Breaking the Traffic Barrier: Unveiling Multi-Format of Protocols via Autonomous Program Exploration

Dingzhao Xue[1,2] *xuedingzhao@iie.ac.cn*
Yibo Qu[1,2] *quyibo@iie.ac.cn*
Bowen Jiang[1,2] *jiangbowen@iie.ac.cn*
Xin Chen[* 1,2] *chenxin1990@iie.ac.cn*
Shuaizong Si[1,2] *sishuaizong@iie.ac.cn*
Shichao Lv[1,2] *lvshichao@iie.ac.cn*
Zhiqiang Shi[1,2] *shizhiqiang@iie.ac.cn*
Limin Sun[1,2] *asunlimin@iie.ac.cn*

[1]Institute of Information Engineering of CAS, Beijing, China
[2]College of Cyberspace Security, Chinese Academy of Sciences, Beijing, China

***Abstract*—Protocol reverse engineering (PRE) aims to infer the protocol formats of unknown protocols. Existing techniques, whether Network-trace based or Execution-trace based methods, face two main limitations: a reliance on the quality and scale of traffic datasets, which often leads to low accuracy and poor generalization; and a failure to adequately consider the multi-format characteristic prevalent in real-world protocols (i.e., the same protocol may support multiple different formats).**

To address these challenges, we propose ProbePRE—a PRE tool that performs multi-format extraction on protocol handlers by autonomously generating packets. ProbePRE employs three key techniques: (1) an execution tracing strategy enhanced with implicit data flow analysis to obtain more detailed execution information; (2) constraint extraction methods tailored for different program structures to pass protocol validation; and (3) an innovative constraint combination algorithm to construct effective packets that guide the protocol handler to execute diverse protocol parsing paths. In our experimental evaluation, we compared ProbePRE with 4 state-of-the-art PRE tools in terms of field segmentation accuracy. The results demonstrated that ProbePRE achieved an F1 score of 0.88, significantly outperforming existing methods. Furthermore, evaluations on 6 protocol handlers indicated that ProbePRE attained 83% completeness in multi-format extraction tasks. Notably, in basic block coverage tests, ProbePRE achieved a 67% improvement over traditional traffic dataset methods, which fully validates the effectiveness of its path exploration capabilities.

***Index Terms*—Protocol Reverse Engineering, Data Flow Analysis, Taint analysis**

I. INTRODUCTION

Various proprietary protocols see widespread use in domains such as the Industrial Internet and the Internet of Things (IoT), and their message formats and communication processes are often hidden from public access. Without prior knowledge of proprietary protocol specifications, conducting security testing on programs (e.g., fuzz testing [1]–[6]) or deploying defensive measures (e.g., intrusion detection [7]–[9]) becomes challenging. Protocol reverse engineering (PRE) aims to infer the protocol formats of unknown protocols and typically involves three core tasks [10], [11]: field segmentation, semantic inference, and state machine extraction [12]–[16].

The purpose of this paper is to perform field segmentation on proprietary protocol packets, which serves as the foundation for subsequent semantic inference and state machine extraction. Unlike existing methods that recover formats for individual packets, the primary focus of this work is on extracting a protocol's multiple message formats directly from the protocol handler itself.

Current methods for field segmentation mainly fall into two categories. Network-trace based methods [17]–[22] rely on statistical features or machine learning models to identify field boundaries from packet traces, but they are often limited by the scale and diversity of traffic datasets, leading to poor generalization and low accuracy. Execution-trace based methods [23]–[25], on the other hand, segment fields by analyzing the execution process of a program parsing packets, which generally improves accuracy and generalization. However, these methods typically focus on single input formats and, consequently, struggle to output all message formats for protocols with multiple message formats or complex structural variations, particularly when input is limited.

Developers implement protocol handlers to parse packets according to protocol specifications; consequently, these handlers are inherently capable of parsing diverse message formats. Building upon this premise, this paper introduces ProbePRE, a system designed for protocol multi-format extraction. ProbePRE operates without relying on large-scale, diverse traffic datasets. Instead, inspired by fuzz testing principles, it leverages program execution information to guide the generation of probe packets. This process triggers the protocol handler's parsing of multiple message formats, ultimately enabling the extraction of the protocol's comprehensive multi-format specification.

To implement our method, we must address the following challenges: (1) Robustness when dealing with diverse protocol handlers. Traditional methods, which primarily rely on heuristic algorithms based on execution traces for field segmentation, are often empirical and rule-based. Consequently, they

Xin Chen is the corresponding authors. This project is supported by the National Natural Science Foundation of China (Grant No. 92467201).

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00173

are typically applicable only to protocol handlers with fixed characteristics and exhibit significant limitations when faced with diverse protocols. (2) Difficulty in satisfying strict field validation. A protocol handler can interrupt the parsing process due to illegal field values (i.e., values outside the constraint range), leading to incomplete field segmentation. Furthermore, obtaining legitimate field constraints is challenging in the absence of traffic data and source code. (3) Complexity of modeling inter-field dependencies. Constructing probe packets that conform to inter-field dependencies is crucial for triggering corresponding parsing paths. However, identifying such dependencies from binary programs and execution information poses difficulty.

To address the first challenge, we introduce an execution tracing strategy enhanced with implicit data flow analysis, designed to capture richer execution information. To mitigate the low robustness often associated with heuristic algorithms, we opt to directly optimize execution tracing strategy. Our key insight is that when a protocol handler parses packets, bytes originating from the same field tend to exhibit similar propagation characteristics at the data flow level. Furthermore, these underlying characteristics are prevalent across a wide array of protocol handlers. By enhancing our capability to trace implicit data flows, our method effectively overcomes the limited robustness inherent in traditional heuristic algorithms.

To address the second challenge, we propose a field constraint extraction method targeting diverse program structures. Our approach integrates two key aspects: dynamic execution tracing and static program analysis. Dynamic execution tracing applying program slicing to instruction sequences pertinent to field comparisons, enabling the direct extraction or recovery of constraint values based on program semantics. Static program analysis identifies all switch structures within the program and accurately matches them to specific fields via data flow analysis, subsequently leveraging these case values as effective constraints. This dual-pronged strategy ensures coverage of various forms of constraint expressions within the program.

To address the third challenge, we propose a constraint combination algorithm leveraging dynamic depth-first search. This algorithm models inter-dependencies among fields and their associated constraints using a tree-like data structure; notably, its dynamic extension mechanism permits the real-time insertion of newly discovered fields and constraints. Through systematic traversal of the entire constraint tree, the algorithm generates probe packets satisfying specified field dependencies, thereby effectively triggering potential protocol parsing paths within the program.

We implemented ProbePRE and evaluated its performance against four state-of-the-art protocol reverse engineering tools [24]–[27]. In terms of field segmentation performance, our experimental results demonstrated that ProbePRE significantly outperformed existing methods, achieving an average precision of 0.84, recall of 0.94, and an F1 score of 0.88. Compared to the results of BinPRE (0.94, 0.75, 0.83), Polyclot (0.80, 0.89, 0.82), AutoFormat (0.77, 0.94, 0.82), and Tupni (0.90, 0.50, 0.55), ProbePRE exhibited significant improvement. To validate its multi-format extraction capability, we applied ProbePRE to six different protocol handlers; the results indicated that the completeness of extracted protocol formats reached 83%. Notably, in basic block coverage tests, ProbePRE achieved a 67% improvement over traditional traffic dataset methods. These experimental results fully demonstrate that ProbePRE not only excels at multi-format extraction tasks, but its performance also surpasses that of existing methods reliant on traffic datasets.

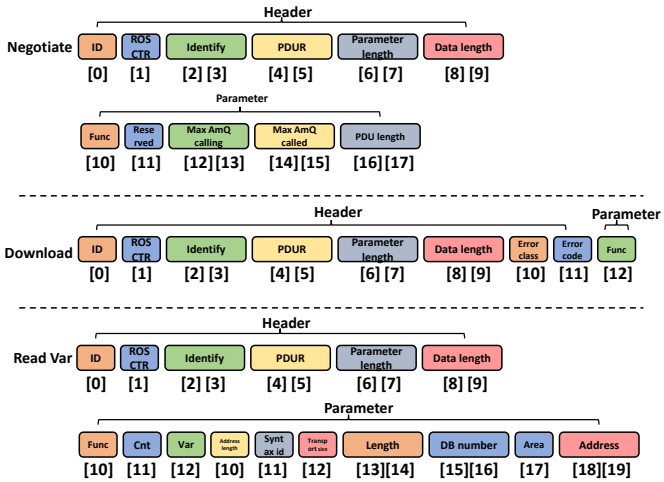

Fig. 1: Multi-format example of S7comm protocol.

In this paper, we make the following contributions:

- We introduce a novel protocol multi-format extraction method that, for the first time, addresses the challenge of protocol multi-format extraction. This approach overcomes the reliance of traditional PRE tools on traffic datasets and enables the extraction of diverse protocol message formats.
- Drawing inspiration from fuzz testing methodologies, we developed techniques for field constraint extraction and probe packet construction. Through iterative packet generation and dispatch, our method progressively increases the number of covered protocol parsing paths within a program, thereby facilitating the extraction of multiple message formats embedded within the program.
- We implemented ProbePRE, which can not only perform multi-format extraction tasks but also outperforms existing methods reliant on traffic datasets. Experimental results show that ProbePRE exhibits significant advantages in key metrics such as field segmentation accuracy and protocol format completeness.

II. MOTIVATION

*a) **Cases of multi-format in single protocol**:* Communication protocols are designed with different communication functions according to varying requirements, where distinct functions may employ different message formats. Taking the S7comm protocol, widely used in the industrial control domain, as an example, Figure 1 illustrates three different mes-

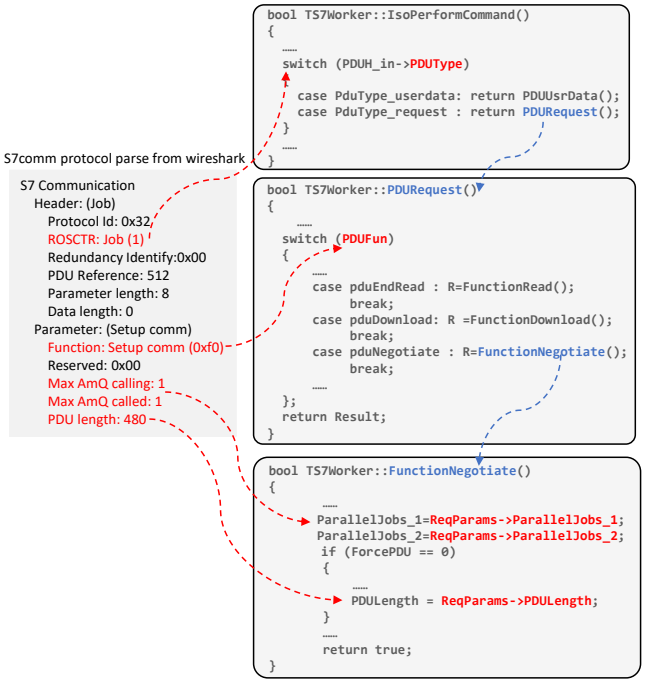

Fig. 2: S7comm handler parsing example.

sage formats: Negotiate, Download, and Read var. Observation reveals that only the field segmentation within the first 9 bytes remains consistent across these formats; subsequent sections exhibit divergent field segmentation, and the lengths of the 'Header section' and 'Parameter section' are also variable.

Specifically, "Negotiate" packet facilitate the communication negotiation phase, accordingly, their 'Parameter section' includes 'Max AmQ calling' and 'Max AmQ called' fields to establish parameters for job handling. "Download end" packet are employed to terminate download operations, their 'Header section' thus contains 'Error class' and 'Error code' fields to report errors encountered during the download process. "Read Var" packets serve to retrieve variable data from a PLC, with their 'Parameter section' consequently including fields such as 'Count' to specify details of the variable read request.

b) ***Relation between format and parse path****:* Field segmentation in message formats manifests within a protocol handler as the reading, storing, and utilization of variables along execution paths. Figure 2 illustrates how the S7comm protocol handler parses Negotiate packets: Initially, the TS7Worker::IsoPerformCommand function determines the packet type by parsing its ROSCTR field (mapped to the PDUType variable). Subsequently, the TS7Worker::PDURequest function selects a specific functional branch based on the Function field (corresponding to the PDUFun variable); in Figure 2, the FunctionNegotiate function is chosen. Finally, the TS7Worker::FunctionNegotiate function proceeds to parse the parameter section. Observation reveals that protocol parsing paths within a protocol handler effectively define the protocol format specifications.

Existing methods often overlook the possibility that a single protocol may encompass diverse message formats. Theoretically, if an input traffic dataset were to cover all such formats, these methods could indeed achieve multi-format output. However, constructing high-quality, comprehensive traffic datasets for protocol presents considerable challenges, particularly in proprietary protocol scenarios. Existing technical approaches struggle to effectively assess how well collected traffic data covers the variety of protocol formats. We examined the S7comm protocol traffic dataset employed by BinPRE [26]. Our statistical analysis revealed that of S7comm's 13 inherent protocol parsing paths, packets within this dataset covered only 7. Consequently, nearly half of these paths remained untriggered.

c) ***Poor robustness of heuristic methods****:* To illustrate the limitations inherent in heuristic methods, we examine BinPRE's approach as a case in point, which employs instruction similarity for field segmentation. Figure 4 presents a code excerpt from a Modbus protocol handler parsing packets. In this example, for the fields 'slave', 'function', and 'nb_bytes', the corresponding instruction sequence is consistently 'movzx'. For the 'address' and 'nb' fields, the instruction sequence is uniformly 'movzx, movzx, shl, add'. Consequently, in such scenarios, relying solely on instruction sequence similarity for field segmentation proves ineffective, leading to under-segmentation.

III. APPROACH

A. Overview

Figure 3 presents an overview of the ProbePRE tool. Its fundamental principle is to perform field segmentation and field constraint extraction based on the execution trace information of the protocol handler, subsequently recombining the fields and constraints to generate new probe packets that guide the protocol handler to execute potential protocol parsing paths.

The system consists of four modules: Execution Processor, Format Extractor, New Packet Constructor, and Two-layer Clustering. The workflow begins with the Execution Processor, which dynamically traces the protocol handler parsing packets, capturing fine-grained execution information including data flow info, cmp instruction info, and function execution chains. The Format Extractor then processes the collected execution traces to perform field segmentation and constraint extraction, respectively. Based on the results of field segmentation and constraint extraction, the New Packet Constructor constructs new probe packets. These packets are re-inputted into the protocol handler to trigger potential parsing paths. By iteratively executing the aforementioned process, the system continuously expands its knowledge base of protocol formats until no new valid constraint combinations can be discovered. Finally, the Two-layer Clustering module clusters the format information accumulated during the iterative process and outputs the discovered protocol formats.

B. Execution Processor

The Execution Processor employs a dynamic taint analysis technique, enhanced with implicit data flow analysis, to per-

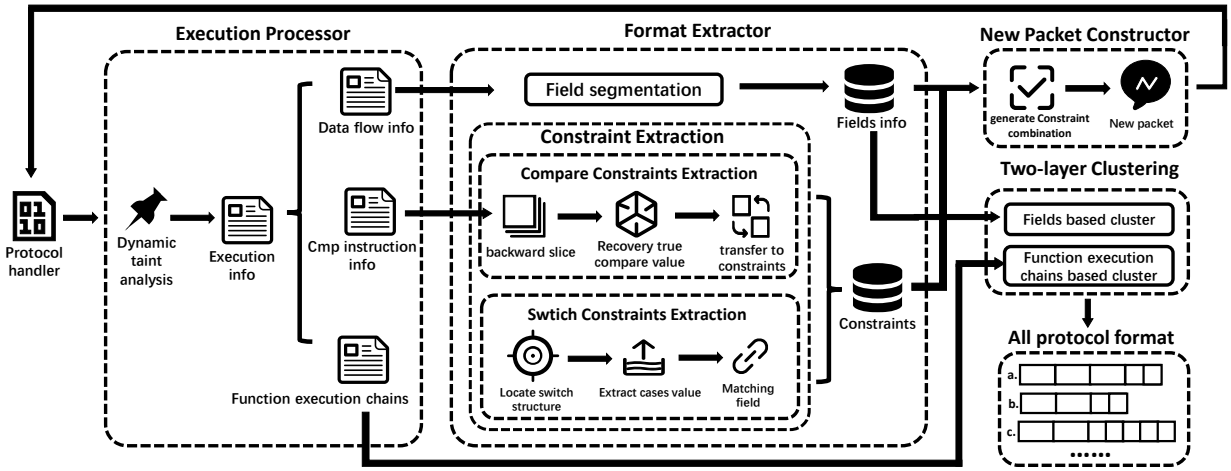

Fig. 3: Overview of ProbePRE.

form fine-grained execution tracing of the protocol handler, comprehensively recording the packet's parsing process within the program. Specifically, this involves a three-dimensional tracing strategy: the Data Operation Dimension initializes each byte of the input packet as an independent taint source, and all explicit instruction operations involving tainted data (including data transfer and arithmetic instructions like MOV, ADD) are tracked; the Control Flow Dimension monitors key instructions involved in tainted data comparison and branching, especially CMP instructions and conditional jump instructions; and the Call Chain Dimension records the complete function execution chains traversed during tainted data propagation.

The execution information captured across these three dimensions exhibits clear semantic correlations: Tainted data flow characteristics directly reflect protocol field segmentation; operands of comparison instructions, combined with branching conditions, together form a complete description of field value range constraints; whereas the diversity of function execution chains characterizes the different parsing paths the program may take.

Notably, the Execution Processor, compared to traditional execution tracing strategies, enhances tracing by including implicit data flows. Traditional execution tracing typically focus solely on explicit data transfer instructions (such as MOV, ADD) and primarily rely on subsequent heuristic algorithms for field segmentation. However, in real-world scenarios, compiler optimizations can generate unconventional instructions to perform numerical calculations. For instance, in the S7comm protocol handler, the actual semantics of the instruction lea edx, ptr [rbp+rdx*1+0xa] are equivalent to an arithmetic sum operation (rbp+rdx+0xa), rather than its nominal address loading operation. By extending taint propagation rules, the system incorporates instructions like LEA, which have implicit computational semantics, into its tracing scope. This enhances the completeness of data flow information and the accuracy of field segmentation. This improvement enables the system to capture critical data propagation paths that traditional methods might miss.

C. Format Extractor

*1) **Field Segmentation**:* The Format Extractor performs protocol field segmentation based on tainted data flow characteristics. Its core mechanism involves detecting whether adjacent bytes of an input packet flow to the same memory location or register. This approach is theoretically grounded in the fundamental behavior of protocol handlers: when a program, adhering to the protocol specification, combines and stores different bytes into a same target variable, the data propagation paths of these bytes will invariably converge.

```
0x46fd movzx   r15d, byte ptr [rsi+rdx];
slave = req[offset-1];
0x4705 movzx   ecx, byte ptr [rsi+rdx];
function = req[offset];
```

```
0x470d movzx   ebx, byte ptr [rbp+rsi+0];
req[offset+1]
0x471b movzx   esi, byte ptr [rbp+rsi+0];
req[offset+2]
0x4724 shl     ebx, 8;
req[offset+1]<<8
0x472f add     ebx, esi;
address = (req[offset+1]<<8)+req[offset+2];
```

```
0x4736 call    qword ptr [rax+28h];
sft.t_id=prepare_response_tid(req,&req_length);
```

```
0x4a78 movzx   eax, byte ptr [rbp+rax+0];
req[offset+3]
0x4a7d movzx   edx, byte ptr [rbp+rdx+0];
req[offset+4]
0x4a82 shl     eax, 8
0x4a85 add     eax, edx;
nb = (req[offset+3]<<8)+req[offset+4]
```

```
0x4a96 movzx   edx, byte ptr [rbp+rdx+0];
nb_bytes = req[offset+5]
```

Fig. 4: Partial assembly & source code: Modbus handler.

Taking the parsing process of a Modbus protocol handler [28] as an example (as illustrated in Figure 4, where black text represents assembly code and blue text indicates corresponding source code), the protocol handler performs protocol parse through a series of memory operation instructions. Initially, the MOV instruction at address 0x46fd reads a single byte from

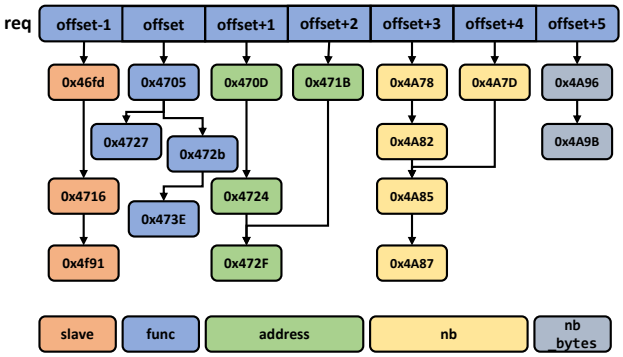

Fig. 5: Data flow visualization of a Modbus handler.

the packet at ”offset-1” and stores it into the ”slave” variable. Subsequently, the instruction at address 0x4705 extracts the byte at ”offset” as the ”function code”. Following this, the assembly code segment from 0x470d to 0x472f left-shifts the byte at ”offset+1” by 8 bits, then performs a bitwise OR operation with the byte at ”offset+2”, storing the combined result into the ”address” variable. Similarly, the assembly code segment from 0x4a78 to 0x4a85 left-shifts the byte at ”offset+3” by 8 bits, performs a bitwise OR operation with the byte at ”offset+4”, and stores the combined result into the ”nb” variable. Finally, the instruction at 0x4a96 stores the byte at ”offset+5” into the ”nb_bytes” variable.

Visual analysis of data flow tracing (Figure 4) shows that the data flow paths associated with the ”slave” variable (derived from the byte at ”offset-1”) and the ”function” variable (derived from the byte at ”offset”) propagate independently, consistent with their single-byte field attributes. In contrast, the data flows from the bytes at ”offset+1” and ”offset+2”—which are combined to form the ”address” field—merge at address 0x472f, exhibiting a distinct path intersection characteristic. The ”nb” field exhibits a similar convergence pattern. These topological characteristics of byte-level data flow propagation paths provide a reliable basis for automated field segmentation.

*2) **Constraint Extraction**:* After completing the field segmentation of input packets, Constraint Extraction proceeds to extract the constraints present in the protocol parsing paths. The essence of protocol parsing is the protocol handler's reading of different bytes from the input packets and the validation of field values. Extracting the constraint range for each field is an important prerequisite for ensuring that the protocol handler parses the protocol correctly.

*a) **Compare Constraints Extraction**:* A compare constraint refers to a constraint imposed on field values through the combination of cmp instructions and jump instructions. When a field value adheres to this constraint specification, the program continues to parse subsequent bytes. Conversely, if the current field value falls outside the legitimate range, execution jumps to error handling code, and a response containing an error description is returned to the client program.

In protocol handlers, cmp instructions exhibit two typical patterns when comparing field values against constraint values: The first is a direct comparison; the second involves comparing a transformed field value (i.e., not the original value) with a constraint value. For the former, ProbePRE can directly deduce the field's constraint range by analyzing the semantics of the constraint value and the associated jump instruction. For the latter, ProbePRE needs to revert the constraint value to make it applicable to the original field value.

First, the calculated field values are identified: by traversing the cmp instruction info, the field value in each cmp instruction is compared with the original value in the data packet. If they are not equal, it is determined that the field value has undergone a calculation transformation. Subsequently, for the target field, backward slicing is performed from the cmp instruction until the field value is equal to the original value, at which point the slicing stops. The obtained program segment has performed calculations on the target field value. Utilizing the code understanding capabilities of a large language model, semantic analysis is performed on the program slice to automatically deduce the calculation formula for the field value. The prompt templates are illustrated in the Figure 6. Based on the extracted calculation formula, the constraint value in the cmp instruction is inversely calculated to restore the true constraint value of the target field.For the combination of the true constraint value and the jump instruction, we design transformation rules to convert the constraint value into a constraint range according to the semantics of the jump instruction. Specifically, based on the semantics of the jump instruction, the constraint value is used as either the left boundary value or the right boundary value of the range, and the other boundary value is completed with 0 or the maximum possible value of the field.

Taking the combination (0x123, jb) as an example ((0x123, jb) represents the instructions cmp field_value, 0x123; jb instr_addr), where 0x123 is the constraint value and jb is the jump instruction), the semantics of the jb instruction is that when the destination operand in the cmp instruction is less than the source operand, the program will jump to the specified address for execution. Therefore, the program will jump when the field value is less than 0x123, meaning the constraint range corresponding to the field is (0, 0x123). If there are multiple constraint ranges for a field, such as (0, 0x123) and (0x12, 0xffff), they need to be merged to obtain the true constraint range, which is (0x12, 0x123).

From the execution information, it is difficult to know whether the jump instruction will direct the program to error-handling code or subsequent parsing code when the field value is within the constraint range. Therefore, to avoid missing subsequent parsing code and to cover all paths as much as possible, the constraint range also needs to be complemented. Still using (0x12, 0x123) as an example, assuming the field length is two bytes, then the maximum and minimum values of the field are 0xffff and 0, respectively. So, after complementing the constraint range, the resulting constraint sequence is (0x0, 0x11), (0x12, 0x123), (0x124, 0xffff).

*b) **Switch Constraints Extraction**:* In the process of protocol data packet parsing, the value of a specific field determines the type of the data packet. Protocol handlers

Prompt Template

Task:

You are a professional binary analyst, specialized in data flow analysis of assembly code. Given a snippet of assembly code, please analyze the complete data flow of a specified register or value within the snippet, and output the final expression with respect to the initial value x.

Requirements:

1. Do not explain the analysis process; only output a concise mathematical expression.
2. Expression rules:
   - Direct transfer (e.g., MOV): x
   - Arithmetic/logic operations (e.g., ADD, LEA): reflect the operation (e.g., x + 1)
   - If the value is not used or affected in the snippet: output [unchanged]
   - All numbers should be in decimal
   - Use parentheses only when necessary

Example:

Code snippet:
mov eax, ebx
add eax, 0x1

Target of analysis:
data flow of ebx

Output:
x + 1

To analyze:

Code snippet:
[Code snippet]

Target of analysis:
[Target register]

Fig. 6: Prompt template for generating constraint computation formulas

typically employ a switch structure to implement multi-way branch parsing based on field values. Accurately obtaining the complete value range of such fields is a key prerequisite for triggering all potential parsing paths. To this end, we designed a switch constraints extraction strategy to extract the constraint range of such fields. Different from the extraction method for cmp constraints, the extraction of switch constraints adopts a hybrid static and dynamic analysis approach.

Static Localization Phase: By identifying jump table instructions, all functions containing switch structures in the program are located. Then, the target functions are disassembled, and the specific values corresponding to each case branch are extracted from the generated pseudo-code. These case values constitute the constraint set for that switch structure.

Dynamic Matching Phase: Since not all switch structures are used for field parsing, it is necessary to establish the correspondence between a switch and a specific field. First, starting from the jump instruction of each switch structure's jump table, static backward data flow analysis is performed to trace the data source of the switch condition variable. All instruction addresses involving this variable from the function entry to the jump instruction are recorded. Combined with data flow info (which explicitly records the mapping relationship between instructions and input fields), the protocol field actually corresponding to each switch structure is determined. The constraint range of this field is the constraint set obtained in the static phase.

Starting from the instruction that jumps to the jump table, we conduct static reverse data flow analysis on each switch

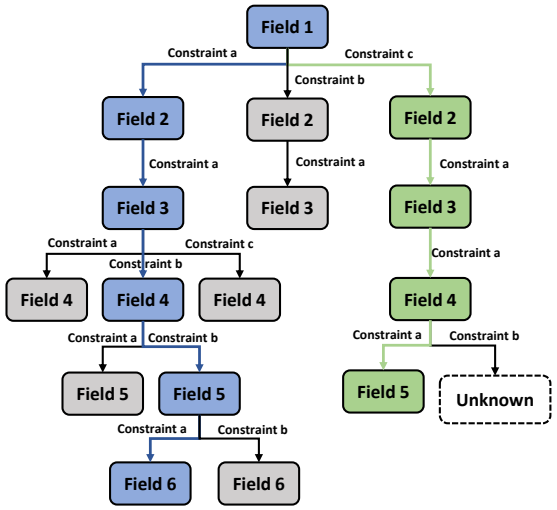

Fig. 7: The example of format tree.

structure, track the source of the data in the variables parsed by the switch, and all the instruction addresses involving this data between the function entry and the jump table instruction will be recorded for subsequent matching.

The relationship between each instruction and the input data packet field is recorded in the data info. We search for the recorded instruction address in the data info and finally obtain the field corresponding to the switch constraints.

D. New Packet Constructor

The New Packet Constructor models the extracted fields and constraint information as a tree structure, termed a format tree. In this structure, nodes at each level correspond to a field, and different branches of each node correspond to different constraints of the field. A path from the root node to a leaf node represents a potentially existing protocol format. The New Packet Constructor traverses all possible paths in the format tree to construct new packets that satisfy the constraints and sends these packets to the program under test for the next round of analysis. As shown in Figure 7, in the constructed format tree, the blue path and the green path respectively represent two different protocol formats that the protocol handler can correctly parse, while the node marked "unknown" indicates that when field4 selects constraint b, the newly generated probe packet may discover new fields or constraints. As the probing process deepens, this format tree will gradually expand and be refined, eventually converging to the true protocol format representation.

To achieve a complete traversal of all field constraint combinations, the New Packet Constructor employs a depth-first search algorithm as its basic traversal strategy. Considering newly discovered fields or constraints during the probing process, we have specifically designed a real-time update mechanism. This mechanism allows for the dynamic expansion of the tree structure without interrupting the current traversal process. Simultaneously, for newly discovered fields or constraint paths, the New Packet Constructor will trigger a

backtracking mechanism to re-probe the current path to obtain potentially more format information.

Algorithm 1: Constraint Combination Generator

Input : F: List of fields (field offsets)
C: Constraint dictionary {offset:[constraints]}
SC: Current DFS status stack
SR: Restored DFS status stack

Output: new_path: Combination dictionary {offset: constraint}

```
1  Procedure UpdateInfo(F, C)
2      f ← UpdateFields()
3      c ← UpdateConstraints()
4      if f ≠ F ∨ c ≠ C then
5          F ← f
6          C ← c
7          return True
8      else
9          return False
10 Procedure GenerateCombinations(F, C, SR)
11     if UpdateInfo(F, C) then
12         SC ← SR
13     while SC ≠ ∅ do
14         SR ← SC
15         (F_idx, C_idx, current_path) ← SC.Pop()
16         if ¬Validate(F_idx, C_idx) then
17             continue
18         offset ← F[F_idx]
19         constraints ← C[offset]
20         constraint ← constraints[C_idx]
21         new_path ← current_path ∪ {offset :
             constraint}
22         if C_idx + 1 < |constraints| then
23             SC.Push((F_idx, C_idx +
               1, current_path))
24         if F_idx + 1 < |F| then
25             SC.Push((F_idx + 1, 0, new_path))
26         else
27             return new_path
28     return None
```

Based on these design considerations, we propose Algorithm 1 to implement the construction and traversal functions of the format tree. This algorithm uses a field list and a constraint dictionary to store the format tree information. The update_info function (lines 1-9) is responsible for handling the dynamic updates of fields and constraints. When an update operation is detected, lines 11-12 ensure that the DFS traversal state can correctly backtrack to the state before the update. Lines 13-27 implement the depth-first traversal logic with memoization, ensuring that all paths can be completely explored by the system.

E. Two-layer Clustering

ProbePRE terminates the exploration of potential parsing paths of the protocol parser when the complete traversal of the format tree is finished and no new fields or constraints are discovered. Subsequently, the Two-layer Clustering module summarizes the protocol formats obtained from the exploration through a two-stage clustering process. Each complete parsing path corresponds to a specific protocol format, and different protocol formats may exhibit similarities in their field partitioning structure, with their differences primarily manifested in path selection behavior determined by field constraint values. Based on this observation, the system first uses the field partitioning structure as the first-layer clustering feature to perform a coarse-grained categorization of protocol formats. On this basis, it further uses the function call chains recorded during program execution as the second-layer clustering feature to perform a fine-grained sub-categorization of protocol formats within each coarse-grained category. Finally, all protocol formats extracted by ProbePRE are outputted.

F. Implementation

The Execution processor module tracks the execution of the protocol processor through the intel pin tool [29], [30]. Compare constraints extraction utilizes the semantic understanding ability of the Deepseek large model [31] to extract the calculation formulas of field values. Switch constraints extraction relies on the IDA pro [32] tool for the positioning and disassembly of swtich structures. The New packet constructor uses Scapy [33] to construct the protocol stack and sends it to the program under test. The rest of the content is implemented based on python3 [34].

IV. EXPERIMENTS

In this section, we address the following research questions:

- **RQ1**: How accurate is ProbePRE in field segmentation?
- **RQ2**: How comprehensive is ProbePRE in extracting the variety of message formats?
- **RQ3**: How significantly does the constraint extraction strategy impact ProbePRE's performance?
- **RQ4**: How much does ProbePRE improve program coverage compared to input datasets?
- **RQ5**: How do different LLMS affect the results of ProbePRE?
- **RQ6**: How ProbePRE can be used in real-world settings as protocol fuzzing?

A. Setup

a) Baseline: We compare the field segmentation accuracy of ProbePRE with four state-of-the-art PRE tools.

- BinPRE [26] is an Execution-trace based tool that performs field segmentation by employing instruction similarity analysis based on execution tracing of input data.
- Polyglot [24] is an Execution-trace based tool that treats bytes subjected to loop comparisons as delimiters, using their positions as field boundaries.

TABLE I: Field segmentation results per tool across various protocols. **Bold** indicates the best.

Protocol	PROBEPRE			BINPRE			POLYCLOT			AUTOFORMAT			TUPNI		
	Pre.	Rec.	F1	Pre.	Rec.	F1	Pre.	Rec.	F1	Pre.	Rec.	F1	Pre.	Rec.	F1
S7comm [35]	0.92	0.97	**0.94**	**0.99**	0.80	0.89	0.89	**0.99**	0.93	0.90	**0.99**	0.94	0.89	0.98	0.93
Libmodbus [28]	0.75	0.85	0.80	**1.0**	0.84	**0.91**	0.77	0.97	0.86	0.63	**0.98**	0.77	**1.0**	0.13	0.24
Freemodbus [36]	0.87	**0.99**	0.93	**1.0**	0.99	**0.99**	0.87	0.99	0.93	0.87	0.99	0.93	0.77	0.99	0.87
IEC104 [37]	0.80	**0.80**	0.79	0.76	0.48	0.57	**0.89**	0.73	**0.81**	0.89	0.62	0.71	0.84	0.50	0.59
BACnet [38]	0.82	0.95	0.87	**0.93**	0.85	**0.88**	0.82	**0.97**	0.88	0.79	0.97	0.86	0.78	0.43	0.51
DNP3.0 [39]	**0.92**	**1.0**	**0.96**	0.88	0.63	0.74	0.88	0.55	0.67	0.92	1.0	0.96	1.0	0.39	0.56
Ethernet/IP [40]	0.81	**1.0**	**0.89**	**1.0**	0.69	0.81	0.5	1.0	0.66	0.38	1.0	0.55	1.0	0.07	0.14
Average	0.84	**0.94**	**0.88**	**0.94**	0.75	0.83	0.80	0.89	0.82	0.77	0.94	0.82	0.90	0.50	0.55

- AutoFormat [27] is an Execution-trace based tool that merges consecutive records with the same execution context into a single field.
- Tupni [25] is an execution-trace based tool that identifies field formats from instruction frequencies and record sequences.

b) Benchmarks: We selected six protocols widely used in the Industrial Internet and Internet of Things (IoT) domains. Four of these protocols—Modbus, S7comm, DNP3.0, and FTP—have been experimented with in prior work. For each protocol, we set up 300 packets for comparison with ProbePRE.

c) Metrics: We adopt different metrics for different tasks. The field segmentation task is evaluated using precision (i.e., the number of true field boundaries inferred out of all inferred field boundaries), recall (i.e., the number of true field boundaries inferred out of all true boundaries), and F1 score. For the format extraction task, we use completeness as the evaluation metric. Because the protocols under test lack unified and standardized documentation, we manually analyze protocol handlers to count the variety of formats. Finally, we use basic block coverage to evaluate the coverage of protocol handlers achieved by ProbePRE and traffic datasets.

B. RQ1: The accuracy of field segmentation

To evaluate the field segmentation performance of ProbePRE, we selected seven typical protocol handlers and their corresponding traffic datasets for testing. We also compared it with four state-of-the-art Execution-trace based PRE tools. Table I shows the detailed experimental results.

The experimental results indicate that ProbePRE significantly outperforms existing methods in overall performance, with its average precision, recall, and F1 score reaching 0.84, 0.94, and 0.88, respectively, representing a performance improvement of 12%-17% compared to baseline methods. This advantage primarily stems from ProbePRE integrating implicit data flow analysis into execution tracing, thereby obtaining more detailed format information.

The other four PRE tools employ different heuristic methods: BinPRE utilizes instruction semantic similarity analysis, making it particularly suitable for handling string fields (due to their common semantics of loop-based reading), and thus performs exceptionally well on the mixed protocol Bacnet; Polyclot, on the other hand, focuses on the program's comparison behavior with input data to locate field delimiters, enabling it to achieve high accuracy on the Iec104 protocol. However, the specific protocol characteristics these tools rely on are not universally present, leading to unstable performance. In contrast, ProbePRE, through optimization of the underlying execution tracing mechanism, demonstrates stronger universality, thus maintaining a leading edge in the overall evaluation.

It should be noted that, since key fields that determine execution paths will inevitably be read by the protocol handler and will definitely be identified, even if some non-critical fields are not identified, it will not affect subsequent path exploration.

TABLE II: Format extraction results of ProbePRE on various protocol handlers. (#Formats denotes the number of extracted formats and #RealFmt denotes the number of real formats.)

Binary	Single Run		Avg. of 5 Runs		#RealFmt
	#Formats	Time(min)	#Formats	Time(min)	
S7comm [35]	13	3.33	13	3.59	13(100%)
Libmodbus [28]	9	1.60	9	1.55	10(90%)
Freemodbus [36]	1	0.30	1	0.63	10(10%)
IEC104 [37]	8	6.69	8	6.69	8(100%)
FTP [41]	7	1.31	7	1.52	7(100%)
Ethernet/IP [42]	8	0.88	8	0.90	8(100%)
Average	7.67	2.35	7.67	2.48	9.33(83%)

C. RQ2: The completeness of format extraction

Due to the lack of standardized format documentation for the protocols under test, to objectively quantify the evaluation effectiveness, we manually analyze protocol handlers to count the number of parsing paths for different formats: for example, in the S7comm protocol handler shown in Figure 2, we consider "IsoPerformCommand→PDURequest→FunctionNegotiate" as a parsing path for Negotiate packets.

We tested ProbePRE's format extraction capability on six different protocol handlers. The experimental data show that ProbePRE's average format extraction completeness reached 83%. As shown in Table II, the tool performed excellently on four protocols: S7com, Iec104, Ftp, and Enip, where the number of extracted formats perfectly matched the actual number, achieving 100% accuracy. Particularly noteworthy is that

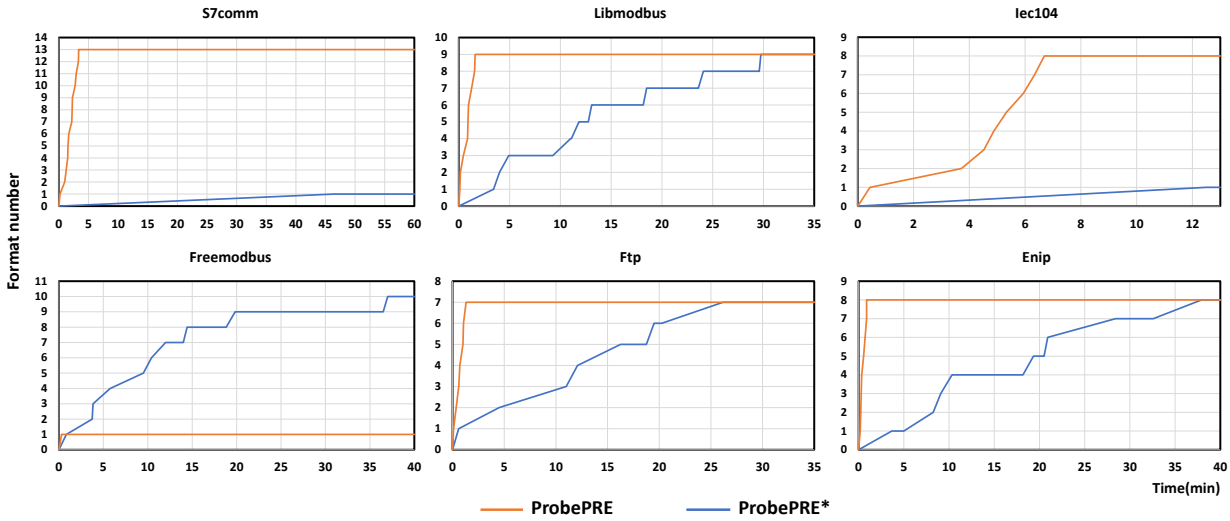

Fig. 8: Comparison of time costs: ProbePRE and ProbePRE*.

when processing the S7comn protocol with complex formats, ProbePRE not only accurately identified two key fields but also precisely extracted their constraint ranges, effectively reducing the exploration space and significantly decreasing the probing time.

However, the test results on the Freemodbus protocol handler revealed a limitation: ProbePRE only successfully extracted one message format. Analysis revealed that Freemodbus employs an atypical path selection mechanism—unlike the switch or if-else structures commonly used in protocols, it uses a loop structure to match key field values. After compilation, loop structures are also manifested as consecutive cmp instructions in the assembly. However, unlike the if-else structure where constraint values are explicitly encoded as immediates in cmp instructions, loop structures implicitly store the comparison values in registers or memory. This makes it difficult for ProbePRE's compare constraints extraction strategy to directly obtain the corresponding constraint values, thereby limiting its ability to effectively guide the exploration of alternative parsing paths. This observation highlights a promising direction for improvement: we envision refining the Pin-based dynamic tracing strategy to output the comparison values stored in registers or memory within loop-based cmp instructions. Furthermore, ProbePRE's compare constraints extraction strategy can be extended to support such cmp instructions with implicit comparison values.

In addition, we conducted five experiments to demonstrate the determinism of ProbePRE. The table reports both the result of a single run and the average over five runs, which show no significant differences. This is because the constraint information of fields is embedded in the protocol parser itself and does not change across runs. Moreover, ProbePRE constructs new packets by traversing constraint combinations using a depth-first algorithm, rather than randomly combining field constraints or sampling field values. Therefore, the new packets generated by the New Packet Constructor have no randomness.

D. RQ3: The influence of constraint extraction strategies

This section evaluates, through an ablation study, how the constraint extraction module enhances system performance. We compared two strategies: the full version of ProbePRE (including the constraint extraction module) and a simplified version that only employs a random mutation strategy, measuring the time cost of both. The experimental results are detailed in Figure 8 and Table III.

TABLE III: The result of influence of constraint extraction strategy.

Binary	ProbePRE*		ProbePRE	
	#Format	Time(min)	#Format	Time(min)
S7comm [35]	1	46.42	13	3.33
Libmodbus [28]	9	29.75	9	1.60
Freemodbus [36]	10	37	1	0.30
IEC104 [37]	1	12.5	8	6.69
FTP [41]	7	26.17	7	1.31
Ethernet/IP [40]	8	37.92	8	0.88
Average	6	31.63	7.67	2.35

The experimental data show that the constraint extraction module significantly enhances system efficiency. The full version of ProbePRE completes format extraction with an average time cost controlled within 3 minutes, not exceeding 8 minutes at most, achieving an efficiency improvement of up to 90% compared to the random mutation strategy. This performance advantage stems from the constraint extraction module's ability to accurately obtain field constraint ranges, thereby substantially reducing the exploration space.

Further analysis reveals that although the random mutation strategy is time-consuming on protocols such as Libmodbus,

Freemodbus, FTP, and Ethernet/IP (averaging over 30 minutes), it can still discover some parsing paths. This is because the path selection in these protocols depends only on a single key field. However, for the S7comm and IEC104 protocols, which have multi-field dependency characteristics, the random mutation strategy performs poorly: it not only struggles to simultaneously satisfy the legitimate value requirements for multiple fields but also cannot guarantee the dependencies between fields. In contrast, the constraint extraction module can accurately capture the constraint relationships between fields and works in conjunction with the packet construction module to effectively guide the program to execute different parsing paths.

TABLE IV: The result of comparing the basic block coverage of ProbePRE and the traffic dataset.

Binary	ProbePRE(#BBs)	Dataset(#BBs)	Increase
S7comm [35]	11592	8526	36%
Libmodbus [28]	767	438	75%
IEC104 [37]	5586	2206	153%
Ethernet/IP [40]	1746	1678	4%
Average	4923	3212	67%

E. RQ4: Comparison with traffic datasets

To quantitatively evaluate the effectiveness of ProbePRE in exploring the protocol parsing paths of protocol handlers, we use basic block coverage as the core metric. We compare the number of basic blocks triggered in protocol handlers by traffic datasets with the number of basic blocks triggered by ProbePRE through probing.

In the experimental design, we collected baseline datasets for four protocols: Ethernet/IP [42], IEC104 [43], [44], Libmodbus [45], and S7comm [46], with each protocol dataset containing 300 real traffic packets. The basic block coverage obtained after inputting these packets into their respective programs serves as the baseline. The experimental results in Table IV show that the constraint extraction module significantly improves basic block coverage: a 36% improvement for S7comm, a 75% improvement for Libmodbus, a 153% improvement for IEC104, and a 4% improvement for Ethernet/IP, with an average improvement of 67%.

The improvement in coverage has a dual value: on one hand, it provides more comprehensive data flow information for format extraction, and on the other hand, it significantly enhances the vulnerability discovery capability of downstream security analysis tasks such as fuzz testing. The remarkable 1.53 improvement on the IEC104 protocol, in particular, demonstrates that our method has a unique advantage when dealing with complex protocol formats. Conversely, the smaller 4% improvement on the Ethernet/IP protocol reflects the characteristic of this protocol having relatively fixed execution paths.

F. RQ5: Comparison of Different LLMs

In ProbePRE, the LLM is responsible for interpreting the semantics of assembly code and deriving formulas to recover the true constraint values. To evaluate the impact of different LLMs on ProbePRE, we tested DeepSeek-v3 [47], Moonshot-Kimi-K2-Instruct [48], and Qwen3-235b-a22b-instruct-2507 [49].

As shown in Table V, using different LLMs does not lead to significant differences in format extraction results. This is because current open-source LLMs, trained on extensive code corpora, can reliably understand the semantics of input assembly snippets [50] [51] [52]. Moreover, the computation logic for field values is simple, with only a small amount of code involved, thus placing low demands on the model's capabilities. Compared to heuristic-based approaches for identifying code semantics, incorporating LLMs improves both the accuracy and the stability of ProbePRE's results.

TABLE V: Comparison of different LLMs on ProbePRE

Protocol	DeepSeek		KIMI		Qwen	
	#Format	Time(min)	#Format	Time(min)	#Format	Time(min)
S7comm [35]	13	3.38	13	3.43	13	3.26
Libmodbus [28]	9	1.40	9	1.36	9	1.30
IEC104 [37]	8	6.38	8	6.27	8	6.73
Ethernet/IP [40]	8	1.41	8	1.40	8	1.41
Average	9	2.73	9	2.88	9	2.93

G. RQ6: Effectiveness of ProbePRE in Real Settings

We applied the results of ProbePRE on the S7 protocol parser to Boofuzz [3] in order to evaluate its effectiveness in seed generation. Specifically, we adopted code coverage as the evaluation metric. Using Boofuzz, we generated 10,000 test cases under two settings: (i) with a single-format seed template, and (ii) with multi-format seed templates derived from ProbePRE. The generated test cases were then executed against the protocol parser, and the corresponding code coverage was measured. As shown in Table VI, test cases generated with ProbePRE-based seed templates achieved 40% coverage, representing a 232% improvement compared to those generated from a single-format template. The Figure 9 further illustrates the efficiency of input cases: while Boofuzz with a single format reached its maximum coverage within the first 10 test cases, Boofuzz with ProbePRE achieved its maximum coverage at the 1,710th test case. Relative to the total of 10,000 test cases, Boofuzz with ProbePRE reached higher coverage with substantially fewer effective test cases. This is because the extracted field constraints from ProbePRE guided Boofuzz to avoid unnecessary mutated field values.

These results demonstrate the effectiveness of ProbePRE's format extraction in downstream tasks. With the aid of multi-format protocol specifications, the fuzzer can both pass the parser's validation and reduce redundant mutations.

V. Discussions

In this section, we discuss the limitations of ProbePRE and outline potential future work.

TABLE VI: Comparison of line coverage (Boofuzz-SF denotes Boofuzz with a single-format seed, and Boofuzz-PP denotes Boofuzz with ProbePRE-generated multi-format seeds.)

Source Code File	Boofuzz-SF Line Cov.	Boofuzz-PP Line Cov.	Increase
cpp/server.cpp	62% (25 / 40)	62% (25 / 40)	0%
cpp/snap7.cpp	7% (22 / 315)	7% (22 / 315)	0%
core/s7_isotcp.cpp	27% (72 / 271)	35% (94 / 271)	30%
core/s7_server.cpp	8% (102 / 1328)	47% (625 / 1340)	512%
core/s7_server.h	100% (3 / 3)	100% (3 / 3)	0%
core/s7_text.cpp	13% (70 / 535)	38% (206 / 535)	194%
Sum	12% (294/2492)	40% (975/2492)	232%

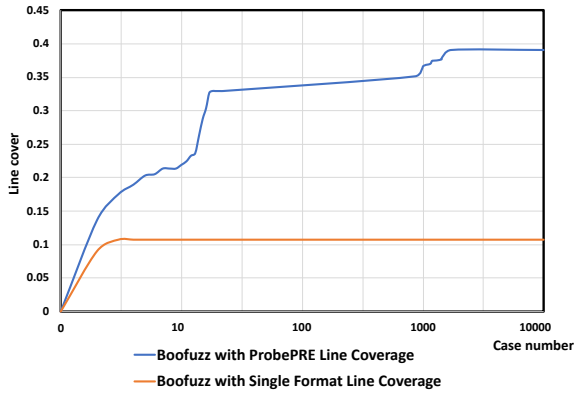

Fig. 9: Comparison of line coverage efficiency

Firstly, although ProbePRE, compared to traditional protocol reverse engineering methods, eliminates the dependency on input traffic datasets, the completeness and accuracy of protocol format extraction depend on the program's implementation. Different versions or programs under different scenarios may have varying levels of implementation completeness for the same protocol. If a program ignores a certain field, it means the program will not read or compute this field, and ProbePRE cannot identify this field from the execution trace; if a program ignores a certain format of the protocol, it means there is no processing path for that format in the program, and ProbePRE cannot extract the corresponding protocol format. Furthermore, in practice, we have found that some protocols use bit-level fields, and protocol handlers often determine path selection based on the values of these bit-level fields. Because it is not possible to trace data at bit granularity, ProbePRE, like traditional methods, cannot identify such fields, nor can it extract their constraint ranges. Then, ProbePRE currently cannot further extract protocol state machines based on protocol formats. This is because ProbePRE treats each input as a single state and does not consider the sequential order between different protocol states.

To address the aforementioned limitations of ProbePRE, we plan to implement a multi-program format extraction method that extracts formats from multiple program implementations of the same protocol, thereby obtaining a complete protocol format; by identifying program behaviors such as bitwise operations, we aim to achieve the segmentation and constraint extraction of bit-level fields.

VI. RELATEDWORK

Network-trace based protocol reverse engineering methods identify field boundaries and semantics by analyzing traffic datasets using statistical or deep learning techniques. For example, Binaryinferno [17] designs different detectors for various field types and uses Shannon entropy to detect field boundaries; Netplier [21] and Netzob [53] obtain similar packets through keyword clustering and apply sequence alignment techniques to intra-cluster packets to identify field boundaries; NEMESYS [19] proposes a method to extract formats from a single message, identifying field boundaries by calculating the bit similarity and bit similarity difference of adjacent bytes in a packet; Discoverer [54] recursively clusters messages of the same type; ProsegDL [55] stacks collected traffic into images and uses medical image processing models to analyze these images and infer field boundaries.

Execution-trace based protocol reverse engineering methods infer protocol formats by dynamically tracing the program's parsing process or statically analyzing source code. For example, Polyglot [54] pioneered the use of taint analysis to record the execution traces of input data within a program, identifying delimiters by analyzing comparison operations on the input data; Tupni [25] and AutoFormat [27], building on Polyglot, the former identifies fields by analyzing how instructions read input data, while the latter identifies field types by generating field trees; BinPRE [26] performs field segmentation through inter-instruction similarity analysis; Netlifter [56] analyzes program source code, outputting protocol formats by constructing an abstract format graph (AFG), performing local path-sensitive analysis, and reordering the AFG.

Both methods are constrained by the quality of the input traffic, suffering from poor generalization and difficulty in handling multiple protocol formats.

VII. CONCLUSION

We propose a novel protocol reverse engineering (PRE) method that reduces the reliance on large-scale, diverse traffic datasets typical of traditional approaches. Our method records execution traces via taint analysis, performs field segmentation and constraint extraction from these traces, and generates new traffic packets from identified field constraints to explore uncharted protocol parsing paths in protocol handlers. Consequently, ProbePRE does not necessitate the collection of extensive or highly diverse traffic datasets, instead leveraging protocol format information embedded within protocol handlers to extract all discernible formats. This approach overcomes the conventional over-reliance on such comprehensive traffic datasets and broadens the range of discoverable protocol formats. Experimental results demonstrate ProbePRE comprehensively extracts all protocol formats from protocol handlers, achieving superior coverage over traffic dataset-based approaches. Moreover, for field segmentation, ProbePRE generally outperforms traditional PRE tools.

REFERENCES

[1] D. Fang, Z. Song, L. Guan, P. Liu, A. Peng, K. Cheng, Y. Zheng, P. Liu, H. Zhu, and L. Sun, "Ics3fuzzer: A framework for discovering protocol implementation bugs in ics supervisory software by fuzzing," in *Proceedings of the 37th Annual Computer Security Applications Conference*, 2021, pp. 849–860.

[2] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Proceedings 11*. Springer, 2015, pp. 330–347.

[3] "GitHub - jtpereyda/boofuzz: A fork and successor of the Sulley Fuzzing Framework — github.com," https://github.com/jtpereyda/boofuzz, [Accessed 26-05-2025].

[4] "GitHub - MozillaSecurity/peach: Peach is a fuzzing framework which uses a DSL for building fuzzers and an observer based architecture to execute and monitor them. — github.com," https://github.com/MozillaSecurity/peach, [Accessed 26-05-2025].

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

[6] P. Liu, Y. Zheng, Z. Song, D. Fang, S. Lv, and L. Sun, "Fuzzing proprietary protocols of programmable controllers to find vulnerabilities that affect physical control," *Journal of Systems Architecture*, vol. 127, p. 102483, 2022.

[7] S. A. Bakhsh, M. A. Khan, F. Ahmed, M. S. Alshehri, H. Ali, and J. Ahmad, "Enhancing iot network security through deep learning-powered intrusion detection system," *Internet of Things*, vol. 24, p. 100936, 2023.

[8] "Wireshark · Go Deep — wireshark.org," https://www.wireshark.org/, [Accessed 26-05-2025].

[9] F. Wei, H. Li, Z. Zhao, and H. Hu, "{xNIDS}: Explaining deep learning-based network intrusion detection systems for active intrusion responses," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4337–4354.

[10] J. Narayan, S. K. Shukla, and T. C. Clancy, "A survey of automatic protocol reverse engineering tools," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, pp. 1–26, 2015.

[11] Y. Huang, H. Shu, F. Kang, and Y. Guang, "Protocol reverse-engineering methods and tools: A survey," *Computer Communications*, vol. 182, pp. 238–254, 2022.

[12] J. De Ruiter and E. Poll, "Protocol state fuzzing of {TLS} implementations," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 193–206.

[13] B. Yu, P. Wang, T. Yue, and Y. Tang, "Poster: Fuzzing iot firmware via multi-stage message generation," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 2525–2527.

[14] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. De Ruiter, K. Sagonas, and J. Somorovsky, "Analysis of {DTLS} implementations using protocol state fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2523–2540.

[15] Q. Shi, X. Xu, and X. Zhang, "Extracting protocol format as state machine via controlled static loop analysis," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7019–7036.

[16] C. Y. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song, "{MACE}:{Model-inference-Assisted} concolic exploration for protocol and vulnerability discovery," in *20th USENIX Security Symposium (USENIX Security 11)*, 2011.

[17] J. Chandler, A. Wick, and K. Fisher, "Binaryinferno: A semantic-driven approach to field inference for binary message formats." in *NDSS*, 2023.

[18] A. Hahn, "Operational technology and information technology in industrial control systems," *Cyber-security of SCADA and other industrial control systems*, pp. 51–68, 2016.

[19] S. Kleber, H. Kopp, and F. Kargl, "{NEMESYS}: Network message syntax reverse engineering by analysis of the intrinsic structure of individual messages," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[20] Z. Luo, K. Liang, Y. Zhao, F. Wu, J. Yu, H. Shi, and Y. Jiang, "Dynpre: Protocol reverse engineering via dynamic inference," in *Proc. NDSS*, 2024, pp. 1–18.

[21] Y. Ye, Z. Zhang, F. Wang, X. Zhang, and D. Xu, "Netplier: Probabilistic network protocol reverse engineering from message traces." in *NDSS*, 2021.

[22] S. Zhao, S. Yang, Z. Wang, Y. Liu, H. Zhu, and L. Sun, "Crafting binary protocol reversing via deep learning with knowledge-driven augmentation," *IEEE/ACM Transactions on Networking*, 2024.

[23] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 621–634.

[24] N. Nystrom, M. R. Clarkson, and A. C. Myers, "Polyglot: An extensible compiler framework for java," in *International Conference on Compiler Construction*. Springer, 2003, pp. 138–152.

[25] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 391–402.

[26] J. Jiang, X. Zhang, C. Wan, H. Chen, H. Sun, and T. Su, "Binpre: Enhancing field inference in binary analysis based protocol reverse engineering," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 3689–3703.

[27] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution." in *NDSS*, vol. 8, 2008, pp. 1–15.

[28] "GitHub - stephane/libmodbus: A Modbus library for Linux, Mac OS, FreeBSD and Windows — github.com," https://github.com/stephane/libmodbus, [Accessed 25-05-2025].

[29] "Pin - A Dynamic Binary Instrumentation Tool — intel.cn," https://www.intel.cn/content/www/cn/zh/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html, [Accessed 26-05-2025].

[30] "GitHub - nesclab/ProtocolTaint: A protocol reverse engineering tool for industrial binary protocol based on pin tool — github.com," https://github.com/nesclab/ProtocolTaint, [Accessed 26-05-2025].

[31] DeepSeek, "Deepseek: Cutting-edge ai research and development," https://www.deepseek.com/, 2024, accessed: 2024-06-20.

[32] "IDA Pro — hex-rays.com," https://hex-rays.com/ida-pro, [Accessed 26-05-2025].

[33] "Scapy — scapy.net," https://scapy.net/, [Accessed 26-05-2025].

[34] "Welcome to Python.org — python.org," https://www.python.org/, [Accessed 26-05-2025].

[35] "Snap7 - Browse Files at SourceForge.net — sourceforge.net," https://sourceforge.net/projects/snap7/files/, [Accessed 24-05-2025].

[36] "GitHub - cwalter-at/freemodbus: BSD licensed MODBUS RTU/ASCII and TCP slave — github.com," https://github.com/cwalter-at/freemodbus, [Accessed 26-05-2025].

[37] "GitHub - scadapy/iec104server — github.com," https://github.com/scadapy/iec104server, [Accessed 26-05-2025].

[38] "BACnet Protocol Stack — sourceforge.net," https://sourceforge.net/projects/bacnet/, [Accessed 26-05-2025].

[39] "GitHub - dnp3/opendnp3: DNP3 (IEEE-1815) protocol stack. Modern C++ with bindings for .NET and Java. — github.com," https://github.com/dnp3/opendnp3, [Accessed 26-05-2025].

[40] "GitHub - EIPStackGroup/OpENer: OpENer is an EtherNet/IP stack for I/O adapter devices. It supports multiple I/O and explicit connections and includes objects and services for making EtherNet/IP-compliant products as defined in the ODVA specification. — github.com," https://github.com/EIPStackGroup/OpENer, [Accessed 26-05-2025].

[41] "GitHub - rovinbhandari/FTP: Implementation of a simple FTP client and server — github.com," https://github.com/rovinbhandari/FTP, [Accessed 26-05-2025].

[42] "ICS-Security-Tools/pcaps/EthernetIP/EthernetIP-CIP.pcap at master · ITI/ICS-Security-Tools — github.com," https://github.com/ITI/ICS-Security-Tools/blob/master/pcaps/EthernetIP/EthernetIP-CIP.pcap, [Accessed 26-05-2025].

[43] "ICS-pcap/IEC 60870/iec104/iec104.pcap at master · automayt/ICS-pcap — github.com," https://github.com/automayt/ICS-pcap/blob/master/IEC%2060870/iec104/iec104.pcap, [Accessed 26-05-2025].

[44] "SampleCaptures - Wireshark Wiki — wiki.wireshark.org," https://wiki.wireshark.org/samplecaptures#iec-60870-5-104, [Accessed 26-05-2025].

[45] "Public PCAP files for download — netresec.com," https://www.netresec.com/?page=PcapFiles, [Accessed 26-05-2025].

[46] "SampleCaptures - Wireshark Wiki — wiki.wireshark.org," https://wiki.wireshark.org/SampleCaptures#s7comm-s7-communication, [Accessed 24-05-2025].
[47] DeepSeek-AI, "Deepseek-v3: Open large language model," https://github.com/deepseek-ai/DeepSeek-V3, 2024.
[48] M. AI, "Moonshot kimi k2 instruct model," https://kimi.moonshot.cn, 2024.
[49] A. Cloud, "Qwen3-235b-a22b-instruct model," https://huggingface.co/Qwen, 2025.
[50] W. Sun, Y. Miao, Y. Li, H. Zhang, C. Fang, Y. Liu, G. Deng, Y. Liu, and Z. Chen, "Source code summarization in the era of large language models," *arXiv preprint arXiv:2407.07959*, 2024.
[51] X. Shang, S. Cheng, G. Chen, Y. Zhang, L. Hu, X. Yu, G. Li, W. Zhang, and N. Yu, "How far have we gone in binary code understanding using large language models," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2024, pp. 1–12.
[52] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.
[53] "Netzob: Reverse Engineering Communication Protocols — netzob.org," https://netzob.org/, [Accessed 26-05-2025].
[54] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic protocol reverse engineering from network traces." in *USENIX Security Symposium*. Boston, MA, USA, 2007, pp. 1–14.
[55] S. Zhao, J. Wang, S. Yang, Y. Zeng, Z. Zhao, H. Zhu, and L. Sun, "Prosegdl: Binary protocol format extraction by deep learning-based field boundary identification," in *2022 IEEE 30th International Conference on Network Protocols (ICNP)*. IEEE, 2022, pp. 1–12.
[56] Q. Shi, J. Shao, Y. Ye, M. Zheng, and X. Zhang, "Lifting network protocol implementation to precise format specification with security applications," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1287–1301.