ambiguity in network protocol specifications and make changes accordingly?** The answer to this question would help many stakeholders in the context of protocol specifications. For example, working groups and standards writers could leverage techniques to avoid or reduce back-and-forth

ANRW '21, July 24–30, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8618-0/21/07.
https://doi.org/10.1145/3472305.3472314

communication with RFC editors to speed up the RFC publication process. RFC reviewers could benefit from less work to pinpoint textual ambiguity. Those who build reference implementations could leverage such automated ambiguity analysis to develop assertions and/or unit tests, or relax some constraints to accommodate different implementations for interoperability (Figure 1).

In this paper, we first review how a typical RFC is written and reviewed, and consider quantitatively how many drafts are reviewed per year. Then we consider what tools have the potential to improve the whole RFC authoring and publication process. We envision additional tools that can be used to reduce ambiguities from natural language descriptions and can be used to generate code, if applicable, for verifying logical/functional objectives. We then consider both what aspects of ambiguity have been addressed in prior research and what remains to be understood. In this context, we introduce our recent work, SAGE [27], which identifies ambiguities in protocol specification text with Combinatory Categorial Grammar (CCG) [3] analysis, disambiguates with set of rules, and, given unambiguous specification text, performs per-sentence translation to C++ code. In that work we considered how to handle packet generation and session management across multiple protocols. However, SAGE is not fully general nor is it usable by the average RFC author or editor. Thus, finally, we discuss what remains to be explored to further reduce the effort required of a human editor.

2 Overview

In this section, we describe what is currently entailed in writing an RFC (though our description is oversimplified). With an understanding of which procedure takes significant human effort, we point out what mechanisms would be helpful to improve the end-to-end RFC publication process.

2.1 RFC drafting and reviewing process

An RFC is generated to describe one or more networking behaviors, where such behaviors are usually related to how a networking system/protocol works. For RFC authors/drafters to design such a system/protocol, one common approach is to alternate between spec writing and implementation, revising both until they are satisfied with a final version (Figure 2). After a draft is generated it is often shared within a working group and then eventually will make its way to an RFC editor for review. An RFC editor may perform similar steps to comment on RFC drafts, to consider any unclear/confusing descriptions; the editor may request the RFC's authors revise according to those comments. In some cases, when a reference implementation is provided, software verification methods can also be applied to discover design flaws. This potentially assists RFC authors to reflect changes in both RFC drafts and system code.

YEAR	2016	2017	2018	2019	2020
RFC COUNT	310	263	208	180	209

Table 1: RFCs reviewed per year from 2016 to 2020.

2.2 Human effort in specification review

To understand how much effort is put into publishing RFCs each year, we collected the number of reviewed RFCs from the RFC editor site (Table 1) [11]. We note that an RFC document is usually tens of pages of text. Therefore, roughly speaking, the networking community is drafting and reviewing thousands of pages of specification text in total every year. Many of these drafts likely contain language ambigutity, making this work quite challenging.

2.3 Tools in the publication process

We identify two additional kinds of tools that can be added in the RFC publication process to reduce the work that must be done by the people involved in the process.

Ambiguity identification. In the process of drafting and reviewing RFCs (Figure 2), RFC drafters may encounter a problem: readers may interpret a statement inconsistently with respect to the drafters' intention. Before reaching agreement with editors or reviewers, drafters have to repeat the process of editing drafts, waiting for reviews, and repeating this process through multiple rounds of feedback and revision. A tool that can reduce the tedious nature of this process may be one that can identify ambiguities in natural language sentences.

Compilation from natural language to code. RFC drafters commonly write their system or reference code and RFC drafts in parallel. While the drafters may believe their translations between RFC drafts and reference code are consistent, they cannot always guarantee that natural language sentences and implementations are logically equivalent. When such inconsistency occurs, as it always does, if no reference code is provided, readers of RFCs have to depend on natural language sentences and implement a potentially non-interoperable implementation. Even if reference code is provided, readers may also get confused about whether to depend upon the natural language text or the reference code, which may lead to the specification and the protocol "in the wild" diverging. To reduce the frequency of such an inconsistency problem, one useful tool could be a compiler that takes in natural language descriptions and turns the descriptions into intermediate representation such as pseudocode, formal specifications, or executable code. Such a tool need not provide a unique implementation of the specification, but simply provide a piece of code that can used to verify if the functionality is logically equivalent.

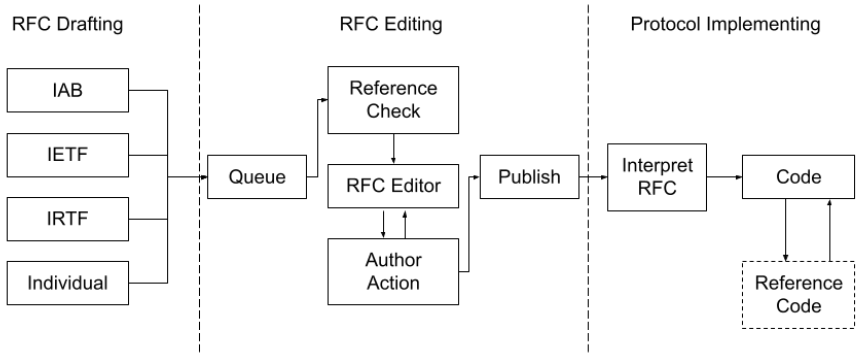

Figure 1: The RFC editing process.

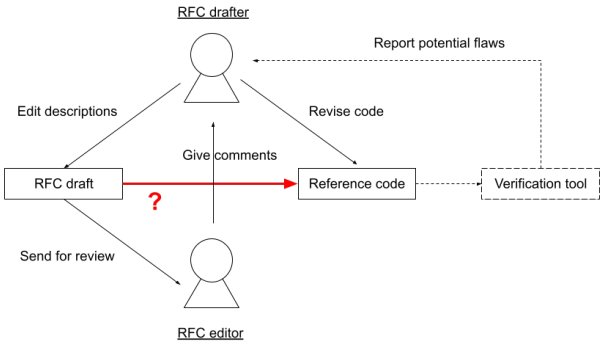

Figure 2: The process of drafting and reviewing RFCs.

As we previously mentioned, since RFC drafters may implement their system alongside the specification, neither may be representative of the community's norms or expectations. Building an implementation without a specification can be challenging. With a compiler that converts natural language into some form of code, RFC drafters can leverage the compiler's output and use verification tools to discover hidden bugs, speeding up the process of perfecting the design and reflecting the requisite changes in RFC drafts.

3 Related Work

3.1 Formal specification languages

Over the past few decades, numerous protocol languages have been proposed to mitigate the challenges of using natural language. Estelle [8] and LOTOS [6] provided formal descriptions for OSI protocol suites to properly specify protocol behaviors. Moreover, Estelle used finite state machine specs to depict how protocols communicate in parallel, passing on complexity, unreadability, and rigidity [7, 25, 26]. Other research such as RTAG [2], x-kernel [13], Morpheus [1], Prolac [16], Network Packet Representation [21], and NCT [20] gradually improved readability, structure, and performance of protocols, spanning specification, testing, and implementation. However, the networking community has found through experience that English-language specifications are more readable than such protocol languages.

3.2 Protocol analysis

Some previous research also developed techniques to reason about protocol behaviors in an effort to minimize bugs. One such direction is to use specifications of finite state machines, higher-order logic, or domain-specific languages to verify protocols [4–6]. Another thread of work [14, 15, 18] explores the use of explicit-state model-checkers to find bugs in protocol implementations. This thread also inspired work (*e.g.,* [22]) on discovering non-interoperabilities in protocol implementations. While we share a similar goal—to identify the bugs—our focus is end-to-end, from specification to implementation, and on identifying where ambiguity in specifications leads to bugs.

3.3 Semantic parsing and code generation

Some previous efforts proposed to use deep-learning based approaches in semantic parsing [10, 17, 29] and in certain types of automatic code generation [19, 24, 28]. Although such methods have proven effective, they cannot be directly applied to our task. First, deep learning typically trains in a black-box manner. Since we aim to identify ambiguity in specifications, we aim to interpret intermediate steps in the parsing process and maintain all valid parsings. Second, such methods require large-scale annotated datasets. It is impractical to collect sufficient high-quality data that maps network protocol specifications to expert-annotated logical forms (for supervised learning).

4 SAGE

A tool for ambiguity identification should be able to analyze a sentence and present how it can be interpreted with more than one semantic meaning. As for implementing a compiler that compiles a natural language specification into

code, this compiler is required to handle multiple elements in the specifications, where each element can have different assumptions to be converted into executable code.

Our recent work, SAGE [27], is a significant first step towards implementing the two potentially useful tools we described earlier. First, SAGE enables per-sentence ambiguity analysis by leveraging the CCG approach to extract semantic meanings of sentences, and by leveraging domain-specific knowledge rules to disambiguate nonsense results of semantic extraction. Second, SAGE shows its ability to handle a number of common elements across multiple RFCs.

4.1 Semantic parsing

Semantic parsing is the task of extracting meanings from natural language utterances. While much prior work developed semantic parsing tools and methods, most select only one meaning as a result. However, to satisfy our needs in identifying ambiguities, we want to store any valid parsing results and justify whether a result shall be kept or not. For this reason, we use the Combinatory Categorial Grammar (CCG) formalism, which enables coupling syntax and semantics in the parsing process and retains the flexibility to customize hand-crafted lexicons for domain-specific terminologies (for example, in the context of a networking protocol, terms like *one's complement* or *checksum*).

CCG introduction. The CCG formalism relies on combinatory logic to combine entities in a sentence. Each entity is specified with syntax to explain how it can combine with neighboring entities and its semantics with lambda expressions. CCG parsing generates *logical forms* that capture the semantics of the phrases. For example, we added syntax and semantics for each word in a sentence "checksum is zero". CCG parsing outputs us a logical form {S: @Is("checksum", @Num(0))}, which shows a checksum variable exists with an assignment relationship to a value of zero.

Running CCG parsing. We run CCG parsing on each sentence of an RFC. Ideally, an unambiguous sentence results in exactly one logical form. In practice, a CCG parser may output zero or more logical forms, some of which are due to limitations in the parser itself and some from ambiguities inherent in a sentence.

4.2 Disambiguation

Using the CCG parsing results (i.e. how many logical forms are generated) directly as the criteria to define ambiguity might not provide RFC drafters a clear suggestion of how to remove that ambiguity. Some parsing results actually represent exactly one semantic meaning after recovering some information and/or fixing CCG's own limitations. Filtering those results could leave the RFC drafters with true ambiguities they should edit.

Name	Description
♦ **Packet Format**	Packet anatomy (i.e., field structure)
♦ **Field Descriptions**	Packet header field descriptions
♦ **Constraints**	Constraints on field values
♦ **Protocol Behaviors**	Reactions to external/internal events
System Architecture	Protocol implementation components
+ **State Management**	Session information and/or status
Comm. Patterns	Message sequences (e.g., handshakes)

Table 2: Protocol specification components. SAGE supports those marked with ♦ (fully) and + (partially) [27].

Cases of zero logical form. Zero logical form cases are incomplete sentences (due to missing subjects or incorrect grammar). For missing subjects, we often can recover this information from analyzing structural information (for example, content hierarchy or descriptive lists). Ideally, we expect that drafters would strive to use complete sentences, but in practice some sentences have missing subjects. As long as the missing subject information can be recovered with other structural information from an RFC, we do not consider missing subjects as a kind of ambiguity. On the other hand, if no additional information can assist us to recover the information, such sentences are considered to have incorrect grammar which cannot be processed by CCG successfully and is considered as a true ambiguity. In such cases, SAGE triggers an alarm that the sentence needs to be revised to resolve ambiguity.

Cases of more than 1 logical form. Cases with more than 1 logical form can be due to limitations in CCG parsing itself or true ambiguities inherent in a sentence. Our aim is to find out only true ambiguity that could lead to different semantic meanings. Therefore, those cases that are due to CCG parsing limitations shall be filtered (for example, entities are combined in a logical form whose semantic meaning makes no sense; entities can switch their orders in a logical form without messing up the semantic meanings; logical forms with the same semantic meanings can be expressed in a verbose or a concise form). SAGE applies sets of rules to disambiguate those logical forms.

4.3 Code generation

RFCs are used to describe networking behaviors from diverse perspectives. Protocol RFCs are a kind of RFC that come with the intent to program the protocol design as executable code. A protocol RFC contains multiple elements (Table 2), where each component contributes to part of the whole protocol code. From Table 2, we marked the components that SAGE can process from sentence description together with structural and/or syntactical information to executable C++ code.

SAGE's code generator takes a logical form as an input, and uses a post-order traversal of the logical form obtained after disambiguation to convert the relations of entities into C++ code. For example, "@Is("checksum", @Num(0))" has an assignment relation between variable "checksum" and number 0, and SAGE outputs a line of executable code "hdr->checksum = 0".

Generality. We evaluated SAGE's functioning across multiple protocols for the specified components (Table 2), including ICMP and partial support for IGMP, NTP, and BFD. We verified that SAGE-generated code can interoperate with standard implementations of `tcpdump`, `ping`, and `traceroute`. In addition, SAGE shows the ability to parse complicated components *e.g.,* state management.

5 Challenges

While SAGE is a significant first step to improve the RFC publication process, there remain many challenges to be addressed. Thus far, SAGE can parse one sentence at a time and establish rules to identify ambiguities. SAGE can also compile an unambiguous sentence into a line of code, and sequentially execute a block of code when the descriptions contain clear instructions of execution order. Below, we discuss what challenges could be considered for future directions that build upon SAGE.

Paragraph analysis instead of per-sentence analysis. SAGE mostly parses sentence by sentence to determine a sentence's ambiguity and generates code according to the order of descriptions. Therefore, SAGE has some parsing limitations in some cases. For example, a sentence might use a pronoun and there could be multiple candidates that the pronoun refers to. The candidates could be discovered from multiple neighboring sentences that are within the same paragraph. If more than one candidate exists, this can be considered as a kind of ambiguity that can confuse RFC readers.

When a compiler compiles a paragraph of text, it is possible that the order of descriptions do not follow sequential execution order. For example, a protocol RFC can describe handshakes that resemble a state machine, with all the involved connection states and their transitions are described in a large paragraph. The compiler should no longer assume the generated code should follow the description order to convert sentence by sentence. Instead, the compiler should be aware that all connection states are of equal importance to be executed, and the generated code should be able to switch to any state for execution given the current state. In other words, the compiler has to learn the context of a whole paragraph, determine a correct code block template, and decide whether a variable value will be reused for the next event.

Semantic Meaning and Classifications SAGE has evaluated a number of protocol RFCs with successful code generation. However, the type of generated code remains limited. SAGE can parse descriptions that assign, associate, and rewrite values and simple if-else statements. While these sets of operations have covered a large portion of system code, there are other types of code that should be considered such as code for asserting values, code for adding constraints, code for logging events, etc.

Although we can use SAGE to disambiguate sentences and generate exactly one logical form, the conversion from logical form to code should not be limited to exactly one kind of code. We have to identify when/whether a description can be interpreted as more than one type of code, and determine what the execution order is when different types of code coexist. In other words, classification of semantic meanings also takes an important role in code generation.

Mis-matched/ Mis-captured behaviors. Among RFC components, many components (such as packet formulation or state machine context) are presented with both text and syntactical components, where syntactical components can be diagrams, listings, tables or figures. In some cases, textual sentences/paragraphs are used to explain what the syntactical components represent, or extend what syntactical components have covered. Thus, text and syntactical components are complementary to each other. In other cases, due to mistakes or some other reason, texts and syntactical components can be inconsistent. This situation causes confusion for the RFC reader whether to believe in textual descriptions or the meaning of syntactical components.

When it comes to code generation for any case, the process of code generation has to (1) correctly associate the same semantic meanings parsed from textual descriptions and syntactical components, (2) identify any missing semantic meanings from either representations, and (3) identify discrepancies between texts and syntactical components. The generated code should not repeat the same semantic meaning output, or miss any mentioned semantic meanings. The code generator should also report/alarm discrepancies to RFC drafters to reduce confusion.

Alternative code representations. The aim of the compiler (§2.3) is to accept natural language specifications and turn them into any structured representation, which can include pseudocode, formal specification language text, and implementations in a variety of programming languages. Thus far, SAGE has illustrated the possibility to turn natural language texts into working C++ code. While every kind of

representations has its advantages, the conversion among different representations could face different challenges. For example, RFC drafters commonly put pseudocode snippets in their drafts to better explain their ideas to their readers, but pseudocode is not executable and maintains some level of flexibility in the expressions given. If pseudocode is generated by the compiler and a drafter would like to compare its logical behavior with a manually written pseudocode implementation, what the criteria is to determine whether the functionalities are equivalent. For another example, some specifications expressed in the protocol's design in TLA+; the TLA+ language itself has different expressions than logical form and C++, and future work is required to identify the limitation of compiling natural language to TLA+.

Standalone RFC or multiple RFCs. Ideally, every perspective of a protocol can be completely expressed in a single RFC. In practice, a protocol can be explained over multiple RFCs for the ease of reading by topic. For example, one RFC may describe the functionality of the protocol and explain the design of packets, and a separate complementary RFC may explain what each value represents for the fields presenting in the protocol and what additional constraints would be under a different scenario.

When multiple RFCs are given, we need to discover how to merge the content by concept without missing or violating any constraints that may exist. From the perspective of an RFC drafter, the drafter must confirm the consistency not only within a standalone RFC but also across all relevant RFCs. Any discrepancy among RFCs could similarly lead to interoperability issues. An ideal compiler should thus take inconsistency into consideration and automatically compare/label/alarm where the discrepancies happen.

Single protocol or stack of protocols. While RFC specifications are usually limited to the discussion of a single protocol, this single protocol has to interact/stack cleanly with other protocols when it's applied in a networked system. For example, the ICMP RFC describes its design, but it has to be built on top of IP. In such cases, a compiler that is considering both protocols together first has to identify the dependency between the two protocols and/or the constraints to be considered. When the compiler is stacking the two protocols together, it should be aware of, for example, the general IP header description selecting the exact value for ICMP as its next protocol field, and that the total length field in the IP header needs to add the ICMP header and the payload.

Logic vs. performance. A protocol RFC usually focuses on describing its logical functionality and leaves the flexibility of implementing code to any reader of the RFC (unless the RFC provides a reference implementation and expects the future protocol implementer to directly apply the reference implementation in all contexts). In other words, we could reasonably expect the generated code from the compiler is valued for its correct logic instead of its performance. If a drafter suggests a performance-oriented implementation, some mechanism could be supported to identify which natural-language statements convey performance considerations and which convey logical considerations. Moreover, the compiler can optionally output different versions of generated code according to the needs of the user.

6 Discussion

We have mentioned a number of problems yet to be solved. Here we discuss some existing techniques that might assist us to resolve them with proper modification to SAGE.

For example, to handle multiple candidate problems, we might employ tools to identify what terms/tokens are used to refer the same concept, which is commonly recognized as a "coreference resolution" problem. Although the coreference resolution problem has been studied for years and the performance of tools for this task is generally considered good, we find generic coreference resolution tools are still insufficient to handle many domain-specific terms. In particular, such tools do not perform well when multiple terms seemingly are the same but are actually different because of naming styles *e.g.,* ADMINDOWN, admin_down, AdminDown. The generic coreference resolution tools provide inflexible APIs, which disable customization rules that could group terms together. If we would like to resolve such problems, we could pre-process text, or post-process results to address the limitations of existing tools.

As another example, to handle multiple RFCs that combine to form a single protocol, one method is to leverage proper markdown or reference links to identify the relationships between RFCs. Given proper structural information (*e.g.,* titles) and categorized labels, the same specification documents can be analyzed together and used to discover whether there are any discrepancies.

We do not mean to claim that all the mentioned challenges can be easily solved; these examples still need to undergo comprehensive discussion and evaluation to prove their effectiveness. Our focus in this paper is to provide some perspectives from our experience and identify a number of research directions that can be explored that may be of benefit to the networking standards development community.

References

[1] Abbott, M. B., and Peterson, L. L. A language-based approach to protocol implementation. *IEEE/ACM transactions on networking* (1993).

[2] Anderson, D. P. Automated protocol implementation with rtag. *IEEE Transactions on Software Engineering 14*, 3 (1988), 291–300.

[3] Artzi, Y., FitzGerald, N., and Zettlemoyer, L. S. Semantic parsing with combinatory categorial grammars. *ACL (Tutorial Abstracts) 3*

(2013).

[4] Bhargavan, K., Obradovic, D., and Gunter, C. A. Formal verification of standards for distance vector routing protocols. *Journal of the ACM (JACM) 49*, 4 (2002), 538–576.

[5] Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., and Wansbrough, K. Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications* (2005), pp. 265–276.

[6] Bolognesi, T., and Brinksma, E. Introduction to the iso specification language lotos. *Computer Networks and ISDN systems 14*, 1 (1987).

[7] Boussinot, F., and De Simone, R. The esterel language. *Proceedings of the IEEE 79*, 9 (1991), 1293–1304.

[8] Budkowski, S., and Dembinski, P. An introduction to estelle: a specification language for distributed systems. *Computer Networks and ISDN systems 14*, 1 (1987), 3–23.

[9] Daigle, L., and Internet Architecture Board. Process for Publication of IAB RFCs. RFC 4845, 2007.

[10] Dong, L., and Lapata, M. Coarse-to-fine decoding for neural semantic parsing. *arXiv preprint arXiv:1805.04793* (2018).

[11] Editor, R. Number of rfcs published per year. https://www.rfc-editor.org/rfcs-per-year/.

[12] Falk, A. Definition of an Internet Research Task Force (IRTF) Document Stream. RFC 5743, 2009.

[13] Hutchinson, N. C., and Peterson, L. L. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software engineering*, 1 (1991), 64–76.

[14] Killian, C., Anderson, J. W., Jhala, R., and Vahdat, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)* (2007), NSDI, USENIX Association.

[15] Killian, C. E., Anderson, J. W., Braud, R., Jhala, R., and Vahdat, A. M. Mace: language support for building distributed systems. *ACM SIGPLAN Notices 42*, 6 (2007), 179–188.

[16] Kohler, E., Kaashoek, M. F., and Montgomery, D. R. A readable tcp in the prolac protocol language. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication* (1999), pp. 3–13.

[17] Krishnamurthy, J., Dasigi, P., and Gardner, M. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing* (2017), pp. 1516–1526.

[18] Lee, H., Seibert, J., Killian, C. E., and Nita-Rotaru, C. Gatling: Automatic attack discovery in large-scale distributed systems. In *NDSS* (2012), Citeseer.

[19] Lin, X. V., Wang, C., Pang, D., Vu, K., and Ernst, M. D. Program synthesis from natural language using recurrent neural networks. *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03-01* (2017).

[20] McMillan, K. L., and Zuck, L. D. Formal specification and testing of QUIC. In *Proceedings of ACM SIGCOMM* (2019).

[21] McQuistin, S., Band, V., Jacob, D., and Perkins, C. Parsing protocol standards to parse standard protocols. In *Proceedings of the Applied Networking Research Workshop* (New York, NY, USA, 2020), ANRW '20, Association for Computing Machinery, p. 25–31.

[22] Pedrosa, L., Fogel, A., Kothari, N., Govindan, R., Mahajan, R., and Millstein, T. Analyzing protocol implementations for interoperability. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)* (2015), pp. 485–498.

[23] Postel, J. Internet Control Message Protocol. RFC 792, 1981.

[24] Rabinovich, M., Stern, M., and Klein, D. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535* (2017).

[25] Sidhu, D., and Chung, A. *A formal description technique for protocol engineering*. University of Maryland at College Park, 1990.

[26] von Bochmann, G. *Methods and tools for the design and validation of protocol specifications and implementations*. Université de Montréal, Département d'informatique et de recherche …, 1987.

[27] Yen, J., Lévai, T., Ye, Q., Ren, X., Govindan, R., and Raghavan, B. Semi-automated protocol disambiguation and code generation. In *Proceedings of ACM SIGCOMM* (2021).

[28] Yin, P., and Neubig, G. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696* (2017).

[29] Yin, P., Zhou, C., He, J., and Neubig, G. Structvae: Tree-structured latent variable models for semi-supervised semantic parsing. *arXiv preprint arXiv:1806.07832* (2018).