BinPRE: Enhancing Field Inference in Binary Analysis Based Protocol Reverse Engineering

Jiayi Jiang
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
Shanghai, China
jyjiangsunny@gmail.com

Xiyuan Zhang
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
Shanghai, China
zxy6538@gmail.com

Chengcheng Wan
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
Shanghai, China
ccwan@sei.ecnu.edu.cn

Haoyi Chen
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
Shanghai, China
haoyichense@gmail.com

Haiying Sun
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
Shanghai, China
hysun@sei.ecnu.edu.cn

Ting Su
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
Shanghai, China
tsu@sei.ecnu.edu.cn

Abstract

Protocol reverse engineering (PRE) aims to infer the specification of network protocols when the source code is not available. Specifically, field inference is one crucial step in PRE to infer the field formats and semantics. To perform field inference, binary analysis based PRE techniques are one major approach category. However, such techniques face two key challenges — (1) the format inference is fragile when the logics of processing input messages may vary among different protocol implementations, and (2) the semantic inference is limited by inadequate and inaccurate inference rules.

To tackle these challenges, we present BinPRE, a binary analysis based PRE tool. BinPRE incorporates (1) an instruction-based semantic similarity analysis strategy for format extraction; (2) a novel library composed of atomic semantic detectors for improving semantic inference adequacy; and (3) a cluster-and-refine paradigm to further improve semantic inference accuracy. We have evaluated BinPRE against five existing PRE tools, including Polyglot, AutoFormat, Tupni, BinaryInferno and DynPRE. The evaluation results on eight widely-used protocols show that BinPRE outperforms the prior PRE tools in both format and semantic inference. BinPRE achieves the perfection of 0.73 on format extraction and the F1-score of 0.74 (0.81) on semantic inference of types (functions), respectively. The field inference results of BinPRE have helped improve the effectiveness of protocol fuzzing by achieving 5~29% higher branch coverage, compared to those of the best prior PRE tool. BinPRE has also helped discover one new zero-day vulnerability, which otherwise cannot be found.

1 INTRODUCTION

Protocol reverse engineering (PRE) aims to infer the specifications (*e.g.*, field formats, semantics, and state machines) of network protocols, assuming only the protocol messages and/or the program binaries implementing the protocols are available [39, 53]. The inferred protocol specifications can be applied in many scenarios like protocol fuzzing (*e.g.*, Peach [15] and Boofuzz [1]), and network traffic analysis and auditing (*e.g.*, Wireshark [22]). Therefore, building effective PRE techniques is important.

Specifically, *field inference* is one important and necessary step of PRE. It includes two major relevant tasks: (1) *format extraction*, *i.e.*, inferring the field boundaries of the input messages, and (2) *semantic inference*, *i.e.*, inferring the corresponding semantics of the fields identified from (1). Moreover, field inference is the requisite for inferring protocol state machines [35]. Thus, in this paper, we focus on improving the effectiveness of field inference because of the importance in its own right.

In the literature, there are two major approaches to achieving field inference, *i.e.*, network-traffic based (NetT-based) [31, 37, 42, 49, 65] and execution-trace based (ExeT-based) PRE techniques [27, 30, 36, 46]. The NetT-based PRE techniques take static network traces as input and perform statistical analysis to mine the format characteristics exhibited in the traces. However, their inference accuracy relies on the availability of high-quality network traces that contain diverse protocol messages, which are usually difficult to obtain in practice. The ExeT-based PRE techniques (also termed as *binary analysis based* PRE techniques throughout this paper), on the other hand, are resilient to the quality of input messages. They can obtain rich runtime semantics by monitoring the executed instructions of the protocol implementations. However, despite the rich runtime semantics, these techniques in practice still face two key challenges in achieving effective field inference.

The *first* challenge is that the analysis strategies of format extraction in classic PRE techniques [30, 36, 46] are fragile to the actual protocol implementations. Since these strategies are usually implemented based on some heuristic patterns of the executed instructions, they may become fragile when the logics of processing input messages vary among different protocols. As a result, these analysis strategies would significantly suffer from the *over-segmentation* errors (*i.e.*, introducing spurious field boundaries) and the *under-segmentation* errors (*i.e.*, missing true field boundaries). These errors degrade the accuracy of format extraction, which we will illustrate in Section 2.2. Even worse, these errors would further undermine the subsequent task of semantic inference. Because semantic inference would fail on the inaccurate field segmentation.

The *second* challenge is that the semantic inference abilities of the classic PRE techniques [30, 36, 46] are limited due to the inadequate and inaccurate inference rules. For example, *types* and *functions* are the two important field semantics, commonly used by network traffic auditing tools like Wireshark [22] and protocol fuzzing tools like Peach [15] and Boofuzz [1]. Take Boofuzz as an example, it by default supports *five* and *six* major categories of types and functions, respectively. However, the classic PRE techniques cannot infer types and only infer four categories of functions with low accuracy. For example, our investigation reveals that these techniques can only infer the *command* field with the F1-score of 0.14, the *delimiter* field with the F1-score of 0.22, and the *length* field with the F1-score of 0.56. The inaccurate results of semantic inference may further affect the downstream applications of PRE.

In this paper, we introduce two *key* ideas to tackle the preceding two challenges, respectively. To mitigate the first challenge, we introduce an *instruction-based semantic similarity analysis strategy* to enhance the classic format extraction. Our *key* insight is that *the bytes from the same field should have similar semantics*. Specifically, we use *the sequences of instruction operators* accessing these bytes as the proxy of the semantics. In this way, the bytes accessed by similar sequences of instruction operators are merged into the same field. This strategy is simple yet effective in tackling segmentation errors. Because it is resilient to the logic of processing input messages in the different protocol implementations.

To mitigate the second challenge, we build a novel library composed of atomic semantic detectors to improve the adequacy and accuracy of semantic inference. More importantly, based on the preceding inference results, we introduce a *cluster-and-refine strategy* to further improve the accuracy of semantic inference. The key idea is that *the messages with similar formats can offer useful contextual information (*e.g.*, the data values of the same byte offsets) to refine the inference results*. Specifically, we cluster the messages with similar formats based on the results of our format extraction.

We implemented a binary analysis based PRE tool named BinPRE to support the application of our ideas. We evaluated it against 5 state-of-the-art PRE tools on 8 popular extant protocols. The experiment results show that BinPRE outperforms prior PRE tools in both format extraction and semantic inference. For format extraction, BinPRE achieves the perfection score and the F1-score of (0.73, 0.86) for format extraction, while the classic PRE tools Polyglot [30], AutoFormat [46], Tupni [36], BinaryInferno [31], and DynPRE [49] achieve the perfection scores and F1-scores of (0.60, 0.66), (0.59, 0.64), (0.49, 0.68), (0.13, 0.35), and (0.24, 0.54), respectively. During format extraction, BinPRE reduces 63~70% segmentation errors *w.r.t.* these prior PRE tools on 8 protocols. For semantic inference, BinPRE achieves the average precision and recall values of (0.72, 0.77) and (0.77, 0.91) for inferring field types and functions, respectively. Moreover, the field inference results of BinPRE have helped improve the effectiveness of protocol fuzzing by achieving 5~29% higher branch coverage, compared to those of the best prior PRE tool, and discover one new zero-day vulnerability.

Finally, it is important to note that *almost all* the prior binary analysis based PRE tools are not publicly available. We took significant efforts to re-implement the field inference strategies of these prior PRE tools (including Polyglot, AutoFormat and Tupni), and carefully validated our implementations by replicating the experiments of these tools. Thus, we believe one of our contributions is to make these implementations readily and publicly available for fair comparison and replication.

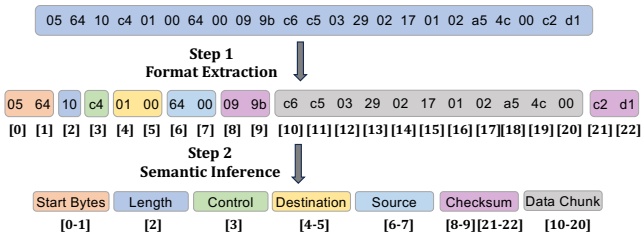

Figure 1: An example message of `DNP3.0` protocol (the fields with different semantics are annotated by different colors).

In this paper, we have made the following contributions:

- We introduce an instruction-based semantic similarity analysis strategy to enhance the classic field segmentation and effectively mitigate the improper segmentation errors in format extraction (Section 3.3).
- We build a novel library of atomic semantic detectors to improve the adequacy of semantic inference (Section 3.4) and introduce a *cluster-and-refine* paradigm to improve the accuracy of semantic inference (Section 3.5).
- We implement a tool BinPRE to support the application of our ideas. BinPRE outperforms the prior PRE tools in field inference and demonstrates its usefulness in improving such downstream tasks as protocol fuzzing (Section 4).
- We have made BinPRE and the re-implementations of prior PRE tools publicly available at *https://github.com/BinPRE/BinPRE* to facilitate replication of our experiments and benefit the community for studying binary analysis based PRE techniques.

2 Background and Challenges

In this section, we give the necessary background of protocol reverse engineering, especially field inference, and illustrate the challenges of classic PRE techniques for field inference.

2.1 Background and Definitions

The main goal of PRE is to infer the protocol fields and the finite state machines of protocols. This paper focuses on field inference, as it serves as the fundamental pillar of PRE. In protocol reverse engineering, *field inference* includes two major tasks: (1) *format extraction*, *i.e.*, inferring the field boundaries, and (2) *semantic inference*, *i.e.*, inferring the corresponding semantics (*e.g.*, *type* and *function*) of the fields. For example, Figure 1 shows a raw binary message in hexadecimal values from `DNP3.0`, a popular communication protocol used in industrial control systems. To achieve field inference, we need to perform format extraction (Step 1 in Figure 1), and semantic inference (Step 2 in Figure 1). Take the second byte (*i.e.*, `0x10`) in this message as an example, an ideal field inference should segment this byte into a field, and infer that this field's type is *integer* and its function is the *length* of the message. In the setting of binary analysis, the tasks of *format extraction* and *semantic inference* can be formulated as follows.

Listing 1: Automatak-DNP3 source code snippet.

```
1  bool ShiftableBuffer::Sync(){
2  //2 bytes in f0,1 are separately accessed -> Over-seg. errors in the three prior tools
3     while (this->NumbytesRead()>1){//f0,1
4        if (this->ReadBuffer()[0]==0x05
5           && this->ReadBuffer()[1]==0x64){...}}}
6
7  bool LinkFrame::ValidateBodyCRC(const uint8_t* pBody, size_t length){
8     length = this->ReadBuffer()[2];//f2,2
9     while (length > 0){
10       size_t max = LPDU_DATA_BLOCK_SIZE;
11       size_t num = (length <= max) ? length : max;
12       if (CRC::IsCorrectCRC(pBody, num)){//f10,20, f21,22
13          pBody += (num + 2);//Under-seg. errors in Polyglot
14          length -= num;}...}}
15
16 bool CRC::IsCorrectCRC(const uint8_t* input, size_t length){
17    ser4cpp::rseq_t buffer(input + length, 2);
18    uint16_t crcValue;
19    ser4cpp::UInt16::read_from(buffer, crcValue);//f21,22, f8,9
20    uint16_t CalcCrcValue = 0;
21 //11 bytes in f10,20 are separately accessed -> Over-seg. errors in three prior tools
22 //8 bytes in f0,7 are accessed in a loop -> Under-seg. errors in Tupni
23    for (uint32_t i = 0; i < length; ++i){//f10,20, f0,7
24       uint8_t index = (CalcCrcValue ^ input[i]) & 0xFF;
25       CalcCrcValue = crcTable[index] ^ (CalcCrcValue >> 8);
26    }
27    CalcCrcValue = ~CalcCrcValue;
28    return CalcCrcValue == crcValue;}
```

Format Extraction. Let M be a raw message in bytes, *i.e.*, $M = [b_0, \cdots, b_i, \cdots, b_m]$ (b_i denotes one data byte). Let $INST_M$ be the trace of the executed binary instructions when the program binary processes M, *i.e.*, $INST_M = [inst_1, \cdots, inst_j, \cdots, inst_n]$ ($inst_j$ denotes one executed binary instruction). By leveraging such classic program analysis as taint analysis [41], binary analysis based PRE techniques can track which bytes in M have been accessed by which instructions in $INST_M$. Based on such information, such techniques can extract field formats. For example, one common strategy used by prior PRE techniques [30, 36, 46] is that, if the instruction $inst_j$ accesses the consecutive bytes $[b_k, \cdots, b_l]$ ($0 \le k \le l \le m$) in M, the bytes $[b_k, \cdots, b_l]$ is inferred as a field f. As a result, the field boundaries are inferred between the b_{k-1}-th and b_k-th bytes, and the b_l-th and b_{l+1}-th bytes in M. This field f is composed of the consecutive bytes from k-th to l-th in the message M. Here, we can represent f as $f_{k,l} = M[k, l]$. *Format extraction* is such a process of segmenting M into a number of fields.

Semantic Inference. Based on the results of format extraction, *semantic inference* is the process of inferring the semantics of these extracted fields. Specifically, *type* and *function* are the two important field semantic attributes, which are commonly used by protocol fuzzing tools (*e.g.*, Peach [15] and Boofuzz [1]). The prior PRE techniques infer the field semantics based on different strategies [30, 31, 36, 49]. For example, in Figure 1, assuming the field $f_{2,2}$ is identified, one common strategy used by prior work [30] will infer the function of this field to be *Length* based on some behavioral features of $INST_M$.

2.2 Challenges of Format Extraction

Format extraction infers the field boundaries of input messages. However, the strategies of format extraction in classic PRE techniques [30, 36, 46] are fragile to different protocol implementations. As a result, these strategies suffer from the *over-segmentation* errors (*i.e.*, introducing spurious field boundaries) and *under-segmentation* errors (*i.e.*, missing true field boundaries). We use the code snippet (Listing 1) from `Automatak-DNP3` [13] as example, which is an implementation of the protocol `DNP3.0`. It processes the input message shown in Figure 1.

Over-segmentation errors. The classic PRE techniques (*e.g.*, Polyglot [30], AutoFormat [46], and Tupni [36]) assume that different fields have independent data flow from each other —the bytes belonging to different fields are rarely used in the same instruction. Thus, they adopt a common strategy for format extraction: *the (consecutive) bytes accessed by one instruction are within the same field* (*cf.* Section 6 in [30], Section 3.2.1 in [46], Section 3.3 in [36]). However, in real-world protocol implementations, the bytes belonging to one field may still be accessed by different instructions, thus leading to over-segmentation errors. For example, the two bytes of field $f_{0,1}$ (*i.e.*, `0x05` and `0x64`) in the input message, denoting the starting bytes, are accessed by two different instructions (corresponding to lines 4 and 5 in Listing 1). As a result, the classic PRE techniques (like Polyglot, AutoFormat, and Tupni) will over-segment field $f_{0,1}$ into two fields $f_{0,0}$ and $f_{1,1}$. As another example, the eleven bytes of field $f_{10,20}$, denoting the data chunk of the input message, are accessed by different instructions (corresponding to line 24 in the loop of lines 23-26 in Listing 1). As a result, the classic PRE techniques will over-segment $f_{10,20}$ into eleven different fields.

Under-segmentation errors. The classic PRE techniques also implement other heuristic strategies, which may lead to *under-segmentation* errors. For example, Polyglot infers the boundaries of variable-length fields based on the *length* fields. Polyglot correctly infers field $f_{2,2}$ as the *length* field, and identifies its pointer relationship with $f_{10,20}$ and $f_{21,22}$ (corresponding to lines 12-14). Therefore, these two fields are erroneously merged into one variable-length field. As another example, the function `IsCorrectCRC` computes the CRC value of the contents in the fields $f_{0,1}$, $f_{2,2}$, $f_{3,3}$, $f_{4,5}$, $f_{6,7}$ (corresponding to lines 23-26) and checks against the checksum stored in the field $f_{8,9}$ (read by line 19). Since Tupni merges the consecutive bytes processed by "mostly" the same instructions in a loop into one field (*cf.* Section 3.4~3.5 in [36]), the fields $f_{0,1}$, $f_{2,2}$, $f_{3,3}$, $f_{4,5}$, $f_{6,7}$ are erroneously merged into one field.

Appendix D (Figure 13) gives the format extraction results of the three classic PRE techniques (Polyglot, AutoFormat, Tupni) when processing the input message in Figure 1.

> **Challenge-1**: The classic binary analysis based PRE techniques suffer from the errors of over-segmentation and under-segmentation in performing format extraction. This challenge may affect the subsequent task of semantic inference.

To mitigate the preceding challenge, we introduce an *instruction-based semantic similarity analysis strategy*, which is resilient against the actual protocol implementations (detailed in Section 3.3).

2.3 Challenges of Semantic Inference

Semantic Inference aims to infer the semantics of the fields in a message. For example, *type* and *function* are the two important field semantics, which indicate the data type and the meaning of data of a field. However, the semantic inference abilities of the classic

Table 1: F1-scores achieved by the five state-of-the-art PRE tools in performing semantic inference.

Tool	*Type*					*Function*					
	Static	Integer	Group	Bytes	String	Command	Length	Delim	Checksum	Aligned	Filename
Polyglot	-	-	-	-	-	0.14	0.56	0.22	-	-	-
AutoFormat	-	-	-	-	-	-	-	-	-	-	-
Tupni	-	-	-	-	-	-	0.56	-	1.0	-	-
BinaryInferno	-	-	-	0	-	-	0.50	-	-	-	-
DynPRE	-	-	-	-	-	0.08	-	-	-	-	-

* "-" denotes that the corresponding semantic (*type* or *function*) is not supported.
* F1-score = 2×precision×recall/(precision+recall)

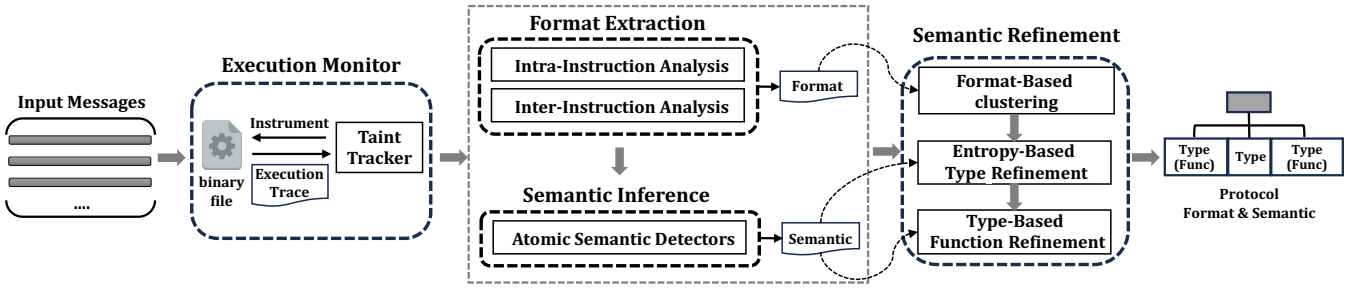

Figure 2: BinPRE overview

PRE techniques are limited due to the inadequate and inaccurate inference rules. It affects such the downstream tasks of PRE as network traffic auditing and protocol fuzzing.

To illustrate the limitations, we assess how effective the classic PRE techniques are in supporting protocol fuzzing, one prominent downstream task of PRE, from the perspective of the adequacy and accuracy of semantic inference. Specifically, we selected Boofuzz [1], a popular generation-based protocol fuzzing tool, as the reference, and investigated the semantic inference abilities of three classic ExeT-based PRE tools (Polyglot, AutoFormat, and Tupni) and two NetT-based PRE tools (BinaryInferno and DynPRE).

Boofuzz by default supports five and six major categories of types and functions [2]. Table 1 summarizes the semantic inference adequacy and accuracy of the prior PRE tools. We can see that none of Polyglot, AutoFormat, and Tupni infers the five types, while Polyglot only infers three functions and Tupni infers two functions. The two NetT-based PRE tools BinaryInferno and DynPRE, on the other hand, only infer one function, respectively. In Appendix A, we give more details of our investigation against Boofuzz as well as another popular protocol fuzzing tool Peach [16]. Moreover, by replicating the semantic inference of the ExeT-based PRE tools, we find that the ExeT-based PRE tools can only infer the *command* field with an F1-score of 0.14, the *delimiter* field with an F1-score of 0.22, and the *length* field with an F1-score of 0.56 (detailed in Section 4). This indicates that the semantic inference accuracy of the prior PRE tools is far from satisfactory. It may affect such downstream tasks as protocol fuzzing and network auditing.

Challenge-2: The semantic inference abilities of the classic binary analysis based PRE techniques are limited due to the inadequate and inaccurate inference rules. This challenge may affect supporting the downstream tasks of PRE.

To mitigate the preceding challenge, we build a novel library composed of atomic semantic detectors to improve inference adequacy and introduce a *cluster-and-refine* paradigm to improve the accuracy of semantic inference (detailed in Sections 3.4 and 3.5).

3 BinPRE DESIGN

To tackle the challenges in field inference, we introduce BinPRE, a binary analysis based PRE tool that accurately and comprehensively reverse engineers protocol formats and semantics.

3.1 Overview

As shown in Figure 2, BinPRE comprises four major modules: *Execution Monitor*, *Format Extraction*, *Semantic Inference*, and *Semantic Refinement*. Given a protocol message as input, the *Execution Monitor* module tracks its parsing process and records the execution information of the server's instrumented binary. The *Format Extraction* module then analyzes the recorded execution information and extracts field format through instruction-based semantic similarity analysis. Based on the inferred format and behavioral features within execution information, the *Semantic Inference* module utilizes a library of atomic semantic detectors to identify the semantics of each field. Finally, to further improve semantic inference, the *Semantic Refinement* module clusters protocol messages to capture contextual features and refines the semantic inference results.

3.2 Execution Monitor

The *Execution Monitor* module incorporates taint analysis [30, 46, 50] to capture execution information, including data-flow and control-flow when the server parses protocol messages. It taints protocol message data at the byte level and captures their propagation traces to understand how data is processed by the server program.

Specifically, the *Execution Monitor* records detailed execution information for each tainted byte, including the taint propagation path, changes in taint value (*i.e.*, the values of tainted bytes), and the instructions that manipulate the bytes. It instruments the binary executable of the server at three levels: function, basic block, and instruction. The first two levels capture the control-flow propagation, and the latter tracks the data-flow propagation. It pays extra attention to branch conditions and loop iterations, which reflect the dependencies between fields. The outputs from all these levels, referred to as behavioral features, are then passed to the *Format Extraction* module and the *Semantic Inference* module.

3.3 Format Extraction

Given a protocol message and its taint analysis results, BinPRE uses *an instruction-based semantic similarity analysis strategy* to extract protocol fields and determines field boundaries. Our key insight is that the bytes from the same field should have similar semantics. Specifically, the semantics could be approximated by the sequences of instruction operators accessing these bytes.

Algorithm 1 implements our key insight to achieve format extraction. The algorithm takes as input an input message M and the instruction set $INST_M$ composed of instructions accessing the bytes in M. Recall that $INST_M$ is obtained from the *Execution Monitor* module. The *Format Extraction* module first conducts *intra-instruction analysis* (lines 3–8) to identify the adjacent bytes accessed by the same instruction. It iterates over all the instructions in $INST_M$ to obtain the field candidates accessed by each instruction. Specifically, for each instruction, BinPRE extracts the sequence of bytes from execution information and treats them as a candidate field (lines 4–5). It maintains a set of the extracted field candidates S and adds each new identified field candidate to S (line 6).

To further determine the field boundaries, the *Format Extraction* module then performs *inter-instruction analysis* (lines 9–19). It iterates over all the obtained field candidates to examine the semantic similarity between adjacent candidate field candidates (line 11). BinPRE uses the instruction operators as the proxy of semantics of assembly instructions. Because the instruction operators reflect the core functionalities of these instructions. In practice, BinPRE extracts the operator sequences accessing each field candidate and adopts the Needleman-Wunsch (NW) algorithm [45] (implemented in the procedure SemanticSimilar) to calculate the sequence similarly (lines 21-25). Note that Needleman-Wunsch (NW) algorithm is a classic algorithm for sequence similarity matching scenarios. When the two operator sequences are similar, their semantics are assumed to be similar. If the two adjacent field candidates have similar semantics, they will be merged (lines 12–13).

Specifically, given two adjacent field candidates f_i and f_j, their semantic similarity score NW_o is calculated as

$$NW_o(i,j) = \max \begin{cases} NW_o(i-1,j-1) + C(I[i],J[j]), \\ NW_o(i-1,j) + \texttt{GAP\_SCORE}, \\ NW_o(i,j-1) + \texttt{GAP\_SCORE} \end{cases}$$

$$C(a,b) = \begin{cases} \texttt{MA\_SCORE}, & if \quad OP(a) = OP(b) \\ \texttt{MISMA\_SCORE}, & otherwise \end{cases}$$

where $OP(a)$ indicates the operator of instruction a; $C(a,b)$ compares whether the operators of the two instructions, *i.e.*, a and b, are the same; GAP_SCORE (the default value is set as -2) is for penalizing discontinuities between operator sequences; MA_SCORE (the default value is set as 1) is for encouraging matches of the same operators; and MISMA_SCORE (the default value is set as -1) is for penalizing mismatches of different operators. Following the standard convention [54], the semantic similarity threshold is set as 0.8. This threshold has been justified by our further evaluation on the effect of different thresholds (detailed in Appendix C). This threshold value ensures a high degree of consistency in deciding the semantic similarity. By using NW_o, BinPRE merges semantically similar fields (lines 12–13). Finally, it removes all the null fields in L and outputs F as the final format extraction results (lines 17–18).

Algorithm 1: Format extraction.

```
   Input: M: an input message, INST_M: instructions accessing the bytes in M
   Output: F: the fields extracted from M
1  Procedure FormatExtraction(M, INST_M):
2      S := {}                                   ▷S is the set of field candidates
3      for inst_i in INST_M do
4          bs_inst_i := ExtractByteSeq(inst_i)
5          f_inst_i := BytesToField(bs_inst_i)
6          S := S ⋃ {f_inst_i}
7      end
8      L := SortWithOffset(S)                    ▷L is a list of field candidates
9      i := 0
10     while i < (len(L) − 1) do
11         if SemanticSimilar (L[i], L[i + 1]) then
12             L[i + 1] := MergeFields(L[i], L[i + 1])
13             L[i] := Null
14         end
15         i := i + 1
16     end
17     F := RemoveNull(L)
18     return F                                  ▷F is a list of extracted fields
19 End Procedure
20 Procedure SemanticSimilar(f_i, f_j):
21     I := {inst ∈ INST_M | ExtractInstSeq(inst) ∈ f_i}
22     J := {inst ∈ INST_M | ExtractInstSeq(inst) ∈ f_j}
23     m, n := len(I), len(J)
24     similarity := NW_o(m, n)/max(m, n)
25     return similarity > 0.8
26 End Procedure
```

To further illustrate the mechanism of *Format Extraction* module, we provide two concrete examples:

Example-1. Take the first two bytes $[b_0, b_1]$ of the input message in Figure 1 as an example. During the intra-instruction analysis, BinPRE obtains the two field candidates $f_{0,0}$ and $f_{1,1}$ from these two bytes. In Figure 3, the gray boxes show the sequences of instruction operators accessing $f_{0,0}$ and $f_{1,1}$, respectively. Then, during the inter-instruction analysis, since the sequences of instruction operators accessing $f_{0,0}$ and $f_{1,1}$ are identical (*i.e.*, $[movzx, cmp]$), they have the similar semantic (*i.e.*, the similarity value is 1.0). As a result, BinPRE merges them into one single field $f_{0,1}$.

Example-2. Take the bytes $[b_{10}, \cdots, b_{22}]$ of the input message in Figure 1 as an example. As shown in Figure 4, all the bytes $[b_{10}, \cdots, b_{20}]$ are processed in the same loop with the identical sequences of instruction operators $[cmp, movzx, xor, mov, movzx]$, while the bytes $[b_{21}, b_{22}]$ are accessed by a different sequence of instruction operators $[or, movzx, cmp]$. As a result, BinPRE assume

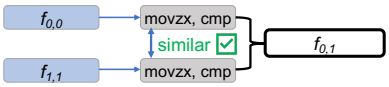

Figure 3: The process of analyzing a field that is prone to the over-segmentation error.

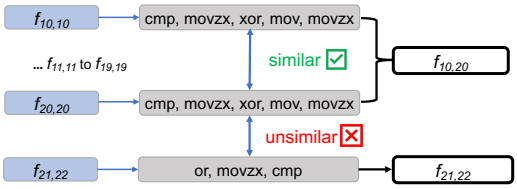

Figure 4: The process of analyzing a field that is prone to the under-segmentation error.

the bytes $[b_{10}, \cdots, b_{20}]$ and $[b_{21}, b_{22}]$ have different semantics and thus separates these bytes into two separate fields: $f_{10,20}$ and $f_{21,22}$.

3.4 Semantic Inference

With behavioral features, the *Semantic Inference* module infers semantic types and functions of each field. Guided by the execution information from *Execution Monitor*, we construct a novel library of atomic semantic detectors for inferring each field's *type* and *function*. The semantic detectors utilize two types of information: (1) $I(f)$: the set of instructions accessing field f; and (2) $V(f)$: the value of f. Note that, all the fields in an input message should have semantic types, and only some fields have semantic functions.

Overall, the *Semantic Inference* module supports five(six) semantic types(functions), which are summarized in Table 2. These semantics align with those supported by BooFuzz (detailed in Section 2.3).

Static type has a fixed value and location in the message, regardless of the message content and context. Its primary purpose is to validate the message and indicate the flags related to protocols, sessions, or messages (*e.g.*, Protocol Version ID). BinPRE regards a field as Static only when 1) the field is compared to a fixed value and the result yields true; *and* 2) no additional functional operations are performed. Note that, functional operations are instructions other than those whose operators are of the "mov" series.

Integer type represents a number recording metadata like data size and length. Given its numeric nature, BinPRE regards a field as an Integer when 1) it involves arithmetic or bit-wise operations; *or* 2) it is compared with multiple consecutive values.

Group type comprises a list of static values that encompass all the possible values for the current field. It is typically employed to support multiple options, parsing the protocol message based on its value. Hence, BinPRE regards a field as Group only when compared with multiple different constants via conditional branches.

Bytes type denotes a sequence of binary bytes of arbitrary length. It usually serves as data chunks for transmission. BinPRE regards a field as Bytes only when all its bytes belonging to the same structure, identified through identical operations within a loop.

String type is similar to the Bytes type except that it is delimited by a specific delimiter. BinPRE regards a field as String type if the multiple consecutive bytes within this field are compared to the same constant (*i.e.*, a delimiter). Therefore, BinPRE distinguishes the String type from the Bytes type by the occurrences of specific consecutive comparison.

Command function indicates the message types, the most critical field in the messages. It has different names (*e.g.*, function code, command field, and keyword) in different protocol specifications. BinPRE regards a field as Command only when 1) the field is compared to a fixed value and the result yields true; *and* 2) when the result yields true, some function jump is immediately triggered.

Length function records the length of the whole or some part of a message. BinPRE regards a field as Length when 1) it serves as the termination condition of loop structure; *or* 2) it is retrieved by library APIs (*e.g.*, function `recv` which takes Length as input); *or* 3) it involves pointer increment and counter decrement operations.

Delim function is inextricably linked to field slicing in protocol implementations, indicating the end of a text protocol field. BinPRE regards a field as Delim when 1) it serves as the termination condition of a loop; *and* 2) it delimits its adjacent fields. Note that, the Delim function is often related to adjacent fields in text protocols, while the Length function is often related to a subsequent block of data or message in binary protocols.

Checksum function verifies the integrity of messages or data, assisting message interaction and communication. BinPRE regards a field as Checksum only when it is compared to the output of a loop which iterates multiple consecutive bytes.

Filename function stores a file's the path or name. As it is rarely modified or used for control-flow decisions, BinPRE identifies it by content rather than execution information, considering a field as Filename when it conforms to a common file naming convention.

Aligned function represents the fields that are rarely retrieved by the server, which are mainly for data alignment. When the execution information is missing, such fields are difficult to be identified. BinPRE regards a field as aligned only when it does not involve any functional operations.

Listing 2: $I(f)$

```
1  mov qword ptr [rbp−0x40], rsi
2  mov rax, qword ptr [rbp−0x40]
3  movzx eax, byte ptr [rax]
4  movzx edx, byte ptr [rdx]
5  shl edx, 0x8
6  or eax, edx
7  'LOOP':
8     cmp qword ptr [rbp−0x20], rax
9     movzx eax, byte ptr [rax]
10    xor eax, ecx
11    mov byte ptr [rbp−0x7], al
12    movzx edx, byte ptr [rbp−0x7]
13    ...'repeat for bytes 11−20'
14    cmp qword ptr [rbp−0x20], rax
15 movzx edx, word ptr [rbp−0x22]
16 cmp ax, dx
```

Listing 3: $V(f)$.

```
1  f2,2      V(f):0xb
2  f2,2      V(f):0xb
3  f21,21    V(f):0xc2
4  f22,22    V(f):0xd1
5  f22,22    V(f):0xd1
6  f21,22    V(f):0xc2,0xd100
7
8  f2,2      V(f):0xb
9  f10,10    V(f):0xc6
10 f10,10    V(f):0xc6
11 f10,10    V(f):0xc6
12 f10,10    V(f):0xc6
13
14 f2,2      V(f):0xb
15 f21,22    V(f):0xd1c2
16 f21,22    V(f):0xd1c2
```

Figure 5: Assembly instructions for lines 16-28 of Listing 1

Example-3. Take $f_{21,22}$ in Figure 1 as an example. The execution information of $f_{21,22}$ is shown in Figure 5. BinPRE regards its type as Integer, as it involves bit-wise operations (lines 5–6 of Figure 5). Meanwhile, the value of $f_{21,22}$ is compared with a value output

Table 2: Different semantic types and functions supported by BinPRE's library of atomic semantic detectors.

Semantic	Attributes	Rules of Atomic Semantic Detectors
Static	Fixed value across messages	Comparison OPs with a fixed value which yields true *and* without other operations
Integer	Represent a number of variable length	Arithmetic/bit-wise OPs *or* comparison OPs with multiple consecutive values
Group	Comprise a list of available values	Comparison OPs with multiple different constants
Bytes	Denote a sequence of binary bytes of arbitrary length	Field bytes' shared OPs within a loop
String	Denote a sequence of text characters	Comparison OPs of consecutive field bytes with the same constant *and* field bytes' shared OPs within a loop
Command	Denote message type	Comparison OPs with a fixed value which yields true *and* corresponding jumping OPs
Length	Indicate the length of message or data	Termination OPs of loops, *or* message parsing OPs, *or* pointer-subtraction/counter-decrement OPs
Delim	Seperate two fields	Termination OPs of loops *and* delimit OPs of adjacent fields
Checksum	Verify the integrity of messages or data	Comparison OPs with the output from iterations over multiple consecutive bytes
Filename	Identify a file within the file system	Common file naming convention
Aligned	Align contents to a certain number of bytes	Without functional OPs

from a loop that analyzes consecutive bytes (lines 7–16). Therefore, BinPRE regards its function as Checksum.

Discussion. To balance accuracy and generality, BinPRE supports 5 semantic types and 6 semantic functions that are common in protocols and have distinctive features. In real-world applications, there might be other field semantics. BinPRE could easily support them by extending the library of atomic semantic detectors. Note that that it is fundamentally impossible for the classic PRE techniques [30, 36, 46] to achieve similar inference results *w.r.t.* BinPRE, even if we extend these PRE techniques with our library of atomic semantic detectors (detailed in Appendix B).

3.5 Semantic Refinement

The *Semantic Refinement* module enhances the semantic inference results with a *cluster-and-refine* paradigm, incorporating contextual features of fields. Based on the format extraction results, it first explores the command field position and cluster messages with the most similar formats. Within each message cluster, it adopts an entropy-based approach [58] to characterize field content variation and refine the results of semantic types. Afterwards, it refines the results of semantic functions, which utilizes the constraints between the semantic functions and the semantic types.

3.5.1 Format-Based Clustering. With the formats from the *Format Extraction* module, BinPRE identifies the command field and clusters protocol messages. This is motivated by the observation that messages with the same command field typically have similar formats [65]. Utilizing the similarity of message formats, BinPRE explores the optimal basis, *i.e.*, command field position, to cluster messages. These message clusters contain important contextual features (*e.g.*, field content variations within each cluster) that would be used to further refine the semantics in *entropy-based type refinement* (*cf.* Section 3.5.2) and *type-based function refinement* (*cf.* Section 3.5.3) steps. Note that, this identified command field is also used for refining the earlier inference results.

As summarized in Algorithm 2, BinPRE iterates over all message fields and identifies the best basis for format-based clustering. Given a set of messages $MESSAGES := \{m_1, m_2, \ldots\}$, and the corresponding set of formats $FORMATS := \{F_1, F_2, \ldots\}$ obtained through *Format Extraction* module. For each message m_i, BinPRE iterates over the boundaries in its format results F_i (lines 6–14). It extracts a candidate command field f_{comm} from each pair of adjacent boundaries in F_i and clusters messages with it (lines 7–8). It then calculates the alignment score as the average NW_f score of the format sequences of each message pair within the cluster (line 9). Specifically, given two message format sequences, which contain the field boundaries within the messages, *i.e.*, F_a and F_b, their format alignment score NW_f is calculated as

$$NW_f(i,j) = \max \begin{cases} NW_f(i-1,j-1) + C(F_a[i], F_b[j]), \\ NW_f(i-1,j) + \texttt{GAP\_SCORE}, \\ NW_f(i,j-1) + \texttt{GAP\_SCORE} \end{cases}$$

$$C(m,n) = \begin{cases} \texttt{MA\_SCORE}, & if \quad Boundary(m) = Boundary(n) \\ \texttt{MISMA\_SCORE}, & otherwise \end{cases}$$

where $Boundary(m)$ indicates the offset of boundary m in the message; $C(F_a[i], F_b[j])$ compares whether the `i`th boundary of the message `a` and the `j`th boundary of the message `b`, *i.e.*, $F_a[i]$ and $F_b[j]$, are the same; `GAP_SCORE` (the default value is set as -2) is for penalizing discontinuities between format sequences; `MA_SCORE` (the default value is set as 1) is for encouraging matches of the same boundaries; and `MISMA_SCORE` (the default value is set as -1) is for penalizing mismatches of different boundaries. In the end of each iteration, the algorithm updates the highest alignment score max_{score} and the optimal clustering basis $command_{pos}$ (lines 10–12). Finally, BinPRE accepts the boundary pair of highest alignment score as the Command field, and clusters messages with their Command field value (lines 16–17).

Algorithm 2: Message clustering.

```
   Input: MESSAGES, FORMATS
   Output: CLUSTERS: the optimal clustering for messages
           command_pos: the optimal command position for clustering
1  Procedure ExploreOptimal(MESSAGES, FORMATS):
2      max_score := 0
3      command_pos := −1
4      for m_i in MESSAGES do
5          F_i := FORMATS[m_i]                          ▷ format result of m_i
6          for b_j in F_i do
7              f_comm := ExtractField(b_j, b_{j+1})
8              clusters := Clustering(f_comm, MESSAGES)
9              curr_score := AlignScore(clusters, FORMATS)
10             if curr_score>max_score then
11                 max_score := curr_score
12                 command_pos := f_comm
13             end
14         end
15     end
16     CLUSTERS := Clustering(comm_pos, MESSAGES)
17     return CLUSTERS, command_pos
18 End Procedure
```

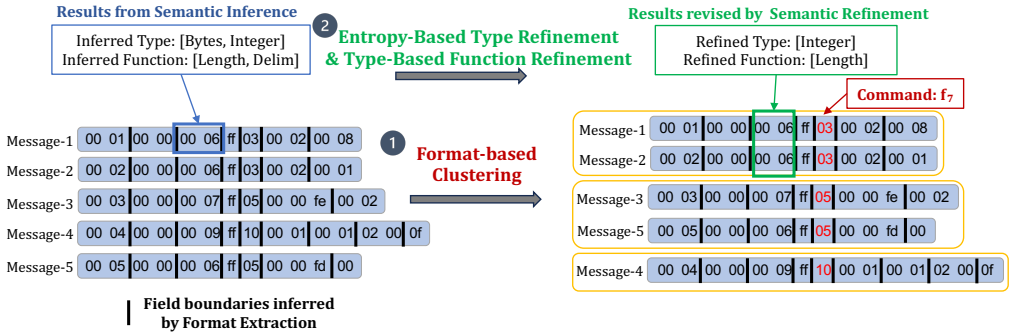

Figure 6: An illustrative example of *Semantic Refinement* module.

Table 3: Constraints between semantic functions and types.

Field Function	Constraint on Field Type
Command	Should be *Group*
Length	Should be *Integer*
Delim	Should be *Static* or *Group*
Aligned	Should be *Group* or *Bytes*
Checksum	Should be *Integer*
Filename	Should be *String*

3.5.2 Entropy-Based Type Refinement. Observing that field types mirror the underlying patterns of field value variations, we adopt entropy-based characteristics to refine the results of type inference.

In the *Semantic Inference* module, BinPRE utilizes execution information to infer semantic types. However, the inference results may not be reliable for the fields that are rarely retrieved. Therefore, BinPRE utilizes the content variation patterns of a field within the same message clusters to refine these results. It employs an information-theoretic approach to capture the variation patterns of field contents across the messages, with the assumption that messages within the same cluster have similar formats and semantics. BinPRE leverages Shannon entropy [58] to characterize the variation patterns. For field $f_{i,j}$, whose boundaries are of offset i and j, its Shannon entropy is

$$H(f_{i,j}) = -\sum_{v \in C_{i,j}} P(v) \log P(v) \ ,$$

where $C_{i,j}$ is the collection of the i-th to j-th byte of messages in the Cluster C. The probability of a value v in $C_{i,j}$, denoted as $P(v)$, is calculated as its frequency in $C_{i,j}$.

To reduce false positives, BinPRE only uses extreme entropy patterns to refine the inferred semantic types. The Static and Bytes types present two extremes of conveyed information: the Shannon entropy of the former is expected to be relatively small, and the latter to be large. BinPRE then calculates the median Shannon entropy of all fields within the same cluster C. The inference result of the Static (Bytes) type is valid only when its Shannon entropy is smaller (larger) than the median.

BinPRE also uses entropy characteristics to infer the semantic type of fields that lack execution information. It is based on the intuition that similar Shannon entropy usually appears in fields of the same semantic types. For a field of unknown type, BinPRE finds the field with the closest Shannon entropy within the same message cluster, and regards them as sharing the same type.

3.5.3 Type-Based Function Refinement. We observe that there are some constraints between semantic types and functions. For a certain semantic function, its corresponding semantic type is constrained to a small subset. For example, the Length field should be the Integer type, as the data length is an integer value. Therefore, we utilize the refined semantic types to refine the semantic functions. BinPRE applies such constraints to remove the unreasonable inference results of semantic functions. Table 3 summarizes the constraints. The Command field should be of Group type, as the server typically selects one of the options for parsing an input message based on the Command value; The Length field and Checksum field should be of Integer type according to their definitions; The Delim field should be of Static or Group type, as it serves as data boundaries; The Aligned field should be of Group or Bytes type, according to its typical implementations; Similarly, the Filename field should be of String type.

Example-4. Figure 6 illustrates the process of *Semantic Refinement* module. Assume it takes the given five messages as inputs. The left part shows the format and semantic results obtained from the *Format Extraction* and *Semantic Inference* modules. The right part shows the results obtained after executing the *Semantic Refinement* module. BinPRE first iterates over all the message fields to identify the optimal clustering basis f_7 based on the similarity of message formats. It then figures out that the Shannon entropy of $f_{4,5}$ in the first cluster is zero, and thus removes the inference result of Bytes through entropy-based type refinement. Similarly, it also removes the unreasonable inference result Delim of $f_{4,5}$ through type-based function refinement. With the *Semantic Refinement module*, BinPRE finally achieves the correct semantic inference of $f_{4,5}$.

3.6 Implementation

BinPRE is implemented in Python3 and C++. The *Execution Monitor* module taint tracks binary files with the dynamic instrumentation tool Pin [17]. It uses Scapy [19] to derive execution information of messages, aiding subsequent field inference. The *Format Extraction* module implements instruction-based semantic similarity analysis from Section 3.3 to extract the message formats. The *Semantic Inference* module infers field semantics with the rules in Table 2, and validates filename conventions through file path syntax. The *Semantic Refinement* module follows the three-step process from Section 3.5, utilizing the results from *Format Extraction* and *Semantic Inference* modules to improve the overall correctness of field

inference. BinPRE is modularly designed. The modules export uniform interfaces and could be easily extended to support additional field semantics.

4 EVALUATION

Our evaluation aims to answer the following research questions:

- RQ1: How accurate is BinPRE in performing format extraction?
- RQ2: How accurate is BinPRE in performing semantic inference?
- RQ3: How effective are BinPRE's semantic refinement components in improving the accuracy of semantic inference?
- RQ4: How useful are the field inference results of BinPRE in improving such downstream tasks as protocol fuzzing?

4.1 Setup

4.1.1 Baselines. We compared BinPRE with five state-of-the-art PRE tools:

- Polyglot [30] is an ExeT-based tool that infers key field semantics with simple heuristics to facilitate further format extraction.
- AutoFormat [46] is an ExeT-based tool that extracts formats with hierarchical, parallel, and sequential relationships.
- Tupni [36] is an ExeT-based tool that identifies field formats from instruction frequency and record sequences.
- BinaryInferno [31] is a NetT-based tool that infers fields with multiple atomic detectors of different heuristic rules.
- DynPRE [49] is a NetT-based tool that extracts field formats through the interactive capabilities of the server.

Among all baselines, AutoFormat cannot infer semantics, while the others have very limited support. Since Polyglot, AutoFormat, and Tupni are not publicly available, we re-implemented our versions of their techniques and validated with the examples in their papers before the comparison (detailed in Section 4.3).

4.1.2 Benchmarks. We constructed the benchmarks as follows:
Protocols. We selected 7 popular protocols evaluated by the prior PRE tools Polyglot [30], AutoFormat [46], Tupni [36], BinaryInferno [31] and DynPRE [49]. These 7 protocols are widely-adopted and covering textual, binary, and mixed categories. Their application scenarios include industrial control and network communication. Among them, S7comm was a proprietary protocol, and the others are open-sourced. We additionally selected another real-world protocol Ethernet/IP from prior work [59, 61]. It is an industrial protocol that is widely adopted across many industrial sectors, including factory automation and hybrid process control. Thus, we evaluated BinPRE with 8 real-world protocols. Among these 8 protocols, 7 have been evaluated by at least one prior PRE tool, and 4 have been evaluated by at least two prior tools. In particular, BinPRE has 2, 1, 4, 2, 4 common protocols with Polyglot, AutoFormat, Tupni, BinaryInferno and DynPRE, respectively.
Messages. Each protocol is tested with 50 messages with diverse and real-world usage scenarios. The messages are collected from two sources: (1) open-source network trace datasets; and (2) protocol clients or other relevant tools when the former is unavailable. Since the binary analysis based PRE techniques like BinPRE are not sensitive to the sizes of input messages, we did not use a larger dataset size. On the other hand, the two network-based PRE techniques (BinaryInferno and DynPRE) are designed to be capable of handling small-sized input messages. Table 9 in the Appendix lists the evaluated protocols and their characteristics including the server under testing, the message sources and the number of different message types. For Ethernet/IP and HTTP, we ran the protocol client and the command line tool *curl* with different options/parameters, respectively, to obtain diverse messages. For other protocols, we randomly filtered the input messages with different types and contents from their open-source traces to improve diversity and reduce dataset biases.
Ground-truth. We adopted protocol specifications as the ground-truth. For each message, its ground-truth includes field boundaries (format) and field type/functions (semantics). Specifically, for each protocol's 50 messages, we use Wireshark's packet dissectors to parse each message and obtain the ground-truths of field formats and semantics. We also referred to the protocol RFCs for validation.

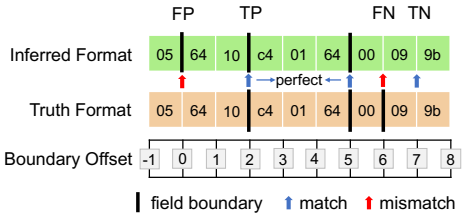

Figure 7: An illustration of format extraction metrics.

4.1.3 Protocol fuzzing. Protocol fuzzing is an important application scenario of PRE. To evaluate how well the PRE tools assist the downstream task performance, we incorporate their field inference results into a classical generation-based fuzzer, Boofuzz [1], to guide test generation. Specifically, for the fields supported by PRE tools, we applied the corresponding types/functions in Boofuzz to these fields. For the unsupported fields, we adopted the *random* strategy, which only leverages the format extraction results to limit the boundaries of each random field. Each scheme was tested for 10 hours on a machine with a Core i7-13700 CPU and 16GB memory.

4.1.4 Metrics. We adopt different metrics for different tasks.

Format extraction task is evaluated with the accuracy and F1-score (a combination of precision and recall) of field boundary detection [31, 49]. As illustrated in Figure 7, each boundary offset position between two adjacent bytes can fall into one of the four categories: FP, TP, FN, and TN. Based on these definitions, we calculated *accuracy* (*i.e.*, the number of correctly inferred positions out of all offset positions), *precision* (*i.e.*, the number of inferred true field boundaries out of all inferred boundaries), and *recall* (*i.e.*, the number of inferred true field boundaries out of all true boundaries). We also counted the number of *perfect* fields, of which both boundaries are accurately detected, and calculated *perfection* (*i.e.*, the number of perfectly inferred fields out of all true fields).

Semantic inference task is evaluated with the precision, recall, and F1-score (a combination of precision and recall) of field semantic type/function identification. As different tools support different semantic types/functions (see Section 2.3), we evaluated type/function separately. Specifically, *precision* is calculated as the number of inferred true types/functions out of all inferred types/functions, and *recall* is calculated as the number of inferred true types/functions out of all true types/functions.

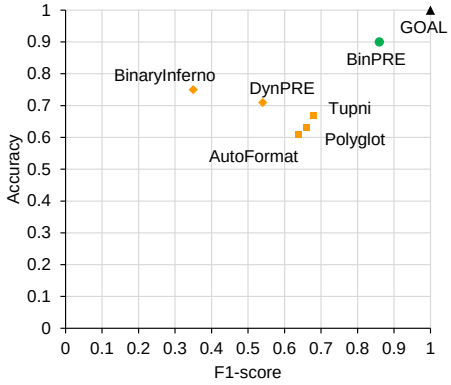

Figure 8: Average format extraction performance (the more upper-right the point is, the better the tool)

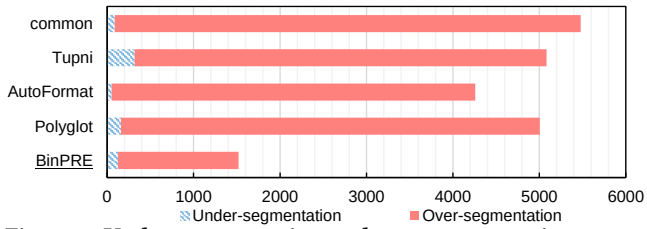

Figure 9: Under-segmentation and over-segmentation errors of the common strategy used by prior ExeT-based tools, the three ExeT-based baselines, and BinPRE (The segmentation errors caused by unused fields are excluded in this analysis).

Protocol fuzzing is evaluated with branch coverage improvement. We used SanitizerCoverage [18] to identify covered unique branches. As any server's startup executes fixed program paths, we focus on the branches covered after the server has started.

4.2 Results

4.2.1 RQ1: Format. Figure 8 summarizes the average performance of each tool on format extraction, and Table 4 details the results.

Overall, BinPRE significantly outperforms the baselines, achieving an average accuracy/F1-score/perfection of 0.90/0.86/0.73, due to its effective instruction-based semantic similarity analysis. In comparison, the baselines are 13-60% lower.

In some cases, BinPRE has similar or slightly lower performance than the best baseline, due to the correlation between the format extraction algorithm and protocol characteristics. For example, the S7comm protocol has compact message structures and the behavioral differences between fields are relatively small (*i.e.* several adjacent fields only retrieved by the parsing process). Meanwhile, Polyglot leverages the typical patterns of the single instruction-based field segmentation, which is particularly effective on such protocols. Therefore, it perfectly identifies 27% more fields than BinPRE. As another example, BinPRE does not consistently achieve the highest perfection on the HTTP protocol, as the server uses delimiters as the starting point of the key-value pair field: the delimiters and their subsequent fields share similar semantics and are processed in similar ways. Thus, BinPRE fails to identify their boundaries in some cases.

Figure 9 shows the numbers of over-segmentation and under-segmentation errors of the common strategy for format extraction

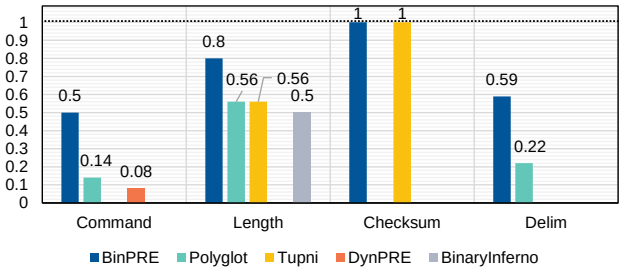

Figure 10: Average F1-scores of semantic inference across 8 protocols achieved by the evaluated PRE tools. The detailed results of each protocol/tool are given in Table 10 in the Appendix (Only the semantics supported by the tools are included).

used by the prior ExeT-based tools (detailed in Section 2.2), the three ExeT-based baselines and BinPRE, respectively. We excluded the fields which have not been accessed by the servers because these fields cannot be inferred from the instruction traces. Compared with the alternatives, BinPRE has 63~70% fewer segmentation errors, and achieves the best trade-off between over-segmentation and under-segmentation errors. While AutoFormat has the least under-segmentation errors, it comes at the cost of more over-segmentation errors. Moreover, BinPRE has 72% fewer segmentation errors than the common strategy used by the prior ExeT-based tools. BinPRE's improvement is achieved by its effective instruction-based semantic similarity analysis strategy.

Answer to RQ1 (Format): BinPRE identifies field formats with the perfection score of 0.73, which outperforms all the baselines.

4.2.2 RQ2: Semantics. Overall, BinPRE achieves an average precision, recall, and F1-score of 0.72/0.77/0.74 for semantic type inference and 0.77/0.91/0.81 for semantic function inference.

We compared BinPRE with the baselines, except for AutoFormat which does not support semantics inference. Figure 10 shows the average F1-score on each semantic which is supported by at least one baseline. BinPRE greatly outperforms baselines in all cases. Specifically, Polyglot and Tupni are 24% worse than BinPRE on the Length field, as they heavily rely on behavior-based heuristic rules, which are not resilient towards diverse protocol implementations. DynPRE's and BinaryInferno's poor format extraction performance results in a high number of errors on semantic inference. Furthermore, the context-based rules of BinaryInferno are not suitable for datasets of diverse formats.

BinPRE also has relatively high performance on the other seven semantics that are unsupported by baselines: it has the F1-scores of 0.56 for Static, 0.61 for Integer, 0.50 for Group, 0.54 for Bytes, 0.15 for String, 0.00 for Aligned and 0.67 for Filename. BinPRE fails to identify the fields Aligned in our benchmark suite. Because the fields Aligned typically do not serve any functionality, and thus have little behavioral and contextual features.

Answer to RQ2 (Semantics): BinPRE achieves the F1-scores of 0.74/0.81 on semantic inference for types/functions. It outperforms all the baselines on each semantic type/function.

Table 4: Format extraction result summary, including accuracy, F1-score, and perfection (Bold numbers indicate the best results).

Protocol	BinPRE			Polyglot			AutoFormat			Tupni			BinaryInferno			DynPRE		
	Acc.	F1.	Perf.	Acc.	F1.	Perf.	Acc.	F1.	Perf.	Acc.	F1.	Perf.	Acc.	F1.	Perf.	Acc.	F1.	Perf.
Modbus	**1.00**	**1.00**	**0.99**	0.91	0.93	0.84	0.91	0.93	0.84	0.82	0.87	0.61	0.79	0.79	0.32	0.71	0.79	0.39
S7comm	0.79	0.83	0.60	0.85	0.90	**0.87**	0.82	0.88	0.80	**0.86**	**0.91**	0.85	0.54	0.52	0.09	0.65	0.71	0.28
Ethernet/IP	**0.95**	**0.92**	0.66	0.75	0.71	**0.79**	0.52	0.56	0.38	0.72	0.69	0.75	0.77	0.57	0.28	0.90	0.83	0.42
DNP3.0	**0.95**	**0.95**	**0.88**	0.90	0.84	0.63	0.65	0.72	0.75	0.75	0.67	0.25	0.61	0.50	0.25	0.59	0.36	0.08
DNS	**0.74**	**0.66**	**0.62**	0.73	**0.66**	0.58	0.39	0.46	0.44	0.63	0.58	0.47	0.82	0.46	0.11	0.45	0.41	0.26
FTP	**0.88**	**0.86**	**0.57**	0.42	0.59	0.24	0.52	0.64	**0.57**	0.56	0.63	0.07	0.72	0.00	0.00	0.79	0.67	0.36
TFTP	**0.99**	**0.96**	**0.89**	0.25	0.36	0.30	0.44	0.43	0.30	0.83	0.74	0.30	0.89	0.00	0.00	0.77	0.46	0.07
HTTP	**0.86**	**0.72**	0.60	0.24	0.33	0.56	0.63	0.51	**0.65**	0.20	0.32	0.58	0.83	0.00	0.00	0.82	0.14	0.03
Average	**0.90**	**0.86**	**0.73**	0.63	0.66	0.60	0.61	0.64	0.59	0.67	0.68	0.49	0.75	0.35	0.13	0.71	0.54	0.24

Table 5: F1-score of semantic inference for field Command.

Protocol	BinPRE#	BinPRE
Modbus	0.85	1.00
S7comm	0.22	1.00
Ethernet/IP	0.67	1.00
DNP3.0	0.00	0.00
TFTP	0.00	1.00
Average	0.35	0.80

* BinPRE# is a variant of BinPRE, which omits format-based clustering.

4.2.3 RQ3: Ablation. We conducted ablation studies to assess the effectiveness of *Semantic Refinement* module in semantic inference.

Format-based clustering. Table 5 shows the Command field identification results with and without format-based clustering (BinPRE#). As the format extraction module fails to correctly segment the command fields of DNS, FTP, and HTTP, we focus on the remaining protocols. The result shows that format-based clustering offers effective semantic refinement: BinPRE achieves the F1-score of 0.8, while BinPRE# drops to 0.35. Both BinPRE# and BinPRE fail on DNP3.0, as DNP's messages of different commands share similar formats, which conflicts with BinPRE's conjunctures.

Type/function refinement. We compared BinPRE with its variant which omits type/function refinement. As shown in Table 6, adopting type (function) refinement improves the semantic inference's precision by 0.08 (0.25), recall by 0.07 (0.06), and F1-score by 0.08 (0.19). It is worthy noting that adopting both refinements improves the F1-score of Command by 0.29, Length by 0.26, Bytes by 0.18, and Group by 0.15. These improvements are achieved by handling the complex and irregular protocol implementations by refining inference results with extra contextual information.

Answer to RQ3 (Ablation): All the three components of *Semantic Refinement* module are effective in refining semantic inference results. *Format-based clustering* improves the F1-score of command inference from 0.35 to 0.80. *Type(function) refinement* improves the F1-score of semantic results from 0.66(0.62) to 0.74(0.81).

4.2.4 RQ4: Downstream tasks. We evaluated the field inference results from BinPRE and the baselines to enhance the downstream task of protocol fuzzing. Since BinPRE provides the information of both field formats and semantics (types and functions), we also evaluated the two variants of BinPRE: (1) BinPRE* (only keeping the field formats and types) and (2) BinPRE** (only keeping the field formats). Figures 11 and 12 summarize the results of branch coverage by fuzzing four representative protocols, *i.e.*, Modbus, Ethernet/IP, HTTP and FTP, from our benchmark suite. Modbus and Ethernet/IP are binary protocols, while HTTP and FTP are text protocols. To mitigate the randomness during fuzzing, we ran each configuration for three times and computed the average values.

	Modbus		Ethernet/IP		HTTP		FTP	
BinPRE+	109	+77	133	+76	192	+159	99	+59
BinPRE	109	+77	121	+64	179	+146	87	+47
BinPRE*	107	+75	121	+64	172	+139	72	+32
BinPRE**	85	+54	108	+51	185	+152	92	+52
Polyglot	91	+59	103	+46	182	+149	92	+52
Tupni	92	+60	97	+40	173	+140	90	+50
AutoFormat	92	+60	96	+39	176	+143	92	+52
BinaryInferno	69	+37	77	+20	34	+1	40	+0
DynPRE	72	+40	109	+52	166	+133	47	+7
Random	32	+0	57	+0	33	+0	40	+0

Figure 11: Numbers of unique branches covered by PRE-enhanced Boofuzz (averaged across three repeated runs).

Branch Coverage. In Figure 11, BinPRE+ denotes the union results of unique branches covered by BinPRE, BinPRE* and BinPRE**. It indicates the overall usefulness of the PRE results (with different levels of information) of BinPRE to enhance Boofuzz. The Random strategy denotes the results of the vanilla Boofuzz without given any results of the PRE tools (*i.e.*, Boofuzz simply generate random message contents). In Figure 11, for each protocol, we give the numbers of branches covered by Boofuzz enhanced by the results of different PRE tools, and the improved branch coverage *w.r.t.* the random strategy. On average, BinPRE-enhanced Boofuzz (denoted by BinPRE+) achieves 20%/29%/5%/8% (51%/22%/16%/111%) higher branch coverage on Modbus/Ethernet IP/HTTP/FTP, compared to the best ExeT-based (NetT-based) baseline Polyglot (DynPRE). BinPRE-enhanced Boofuzz improves the performance of vanilla Boofuzz (denoted by Random) by covering 244%/133%/482%/148% more branches on Modbus/Ethernet IP/HTTP/FTP. BinPRE is particularly effective in improving fuzz testing on binary protocols because the binary protocols have compact and diverse field formats and semantics. It covers 244% more branches on Modbus, and 133% more on Ethernet/IP. On the loosely formatted text protocols, the information of field formats is more useful for fuzzing: BinPRE** covers more branches on HTTP/FTP than BinPRE/BinPRE*.

Efficiency Figure 12 shows the numbers of unique branches covered by Boofuzz enhanced by the results of different PRE tools. We can see that BinPRE-enhanced Boofuzz variants (*i.e.*, BinPRE, BinPRE* and BinPRE**) outperforms those enhanced by the prior PRE tools in most cases. In Table 7, we computed the speedup of BinPRE-enhanced Boofuzz variants over those enhanced by the prior PRE tools in achieving highest identical branch coverage. The numbers greater than 1.0x indicate higher speeds of achieving branch coverage. We can see that *in most cases* BinPRE-enhanced Boofuzz variants are faster than the other baselines.

Table 6: BinPRE's semantic inference results w/ and w/o type/function refinement (Bold numbers indicates the best results).

Protocol	BinPRE Semantic Type w/o Refinement			BinPRE Semantic Type w/ Refinement			BinPRE Semantic Function w/o Refinement			BinPRE Semantic Function w/ Refinement		
	Pre.	Rec.	F1.	Pre.	Rec.	F1.	Pre.	Rec.	F1.	Pre.	Rec.	F1.
Modbus	**0.83**	0.61	0.70	**0.83**	**0.84**	**0.83**	0.43	0.87	0.57	**0.77**	**1.00**	**0.87**
S7comm	0.50	0.53	0.52	**0.51**	**0.55**	**0.53**	0.42	**0.56**	0.48	**0.85**	**0.56**	**0.67**
Ethernet/IP	0.60	0.77	0.67	**0.80**	**0.90**	**0.85**	0.46	1.00	0.63	**1.00**	**1.00**	**1.00**
DNP3.0	0.55	**0.70**	0.62	**0.71**	**0.70**	**0.71**	0.43	**1.00**	0.60	**0.77**	**1.00**	**0.87**
DNS	**0.59**	**0.79**	**0.67**	**0.59**	**0.79**	**0.67**	0.25	**0.98**	0.40	**0.29**	**0.98**	**0.45**
FTP	**0.62**	**0.62**	**0.62**	**0.62**	**0.62**	**0.62**	0.87	**1.00**	0.93	**0.90**	**1.00**	**0.95**
TFTP	0.50	0.67	0.57	**0.75**	**1.00**	**0.86**	0.33	0.50	0.40	**0.67**	**1.00**	**0.80**
HTTP	**0.95**	**0.92**	**0.94**	0.94	0.78	0.85	0.94	**0.93**	**0.93**	**0.95**	0.77	0.86
Average	0.64	0.70	0.66	**0.72**	**0.77**	**0.74**	0.52	0.85	0.62	**0.77**	**0.91**	**0.81**

Table 7: Speedup of BinPRE-enhanced Boofuzz variants (*i.e.*, BinPRE, BinPRE* and BinPRE) over those enhanced by prior PRE tools in achieving highest identical branch coverage (Numbers larger than 1x indicate higher speeds of achieving coverage).**

	Modbus			Ethernet/IP			HTTP			FTP		
	BinPRE	BinPRE*	BinPRE**	BinPRE	BinPRE*	BinPRE**	BinPRE	BinPRE*	BinPRE**	BinPRE	BinPRE*	BinPRE**
Polyglot	408.7x	670.7x	0.2x	513.3x	365.8x	154.8x	0.5x	2.7x	3.9x	0.7x	3.1x	5.4x
Tupni	975.1x	1628.8x	0.4x	20.1x	14.5x	2.1x	5.1x	1.1x	1.7x	0.8x	2.5x	6.7x
AutoFormat	3585.9x	5986.5x	0.5x	49.9x	35.9x	3.7x	26.9x	4.7x	8.6x	0.7x	4.7x	38.9x
BinaryInferno	82.6x	76.0x	45.1x	6989.0x	4997.4x	1752.6x	3972.0x	4038.4x	2383.0x	84.0x	70.2x	101.1x
DynPRE	5.5x	4.6x	2.7x	1134.1x	807.5x	0.0006x	1.4x	0.7x	1.0x	1233.6x	1761.9x	1606.7x
Random	29.5x	29.4x	35.2x	8.7x	6.1x	3.9x	36.1x	36.7x	12.8x	133.6x	111.6x	160.7x

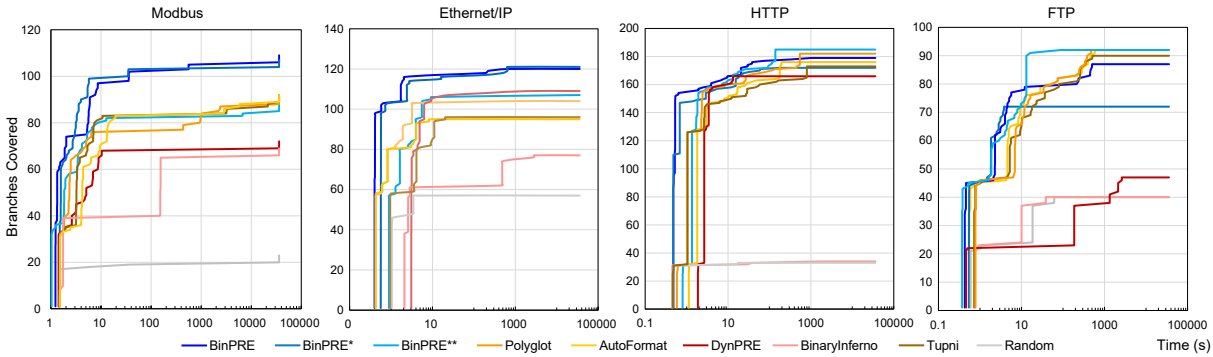

Figure 12: Numbers of unique branches covered by Boofuzz in ten hours enhanced by the results of different PRE tools.

Revealed CVEs. BinPRE helped Boofuzz discover a new zero-day vulnerability CVE-2024-31504 (CVSS score of 7.5 HIGH) in FreeMODBUS. It could cause a buffer overflow and thus lead to a denial of service. This vulnerability is related to two critical fields: (1) a length field (in the message's header) and (2) a variable-length field (in the message's payload). It is triggered when the value of the former is large enough to allow the latter to exceed the buffer limit, which causes a buffer overflow when FreeMODBUS tries to read it. Specifically, a specific protocol variable *n* is overwritten by a variable-length field, which indexes the array *fd_bits*, causing a segmentation fault when *n* is overwritten to a large integer.

Only BinPRE helps Boofuzz discover this CVE, as BinPRE is the only PRE tool that correctly segments the two critical fields and correctly infers their types/functions — a Length field with integer type, and a field with Bytes type. During the fuzzing, the inferred Length function renders the server to process the entire message payload, and the inferred Bytes type facilitates the generation of byte sequences with arbitrary length. However, all the baseline tools fail to identify these two fields. AutoFormat, Tupni, BinaryInferno, and DynPRE fail in format extraction, while the other tools fail in semantic inference. Therefore, the baselines cannot find this CVE. Additionally, BinPRE also helps Boofuzz find one CVE-requested bug (a buffer overflow vulnerability in FreeMODBUS) and one known vulnerability CVE-2020-29596 in Miniweb.

> *Answer to RQ4 (Downstream tasks)*: BinPRE can enhance protocol fuzzing, and help discover new or known vulnerabilities. The BinPRE-enhanced Boofuzz achieves 5~29% higher branch coverage, compared to the best prior ExeT-based PRE tool.

4.3 Discussions

Cost of BinPRE. The runtime cost of BinPRE includes two parts: (1) tracking execution information of input messages; and (2) inferring format and semantics from the execution information. For the first part, BinPRE's instrumentation introduces average 59% runtime overhead. For the second part, analyzing 50 messages takes less than 2 minutes on a machine with a Core i5-1038NG7 CPU and 8GB memory. As protocol reverse engineering is typically a one-time cost, we believe that such overhead is affordable, considering BinPRE's benefits on downstream tasks.

Reproduction of Baselines. Three of the baselines (Polyglot, AutoFormat and Tupni) are not publicly available. Therefore, we re-implemented them according to their algorithm descriptions and validated our implementations with the given examples in their original papers. We observe that our evaluation results are also consistent with their claims.

- Polyglot: Sections 3–6 of its paper were re-implemented. Its format extraction algorithm and direction field identification strategy are validated on the DNS protocol. Its keyword and separator (delimiter) identification strategies are validated on the HTTP protocol. Our implementation achieved the same results as described in the paper. Note that Command and Length are the most common keyword and direction fields, respectively [30]. Therefore, their identification strategies were implemented and compared in our evaluation.
- AutoFormat: Section 3 (excluding 3.2.2) of its paper was re-implemented. Its format extraction algorithm is validated on the HTTP protocol, achieving the same results as in the paper.
- Tupni: Sections 3.1–3.5 of its paper were re-implemented. Its format extraction algorithm is validated on the HTTP protocol. Our implementation has slightly different outputs from the paper. These discrepancies are caused by the unused fields and delimiter checking within the server, which is consistent with the root causes *i.e.*, "a program ignores certain parts of an input". Note that neither the original Tupni nor our implementation supports delimiter recognition. Therefore, we believe that our implementation could reflect the original capability of Tupni.

The other baselines are open-sourced and we directly used them. While their performances align with their paper descriptions, there are variations on DynPRE. For instance, DynPRE performed worse on the HTTP protocol in our evaluation because it segments fields based on dynamic analysis of server responses, limiting context information when a protocol generates similar responses for different inputs. This issue was confirmed by DynPRE's authors.

Effect of Data Size. We also evaluated the effectiveness of BinPRE on small-sized input messages. We created a 5-message dataset by randomly sampling from the 50-message one used in our evaluation. BinPRE has achieved similar performance on the 5-message dataset with the average F1-scores of 0.87 for format extraction and 0.75/0.84 for semantic type/function inference, respectively. It shows that BinPRE does not rely on large-sized input messages and could maintain the similar performance on small-sized ones.

Limitations. The taint analysis module of BinPRE only works at byte-level. It cannot accurately deal with bit-level fields like bit-flags. To our knowledge, all existing PRE tools take byte as the basic unit for field inference due to the dominance of byte-level fields.

5 RELATED WORK

5.1 Automated Protocol Reverse Engineering

Existing work proposes two types of PRE techniques: (i) network traffic-based (NetT-based) inference; and (ii) execution trace-based (ExeT-based) inference [39]. NetT-based inference [25, 26, 32, 35, 42, 56, 62, 65] leverages numerous messages within sessions and investigates their relationships. It infers protocol specifications by capturing message features from sessions. ExeT-based inference [27, 30, 33, 34, 36, 46–48, 60, 63] takes the binary files of servers as inputs. It monitors the execution trace and tracks the message parsing process in binary files, which reveals the internal design logic. With these execution information, it segment messages into fields and infers field semantics. While BinPRE and the prior ExeT-based work all use taint analysis, they have distinct differences in format extraction, semantic inference and refinement. We will discuss the differences in the next two sections.

5.2 PRE for Format Extraction

Format extraction is an essential PRE task. NetT-based approaches identify field boundaries through statistical methods. Netzob [26] and Netplier [65] adopt alignment-based methods. Discoverer [35] recursively clusters messages of the same type. BinaryInferno [31] treats messages as information sequences and proposes an information theoretic approach method. DynPRE [49] further utilizes response messages to enhance boundary identification. These approaches rely on diverse large-scale message data.

ExeT-based approaches monitor the execution traces to extract field boundaries. Polyglot [30] is one of the first efforts to employ taint analysis techniques to monitor the binary files of target protocols and capture execution traces. It uses three specific field functions (*direction field*, *keyword*, and *separator*), which are inferred by heuristic patterns of execution traces, to segment fields. Building upon the format partitioning framework of Polyglot, AutoFormat [46] and Tupni [36] propose hierarchical and packet field recognition methods. The former segment fields are based on the locality of executed instructions and the similarity of their call stacks. The latter identifies candidate fields based on byte occurrences and merges the consecutive bytes processed in one loop into one field. Prospex [34] further extends AutoFormat to the session level, tracking and dividing the message parsing process within a complete session based on system calls. These techniques rely on the behavioral features observed in execution traces of message parsing. However, due to the variations in protocol implementations, these features may not reflect the actual message formats. To address this issue, BinPRE compares the semantic similarity by approximating the operator sequences between executed instruction sequences. The instruction-based semantic similarity analysis of BinPRE can capture deeper internal relationship between bytes and is more resilient to different protocol implementations.

5.3 PRE for Semantic Inference

Semantic inference focuses on understanding the meanings of each field in the messages [29]. NetT-based approaches face obstacles to inferring protocol semantics, as the network traces only contain syntax information [28]. While researchers propose clustering [23, 24, 43], supervised deep-learning [64, 67] and other heuristic [44, 51] solutions to additionally support semantic inference, they highly rely on large-scale diverse network packets. In addition, they are only able to handle limited field semantics and have low inference accuracy (illustrated in Table 1).

ExeT-based inference approaches extract field semantics from the execution information of the protocol binaries. One line of work, including Dispatcher [27], captures library function semantics to infer field semantics. Another line of work [30, 36], designs

heuristic rules to analyze the execution traces of fields and identify behavioral semantic features, which are then mapped to field semantics. However, these work suffer from inadequate and inaccurate semantic inference due to their limited and inaccurate behavior based rules. For instance, Polyglot and Tupni infer only three and two semantic functions with limited accuracy, respectively.

BinPRE differs from them in two aspects: 1) we are the first to construct a library of atomic semantic detectors for semantic inference; and 2) BinPRE utilizes contextual information to refine the semantic inference results in a systematic way. The former addresses the inadequate rules of semantic inference, while the latter mitigates the inaccurate results of semantic inference.

5.4 Protocol Fuzzing

Protocol fuzzing is widely used for testing protocol implementations, and generates massive test cases to trigger abnormal runtime behaviors [38, 66].

Mutation-based fuzzers (*e.g.*, NSFuzz [57], AFLNet [55], and ChatAFL [52]) require initial seeds with proper formatting and content. They then employ various mutation methods on these message seeds. Generation-based fuzzers (*e.g.*, Boofuzz [1] and Peach [15]) generate semi-valid messages based on the given templates. Their effectiveness heavily depends on the pre-defined protocol structures. However, real-world protocols typically have complex structures and communication logic. Manually constructing initial seeds and templates requires huge efforts and also knowledge of the protocol specifications. BinPRE tackles a different problem from these work. It provides automated PRE solutions to assist protocol fuzzing.

6 CONCLUSION

Protocol reverse engineering aims to infer the specifications of closed-source protocols. In this paper, we propose BinPRE, a binary analysis based PRE tool that supports both format extraction and semantic inference. It incorporates an instruction-based semantic similarity analysis for format extraction, and a novel library composed of atomic semantic detectors and a cluster-and-refine paradigm for improving semantic inference. We evaluate BinPRE with a variety of protocols and achieve high accuracy on the format extraction and semantic inference tasks. It also effectively supports the downstream task of protocol fuzzing.

References

[1] 2024. *Boofuzz.* A fork and successor of the Sulley Fuzzing Framework. https://github.com/jtpereyda/boofuzz.git.

[2] 2024. *Boofuzz.* The official guide to protocol definition in boofuzz. https://boofuzz.readthedocs.io/en/stable/user/protocol-definition.html.

[3] 2024. *curl.* Command line tool for capturing HTTP messages. https://curl.se/.

[4] 2024. *dnsmasq.* Dnsmasq fork with fast ipset/server/address lookup. https://github.com/infinet/dnsmasq.

[5] 2024. *freemodbus.* BSD licensed MODBUS RTU/ASCII and TCP slave. https://github.com/cwalter-at/freemodbus.git.

[6] 2024. *LightFTP.* Small x86-32/x64 FTP Server. https://github.com/hfiref0x/LightFTP.

[7] 2024. *miniweb.* Small, cross-platform HTTP server with useful directory listing. https://github.com/avih/miniweb/tree/master.

[8] 2024. *Open real-world traces for DNP3.0.* The DNP3 protocol dataset used in the paper 'Deterministic Dendritic Cell Algorithm Application to Smart-Grid Cyber Attack Detection'. https://github.com/igbe/DNP3-Dataset-Plus-SnortRules/blob/master/dnp3dataset_capture.pcap.

[9] 2024. *Open real-world traces for FTP.* Anonymous FTP traces published by the Lawrence Berkeley National Laboratory. ftp://ita.ee.lbl.gov/new/lbnl.anon-ftp.03-01-10.tcpdump.gz.

[10] 2024. *Open real-world traces for Modbus and DNS.* A list of public packet capture (PCAP) repositories, which are freely available on the Internet. https://www.netresec.com/?page=PcapFiles.

[11] 2024. *Open real-world traces for S7comm.* Wireshark's sample capture library, which includes real-world traces for S7comm. https://wiki.wireshark.org/SampleCaptures#s7comm-s7-communication.

[12] 2024. *Open real-world traces for TFTP.* A TFTP pcap file on the CloudShark website. https://www.cloudshark.org/captures/07ebe14c792b.

[13] 2024. *opendnp3.* DNP3 (IEEE-1815) protocol stack. https://github.com/dnp3/opendnp3.

[14] 2024. *OpENer.* An EtherNet/IP stack for I/O adapter devices. https://github.com/EIPStackGroup/OpENer.git.

[15] 2024. *Peach.* A fuzzing framework which uses a DSL for building fuzzers and an observer based architecture to execute and monitor them. https://github.com/MozillaSecurity/peach.git.

[16] 2024. *Peach.* The official guide to protocol definition in peach. https://peachtech.gitlab.io/peach-fuzzer-community/.

[17] 2024. *Pin.* A dynamic binary instrumentation framework for the IA-32 and x86-64 instruction-set architectures that enables the creation of dynamic program analysis tools. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[18] 2024. *SanitizerCoverage.* A simple code coverage instrumentation built in LLVM. https://clang.llvm.org/docs/SanitizerCoverage.html.

[19] 2024. *Scapy.* A powerful interactive packet manipulation library. https://scapy.net/.

[20] 2024. *snap7.* 32/64 bit multi-platform Ethernet S7 PLC communication suite. https://sourceforge.net/projects/snap7.

[21] 2024. *tftpd-hpa.* A conglomerate of a number of versions of the BSD TFTP code. https://git.kernel.org/pub/scm/network/tftp/tftp-hpa.git.

[22] 2024. *Wireshark.* The world's most popular network protocol analyzer. https://www.wireshark.org.

[23] Marshall A Beddoe. 2004. Network protocol analysis using bioinformatics algorithms.

[24] Ignacio Bermudez, Alok Tongaonkar, Marios Iliofotou, Marco Mellia, and Maurizio M Munafo. 2015. Automatic protocol field inference for deeper protocol understanding. In *IFIP Networking*. 1–9. https://doi.org/10.1109/IFIPNetworking.2015.7145307

[25] Ignacio Bermudez, Alok Tongaonkar, Marios Iliofotou, Marco Mellia, and Maurizio M Munafò. 2016. Towards automatic protocol field inference. *ComCom* 84 (2016), 40–51. https://doi.org/10.1016/j.comcom.2016.02.015

[26] Georges Bossert, Frédéric Guihéry, and Guillaume Hiet. 2014. Towards automated protocol reverse engineering using semantic information. In *ASIA CCS*. 51–62. https://doi.org/10.1145/2590296.2590346

[27] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Xiaodong Song. 2009. Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In *CCS*. 621–634. https://doi.org/10.1145/1653662.1653737

[28] Juan Caballero and Dawn Song. 2007. Rosetta: Extracting protocol semantics using binary analysis with applications to protocol replay and NAT rewriting. *Cylab* (2007).

[29] Juan Caballero and Dawn Song. 2013. Automatic protocol reverse-engineering: Message format extraction and field semantics inference. *COMNET* 57, 2 (2013), 451–474. https://doi.org/10.1016/J.COMNET.2012.08.003

[30] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Xiaodong Song. 2007. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *CCS*. 317–329. https://doi.org/10.1145/1315245.1315286

[31] Jared Chandler, Adam Wick, and Kathleen Fisher. 2023. BinaryInferno: A Semantic-Driven Approach to Field Inference for Binary Message Formats. In *NDSS*. https://doi.org/10.14722/ndss.2023.23131

[32] Yige Chen, Tianning Zang, Yongzheng Zhang, Yuan Zhou, Peng Yang, and Yipeng Wang. 2021. Inspector: A Semantics-Driven Approach to Automatic Protocol Reverse Engineering. In *CollaborateCom*. 348–367. https://doi.org/10.1007/978-3-030-92635-9_21

[33] Chia Yuan Cho, Domagoj Babic, Eui Chul Richard Shin, and Dawn Song. 2010. Inference and analysis of formal models of botnet command and control protocols. In *CCS*. 426–439. https://doi.org/10.1145/1866307.1866355

[34] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Krügel, and Engin Kirda. 2009. Prospex: Protocol Specification Extraction. In *S&P*. 110–125. https://doi.org/10.1109/SP.2009.14

[35] Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. 2007. Discoverer: Automatic Protocol Reverse Engineering from Network Traces. In *USENIX Security*. https://doi.org/10.5555/1362903.1362917

[36] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irún-Briz. 2008. Tupni: automatic reverse engineering of input formats. In *CCS*. 391–402. https://doi.org/10.1145/1455770.1455820

[37] Adam Hahn. 2016. Operational technology and information technology in industrial control systems. *Cyber-security of SCADA and other industrial control systems* (2016), 51–68. https://doi.org/10.1007/978-3-319-32125-7_4

[38] Zhihao Hu and Zulie Pan. 2021. A Systematic Review of Network Protocol Fuzzing Techniques. In *IMCEC*, Vol. 4. 1000–1005. https://doi.org/10.1109/IMCEC51613.2021.9482063

[39] Yuyao Huang, Hui Shu, Fei Kang, and Yan Guang. 2022. Protocol Reverse-Engineering Methods and Tools: A Survey. *ComCom* 182 (2022), 238–254. https://doi.org/10.1016/J.COMCOM.2021.11.009

[40] Obinna Igbe, Ihab Darwish, and Tarek N. Saadawi. 2017. Deterministic Dendritic Cell Algorithm Application to Smart Grid Cyber-Attack Detection. In *CSCloud*. 199–204. https://doi.org/10.1109/CSCLOUD.2017.12

[41] Junhyoung Kim, TaeGuen Kim, and Eul Gyu Im. 2014. Survey of dynamic taint analysis. In *ICNIDC*. 269–272. https://doi.org/10.1109/ICNIDC.2014.7000307

[42] Stephan Kleber, Henning Kopp, and Frank Kargl. 2018. NEMESYS: Network Message Syntax Reverse Engineering by Analysis of the Intrinsic Structure of Individual Messages. In *WOOT*.

[43] Tammo Krueger, Nicole Krämer, and Konrad Rieck. 2010. ASAP: Automatic Semantics-Aware Analysis of Network Payloads. In *PSDML (Lecture Notes in Computer Science, Vol. 6549)*. 50–63. https://doi.org/10.1007/978-3-642-19896-0_5

[44] Gergo Ládi, Levente Buttyán, and Tamás Holczer. 2018. Message Format and Field Semantics Inference for Binary Protocols Using Recorded Network Traffic. In *SoftCOM*. 1–6. https://doi.org/10.23919/SOFTCOM.2018.8555813

[45] Vladimir Likic. 2008. The Needleman-Wunsch algorithm for sequence alignment. *Lecture given at the 7th Melbourne Bioinformatics Course, Bi021 Molecular Science and Biotechnology Institute, University of Melbourne* (2008), 1–46.

[46] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. 2008. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In *NDSS*, Vol. 8. 1–15.

[47] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic Reverse Engineering of Data Structures from Binary Execution. In *NDSS*. Article 5, 1 pages.

[48] Min Liu, Chunfu Jia, Lu Liu, and Zhi Wang. 2013. Extracting Sent Message Formats from Executables Using Backward Slicing. In *EIDWT*. 377–384. https://doi.org/10.1109/EIDWT.2013.71

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

[50] Rongkuan Ma, Hao Zheng, Jingyi Wang, Mufeng Wang, Qiang Wei, and Qingxian Wang. 2022. Automatic protocol reverse engineering for industrial control systems with dynamic taint analysis. *FITEE* 23, 3 (2022), 351–360. https://doi.org/10.1631/FITEE.2000709

[51] Moti Markovitz and Avishai Wool. 2017. Field classification, modeling and anomaly detection in unknown CAN bus networks. *VEHCOM* 9 (2017), 43–52. https://doi.org/10.1016/J.VEHCOM.2017.02.005

[52] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *NDSS*. https://doi.org/10.14722/ndss.2024.24556

[53] John Narayan, Sandeep K. Shukla, and T. Charles Clancy. 2016. A Survey of Automatic Protocol Reverse Engineering Tools. *CSUR* 48, 3 (2016), 40:1–40:26. https://doi.org/10.1145/2840724

[54] Krunal Patel and Kajal T. Claypool. 2006. SUSAX: Context-Specific Searching in XML Documents Using Sequence. In *ICDE*. 84. https://doi.org/10.1109/ICDEW.2006.141

[55] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *ICST*. 460–465. https://doi.org/10.1109/ICST46399.2020.00062

[56] Johannes Pohl and Andreas Noack. 2019. Automatic wireless protocol reverse engineering. In *WOOT*.

[57] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. 2023. NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing. *TOSEM* 32, 6 (2023), 160:1–160:26. https://doi.org/10.1145/3580598

[58] Claude Elwood Shannon. 1948. A mathematical theory of communication. *The Bell system technical journal* 27, 3 (1948), 379–423.

[59] Kyu-Seok Shim, Young-Hoon Goo, Min-Seob Lee, and Myung-Sup Kim. 2020. Clustering method in protocol reverse engineering for industrial protocols. *NEM* 30, 6 (2020). https://doi.org/10.1002/NEM.2126

[60] Rui Wang, XiaoFeng Wang, Kehuan Zhang, and Zhuowei Li. 2008. Towards automatic reverse engineering of software security configurations. In *CCS*. 245–256. https://doi.org/10.1145/1455770.1455802

[61] Xiaowei Wang, Kezhi Lv, and Bo Li. 2020. IPART: an automatic protocol reverse engineering tool based on global voting expert for industrial protocols. *International Journal of Parallel, Emergent and Distributed Systems* 35, 3 (2020), 376–395. https://doi.org/10.1080/17445760.2019.1655740

[62] Yipeng Wang, Xiaochun Yun, M Zubair Shafiq, Liyan Wang, Alex X Liu, Zhibin Zhang, Danfeng Yao, Yongzheng Zhang, and Li Guo. 2012. A semantics aware approach to automated reverse engineering unknown protocols. In *ICNP*. 1–10. https://doi.org/10.1109/ICNP.2012.6459963

[63] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. 2009. ReFormat: Automatic Reverse Engineering of Encrypted Messages. In *ESORICS (Lecture Notes in Computer Science, Vol. 5789)*. 200–215. https://doi.org/10.1007/978-3-642-04444-1_13

[64] Chenglong Yang, Cai Fu, Yekui Qian, Hong Yao, Guanyun Feng, and Lansheng Han. 2020. Deep Learning-Based Reverse Method of Binary Protocol. In *SPDE (Communications in Computer and Information Science, Vol. 1268)*. 606–624. https://doi.org/10.1007/978-981-15-9129-7_42

[65] Yapeng Ye, Zhuo Zhang, Fei Wang, Xiangyu Zhang, and Dongyan Xu. 2021. NetPlier: Probabilistic Network Protocol Reverse Engineering from Message Traces. In *NDSS*. https://doi.org/10.14722/NDSS.2021.24531

[66] Xiaohan Zhang, Cen Zhang, Xinghua Li, Zhengjie Du, Yuekang Li, Yaowen Zheng, Yeting Li, Bing Mao, Yang Liu, and Robert H. Deng. 2024. A Survey of Protocol Fuzzing. *CoRR* abs/2401.01568 (2024). https://doi.org/10.48550/ARXIV.2401.01568

[67] Sen Zhao, Jinfa Wang, Shouguo Yang, Yicheng Zeng, Zhihui Zhao, Hongsong Zhu, and Limin Sun. 2022. ProsegDL: Binary Protocol Format Extraction by Deep Learning-based Field Boundary Identification. In *ICNP*. 1–12. https://doi.org/10.1109/ICNP55882.2022.9940264

Appendix.A Semantic Types and Functions in Boofuzz and Peach

A.1 Semantics supported by Boofuzz

Boofuzz supports all the semantics summarized in Table 1, which includes five types and six functions.

The five types are:

- Integer: it represents a number of variable length. It is an abstraction of `BitField`, `Byte`, `Word`, `DWord`, and `QWord` in Boofuzz.
- Static: it is fixed and does not vary with specific content, which is equivalent to `Static` in Boofuzz.
- Group: it comprises a list of available values, which is equivalent to `Group` in Boofuzz.
- Bytes: it denotes a sequence of binary bytes of arbitrary length, which is equivalent to `Bytes` in Boofuzz.
- String: it denotes a sequence of characters of arbitrary length, which is equivalent to `String` in Boofuzz.

The six functions are:

- Command: it denotes the message type, which distinguishes the format of messages with different types in Boofuzz.
- Length: it records the size of the message or data, which is equivalent to `Size` in Boofuzz.
- Filename: it refers to a file within the file system, which is equivalent to `Fromfile` in Boofuzz.
- Delim: it indicates the end of a text protocol field, which is equivalent to `Delim` in Boofuzz.
- Checksum: it verifies the integrity of messages or data, which is equivalent to `Checksum` in Boofuzz.
- Aligned: it aligns data, which is equivalent to `Aligned` in Boofuzz.

A.2 Semantics supported by Peach

Peach supports three types and four functions in Table 1.

The three types are:

- Integer: Peach has `Flag`/`Flags` field, which defines a specific bit field, and `Number` that defines a binary number of the specified length. They serve as Integer fields.
- Group: it refers to the `Choice` field.
- String: it refers to the `String` field.

The four functions are:

- Command: it denotes the message type, which distinguishes the `DataModel` in Peach.
- Length: the *Size-of Relation* in `Relation` field is consistent with the Length function.
- Checksum: the *Checksum Fixups* in `Fixup` field provides the Checksum function.
- Aligned: it refers to the `Padding` field in Peach.

Note that, Peach also supports two additional functions: the `Placement` function determines the movement of specific elements after parsing the input stream; and the `Transformer` function performs static transformations or encoding on the parent element.

Appendix.B Extending prior work with our library of atomic semantic detectors

It is fundamentally impossible for prior work to achieve similar semantic inference results if we extend them with our own library of atomic semantic detectors. Because the semantic inference relies on the quality of format extraction. To this end, we did an additional experiment by integrating our own library of atomic semantic detectors (Section 3.4) into Polyglot, AutoFormat, and Tunpi. The evaluation results are shown in Table 8. On average, Polyglot, AutoFormat and Tunpi only achieved the F1-scores of 0.34, 0.28, and 0.25 for type inference, and 0.22, 0.23, and 0.22 for function inference, respectively. In comparison, BinPRE achieved the F1-scores of 0.53 (0.46) for type inference, and 0.64 (0.45) for function inference w/ (w/o) semantic refinement, respectively.

Appendix.C Justifying the similarity threshold used in the Needleman-Wunsch (NW) algorithm

The standard convention often sets the similarity in the Needleman-Wunsch (NW) algorithm at 0.8. To further study the effectiveness of this threshold on BinPRE, we explored the 0-1 interval in 0.1 steps to identify the threshold range where BinPRE yielded the most optimal results in evaluating each protocol. For DNS, TFTP, DNP3.0, Modbus, and FTP, the optimal threshold range is 0.1-1.0. For S7comm and Ethernet/IP, the optimal threshold ranges are 0.5-1.0 and 0.6-1.0, respectively. For HTTP, the optimal threshold range is 0.1-0.4. Since 0.8 is within the optimal threshold ranges of most protocols, it is appropriate for our method to set the NW threshold at 0.8.

Appendix.D Results of format extraction of the classic PRE techniques.

Figure 13 illustrates the format extraction results of three classic PRE techniques and BinPRE when applied to process the message in Figure 1.

Polyglot, AutoFormat, and Tupni all suffer from the over-segmentation errors when the bytes belonging to one field are accessed by different instructions, as they follow the common strategy (detailed in Section 2.2). Despite incorporating their own heuristic strategies to mitigate over-segmentation errors, these strategies may unintentionally lead to under-segmentation errors.

Comparatively, BinPRE introduces an instruction-based semantic similarity analysis strategy for message format extraction. This strategy mitigates the errors of both over-segmentation and under-segmentation, and accurately segments the protocol messages.

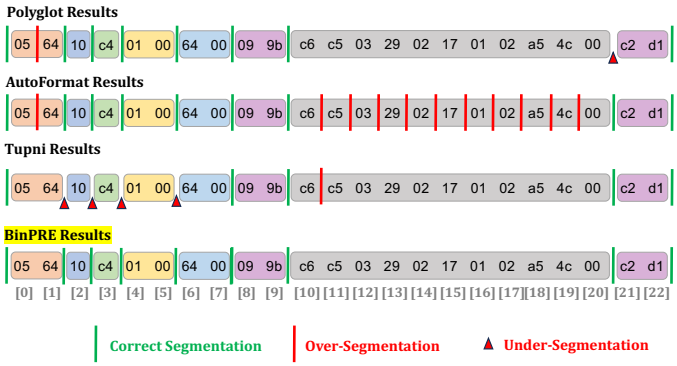

Figure 13: The format extraction results of classic PRE techniques on the example message (the fields with different semantics are annotated by different colors).

Table 8: F1-scores of semantic inference of the prior PRE tools by extending them with our library of atomic semantic detectors.

Protocol	Semantic Type					Semantic Function				
	BinPRE	BinPRE$^-$	Polyglot$^+$	AutoFormat$^+$	Tupni$^+$	BinPRE	BinPRE$^-$	Polyglot$^+$	AutoFormat$^+$	Tupni$^+$
Modbus	0.80	0.70	0.68	0.68	0.52	0.86	0.59	0.38	0.38	0.34
S7comm	0.36	0.38	0.47	0.45	0.47	0.55	0.44	0.23	0.23	0.23
Ethernet/IP	0.63	0.53	0.48	0.19	0.45	1.00	0.54	0.24	0.23	0.23
DNP3.0	0.55	0.49	0.42	0.31	0.00	0.76	0.55	0.40	0.36	0.44
DNS	0.48	0.50	0.47	0.24	0.37	0.27	0.24	0.23	0.18	0.27
FTP	0.27	0.28	0.00	0.19	0.00	0.55	0.53	0.09	0.30	0.00
TFTP	0.79	0.54	0.00	0.00	0.00	0.80	0.40	0.00	0.00	0.00
HTTP	0.35	0.28	0.18	0.18	0.18	0.35	0.34	0.18	0.17	0.21
Average-50	0.53	0.46	0.34	0.28	0.25	0.64	0.45	0.22	0.23	0.22

* BinPRE$^-$ is a variant of BinPRE, which omits semantic refinement module.
* Polyglot$^+$ is a variant of Polyglot, which is extended to include our library.
* AutoFormat$^+$ is a variant of AutoFormat, which is extended to include our library.
* Tupni$^+$ is a variant of Tupni, which is extended to include our library.

Table 9: Protocols and messages of our benchmark suite.

Protocol	Content	Scenario	Project	Server Under Testing	Message Source & # of Message Types	
Modbus	binary	control	freemodbus[5]	/demo/LINUXTCP/tcpmodbus	Open-Source traces[10]	7
S7comm	binary	control	snap7[20]	/examples/cpp/x86_64-linux/server	Open-Source traces[11]	5
Ethernet/IP	binary	control	OpENer[14]	/bin/posix/src/ports/POSIX/OpENer	/bin/posix/src/ports/POSIX/OpENer	4
DNP3.0	binary	control	automatak-dnp3[13]	/cpp/examples/outstation/outstation-demo	Open-Source traces[8][40]	3
FTP	text	network	LightFTP[6]	/Source/Release/fftp	Open-Source traces[9]	11
TFTP	mixed	network	tftpd-hpa[21]	/usr/sbin/in.tftpd	Open-Source traces[12]	2
DNS	mixed	network	dnsmasq[4]	/src/dnsmasq	Open-Source traces[10]	2
HTTP	text	network	miniweb[7]	/miniweb	Command line tool `curl`[3]	7

Table 10: Average F1-scores of semantic inference task across 8 protocols achieved by the evaluated PRE tools.

Protocol	Command			Length				Checksum		Delim	
	BinPRE	Polyglot	DynPRE	BinPRE	Polyglot	Tupni	BinaryInferno	BinPRE	Tupni	BinPRE	Polyglot
Modbus	1.00	0.64	0.17	0.74	1.00	1.00	0.00	-	-	-	-
S7comm	1.00	0.00	0.00	0.44	0.25	0.25	0.00	-	-	-	-
Ethernet/IP	1.00	0.00	0.50	1.00	0.00	0.00	1.00	-	-	-	-
DNP3.0	0.00	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	-	-
DNS	0.00	0.00	0.00	-	-	-	-	-	-	0.26	0.00
FTP	0.00	0.00	0.00	-	-	-	-	-	-	0.74	0.00
TFTP	1.00	0.50	0.00	-	-	-	-	-	-	-	-
HTTP	0.00	0.01	0.00	-	-	-	-	-	-	0.78	0.67
Average	0.50	0.14	0.08	0.80	0.56	0.56	0.50	1.0	1.0	0.59	0.22

* "-" means that the protocol does not contain the fields with the corresponding semantic function.