Inferring State Machine from the Protocol Implementation via Large Language Model

ANONYMOUS AUTHOR(S)

State machines play a pivotal role in augmenting the efficacy of protocol analyzing to unveil more vulnerabilities. However, inferring state machines from network protocol implementations presents significant challenges, mainly because of the complicated code syntax and semantics. Traditional methods based on dynamic analysis often overlook crucial state transitions due to limited coverage, while static analysis suffers from path explosion facing to protocol implementations. To address these limitations, we propose an innovative state machine inference approach powered by Large Language Models (LLMs) named ProtocolGPT. Utilizing retrieval augmented generation technology, this method augments pre-trained model with specific knowledge drawn from protocol implementations. Through targeted prompt engineering, we systematically identify and infer the underlying state machines. Our evaluation across six protocol implementations demonstrates the method's high efficacy, achieving precision exceeding 90% and successfully delineating differences on state machines among various implementations of the same protocol. Integrating our approach with protocol fuzzing significantly improves AFLNet's code coverage by 22.23% over baselines and detects two zero-day vulnerabilities. Our proposed method represents a major advancement in accurate state machine inference and highlights the substantial potential of LLMs in enhancing network protocol security analysis.

CCS Concepts: • **Networks → Protocol testing and verification**; • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Large Language Models, Protocol Reverse Engineering, State Machine, Software Security

ACM Reference Format:
Anonymous Author(s). 2024. Inferring State Machine from the Protocol Implementation via Large Language Model. 1, 1 (November 2024), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Finite State Machines (FSMs) are essential components of protocols, comprising various states and the transitions between these states based on specific conditions or inputs, which guide the implementation of protocols [37]. FSM inference serves as a fundamental cornerstone in a wide range of applications, including vulnerability detection, software engineering, and network protocol analysis [49] while improper protocol implementations may lead to serious security issues [18] and most of the vulnerabilities in network protocols are intricately linked to these states [45]. Therefore, accurately inferring protocol state machines is critical for understanding protocol functionalities [17], protocol verification [3], and unveiling vulnerabilities [2].

Most protocol implementations adhere to the specifications outlined in the network protocol RFCs, which define communication functions, input formats, and state machines [24]. Recently, some researchers have employed natural language processing (NLP) techniques to infer state machines from these specifications [43][24]. Due to the inherent subjectivity in the definitions within RFCs, different developers may interpret them in varying ways. As a result, protocol

ACM XXXX-XXXX/2024/11-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

implementations often include custom elements, leading to discrepancies between the FSMs in protocol implementations and those defined in the RFCs. Even different implementations of the same protocol can exhibit variations considerably in their FSMs, which are the most often hotspots for vulnerabilities. Therefore, the state machines extracted from specific protocol implementations instead of RFCs are more precise and important for protocol security analysis, and can unleash more potential for protocol security analysis.

Research on inferring state machines from protocol implementations can be categorized into two principal types: static analysis [48] and dynamic analysis [9] [27] [58] [28] [19] [22]. Static analysis methods, such as control flow analysis, and data flow analysis, infer state machines by deeply understanding the code structure, eliminating the need for program execution. While static analysis provides deep insights without running the code, it struggles with complicated code structures and uncertain dynamic behavior contexts, often resulting in the path explosion problem. Conversely, dynamic analysis extracts state machines by tracing program behavior and monitoring runtime data during execution. Although dynamic analysis is beneficial for capturing real-time program behavior, it may overlook critical state transitions due to limited coverage, and its effectiveness significantly depends on the quality and representativeness of test cases. These limitations underscore the need for a scalable and comprehensive method to infer state machines from protocol implementations.

Recent studies have demonstrated that LLM possess extensive potential for applications in program analysis. By generating both natural and programming languages, they can assist developers with tasks such as code writing, code repair, vulnerability detection, code interpretation, and documentation generation [57] [34] [55] [16] [10]. Moreover, LLMs have the capability to guide protocol fuzzing [36]. However, these usages has been limited to generating protocol inputs, without the ability to infer the protocol finite state machine. And we have found that LLMs can infer protocol states and state transitions within program code to a certain extent in several cases. Based on the observations above, we posit that LLMs possess the potential to deduce state machines from protocol implementations. In this paper, we primarily investigate FSM in protocol implementations, utilize LLMs to identify protocol states, message types, and transition relationships from the source code. Limited by context size limitations, sufficient information will not be provided to LLMs resulting the inefficient for LLMs to analyze complicated codebases.

Moreover, as the volume and complexity of the code increase, the accuracy of LLMs correspondingly decreases. To address this dilemma, we propose a method for inferring FSMs from protocol implementations using LLM augmented with source code. First, we filter out FSM-related code from large and complicated protocol implementations to avoid unnecessary analysis. Subsequently, the FSM-related code is processed through an embedding model and converted into a vector store which is utilized as augmenting the pre-trained LLM. Finally, we guide the augmented LLM to infer protocol states, message types, and state transitions using chain-of-thought techniques [53].

Our proposed approach aims to overcome several prevailing obstacles in inferring protocol FSMs, offering distinct advantages over traditional techniques such as dynamic program analysis, which typically depend on network traffic samples, and static analysis, which is generally limited by the situation mentioned above. Moreover, our method effectively infers the authentic FSMs from protocol implementations, enabling the identification of discrepancies between the actual protocol implementations and their respective RFC specifications.

To rigorously examine the proficiency of LLM in inferring state machines from protocol implementations, we meticulously infer the state machines of six distinct protocols: IKEv2, TLS1.3, TLS1.2, BGP, RTSP, and L2TP. The empirical findings from our study reveal that the state machines inferred via our methodology exhibit an average precision rate exceeding 90%. Notably, within these deduced state machines, we discern variances across disparate implementations of identical protocols. Furthermore, we apply the inferred FSMs to augment the efficacy of the protocol fuzzing tool AFLNet, thereby facilitating enhance code coverage. In essence, our contributions are threefold:

- We proposed a method to augment LLM with protocol implementation. This approach addresses limitations encountered by LLMs while analyzing source code.
- We pioneered a new approach, which was not proposed before, to infer protocol FSMs using augmented LLMs. This novel method has the capability to infer protocol states, message types, and state transitions from protocol implementations.
- We implemented our prototype, named ProtocolGPT, which can automatically infer FSMs from protocol implementations. We designed and conducted experiments demonstrating that the augmented LLMs achieve preicsion exceeding 90% in state machine inference. Through the application of ProtocolGPT, we uncovered discrepancies within the FSMs across different implementations of the same protocol. Additionally, AFNet enhanced by ProtocolGPT increases line coverage by 22.23% on average compared to baselines. We discovered two zero-day vulnerabilities and 18 previously known vulnerabilities.

2 BACKGROUND

In this section, we introduce protocol state machine inference techniques and large language models. We start by presenting several protocol state machine inference techniques, highlighting their respective advantages and limitations. Subsequently, we provide some background on large language models.

2.1 Protocol State Machine Inference

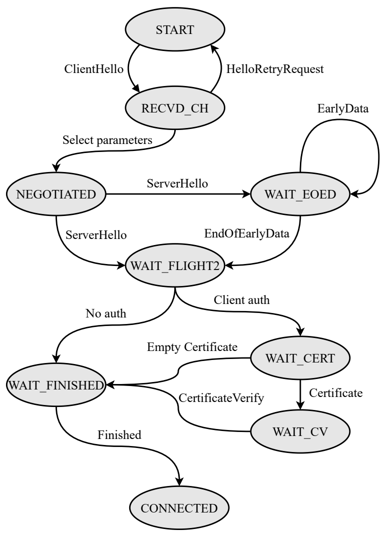


Fig. 1. The state machine of the TLS 1.3 protocol server defined in RFC 8446.

The protocol state machine is a model used to describe the interaction logic and behavior between peers. The essential components of a protocol state machine include the states of peers, inputs, and state transitions. The set of protocol states encompasses all possible states that can be reached by peers during the interactions. Input refer to all messages that can be sent or received by the server and client. Each protocol specification defines its unique input format, and both parties must strictly adhere to this format during interactions. The reception or transmission of a message at the communication terminal may lead to a state transition. The state machine of TLS 1.3 server is shown in Figure 1, highlighting states like START, RECVD_CH, and NEGOTIATED, and detailing the exchange of messages such as ClientHello, ServerHello, and Certificate. This interaction initiates in the START state and progresses through a series of exchanges to the CONNECTED state, establishing a secure connection between the server and client.

Previous efforts to infer protocol state machine include dynamic analysis [9] [27] [58] [28] [19] [22], static analysis [48], and natural language processing [43] [44] [54]. Dynamic program analysis investigates the data flow or control flow during message parsing, leveraging previously captured network traffic and utilizing technologies such as statistical method and active learning to deduce protocol formats. Conversely, static analysis engages in a direct examination of source code, employing symbolic execution and data flow analysis to infer protocol message formats. NLP, meanwhile, delves into the elucidation of transition relationships between protocol states using natural language inputs, such as RFCs and technical documentation.

2.2 Large Language Model

Large language models are a cutting-edge advancement in artificial intelligence, rooted in deep learning and natural language processing. Capable of performing tasks like text generating [5] [52] [8], translating [13] [25] [6], and summarizing text [51] [1], these models undergo a process of pre-training and fine-tuning to grasp the intricate structure and semantics of language. Recently, LLMs have excelled in software engineering, including code generating [12] [38], code repairing [56] [14] [26], and vulnerability detection [21] [34]. Almost all applications of LLMs in software engineering are based on prompt engineering and fine-tuning. With massive train data contained network protocol and source code, LLMs have the potential to deduce state machines from protocol implementations, showcasing their pivotal role in enhancing protocol security and efficiency.

LLMs demonstrate exceptional potential in program analysis. Nevertheless, they encounter specific challenges in conducting program analysis [7]. Firstly, the output generated by LLMs may occasionally manifest randomness, leading to the creation of non-existent facts. Secondly, in the face of intricate analysis tasks, LLMs are incapable of rendering efficient completions in a singular effort, necessitating incremental guidance via human-formulated prompts. Moreover, in scenarios involving complicated and advanced programs, the precision of LLMs falls short when compared to some conventional analysis methodologies. This discrepancy can largely be ascribed to the constrained scope of LLMs' context windows, which hampers their ability to thoroughly examine highly complicated and voluminous codebases in a singular analytical session.

3 MOTIVATION AND CASE STUDY

In this section, we use several cases as motivations to demonstrate the reasons and challenges of inferring state machines from the protocol implementations via LLM.

3.1 Inconsistencies between Protocol Implementations and RFCs

To investigate the inconsistencies between protocol specifications and implementations, we conduct a manual analysis of four implementations of the IKEv2 protocol, in conjunction with RFC 7296. Table 1 provides a summary of the number of states and state transitions for each implementation. To ensure the reliability of our results, two domain experts thoroughly examined both the RFC and the protocol implementations.

As illustrated in the table, there are marked discrepancies between the state machines described in the RFC and those observed in the protocol implementations. Several factors contribute to these discrepancies. Firstly, protocol specifications are typically documented in written form, leading to

Table 1. The state and state transition discrepancies between four IKEv2 implementations and the RFC.

Projects	States	Transitions
RFC [46]	8	17
strongSwan [50]	8	23
libopenikev2 [32]	22	65
Libreswan [33]	22	29
openswan [42]	12	19

Table 2. The comparison between state machines inferred by pre-trained model and augmented model.

Models	States	Transitions	Precision
Pre-trained LLM	4	12	43.48%
Augmented LLM	8	20	95.00%

varying interpretations and understandings among different developers or teams. This variability can result in significant differences between the FSMs of the protocol implementations and those defined in the RFC. Additionally, the official RFC guidelines for communication mechanisms include several open-ended definitions, allowing developers to design protocol implementations tailored to their specific application scenarios and functional requirements. Furthermore, developers may introduce custom features or extensions to the protocol implementations to address particular needs, which may not be supported by other implementations, thus leading to inconsistencies.

Consequently, there are inherent inconsistencies between the state machines of RFC and implementations. The state machines inferred from implementations, as opposed to those strictly defined in the RFC, can provide better insights for analyzing software behavior. However, as highlighted in Section 2, inferring state machines from implementations remains a complicated challenge.

Finding 1: Discrepancies between specifications and protocol implementations are ubiquitous across various systems.

3.2 Inferring protocol state machine via LLM

LLMPF [36] leverages the syntactic information generated by the LLM to enrich the seeds corpus used for fuzzing. Rather than utilizing the LLM augmented with specific knowledge, LLMPF directly employs a pre-trained model. To the best of our knowledge, augmented models can deal with more specific problems than pre-trained models. We thus compare the pre-trained model with the augmented model by employing a simple state machine inference case. For the pre-trained LLM, we adopt an approach akin to LLMPF, employing prompt engineering to incrementally guide the model in generating protocol states and transitions. In contrast, we utilize a pre-trained large language model augmented with source code to infer the state transitions. Initially, the pre-trained model is enhanced with the protocol implementation, followed by the application of the chain-of-thought technique to guide the augmented LLM in inferring the state machine. The methodology for state machine inference using the augmented LLM will be elaborated in Section 4. For the IKE protocol implementation, we selected strongSwan[50], a widely used solution in the industry.

Table 2 presents a comparison of the state machine inferred by the pre-trained model and the augmented model. The detailed experimental methodology is discussed in Section 5. The findings reveal that the state transitions inferred by the augmented model is 51.52% more accurate than that derived from pre-trained model. The enhanced LLM demonstrates the capability to extract reliable state machine information from the source code. While the pre-trained LLM can offer some

Table 3. The detail information about protocol implementations in our benchmark.

Protocols	Implementations	Version	Tokens
IKEv2	strongSwan[50]	f994e0a	5556945
TLS 1.3	s2n-tls [47]	025f3b2	2685894
TLS 1.2	s2n-tls [47]	025f3b2	2685894
BGP	openbgpd [40]	08b59c1	958486
RTSP	feng [15]	d302a1c	84763
L2TP	openl2tp [41]	be6c288	455270

insights related to FSMs, its accuracy is limited to 43.48%. This discrepancy may be attributed to the inherent randomness present during the generation process of the LLM.

Finding 2: Compared to the pre-trained model, the LLM augmented with source code exhibits a significantly enhanced capacity to generate a more reliable state machine.

3.3 Context Window of LLM

The performance of LLM is heavily dependent on the context information provided [31]. However, the length of context is limited by the input capacity of model. Moreover, the performance of model tends to degrade with an increase in input length. This limitation applies to both closed-source and open-source models. For example, OpenAI's GPT-4 model has a context window of only 8192 tokens. While this input length suffices for typical NLP tasks, it is inadequate for program analysis. As illustrated in Table 3, the number of tokens required for the implementations of protocols far exceeds the context window of the GPT-4 model.

When inferring state machines directly from implementation using pre-trained model, the complexity of the code and the sheer volume of files involved can hinder the model from producing accurate results. Protocol implementations typically comprise components such as state machines, configurations, tests, and encryption mechanisms. However, the code related to the state machine is often concentrated in specific files. For instance, the message types are usually defined in a single file. Identifying these specific files can narrow the scope of the code to be analyzed. Conversely, analyzing code related to configurations, encryption, authentication, and testing does not yield relevant information about the state machine itself.

Finding 3: The limitation of context window renders inferring state machine via LLM a challenging endeavor.

4 PROTOCOLGPT

Motivated by the findings discussed in the Section 3, we propose an approach (ProtocolGPT) for inferring state machines based on LLM. The core idea of our approach is to harness the code analysis capabilities of large language models to infer protocol finite state machines from complex protocol implementation. To overcome the challenges aforementioned, we use the augmented LLM that combines the strengths of both retrieval models and generative models. Figure 2 illustrates the overall workflow of ProtocolGPT, which consists of two primary components: LLM augmentation with source code and state machine inference.

LLM Augmentation with Source Code Initially, we isolate sections pertinent to the protocol state machine from the extensive and complex protocol implementation codebase. To guarantee efficient statement inference, the source code is segmented into smaller chunks to bypass limitations of LLM, such as the constraint on context window. Moreover, ProtocolGPT safeguards the

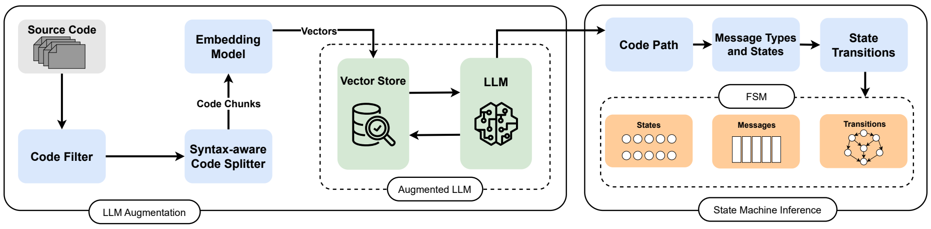


Fig. 2. The overview of ProtocolGPT.

integrity of the original code structure by syntax-aware segmenting according to classes, functions, and other structural components during the code segmentation. Our proposed approach ensures the preservation of semantic and contextual information, mitigating potential losses typically associated with traditional text segmentation technique. By embedding the LLM with these code chunks, ProtocolGPT could more precisely retrieve the most relevant code from vector store during the state machine inference phase.

State Machine Inference To enable the augmented LLM to infer FSM, we integrate the chain-of-thought [53] and background-augmented prompting techniques [35]. Leveraging domain-specific expertise, we formulate prompts tailored to the task at hand, which guide the augmented LLM in retrieving code segments that define both messages and states. The augmented LLM subsequently generates the message types and states based on the retrieved code. Finally, using the extracted message types and states, we traverse the entire state space to infer the corresponding transitions.

4.1 LLM Augmentation with Source Code

While LLMs exhibit remarkable proficiency in code analysis, their efficacy diminishes significantly when tasked with analyzing large-scale, structurally complex programs. To address this limitation, ProtocolGPT leverages retrieval-augmented generation (RAG) techniques[30], which empower LLMs by integrating customizable external knowledge sources, such as code repositories. Although RAG models mitigate the inherent limitation of relying exclusively on the training data and learned patterns, their performance deteriorates significantly if the code repositories are not adequately preprocessed. Specifically, protocol implementations frequently include code snippets that exceed the LLM's context window and often contain substantial amounts of code that are irrelevant to state machines, which may adversely impact retrieval accuracy, thereby compounding the challenges faced in effective code analysis. Consequently, preprocessing the code becomes a critical step in ensuring the LLM can be effectively augmented and leveraged for state machine inference.

First, the protocol implementation must be filtered to extract sections relevant to the state machine. The source code is then segmented into chunks to comply with the limitation of context window. The code chunks are then embedded and stored in a vector database, contributing to the enhancement of the pre-trained model's performance by enabling efficient retrieval and preserving contextual relevance. During the inference phase, relevant code chunks are retrieved from the vector database and fed into the LLM, supporting the generation of the state machine.

4.1.1 Code Filtering. To ensure scalability and maintainability, protocol implementations are typically structured into distinct modules, including state machines, configuration management, security and testing. Filtering out irrelevant code not only reduces the cost of LLM prompts but also enhances the accuracy and efficiency of FSM inference. Consequently, it is essential to focus on state machine module when inferring FSM from protocol implementations, as irrelevant modules do not contribute to the inference process. As discussed in Section 3.3, the tokens in protocol implementations significantly exceeds the context window of LLM, preventing analysis of the implementation. Therefore, the first step in augmenting the LLM is to filter out the code related to the protocol state machine from the entire codebase.

We observe that variables, constants, structures, and functions related to the state machine are often named according to terms defined in the RFCs. For instance, the CLIENT_HELLO message in s2n-tls[47] corresponds to the Client Hello message in the RFC. Additionally, keywords such as *state* and *message* frequently appear in these context-specific names. Code filter identifies the directory containing the state machine components within the protocol implementation through regular expression matching. Based on these observation, we define a comprehensive set of keywords related to protocol state machines, derived from both RFCs and expert knowledge. The keyword set includes message type and state keywords defined in the RFCs, supplemented by custom keywords based on expert knowledge. For instance, in the IKE protocol, Security Association (SA) appears frequently since the state of the SA essentially represents the state of the peer. These specially keywords facilitate the identification of state machine-related code.

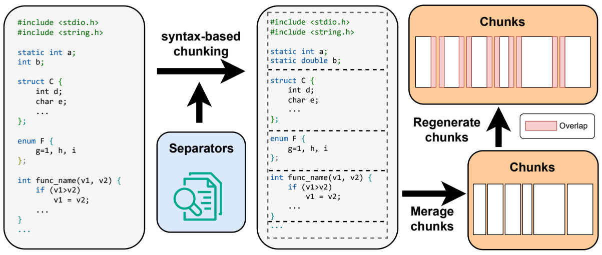
This pre-constructed keyword set is employed to perform regular expression matching on each document within the protocol implementation, and Matching documents are marked as state machine-related code. And the subdirectory where the documents have the highest hit rate will be selected as the state machine module, providing domain knowledge for LLM augmentation.

4.1.2 Syntax-aware Code Segmentation. For the RAG model, the retrieved data is directly provided as context to the prompt, enabling the LLM to generate a response. To retrieve data from the vector database, the documents are transformed into chunks and embeded by the embedding model which also has a limited context window. However, traditional character-based splitting strategies risk disrupting semantically meaningful elements and may result in the loss of contextual information. Consequently, two critical factors must be taken into account during code segmentation: the size of code chunks and the syntactic structure of the code.

Embeddings compress token-level representations into a single fixed-length vector, encapsulating the meaning of the entire input sequence. Appropriately sized chunks contain more precise contextual information, resulting in improved matching accuracy. The input provided to the LLM includes not only the retrieved documents but also instructions and relevant domain knowledge. However, if the length of the retrieved document approaches or exceeds the context window of LLM, the performance of the model may be adversely affected.

Conventional text segmentation overlooks the structural composition of the document. When applied to data such as code, which is characterized by inherent syntax, these strategies can compromise essential components, including structural and functional elements. Moreover, the rich contextual information embedded within the code may be degraded during the segmentation process.

Building upon the preceding observations, we propose a syntax-aware code splitting strategy. This method break down protocol implementation documents into appropriately sized chunks, ensuring that the syntactic integrity of the code remains intact while also preserving a sufficient level of contextual information. Figure 3 illustrates the workflow of our proposed syntax-aware splitting technique. (1) Initially, we define a maximum allowable chunk size, referred to as *MaxChunkSize*. We then recursively decompose the documents using a series of language-specific separators until the size of each chunk is smaller than *MaxChunkSize*. These separators consist of keywords that define critical elements such as structures, classes, functions, and other constructs. For instance, in the C programming language, separators include terms such as struct, class, enum, static, etc. To facilitate the application of our method across multiple programming languages, we compiled a comprehensive list of separators for languages including C, C++, Python, and others. (2) To prevent



```
#include <stdio.h>
#include <string.h>

static int a;
int b;

struct C {
    int d;
    char e;
    ...
};

enum F {
    g=1, h, i
};

int func_name(v1, v2) {
    if (v1>v2)
        v1 = v2;
    ...
}
...
```

```
#include <stdio.h>
#include <string.h>

static int a;
static double b;
```

```
struct C {
    int d;
    char e;
    ...
};
```

```
enum F {
    g=1, h, i
};
```

```
int func_name(v1, v2) {
    if (v1>v2)
        v1 = v2;
    ...
}
```

```
...
```

Fig. 3. The workflow of Syntax-aware Code Segmentation Strategy.

the creation of overly small chunks that may lack sufficient contextual information, we establish a predefined minimum threshold for chunk size, named *MinChunkSize*. If the length of the chunk falls below this threshold, it is merged with an adjacent chunk to ensure coherence. The value of *MinChunkSize* is specified by users. (3) Finally, to preserve some level of contextual continuity between chunks, we regenerate new chunks based on the previously segmented ones, ensuring a specified degree of overlap is maintained between them.

4.1.3 Vector Store. The augmented model employs semantic search to retrieve relevant documents from a vector store. Code snippets extracted from these documents are then appended to the original prompt, which is subsequently input into the generative model. Data retrieval from the vector store is facilitated by converting code into embeddings (high-dimensional vectors) that facilitate efficient search via a retrieval engine. The vector store is specifically optimized for storing embeddings of repository code from the protocol implementation, enabling the use of advanced search algorithm to identify matches between similar vectors.

When ProtocolGPT extracts information, it initially embeds the prompt to create an embedded version. It then retrieves the vectors from the vector store that most closely match the embedded prompt. Finally, the augmented LLM generate a more contextually relevant response based on the prompt posed by the original version and the code corresponding to these highly similar vectors.

For the storage and retrieval of unstructured code data, the optimal approach is to embed the data and store the resulting embedding vectors. Our approach leverages an embedding model provided by OpenAI[39] to transform code chunks into vector data. This vector data is subsequently stored and retrieved using a vector store facilitated by FAISS[11]. Each vector in the vector store corresponds to a specific piece of code data. The retrieval model employs the approximate nearest neighbor algorithm to retrieve the vectors relevant to the given query from the vector store.

4.2 FSM Inference

Before inferring FSM using the augmented LLM, it is essential to define the FSM structure that the model generates. Our primary objective is to examine the transition relationships between different states within the protocol implementation. Consequently, we anticipate that the FSM generated by the LLM serves as a high-level abstract model that encompasses both the set of states within the protocol implementation and the transitions between these states. Specifically, the FSM details the potential states the protocol server can attain, the pairs of source and destination states, and the conditions necessary for transitioning from the source state to the destination state. To facilitate more effective support for downstream applications, FSM discussed in this paper is assumed to be a non-deterministic FSM that can be transformed into a deterministic FSM. Notably, an non-deterministic FSM may possess multiple initial states, and a single state can transition to several successor states in response to the same input. Inspired by RFCNLP[43] and StateLifter[48], we define an non-deterministic FSM in Definition **1** as a formal model.

Definition 1. An FSM is a quintuple $(\Sigma, S, S_0, \delta, T)$ where

- Σ *is a non-empty set of all message types defined in the protocol implementation.*
- S *is a non-empty set of protocol states.* $S_0 \subseteq S$ *is a non-empty set of all initial states of the protocol application.*
- $\delta : S \times \Sigma \mapsto 2^S$ *is a state transition function. This function indicates that when the protocol application receives a specific message in a specific state, it will transition to the next state.*
- $T = \{(S', S'', m) | S' \subseteq S, S'' \subseteq S, m \subseteq \Sigma\}$ *is a non-empty set of all state transition relations.* (S', S'', m) *represents a state transfer relationship. When the protocol application is in the* S' *state, if message m is received, it will be transferred to the* S'' *state.*

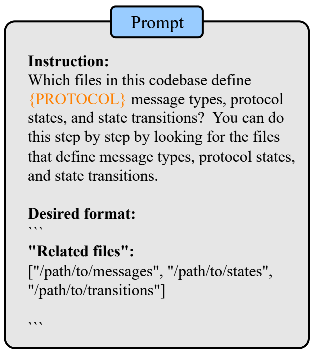
Moreover, a significant challenge in utilizing augmented LLMs for downstream tasks is ensuring that their outputs are machine-readable. Unlike static and dynamic analysis methods, the inputs and outputs of generative model are formulated in natural language. Therefore, without imposing constraints on the output of model, the resulting FSM would be unusable. This challenge can be addressed through fine-tuning prompts, which guide the LLM to perform specific tasks and generate outputs according to the given patterns. This approach ensures that the output generated by the LLM adheres to our desired format.

The FSM inference phase leverages prompt engineering to direct the augmented LLM in deriving the FSM from the source code. However, this process is not instantaneous. As the complexity of a task increases with the number of steps, the probability of errors in the LLM's output also rises. In contrast, the model demonstrates exceptional accuracy when executing single instructions. To mitigate this limitation, we employ the chain-of-thought technique [53], guiding the augmented LLM through each step of the process. Specifically, the FSM inference is structured into three distinct stages: (1) Retrieving the code snippets that define states, messages, and state transition relationships. (2) Extracting all the messages and states defined in the protocol implementation. (3) Identifying all the state transitions and the corresponding message types that trigger these transitions. By following these steps, ProtocolGPT gradually extracts the information which forms the FSM of the protocol implementation.

Moreover, the performance of LLMs may be less effective for domain-specific tasks that require specialized knowledge, primarily due to their limited exposure to relevant data. To address this, we employ the background-augmented prompting technique [35], which leverages domain knowledge to construct task-specific prompts. Specifically, we incorporate expert knowledge into the prompts concerning the particular protocol implemented by the repository code.

4.2.1 Code Paths. Message types and states are fundamental elements of network protocols. A state transition on the server is triggered when it resides in a particular state and a specific message is transmitted or received. As a result, message types and states are frequently invoked. To improve development efficiency and minimize system complexity, most protocol implementations encapsulate states and message types within defined structures or enumerations. These definitions are typically centralized in dedicated code files. For example, in the TLS protocol implementation s2n-tls, the *message_type_t* that defines all supported message types for s2n-tls is located in the file /s2n-tls/blob/main/tls/s2n_handshake.h. Identifying the relevant documents significantly facilitates the process by which the LLM derives the FSM's states, denoted as S and Σ.

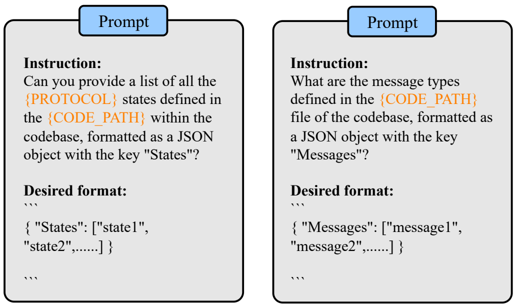
Figure 4 illustrates the prompt template designed to guide the augmented LLM in identifying the code paths related to state machine. For each protocol implementation, the prompt specifies the protocol type to ensure that the LLM retains all relevant information during generation.



Instruction:
Which files in this codebase define {PROTOCOL} message types, protocol states, and state transitions? You can do this step by step by looking for the files that define message types, protocol states, and state transitions.

Desired format:
```
"Related files":
["/path/to/messages", "/path/to/states", "/path/to/transitions"]
```

Fig. 4. Prompt template for obtaining the code paths related to state machine.


Prompt
Prompt

Instruction:
Can you provide a list of all the {PROTOCOL} states defined in the {CODE_PATH} within the codebase, formatted as a JSON object with the key "States"?

Desired format:
```
{ "States": ["state1", "state2",......] }
```

(a) States

Instruction:
What are the message types defined in the {CODE_PATH} file of the codebase, formatted as a JSON object with the key "Messages"?

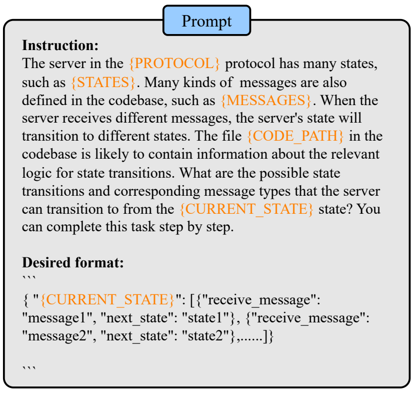
Desired format:
```
{ "Messages": ["message1", "message2",......] }
```

(b) Messages

Fig. 5. Prompt template used to obtain the states and message types defined in the protocol implementation.

4.2.2 Message types & States. The prompt templates employed to extract the set of message types and states are illustrated in Figure 5. By incorporating previously extracted code paths, these template ensure that the LLM maintains contextual continuity. These prompts guide the augmented LLM in analyzing the previously obtained code files, thereby facilitating the extraction of message types and states defined within the protocol implementation. Fundamentally, our approach leverages the LLM's code analysis capabilities to extract states and message types from structures or enumerated type variables. It is worth noting that our method can infer the FSM from the protocol implementation only when explicit definitions of states and message types are present. Nevertheless, our findings indicate that the majority of protocol implementations provide such explicit definitions, thereby making our approach viable.

The performance of LLMs may be less effective for domain-specific tasks that require specialized knowledge, primarily due to their limited exposure to relevant data. To address this, we employ the background-augmented prompting technique [35], which leverages domain knowledge to construct task-specific prompts. Specifically, we incorporate expert knowledge into the prompts concerning the particular protocol. Additionally, we specify the *Desired Format* in the prompt to ensure that the LLM outputs machine-readable results. The Augmented LLM will produce all message types and states in JSON format. These message types and states, encapsulated in the JSON data, will then be utilized to assist the augmented LLM in inferring state transitions.


Prompt

Instruction:
The server in the {PROTOCOL} protocol has many states, such as {STATES}. Many kinds of messages are also defined in the codebase, such as {MESSAGES}. When the server receives different messages, the server's state will transition to different states. The file {CODE_PATH} in the codebase is likely to contain information about the relevant logic for state transitions. What are the possible state transitions and corresponding message types that the server can transition to from the {CURRENT_STATE} state? You can complete this task step by step.

Desired format:
```
{ "{CURRENT_STATE}": [{"receive_message": "message1", "next_state": "state1"}, {"receive_message": "message2", "next_state": "state2"},......]}
```

Fig. 6. Prompt template for obtaining state transitions contained in protocol implementations.

4.2.3 State Transitions. The prompt template employed by ProtocolGPT to infer state transitions is depicted in Figure 6. We guide the augmented LLM to infer state transitions and the corresponding message types by utilizing code paths, predefined states, and message types. Inferring all state transitions within a protocol implementation is inherently a highly complex task. If the LLM were to attempt this task in a single step, it may result in suboptimal performance. Instead, we guide the LLM to systematically infer state transition relationships by iterating through all states.

We also provide *Desired format* in the prompt, which defines the pattern for state transitions. The LLM outputs state transitions in JSON format, following this specified pattern. Each JSON object includes a *current_state* key, whose value is an array of objects. Each object in the array contains two properties: *receive_message* and *next_state*. The *current_state* represents the peer's current status, the *next_state* denotes the adjacent state to which the transition occurs, and the *receive_message* specifies the type of message triggering the state transition. The peers can directly transition from the *current_state* to the *next_state* without passing through any intermediate states.

The example provided in Figure 1 illustrates the state transition. If the *current_state* is WAIT_CERT and a certificate message is received, the server transitions to the WAIT_CV state. Conversely, if an empty certificate message is received, the server transitions to the WAIT_FINISHED state. To capture all possible state transitions, ProtocolGPT generates multiple prompts based on the extracted states and the template shown in Figure 6, systematically replacing the *current_state* field with previously identified states until all states are traversed.

Furthermore, to address the issue identified in Section 3.2, where the LLM tends to generate random responses when extracting state machine-related information, we implemented a solution to enhance accuracy. To mitigate this problem, we conduct 20 iterations of dialogue using the augmented LLM for extracting states, message types, and state transitions. We then select the responses that appear with a probability greater than 80% across all iterations as the final result.

4.3 Implementation

We implemented our proposed method using the LangChain framework[29] which is a widely adopted open-source framework for developing agents based on LLMs. The framework provides an extensive suite of interfaces, including Model I/O, Retrieval, and Vector Stores, allowing developers to implement customized functionalities. ProtocolGPT implements both LLM augmentation and FSM inference functionalities within the LangChain framework. During the LLM augmentation phase, regular expressions are first used to match and filter out code files which are unrelated to the FSM. Documents are then segmented into manageable code chunks by our proposed syntax-aware splitting strategy which is implemented based on the text splitter provided by LangChain. The code chunks are then embedded by the model provided by OpenAI[39] and FAISS[11] and stored in a vector store. Ultimately ProtocolGPT employs the chain-of-thought technique and background-augmented prompting technique to guide the augmented LLM in inferring the protocol state machine. We will release the source code of ProtocolGPT and experimental data.

5 EXPERIMENTAL EVALUATION

To evaluate the effectiveness of ProtocolGPT, our assessment seeks to answer the following research questions:
RQ1: How effective is ProtocolGPT in inferring state machines compared to other methods?
RQ2: What the impact of the LLM augmentation on the performance of ProtocolGPT?
RQ3: What discrepancies can ProtocolGPT find between implementations of the same protocol?
RQ4: How much more code coverage does fuzzer enhanced by ProtocolGPT achieve compared to baselines?

ProtocolGPT is designed to be compatible with a broad range of LLMs, including offerings from OpenAI [39], Hugging Face [23], etc. For evaluation purposes, we select the widely acclaimed GPT-4 model, renowned for its extensive 1.7 trillion parameters. To provide a consistent and controlled environment for our experiments, they were conducted on a machine boasting 32GB of memory and a 12th Gen Intel(R) Core(TM) i7-12700 CPU, under Ubuntu 20.04.

Benchmark Our benchmark encompasses six network protocol implementations that are widely used in the Internet of Things, spanning a broad range of categories. These protocols vary in format and security levels, covering text-based protocols such as RTSP, binary protocols like IKEv2, encrypted protocols like TLS 1.2, and non-encrypted ones like BGP. For each protocol, we identify the implementation with explicitly defined states on GitHub and select the one with the highest number of stars, as shown in Table 3. As no ground truth was available in public datasets, we manually constructed the ground truth based on protocol specifications and implementations. Three protocol experts independently audited the RFCs and repository code of the six implementations, summarized the state machines, and then collaboratively refined the results to minimize bias. The final outcome serves as the ground truth, with the experts dedicating approximately 72 hours to this task. Note that manually constructing ground truth from protocol specifications and source code is a common practice on protocol reverse engineering, as demonstrated by prior works such as RFCNLP[43], NetPlier[58], and StateLifter[48].

5.1 Effectiveness

We selected three popular protocol reverse engineering tools and ChatGPT as baselines: RFCNLP[43], NetZob[4], NetPlier[58], and GPT-4[20]. RFCNLP, NetZob, and NetPlier represent the state-of-the-art in protocol reverse engineering and have been published in leading academic conferences. To compare the performance of augmented model with pre-trained model, we included GPT-4 as an additional baseline. For FSM inference with GPT-4, no source code hints were provided; instead, we directly used the prompt from Section 4.2 to generate state machines. As for static analysis technique, We fail to find any open-source static analysis methods capable of inferring state transitions. Existing static analysis techniques are limited to inferring protocol formats.

We conducted a manual analysis of the state machines inferred by ProtocolGPT, GPT-4, RFCNLP, Netzob, and NetPlier. In these analyses, we focused on the validity of state transitions as it reflects the validity of the states. State transitions are categorized into four types: correct, partially correct, incorrect, and not found. A state transition T is defined as $\{S_i, M, S_t\}$, where S_i is the initial state,

Table 4. State machines inferred by ProtocolGPT, GPT-4, RFCNLP, Netzob, and NetPlier. The precision is the ratio of correct state transitions to all inferred transitions; and the recall is the ratio of correct state transitions to all transitions in the ground truth.

Approach	Protocols	States	Transitions	Correct	Partially correct	Incorrect	Not Found	Precision(%)	Recall(%)
ProtocolGPT	**IKEv2**	8	20	19	0	1	4	95.00	82.61
	TLS 1.3	17	32	30	0	2	1	93.75	96.77
	TLS 1.2	16	30	29	0	1	2	96.67	93.55
	BGP	7	90	85	2	3	1	94.44	98.86
	RTSP	6	17	12	4	1	6	70.59	54.54
	L2TP	5	52	51	0	1	2	98.08	96.23
GPT-4	**IKEv2**	4	12	10	0	2	13	83.33	43.48
	TLS 1.3	6	10	3	1	6	18	30.00	10.71
	TLS 1.2	4	11	3	5	3	20	27.27	9.68
	BGP	6	27	14	3	10	9	51.85	60.87
	RTSP	6	27	4	0	23	18	14.81	18.19
	L2TP	6	13	1	3	9	52	7.69	1.92
RFCNLP	**IKEv2**	3	4	4	0	0	19	100.00	17.39
	TLS 1.3	9	10	7	0	3	26	70.00	21.21
	TLS 1.2	3	0	0	0	0	0	0	0
	BGP	6	151	67	60	24	21	44.37	76.14
	RTSP	4	13	11	0	2	11	84.62	50.00
	L2TP	4	16	5	0	11	48	31.25	9.43
Netzob	**IKEv2**	39	43	7	0	36	16	16.28	30.43
	TLS 1.3	12	16	7	5	4	19	43.75	21.21
	TLS 1.2	12	16	9	1	6	21	56.25	30.00
	BGP	6	10	5	0	5	83	50.00	21.73
	RTSP	17	24	13	3	8	6	54.17	59.09
	L2TP	13	15	5	2	8	46	33.33	9.43
NetPlier	**IKEv2**	60	62	36	0	26	12	82.26	47.83
	TLS 1.3	8	12	7	4	1	22	58.33	21.21
	TLS 1.2	16	19	5	0	14	25	26.31	16.67
	BGP	8	11	6	0	5	16	54.55	26.09
	RTSP	8	11	7	0	4	15	63.64	31.82
	L2TP	13	16	13	0	3	40	81.25	24.53

M is the message triggering the state transition, and S_t is the target state. If S_i, M, and S_t are all correct, then T is classified as correct. If one of S_i, M, or S_t is incorrect, then T is partially correct. If more than two elements of T are incorrect, then T is classified as incorrect. We then respectively compute the precision and recall of the inferred FSM as follows:

$$Precision = \frac{TP(T')}{TP(T') + FP(T')}; Recall = \frac{TP(T')}{|G|} \tag{1}$$

where $TP(T')$ refers to the number of correct state transition in the inferred FSM. $FP(T')$ denotes the total count of incorrect and partially correct state transitions, and G represents all the state transitions in the ground truth.

Table 4 presents the states and transitions inferred by ProtocolGPT, GPT-4, RFCNLP, Netzob, and NetPlier. The result demonstrates that the average precision and recall of state transitions inferred by ProtocolGPT exceed 90%. In contrast, GPT-4 demonstrates lower precision and recall, indicating that it failed to identify a significant number of state transitions. This highlights that LLM augmentation can effectively mitigate the generation of hallucinations by pre-trained models when inferring FSMs. RFCNLP demonstrates higher precision than GPT-4 but remains approximately 30% lower than our method, which can be attributed to the discrepancies between FSMs described in RFCs and those implemented in real-world protocols. While RFCNLP identifies FSMs based on RFCs, discrepancies frequently arise due to developer discretion and the optional nature of certain

Table 5. Comparison of the state machines inferred by the four variants of ProtocolGPT. The precision (P) is the ratio of correct state transitions to all inferred transitions; and the recall (R) is the ratio of correct state transitions to all transitions in the ground truth.

ProtocolGPT variant	IKEv2		TLS 1.3		TLS 1.2		BGP		RTSP		L2TP		AVG	
	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)
V0	83.33	43.48	30.00	10.71	27.27	9.68	51.85	60.87	14.81	18.19	7.69	1.92	35.83	24.14
V1	33.34	52.17	20.00	3.03	6.25	6.67	17.86	21.74	58.33	31.82	0.00	0.00	22.63	19.24
V2	37.50	26.09	40.00	25.81	51.72	46.67	35.00	30.43	70.00	31.82	68.19	28.30	50.40	31.52
V3	95.00	82.61	93.75	96.77	96.67	93.55	94.44	98.86	70.59	54.54	98.08	96.23	91.42	87.09

RFC requirements. Furthermore, as a deep learning model, RFCNLP relies on manually labeled data, which demands considerable human expertise and resources. In comparison to Netzob and NetPlier, our method achieves approximately 70% higher recall rates, as both tools are heavily reliant on the comprehensiveness of the test input coverage.

5.2 Ablation Study

The LLM augmentation process incorporates three primary strategies: code filtering, syntax-aware code segmentation, and vector store. To evaluate the individual contributions of these strategies, we conducted an ablation study. For this purpose, we developed four distinct variants of ProtocolGPT:

- V0: GPT-4, with all strategies were disabled.
- V1: V0 plus the embedding model.
- V2: V1 plus the syntax-aware code segmentation.
- V3: V2 plus the code filter.

Table 5 shows the results in terms of precision and recall. Compared to V0, V3 achieves a 55.59% increase in precision and a 62.95% increase in recall for inferred state machines. This indicates that LLM augmentation plays a crucial role in protocol state machine inference.

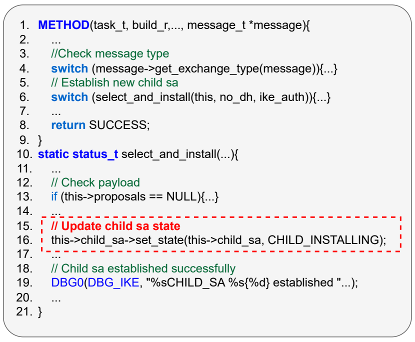
When comparing V1 with V0, there was a noticeable decrease in both precision and recall for V1. We attribute this decline to the different dependent data used by V0 and V1. Specifically, V0 exclusively utilized the pre-trained model, while V1 employed an augmented LLM. During state machine inferring, the output of V0 was derived from the extensive, training data of GPT-4, whereas V1's augmented LLM relied solely on specific knowledge drawn from external datasets. Notably, V1 do not segment the source code but instead embedded it directly into high-dimensional vectors. The excessive number of tokens within these vectors significantly degraded the model's performance. For example, during the inference of the L2TP state machine, V1 failed to function properly due to the token count far exceeding the context window of generative model.

A comparison between V2 and V1 reveals the substantial impact of the syntax-aware code splitting strategy. Specifically, V2 demonstrated an average increase of 27.77% in precision and 12.28% in recall, underscoring the significant benefits of this approach for state machine inference. This improvement is primarily due to the syntax-aware code splitting mechanism, which segments large portions of source code into smaller, more tractable units while preserving the integrity of the syntactic structure. This approach ensures that semantic information is retained, effectively mitigating the risk of meaning loss during the partitioning process. It effectively addresses the problem of token counts exceeding the context window of the generative model. By alleviating this constraint, the overall performance of the model is significantly enhanced.

The comparison between V3 and V2 highlights the impact of the code filtering mechanism. V3 exhibited a substantial increase in precision and recall, by 41.02% and 55.57%, respectively. This

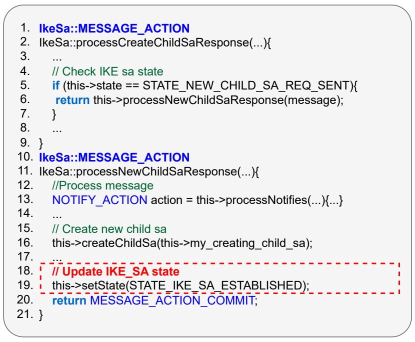
Table 6. The state machines of different IKEv2 protocol implementations inferred by ProtocolGPT.

Implementations	States	Transitions	Correct	Partially correct	Incorrect	Not Found	Precision(%)	Recall(%)
strongSwan	8	20	19	0	1	4	95.00	82.61
libopenikev2	22	43	32	0	11	22	74.42	66.67
Libreswan	22	30	29	0	1	0	96.67	100.00
Openswan	12	20	18	0	2	0	90.00	94.74



```
METHOD(task_t, build_r,..., message_t *message){
    ...
    //Check message type
    switch (message->get_exchange_type(message)){...}
    // Establish new child sa
    switch (select_and_install(this, no_dh, ike_auth)){...}
    ...
    return SUCCESS;
}
static status_t select_and_install(...){
    ...
    // Check payload
    if (this->proposals == NULL){...}
    ...
    // Update child sa state
    this->child_sa->set_state(this->child_sa, CHILD_INSTALLING);
    ...
    // Child sa established successfully
    DBG0(DBG_IKE, "%sCHILD_SA %s{%d} established "...);
    ...
}
```

(a) strongSwan



```
IkeSa::MESSAGE_ACTION
IkeSa::processCreateChildSaResponse(...){
    ...
    // Check IKE sa state
    if (this->state == STATE_NEW_CHILD_SA_REQ_SENT){
     return this->processNewChildSaResponse(message);
    }
    ...
}
IkeSa::MESSAGE_ACTION
IkeSa::processNewChildSaResponse(...){
    //Process message
    NOTIFY_ACTION action = this->processNotifies(...){...}
    ...
    // Create new child sa
    this->createChildSa(this->my_creating_child_sa);
    ...
    // Update IKE_SA state
    this->setState(STATE_IKE_SA_ESTABLISHED);
    return MESSAGE_ACTION_COMMIT;
}
```

(b) libopenikev2

Fig. 7. Examples of state transition differences between protocol implementations.

improvement is largely attributable to the code filter's ability to eliminate code that is irrelevant to the state machine, thus reducing the search space for the retrieval engine. By restricting the augmented LLM from retrieving unrelated vectors, the efficiency of state machine inference is significantly enhanced. However, for protocols with smaller codebases, the effect of the code filter is less pronounced. For example, when inferring the RTSP state machine, the relatively compact size of the protocol's implementation resulted in minimal differences between the filtered and unfiltered code. As a result, the performance gains of V3 over V2 in this context are marginal.

5.3 Inconsistencies Between Protocol Implementations

To assess ProtocolGPT's ability to discern variations between different protocol implementations, we applied it to infer four distinct implementations of the IKEv2 protocol which is a critical component of IPSec suite. A detailed comparative analysis of these implementations is provided in Table 6. Our evaluation revealed substantial discrepancies in the number of states and state transitions among the various IKEv2 implementations, highlighting the nuanced differences in their operational behaviors. For example, strongSwan[50] is characterized by having just 23 state transitions, in contrast to libopenikev2[32], which exhibits as many as 43 state transitions. This notable difference in count of state transition underscores the varying complexity and operational intricacies between these IKEv2 implementations.

Our examination of the state transition relationships in FSMs reveals distinct differences among various protocol implementations. For example, in IKEv2, when the communicating parties complete the establishment of an IKE SA, different protocol implementations exhibit varied state transitions upon the creation of a Child SA. Figure 7 demonstrate the state transition processes for creating a Child SA in strongSwan [50] and libopenikev2 [32], respectively.

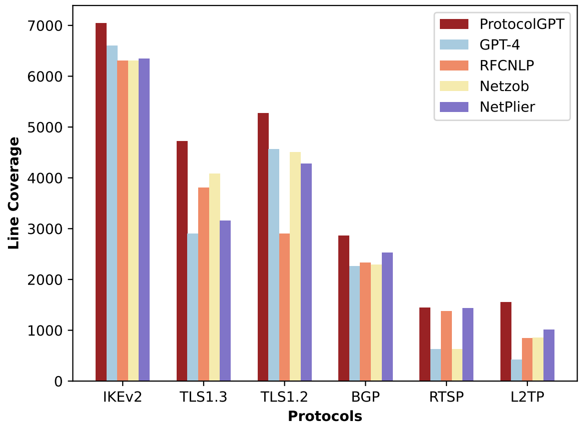


Fig. 8. Average line coverage of AFLNet enhanced by ProtocolGPT, GPT-4, RFCNLP, Netzob, and NetPlier in 5 runs of 24 hours.

Figure 7(a) illustrates that in strongSwan, once the peer completes the establishment of the IKE SA (entering the ESTABLISHED state) and initiates a request to create a Child SA with the other peer, the state of the IKE SA remains unchanged. Instead, this action triggers a state transition for the Child SA. In strongSwan, the IKE SA and Child SA manage their state transitions independently. In contrast, as shown in Figure 7(b), in libopenikev2, after the IKE SA is established (in the STATE_IKE_SA_ESTABLISHED state), sending a request to create a Child SA results in a state transition for the IKE SA to STATE_NEW_CHILD_SA_REQ_SENT. Once the Child SA is created, the state reverts to STATE_IKE_SA_ESTABLISHED.

5.4 Security Application

In protocol fuzzing, state machines are instrumental in exploring more code branches and states effectively. AFLNet leverages protocol response codes to construct protocol state machines, enabling targeted testing of the protocol implementation's various states as per the state diagram. Utilizing effective message sequences from the corpus as seeds, AFLNet incorporates lightweight mutation algorithms to enhance test coverage. By deriving message sequences from the FSMs inferred by ProtocolGPT, we generate targeted seeds for AFLNet testing. We adopt the same approach to generate fuzz seeds through the state machines inferred by RFCNLP[43], Netzob[4] and NetPlier[58]. To ensure the reliability of our experimental data, each protocol implementation is tested five times, with each session lasting 24 hours, thereby minimizing randomness in the results.

Figure 8 clearly demonstrates that fuzzers enhanced by ProtocolGPT achieve a 22.23% increase in code coverage, underscoring the capability of ProtocolGPT-inferred state machines to uncover more code execution paths and states than those inferred by RFCNLP, Netzob, and NetPlier. Apart from enhanced coverage, we discovered two zero-day vulnerabilities and 18 previously known vulnerabilities. In contrast, other methods identified only five known vulnerabilities. This is attributed to ProtocolGPT's ability to identify a greater number of state transitions, facilitating the discovery of additional code paths. However, in the context of encryption protocols such as IKEv2, TLS1.2, and TLS1.3, AFLNet encounters significant constraints due to its inability to access cryptographic keys during fuzzing, thereby limiting the testing scope to pre-key negotiation phases. The implementation of the BGP protocol represents a distinctive case. Its state transitions, driven by events such as the establishment of connections between autonomous systems, updates to routing information, and administrative operations, differ substantially from those in other protocols. Consequently, message-based seeds alone may prove insufficient to achieve comprehensive code coverage when fuzzing BGP protocol implementations, underscoring the necessity of integrating event-driven triggers into the fuzzing strategy.

6 RELATED WORK

Natural Language Processing Pacheco et al. advanced a novel methodology for inferring protocol state machines, anchored in NLP and adopting a data-driven paradigm [43]. This method involves the training of deep learning models on a comprehensive corpus of natural language data, encompassing technical documentation and protocol specification documents. Subsequently, leveraging labeled data, this approach facilitates the derivation of relationships between variables within the protocol specifications via the model, culminating in the articulation of a finite state machine representative of the protocol.

Dynamic Analysis Kleber et al. unveiled a novel strategy for the categorization of message types within binary protocols [28]. This strategy relies on a similarity metric using continuous value ranges to compare feature vectors based on message fields instead of discrete byte values, thereby facilitating an enhanced discernment of message format patterns. Further contributing to the field, Ye et al. advanced a probabilistic approach to the reversal of network protocols [58]. By constructing a joint distribution between these random variables and the observed values within messages, probabilistic reasoning is employed to ascertain potential keywords. Gopinath et al. introduced a comprehensive algorithm designed to deduce the syntax of inputs from programs, distinguishing itself by eschewing conventional data flow analysis methodologies [19]. Uniquely, this methodology obviates the necessity for heuristic strategies traditionally employed to discern parsing patterns, rendering it universally applicable across all recursive parsers that operate on the basis of program stacks.

Static Analysis Shi et al. pioneered the development of a protocol state machine inference methodology predicated on static analysis [48]. This innovative approach employs static analysis to examine protocol formats derived from parsers within the protocol implementations. The extraction of state machines is achieved by interpreting each iteration of a loop as a distinct state, with the interdependencies among loop iterations being regarded as transitions between states.

7 CONCLUSION

Inferring state machines from protocol implementations is inherently laden with complexities and challenges. In the context of intricate protocol implementations, conventional methodologies exhibit limitations in accurately inferring state machine. On one hand, static analysis contends with the dilemma of state explosion during the inference process. On the other hand, with respect to the specific endeavor of state machine inference, the outcomes yielded by static analysis lack the requisite precision and comprehensiveness. Consequently, We propose an innovative state machine inference approach powered by augmented LLM and conduct experiments on the benchmark. The outcomes of these experiments underscore the capability of LLMs to facilitate the derivation of relatively robust state machines from protocol implementations. Moreover, employing ProtocolGPT enabled us to conduct a comparative analysis of state machines across various implementations of identical protocols, through which we successfully delineated discrepancies among the state machines corresponding to these implementations. Conclusively, our investigations corroborated that the state machines inferred by ProtocolGPT could significantly augment protocol fuzzers by enhancing code coverage and facilitating the uncovering of previously unidentified bugs.

REFERENCES

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 3255–3272.

[3] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. 2017. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 483–502.

[4] Georges Bossert, Frédéric Guihéry, and Guillaume Hiet. 2014. Towards automated protocol reverse engineering using semantic information. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security* (Kyoto, Japan) *(ASIA CCS '14)*. Association for Computing Machinery, New York, NY, USA, 51–62. https://doi.org/10.1145/2590296.2590346

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[7] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. 2023. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595* (2023).

[8] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.

[9] Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. 2007. Discoverer: Automatic Protocol Reverse Engineering from Network Traces.. In *USENIX Security Symposium*. Boston, MA, USA, 1–14.

[10] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[11] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. (2024). arXiv:2401.08281 [cs.LG]

[12] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 865–865.

[13] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.

[14] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1469–1481.

[15] feng 2024. *Feng - standard compliant streaming server*. Retrieved October 30, 2024 from https://github.com/lscube/feng

[16] Sidong Feng and Chunyang Chen. 2024. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[17] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri De Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of {DTLS} implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. 2523–2540.

[18] Paul Fiterau-Brostean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. 2023. Automata-Based Automated Detection of State Machine Bugs in Protocol Implementations.. In *NDSS*.

[19] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining input grammars from dynamic control flow. In *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 172–183.

[20] GPT-4 2024. *GPT-4 is OpenAI's most advanced system, producing safer and more useful responses.* Retrieved October 30, 2024 from https://openai.com/index/gpt-4/

[21] Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1865–1879.

[22] Matthias Höschele and Andreas Zeller. 2016. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 720–725.

[23] Hugging Face 2024. *The AI community building the future.* Retrieved October 30, 2024 from https://huggingface.co/

[24] Abdullah Al Ishtiaq, Syed Md Mukit Rashid Sarkar Snigdha Sarathi Das, Kai Tu Ali Ranjbar, Zhezheng Song Tianwei Wu, Mujtahid Akon Weixuan Wang, and Syed Rafiul Hussain Rui Zhang. 2024. Hermes: Unlocking Security Analysis of Cellular Network Protocols by Synthesizing Finite State Machines from Natural Language Specifications. In *33st USENIX Security Symposium (USENIX Security 24)*.

[25] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*. 1219–1231.

[26] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1430–1442.

[27] Stephan Kleber, Henning Kopp, and Frank Kargl. 2018. {NEMESYS}: Network message syntax reverse engineering by analysis of the intrinsic structure of individual messages. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*.

[28] Stephan Kleber, Rens W van der Heijden, and Frank Kargl. 2020. Message type identification of binary network protocols using continuous segment similarity. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2243–2252.

[29] langChain 2024. *LangChain is a framework for developing applications powered by language models.* Retrieved October 30, 2024 from https://www.langchain.com

[30] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 9459–9474. https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

[31] Dacheng Li, Rulin Shao, Anze Xie, Ying Sheng, Lianmin Zheng, Joseph Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. 2023. How Long Can Context Length of Open-Source LLMs truly Promise?. In *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*.

[32] libopenikev2 2024. *A library providing the core IKEv2 funcionability.* Retrieved October 30, 2024 from https://github.com/OpenIKEv2/libopenikev2

[33] libreswan 2024. *An Internet Key Exchange (IKE) implementation for Linux, FreeBSD, NetBSD and OpenBSD.* Retrieved October 30, 2024 from https://github.com/libreswan/libreswan

[34] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2023. Testing the limits: Unusual text inputs generation for mobile app crash detection with large language model. *arXiv preprint arXiv:2310.15657* (2023).

[35] Ziyang Luo, Can Xu, Pu Zhao, Xiubo Geng, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Augmented Large Language Models with Parametric Knowledge Guiding. arXiv:2305.04757 [cs.CL] https://arxiv.org/abs/2305.04757

[36] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*.

[37] John Narayan, Sandeep K Shukla, and T Charles Clancy. 2015. A survey of automatic protocol reverse engineering tools. *ACM Computing Surveys (CSUR)* 48, 3 (2015), 1–26.

[38] Erik Nijkamp, Hiroaki Hayashi Bo Pang, Huan Wang Lifu Tu, Silvio Savarese Yingbo Zhou, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[39] OpenAI 2024. *OpenAI is an AI research and deployment company.* Retrieved October 30, 2024 from https://openai.com/

[40] openbgpd 2024. *OpenBGPD is a FREE implementation of the Border Gateway Protocol.* Retrieved October 30, 2024 from https://www.openbgpd.org/

[41] openl2tp 2024. *OpenL2TP is a complete implementation of RFC2661.* Retrieved October 30, 2024 from https://github.com/Distrotech/openl2tp

[42] Openswan 2024. *An IPsec implementation for Linux.* Retrieved October 30, 2024 from https://github.com/xelerance/Openswan

[43] Maria Leonor Pacheco, Max von Hippel, Ben Weintraub, Dan Goldwasser, and Cristina Nita-Rotaru. 2022. Automated attack synthesis by extracting finite state machines from protocol specification documents. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 51–68.

[44] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. {WHYPER}: Towards automating risk assessment of mobile applications. In *22nd USENIX Security Symposium (USENIX Security 13)*. 527–542.

[45] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460–465.

[46] RFC 7296 2024. *Internet Key Exchange Protocol Version 2 (IKEv2).* Retrieved October 30, 2024 from https://datatracker.ietf.org/doc/html/rfc7296

[47] s2ntls 2024. *s2n-tls is a C99 implementation of the TLS/SSL protocols.* Retrieved October 30, 2024 from https://github.com/aws/s2n-tls

[48] Qingkai Shi, Xiangzhe Xu, and Xiangyu Zhang. 2023. Extracting protocol format as state machine via controlled static loop analysis. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7019–7036.

[49] Zhan Shu and Guanhua Yan. 2022. Iotinfer: Automated blackbox fuzz testing of iot network protocols guided by finite state machine inference. *IEEE Internet of Things Journal* 9, 22 (2022), 22737–22751.

[50] strongSwan 2024. *strongSwan is an OpenSource IPsec-based VPN solution.* Retrieved October 30, 2024 from https://github.com/strongswan/strongswan

[51] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. 2022. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239* (2022).

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[53] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 24824–24837. https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf

[54] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. 2015. Dase: Document-assisted symbolic execution for improving automated software testing. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 620–631.

[55] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. *Proc. IEEE/ACM ICSE* (2024).

[56] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1482–1494.

[57] Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. 2024. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[58] Yapeng Ye, Zhuo Zhang, Fei Wang, Xiangyu Zhang, and Dongyan Xu. 2021. NetPlier: Probabilistic Network Protocol Reverse Engineering from Message Traces.. In *NDSS*.