

MultiFuzz: A Dense Retrieval-based Multi-Agent System for Network Protocol Fuzzing

Youssef Maklad*, Fares Wael*, Ali Hamdi*, Wael Elersy*, Khaled Shaban†

**Dept. of Computer Science, MSA University, Giza, Egypt*

{youssef.mohamed88, fares.wael, ahamdi, wfarouk}@msa.edu.eg

†Dept. of Computer Science, Qatar University, Doha, Qatar

khaled.shaban@qu.edu.qa

Abstract—Traditional protocol fuzzing techniques, such as those employed by AFL-based systems, often lack effectiveness due to a limited semantic understanding of complex protocol grammars and rigid seed mutation strategies. Recent works, such as ChatAFL, have integrated Large Language Models (LLMs) to guide protocol fuzzing and address these limitations, pushing protocol fuzzers to wider exploration of the protocol state space. But ChatAFL still faces issues like unreliable output, LLM hallucinations, and assumptions of LLM knowledge about protocol specifications. This paper introduces MultiFuzz, a novel dense retrieval-based multi-agent system designed to overcome these limitations by integrating semantic-aware context retrieval, specialized agents, and structured tool-assisted reasoning. MultiFuzz utilizes agentic chunks of protocol documentation (RFC Documents) to build embeddings in a vector database for a retrieval-augmented generation (RAG) pipeline, enabling agents to generate more reliable and structured outputs, enhancing the fuzzer in mutating protocol messages with enhanced state coverage and adherence to syntactic constraints. The framework decomposes the fuzzing process into modular groups of agents that collaborate through chain-of-thought reasoning to dynamically adapt fuzzing strategies based on the retrieved contextual knowledge. Experimental evaluations on the Real-Time Streaming Protocol (RTSP) demonstrate that MultiFuzz significantly improves branch coverage and explores deeper protocol states and transitions over state-of-the-art (SOTA) fuzzers such as NSFuzz, AFLNet, and ChatAFL. By combining dense retrieval, agentic coordination, and language model reasoning, MultiFuzz establishes a new paradigm in autonomous protocol fuzzing, offering a scalable and extensible foundation for future research in intelligent agentic-based fuzzing systems.

Index Terms—Protocol Fuzzing, Network Security, Finite-State Machine, Reverse Engineering, Large Language Models, Multi-Agent Systems, Dense Retrieval, Retrieval-Augmented Generation, Chain-of-Thoughts

I. INTRODUCTION

Network protocols form the backbone of modern communication systems, yet remain vulnerable to many flaws that can compromise entire infrastructures’ security. Protocol fuzzing has long been an effective technique for uncovering these vulnerabilities through automated test generation, and it has long been recognized as an effective technique for uncovering these software vulnerabilities [1]. As network services grow in complexity and scale, the importance of discovering imple-

mentation flaws, especially in stateful protocols with finite-state machines (FSMs), increases. Network protocol fuzzing attempts to systematically test protocol implementations by generating malformed, unexpected, or semi-valid protocol messages to identify anomalous behavior. However, traditional fuzzing methods often struggle with unique protocol challenges, such as handling complex grammar formats, managing deep-protocol state transitions, and maintaining valid session semantics across multi-packet interactions [2].

Recent research highlight multiple directions in the advancement of protocol fuzzing, including state-aware input generation and automated reverse engineering of undocumented protocols [3], [4]. Despite these developments, achieving high coverage and deeper state exploration remains difficult, particularly for closed-source or proprietary protocols. This has motivated the integration of more intelligent components into the fuzzing loop.

The rise of LLMs has opened new avenues for automating traditionally manual tasks in software engineering. LLMs have demonstrated strong capabilities in reasoning, code understanding, and program synthesis [5], [6]. Their potential in security applications, including fuzzing, has begun to evolve. Studies show that LLMs can infer message grammars, generate valid input sequences, and even simulate stateful behavior without access to source code [7], [8]. Recent works such as ChatAFL introduced LLM-guided protocol fuzzing, resulting in improved protocol state coverage [9]. These developments highlight LLMs as promising assistants for fuzzing complex, stateful, and security-critical systems.

In this work, we present *MultiFuzz*, a multi-agent system built on top of ChatAFL, designed to enhance network protocol fuzzing by unleashing the capabilities of LLMs and dense retrieval [10]. Inspired by recent advancements in retrieval-augmented generation and ReAct-based chain-of-thought reasoning [11], [12], *MultiFuzz* is structured around collaborative agents, each responsible for a specific phase of the ChatAFL fuzzing pipeline. Unlike traditional fuzzers or single LLM approaches, *MultiFuzz* orchestrates tool-augmented agents to maintain semantic context support and protocol-specific inference.

The main contributions of this work are as follows:

- We propose *MultiFuzz*, a multi-agent system for protocol fuzzing, integrated with the ChatAFL framework, where each group of agents is dedicated to a specific subtask and enhanced with tool integration, vector database context awareness, and CVE-driven vulnerability knowledge.
- We introduce an agentic-chunking method and embedding strategy for protocol RFC documents, enabling semantic indexing of protocol knowledge for agent use.
- We integrate dense retrieval into the agent reasoning process to maintain a protocol-aware context and guide more effective fuzzing actions.
- We evaluate *MultiFuzz* on stateful protocol targets, and the results demonstrate improvements in branch coverage, number of states explored, and state transitions compared to SOTA fuzzers such as NSFuzz, AFLNet, and ChatAFL.

Through *MultiFuzz*, we aim to bridge the gap between protocol-aware fuzzing needs and the generative coordination capabilities of modern agentic-AI architecture, allowing more intelligent and effective protocol fuzzing.

The paper is structured as follows: Section II introduces background on protocol fuzzing, LLMs, and multi-agent systems. Section III reviews the related work. Section IV explains our methodology. Section V explores the research questions, experiments’ setup, and evaluation metrics. Section VI highlights the experimental results. Finally, section VII concludes the paper and suggests potential directions for future works.

II. BACKGROUND

The following subsections review key concepts and recent developments in network protocol fuzzing, LLMs, and multi-agent systems, providing background information for their integration in modern frameworks.

A. Network Protocol Fuzzing

Protocol fuzzing is a specialized security-testing technique that targets the finite-state behavior of communication protocols by injecting crafted or mutated packets to uncover flaws. It relies on intelligently generating seed inputs that exercise different protocol states, since exploring deep protocol behaviors often exposes hidden vulnerabilities. Network protocol fuzzing focuses on testing stateful network services by feeding packet sequences through the protocol’s FSM in the server under test (SUT). The goal is to traverse unusual protocol paths and trigger implementation bugs or security flaws. Fuzzers can be broadly classified by their test case generation strategy. Mutation-based approaches modify existing valid packets using bit-flipping, arithmetic, block-level, or dictionary-based transformations [13]. In contrast, generation-based approaches synthesize packets from protocol specifications or templates. While mutation methods may struggle with diverse data types and protocol constraints, generation-based methods often face difficulties in acquiring or modeling accurate protocol specifications. Additionally, based on the level of knowledge about the target system, fuzzing can be categorized into blackbox, whitebox, and graybox approaches. Blackbox fuzzers operate

without internal knowledge of the protocol implementation, relying solely on input/output observation. White-box fuzzers analyze the source code to guide test case generation, while gray-box fuzzers use lightweight instrumentation as code coverage feedback to guide mutations more effectively. In practice, graybox fuzzing offers a balanced trade-off and is widely used due to limited access to source code in real-world protocol implementations [14].

B. Large Language Models

LLMs have recently demonstrated powerful capabilities in generating and reasoning over complex inputs, opening new opportunities for automation in domains like software testing and cybersecurity [7], [8]. They are deep transformer-based neural networks [15], trained on massive text corpora, enabling them to generate coherent language and perform complex reasoning [12]. Their rich knowledge and generative power have been harnessed in many domains. In cybersecurity, LLMs have shown remarkable utility. For example, ChatPhishDetector uses an LLM to detect phishing websites [16]. Maklad, Y. et al demonstrated how LLMs, enhanced by RAG and chain-of-thought reasoning, can be used to evaluate seed enrichment tasks and network packet generation [17]. SeedMind explored the use of LLMs for building fuzzing seed generators [18]. Codamosa highlights the use of LLMs in overcoming coverage plateaus in test generation [19]. LLMs have also been integrated into automation workflows such as Robotic Process Automation (RPA) and OCR [20]–[22]. These results highlight that LLMs, when incorporated intelligently, can enhance the accuracy and efficiency of automation systems.

C. Multi-Agent Systems

Multi-agent systems consist of multiple autonomous entities (agents) that interact and collaborate to solve tasks. By harnessing the diverse capabilities and roles of individual agents, multi-agent systems can tackle complex problems more effectively than a single agent could [23]. For example, agents might divide a workflow so that some gather information, others perform reasoning, and yet others execute actions. However, orchestrating agents also introduces challenges like optimal task allocation, sharing complex context information, and memory management, which become critical in LLM-based multi-agent architectures. In security applications, multi-agent LLM architectures have begun to emerge. A recent effort has introduced PentestAgent [24], a framework in which multiple agents collaborate to automate penetration testing and vulnerability analysis. This demonstrates how multi-agent systems can decompose a complex task (like pentesting) into subtasks handled by specialized LLM agents, improving overall efficiency.

III. RELATED WORK

A. Protocol Fuzzing

Fuzzing has proven to be one of the most effective techniques for vulnerability discovery. Protocol fuzzing, in particular, poses unique challenges due to its reliance on structured

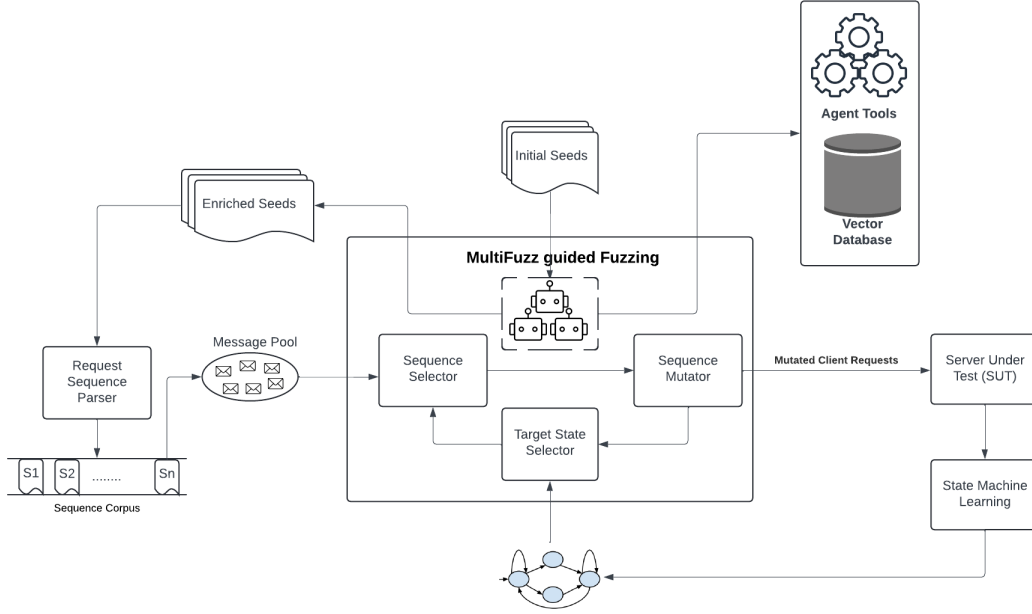


Fig. 1. High-Level System Architecture of the *MultiFuzz* Framework, based on AFLNet and ChatAFL

formats and stateful interactions. Existing techniques outlined earlier in section II, which are blackbox, whitebox, and graybox fuzzing approaches, offer different trade-offs between scalability, precision, and required prior knowledge of the target protocol.

1) **Blackbox Fuzzing:** Blackbox fuzzers operate without any internal knowledge of the target and typically rely on traffic observation or mutation of recorded protocol messages. Tools like SPIKE and Peach exemplify early blackbox efforts, relying on manual specification of protocol structures [25]. PULSAR and BBuzz extract message formats from captured network traffic using protocol reverse engineering techniques [26], [27]. These methods are simple to deploy but struggle with exploring deeper protocol states, often failing to maintain session validity across multi-message interactions.

2) **Whitebox Fuzzing:** Whitebox fuzzers leverage full access to source code or binaries to systematically explore execution paths. Symbolic execution and taint analysis are commonly used to analyze input-dependent behaviors. Polar combines static analysis with dynamic taint tracking to extract input-related conditions from protocol code [14]. While these techniques offer fine-grained insight and deeper coverage, their scalability is limited by path explosion and instrumentation overhead. Whitebox fuzzers are less commonly used in network protocol contexts due to the complexity of protocol stacks and message interleaving.

3) **Graybox Fuzzing:** Graybox fuzzers balance insight and scalability by utilizing lightweight instrumentation to guide input mutations. AFL and its extensibles, like AFL++ and AFLNet, employ coverage feedback to direct test-case generation [28]–[30]. AFLNet extends AFL to stateful network protocols by using response codes to infer protocol state

transitions. NSFuzz further improves graybox fuzzing by extracting program variables associated with state changes to better synchronize test inputs with protocol logic [31]. These tools have proven effective on real-world network services and are the foundation for many modern protocol fuzzers.

B. Large Language Model-assisted Fuzzing

Recent research has explored the integration of LLMs into fuzzing pipelines. These models offer the ability to generate syntactically correct and semantically meaningful inputs by leveraging knowledge learned during pre-training. ChatFuzz uses OpenAI’s ChatGPT to mutate existing seed inputs, resulting in improved edge coverage compared to AFL++ [32]. ChatAFL constructs message grammars and predicts the next protocol message using GPT models, achieving significant gains in state and code coverage over AFLNet and NSFuzz [9]. MSFuzz extracts abstract syntax trees from the protocol source code via LLMs to guide syntax-aware mutation [33]. TitanFuzz and FuzzGPT show that LLMs can function as zero-shot fuzzers for deep learning libraries by generating edge case inputs and exploiting rare model behaviors without instrumentation or prior seeds [34], [35]. These works demonstrate that LLMs can serve as powerful assistants in automating grammar extraction, seed generation, and mutation strategies for protocol fuzzing.

IV. METHODOLOGY

This section details the design of the *MultiFuzz* framework and its agent-based workflows. The *MultiFuzz* framework APIs are integrated in the ChatAFL framework, on top of the AFLNet architecture. The whole system architecture can be shown in Figure 1. *MultiFuzz* is structured around three

specialized crews of agents: the Grammar Extraction Crew, the Seed Enrichment Crew, and the Coverage Plateau Crew. Each crew operates over a shared semantic context retrieved at inference by a common dense retrieval agent, which retrieves the agentic chunked embeddings in the vector store. The workflow begins with preprocessing protocol RFCs, then transforms the content into propositional transformation, followed by agentic chunking, and finally, collaborative agent reasoning at inference.

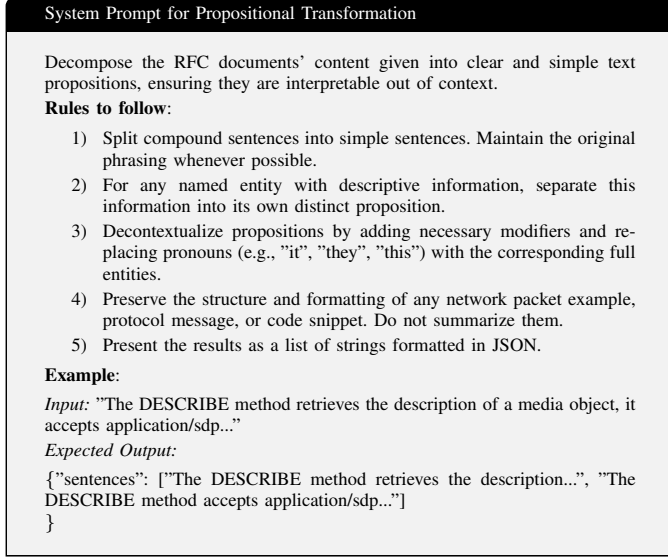


Fig. 2. System prompt used for propositional transformation of filtered RFC documents

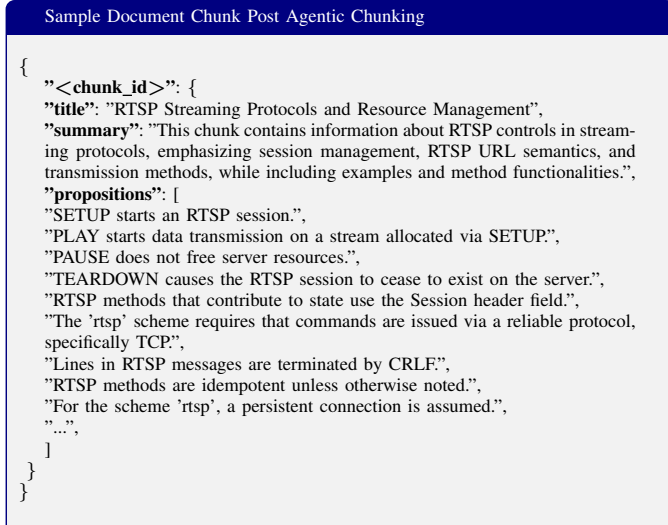


Fig. 3. Sample document chunk after the agentic chunking phase of text propositions.

A. RFC Documents Preprocessing

We first process the RFC documents, where each RFC is manually segmented into paragraphs and then passes through a series of filters to extract technical sections, including stateful interactions, command formats, and response rules. We define an RFC as a sequence of paragraphs $R = \{r_1, r_2, \dots, r_n\}$.

A semantic classifier f_{filter} maps each paragraph to a boolean label:

$$f_{\text{filter}}(r_i) = \begin{cases} 1 & \text{if } r_i \text{ is protocol-relevant} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Only filtered paragraphs $R' = \{r_i \in R \mid f_{\text{filter}}(r_i) = 1\}$ are retained for downstream chunking and proposition extraction.

B. Propositional Transformation

Once the RFC content has been semantically filtered and structured into coherent sections using specific delimiters (###, ---, @@@), the next step in the pipeline is to transform these technical paragraphs into interpretable, context-independent atomic propositions. To perform this transformation, we constructed an LLM-powered pipeline. Each filtered section is first processed using a carefully designed system prompt, shown in Figure 2. The prompt is executed using the *gpt-4o-mini* model with structured output enforced by a JSON schema. Each chunk of the RFC document is passed through this pipeline, producing a list of minimal, decontextualized statements that accurately capture the semantics of the protocol specification. Formally, for a given input chunk $C_i \in \mathcal{C}$ where \mathcal{C} is the set of smart RFC chunks, the transformation function T produces:

$$T(C_i) = \{p_1, p_2, \dots, p_k\}, \quad \text{where each } p_j \in \mathbb{P}$$

Here, \mathbb{P} denotes the proposition space containing linguistically simple, context-independent units of meaning. As a result, each paragraph C_i is mapped to a finite set of logically coherent propositions, and the global proposition set \mathbb{P} becomes the knowledge substrate for subsequent dense retrieval and crew-based inference modules. In our experiments on RFC-2326 (RTSP), this step yielded 445 unique and precise propositions.

C. Agentic Chunking Module

Following the propositional transformation step, we employ an intelligent chunking mechanism termed the *Agentic Chunker* to group semantically similar propositions into cohesive and operationally meaningful units. This process creates the foundation for precise retrieval and role-specific agent inference in later stages of the *MultiFuzz* framework.

Formally, given a set of propositions $\mathbb{P} = \{p_1, p_2, \dots, p_n\}$ derived from the RFC document, the goal is to partition \mathbb{P} into a set of non-overlapping semantic chunks $\mathcal{Z} = \{z_1, z_2, \dots, z_m\}$, where each $z_j \subseteq \mathbb{P}$ and $\bigcup_{j=1}^m z_j = \mathbb{P}$. The chunking objective can be viewed as an unsupervised grouping problem constrained by topic cohesion, guided by an LLM.

The chunking process is agentic in nature: each incoming proposition p_i is evaluated using a prompt-driven LLM flow by *gpt-4o-mini*. This LLM first examines the current set of chunk summaries and determines whether p_i semantically aligns with any existing chunk z_j . If alignment is detected, p_i is appended to that chunk. Otherwise, a new chunk is instantiated.

Each chunk z_j maintains three evolving elements:

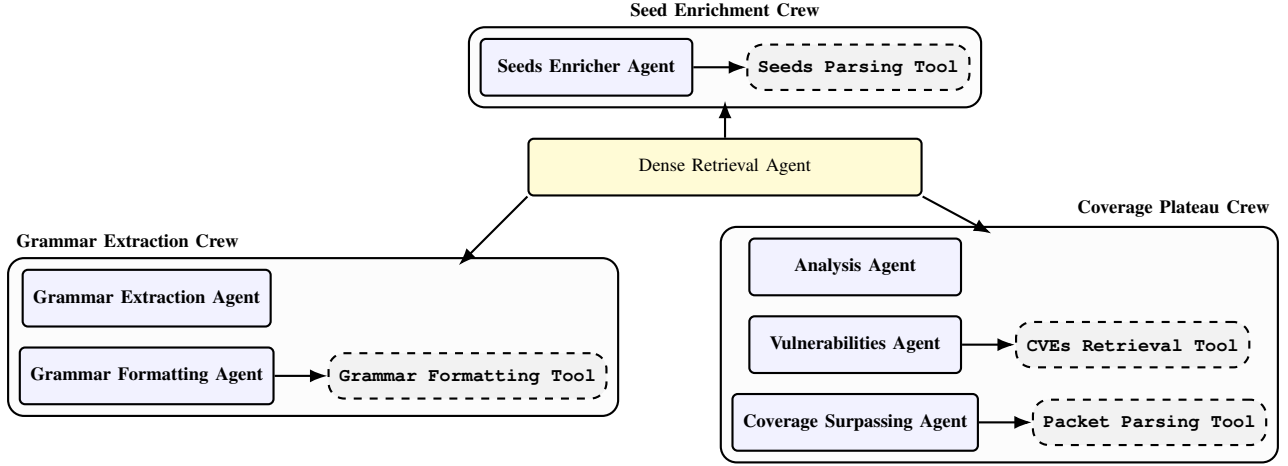


Fig. 4. Summary of the *MultiFuzz*'s crews, showing each crew's internal agents and integrated tools. All three crews share a Dense Retrieval Agent for semantic context fetching.

- A list of constituent propositions $\{p_k\}_{k=1}^K$,
- A concise chunk summary s_j generated by the LLM.
- A descriptive, technically precise chunk title t_j .

The internal logic can be modeled as a two-stage LLM pipeline:

- 1) **Chunk Selection:** Given current chunk outlines and a new proposition p_i , select the most semantically compatible chunk z_j such that:

$$z_j = \arg \max_{z \in Z} \text{sim}(p_i, s_z)$$

If $\max \text{sim} < \theta$, where θ is a system-defined compatibility threshold, a new chunk is created.

- 2) **Metadata Refinement:** After assignment, the system regenerates the chunk's summary s_j and title t_j using structured prompt templates conditioned on the current list of propositions.

The final output is a collection of richly annotated document objects, each encapsulating a semantic group of RFC-derived propositions, along with human-readable summaries and titles. These document objects were then embedded using OpenAI's *text-embedding-ada-002* model and indexed into a *Chroma*-based dense vector database. A sample document object can be shown in Figure 3.

D. Dense Retrieval Agent

The first common agent in all crews is the dense retrieval agent. This agent is responsible for querying a *Chroma*-based dense vector store populated with semantically grouped RTSP agentic chunks. It utilizes a *Custom RAG Tool*, to perform approximate nearest neighbor search against the indexed chunks. The output of this agent is a context-rich corpus of relevant documents passed to assist all agents with their tasks.

E. Grammar Extraction Crew of Agents

The *Grammar Extraction Crew* is the first crew of agents designed to extract structured RTSP client request templates

for ChatAFL. It operates through three agents: a dense retrieval agent, a grammar extraction agent, and a grammar formatting agent.

Grammar Extraction Agent: This agent uses the retrieved context to produce JSON-formatted RTSP request templates, where each method (e.g. PLAY, DESCRIBE) maps to a list of headers containing `<<VALUE>>` placeholders and `\r\n` terminators.

Grammar Formatting Agent: It refines the raw JSON output into a clean, numbered textual format using the Grammar Extraction Formatting Tool, making it easier to parse in the ChatAFL grammar parsing module.

F. Seed Enrichment Crew of Agents

The *Seed Enrichment Crew* is a two-agent crew designed to enhance a given sequence of RTSP client requests by inserting new protocol-compliant packets at semantically correct positions. This enrichment supports fuzzers by generating deeper, more state-aware input sequences. The first agent is the dense retrieval agent, and the second is the seeds enricher agent.

Seeds Enricher Agent: This agent interprets the protocol's FSM and uses retrieved context from the dense retrieval agent to insert two desired client requests, typically absent from the original seed into their appropriate positions as adopted in ChatAFL. It ensures server responses are excluded and leverages the *Seeds Parsing Tool* to generate structured outputs of continuous enriched network packets. These enriched seeds are structured to be easily parsed by the ChatAFL parsing module.

G. Coverage Plateau Surpassing Crew of Agents

The *Coverage Plateau Surpassing Crew* is designed to help the fuzzer escape stagnation points during test execution, where no new protocol states or code paths are being explored as observed in ChatAFL. This crew of agents aims to generate packets that can trigger new transitions by analyzing communication history, retrieved context, and optionally exploiting known CVEs.

Analysis Agent: This agent performs deep context analysis of the context retrieved from the dense retrieval agent and the fuzzer’s communication history to construct a detailed generation prompt. Rather than producing packets directly, it crafts precise instructions to guide the next agent in generating a coverage-enhancing input.

Vulnerabilities Agent: To improve the chance of producing impactful packets, this agent enriches the generation prompt with insights from real CVEs, fetched using a CVEs Retrieval Tool which uses the NVD (National Vulnerability Database) API to obtain the Live555 server vulnerabilities [36]. If any vulnerability discovered is relevant to the current communication context, the prompt is refined accordingly; otherwise, it is forwarded unchanged.

Coverage Surpassing Agent: Finally, this agent consumes the refined prompt and generates a valid RTSP client request designed to surpass the coverage plateau. The agent uses a Packet Parsing Tool to structure the final packet and log it along with an explanation of its purpose. A generated sample prompt can be shown in Figure 5.

Sample Prompt to generate Coverage Plateau Packet

```

"prompt": {
  "To surpass the current coverage plateau, Generate a PAUSE request that will transition the server from the Playing state to the Ready state. The PAUSE method should be sent with the appropriate headers, including CSeq: 5, Session: 000022B8, and the method set to PAUSE. This will explore the state transition from Playing to Ready, potentially revealing new server behaviors and increasing coverage."
}

```

Fig. 5. Sample prompt asking the final agent to generate a coverage plateau surpassing packet

H. Implementation

We have developed *MultiFuzz* on top of two agentic-AI frameworks: *LangChain* [37] and *CrewAI* [38]. *LangChain* provides abstractions for building applications on top of LLMs. We use the *LangChain* framework combined with the *Chroma* vector store for embedding and indexing. We utilize its features specifically in the RFC processing stage for RFC document agentic chunking and during the dense retrieval inference in all agent tasks. *CrewAI*, on the other hand, is used to build autonomous multi-agent systems and provides modular assignment of agents to specific and unique roles. It supports integration with multiple LLM API providers and offers native support for tool-assisted workflows, enabling agents to interact with file systems and vector databases. We use *CrewAI* in defining and orchestrating the agents that compose the *MultiFuzz* framework. These structured agent groups, or “crews”, coordinate within the ChatAFL framework using event-driven task scheduling augmented with custom structured tools.

V. EXPERIMENTAL DESIGN AND EVALUATION

We evaluate the proposed *MultiFuzz* framework by measuring its effectiveness in fuzzing stateful protocol implementa-

tions using a multi-agent-based architecture. Our evaluation aims to answer the following research questions:

- **RQ1:** How effective is *MultiFuzz* in improving branch coverage and state exploration compared to SOTA protocol fuzzers?
- **RQ2:** How does the multi-agent collaboration strategy improve over single-LLM approaches such as ChatAFL?

To conduct the evaluation, we test *MultiFuzz* on the RTSP protocol implemented by the Live555 media streaming server. RTSP was selected due to its rich stateful behavior, complexity in session semantics, and widespread use in multimedia transmission. It presents non-trivial state transitions that make it a fitting candidate for state-aware fuzzing. The framework is powered by Llama-based language models obtained via the Groq-Cloud API [39], which are: *llama3.3-70b-versatile*, *deepseek-r1-distill-llama-70b*, *llama3-70b-8192*, *llama-4-scout-17b-16e-instruct*, and *llama-3.1-8b-instant*, chosen for their reasoning abilities, and long-context window capacities. Throughout the experimentation process, we explored different combinations of these models across the various agent groups in the framework. Tasks such as grammar extraction, seed enrichment, and plateau surpassing were assigned to different models iteratively until the most effective model was identified for each specific subtask, optimizing the overall performance of *MultiFuzz*.

A. Experiments Setup

To evaluate the fuzzing effectiveness of *MultiFuzz*, we conducted a 24-hour three fuzzing sessions using our framework alongside the three SOTA baseline fuzzers: NSFuzz, AFLNet, and ChatAFL under the same experimental conditions. All experiments were performed on a local machine running Ubuntu 24.04.02 LTS, equipped with an Intel Core i5-11300H processor and 16 GB of RAM. The fuzzers were evaluated against the RTSP protocol implemented in the Live555 media streaming server. Each fuzzer was independently executed with default settings. We measured the effectiveness of each fuzzer across several key metrics, including unique crashes, state coverage, branch coverage, and total paths explored. This setup allows us to assess the relative performance of *MultiFuzz* in contrast with existing approaches.

B. Evaluation Metrics

To evaluate *MultiFuzz*’s fuzzing performance, we adopted standard coverage-based metrics inspired by existing works such as AFLNet and ChatAFL:

- **Branch Coverage:** Number of unique conditional branches exercised in the code.
- **Number of States:** Number of FSM states reached and explored during fuzzing.
- **Number of State Transitions:** The total count of valid state transitions triggered within the protocol’s FSM, reflecting the depth of state space exploration.

We use ProFuzzBench [40] as our benchmarking platform due to its automated nature in a containerized environment

using Docker and to baseline with the previous SOTA protocol fuzzers. All experiments were repeated multiple times to ensure consistency, and results were averaged over time windows to account for variability in execution. This setup allows us to rigorously measure *MultiFuzz*'s capability to intelligently generate protocol-aware inputs and uncover deep-state vulnerabilities.

C. Experimental Results of Fuzzing on Code Coverage

Table I presents the branch coverage results, demonstrating *MultiFuzz*'s substantial superiority in code coverage metrics. *MultiFuzz* achieves an average branch coverage of 2940 branches, representing dramatic improvements of 1.0% over ChatAFL (2912.67), 2.8% over AFLNet (2860), and 2.3% over NSFuzz (2807). Although these percentage improvements may appear modest, the absolute differences are significant in the context of protocol fuzzing, where each additional branch represents potential discovery of critical vulnerabilities. The consistency of *MultiFuzz*'s performance is particularly noteworthy, with coverage ranging from 2970 to 2940 branches across experiments, demonstrating reliable and predictable performance. In contrast, ChatAFL shows higher variability (2890-2998 branches), while AFLNet exhibits perfect consistency but at significantly lower coverage levels. NSFuzz demonstrates the most variability, with coverage ranging from 2795 to 2826 branches.

VI. EXPERIMENTAL RESULTS AND DISCUSSION

A. Experimental Results of Fuzzing on State Exploration

Table II and Table III demonstrate that compared to NSFuzz, AFLNet, and ChatAFL, *MultiFuzz* achieves superior performance in both state transitions and state exploration across all three experimental runs. In terms of state transitions, *MultiFuzz* achieves an average of 163.33 transitions, representing a significant improvement of 2.3% over ChatAFL (159.67), 94.5% over AFLNet (84.0), and 81.2% over NSFuzz (90.33). The state exploration results further validate *MultiFuzz*'s effectiveness, with an average of 14.67 states explored compared to ChatAFL's 14.33 states, AFLNet's 10.0 states, and NSFuzz's 11.7 states. This represents improvements of 2.4%, 46.7%, and 25.4% respectively.

TABLE I: Branch coverage achieved by *MultiFuzz* and baseline SOTA fuzzers

Experiment	MultiFuzz	ChatAFL	AFLNet	NSFuzz
1	2970	2890 \uparrow 2.8%	2850 \uparrow 4.2%	2800 \uparrow 6.1%
2	2910	2998 \downarrow -2.9%	2870 \uparrow 1.4%	2795 \uparrow 4.1%
3	2940	2850 \uparrow 3.2%	2860 \uparrow 2.8%	2826 \uparrow 4.0%
Average	2940.0	2912.67 \uparrow 0.9%	2860.0 \uparrow 2.8%	2807.0 \uparrow 4.7%

TABLE II: Number of state transitions achieved by *MultiFuzz* and baseline SOTA fuzzers

Experiment	MultiFuzz	ChatAFL	AFLNet	NSFuzz
1	163	159 \uparrow 2.5%	80 \uparrow 103.8%	88 \uparrow 85.2%
2	165	158 \uparrow 4.4%	85 \uparrow 94.1%	91 \uparrow 81.3%
3	162	162 \uparrow 0.0%	87 \uparrow 86.2%	92 \uparrow 76.1%
Average	163.33	159.67 \uparrow 2.3%	84.0 \uparrow 94.4%	90.33 \uparrow 80.8%

TABLE III: Number of states explored by *MultiFuzz* and baseline SOTA fuzzers

Experiment	MultiFuzz	ChatAFL	AFLNet	NSFuzz
1	14	14 \uparrow 0.0%	9 \uparrow 55.6%	12 \uparrow 16.7%
2	15	14 \uparrow 7.1%	11 \uparrow 36.4%	11 \uparrow 36.4%
3	15	15 \uparrow 0.0%	10 \uparrow 50.0%	12 \uparrow 25.0%
Average	14.67	14.33 \uparrow 2.4%	10.0 \uparrow 46.7%	11.7 \uparrow 25.4%

B. Observations

The dense retrieval-based multi-agent system of *MultiFuzz* enables more systematic state space exploration by leveraging the agented-chunked and embedded protocol specifications and coordinated agent interactions. Unlike baseline fuzzers that rely on conventional feedback-driven exploration, *MultiFuzz*'s multi-agent architecture facilitates comprehensive state discovery through intelligent coordination and knowledge sharing among specialized agents. The results indicate that *MultiFuzz*'s dense retrieval mechanism effectively identifies and prioritizes valuable states that serve as critical transition points within the protocol state machine. The multi-agent coordination allows for parallel exploration strategies while maintaining systematic coverage of the state space. This approach significantly outperforms traditional fuzzing methods that rely on random mutation and single LLM approaches.

VII. CONCLUSION AND FUTURE WORK

Protocol fuzzing continues to be a foundational technique for uncovering implementation flaws in communication systems. However, traditional fuzzers often face significant limitations when applied to stateful or proprietary network protocols, particularly due to difficulties in handling complex message grammars, managing multi-step state transitions, and maintaining valid interactions across sessions. We proposed *MultiFuzz*, a dense retrieval-based multi-agent system designed to address these limitations by leveraging an agentic-RAG-based architecture empowered by chain-of-thought reasoning. Our approach builds upon prior advances in LLM-assisted fuzzing but distinguishes itself by introducing multi-agent coordination instead of a single LLM. It proposes agentic-based chunking of protocol documents and context-aware inference about protocol specifications and vulnerabilities. These additions help overcome key challenges in stateful fuzzing, such as low coverage and stagnation during long-running sessions. Evaluation across real-world protocol implementations has shown that *MultiFuzz* surpasses existing tools such as NSFuzz, AFLNet, and ChatAFL in terms of state exploration and branch coverage. These findings bridge the gap between traditional fuzzing methodologies and recent advances in agentic-AI, as they open promising opportunities for more effective and adaptive security testing.

Looking forward, we suggest several paths to extend this work. Firstly, enhancing automation by tightly integrating reverse engineering tools, symbolic analyzers, and traffic parsers can further streamline the entire pipeline. Lastly, fine-tuning agent behaviors using domain-specific interaction data could improve their effectiveness in specialized protocol domains.

VIII. ACKNOWLEDGMENT

Heartfelt gratitude is extended to AiTech AU, *AiTech for Artificial Intelligence and Software Development* (<https://aitech.net.au>), for funding this research and enabling its successful completion.

REFERENCES

- [1] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.
- [2] S. Jiang, Y. Zhang, J. Li, H. Yu, L. Luo, and G. Sun, “A survey of network protocol fuzzing: Model, techniques and directions,” *arXiv preprint arXiv:2402.17394*, 2024.
- [3] J. Narayan, S. K. Shukla, and T. C. Clancy, “A survey of automatic protocol reverse engineering tools,” *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, pp. 1–26, 2015.
- [4] D. Liu, V.-T. Pham, G. Ernst, T. Murray, and B. I. Rubinstein, “State selection algorithms and their impact on the performance of stateful network protocol fuzzing,” in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 720–730.
- [5] 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.
- [6] G. Kim, P. Baldi, and S. McAleer, “Language models can solve computer tasks,” *Advances in Neural Information Processing Systems*, vol. 36, pp. 39 648–39 677, 2023.
- [7] L. Huang, P. Zhao, H. Chen, and L. Ma, “Large language models based fuzzing techniques: A survey,” *arXiv preprint arXiv:2402.00350*, 2024.
- [8] M. A. Ferrag, F. Alwahedi, A. Battah, B. Cherif, A. Mechri, and N. Tihanyi, “Generative ai and large language models for cyber security: All insights you need,” *Available at SSRN 4853709*, 2024.
- [9] 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)*, 2024.
- [10] T. Chen, H. Wang, S. Chen, W. Yu, K. Ma, X. Zhao, H. Zhang, and D. Yu, “Dense x retrieval: What retrieval granularity should we use?” in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024, pp. 15 159–15 177.
- [11] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, “Retrieval-augmented generation for knowledge-intensive nlp tasks,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.
- [12] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafraan, K. Narasimhan, and Y. Cao, “React: Synergizing reasoning and acting in language models,” *arXiv preprint arXiv:2210.03629*, 2022.
- [13] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, “The art, science, and engineering of fuzzing: A survey,” *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.
- [14] 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.
- [15] A. Vaswani, “Attention is all you need,” *Advances in Neural Information Processing Systems*, 2017.
- [16] T. Koide, H. Nakano, and D. Chiba, “Chatphishdetector: Detecting phishing sites using large language models,” *IEEE Access*, 2024.
- [17] Y. Maklad, F. Wael, W. Elersy, and A. Hamdi, “Retrieval augmented generation based llm evaluation for protocol state machine inference with chain-of-thought reasoning,” *arXiv preprint arXiv:2502.15727*, 2025.
- [18] W. Shi, Y. Zhang, X. Xing, and J. Xu, “Harnessing large language models for seed generation in greybox fuzzing,” *arXiv preprint arXiv:2411.18143*, 2024.
- [19] 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.
- [20] O. H. Abdellaif, A. N. Hassan, and A. Hamdi, “Erpa: Efficient rpa model integrating ocr and llms for intelligent document processing,” in *2024 International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)*. IEEE, 2024, pp. 295–300.
- [21] O. H. Abdellaif, A. Nader, and A. Hamdi, “Lmrpa: Large language model-driven efficient robotic process automation for ocr,” *arXiv preprint arXiv:2412.18063*, 2024.
- [22] O. Abdellatif, A. Ayman, and A. Hamdi, “Lmv-rpa: Large model voting-based robotic process automation,” *arXiv preprint arXiv:2412.17965*, 2024.
- [23] S. Han, Q. Zhang, Y. Yao, W. Jin, and Z. Xu, “Llm multi-agent systems: Challenges and open problems,” *arXiv preprint arXiv:2402.03578*, 2024.
- [24] X. Shen, L. Wang, Z. Li, Y. Chen, W. Zhao, D. Sun, J. Wang, and W. Ruan, “Pentestagent: Incorporating llm agents to automated penetration testing,” *arXiv preprint arXiv:2411.05185*, 2024.
- [25] Z. Luo, F. Zuo, Y. Shen, X. Jiao, W. Chang, and Y. Jiang, “Ics protocol fuzzing: Coverage guided packet crack and generation,” in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.
- [26] B. Blumbers and R. Vaarandi, “Bbuzz: A bit-aware fuzzing framework for network protocol systematic reverse engineering and analysis,” in *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*. IEEE, 2017, pp. 707–712.
- [27] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, “Pulsar: Stateful black-box fuzzing of proprietary network protocols,” in *Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Proceedings 11*. Springer, 2015, pp. 330–347.
- [28] M. Zalewski, “American fuzzy lop,” 2014, <http://lcamtuf.coredump.cx/afl/>.
- [29] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, “{AFL++}: Combining incremental steps of fuzzing research,” in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.
- [30] V.-T. Pham, M. Böhme, and A. Roychoudhury, “Aflnet: a greybox fuzzer for network protocols,” in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 460–465.
- [31] S. Qin, F. Hu, Z. Ma, B. Zhao, T. Yin, and C. Zhang, “Nsfuzz: Towards efficient and state-aware network service fuzzing,” *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–26, 2023.
- [32] J. Hu, Q. Zhang, and H. Yin, “Augmenting greybox fuzzing with generative ai,” *arXiv preprint arXiv:2306.06782*, 2023.
- [33] M. Cheng, K. Zhu, Y. Chen, G. Yang, Y. Lu, and C. Lu, “Msfuzz: Augmenting protocol fuzzing with message syntax comprehension via large language models,” *Electronics* (2079-9292), vol. 13, no. 13, 2024.
- [34] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, “Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models,” in *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, 2023, pp. 423–435.
- [35] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, “Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt,” *arXiv preprint arXiv:2304.02014*, 2023.
- [36] “NVD - Home,” <https://nvd.nist.gov/>, [Accessed 19-06-2025].
- [37] “LangChain,” <https://www.langchain.com/>, [Accessed 27-05-2025].
- [38] “CrewAI,” <https://www.crewai.com/>, [Accessed 27-05-2025].
- [39] “Groq is Fast AI Inference,” <https://groq.com/>, [Accessed 19-06-2025].
- [40] R. Natella and V.-T. Pham, “Profuzzbench: A benchmark for stateful protocol fuzzing,” in *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, 2021, pp. 662–665.