Large Language Models for Automated Network Protocol Testing: A Survey

Vy Dang Phuong Nguyen
Dept. of Computer Science
University of the Pacific
Stockton, CA, USA
v_nguyen110@u.pacific.edu

Sepehr Amir-Mohammadian
Dept. of Computer Science
University of the Pacific
Stockton, CA, USA
samirmohammadian@pacific.edu

***Abstract*—This paper surveys state-of-the-art methods leveraging Large Language Models (LLMs) for automated test case generation in network protocols, with a focus on addressing ambiguities in protocol specifications and enhancing test coverage. Traditional approaches face challenges in scaling and ensuring thorough compliance. LLMs, with their natural language understanding capabilities, offer a promising solution by automating the extraction of specifications from Request for Comments (RFCs) and generating structured models and test cases. Relevant studies, tools and frameworks are reviewed and analyzed to understand how LLMs are used in this area. The paper discusses their strengths and limitations, aiming to support further research on LLMs to improve the reliability and efficiency of network protocol testing.**

***Index Terms*—Large Language Models, Network Protocols, Request for Comments, Testing**

I. Introduction

Advancements in artificial intelligence and machine learning have revolutionized the field of software engineering. The development of large language models (LLMs), notably OpenAI's GPT-3, introduced in 2020, has achieved excellent results in tasks such as code generation [1]. Powered by the later improved version of GPT-3, ChatGPT gained rapid popularity worldwide by reaching 123 million monthly active users within less than three months of its launch in November 2022 [2]. ChatGPT outperforms other learning-based techniques in terms of syntactic, compilation, and execution correctness [3]. LLMs such as ChatGPT have shown their potential to enhance the efficiency of software testing, thus ensuring the quality, reliability and compliance of programs with industry standards.

Established software testing methods, including unit tesing and model-based testing, are prevalent in finding bugs to ensure software reliability. However, traditional applications of these methods often require significant manual effort in planning, writing, and evaluating tests, especially for complicated systems such as programmable logic controllers and distributed control systems [4].

Public unit testing frameworks and tools such as TcUnit [5] and CODESYS Test Manager [6] can automate test execution, but they still rely on manually written test cases. To address this challenge, newer iterations of LLMs, including OpenAI's GPT-4 [7], are capable of generating test cases with high statement coverage for low-to-medium complexity software programs [4]. Building on more advancements in LLMs, algorithms such as CODAMOSA [8] leverage OpenAI's Codex [9] model to automatically generate test cases for areas of the program where traditional search-based software testing methods stall.

Despite recent advances in applying LLMs to software testing, their use in network protocol implementation testing, particularly in generating test cases from Request for Comments (RFC)-based specifications remains underexplored. RFCs are formal documentations that define the standards for internet protocols [10]. They are written in natural language, which can be ambiguous and makes it challenging to write tests without first understanding the protocol specifications. Protocol implementation testing is crucial for identifying bugs that may lead to incorrect or inconsistent behavior, security vulnerabilities, and more [11]. This survey paper reviews a range of methods and tools related to generating test cases from RFC-based specifications, including manual approaches and recent work that apply LLMs to related tasks. While some methods do not fully automate test generation directly from RFCs using LLMs, frameworks such as PROSPER [12] use LLMs to extract protocol finite state machines (FSMs) from RFCs, which could be used in further tasks including test generation. By analyzing these tools and frameworks, this paper provides an overview of the current state of research, identifies existing challenges, and explores opportunities for advancing network protocol testing through LLMs. The goal is to provide a comprehensive overview of the current approaches and highlight areas for future research in automated protocol testing.

The paper is organized as follows: Section II outlines research questions and methodology. Section III provides a background on manual and automated testing approaches. Section IV reviews network protocol testing methods, with a focus on recent tools and approaches that use LLMs. Section V outlines future research directions, and Section VI concludes the paper.

II. Methodology

This section outlines the approach taken to conduct a systematic and focused survey of literature related to the use of LLMs in network protocol testing. Our goal is to identify and analyze trends, methodologies, tools, and research gaps in this emerging area.

A. Research Questions

This survey paper aims to answer the following questions about protocol testing with a focus on the use of LLMs.

RQ1: What methods currently exist for testing network protocol implementations?

RQ2: To what extent are LLMs being used in automating network protocol testing?

RQ3: What are the strengths and limitations of state-of-the-art tools and approaches?

RQ4: What are the future directions for automating protocol testing using LLMs?

B. Search Strategies

To ensure a comprehensive survey of related work, we followed three steps.

First, we searched through online academic databases to find peer-review articles, survey papers and conference papers. The primary used databases were Google Scholar, IEEE Xplore, arXiv, and ACM Digital Library. We queried for papers using a combination of keywords such as "automated test generation", "protocol testing", "LLM for protocol testing", and "RFC compliance". The search preference was given to papers published from 2020 to 2025 to ensure the content remains relevant and up to date. However, earlier studies were also included if they described tools or methods still significant to the topic. Many of the sources reflect recent work on LLMs and their applications in automated testing, including developments in state-of-the-art models.

Second, we reviewed abstracts of papers to filter out papers that were:

1) not related to network protocol, LLMs, or software testing;
2) not supported by results or evaluation of the proposed approaches or tools;
3) not accessible online;
4) not written in English.

Third, we selected the most relevant papers for in-depth analysis by examining each paper's objective, methodology, strengths and limitations of proposed approaches or tools, and its relevance to key themes such as the use of LLMs, protocol testing, and automated tools or frameworks.

III. Background

This section compares the manual and automated approach for generating test cases in the context of network protocol implementations.

A. Manual Approaches to Test Case Generation

In traditional software testing, test planning and effort estimation are done manually based on engineers' understanding of the system and its specifications. In a survey on software testing practices, 57% of testing experts implied that they do not use any tools for test estimation and instead rely on their own personal knowledge [13]. In the context of computer networking, RFCs outline specific structures and behaviors that a network protocol must follow, such as message formats, command syntax, or error handling. They usually consist of tens of pages of text, requiring engineers to carefully read and interpret the specifications, and manually translate them into tests [14]. While this approach allows customizing test cases for specific functionalities of the programs, it requires significant manual effort and can delay the development process. It is also prone to human errors, such as missing edge cases, inconsistent code coverage, or misinterpreting specifications, which can reduce the quality of the testing phase [15], [16].

B. Automated Approaches to Test Case Generation

To address these challenges, automated testing approaches are a more efficient alternative to manual methods. Techniques such as model-based testing [17]–[21], symbolic execution [22], [23], search-based testing [24], [25], fuzzing [26]–[31], and constraint-based testing [32]–[34] are widely studied. These approaches allow engineers to systematically explore input spaces, parse specifications, and uncover program behaviors that might be overlooked with manual efforts.

More recently, the use of LLMs in analyzing protocol specifications written in natural language, such as RFCs, has gained significant attention. LLMs are capable of automatically interpreting and translating these specifications into structured formal models [12], [35]. These outputs can be used to validate system behavior, particularly in cases where specifications are ambiguous or under-specified. Utilizing LLMs for test generation reduces the manual effort required for translating specifications and allows more accurate and consistent test cases, which ensures software compliance and reliability.

One relevant tool is SAGE [36], which specifically addresses ambiguities and under-specifications in protocol RFCs. It uses natural language processing (NLP) to parse and process key components such as packet format, pseudocode, communication patterns, interoperation, etc. Once ambiguities are resolved, SAGE can automatically generate executable protocol code. It has successfully been applied to the ICMP protocol and sections of other protocols, including BFD, IGMP, and NTP. However, SAGE still requires human intervention to rewrite ambiguous or under-specified sentences. It also faces difficulties in understanding protocols with complex state machine descriptions, e.g., TCP, or architectural communication patterns, e.g., BGP.

IV. Protocol Compliance Testing

This section reviews major automated testing methods for network implementations, highlighting the strengths and weaknesses of relevant frameworks and tools. In particular, we focus on approaches that have been explored in combination with LLMs.

A. Model-based Testing

Model-based testing (MBT) is a prevalent approach in network protocol testing by creating abstract models of the system and compare them against the implementations to detect compliance issues [37]–[40]. MBT is particularly effective for complex systems with well-defined behaviors, where manually creating test cases for every possible state would be impractical. It can cover different edge cases and conditions to ensure high coverage. For example, Li et al. [38] present an automatic testing framework for the HTTP/1.1 protocol that models both the server and network behavior using interaction trees. These trees are then executed to systematically generate test cases that target under-specified behaviors in RFCs. Their framework uncovered several violations of RFC 7232 in Apache and Nginx. However, one significant limitation of MBT is that generating models that accurately represent the behavior of time-sensitive processes and systems can be challenging [19]–[21]. This is particularly relevant to protocols such as TCP and QUIC, which rely on asynchronous message exchange and time-sensitive operations.

Another type of such abstract models are FSMs, which represent the system through a set of states and transitions. Pacheco et al. [35] present a methodology for testing network implementations by automatically extracting FSMs from RFC specifications. Their approach begins by training a model on 8,858 RFCs to learn the distributed word representation of the technical language used in these documents. Next, they apply a zero-shot learning approach to extract structured protocol information, which is then converted into an FSM specific to that protocol. They successfully extract FSMs for six different protocols, with the highest exact match accuracy of 98.73% for PPTP. These FSMs are applied to synthesize attacks against protocols such as TCP and DCCP. However, the approach has limitations in extracting complete models of the protocol behavior, known as canonical FSMs, due to the ambiguity and under-specifications in RFCs, which may result in incomplete state coverage or incorrect transitions.

Sharma et al. [12] propose a similar framework named PROSPER that uses GPT-3.5 Turbo [41] to automate the extraction of FSMs from RFCs. They developed a helper tool named ArtifactMiner that can extract non-textual artifacts such as diagrams, topologies or message structures. Their approach achieved 1.3x more true positives and 6.5x fewer false positives than existing approaches for the DCCP protocol. The extracted FSMs are proposed to be used in further tasks such as intrusion detection and protocol interface generation. By automating FSM extraction, PROSPER supports model-based testing by reducing the manual effort in modeling protocol behaviors. However, PROSPER's utilization of an LLM introduces limitations such as inconsistent output formats, occasional hallucinations, and misinterpretation of the protocol context.

Another notable work on MBT is IVy, a formal verification tool used to specify, model, implement and verify protocol requirements [42], [43]. First, engineers manually write formal specifications of the system such as packets, frames, and connection states in IVy. Then, IVy uses these specifications to generate randomized test cases, which are run against real-life implementations to verify whether the system complies with the protocol.

IVy has been applied to verify the QUIC protocol [44] as well as a blockchain consensus protocol [45]. Using IVy, Crochet et al. [44] extended the formal IVy model of QUIC to include more specifications such as transport error codes and connection ID management. They applied the generated tests against 7 existing QUIC implementations and discovered the inconsistencies across them. These insights are incredibly valuable for improving both the implementations and the QUIC specification itself.

However, IVy struggles to model time-sensitive behaviors, a limitation previously noted in MBT, due to its lack of support for presenting time-related properties. To address this, Rousseaux et al. [46] propose a novel MBT-based methodology called Network Simulator-Centric Compositional Testing (NSCT) to address time-varying network properties in protocol testing using IVy. NSCT works by combining formal models in IVy with a Shadow network simulator to generate and execute tests against network implementations. This approach was used to uncover an error in picoquic, an implementation of QUIC. Despite its strengths, IVy requires manual effort to define and update formal specifications from RFCs, which can limit its scalability for protocols with frequent updates. This limitation highlights an opportunity for future work explored further in Section V.

B. Symbolic Execution

Symbolic execution is an automatic software testing method that explores different execution paths of the program by treating inputs as symbolic variables instead of fixed values [22]. It uses constraint solvers to generate inputs that satisfy specific path conditions and use them to generate test cases. This approach is particularly useful for uncovering edge cases and logic errors that might not be found with random or manually written test cases.

Song et al. [47] present SYMBEXNET, a framework that combines symbolic execution and rule-based specifications from RFCs to test network protocol implementations. While it does not incorporate LLMs, it can automatically generate test input packets that explore deep execution paths in DHCP and Zeroconf implementations. These inputs are validated against manually written rules that describe protocol behaviors to detect semantic bugs, implementation flaws, and interoperability issues across 5 network daemons. SYMBEXNET uncovered 39 unique bugs across these implementations and achieved significantly higher code coverage compared to random testing. However, it is limited to C-based implementations and relies on manual rule extraction from protocol documents, a challenge addressed by LLM-based approaches discussed in previous sections.

Fakhoury et al. [48] introduce 3DGen, a framework that uses LLM-based agents to convert protocol specifications, such as RFCs and packet examples, into a formal language called 3D. The framework uses symbolic execution to generate test inputs that might be overlooked by human-written specifications and validate them against test oracles such as Wireshark. This process allows 3DGen to identify errors, refine specifications, and distinguish between plausible formats. It is validated across 20 protocols including TCP, UDP, and IPv6. However, 3DGen's performance depends on the quality and completeness of the test oracles and is prone to syntax errors and misinterpretation from parsing ambiguous RFCs.

Karkala et al. [11] propose a new testing approach called SCALE (Small-scope Constraint-driven Automated Logical Execution) to address RFC compliance bugs in DNS implementations. The approach starts by using RFCs to create a logical model of the protocol behaviors. SCALE then performs symbolic execution on an executable version of the model to generate high-coverage tests that can be applied to many DNS implementations, including black-box systems.

Through this process, SCALE automatically generates both zone files and corresponding DNS queries based on the symbolic execution of a formal DNS model implemented in a modeling language called Zen. Based on this model, the authors introduce a testing tool called FERRET that runs the generated tests across 8 existing DNS implementations and compare their responses. FERRET has generated over 13,500 test cases, with 62% resulting in inconsistent behavior across existing implementations such as BIND, Knot, NSD, and PowerDNS. It also discovered 30 unique bugs, including crashes and violations of expected RFC behavior.

However, FERRET has some limitations, including generating valid zone files that follow the strict rules in RFCs. If the files are incorrect, the tests can fail before the DNS logic is even tested. The tool also limits the size of the DNS files it generates, which can cause it to miss bugs that only appear in larger programs. While FERRET does not integrate LLMs, it aligns with the goal of uncovering RFC compliance issues in network protocols through automated test generation. The authors' proposed SCALE approach demonstrates its ability in identifying critical bugs and ensuring RFC compliance with potential to extend its application beyond DNS.

C. Fuzzing

Fuzzing is a technique in detecting software vulnerabilities by sending malformed test data to the target program and monitor its behaviors [28]. Traditional fuzzing treats each data input independently, whereas protocol fuzzing must generate sequences of messages that follow the current state [29]. Fuzzing test cases can be mutation-based, where inputs are slightly modified, and generation-based, where inputs are constructed based on a model or grammar defined by the protocol.

Several traditional fuzzing tools have been developed specifically for testing network protocol implementations, which often incorporate variations of FSMs to improve validity and coverage. However, they do not utilize LLMs but instead rely on manual or rule-based methods, making it challenging to scale to ambiguous or complex protocol specifications. Prospex [49] is a protocol reverse engineering tool that uses dynamic taint analysis to extract state machines from network traffic. It analyzes how input data affects server behavior and uses the extracted state machines to guide fuzzing to uncover program vulnerabilities. Netzob [30] is another open-source tool that infers protocol specifications through lexical and syntactic analysis of network traffic. It generates state machines to simulate protocol behavior and perform fuzzing on a wide range of protocols, including both proprietary or undocumented protocols and well-documented ones such as HTTP, IP, and TCP.

A more recent effort by Meng el al. [31] introduces CHATAFL, a fuzzing engine that incorporates LLMs. Rather than extracting specifications from RFCs, it uses knowledge from GPT-3.5-turbo to construct grammars for protocol message types, mutate the inputs and generate message sequences using predicted protocol states. CHATAFL achieved 47.60% and 42.69% more state transitions in comparison to AFLNET and NSFUZZ fuzzers, respectively, when evaluated on protocol implementations from ProFuzzBench. It also discovered 9 new vulnerabilities in three existing protocol implementations, including Live555 (RTSP), ProFTPD (FTP), and Kamailio (SIP). However, CHATAFL faces certain limitations such as occasional inconsistencies for message types or session IDs, resulted by the LLM incorporation, and its evaluation is currently restricted to text-based protocols.

D. Limitations of LLMs for Protocol Testing

While LLMs have shown significant potential in advancing network protocol testing, they have key limitations that affect their accuracy. These models face challenges in parsing natural language, which can lead to incorrect or inconsistent inputs [50]–[53]. For example, LLMs may hallucinate false positives by extracting transition states that do not exist in the original RFC [12]. Even highly advanced LLMs, such as GPT-4, suffer from occasional hallucinations [54]. Furthermore, these models may generate incorrect or fragile outputs and face performance issues due to their reliance on external systems. Because of their probabilistic nature, these models may also produce different test cases for the same input prompt [55].

Another limitation of LLMs is contextual misunderstanding, meaning the model may misinterpret a prompt due to the lack of context. For example, in the case of PROSPER [12], the model confuses message structures with FSM states when extracting from PPTP. LLMs occasionally struggle with accurately interpreting and applying the existing logic for functions that are more complex [4]. Those limitations may result in inaccurate representations of how the system works, which can cause issues in the software implementation. Incorrect implementation of protocol specifications can lead to critical security bugs and miscommunication between systems [56], [57].

Despite these limitations, the use of LLMs in automating protocol testing can still be beneficial, as even incorrect outputs can help identify defects and test corner cases [58]. However, additional steps are needed to refine the input prompts, identify mismatches in the generated codes, and verify their functionalities. Manual interventions are still necessary to ensure the accuracy and consistency of LLM-generated code or specifications. This approach is also referred to as "human-in-the-loop", where humans would leverage AI to automate tasks while refining its outputs to ensure accuracy before finalizing the results [59]–[61].

V. Future Work

In this section, we describe different potential areas for future work, including:

1) Improving the accuracy of specification extraction: Current methods for extracting specifications from RFCs using LLMs can be enhanced to improve precision and address ambiguities. Future work could focus on fine-tuning domain-specific LLMs to better understand protocol-specific terminologies and structures. Additionally, exploring hybrid approaches that combine LLMs with rule-based techniques or human-in-the-loop validation could further enhance accuracy in this domain.

2) Formalization of complex protocols: Future efforts could focus on adapting the discussed approaches to handle more complex, multilayered protocols that involve time-sensitive or asynchronous interactions. Improved testing methods could account for these properties to better simulate real-world protocol behavior under various network conditions for more accurate and comprehensive protocol validation.

3) Integration of real-world testing scenarios: Beyond RFC-based specifications, future research could investigate the application of this method to real-world implementations, ensuring that generated test cases address not only theoretical compliance but also practical interoperability challenges. This could include incorporating real-time data, e.g., network logs or traces, to refine test case generation.

4) Automating feedback loops for test refine: Future work could explore creating an automated feedback approach that analyzes test execution results to iteratively refine tests generated by LLMs. This could include detecting failed test cases, analyzing the causes, and using reinforcement learning techniques to prompt the LLM to adjust its understanding or regenerate tests accordingly.

5) Scaling existing approaches to a broader range of protocols: Extending the discussed tools and frameworks to test protocols across various domains, e.g., IoT, telecommunications and blockchain, could demonstrate their versatility and scalability.

6) Evaluating performance and usability: Future work could systematically benchmark the proposed approach against traditional and other automated testing methodologies to quantify improvements in efficiency, coverage, and error detection.

VI. Conclusion

This paper provides a comprehensive survey of approaches that apply LLMs to automate test generation for network protocol implementations, with a focus on RFC-based specifications. LLMs have shown ability in reducing the manual effort required to interpret natural language specifications by generating structured models that support protocol compliance testing. LLMs have been also applied across a range of protocol testing methods, including model-based testing, symbolic execution and fuzzing. Additionally, recent work has proposed new approaches that further contribute to enhancing testing capabilities

While these approaches show significant results, LLMs still have certain limitations. They can struggle with ambiguity, hallucinations, inconsistent outputs, and contextual misunderstanding. The primary objective of this paper is to establish a foundation for future research by reviewing existing tools and approaches, evaluating their capabilities, and identifying challenges in the application of LLMs to network protocol testing. As LLM capabilities continue to advance, future work can build on these foundations to explore new opportunities for enhancing the scalability, accuracy, and real-world applicability of protocol testing.

References

[1] M. Zhang and J. Li, "A commentary of GPT-3 in MIT technology review 2021. fundamental research, 1 (6), 831-833," 2021.

[2] J. Rudolph, S. Tan, and S. Tan, "War of the chatbots: Bard, Bing Chat, ChatGPT, Ernie and beyond. the new AI gold rush and its impact on higher education," *Journal of Applied Learning and Teaching*, vol. 6, no. 1, pp. 364–389, 2023.

[3] 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.

[4] H. Koziolek, V. Ashiwal, S. Bandyopadhyay, and K. Chandrika, "Automated control logic test case generation using large language models," in *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2024, pp. 1–8.

[5] M. Hollender, *Collaborative process automation systems*. ISA, 2010.

[6] M. Tiegelkamp and K.-H. John, *IEC 61131-3: Programming industrial automation systems*. Springer, 2010, vol. 166.

[7] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "GPT-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[8] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 919–931.

[9] OpenAI, "OpenAI Codex." [Online]. Available: https://openai.com/index/openai-codex/

[10] S. McQuistin, M. Karan, P. Khare, C. Perkins, G. Tyson, M. Purver, P. Healey, W. Iqbal, J. Qadir, and I. Castro, "Characterising the IETF through the lens of RFC deployment," in *Proceedings of the 21st ACM Internet Measurement Conference*, 2021, pp. 137–149.

[11] S. K. R. Kakarla, R. Beckett, T. Millstein, and G. Varghese, "Ferret: Automatically finding RFC compliance bugs in DNS nameservers."

[12] P. Sharma and V. Yegneswaran, "PROSPER: Extracting protocol specifications using large language models," in *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, 2023, pp. 41–47.

[13] J. Lee, S. Kang, and D. Lee, "Survey on software testing practices," *IET software*, vol. 6, no. 3, pp. 275–282, 2012.

[14] J. Yen, R. Govindan, and B. Raghavan, "Tools for disambiguating RFCs," in *Proceedings of the 2021 Applied Networking Research Workshop*, 2021, pp. 85–91.

[15] D. S. Taley and B. Pathak, "Comprehensive study of software testing techniques and strategies: a review," *Int. J. Eng. Res*, vol. 9, no. 08, pp. 817–822, 2020.

[16] K. Meinke, "Code coverage and test automation: State of the art," *arXiv preprint arXiv:2108.11723*, 2021.

[17] M. N. Zafar, W. Afzal, E. Enoiu, A. Stratis, A. Arrieta, and G. Sagardui, "Model-based testing in practice: An industrial case study using graphwalker," in *Proceedings of the 14th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference)*, 2021, pp. 1–11.

[18] P. van Spaendonck, “Efficient dynamic model based testing: Using greedy test case selection,” in *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer, 2023, pp. 173–188.

[19] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches,” *Software testing, verification and reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[20] A. Aerts, M. Reniers, and M. R. Mousavi, “Model-based testing of cyber-physical systems,” in *Cyber-Physical Systems*. Elsevier, 2017, pp. 287–304.

[21] J. Vain, E. Halling, G. Kanter, A. Anier, and D. Pal, “Model-based testing of real-time distributed systems,” in *International Baltic Conference on Databases and Information Systems*. Springer, 2016, pp. 272–286.

[22] E. Kurian, D. Briola, P. Braione, and G. Denaro, “Automatically generating test cases for safety-critical software via symbolic execution,” *Journal of Systems and Software*, vol. 199, p. 111629, 2023.

[23] J. Jaffar, R. Maghareh, S. Godboley, and X.-L. Ha, “Tracerx: Dynamic symbolic execution with interpolation (competition contribution),” *Fundamental Approaches to Software Engineering*, vol. 12076, p. 530, 2020.

[24] F. Ahsan and F. Anwer, “A critical review on search-based security testing of programs,” *Computational Intelligence: Select Proceedings of InCITe 2022*, pp. 207–225, 2023.

[25] F. Pecorelli, G. Grano, F. Palomba, H. C. Gall, and A. De Lucia, “Toward granular search-based automatic unit test case generation,” *Empirical Software Engineering*, vol. 29, no. 4, pp. 1–49, 2024.

[26] Z. Yu, Z. Liu, X. Cong, X. Li, and L. Yin, “Fuzzing: Progress, challenges, and perspectives,” *CMC-COMPUTERS MATERIALS & CONTINUA*, vol. 78, no. 1, pp. 1–29, 2024.

[27] A. Zhang, Y. Zhang, Y. Xu, C. Wang, and S. Li, “Machine learning-based fuzz testing techniques: A survey,” *IEEE Access*, 2023.

[28] L. Huang, P. Zhao, H. Chen, and L. Ma, “Large language models based fuzzing techniques: A survey,” *arXiv preprint arXiv:2402.00350*, 2024.

[29] Z. Zhang, H. Zhang, J. Zhao, and Y. Yin, “A survey on the development of network protocol fuzzing techniques,” *Electronics*, vol. 12, no. 13, p. 2904, 2023.

[30] Netzob, “Netzob: Protocol reverse engineering, modeling and fuzzing.” [Online]. Available: https://github.com/netzob/netzob

[31] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, “Large language model guided protocol fuzzing,” in *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, vol. 2024, 2024.

[32] S. Bardin, B. Botella, F. Dadeau, F. Charreteur, A. Gotlieb, B. Marre, C. Michel, M. Rueher, and N. Williams, “Constraint-based software testing,” *Journée du GDR-GPL*, vol. 9, p. 1, 2009.

[33] A. Gotlieb, “Euclide: A constraint-based testing framework for critical c programs,” in *2009 International Conference on Software Testing Verification and Validation*. IEEE, 2009, pp. 151–160.

[34] M. Fleischmann, D. Kaindlstorfer, A. Isychev, V. Wüstholz, and M. Christakis, “Constraint-based test oracles for program analyzers,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 344–355.

[35] M. L. Pacheco, M. von Hippel, B. Weintraub, D. Goldwasser, and C. Nita-Rotaru, “Automated attack synthesis by extracting finite state machines from protocol specification documents,” in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 51–68.

[36] 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*, 2021, pp. 272–286.

[37] I. Schieferdecker and A. Hoffmann, “Model-based testing,” *IEEE software*, vol. 29, no. 1, pp. 14–18, 2012.

[38] Y. Li, B. C. Pierce, and S. Zdancewic, “Model-based testing of networked applications,” in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 529–539.

[39] W. H. Allen, C. Dou, and G. A. Marin, “A model-based approach to the security testing of network protocol implementations,” in *Proceedings. 2006 31st IEEE Conference on Local Computer Networks*, 2006, pp. 1008–1015.

[40] A. Sosnovich, O. Grumberg, and G. Nakibly, “Formal black-box analysis of routing protocol implementations,” 2017.

[41] OpenAI, “GPT-3.5 Turbo.” [Online]. Available: https://platform.openai.com/docs/models/gpt-3.5-turbo

[42] K. L. McMillan and L. D. Zuck, “Compositional testing of internet protocols,” in *2019 IEEE Cybersecurity Development (SecDev)*. IEEE, 2019, pp. 161–174.

[43] K. L. McMillan and O. Padon, “Ivy: A multi-modal verification tool for distributed algorithms,” in *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II 32*. Springer, 2020, pp. 190–202.

[44] C. Crochet, T. Rousseaux, M. Piraux, J.-F. Sambon, and A. Legay, “Verifying QUIC implementations using Ivy,” in *Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC*, 2021, pp. 35–41.

[45] M. Praveen, R. Ramesh, and I. Doidge, “Formally verifying the safety of pipelined moonshot consensus protocol,” *arXiv preprint arXiv:2403.16637*, 2024.

[46] T. Rousseaux, C. Crochet, J. Aoga, and A. Legay, “Network simulator-centric compositional testing,” in *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer, 2024, pp. 177–196.

[47] J. Song, C. Cadar, and P. Pietzuch, “Symbexnet: Testing network protocol implementations with symbolic execution and rule-based specifications,” *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 695–709, 2014.

[48] S. Fakhoury, M. Kuppe, S. K. Lahiri, T. Ramananandro, and N. Swamy, “3dgen: Ai-assisted generation of provably correct binary format parsers,” *arXiv preprint arXiv:2404.10362*, 2024.

[49] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, “Prospex: Protocol specification extraction,” in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 110–125.

[50] N. Soni, H. A. Schwartz, J. Sedoc, and N. Balasubramanian, “Large human language models: A need and the challenges,” *arXiv preprint arXiv:2312.07751*, 2023.

[51] I. O. Gallegos, R. A. Rossi, J. Barrow, M. M. Tanjim, S. Kim, F. Dernoncourt, T. Yu, R. Zhang, and N. K. Ahmed, “Bias and fairness in large language models: A survey,” *Computational Linguistics*, vol. 50, no. 3, pp. 1097–1179, 2024.

[52] S. Raza, A. Raval, and V. Chatrath, “Mbias: Mitigating bias in large language models while retaining context,” *arXiv preprint arXiv:2405.11290*, 2024.

[53] A. Hajikhani and C. Cole, “A critical review of large language models: Sensitivity, bias, and the path toward specialized AI,” *Quantitative Science Studies*, pp. 1–22, 2024.

[54] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin *et al.*, “A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions,” *ACM Transactions on Information Systems*, 2023.

[55] V. Hnatushenko and I. Pavlenko, “The use of generative artificial intelligence in software testing,” vol. 2, no. 151, pp. 113–123, 2024.

[56] K. Alshmrany and L. Cordeiro, “Finding security vulnerabilities in network protocol implementations,” *arXiv preprint arXiv:2001.09592*, 2020.

[57] M. Kosek, L. Blöcher, J. Rüth, T. Zimmermann, and O. Hohlfeld, “Must, should, don’t care: TCP conformance in the wild,” in *Passive and Active Measurement: 21st International Conference, PAM 2020, Eugene, Oregon, USA, March 30–31, 2020, Proceedings 21*. Springer, 2020, pp. 122–138.

[58] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, “Software testing with large language models: Survey, landscape, and vision,” *IEEE Transactions on Software Engineering*, 2024.

[59] D. Sammon, S. McCarthy, B. V. Thummadi, A. Wibisono, and B. Fitzgerald, “Exploring the potential of large language models (LLMs) for grounded theorizing: A human-in-the-loop configuration,” 2024.

[60] E. Mosqueira-Rey, E. Hernández-Pereira, D. Alonso-Ríos, J. Bobes-Bascarán, and Á. Fernández-Leal, “Human-in-the-loop machine learning: a state of the art,” *Artificial Intelligence Review*, vol. 56, no. 4, pp. 3005–3054, 2023.

[61] X. Wu, L. Xiao, Y. Sun, J. Zhang, T. Ma, and L. He, “A survey of human-in-the-loop for machine learning,” *Future Generation Computer Systems*, vol. 135, pp. 364–381, 2022.