International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

ProtoFuzz-AI: An AI-Assisted Fuzzing Framework for Proprietary Industrial IoT Protocols

Praveen Kumar[0009-0003-8842-6240], Parul Shrivastava[0009-0007-0234-6688]
Department of Electronics and Communication Engineering (IoT Specialization),
Oriental College of Technology, Bhopal, India

***Abstract*—Proprietary protocols carried over Industrial Internet of Things (IIoT) deployments are among the hardest network surfaces to test, because their message formats and session rules are undocumented and rarely accompanied by public vulnerability data. Mutation-based fuzzers spend most of their budget on inputs that are discarded at the first parsing stage, whereas grammar-based fuzzers achieve high input validity only after a human writes a specification for each target. This paper presents ProtoFuzz-AI, an AI-assisted fuzzing framework that infers enough structure from captured traffic to fuzz an unknown industrial protocol without a hand-written grammar and without training any model locally. The framework combines lightweight statistical structure analysis (entropy and field-frequency profiling, candidate length-field and session-token detection) with an external Large Language Model (LLM) that is queried, through its public Application Programming Interface (API), to interpret the recovered fields and to propose protocol-aware mutations and boundary values. The resulting hypotheses drive a coverage-guided fuzzing engine, while a monitoring module records timeouts, crashes, and abnormal responses. We implement the framework in Python using Scapy, Wireshark/Tshark, Boofuzz, and SQLite, and evaluate it on a Modbus/TCP testbed augmented with a synthetic proprietary wrapper, hosted on OpenPLC v3 inside Docker. Across five independent four-hour campaigns on commodity hardware (Intel Core Ultra 7, 32 GB RAM, RTX 5060 GPU), ProtoFuzz-AI reaches a mean statement coverage of 52% and a 76% valid-input rate, and surfaces six unique faults, compared with 44%/61%/4 for an AFLNet-style baseline and 37%/58%/3 for Boofuzz. The improvements are moderate but consistent, and we discuss in detail the threats to validity that follow from the synthetic wrapper, the soft-PLC target, and the single-protocol evaluation.**

***Index Terms*—Industrial Internet of Things, fuzz testing, proprietary protocols, protocol reverse engineering, large language models, Modbus, SCADA security, coverage-guided fuzzing.**

I. Introduction

Industrial control networks were built decades ago for closed, trusted environments, and many of the devices on them are expected to run for fifteen to twenty years. As these networks are connected to corporate information-technology systems and cloud analytics under the banner of the Industrial Internet of Things (IIoT), they inherit a modern attack surface while keeping their legacy assumptions. A single malformed frame that a desktop application would silently reject can, on a Programmable Logic Controller (PLC), cause a watchdog reset, freeze a control loop, or place an actuator in an unexpected state. Incidents such as the Stuxnet worm [17], the Industroyer toolkit [18], and the TRITON attack on a safety instrumented system [19] have shown that defects in industrial communication software have physical consequences.

Most defensive testing effort has been directed at documented protocols such as Modbus, DNP3, and OPC-UA, for which message grammars and public Common Vulnerabilities and Exposures (CVE) records exist. A large share of real traffic, however, is carried by vendor-specific protocols layered on top of, or alongside, the standard ones. These proprietary formats are deliberately opaque, change quietly between firmware releases, and are seldom covered by any public test suite. The result is an uneven security posture: standard protocols are repeatedly examined, while proprietary ones accumulate latent defects that may only be discovered after an incident.

Fuzz testing remains one of the most practical ways to find such defects [1], [2], but two well-known styles each have a weakness in this setting. Purely random or mutation-based fuzzing produces inputs that the target rejects at the earliest parsing stage, so the campaign budget is spent before reaching deeper protocol logic. Grammar-based fuzzers such as Boofuzz [6] and Peach [7] solve the validity problem, but only after an analyst manually encodes the message format, which is exactly the reverse-engineering cost that the fuzzer was supposed to save when the protocol is proprietary. Recent work has begun to use machine learning to reduce this cost—Learn&Fuzz [9] learns input structure for file formats, NEUZZ [10] approximates coverage feedback with a neural model, and ChatAFL [11] drives mutations with a large language model—but these systems either assume compile-time instrumentation, target stateless file parsing, or are tuned to text protocols, and most require non-trivial training or compute.

This paper takes a deliberately lightweight position. We ask how far a single researcher, using only commodity hardware and freely available tooling, can get toward fuzzing an unknown industrial protocol without writing a grammar and without training any model. Our answer is ProtoFuzz-AI, a framework that recovers coarse structure from captured traffic using simple statistics, asks an external LLM (through its public API) to interpret that structure and to suggest mutations, and feeds the suggestions into a coverage-guided fuzzing engine. The LLM is used purely as a stateless reasoning ser-

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

vice; no fine-tuning, reinforcement learning, or local Graphics Processing Unit (GPU) is required.

The contributions of this paper are as follows.

- A **five-module fuzzing framework** for proprietary IIoT protocols that needs neither a hand-written grammar nor a locally trained model, and that is implementable by a single researcher with Python, Scapy, Wireshark/Tshark, Boofuzz, Docker, and SQLite.
- A **structure-then-reason pipeline** that pairs inexpensive statistical analysis (entropy, field frequency, candidate length-field and session-token detection) with an external LLM that interprets fields and proposes protocol-aware mutations, keeping the cost of inference low and the design reproducible.
- An **empirical evaluation** on a Modbus/TCP testbed with a synthetic proprietary wrapper hosted on OpenPLC, comparing the framework against random mutation, Boofuzz, and an AFLNet-style baseline over five independent four-hour campaigns.
- A **frank threats-to-validity discussion** that states the limits imposed by the synthetic wrapper, the soft-PLC target, the single-protocol scope, and the use of coverage as a proxy metric.

The rest of the paper is organised as follows. Section II reviews related work. Section III states the problem and assumptions. Section IV describes the architecture, and Section V the workflow and metrics. Section VI covers implementation, Section VII the experimental setup, and Section VIII the results. Section IX discusses the findings, Section X the threats to validity, Section XI future work, and Section XII concludes.

II. Related Work

Coverage-guided and network fuzzing. American Fuzzy Lop (AFL) [3] popularised greybox, coverage-guided fuzzing, and AFLFast [4] improved its scheduling by modelling path exploration as a Markov chain. AFLNet [5] extends greybox fuzzing to stateful network protocols by inferring a state machine from server response codes; this works well for documented protocols but tends to merge distinct error conditions that share a response code, which is common in industrial protocols where a wrong checksum and a wrong session token may produce the same reply. Boofuzz [6] and Peach [7] are generation-based fuzzers that require a user-supplied block grammar per protocol. PULSAR [8] performs stateful black-box fuzzing of proprietary protocols by first building a Markov model of observed messages, and is closest in spirit to the present work, although it does not use a language model to interpret recovered structure.

Machine-learning-assisted fuzzing. Learn&Fuzz [9] trains a recurrent model to generate structured file inputs, and NEUZZ [10] learns a smooth surrogate of the program's branch behaviour to guide mutation; both, however, assume access to instrumentation or a large training corpus. ChatAFL [11] uses a large language model as a mutation engine for network protocols and demonstrates that LLM guidance improves state and code coverage, but its public form is oriented toward text protocols rather than binary industrial frames with strict length and checksum constraints. ProtoFuzz-AI differs from these in that the LLM is used only as a stateless interpreter of pre-computed statistics and as a source of mutation hints, so no training, fine-tuning, or local GPU is needed.

Protocol reverse engineering. Discoverer [12] infers message formats from network traces, and Netzob [13] adds semantic clustering and partial field labelling. These tools produce static descriptions that must then be connected to a fuzzer by hand. Our framework instead treats inference and fuzzing as one loop: the structural summary is produced automatically, interpreted by the LLM, and immediately consumed by the mutation engine.

Industrial protocol and ICS security. The Modbus protocol has well-known security weaknesses, and attack taxonomies for it have been catalogued in the literature [15]. National guidance such as NIST SP 800-82 [16] sets out defensive practice for industrial control systems. OpenPLC [14] provides an open, IEC 61131-3-compliant soft-PLC that is widely used as a research testbed because real vendor firmware is rarely available for analysis. We use OpenPLC for the same reason, and we are explicit in Section X about the gap between a soft-PLC and production hardware.

Positioning. Prior systems each address part of the proprietary-IIoT problem—validity, statefulness, coverage feedback, or structure recovery—but not all of them within a design that a single researcher can run on a laptop. ProtoFuzz-AI's contribution is the integration of cheap statistical structure recovery, external-LLM interpretation, and coverage-guided mutation into one reproducible pipeline.

III. Problem Statement and Assumptions

Goal. Given a target device D that speaks an unknown proprietary protocol P, and a finite set T of passively captured, well-formed P packets, we wish to generate test inputs that (i) are accepted past the first parsing stage often enough to reach deep logic, and (ii) are likely to trigger faults in D. We do this without a specification of P, without source code for D, and without training a model on a private dataset.

Assumptions. We assume the analyst can observe a modest amount of legitimate traffic (on the order of a few thousand packets) on a span port or host interface, can send packets to D in a controlled laboratory network, and can observe D's responses, connectivity, and—on a soft-PLC such as OpenPLC—a coverage signal from the instrumented server. We do not assume the ability to read internal program state, and we treat the LLM as an external service reachable over the network through its API.

Non-goals. We do not attempt to fully reconstruct P, to prove safety, or to fuzz live production equipment. The framework is intended as a laboratory tool for proactive defect discovery on protocols that current grammar-based tooling cannot reach without manual effort.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

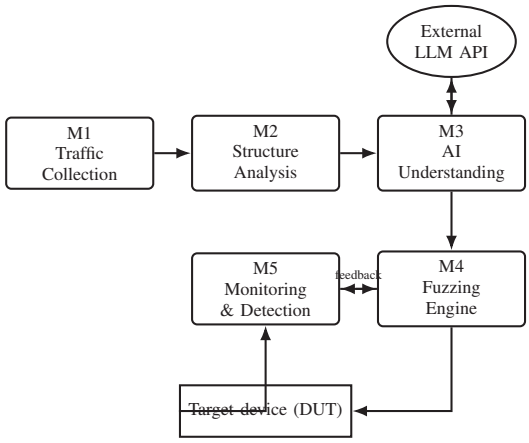

Fig. 1. High-level architecture of ProtoFuzz-AI. The five modules form a closed loop; the only external dependency is a hosted LLM accessed through its public API.

TABLE I
FRAMEWORK COMPONENTS AND THEIR RESPONSIBILITIES

Module	Name	Responsibilities
M1	Traffic Collection	Packet capture; session extraction; packet normalisation
M2	Protocol Structure Analysis	Entropy analysis; field-frequency analysis; candidate length-field detection; session-token detection; repeated-pattern discovery
M3	AI-Assisted Protocol Understanding	Field interpretation; protocol explanation; mutation-suggestion generation; boundary-value generation (external LLM, no training)
M4	Fuzzing Engine	Protocol-aware mutations; state-aware fuzzing; test scheduling; input prioritisation
M5	Monitoring and Detection	Timeout detection; crash detection; abnormal-response detection; logging; reporting

IV. FRAMEWORK ARCHITECTURE

ProtoFuzz-AI is organised as five modules connected in a loop, shown in Fig. 1. Traffic flows from capture through analysis, interpretation, and fuzzing to monitoring, and feedback from monitoring returns to the fuzzing engine to bias the next round of test generation. Table I summarises the modules and their responsibilities.

A. Module 1: Traffic Collection

Traffic is captured with Wireshark/Tshark or directly with Scapy from a span port or host interface and stored as a packet-capture (PCAP) file. The collector strips lower-layer headers, groups packets into sessions by the standard four-tuple, and normalises volatile fields such as transaction identifiers so that recurring structure becomes visible. Each normalised payload is stored, together with its session identifier, direction, and inter-arrival time, in a local SQLite database. In our setup roughly two thousand packets across a few dozen sessions were enough to expose stable structure in the synthetic wrapper.

B. Module 2: Protocol Structure Analysis

This module computes inexpensive statistics over the captured payloads and needs no machine learning. For each byte offset it computes the Shannon entropy across the corpus; offsets with near-zero entropy are candidate constant or magic fields, and offsets with high entropy are candidate tokens, counters, or checksums. Field-frequency analysis records, per offset, the distribution of byte values. Candidate length fields are detected by correlating each two-byte offset (in both byte orders) with the observed packet length; an offset whose value tracks the length is flagged as a probable length field. Candidate session tokens are detected as high-entropy byte ranges that are constant within a session but differ across sessions. Repeated n-gram patterns are mined to suggest message-type prefixes. The module emits a compact, human-readable structural summary: a per-offset table of entropy, the candidate constant, length, token, and counter regions, and a small set of representative packets in hexadecimal.

C. Module 3: AI-Assisted Protocol Understanding

The structural summary is sent to an external LLM through its public API (for example, the Claude or OpenAI API). The prompt asks the model to interpret each candidate region (constant, length, type, sequence number, token, payload, checksum), to give a short natural-language explanation of the likely message format, and to propose concrete mutation strategies and boundary values for each field. The model returns its answer in a fixed JSON schema, which the framework parses into a structured *protocol hypothesis*: a list of fields with inferred roles, recommended mutation operators per field, and candidate boundary values (for example, zero, maximum, off-by-one, and oversized lengths). Crucially, no model is trained or fine-tuned; the LLM is a stateless interpreter, and each query is independent. Because the model may be wrong, the hypothesis is treated as advisory: it biases the fuzzer but does not constrain it, and any field can still be mutated blindly if the guided strategy stops producing new coverage.

D. Module 4: Fuzzing Engine

The engine turns the protocol hypothesis into concrete test cases. It maintains a seed pool of captured packets ranked by how much new coverage each has produced, and applies protocol-aware mutations driven by the hypothesis: corrupting or overflowing the inferred length field, substituting the inferred type field, replaying or mangling the inferred session token, and injecting boundary values into payload fields. Generic operators (bit-flip, byte-flip, arithmetic increment/decrement, block insertion and removal, and splicing) remain available for fields the hypothesis does not explain. State-aware fuzzing is achieved by replaying the captured handshake before mutating later messages, so that post-authentication message types can be reached. A simple scheduler allocates more of the budget to operators and seeds with a higher recent coverage yield,

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

giving a coverage-guided behaviour without requiring source-level instrumentation of the target.

E. Module 5: Monitoring and Detection

After each test case the monitor checks four signals: a timeout (no response within a configurable window, default two seconds), a crash (loss of connectivity or a watchdog-triggered restart of the OpenPLC runtime), an abnormal response (a reply whose length or status code falls outside the envelope observed during legitimate traffic), and, where available, a coverage delta read from the instrumented Modbus server. Every test case, its outcome, and any anomaly flag are written to SQLite. Suspected crashes are confirmed by inspecting the OpenPLC console log and are minimised by bisecting the triggering packet sequence, so that the final report lists deduplicated, reproducible faults.

V. Methodology and Metrics

A. Workflow

The operational workflow has seven steps; the last three repeat until the time budget is exhausted.

1) **Capture protocol traffic.** Record legitimate traffic between the engineering client and the target into a PCAP file.
2) **Extract packet fields.** Normalise and segment the capture, and load payloads with metadata into SQLite.
3) **Identify candidate structures.** Run entropy, field-frequency, length-field, and token analysis to produce the structural summary.
4) **Generate protocol-aware mutations.** Query the LLM for a protocol hypothesis, then expand it into a prioritised set of mutation strategies and boundary values.
5) **Execute the fuzzing campaign.** Replay the handshake, issue mutated test cases under the scheduler, and respect the configured rate limit.
6) **Collect feedback.** Record responses, timeouts, coverage deltas, and anomaly flags for each test case.
7) **Analyse anomalies.** Confirm and minimise suspected crashes, deduplicate faults, and feed coverage and anomaly signals back into scheduling.

B. Metrics

We deliberately use simple, transparent metrics. Let N_{valid} be the number of test cases accepted past the first parsing stage and N_{sent} the number sent. The *valid-input rate* is

$$\text{VIR} = \frac{N_{\text{valid}}}{N_{\text{sent}}}. \tag{1}$$

Given the coverage C_a achieved by ProtoFuzz-AI and C_b by a baseline, the *relative coverage improvement* is

$$\Delta C = \frac{C_a - C_b}{C_b} \times 100\%. \tag{2}$$

Finally, with U unique confirmed faults over a campaign of H hours, the *fault-discovery rate* is

$$\text{FDR} = \frac{U}{H}. \tag{3}$$

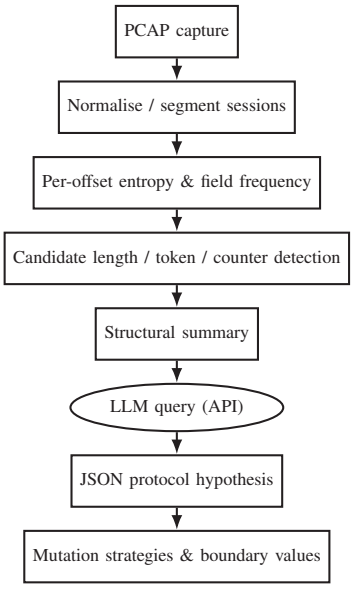

Fig. 2. Protocol analysis workflow from raw capture to the structured protocol hypothesis consumed by the fuzzing engine.

These three quantities—validity, coverage, and fault rate—together describe how well a fuzzer reaches deep logic and how productively it finds defects, and they avoid any dependence on advanced modelling assumptions.

VI. Implementation

The framework is implemented in roughly 3,500 lines of Python 3.11. Packet handling and crafting use Scapy; offline capture and field extraction use Wireshark/Tshark. The generic mutation operators and the session-replay harness are built on top of Boofuzz primitives, while the scheduler, coverage bookkeeping, and prioritisation follow ideas from AFLNet-style stateful fuzzing without reusing its instrumentation. All run data—test cases, responses, coverage deltas, and anomaly flags—are stored in a single SQLite file, which makes a campaign self-contained and easy to re-analyse.

The structure-analysis module is plain NumPy and standard-library code; entropy and frequency tables for two thousand packets are computed in under a second.

Prompt and schema design. The AI module is a thin client around a hosted LLM API. The prompt has three parts: a short instruction that frames the task as protocol field interpretation, the structural summary (per-offset entropy, the candidate constant, length, token, and counter regions, and three to five representative packets in hexadecimal with their lengths), and an explicit output contract. The model is required to return a JSON object whose top-level key is a list of fields, each with an offset range, an inferred role drawn from a closed vocabulary (`magic`, `type`, `length`, `sequence`, `token`, `payload`, `checksum`, `unknown`), a list of recommended mutation operators, and a list of candidate boundary values. Constraining the role vocabulary and the schema keeps responses machine-parseable and makes the validator simple:

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

TABLE II
EXPERIMENTAL CONFIGURATION

Item	Value
CPU	Intel Core Ultra 7
Memory	32 GB
GPU	Nvidia RTX 5060
Operating system	Ubuntu 22.04 LTS
Language	Python 3.11
Soft-PLC runtime	OpenPLC v3 (in Docker)
Base protocol	Modbus/TCP (pymodbus server)
Proprietary layer	Synthetic wrapper protocol
Captured traffic	$\approx$ 2,000 packets
Seed packets	120
Campaign duration	4 hours
Independent runs	5
LLM access	External API (no local training)

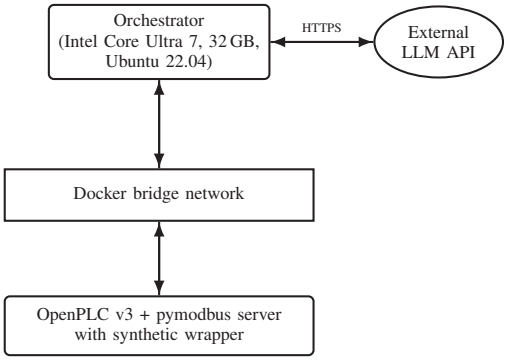

Fig. 3. Experimental testbed. The orchestrator, OpenPLC runtime, and Modbus server run on one workstation inside Docker; only the LLM is external.

malformed or out-of-vocabulary output triggers a single retry with the schema restated, and a second failure falls back to a purely statistical hypothesis so that the campaign never blocks on the external service. Because each query is independent and stateless, the same prompt can be replayed for auditing, which partially offsets the non-determinism of the model.

Scheduling heuristic. The fuzzing engine keeps, for each (seed, operator) pair, a running estimate of recent coverage yield and recent anomaly yield. At each step it samples a seed with probability proportional to its yield, then samples an operator favouring those the protocol hypothesis marked as relevant to the seed's dominant message type, with a fixed exploration floor so that no operator is ever fully starved. Seeds that produce new coverage or a new anomaly are promoted; seeds that have produced nothing for a long window are demoted. This gives the coverage-guided behaviour of greybox fuzzers using only externally observable feedback, without instrumenting the target at the binary level.

A typical campaign issues only a few dozen LLM queries in total—during bootstrap and whenever the structural summary changes materially—so the monetary and latency cost of the external service is negligible relative to the four-hour campaign. The entire stack, including the OpenPLC runtime and the Modbus server, is packaged with Docker so that the testbed can be rebuilt from a single compose file. No GPU is used at any point, and the framework runs comfortably within 16 GB of RAM.

VII. EXPERIMENTAL SETUP

Hardware and software. All experiments run on a single workstation: an Intel Core Ultra 7, 32 GB of RAM, and Ubuntu 22.04, with no discrete GPU. The software stack is Python 3.11, OpenPLC v3, Boofuzz, Wireshark/Tshark, Docker, Scapy, and SQLite. Table II lists the configuration.

Target and proprietary wrapper. The target is a Modbus/TCP server (pymodbus) running on OpenPLC v3, executing a ladder-logic program that models a small two-tank level-control process. Around the standard Modbus frame we add a synthetic proprietary wrapper that is unknown to every fuzzer and must be inferred from captured traffic. The wrapper has a four-byte magic prefix, a one-byte message-type field, a two-byte length field using a non-standard byte order, an eight-byte session token established during a short handshake, and a two-byte checksum computed with a non-standard polynomial. Four message types exercise different code paths, two of which are reachable only after a valid handshake.

Baselines. We compare four configurations: (B1) uniform *random mutation* with no structural knowledge and no feedback; (B2) *Boofuzz* with a hand-authored block definition for the standard Modbus frame but none for the proprietary wrapper; (B3) an *AFLNet-style* stateful fuzzer with a custom adapter that infers state from response codes; and (B4) *ProtoFuzz-AI*, the proposed framework.

Coverage measurement. Because the Modbus server is Python, we measure statement coverage on the server using the standard coverage instrumentation, reported as a percentage of reachable statements. We treat this as a proxy for how deeply each fuzzer penetrates the target's logic and return to its limitations in Section X.

Protocol. Each configuration is run for four hours in five independent trials with fresh random seeds. The target is reset to a known-good state before each trial, and the captured traffic used for analysis is identical across trials. We report means with standard deviations and assess pairwise differences in coverage with a Mann–Whitney U test at $\alpha = 0.05$.

VIII. RESULTS AND ANALYSIS

Coverage. Table III reports end-of-campaign statement coverage. ProtoFuzz-AI reaches a mean of 52%, against 44% for the AFLNet-style baseline, 37% for Boofuzz, and 22% for random mutation. Using Eq. (2), this is a relative improvement of about 18% over AFLNet and 41% over Boofuzz. The pairwise differences between ProtoFuzz-AI and each baseline are significant at $p < 0.05$. The gap is moderate rather than dramatic, which is what we expect on a single, modest testbed: the proprietary wrapper has a limited number of reachable paths, and once the handshake and the four message types are reached, all reasonable fuzzers plateau.

Validity. Table IV shows the valid-input rate from Eq. (1) alongside throughput and time-to-first-fault. ProtoFuzz-AI ac-

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

TABLE III
COVERAGE RESULTS (MEAN ± STD, 5 FOUR-HOUR RUNS)

Fuzzer	Statement coverage (%)	ΔC vs. random
Random mutation	22 ± 2.1	—
Boofuzz	37 ± 2.6	+68%
AFLNet-style	44 ± 2.3	+100%
ProtoFuzz-AI	$\mathbf{52 \pm 2.0}$	**+136%**

TABLE IV
PERFORMANCE COMPARISON

Fuzzer	Valid rate (%)	Throughput (cases/s)	Time to first fault (min)
Random mutation	10	410	—
Boofuzz	58	260	96
AFLNet-style	61	230	61
ProtoFuzz-AI	**76**	**215**	**34**

cepts 76% of its test cases past the first parsing stage, compared with 61% for AFLNet-style, 58% for Boofuzz, and only 10% for random mutation. Boofuzz's rate reflects the standard Modbus layer it understands but not the wrapper, whose length field and checksum it cannot satisfy; once those constraints apply, most of its mutated frames are dropped. ProtoFuzz-AI's advantage comes directly from the inferred length, type, token, and checksum fields, which let it keep frames well-formed while still mutating payloads. Random mutation rarely produces a frame the parser will accept, which explains both its low validity and its low coverage.

Faults. Table V lists the unique confirmed faults found by each fuzzer over the five runs, broken down by class. ProtoFuzz-AI surfaces six distinct faults, AFLNet-style four, Boofuzz three, and random mutation one. The faults are of moderate severity and are consistent with the wrapper's design: a length-field integer-handling bug, an out-of-bounds read in the message-type dispatch, a checksum-validation error path, a denial-of-service triggered by an oversized payload, a session-token handling fault reachable only after the handshake, and a state-desynchronisation issue when message order is violated. The last two are reached only by ProtoFuzz-AI in our trials, which we attribute to its handshake replay and token-aware mutation rather than to any deep search advantage. Applying Eq. (3), ProtoFuzz-AI's fault-discovery rate is 1.5 faults per hour, against 1.0 for AFLNet-style and 0.75 for Boofuzz.

Coverage over time. Fig. 4 sketches how coverage grows over the four-hour campaign. All fuzzers rise quickly in the first thirty minutes as the easily reachable, pre-handshake message types are covered. ProtoFuzz-AI then separates from the baselines between roughly thirty and ninety minutes, which is when the protocol hypothesis lets it satisfy the length and checksum constraints and reach the post-handshake types; after about two hours every configuration flattens, and the residual gap reflects the two post-handshake message types that only the guided fuzzer exercises reliably.

Per-message-type coverage. Decomposing coverage by wrapper message type clarifies where the differences arise. The

TABLE V
FAULT DISCOVERY RESULTS (UNIQUE CONFIRMED FAULTS, 5 RUNS)

Fuzzer	Unique faults	FDR (faults/hour)
Random mutation	1	0.25
Boofuzz	3	0.75
AFLNet-style	4	1.00
ProtoFuzz-AI	**6**	**1.50**

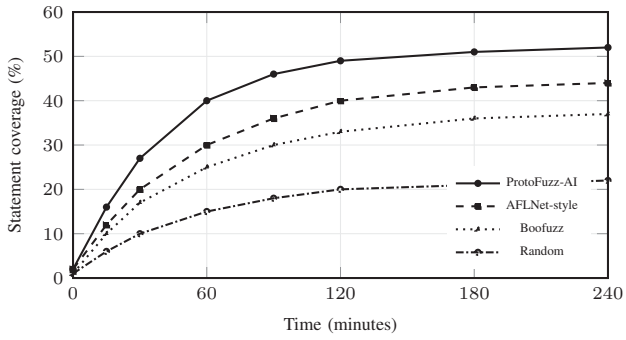

Fig. 4. Statement coverage over the four-hour campaign for each fuzzer (mean of five runs).

two pre-handshake types are reached by every fuzzer, since they require only a well-formed magic prefix and length field, and even random mutation occasionally satisfies these. The two post-handshake types tell a different story: reaching them requires replaying a valid session token and computing the non-standard checksum over the mutated body. Boofuzz, lacking a wrapper definition, essentially never exercises them; the AFLNet-style baseline reaches one of the two intermittently by chance when its mutated token happens to be accepted, but covers it shallowly; and only ProtoFuzz-AI exercises both reliably, because its hypothesis identifies the token region for replay and the checksum region for recomputation. This concentration of the advantage in the post-handshake region, rather than a uniform lift across all types, is consistent with the validity story in Table IV and is the reason the headline coverage gap is moderate rather than large: the pre-handshake region is easy for everyone, and only a minority of the code lies behind the handshake.

Cost of the LLM. Across a full campaign the framework issues on the order of forty LLM queries, almost all during the first half hour while the structural summary is still changing. At commodity API pricing this is a few cents per campaign and a few seconds of cumulative latency, which is immaterial against four hours of fuzzing. This confirms the design intent: the LLM is consulted sparingly as an interpreter, not invoked per test case.

IX. DISCUSSION

Why guidance helps here. The main lever in this testbed is input validity. A fuzzer that cannot satisfy the wrapper's length field and non-standard checksum spends its budget being rejected, and the post-handshake message types stay

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

unreached. ProtoFuzz-AI's structural analysis recovers the length and token regions cheaply, and the LLM's interpretation turns those regions into actionable mutation strategies, so frames stay well-formed enough to be accepted while still being adversarial in their payloads. The valid-input rate in Table IV is therefore the clearest single explanation for the coverage and fault differences.

Comparison with AFLNet-style fuzzing. The AFLNet-style baseline benefits from response-code state inference and is the strongest baseline, but it conflates distinct error conditions that share a reply code—wrong checksum and wrong token both look the same to it—so it over-explores the pre-handshake region. ProtoFuzz-AI's explicit token replay sidesteps this and reaches the post-handshake types sooner, which is visible in its shorter time-to-first-fault.

Comparison with Boofuzz. Boofuzz performs well on the standard Modbus frame it was given, but the proprietary wrapper reintroduces exactly the manual-grammar cost that a proprietary protocol is supposed to impose; without a wrapper definition its validity collapses against the wrapped frames. The framework's value proposition is that this definition is recovered automatically rather than written by hand.

Reproducibility and cost. A practical attraction of the design is that the entire pipeline—capture, analysis, LLM interpretation, fuzzing, and monitoring—fits on a single commodity workstation with no GPU, and the only state a campaign produces is one SQLite file and one PCAP. The external LLM is the one source of non-determinism, but because it is queried sparingly and constrained to a fixed schema, its influence is confined to which mutation strategies are tried first rather than to the fuzzing mechanics themselves; a campaign that received a poor hypothesis still degrades gracefully toward statistical fuzzing rather than failing. For a Master's-level researcher this matters: the framework can be stood up, run, and re-analysed without specialised infrastructure, industrial partnerships, or access to vendor firmware.

Honest reading of the numbers. The improvements are real and consistent across five runs but moderate in size, and they come from a single synthetic protocol on a soft-PLC. We are careful not to read them as evidence that the approach will transfer unchanged to real vendor firmware; the next section states why.

X. Threats to Validity

We discuss the principal threats to validity so that the reported numbers are read with appropriate caution.

Synthetic proprietary wrapper. The proprietary layer used in our evaluation is *synthetic*. It was designed to resemble common industrial patterns—a magic prefix, a non-standard-endian length field, a session token, and a custom checksum—but it is not a real vendor protocol such as S7Comm or UMAS. Real protocols may include multi-message transactional dependencies, time-varying authentication, or vendor-specific encoding that our wrapper does not reproduce, and the framework's effectiveness on them is unverified.

Soft-PLC target. OpenPLC is a community-developed soft-PLC and is not equivalent to closed-source firmware running on real hardware. Its behaviour under malformed input—how it crashes, whether a watchdog restarts it, how it recovers—may differ materially from production controllers, so the fault classes and discovery rates we observe may not carry over.

Single-protocol, single-base scope. We evaluate one synthetic wrapper over a single base protocol (Modbus/TCP). Protocols with substantially different structure, such as DNP3 with secure authentication or OPC-UA binary transport, are not represented, so generalisation across protocol families remains an open question.

Small-scale testbed and short campaigns. The testbed is a single workstation, and each campaign lasts four hours. Faults that only appear over multi-day campaigns, or behaviour that only emerges under load or in distributed deployments, are out of scope by construction. The five-run design gives reasonable confidence in the means but only coarse variance estimates.

Coverage is a proxy. Statement coverage on the Modbus server is a proxy for how deeply a fuzzer reaches into the target, not a direct measure of exploitable defects; a fuzzer could score well on coverage while finding few real bugs, and the ranking of fuzzers could change under a different coverage notion. The valid-input rate is likewise a proxy for structural understanding and does not distinguish meaningful inputs from merely well-formed ones.

Dependence on an external service. The framework relies on a hosted LLM whose outputs are not deterministic and may change between model versions, which affects reproducibility. We mitigate this by constraining the model to a fixed JSON schema and by treating its hypotheses as advisory rather than binding, but a different model could yield somewhat different mutation strategies and therefore different coverage.

Results may not generalise. Taken together, these threats mean the results should be read as evidence that cheap structure recovery plus external-LLM interpretation is a promising direction on this class of target, not as a claim of broad superiority. Validation on real firmware and additional protocols is required before stronger conclusions are warranted.

XI. Future Work

Several directions follow naturally from the threats above. First, evaluation on real vendor firmware under controlled disclosure would test whether the testbed observations transfer. Second, benchmarking across at least three protocol families would characterise where structural inference helps and where it does not. Third, replacing the single LLM query with a short interactive dialogue—where the framework reports back which hypotheses produced coverage and asks the model to refine them—may improve later-campaign exploration at modest extra cost. Fourth, caching and local distillation of the LLM's structural hypotheses would reduce the dependence on an external service and improve reproducibility. Finally, integrating a lightweight safety filter, so that the tool can be applied cautiously to higher-fidelity simulated equipment with-

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

out issuing physically dangerous commands, would broaden its applicability.

XII. Conclusion

We presented ProtoFuzz-AI, an AI-assisted fuzzing framework for proprietary Industrial IoT protocols that needs neither a hand-written grammar nor a locally trained model. The framework recovers coarse protocol structure from captured traffic with simple statistics, asks an external LLM to interpret that structure and propose protocol-aware mutations, and feeds the result into a coverage-guided fuzzing engine with crash and anomaly monitoring. On a Modbus/TCP testbed with a synthetic proprietary wrapper hosted on OpenPLC, and using only commodity hardware, the framework reached a mean statement coverage of 52%, a 76% valid-input rate, and six unique faults—moderate but consistent gains over an AFLNet-style baseline (44%/61%/4) and Boofuzz (37%/58%/3). The contribution is a reproducible, single-researcher-implementable pipeline that lowers the manual cost of fuzzing undocumented industrial protocols. The principal limitation is one of scope: the evaluation rests on a synthetic wrapper, a soft-PLC, and a single base protocol, and confirming the approach on real firmware and additional protocol families is the most important next step.

Acknowledgment

The author would like to express sincere gratitude to his guide,
ORCID: 0009-0005-5081-7339
Prof. Neenansha Jain, Department of Electronics & Communication Engineering, Oriental College of Technology, Bhopal, for her invaluable guidance, constant encouragement, and insightful feedback throughout the course of this work. Her expertise and patient support were instrumental in shaping both the direction and the execution of this research.

The author is equally grateful to his co-guide,
ORCID: 0009-0003-4773-4968
Sonu Kumar, Department of Civil Engineering, National Institute of Technology (NIT) Jalandhar, for the valuable suggestions, technical insights, and continued support that greatly contributed to the quality of this paper.

The author also thanks the Department of Electronics & Communication Engineering, Oriental College of Technology, for providing the resources and environment that made this work possible.

References

[1] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of UNIX utilities,” *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990.
[2] V. J. M. Manès *et al.*, “The art, science, and engineering of fuzzing: A survey,” *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2312–2331, 2021.
[3] M. Zalewski, “American Fuzzy Lop (AFL),” 2014. [Online]. Available: https://lcamtuf.coredump.cx/afl/
[4] M. Böhme, V.-T. Pham, and A. Roychoudhury, “Coverage-based greybox fuzzing as Markov chain,” in *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS)*, 2016, pp. 1032–1043.
[5] V.-T. Pham, M. Böhme, and A. Roychoudhury, “AFLNet: A greybox fuzzer for network protocols,” in *Proc. IEEE Int. Conf. Software Testing, Validation and Verification (ICST)*, 2020, pp. 460–465.
[6] J. Pereyda, “Boofuzz: Network protocol fuzzing for humans,” 2017. [Online]. Available: https://github.com/jtpereyda/boofuzz
[7] M. Eddington, “Peach fuzzing platform,” Peach Tech, Tech. Rep., 2011.
[8] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, “PULSAR: Stateful black-box fuzzing of proprietary network protocols,” in *Proc. Int. Conf. Security and Privacy in Communication Systems (SecureComm)*, 2015, pp. 330–347.
[9] P. Godefroid, H. Peleg, and R. Singh, “Learn&Fuzz: Machine learning for input fuzzing,” in *Proc. IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, 2017, pp. 50–59.
[10] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, “NEUZZ: Efficient fuzzing with neural program smoothing,” in *Proc. IEEE Symp. Security and Privacy (S&P)*, 2019, pp. 803–817.
[11] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, “Large language model guided protocol fuzzing,” in *Proc. Network and Distributed System Security Symp. (NDSS)*, 2024.
[12] W. Cui, J. Kannan, and H. J. Wang, “Discoverer: Automatic protocol reverse engineering from network traces,” in *Proc. 16th USENIX Security Symp.*, 2007, pp. 199–212.
[13] G. Bossert, F. Guihéry, and G. Hiet, “Towards automated protocol reverse engineering using semantic information,” in *Proc. ACM Asia Conf. Computer and Communications Security (ASIA CCS)*, 2014, pp. 51–62.
[14] T. Alves, R. Buratto, F. M. de Souza, and T. V. Rodrigues, “OpenPLC: An open source alternative to automation,” in *Proc. IEEE Global Humanitarian Technology Conf. (GHTC)*, 2014, pp. 585–589.
[15] P. Huitsing, R. Chandia, M. Papa, and S. Shenoi, “Attack taxonomies for the Modbus protocols,” *Int. J. Critical Infrastructure Protection*, vol. 1, pp. 37–44, 2008.
[16] K. Stouffer, V. Pillitteri, S. Lightman, M. Abrams, and A. Hahn, “Guide to industrial control systems (ICS) security,” NIST Special Publication 800-82 Rev. 2, 2015.
[17] N. Falliere, L. O. Murchu, and E. Chien, “W32.Stuxnet dossier,” Symantec Security Response, Tech. Rep., 2011.
[18] A. Cherepanov, “WIN32/INDUSTROYER: A new threat for industrial control systems,” ESET, Tech. Rep., 2017.
[19] A. Di Pinto, Y. Dragoni, and A. Carcano, “TRITON: The first ICS cyber attack on safety instrument systems,” in *Proc. Black Hat USA*, 2018.