Logos: Log Guided Fuzzing for Protocol Implementations

Feifan Wu
BNRist, Tsinghua University
Beijing, China
wufeifan1026@gmail.com

Zhengxiong Luo*
BNRist, Tsinghua University
Beijing, China
fouzhe15@gmail.com

Yanyang Zhao*
BNRist, Tsinghua University
Beijing, China
zhaoyanyang@mail.tsinghua.edu.cn

Qingpeng Du
Beijing University of Posts and
Telecommunications
Beijing, China
duqingpeng10515@gmail.com

Junze Yu
BNRist, Tsinghua University
Beijing, China
yujz21@mails.tsinghua.edu.cn

Ruikang Peng
Central South University
Changsha, China
pengruikang775@163.com

Heyuan Shi
Central South University
Changsha, China
hey.shi@foxmail.com

Yu Jiang
BNRist, Tsinghua University
Beijing, China
jiangyu198964@126.com

Abstract

Network protocols are extensively used in a variety of network
devices, making the security of their implementations crucial. Pro-
tocol fuzzing has shown promise in uncovering vulnerabilities in
these implementations. However traditional methods often require
instrumentation of the target implementation to provide guidance,
which is intrusive, adds overhead, and can hinder black-box test-
ing. This paper presents Logos, a protocol fuzzer that utilizes non-
intrusive runtime log information for fuzzing guidance. Logos first
standardizes the unstructured logs and embeds them into a high-
dimensional vector space for semantic representation. Then, Logos
filters the semantic representation and dynamically maintains a
semantic coverage to chart the explored space for customized guid-
ance. We evaluate Logos on eight widely used implementations
of well-known protocols. Results show that, compared to exist-
ing intrusive or expert knowledge-driven protocol fuzzers, Logos
achieves 26.75%-106.19% higher branch coverage within 24 hours.
Furthermore, Logos exposed 12 security-critical vulnerabilities in
these prominent protocol implementations, with 9 CVEs assigned.

CCS Concepts

• **Security and privacy → Network security**; *Software and ap-
plication security*.

Keywords

Protocol Fuzzing, Vulnerability Detection

*Corresponding authors.

1 Introduction

Protocol implementations are crucial to network infrastructure
and are typically exposed directly to the network. Therefore, it is
essential for them to effectively handle any malformed or malicious
traffic to prevent attackers from exploiting undetected flaws. For
example, the infamous Heartbleed [12] vulnerability discovered
in OpenSSL [40] affected a significant portion of network devices
and services. Identifying and addressing potential vulnerabilities
in protocol implementations prior to their release is essential to
prevent such incidents.

Fuzzing is a widely used testing technique for vulnerability dis-
covery. For testing protocol implementations, fuzzing involves
continuously generating and sending malformed protocol pack-
ets to the under-test implementation while observing for anomalies.
While traditional protocol fuzzing techniques have been widely
adopted and proven effective in identifying various vulnerabilities,
they still have significant limitations. Specifically, while some proto-
col fuzzers use user-defined test models for valid packet generation,
they generate packets randomly without considering the testing
state. As a result, they fail to identify whether the generated pack-
ets trigger new program states, missing opportunities for further
exploration. Recent work attempts to address this limitation by
incorporating feedback mechanisms, with code coverage being a
prevalent choice [35, 38, 45]. Nonetheless, this approach requires
instrumenting the source code or binaries, which can be invasive
and add additional overhead to program execution. It may also be
infeasible for black-box testing scenarios. Conversely, we observe
that programs inherently produce logs, eliminating the need for
additional intrusive procedures. These logs provide information
regarding the program's state and behavior, which is invaluable

for protocol testing. More importantly, the points at which logs are generated are closely scrutinized by developers, as they typically correlate with error-prone areas, making them crucial for identifying vulnerabilities.

This paper introduces Logos, a protocol fuzzer that provides a mechanism for real-time unstructured log modeling and uses it for guided packet generation. To achieve this approach, we need to address two challenges: (i) How to model the heterogeneity of unstructured log formats in a unified way. Since all logs may be from different protocols and implementations, unified modeling is the basis for avoiding manual intervention. (ii) How the log information can be used to efficiently guide fuzzing. Logs can contain a significant amount of fuzzing information, such as statistical data or periodically printed environmental information, which may be irrelevant to the testing process. In addition, this information must be processed into a form that the fuzzing framework can use.

For the first challenge, we propose a non-invasive unified protocol modeling mechanism. This mechanism first stabilizes numerical instabilities within the logs and eliminates redundant cyclic printing to obtain a more stable and normalized data source. Then, it uses deep learning models to extract the semantic meaning of the logs, thereby transforming various information from heterogeneous unstructured log formats into a high-dimensional vector representation called semantic representation. Such a semantic modeling approach bypasses the limitations imposed by specific log formats and directly captures the semantic representation. The resulting semantic representation can adequately express information about the protocol implementations discovered during a fuzzing round, as derived from the logs.

To address the second challenge, we propose a guided fuzzing approach that fully exploits the modeling results. Logos first filters semantic representation to merge semantically similar logs to dynamically maintain the historical highest semantic coverage (*semantic coverage* for short). Semantic coverage can reflect the semantic coverage of logs generated by this round of the fuzzing process. We then calculate the new coverage of log messages generated by the current packet over semantic coverage and semantic representation and output a low-dimensional value rating representing the diversity of log information. We then perform rating-weighted packet generation using a packet pool structure specifically designed for rating to efficiently generate protocol packets and discover additional bugs in the protocol implementations.

We evaluated Logos's performance on eight popular implementations of well-known protocols, including TLS, DNS, and CoAP. We compared Logos against four state-of-the-art protocol fuzzers, including two guided fuzzers (AFLNet [45] and ChatAFL [36]) and two knowledge-driven fuzzers without testing guidance (BooFuzz [26] and Peach [16]). The experimental results show that Logos outperforms these prior works in branch coverage by 39.28%, 26.74%, 106.19% and 40.93% on average, respectively, over 24 hours. Logos also discovered 12 new bugs in these previously well-tested protocol implementations. In comparison, AFLNet, ChatAFL, BooFuzz, and Peach could only expose 5, 5, 4, and 5 of them. Most of these vulnerabilities are security-critical and 9 CVEs were assigned due to their severe security influences. In addition, the ablation study demonstrates that the proposed contrastive learning for log modeling and

the log guidance mechanism is essential for the effectiveness of Logos. Our main contributions are as follows:

- We propose log data of protocol application to guide protocol fuzzing, providing a non-invasive and fully automated approach that requires no expert knowledge.
- We design *Log Modeling* as a unified approach to model logs by extracting semantic information and then leverage the modeled artifacts through *Guided Fuzzing* to guide the packet generation.
- We implement Logos and evaluate it on previously well-tested protocol implementations. The results show that Logos outperforms traditional coverage-guided fuzzers and manual expert knowledge-based approaches. It has also discovered many security-critical vulnerabilities.

2 Observation

We use CoAP [1] to demonstrate the System Under Test (SUT) log information often overlooked by conventional protocol fuzzers. Figure 1 shows an example log from the libcoap server [39]. In a log file from a libcoap server, the recorded data includes basic and advanced details. Basic information includes timestamps, categorization of log types, network identifiers such as IP addresses and port numbers, and session-specific information. In addition, the logs also capture more complex, abstract semantic content: events, actions, and anomalies.

```
1   08:17:01.411 DEBG EVENT: COAP_EVENT_TCP_CLOSED
2   08:17:01.411 DEBG EVENT: COAP_EVENT_SESSION_CLOSED
3   08:17:01.422 DEBG 127.0.0.1:5783 <-> 127.0.0.1:48090 TCP: send 1472 bytes
4   08:17:01.422 DEBG 127.0.0.1:5783 <-> 127.0.0.1:48090 TCP: send 1472 bytes
5   08:17:01.423 DEBG 127.0.0.1:5783 <-> 127.0.0.1:48090 TCP: send 1472 bytes
6   08:17:01.423 DEBG 127.0.0.1:5783 <-> 127.0.0.1:48090 TCP: send 1472 bytes
7   08:17:01.423 DEBG EVENT: COAP_EVENT_SESSION_CONNECTED
8   08:17:01.426 DEBG 127.0.0.1:5783 <-> 127.0.0.1:48090 TCP: recv 92 bytes
9   Exception: timed out
10  CORE DUMP: 17.33.16#2023-9-26
```

Figure 1: Log file example from libcoap (CoAP) server

We observe that the amount of information presented by these logs can vary significantly, ranging from abundant to minimal. In addition, the types of information supported by log data are not consistent, either in terms of basic or semantic content. This suggests that no assumptions can be made about the specific nature of log information. However, we have found that logs with different templates invariably correlate with different branches of code. This relationship is denoted by the function $\phi(\cdot)$, which maps a log to the corresponding branch of the code that generates the log. Assuming that each log entry corresponds to a single line of the code (otherwise different templates in the same branch will always appear at the same time, which can be considered as one template), for two log entries L_1 and L_2, we define the relationship as:

$$L_1 \neq L_2 \Rightarrow \varphi(L_1) \neq \varphi(L_2)$$

The probability of such occurrences is directly proportional to the density of log statements in the code. Table 1 shows log statements' line and function coverage in different SUT instances. From this observation, we can speculate that the coverage of printed logs relative to the total number of log statements can reflect code coverage.

However, in the scenarios described, if two sibling branches do not have different log statements to distinguish them, the logs will

Table 1: The function coverage and line coverage of the log statement in the selected SUTs

SUT	BoringSSL	libcoap	OpenSSL	CycloneDDS
Fun. Cov.	32.93%	27.16%	26.45%	24.33%
Lin. Cov.	19.64%	11.86%	13.22%	11.86%
SUT	Mosquitto	Dnsmasq	DNSPod-sr	smartdns
Fun. Cov.	16.21%	28.62%	17.10%	29.37%
Lin. Cov.	10.92%	17.34%	1.66%	5.74%

not reflect their differences. Figure 1 illustrates that lines 1 and 2 contain identical log statements, but these two different events result in different handling functions, representing two different branches. Nonetheless, by incorporating semantic information from these events, it becomes possible to distinguish between subsequent sibling branches by recognizing the disparities in their semantics.

In particular, we found that common logs can contain several types of semantic information, including (i) *Event*. Event information is the most important message in reactive processing systems. The transfer of information from one part of a protocol implementation to another often relies heavily on the transmission of events. In such reactive systems, the sequence of events also correlates with the sequence of system state transitions. The system states represented by these events provide a direct model of the system state. (ii) *Action*. Actions are direct representations of the behavior of the protocol implementation, and often of the functionality itself. The appearance of a new action in logs typically indicates that new functionality has been covered by the fuzzer, increasing the likelihood of vulnerability discovery. For example, actions such as send(line 3) and recv(line 8) are often logged along with certain values to form the trace of a system, which is the most common method of logging. (iii) *Anomalies*. In addition, evidence of anomalies and inconsistencies within the logs can be used to detect system bugs, allowing developers to more quickly locate bugs and potentially modify subsequent branches. Typically, the places where logs are generated are areas of the code that developers pay close attention to because they are prone to errors, so covering statements related to logs is more advantageous for bug detection. In the logs shown in Figure 1, there are instances of anomaly information such as *timeout* and *CORE DUMP* (line 9-10). Identifying these errors can effectively extend the coverage of protocol testing to different scenarios and increase the likelihood of finding vulnerabilities.

Basic Idea. Building on these observations, it is possible to integrate a mechanism into the protocol fuzzing framework that effectively exploits the semantics of log data. Such a mechanism can efficiently serve as a non-intrusive replacement for the traditional invasive coverage instrumentation: Firstly, accurately reflecting code branch coverage requires extracting log coverage information effectively. In addition, for the additional semantic information provided within the logs, it is necessary to model this information as comprehensively as possible to improve the accuracy of this coverage. This involves identifying a greater number of events, actions, anomalies, etc. Then, considering the prevalence of irrelevant and redundant information in the code, it is critical to effectively filter out such content during the modeling process. The remaining logs,

after filtering, will thus be much more meaningful. However, it is equally important to avoid excessive filtering to avoid any loss in the approximation of the code branch coverage.

Challenges. To use log information to guide protocol fuzzing, we need to address the following two challenges:

C.1: The heterogeneity of unstructured log formats. Different protocols have different semantics, and different implementations produce different logs. Even within the same SUT, different log printing modes, such as DEBUG or INFO, can change the original form of the SUT's logs. As a result, the log formats of different SUTs are almost completely different. The variety of log formats not only makes it difficult to extract information but also poses a significant challenge to the modeling of semantic information. In previous work [2], the use of log information was often limited to a specific domain, such as the *Android Execution Log*s. This limitation allowed the use of simple and uniform rule-based log parsing algorithms. Additionally, expert prior knowledge was applied to categorize potential code points for extraction, thereby streamlining the subsequent information retrieval processes. In the context of protocol testing, the variability of unstructured log formats and the impracticality of experts to pre-categorize information for extraction present considerable challenges. Consequently, the ability to adapt log modeling to different unstructured formats becomes critical and a significant challenge.

C.2: Efficient guidance for protocol fuzzing. Efficient protocol testing requires effective use of modeling results. First, it is critical to eliminate parts of the protocols that are numerically unstable or redundant. In addition, efforts should be made to minimize the loss of approximation to code branch coverage when filtering out semantic-independent information. Then we need to effectively translate the protocol modeling information into scores for generating packets. In previous research, such as the work done with AFL [19], fuzzers typically employ coverage-oriented strategies. They use various statistical metrics, such as whether coverage growth is triggered or the number of mutation rounds, as parameters for heuristic functions. These functions then generate specific scores that guide weighted sampling from the seed pool to select seeds for testing. In fact, the statistical data generated in traditional fuzzing processes is typically low-dimensional, making it easily manageable for heuristic functions. However, log information, once modeled, contains rich semantics, resulting in inherently high-dimensional data. The challenge is to effectively map this high-dimensional semantic information to a low-dimensional probability sampling space for the heuristic function. This process requires sophisticated techniques to distill complex, multi-faceted log data into actionable insights that can efficiently guide the fuzzing process without losing critical semantic information.

3 System Design

Overview. Figure 2 gives an overview of the Logos framework. The *log modeling* module processes logs from the SUT, culminating in the construction of the semantic representation and semantic coverage. The *guided fuzzing* module then uses these modeled entities to continuously generate and send effective packets to the SUT, thereby facilitating the effectiveness of fuzzing.
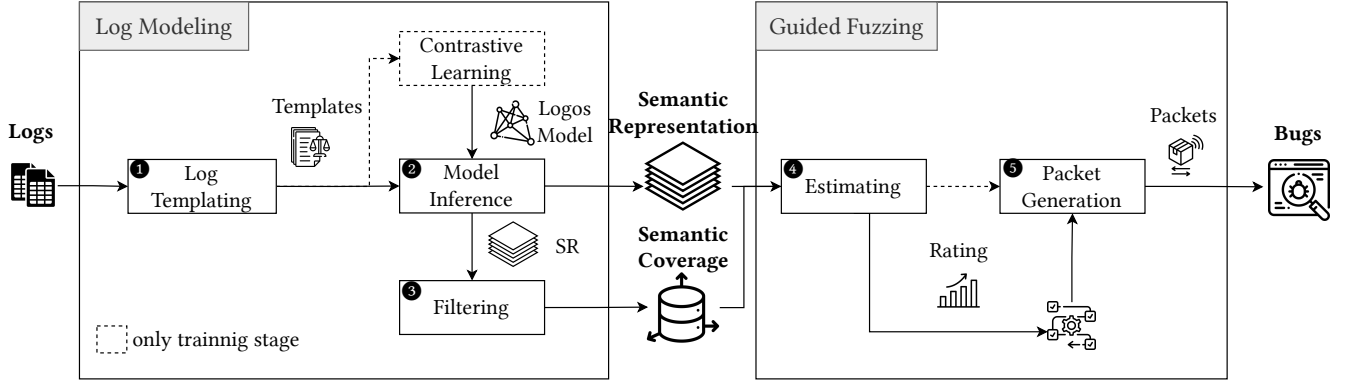
Figure 2: Logos Overview. Logos consists of two modules: log modeling and guided fuzzing. The log modeling module processes logs to train and generate a log model. This model is used to infer the vector representation of the log batch (semantic representation) and dynamically maintain the historical highest semantic coverage (semantic coverage). The guided fuzzing module uses them to continuously generate low-dimensional rating for structural packet generation aimed at finding more bugs.

The *log modeling* module is an integral component of the Logos framework, tasked with converting logs from the SUT into structured, modeled entities, specifically semantic representation and semantic coverage. This transformation is crucial for facilitating the *guided fuzzing* module, which subsequently employs these entities to craft and dispatch effective packets back to the SUT, enhancing the bug detection process. The procedure commences with *log re-templating*, which standardizes logs to counteract the effects of numerical instability and recurring patterns, which ensures that the logs maintain a uniform format. Next, the log model is trained using contrastive learning to capture the semantic information contained in the logs. This self-supervised learning phase aligns the embeddings with the log semantic space, effectively capturing the semantic behavior of the SUT. In the *model inference* phase, the trained log model infers the semantic representation from real-time logs. And *semantic representation filtering* refines the log analysis by filtering out semantic representation vectors in the embedding space that is too proximate, thereby reducing redundancy.

The *guided fuzzing* module is designed to optimize the packet generation process. This module utilise the information encapsulated in the semantic representation and semantic coverage to increase the effectiveness of fuzzing. *Semantic representation estimating* converts high-dimensional vector representations of logs into numerical rating. This mapping is essential for the packet generation phase, where a rating is required to construct extraction probabilities. During packet generation, the *packet pool* is utilized to manage an evolving set of packet sequences, which are then processed by the *packet generation* algorithm. This algorithm aims to optimize the selection and sequencing of packets based on the insights gained from the semantic representation estimating evaluations. By integrating the semantic representation and semantic coverage into the *guided fuzzing* process, the Logos framework continuously adapts its strategy based on real-time feedback. This adaptive approach ensures that the fuzzing process remains targeted and efficient.

3.1 Log Modeling

The log modeling module is used to transform logs generated by the SUT into two modeled entities: semantic representation and semantic coverage. The semantic representation models the information triggered in the current interaction, whereas the semantic coverage represents an accumulated model of the information already triggered in the system. The module has two primary phases: (1) During the training phase, the module processes a curated dataset of logs along with indicators of whether these logs improve coverage. This phase involves tuning the semantic space of the model and training the index for semantic representation filtering. (2) In the inference phase, the system only analyzes logs generated in real-time by the SUT without coverage labels.

Log Re-Templating. During each iteration of the process, *log modeling* continuously collects logs from the log collector throughout the execution. The collected logs are then undergo a process of re-templating into standardized templates. This log re-templating process ensures uniformity and consistency in log format. To enhance the robustness and efficiency of the modeling process, and to counteract the impact of numerical instability and recurrent patterns in the logs. This step is crucial for maintaining the integrity of the model in the face of such variances. Figure 1 shows a raw example log from the CoAP protocol, containing details such as time, IP address, port, and byte count, along with a repeated display of transport information (Lines 3-6).

```
1    %d:%d:%f DEBG EVENT: COAP_EVENT_TCP_CLOSED
2    %d:%d:%f DEBG EVENT: COAP_EVENT_SESSION_CLOSED
3    %d:%d:%f DEBG %a:%d <-> %a:%d TCP: send %d bytes
4    %d:%d:%f DEBG EVENT: COAP_EVENT_SESSION_CONNECTED
5    %d:%d:%f DEBG %a:%d <-> %a:%d TCP: recv %d bytes
6    Exception: timed out
7    CORE DUMP: %d.%d.%d#%d-%d-%d
```

Figure 3: A libcoap (CoAP) log example after re-templating

To address numerical instability issues, we have introduced a regex-based matching and replacement method, and the rules are shown in the Table 2. The primary sources of numerical instability

are typically items 1, 2, and 3. The separation of hexadecimal and decimal expressions is due to their different applications. Items 4, 5, and 6 aim to reduce instability caused by random network and file addresses in multiple tests. Item 7 addresses the impact of random symbols on the NLP process. To address the issue of repetitive logs, we have implemented a neighbor merging technique. After the above replacements are complete, if there are at least two consecutive logs that are identical, they are merged into a single entry. Upon completion of the numerical stability and loop printing processing, the logs will be as concise as shown in Figure 3, thus facilitating more efficient modeling in subsequent stages.

Table 2: Log templating target and replacement

	Target	Replaced By
1	Integers	%d
2	Floats	%f
3	Hexadecimal numbers	%x
4	IP addresses	%a
5	Port	%d
6	File paths	%p
7	Other symbols, i.e., %?@!#$&*	%r

Contrastive Learning. After the re-templating process, the logs become relatively stable, providing a foundation for further processing. The generated templates are then used to train a log model through contrastive learning. At this stage, the deep learning model acquires semantic information from the logs under a self-supervised signal, including details about events, actions, and anomalies. This method enables the model to effectively understand and represent the nuanced information contained in the unstructured log data.

As highlighted in section §2, protocol implementations have heterogeneous protocol formats. To effectively navigate these complex rules, we employ Natural Language Processing (NLP) methods for modeling, which are better suited to adapt to such subtleties. Our proposed methodology utilizes a pre-trained BERT [13] as the base model and integrates contrastive learning [8] to further align the distances between embedding vectors with the differences in code coverage. This approach leverages BERT's language understanding capabilities to effectively process the logs. Using contrastive learning, we introduce self-supervised signals as cues, allowing the distances within semantic representation to represent differences in code coverage. It is used to pair the embedding information output by BERT with the code coverage bitmap of the SUT at the time of protocol generation, forming a single training sample. The loss function is constructed using a triplet formulation:

$$L(s_a, s_+, s_-, \theta) = \max\left(0, m + \theta(s_a, s_+) - \theta(s_a, s_-)\right)$$

s_a denotes the current log sample, s_+ represents a sample sharing the same coverage with s_a, and s_- signifies a randomly selected sample possessing different coverage. The function $\theta(a, b)$ calculates the distance between samples a and b after mapping them into the embedding space. The margin m is employed to regulate the distance between s_+ and s_-. It allows our model to further align the distance of log embeddings with the distance of code coverage. Furthermore, contrastive learning eliminates the need for manual annotation, thereby increasing the efficiency of model training.

Contrastive learning and coverage labels are used only during the training phase, to develop a model that further aligns log embedding information with code coverage. In subsequent applications, it is sufficient to set the model parameters and perform inference directly on the model. This component is deactivated during the inference phase to maximize efficiency, ensuring that the log model generated by the contrastive learning component is used effectively

Model Inference. Model inference occurs in two distinct phases: (i) During the training phase, the log model from the current iteration of contrastive learning is used for inference to generate training semantic representation. These semantic representation are then used to train the semantic representation filter index and to generate packets in the downstream *guided fuzzing* module. (ii) In the inference stage, the fully trained log model developed by contrastive learning is used for inference. No further updating of the model is required at this stage. The logs for inference are obtained exclusively from the SUT in real-time.

As shown in Figure 4, the semantic representation, a vector representation of logs, is conceptualized as a high dimensional tensor of the form $\mathbb{R}^{n \times embed}$, where n is the number of log entries generated by SUT, meaning that each tensor can be associated with a distinct log entry. Meanwhile, *embed* represents the dimensionality of the semantic space $\mathbb{R}^{embed}$ in which each log entry is modeled. Overall, this tensor represents the collective embedding of all logs during a single interaction.
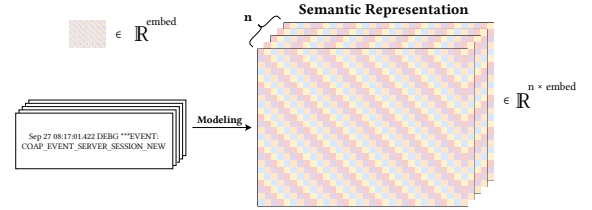

Figure 4: The shape of the semantic representation consists of n tensors, each of which is an *embed*-dimensional tensor

Semantic Representation Filtering. Semantic representation filtering is an essential part of the log analysis process, designed to filter out logs in the embedding space that are too close together. By filtering and removing redundant states, it reduces instability in the subsequent use of the semantic representation. This process effectively extracts a collection of different semantic representation vectors, which ultimately form a semantic coverage for subsequent packet generation. As shown in Figure 5, the semantic coverage dynamically maintains a subset of the semantic representation consisting of vectors that are distinct from each other. In subsequent scheduling processes, it helps guide the packet generation to uncover states that are markedly different from the current ones.

During the training phase, semantic representation filtering uses all collected semantic representation for index training. This ensures that subsequent phases of storage and retrieval during inference are highly efficient. During the inference phase, as shown in Figure 5, semantic representation filtering decomposes logs associated with multiple semantic representation entries and queries each corresponding semantic representation against the semantic

coverage. If any queried semantic representation scores fall below a certain threshold, they are not stored in the semantic coverage; otherwise, they are added. Given the extremely sparse nature of this semantic space, the threshold for error tolerance in selection is quite broad. Consequently, the process of deciding whether to store logs in the semantic coverage effectively completes the semantic representation filtering. Logs retained in the semantic coverage represent those that are significantly away from the current state, reflecting an approximation of code coverage and diverse semantics.
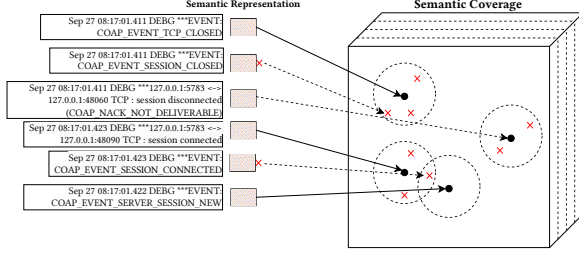

Figure 5: The semantic coverage dynamically maintains a subset of semantic representation

3.2 Guided Fuzzing

After completing training with the *Log Modeling* module, a trained log model and a semantic coverage are obtained. The former continuously transforms real-time logs generated by the SUT into semantic representation, while the latter preserves the current log information. To effectively use the information from these two components for fuzzing, the *guided fuzzing* module has been designed.

Semantic Representation Estimating. It is a critical step in the log analysis process where the goal is to map the high-dimensional vector representations extracted in semantic representation to a number.This transformation is pivotal for the packet scheduling phase, where a numerical rating is necessary to construct extraction probabilities. To address the challenges of efficiently generating data packets from high-dimensional data, we have introduced a mapping technique that effectively uses the semantic coverage to ensure that the generated values accurately reflect the semantic information of the logs corresponding to the current packets. This technique facilitates the precise translation of complex data structures into actionable and meaningful packet data, optimizing the process for accuracy and efficiency.

In practice, each semantic representation is cross-referenced with the semantic coverage. If no similar vector is found, the corresponding semantic representation is added to the semantic coverage, and the number of additions is recorded as the semantic representation's *rating*. The magnitude of this rating reflects the number of new log entries covered by the semantic representation, meaning that a higher rating indicates more new log information is covered. In addition, the semantic representation estimating process transmits its evaluation results to the *packet pool* and facilitates the provision of historical guidance for subsequent scheduling decisions.

Packet Pool and Scheduling. A packet pool is designed to use the results of semantic representation estimation during packet
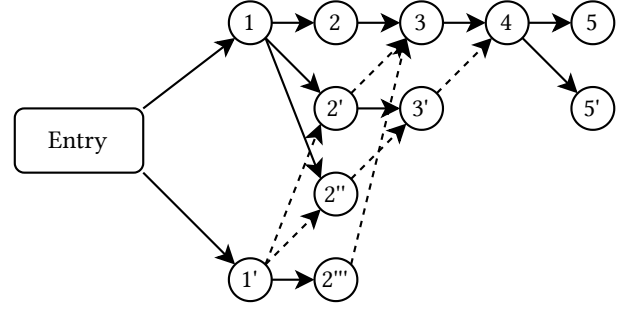

Figure 6: The structure of the packet pool is composed of multiple packet sequences

generation. Packet scheduling aims to optimize the selection and sequencing of packets based on insights from the evaluations, which is essential for the construction of effective packet sequences. Traditional fuzzing methods, such as AFL[19], use seed queue scheduling to determine the next seed for mutation. Similarly, our packet pool represents a collection of packet sequences, where each seed corresponds to an independent packet session. At the start of a session with SUT, packet scheduling extracts seeds from the packet pool based on their ratings.

The structure of the packet pool is shown in Figure 6. If a packet $\mathcal{P}$ is mutated to $\mathcal{P}'$, and sending $\mathcal{P}'$ generates a rating greater than 0, then $\mathcal{P}'$ is added to the packet pool. Also, $\mathcal{P}'$ is connected with a solid line to the previously sent packet $\mathcal{P}_{prev}$, and with a dashed line to the subsequent packets of $\mathcal{P}$. For example, the sequence (1 2 3 4 5) represents the initial seed. The sequence (1 2′ 3 4 5) is generated by mutating p2, resulting in a rating greater than 0. In this sequence, after sending p1 and p2′, if mutating and sending p3 results in a rating greater than 0, then p3′ is added to the pool. P3′ is connected to p2′ with a solid line and to p4 with a dashed line.

By exploiting the structure of the packet pool, we have designed a packet generation algorithm 1, to efficiently generate packets based on their ratings. The packet generation starts with the packet pool and first computes a *path*, which represents a basic sequence of packets. Before transmitting this packet sequence, mutations are applied. If a mutated packet generates a rating > 0, it is kept in the packet pool. The process of calculating a path is described in (lines 13-18). For each selected packet node P, a child packet node is selected based on the ratings and added to the path. This iterative, weighted selection process ensures that the path is likely to consist of sequences with higher ratings while preserving the possibility of exploring paths with lower ratings. After packet scheduling, and before packets are sent to the SUT by the Packet Sender, a decision is made on whether to mutate the packets. If a packet is mutated and the mutated version gives a rating greater than zero, it is then inserted into the packet pool (line 10). After acquiring a mutated packet P' with a rating greater than 0, it is necessary to insert P' into the packet pool. The insertion method, as shown in Figure 6. In addition, it is imperative to update the rating of the packet P and its ancestors, where the update is done by selecting the maximum value at each node with the new rating $\mathcal{R}$.

Algorithm 1: Packet Generation

Input :Packet Pool Q

1 **Algorithm**
2   path ← CalculatePath(Q), mutated ← 0
3   **foreach** *Packet P of* path **do**
4   **if** mutated < limit *and* Random() < $\mathcal{P}_{MUTATE}$ **then**
5   P' ← Mutate(P)
6   resp ← Send(P')
7              $\mathcal{R}$ ← CalculateRating(resp)
8   **if** $\mathcal{R}$ > 0 **then**
9   mutated ← mutated + 1
10   InsertPacket(P', $\mathcal{R}$)
11   **else**
12   Send(P)

13 **Procedure** CalculatePath(Q)
14   P ← Root(Q)
15   **while** P **do**
16   P ← ChooseNextPacket(P)
17   path ← path ∪ P
18   **return** path

4 Evaluation

In this section, we implement and evaluate Logos to answer the following two research questions:

RQ1 How does the performance of Logos compare to state-of-the-art protocol fuzzers? (§4.3)

RQ2 How does each component contribute to the effectiveness of Logos? (§4.4)

4.1 Implementation

We develop a prototype of Logos in Python 3 to demonstrate the effectiveness of the proposed approach.

Log Modeling. Using PyTorch 2, we develop a log model and train it on the *Sentence Transformers* framework, a Python framework for state-of-the-art sentence embedding, to ensure compatibility with its interface. Our model structure is based on *nli-distilroberta-base-v2* [47] and with a 250 dimension of semantic space. Given the log file with the corresponding coverage bitmap, we add a contrast learning layer(with a margin of 1.0) after the base model and use a loss mentioned in §3.1. We use ptrace[27] to get a transient code coverage bitmap as the coverage label. Then we use faiss[17], a library for efficient similarity search and clustering of dense vectors, to build a vector index with an inverted file with an exact post-verification algorithm to filter out similar embeddings(with Euclidean distance as the metric of distance) and emit a semantic coverage.

Guided Fuzzing. To exploit the results of the protocol modeling module, we implement a protocol fuzzing framework using Python 3 and Scapy, a Python framework that allows the user to send, sniff, analyze, and forge network packets. First, we implement the packet mutation operators, including number, string, list, enumeration, and length, using Scapy's field identification support. Second, following the strategy in §3.2, we implement the packet pool with the

mutation operators. Using the packets generated by the mutation operators and the packet pool, we construct a runtime to collect logs, generate packets, send packets, and monitor vulnerabilities with SUT. Besides the widely used local process monitoring, we also implement several network monitors that can remotely detect system crashes to support black-box fuzzing.

4.2 Experiment Setup

Dataset. The dataset $\mathcal{D}$ used for contrastive learning training is derived from the set of protocol implementations $\mathcal{P}$ as selected in §4.2. For the protocols $\mathcal{D}_p$ of each protocol implementation p, we have $\mathcal{D} = \{\mathcal{D}_p | \forall p \in \mathcal{P}\}$. For the model $\mathcal{M}_p$ corresponding to each protocol implementation p, the dataset $\mathcal{D} - \{\mathcal{D}_p\}$ is used. This approach aims to reduce the dependency of our model on specific protocols. The methodology for collecting protocols involves using basic examples provided by the protocol implementations.

Subjects. Our subject selection criteria prioritize diversity in protocol types, implementations with different protocol formats, and implementations with different log statement coverage (including the function coverage and line coverage of the log statement). Guided by the selection criteria outlined, eight protocol implementations were identified, as shown in Figure 7. The protocols selected for this study cover several categories, including security, messaging, transport, and industrial control systems. These protocols are not only emblematic of their respective categories but are also widely used in real-world scenarios. And to test the effectiveness of Logos on different implementations of the same protocol, we specifically chose three different implementations of the DNS protocol: dnsmasq [51], dnspod-sr [14], and smartdns [46]. The function coverage and line coverage of the log statement in these selected subjects are also quite diverse, ranging from 16.21% to 32.92% and from 1.66% to 19.64%, respectively.

Compared Fuzzers. Since Logos is a protocol fuzzer, we select four typical protocol fuzzers, Peach [16], AFLNet [45], BooFuzz [26] and ChatAFL [36], prevalent in academia and industry as baselines. We used server utilities associated with SUTs to evaluate the effectiveness of various protocol fuzzers. Adapting these fuzzers to our chosen subjects involved a meticulous configuration process, closely following the guidelines provided in their tutorials[16, 26, 36, 44]. This configuration was two fold: (1) For generation-based fuzzers Peach and BooFuzz, we developed a detailed test model for each SUT. The test model consists of two key aspects: the data model, which outlines the format and types of protocol messages, and the state model, which defines their sequence and state transitions within a session. To ensure the fairness of the mutation-based fuzzer, we chose the model corresponding to the seed that the mutation-based fuzzer is equipped with. For the data model, we chose the message type contained in the seed; for the state model, we used the state transition contained in the seed. The preparation of the data model and the state model was done according to the Request for Comments(RFC). (2) For mutation-based fuzzers, including AFLNet, ChatAFL, and Logos, our approach was to prepare an initial set of seed inputs. These seed inputs were collected by executing the off-the-shelf utilities in the SUT and simultaneously capture the network traffic to prepare the initial packet seeds. This corpus of initial inputs is critical to the effective

mutation-based fuzzing process. For protocols supported by AFLNet and ChatAFL, the default configuration was used. At the same time, we extend AFLNet and ChatAFL to support the CoAP, MQTT, and RTPS protocols with the corresponding protocol specifications and the tools' extension tutorials.

Experiment Settings. Each fuzzing tool was run on the selected projects for 24 hours to account for the inherent variability in fuzzing performance. This duration was chosen to provide a comprehensive evaluation period for each tool. Recognizing the impact of randomness on the results, we replicated each 24-hour experiment five times. This repetition was critical to ensure the statistical significance of our results, as described in reference [29]. Each fuzzing session was run in a standardized environment. Specifically, we used Docker containers, each configured with identical resources: 1 CPU core and 1 GB of RAM. This uniform setup was essential to ensure that any observed differences in the performance of the fuzzing tools were due to their inherent characteristics rather than variations in the test environment.

4.3 Comparison with Prior Work

We evaluate the effectiveness of Logos by comparing its code coverage and the number of unique bugs detected against those achieved by existing state-of-the-art protocol fuzzers.

Code Coverage. Given the variety of fuzzing methodologies being compared, the adoption of consistent and fair evaluation metrics is critical. Branch coverage is widely accepted and used in software testing to evaluate the effectiveness of fuzzers. It provides a quantifiable measure of the extent to which fuzzers test different code paths in a program. Therefore, we chose it as the primary metric for comparing the effectiveness of different fuzzers. To accurately measure branch coverage, we used LLVM Sanitizer-Coverage [31]. This tool facilitates the calculation of the unique branches covered by each fuzzer in the target program. In our research, this approach provided standardization and objectivity in evaluating and comparing the performance of different fuzzers.

To ensure the reliability of the experiment, each fuzzer was run five times on the protocol implementations under test. Figure 7 highlights the 24-hour average branch coverage and its range of Logos and four compared fuzzers across eight protocol implementations through different colors. On average, Logos outperformed AFLNet, Peach, BooFuzz, and ChatAFL in branch coverage by 39.28%, 40.93%, 106.19%, and 26.74%, respectively, within five 24-hour test runs. In all cases, the minimum branches achieved by Logos exceed the maximum branches of the previous approaches. The Mann-Whitney U test (p<0.01) recommended by Klees et al. [29] and Vargha-Delaney (>0.7) [5] verified the statistical significance of these results, indicating that the superiority of Logos branch coverage is not caused by random changes.

For the DNS protocol, we deliberately chose three different implementations to observe the behavior of Logos under different implementations of the same protocol. We found that for DNSPod-sr and smartdns, the two projects with low log statement line coverage, only 1.66% and 5.74% respectively, Logos's exploration of semantic diversity allowed the potential increase in code coverage to be mitigated. And for dnsmasq, although it did not significantly improve coverage, it found new CVEs that these fuzzers did not find because

of its exploration of log semantics. For Mosquitto, libcoap, and CycloneDDS, it is difficult to test the deeper logic, since these protocols are concerned with special semantic sequences rather than sequences of states, and neither the expert knowledge fuzzers nor the code instrumentation fuzzers generate packets at the semantic level. However, Logos compensates well for this by introducing the packet pool, which makes it possible to generate packets in the direction of exploring richer log semantics.

For tools based on the code instrumentation technique, including AFLNet and ChatAFL, since code coverage is treated the same for all branches, it may take too much time to test meaningless branches. It is more efficient to test a semantically relevant subset of code coverage with high weights. For the expert knowledge-based fuzzers Peach and BooFuzz, the data model and state model written by experts have a large generation space but a limited semantic space, and it is difficult to break through the coverage to the bottleneck due to the lack of SUT's feedback to guide it. Especially on DNS protocol, due to its protocol message format being simple, and expert knowledge-based fuzzers are difficult to model complex semantics, the performance on DNS and coverage-guided tools produce differences.

Comparing AFLNet, Peach, BooFuzz, and ChatAFL, Logos can approximate the effect of the existing SOTA works of code branch coverage by exploring the variety of protocols implemented by the protocol and without intrusive and expert knowledge. And because Logos gives extra weight to log semantics, it actually produces better results than existing tools in OpenSSL and Mosquitto.

Capability of Bug Discovery. To fairly compare the ability of each tool to trigger vulnerabilities, we chose the number of memory security issues as the unified metric. Traditional protocol fuzzing test tools have different abilities to monitor vulnerabilities. For example, Peach mainly detects vulnerabilities through active port scanning or ICMP ping. However, this approach may not trigger all types of vulnerabilities, such as buffer overflows, and other vulnerabilities that do not cause the program or host to crash may not be detected. Therefore, we choose AddressSanitizer [49] and UndefinedBehaviorSanitizer [10] (referred to as ASan and UBSan, respectively) as metrics to detect individual SUT vulnerabilities.

Logos has discovered 12 new vulnerabilities in widely used implementations of protocols, resulting in the assignment of 9 CVE identifiers through a coordinated disclosure process. Some SUTs have been extensively tested. In particular, implementations such as Dnsmasq[21] and libcoap[20] have been integrated into OSS-Fuzz[22], highlighting the ability of Logos to generate error-triggering packets. Table 3 summarizes the vulnerabilities discovered by Logos and the ability of other fuzzers to detect these problems. Notably, Peach, AFLNet, BooFuzz, and ChatAFL were only able to trigger a subset of the vulnerabilities found by Logos. The concentration of vulnerabilities identified by these tools suggests that the reasons for their discovery are likely to be similar. Conversely, the CVE in dnsmasq, introduced as early as 2021 and tested by OSS fuzz, went undetected because of their deep logical complexity. Fuzzers that rely on expert knowledge typically do not model this complex logic. In addition, fuzzers that use code instrumentation often miss this deeper logic because they do not even improve coverage, leading to its neglect. However, by focusing on exploring broader log coverage, Logos facilitates the formation of

Table 3: Previously unknown vulnerabilities exposed by Logos and the statistics of the compared fuzzers

Subject	Information	AFLNet	Peach	Boofuzz	ChatAFL	Logos	Status
libcoap	Remote DoS attack caused by the *coap_context_t*.					•	CVE-2023-51847
dnsmasq	Malformed data in *forward_query* causes integer overflow.					•	CVE-2023-49441
dnspod	Global buffer overflow occurs in *create_new_log* function.					•	CVE-2024-22524
dnspod	SEGV caused by using *strcpy* for an abnormal *send_buf*.					•	CVE-2024-22525
robdns	*lock* size overflow causing heap information leakage.					•	CVE-2024-24192
robdns	Undefined behavior caused by misaligned type casting of *&buf*.	•	•	•	•	•	CVE-2024-24195
robdns	Undefined behavior caused by applying zero offset to null pointer *&token*.					•	CVE-2024-24194
robdns	*parser* size overflow causing in global information leakage.					•	CVE Requested
smartdns	Undefined behavior caused by misaligned type casting of *&sa_family_t*	•	•		•	•	CVE Requested
smartdns	Undefined behavior caused by misaligned type casting of *&buffer*	•	•	•	•	•	CVE-2024-24199
smartdns	Accessing of a misaligned variable *sockaddr_storage*	•	•	•	•	•	CVE-2024-24198
smartdns	Undefined behavior caused by misaligned type casting of *&buffer*	•	•	•	•	•	CVE Requested
SUM		**5**	**5**	**4**	**5**	**12**	**9 CVEs**
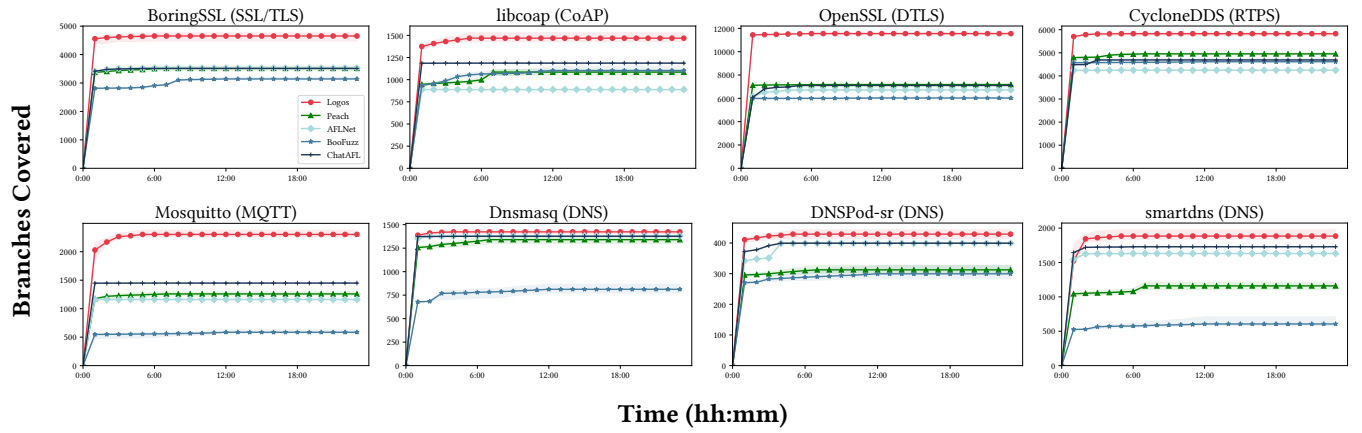
Figure 7: The number of unique branches covered by different fuzzers on each protocol server over ten 24-hour runs.
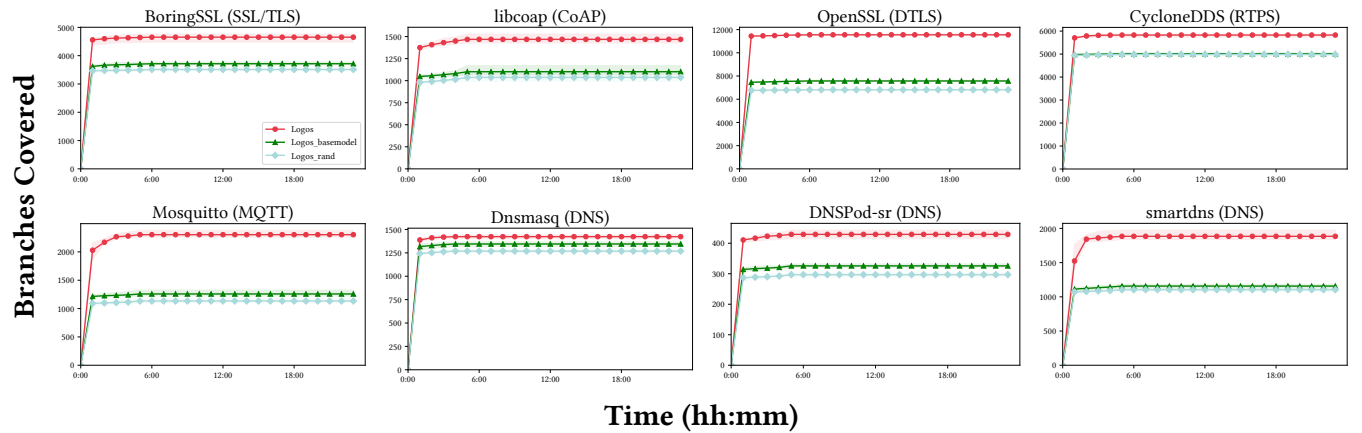
Figure 8: Ablation experiment for Logos, the number of unique branches covered by different variants of Logos on each protocol implementation over five 24-hour runs.

new semantic sequences logs that generate novel log coverage, even without increasing the overall code coverage. This approach allows the guided fuzzing module to pay more attention to the generation of such message types, improving the ability to detect complex logical errors that other tools may miss.

Answer to RQ1. Overall, Logos can achieve higher coverage and discover more vulnerabilities than existing state-of-the-art

protocol fuzzers, which means that Logos can perform well even under non-intrusive conditions.

4.4 Ablation Studies

We use ablation experiments to investigate the contribution of individual components to the effectiveness of Logos. Specifically, we will use a version of Logos_rand that is generated without Logos by bootstrapping randomized packets, and a version of Logos_basemodel that has only pre-trained models but has not done any comparative learning as comparisons for the ablation experiments. We will illustrate the contribution of the different components of Logos to the overall validity from the following two perspectives: (i) the validity of the most central step of the log modeling module, contrast learning and model inference, is tested by ablating the experimental observations of the coverage performance of both Logos and Logos_basemodel; and (ii) the validity of guided fuzzing, is demonstrated by observing Logos_basemodel and Logos_rand comparisons, as well as the relationship between coverage growth and rating growth to demonstrate the effectiveness of semantic representation estimation. It is the most central step of guided fuzzing.

Figure 8 presents the outcomes of the ablation experiments. Logos always outperforms Logos_basemodel and Logos_rand in the above experiments, and even though Logos_basemodel has already improved over Logos_rand, the average improvement brought about by the introduction of contrastive learning can also reach as high as 44.17%, and for the item with the highest improvement, the introduction of contrastive learning brought about a 92.7% improvement. Such a fact fully proves the effectiveness of the contrastive learning step. For the dnsmasq project, it has been observed that the results of various versions of Logos are closely aligned. Similarly, it is noted that the results of different tools depicted in Figure 7 also show comparable closeness. This phenomenon suggests a tendency towards coverage saturation within the dnsmasq project. Consequently, it is concluded that coverage alone may not effectively reflect the testing effectiveness of tools in this project, especially given that only Logos has identified the CVE in the dnsmasq project.

To examine the relationship between coverage and ratings, their association was quantified using the Pearson correlation coefficient. The results showed a positive correlation between coverage and ratings ($p<0.01$), indicating a significant association. This correlation suggests that guidance based on ratings can improve coverage, similar to the improvements with coverage-based guidance.

Answer to RQ2. The ablation studies and the relationship between rating and coverage growth demonstrate that the proposed log modeling significantly contributes to the overall performance. The positive correlation between rating and coverage also suggests that modeling results can effectively guide fuzzing.

5 Discussion

Log Accessibility. The effectiveness of Logos depends on the availability of log information. Since the protocol implementations are generally employed in critical areas and require rigorous testing, they are often equipped with log information, which is essential for debugging and maintenance. Therefore, our method is applicable to most protocol implementations. Besides, the log information can be obtained from various sources, such as the console, log files, serial ports, or network interfaces. Even for embedded systems such as IoT devices, developers often provide the way to access logs in the official documentation. For example, for the Azure IoT devices, the logs can be obtained through the *logtrace* option provided by the Azure IoT Hub client [37].

Log Quality. The quality of the log information is also a critical factor in our method. In the case of low coverage of log statements, we try to compensate for the lack of code coverage approximation by exploring the semantic diversity of logs. Even if several sibling branches lack log coverage, it is feasible to explore more sibling branches if specific events, operations, or exceptions in the previous branches can shed light on the branch's transition logic. For the case where logs lack semantics, we use code coverage approximation as a basic form of log semantics, and as long as there is no shortage of log statements in the project, even lower-quality logs can improve the code coverage approximation. For the case of simultaneous low log statement coverage and lack of semantics, based on Figure 8, we can still improve results beyond the random approach. For cases where logs are unavailable or extremely inefficient, we can use system call trace tools (e.g. strace[28]) to get logs of system calls, or corresponding syslog/device logs as alternatives. This can be discussed as future work in the revision.

Overhead. Traditional intrusive methods(e.g. instrumentation-based approaches) impose performance overhead on the SUT. In contrast, Logos' non-intrusive approach imposes no overhead on the SUT itself. Instead, we use log analysis to replace the instrumentation to obtain runtime execution information.

Table 4 count the execution time ratio of the log modeling component as follows. As shown in the above experiments on fuzzing rounds, even though log modeling introduces some execution time, it does not affect the fuzzing throughput because log model analysis shares the time waiting for the server's response.

Table 4: Execution time ratio of the log modeling component

	BoringSSL	libcoap	OpenSSL	CycloneDDS
$\mathbf{T_{exec}}/\mathbf{T_{total}}$	19.7%	2.2%	2.1%	3.4%

	Mosquitto	Dnsmasq	DNSPod-sr	smartdns
$\mathbf{T_{exec}}/\mathbf{T_{total}}$	2.3%	2.2%	2.2%	2.2%

Dataset Impact. Table 5 show the impact of the dataset on the the model, we constructed datasets of 0/10/100/300(The size of 300 is close to the experimental scale) logs. We use these datasets to train our semantic model and show the average code coverage that Logos can be achieved in the 24-hour fuzzing campaign as follows.

As the dataset size becomes larger, the coverage of each under-test target increases accordingly, which means that the accuracy of the semantic representation improves. For most targets, the coverage improvement from 0 to 100 is greater than that from 100 to 300, which means that the effect of data size on the model gradually stabilizes after reaching a certain size.

Table 5: Dataset size impact to the semantic model

	BoringSSL	libcoap	OpenSSL	CycloneDDS
0	3605	1067	1298	307
10	3655	1087	1308	315
100	4272	1363	1318	397
300	4481	1481	1387	414

	Mosquitto	Dnsmasq	DNSPod-sr	smartdns
0	7188	1228	4791	1091
10	7307	1286	4834	1110
100	11071	2213	5504	1776
300	11120	2226	5672	1856

6 Related Work

Protocol Fuzzing. Numerous studies demonstrate the extensive use of fuzzing in protocol implementation testing [18, 25, 33, 34, 38, 45, 53, 55, 57, 58]. However, traditional protocol fuzzing methods ignore real-time feedback such as coverage data, packet details, and log insights, relying instead on rule-based packet delivery. Tools such as Peach and BooFuzz require up-front state modeling of protocol packets, which requires expert knowledge or protocol reverse techniques such as DynPRE[32] to model the protocol details. The most common approach to incorporating feedback into fuzzing is to use coverage feedback to guide packet generation, using methods such as source code instrumentation [35, 45], dynamic instrumentation via QEMU [15], and compile-time coverage feedback via Intel PT [48]. Despite the effectiveness of coverage feedback in reflecting code state, it introduces intrusiveness (in the case of source code instrumentation) or significant overhead (for dynamic instrumentation with QEMU), resulting in potential performance degradation of the SUT and additional recompilation and customization costs. An alternative method of utilizing feedback is to introduce real-time state information. For instance, AFLNet allows to employ manually crafted state regions to explore the state space [45], while SGFuzz[6] guides fuzzing through program state information obtained through source code analysis, and StateAFL[38] identifies SUT states through runtime memory conditions obtained through source code instrumentation. Although state-aware approaches can accurately capture state information, they often require expert knowledge or source code. Integrating log feedback, Logos effectively captures real-time information during protocol fuzzing and removes the necessity for modifications to the SUT or expert modeling knowledge. This non-intrusive feature allows Logos to easily adapt and test the SUT, highlighting the benefits of using real-time feedback in protocol fuzzing without the drawbacks of other feedback mechanisms.

While there has been significant advancement in techniques that guide the fuzzing process [11, 24, 41, 50, 52, 54, 56], many of these methods recourse to potentially invasive information, such as memory accessing and instrumentation, which can can brings challenges to seed scheduling. To address these complexities, [9] introduced multiple fuzz queues to handle the different packet formats at different stages. AFLGo [7] uses a simulated annealing algorithm to optimize seed scheduling dynamically, improving seed

selection and prioritization through this heuristic approach and allowing targeted exploration of specific program segments. [30] optimizes the scheduler's decisions by reducing excessive testing of specific seeds. Building on these developments, Logos has developed a packet pool structure tailored for protocol fuzzing, as shown in Figure 6. This structure not only allows for effective unified testing of packet sequences but also incorporates unstructured log information into the generation process on a per-packet basis, demonstrating a strategic approach to overcoming the challenges associated with seed scheduling and information invasiveness.

Log Based Testing. The log information utilization has become a common practice in software testing, and numerous studies employ log data to identify bugs in the SUT. For example, research [3, 4] has demonstrated the effectiveness of analyzing event logs to check for errors indicated by program execution. While such analyses are adept at detecting vulnerabilities, they focus mainly on the passive detection of errors in running programs. A more proactive approach is presented in [43], which integrates logs with fuzzing to determine the position of the fuzz layer within the system. This integration allows for an active form of fuzzing, although it still relies on random generation with limited information. As an evolution of this methodology, [2] extracts log information to guide the generation of appropriate payloads, significantly improving the accuracy of fuzzing. This approach requires the manual design of constraint classifications and imposes stringent requirements on the log format. While these studies highlight the critical role of log information, they also expose the invasive and labor-intensive aspects of its application in fuzzing, reflecting a lack of scalability. GLeeFuzz[42] extracts specific types of exception information from specific types of logs (WebGL), which is difficult to use with heterogeneous unstructured logs for protocol testing. Logos, in contrast, proposes a unified model for log information that extracts feedback to guide fuzzing. This unified modeling approach not only reduces dependency on specific formats but also captures rich feedback from the SUT [23]. By leveraging log data at minimal cost, Logos facilitates effective testing and demonstrates a more scalable and less expert-intensive methodology for integrating fuzzing with log.

7 Conclusion

In this paper, we introduce Logos, a log-guided protocol fuzzer that uses unified log modeling and guidance to detect vulnerabilities in protocol implementations. It uses semantic information from unstructured logs to dynamically maintain semantic coverage and applies the proposed packet generation algorithm to efficiently generate packets with the concerning of log diversity. Compared to state-of-the-art protocol fuzzers, Logos achieves higher coverage and detects more bugs in real-world protocol implementations. Logos is a non-invasive protocol fuzzer that requires no expert knowledge or instrumentation.

Acknowledgment

References

[1] RFC 8323. 2018. CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets. *https://datatracker.ietf.org/doc/html/rfc8323* (2018).

[2] Yousra Aafer, Wei You, Yi Sun, Yu Shi, Xiangyu Zhang, and Heng Yin. 2021. Android {SmartTVs} vulnerability discovery via {log-guided} fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*. 2759–2776.

[3] J.H. Andrews. 1998. Testing using log file analysis: tools, methods, and issues. In *Proceedings 13th IEEE International Conference on Automated Software Engineering (Cat. No.98EX239)*. 157–166. *https://doi.org/10.1109/ASE.1998.732614*

[4] J.H. Andrews and Yingjun Zhang. 2003. General test result checking with log file analysis. *IEEE Transactions on Software Engineering* 29, 7 (2003), 634–648. *https://doi.org/10.1109/TSE.2003.1214327*

[5] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250.

[6] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful Greybox Fuzzing. *2022 Usenix Security Symposium* (2022).

[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. *https://doi.org/10.1145/3133956.3134020*

[8] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*. PMLR, 1597–1607.

[9] Yurong Chen, Tian lan, and Guru Venkataramani. 2019. Exploring Effective Fuzzing Strategies to Analyze Communication Protocols. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation* (London, United Kingdom) *(FEAST'19)*. Association for Computing Machinery, New York, NY, USA, 17–23. *https://doi.org/10.1145/3338502.3359762*

[10] Clang. 2024. Clang 19.0.0 documentation, UndefinedBehaviorSanitizer. *https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html* (2024).

[11] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. 2019. MemFuzz: Using Memory Accesses to Guide Fuzzing. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 48–58. *https://doi.org/10.1109/ICST.2019.00015*

[12] CVE-2014-0160. 2014. Heartbleed - A vulnerability in OpenSSL. *http://heartbleed.com* (2014).

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. *https://doi.org/10.18653/V1/N19-1423*

[14] DNSPod. 2024. A faster recursive dns server from DNSPod. *https://github.com/DNSPod/dnspod-sr* (2024).

[15] Pavel Dovgalyuk, Natalia Fursova, Ivan Vasiliev, and Vladimir Makarov. 2017. QEMU-based framework for non-intrusive virtual machine instrumentation and introspection. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 944–948.

[16] Michael Eddington. 2024. GitLab Protocol Fuzzer Community Edition. *https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce* (2024).

[17] facebookresearch. 2024. Faiss: A library for efficient similarity search and clustering of dense vectors. *https://github.com/facebookresearch/faiss* (2024).

[18] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minghui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. 2021. Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference. *2021 ACM SIGSAC CCS* (2021).

[19] Google. 2015. american fuzzy lop - a security-oriented fuzzer. *https://github.com/google/AFL* (2015).

[20] Google. 2021. OSS-Fuzz/Gnutls. *https://github.com/google/oss-fuzz/tree/master/projects/dnsmasq*. (2021).

[21] Google. 2021. OSS-Fuzz/Libressl. *https://github.com/google/oss-fuzz/tree/master/projects/dnsmasq*. (2021).

[22] Google. 2024. OSS-Fuzz - continuous fuzzing for open source software. *https://github.com/google/oss-fuzz* (2024).

[23] Tuomas Haanpää. 2016. *Fuzz testing coverage measurement based on error log analysis*. Master's thesis. T. Haanpää.

[24] Adrian Herrera, Mathias Payer, and Antony L. Hosking. 2023. DatAFLow: Toward a Data-Flow-Guided Fuzzer. *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 132 (jul 2023), 31 pages. *https://doi.org/10.1145/3587156*

[25] Shihao Jiang, Yu Zhang, Junqiang Li, Hongfang Yu, Long Luo, and Gang Sun. 2024. A Survey of Network Protocol Fuzzing: Model, Techniques and Directions. arXiv:2402.17394 [cs.NI]

[26] jtpereyda. 2024. boofuzz: Network Protocol Fuzzing for Humans. *https://github.com/jtpereyda/boofuzz* (2024).

[27] Michael Kerrisk. 2024. ptrace(2) — Linux manual page. *https://man7.org/linux/man-pages/man2/ptrace.2.html* (2024).

[28] Michael Kerrisk. 2024. strace(1) — Linux manual page. *https://man7.org/linux/man-pages/man1/strace.1.html* (2024).

[29] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. *https://doi.org/10.1145/3243734.3243804*

[30] Shaohua Li and Zhendong Su. 2023. Accelerating Fuzzing through Prefix-Guided Execution. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 75 (apr 2023), 27 pages. *https://doi.org/10.1145/3586027*

[31] LLVM. 2024. SanitizerCoverage. *https://clang.llvm.org/docs/SanitizerCoverage.html* (2024).

[32] Zhengxiong Luo, Kai Liang, Zhao Yanyang, Feifan Wu, Junze Yu, Heyuan Shi, and Yu Jiang. 2024. DynPRE: Protocol Reverse Engineering via Dynamic Inference. *https://doi.org/10.14722/ndss.2024.24083*

[33] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. 2023. BLEEM: packet sequence oriented fuzzing for protocol implementations. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4481–4498.

[34] Zhengxiong Luo, Feilong Zuo, Yu Jiang, Jian Gao, Xun Jiao, and Jiaguang Sun. 2019. Polar: Function Code Aware Fuzz Testing of ICS Protocol. *ACM Trans. Embed. Comput. Syst.* 18 (2019), 93:1–93:22.

[35] Zhengxiong Luo, Feilong Zuo, Yuheng Shen, Xun Jiao, Wanli Chang, and Yu Jiang. 2020. ICS Protocol Fuzzing: Coverage Guided Packet Crack and Generation. *ACM/IEEE Design Automation Conference (DAC)* (2020).

[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] Microsoft. 2023. How to collect debug logs from your Azure IoT devices. *https://learn.microsoft.com/en-us/azure/iot-hub/how-to-collect-device-logs?pivots=programming-language-ansi-c* (2023).

[38] Roberto Natella. 2021. StateAFL: Greybox Fuzzing for Stateful Network Servers. *ArXiv* abs/2110.06253 (2021).

[39] obgm. 2024. A CoAP (RFC 7252) implementation in C. *https://github.com/obgm/libcoap* (2024).

[40] OpenSSL. [n. d.]. OpenSSL. Website. *https://github.com/openssl/openssl*.

[41] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2289–2306. *https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund*

[42] Hui Peng, Zhihao Yao, Ardalan Amiri Sani, Dave (Jing) Tian, and Mathias Payer. 2023. GLeeFuzz: Fuzzing WebGL Through Error Message Guided Mutation. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 1883–1899. *https://www.usenix.org/conference/usenixsecurity23/presentation/peng*

[43] Yifei Peng, Xinyi Li, Sudhanshu Arya, and Ying Wang. 2023. DEFT: A Novel Deep Framework for Fuzz Testing Performance Evaluation in NextG Vulnerability Detection. *IEEE Access* 11 (2023), 116046–116064. *https://doi.org/10.1109/ACCESS.2023.3326411*

[44] Thuan Pham. 2024. AFLNet: A Greybox Fuzzer for Network Protocols. *https://github.com/aflnet/aflnet* (2024).

[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] pymumu. 2024. A local DNS server to obtain the fastest website IP for the best Internet experience, support DoT, DoH. *https://github.com/pymumu/smartdns* (2024).

[47] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. *https://arxiv.org/abs/1908.10084*

[48] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Reza Abbasi, and Thorsten Holz. 2022. Nyx-net: network fuzzing with incremental snapshots. *Seventeenth European Conference on Computer Systems* (2022).

[49] Kostya Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. *2012 USENIX Annual Technical Conference* (2012).

[50] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 344–358. *https://doi.org/10.1145/3477132.3483547*

[51] thekelleys. 2024. Dnsmasq - network services for small networks. *https://thekelleys.org.uk/dnsmasq/doc.html* (2024).

[52] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MemLock: memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 765–777. *https://doi.org/10.1145/*

3377811.3380396

[53] Junze Yu, Zhengxiong Luo, Fangshangyuan Xia, Yanyang Zhao, Heyuan Shi, and Yu Jiang. 2024. SPFuzz: Stateful Path based Parallel Fuzzing for Protocols in Autonomous Vehicles. *ACM/IEEE Design Automation Conference (DAC)* (2024).

[54] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu, and Erxue Min. 2018. PTfuzz: Guided Fuzzing With Processor Trace Feedback. *IEEE Access* 6 (2018), 37302–37313. https://doi.org/10.1109/ACCESS.2018.2851237

[55] Xiaohan Zhang, Cen Zhang, Xinghua Li, Zhengjie Du, Yuekang Li, Yaowen Zheng, Yeting Li, Bing Mao, Yang Liu, and Robert H. Deng. 2024. A Survey of Protocol Fuzzing. arXiv:2401.01568 [cs.CR]

[56] Xiaogang Zhu, Shigang Liu, Xian Li, Sheng Wen, Jun Zhang, Seyit Ahmet Çamtepe, and Yang Xiang. 2020. DeFuzz: Deep Learning Guided Directed Fuzzing. *CoRR* abs/2010.12149 (2020). arXiv:2010.12149 https://arxiv.org/abs/2010.12149

[57] Feilong Zuo, Zhengxiong Luo, Junze Yu, Ting Chen, Zichen Xu, Aiguo Cui, and Yu Jiang. 2022. Vulnerability Detection of ICS Protocols via Cross-State Fuzzing. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* (2022).

[58] Feilong Zuo, Zhengxiong Luo, Junze Yu, Zhe Liu, and Yu Jiang. 2021. PAVFuzz: State-Sensitive Fuzz Testing of Protocols in Autonomous Vehicles. *ACM/IEEE Design Automation Conference (DAC)* (2021).