DYNPRE: Protocol Reverse Engineering via Dynamic Inference

Zhengxiong Luo*, Kai Liang†, Yanyang Zhao*, Feifan Wu*, Junze Yu*, Heyuan Shi‡ and Yu Jiang*✉
*KLISS, BNRist, School of Software, Tsinghua University
†School of Computer Science and Engineering, ‡School of Electronic Information, Central South University

***Abstract*—Automatic protocol reverse engineering is essential for various security applications. While many existing techniques achieve this task by analyzing static network traces, they face increasing challenges due to their dependence on high-quality samples. This paper introduces DYNPRE, a protocol reverse engineering tool that exploits the interactive capabilities of protocol servers to obtain more semantic information and additional traffic for dynamic inference. DYNPRE first processes the initial input network traces and learns the rules for interacting with the server in different contexts based on session-specific identifier detection and adaptive message rewriting. It then applies exploratory request crafting to obtain semantic information and supplementary samples and performs real-time analysis. Our evaluation on 12 widely used protocols shows that DYNPRE identifies fields with a perfection score of 0.50 and infers message types with a V-measure of 0.94, significantly outperforming state-of-the-art methods like Netzob, Netplier, FieldHunter, BinaryInferno, and Nemesys, which achieve average perfection and V-measure scores of (0.15, 0.72), (0.16, 0.73), (0.15, 0.83), (0.15, -), and (0.31, -), respectively. Furthermore, case studies on unknown protocols highlight the effectiveness of DYNPRE in real-world applications.**

I. INTRODUCTION

Protocol reverse engineering facilitates the understanding of unknown protocol specifications and thus serves as the basis for various security analyses, including fuzzing [14], [20], model checking [18], [16], [28], automatic exploit generation [35], [2], and code generation [54]. For example, recovering the protocol format and state machine enables the generation of legitimate packet sequences for protocol fuzzing and forms the foundation for model checking.

Existing protocol reverse engineering methods can be divided into two main categories: program analysis based and network trace based. Program analysis based approaches employ techniques such as taint analysis to dynamically monitor the internal execution of a protocol application and track the processing of received messages [7], [11], [27]. These methods can achieve high accuracy due to the availability of rich runtime semantics. However, they typically require access to the source code or binary of the protocol implementation, which is not always available. For instance, analyzing protocols in embedded systems presents challenges in acquiring and emulating firmware. Network trace based approaches, on the other hand, take static network traces as input and perform statistical analysis [53], [6], [9], [12], [3]. They use techniques such as message sequence alignment to mine the format characteristics messages exhibit in the traces. While easy to use, these approaches often suffer from low accuracy due to the limited information available in the network traces. First, their effectiveness relies on high-quality network traces that contain diverse protocol messages and cover most protocol features. Providing such traces requires protocol knowledge and is not always feasible due to the distributional biases present in real-world network messages. Second, they frequently encounter difficulties in capturing field semantics not manifested in value changes across messages, which is a blind spot for their employed statistical analysis.

This paper presents DYNPRE, a network trace based protocol reverse engineering tool that introduces dynamic inference for more accurate analysis. Unlike traditional methods that require high-quality static network traces for comprehensive statistical analysis, DYNPRE establishes active communication with the server using carefully constructed probe messages to extract insightful information and acquire additional samples as needed, making it well-suited for input traces with limited information. To achieve this approach, we need to address two challenges. (i) The first challenge is how to interact with the server in the absence of protocol specifications. Since the server is a stateful system that requires well-formed messages in a specific sequence, randomly constructing requests is ineffective. (ii) The second challenge is to design a strategy that is applicable to various protocols while being able to induce diverse server behaviors to facilitate protocol understanding.

For the first challenge, we leverage the initial input traces – records of actual client-server conversations – as a reference. In some cases, simply acting as a client and replicating requests from the input traces is sufficient for proper interaction. However, this approach falls short with protocols that use session-specific identifiers such as cookies. The server dynamically assigns these identifiers to keep track of contextual information that may vary from session to session. Using outdated values from previous dialogues may cause the server to reject the request. It is therefore crucial to ensure the validity of these identifiers during live sessions. To this end, we take a two-stage approach. First, we leverage heuristics to detect all locations where session-specific identifiers occur and determine how subsequent requests reference these values, i.e., their constraint relationships. Based on these inferred rules, we then implement

✉ Yu Jiang is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24083
www.ndss-symposium.org

on-the-fly message rewriting to dynamically detect and extract these identifiers from server responses and adaptively update the corresponding values in requests during the interaction.

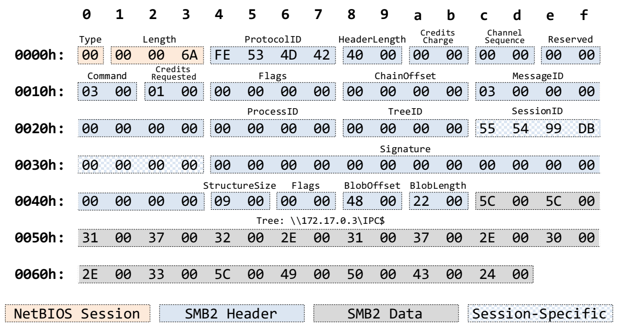
To address the second challenge, we perform modifications to the original requests in the initial traces and observe the server's responses, which serve two purposes. (i) First, these responses reveal the server's interpretation of the requests it receives, providing valuable feedback for request analysis. Since different fields usually exhibit distinct semantics, we approach field identification to identify semantically analogous successive message bytes. To achieve this, DYNPRE modifies each message byte individually and scrutinizes the corresponding server response, the content of which mirrors the internal execution results of the modified message and can, therefore, serve as a semantic indicator of the byte for further field recovery. Meanwhile, to improve the accuracy, we minimize noises stemming from random nonces in the responses and eliminate positional effects of different bytes by analyzing the value propagation under modification. Moreover, considering that the server's input space is tightly regulated by its state, we employ a preprocess to drive the server into the state of accepting the message M to be analyzed, thus maximizing the exposure of insightful feedback regarding M's variants. We achieve this by exploiting the input network trace and the online message rewriting established in the first stage. (ii) Second, these responses can enrich the samples available for statistical analysis. Unlike the traditional passive network trace acquisition through manual operation on protocol parties, which is restricted by protocol specification and runtime context, our proposed active probing approach offers more flexibility. It can cover cases not commonly found in standard conversations, thus helping to uncover hidden nuances for a deeper understanding of the underlying protocol structure. Finally, we introduce a refinement process to further improve the accuracy of the inferred protocol format by synthesizing the results of dynamic inference and statistical analysis. During this process, we also infer the type for each message, facilitating downstream analysis such as state machine inference.

We implement DYNPRE and evaluate it against five state-of-the-art protocol reverse engineering tools on 12 widely used public protocols. The experimental results demonstrate that DYNPRE outperforms prior approaches in both field identification and message type inference. Specifically, DYNPRE identifies fields with a perfection score of 0.50 and infers message types with a V-measure of 0.94, whereas Netzob [6], Netplier [53], FieldHunter [3], BinaryInferno [9], and Nemesys [22] achieve on average (0.15, 0.72), (0.16, 0.73), (0.15, 0.83), (0.15, -), and (0.31, -), respectively. Even when these tools are given datasets enhanced with dynamic interaction, DYNPRE is still superior. In addition, we apply DYNPRE to several unknown protocols employed in real IoT devices to validate its generality. The results show that the messages generated based on the format inferred by DYNPRE effectively trigger behaviors not observed in the initial network traces, demonstrating its effectiveness in practical applications. In summary, our main contributions are as follows:

- We propose the idea of dynamic inference, which exploits the interactive capabilities of the server to obtain additional traffic for protocol reverse engineering.
- We design the adaptive message rewriting to allow proper interaction with the server and propose an intelligent request crafting method to obtain semantic information and supplementary samples for analysis.
- We implement DYNPRE[1] and evaluate it on both public and unknown protocols. The results show that DYNPRE outperforms the state-of-the-art and proves effective in real-world applications.

II. MOTIVATION

We use the SMB2 protocol [29] to illustrate the limitations of traditional network trace based approaches and our work's basic idea and challenges. An example SMB2 message, which typically consists of three main parts as indicated by the different background colors, is shown in Figure 1.



```
        0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
0000h: 00 00 00 6A FE 53 4D 42 40 00 00 00 00 00 00 00
0010h: 03 00 01 00 00 00 00 00 00 00 00 00 03 00 00 00
0020h: 00 00 00 00 00 00 00 00 00 00 00 00 55 54 99 DB
0030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040h: 00 00 00 00 09 00 00 00 48 00 22 00 5C 00 5C 00
0050h: 31 00 37 00 32 00 2E 00 31 00 37 00 2E 00 30 00
0060h: 2E 00 33 00 5C 00 49 00 50 00 43 00 24 00
```

Fig. 1: An example SMB2 request message

Static Network Trace Analysis. Traditional approaches based on static analysis of network traces have two problems:

First, the optimal performance of these approaches usually necessitates high-quality network traces with various protocol messages and extensive feature coverage. For instance, Netzob's partitioning of this message's first four bytes as |0x0000|0x006A| misplaces the highest byte of the *Length* field into the *Type* field. This is because the value of the *Length* field in all input messages is always less than 0x010000. As a result, $M[0..1]$ (where $M[a]$ denotes the a-th byte and $M[a..b]$ indicates the bytes from index a to b) remains 0x0000, while $M[2..3]$ varies. This difference in distribution causes Netzob to erroneously treat $M[0..1]$ as a constant field and $M[2..3]$ as a variable field. Such misidentification of field boundaries can negatively impact protocol comprehension and downstream tasks such as protocol testing and traffic auditing. For instance, misinterpreting the highest byte of a length field as a constant could compromise fuzzing since the length field typically affects the memory behavior of a program. Analogous misidentification also occurs in fields such as the *HeaderLenghth* and *CreditCharge* fields. This problem can be mitigated by increasing message diversity, i.e., providing messages with various values of these fields. However, this process presents significant challenges as it necessitates manual operation in controlled environments based on protocol knowledge and is not always feasible due to the uneven distribution of messages (e.g., some message types are rarely used and observed in practice).

Second, statistical network trace analysis lacks precision in capturing field semantics, especially when these are not

[1]DYNPRE is available at https://github.com/DynPRE/DynPRE

evident in value changes across messages. For instance, the valid values of the *Command* field are drawn from a fixed set. Its highest byte (i.e., $M[17]$ in little-endian) is reserved for customization, and its value is 0x00 in any well-formed SMB2 message, meaning that the semantic of this field is only partially reflected in the value change. As a result, Netzob splits this field into two independent fields, with the $M[17]$ field being considered a constant field. This misinterpretation can hinder misunderstanding the protocol design in code generation [54] or further user extension. Moreover, these methods perform analysis based on byte-value distribution and similarity, which can be misleading since identical or differing byte values do not necessarily indicate semantic similarity or difference. This gap between field semantics and field value's statistical characteristics presents an additional challenge to statistical methods.

Basic Idea of DYNPRE. In most protocol reverse engineering application scenarios, integrating a dynamic approach to exploit the interactive capabilities of the existing server is beneficial. DYNPRE is inspired by two observations:

First, dynamic interaction with the protocol server can generate supplementary samples on top of the original input. By actively generating various probe requests and sending them to the server, we can obtain the response messages, which are legitimate and can enrich the message set for analysis. Unlike the passive generation of input network traces through manual operation on protocol parties, this active probing approach offers more flexibility in ensuring message coverage. In particular, it can cover certain cases not commonly found in standard conversations, which are constrained by the protocol specification and session context. For example, the *MessageID* field $M[28..36]$ uniquely identifies a request/response pair within a session. This field is normally initialized with zero and incremented by one for each new request, so its high bytes are often zero in real conversations. Instead, by actively assigning different values to the *MessageID* field in requests and observing the server's responses, diverse values for this field can be showcased, making it easier for traditional statistical analysis (e.g., Netzob [6]) to uncover hidden nuances and gain a deeper understanding of the underlying structure. In addition, exceptional *MessageID* values can prompt the server to respond with different *CreditsGranted* field values, further increasing message diversity.

Second, dynamic interaction allows for acquiring field semantic information from the server, which already encodes the protocol logic and thus understands the message. Different fields convey distinct semantics, which can be explored by modifying the field content and observing the corresponding server response. For example, consider the request packet in Figure 1. Modifying the *Reserved* field does not affect the server response because it is ignored. For the *Command* field, which specifies the message type, different values of this field can cause the server to respond with different types of messages. Changing the *CreditsRequested* field can affect the number of credits granted by the server, as reflected in the *CreditsGranted* field in the response. These three adjacent fields in the message convey different semantics, which is also reflected in the different server responses to the modifications.

Challenges. To exploit the server's interactive capabilities to obtain valuable information for a more accurate analysis of the target protocol, we must address two challenges:

C.1: Proper interaction with the server. As discussed in Section I, the initial input network traces provide a guideline for interaction, but the session-specific identifiers used in some protocols present additional challenges.

We use an example SMB2 session (Figure 2) to introduce the concept of session-specific identifiers employed in network protocols. This session is initiated with a `NEGOTIATE` exchange (①-②) followed by a `SESSION_SETUP` exchange (③-④). In particular, the value a of the *SessionID* field in message ④ is randomly assigned by the server to identify this unique session and is essential for subsequent communication. All subsequent requests (e.g., $\textcircled{n_1}$ and $\textcircled{n_3}$) in the session must use the same value a for the *SessionID* field; otherwise, the server will reject the request. Once connected, the client can access server resources organized into shares (e.g., file shares) through a `TREE_CONNECT` request ($\textcircled{n_1}$), to which the server responds with a message assigning a unique value b for the *TreeID* field to identify this share *#n* ($\textcircled{n_2}$). Similar to the *SessionID* field, all subsequent requests on share *#n* must use *TreeID* value b until the client disconnects from this share ($\textcircled{n_3}$). For the next share *#(n+1)*, the server would assign a new, different value for the *TreeID* field. In this example, the *SessionID* and *TreeID* fields are session-specific identifiers randomly assigned by the server to identify specific resources within a session. Their values vary from session to session. For a session-specific identifier, we define the *source* as the location that first introduces its dynamic value and the *references* as the subsequent locations that consume that value. In this example, the sources for the *SessionID* and *TreeID* fields are ④ and $\textcircled{n_2}$, respectively, and their references are [$\textcircled{n_1}$, $\textcircled{n_2}$, $\textcircled{n_3}$] and [$\textcircled{n_3}$].

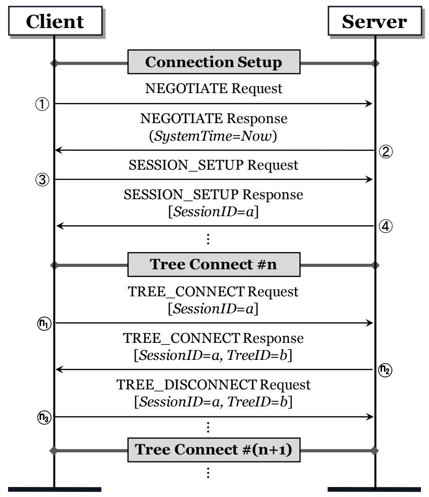

Client
Server
Connection Setup
①
NEGOTIATE Request
NEGOTIATE Response (SystemTime=Now)
②
③
SESSION_SETUP Request
SESSION_SETUP Response [SessionID=a]
④
Tree Connect #n
n1
TREE_CONNECT Request [SessionID=a]
TREE_CONNECT Response [SessionID=a, TreeID=b]
n2
TREE_DISCONNECT Request [SessionID=a, TreeID=b]
n3
Tree Connect #(n+1)

Fig. 2: An example SMB2 session, with session-specific identifiers in brackets and random fields in parentheses.

The session-specific identifier is a common mechanism applied in network protocols. By examining 30 public and 20 proprietary protocols, we found that 10 public and 6 proprietary protocols contain session-specific identifiers. Typical examples include TCP's *SequenceNumber* field [40], HTTP's *Cookie* field [39], SSL's *SessionID* field [38], and FTP's *PassivePort* field [41]. Our experiments on real-world proprietary protocols in Section IV-E also show their use in practice. Without knowing the protocol specification, correctly detecting all identifiers and adjusting them to the valid value in a live session is challenging due to their unpredictable sources and references, diverse constraint relationships between sources and references, and discrepant lifetimes (e.g., the *SessionID* and *TreeID* fields in Figure 2). The large number of sources and references detected in real sessions, as shown in Section IV-D, further demonstrates this.

C.2: Effective exploration of the interactive server for protocol understanding. The second challenge is to effectively elicit and analyze diverse server behaviors. On the one hand, constructing the probe requests to elicit diverse server behaviors is challenging for two reasons. First, to obtain in-depth semantic feedback on the probe messages, we must ensure that the server is in the appropriate state. Otherwise, the server may reject the requests due to state mismatch before analyzing the request in detail. Second, since the protocol messages are highly structured, arbitrary modifications to the existing messages tend to destroy the structure and are less likely to provide valuable results. On the other hand, analyzing the server responses is also challenging. Although these responses contain rich semantic information, this information remains opaque as a comprehensive interpretation requires protocol knowledge. In addition, random noise in the server responses can disrupt the analysis. For example, in Figure 2, the *SystemTime* field in the `NEGOTIATE` response (②) records the server time, which is unrelated to the protocol semantics but varies over time and may interfere with the analysis.

III. System Design

In this section, we first introduce the workflow of DynPRE and then present the attack model and design details.

Overview. Figure 3 gives an overview of DynPRE: Given the captured network traces of the target protocol, DynPRE interacts with the protocol server to infer the protocol format and state machine.

Filtering and Slicing. We first pre-process the input network traces to extract the relevant messages and group them into different traces. The input traces are typically captured during the communication between the protocol server and the client. These traces usually contain messages from irrelevant protocols and multiple sessions of the target protocol. Since the target protocol is typically in the application layer and the underlying layers (e.g., TCP, IP) are standardized, we can leverage the known information of these underlying layers (e.g., IP address, port number, and timestamp) for filtering to focus on analyzing the target protocol. For example, for TCP-based protocols, we can use the TCP port associated with the target protocol to filter out the relevant messages and partition them into different traces based on TCP sessions.

After pre-processing, DynPRE analyzes each network trace sequentially. At a high level, DynPRE consists of two main components. We use the example SMB2 session in Figure 2 to illustrate them as follows.

Session-Specific Identifier Detection. For the trace $\mathit{\Gamma}$ under analysis, this module recognizes all embedded identifiers and extracts their detailed attributes through recursive analysis and validation. Essential attributes of each session-specific identifier include its source, references, and their constraint relationship. These details form $\mathit{\Gamma}$'s message rewrite rules, which are then passed to the dynamic inference module. Table I shows the derived message rewrite rules for the example SMB2 session, where two identifiers are recognized and they correspond to the *SessionID* and *TreeID* fields, respectively. For the first, the source is the bytes $M[44..51]$ in message ④, and the corresponding references are identical bytes in messages $\textcircled{r_1}$, $\textcircled{r_2}$, and $\textcircled{r_3}$. The constraint relationship between the source and its three references is $y = x$, indicating that the value of these references should be consistent with that of the source.

TABLE I: Example message rewrite rules

No.	Source	References	Constraint
1	④: [44..51]	$\textcircled{r_1}$: [44..51], $\textcircled{r_2}$: [44..51], $\textcircled{r_3}$:[44..51]	$y = x$
2	$\textcircled{r_2}$: [40..43]	$\textcircled{r_3}$: [40..43]	$y = x$

Dynamic Inference. Based on the message rewrite rules, DynPRE establishes an on-the-fly message rewriting mechanism that enables proper interaction with the server by mimicking the network trace $\mathit{\Gamma}$. The dynamic inference module infers the protocol format by modifying the initial requests in $\mathit{\Gamma}$, retrieving their corresponding responses, and performing real-time analysis. These responses serve as semantic feedback for the request analysis and provide additional samples for the response analysis. Based on the inferred respective format results of the requests and responses, DynPRE employs a mapping based refinement to improve the overall format results and recognize the message types.

Finally, we employ established methods [19], [5], [53] to infer the state machine. These methods are efficient when the message types are well defined. The basic idea is to derive the message type sequence for each network trace and regard it as a state transition sequence. These sequences are combined to form an original state machine, which is later minimized for conciseness. We omit the details as they are not our contribution. The quality of the inferred state machine depends on the accuracy of the message type inference, for which we provide an evaluation in Section IV.

Attack Model. We assume an active attacker model for our approach. First, aligning with the passive attacker model used in traditional network trace based approaches, our active attacker can sniff traffic to obtain the initial network traces for analysis. Second, our active attacker can actively communicate with the server, i.e., send arbitrary requests and obtain responses. This assumption is realistic because even passive attackers, after analyzing the static network traces, usually need to achieve their goals through active attacks, e.g., sending maliciously crafted messages.

A. Session-Specific Identifier Detector

For a network trace $\mathit{\Gamma}$, this module detects session-specific identifiers within $\mathit{\Gamma}$ and learns a set of message rewrite rules $\mathit{\Upsilon}$ for live communication with the server.

Algorithm 1 provides an overview of the process. The session-specific identifier detector starts with an empty rule set (Line 2) and incrementally learns the rewrite rules via the `inferAndVerify` procedure (Lines 3-21). Each time this procedure is invoked, the detector establishes a new session with the server and stores live responses in $\mathit{LiveResponsePool}$. In each session, it tries to replay the network trace $\mathit{\Gamma}$ with the current rewrite rules $\mathit{\Upsilon}_{cur}$ (Line 5-19). For a request M in $\mathit{\Gamma}$ (e.g., $\textcircled{r_1}$ in Figure 2), before

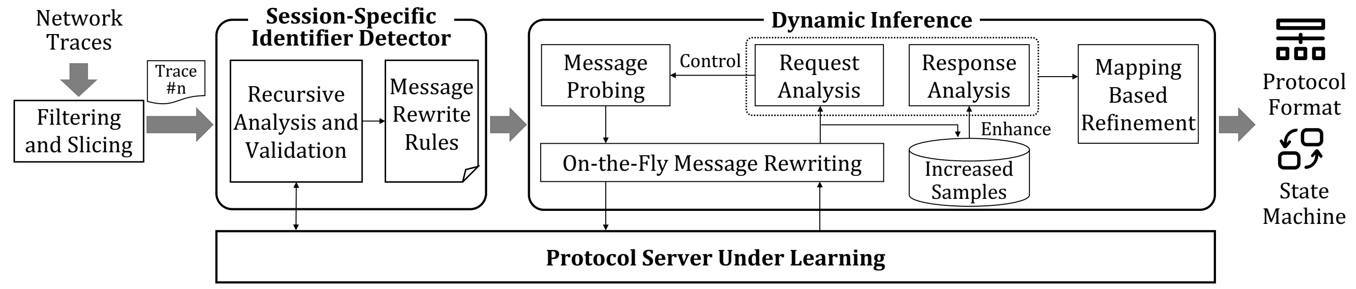


Fig. 3: DYNPRE Overview. For each network trace *#n*, the Session-Specific Identifier Detector first recognizes all session-specific identifiers and derives message rewrite rules. Based on this, the Dynamic Inference module interacts with the server to gain semantic information and additional samples via message probing and applies mapping based refinement to improve the results.

Algorithm 1: Recursive analysis and validation for session-specific identifiers

```
Input  : $\mathit{\Gamma}$ - network trace under analysis
         $Server$ - protocol server under learning
Output: $\Upsilon$ - learned set of message rewrite rules
1  Algorithm
2      inferAndVerify($\varnothing$)
3  Procedure inferAndVerify($\Upsilon_{cur}$)
4      $LiveResponsePool \leftarrow []$
5      foreach message $M$ in $\mathit{\Gamma}$ do
6          if $M$ is request then
7              $P_r = $ REWITE($M, \Upsilon_{cur}, LiveResponsePool$)
8              SENDTO($Server, P_r$)
9          else
10             $M' = $ RECVFROM($Server$)
11             if not IDENTICAL($M, M'$) then
12                 if ISRECORDED($M, \Upsilon_{cur}$) then
13                     $LiveResponsePool.append(M')$
14                 else if CHECKCOMPATIBILITY($M, M'$) then
15                     $\Upsilon_M = $ calculateRules($M, M'$)
16                     foreach rewrite rule $\upsilon$ in $\Upsilon_M$ do
17                         if inferAndVerify($\Upsilon_{cur} \cup \upsilon$) then
18                             return True
19                 return False
20     $\Upsilon = \Upsilon_{cur}$
21     return True
22 Procedure calculateRules($M$, $M'$)
23     $\Upsilon_M \leftarrow \varnothing$
24     $DynRegions = $ GETDIFFREGIONS($M, M'$)
25     foreach region $R$ in $DynRegions$ do
26         $\Upsilon_R = $ SOLVEANDDERIVERULES($\mathit{\Gamma}, R$)
27         $\Upsilon_M = $ COMBINATION($\Upsilon_M, \Upsilon_R$)
28     return $\Upsilon_M$
```

sending it, the detector rewrites it to P_r by using the rules recorded in Υ_{cur} and leveraging the live responses from $LiveResponsePool$ (Lines 6-8). If there is no recorded rule for M, P_r remains the same as M. For a response M in $\mathit{\Gamma}$, the detector attempts to emulate by receiving the live response M' from the server and verifying that whether M and M' are identical (Lines 10-11). If they differ, M may contain session-specific identifiers. For example, for ④ in Figure 2, the message recorded in $\mathit{\Gamma}$ differs from the corresponding one in the live session since the *SessionID* field is dynamically assigned. In this case, the session-specific identifier detector first checks if M is recorded in the rewrite rules Υ_{cur} (Line 12). If it is, this indicates that the session-specific identifiers of M have been inferred in a previous procedure, and the associated rewrite rules need further validation. Hence, the session-specific identifier detector adds M' to $LiveResponsePool$ for subsequent message rewriting and proceeds to the following messages in $\mathit{\Gamma}$ (Line 13). Otherwise, the session-specific identifier detector checks the compatibility between messages M and M', i.e., whether they are of the same length and whether their differing bytes are concentrated in a few continuous segments rather than randomly scattered throughout. (Line 14). If compatible, the session-specific identifier detector calculates all possible message rewrite rules involving M, namely Υ_M, using the `calculateRules` procedure (Lines 15, 22-28). Then, the session-specific identifier detector iterates through each rule suite υ in Υ_M, attempting to recursively validate and infer with an updated rule set, $\Upsilon_{cur} \cup \upsilon$ (Line 17). If all attempts to replay $\mathit{\Gamma}$ fail (Lines 16-18) or M and M' are incompatible (Line 14), the session-specific identifier detector returns False (Line 19), indicating that the given rewrite rules Υ_{cur} are incorrect. When the session-specific identifier detector successfully replays all messages in $\mathit{\Gamma}$ with Υ_{cur}, it records Υ_{cur} as the learned final set of rewrite rules and returns True (Lines 20-21).

The `calculateRules` procedure takes as input two messages, the original M in $\mathit{\Gamma}$ and its variant M' in live communication, and computes the rules involved, Υ_M. It compares M and M', identifies the differing byte regions between them (Line 24), and analyzes these regions one by one (Lines 25-27). For each region R, the procedure solves constraints on R based on the analysis of $\mathit{\Gamma}$ and derives a set of feasible rewrite rules Υ_R (Line 26). Through our initial study of session-specific identifiers, we find that the constraint relationships between the source and references are typically additive or multiplicative. Therefore, we define a constraint-solving list l_c with five terms: $[x, x+1, px, px+1, null]$. Specifically, given the value v_R of R in M, we infer the possible reference of this value in the messages following M in $\mathit{\Gamma}$ by trying each term in l_c and endianness (i.e., big- or little-endian). In particular, we add the term $null$ for the constraint-solving list to indicate that no constraint is needed for the corresponding region, which is designed to reduce noise from fields with random values, such as the *SystemTime* field in Figure 2. Finally, the procedure combines the existing rules Υ_M of analyzed regions with the newly derived rules Υ_R (Line 27). For example, assume there are two dynamic regions α and β by comparing M and M', and their feasible rules are $[\Upsilon_\alpha^1, \Upsilon_\alpha^2]$

and $[\varUpsilon_\beta^1, \varUpsilon_\beta^2]$, respectively, then the derived rules $\varUpsilon_P$ for the message M are $[\varUpsilon_\alpha^1 \cup \varUpsilon_\beta^1, \varUpsilon_\alpha^1 \cup \varUpsilon_\beta^2, \varUpsilon_\alpha^2 \cup \varUpsilon_\beta^1, \varUpsilon_\alpha^2 \cup \varUpsilon_\beta^2]$.

B. Dynamic Inference

The message rewrite rules allow DYNPRE to interact seamlessly with the server and perform dynamic inference. For the input network trace $\varGamma$, DYNPRE sequentially analyzes each request/response pair (M, M°) within $\varGamma$. Specifically, DYNPRE crafts probe messages by subtly modifying M, then monitors the server's responses to these modifications, which allows DYNPRE to infer the format of M. Meanwhile, these newly triggered responses, which are retrieved directly from the server without modification and are therefore legitimate (instead of being modified like the probe messages), can be considered as additional samples for analyzing M°. Given that the server generates M° and the increased samples in response to marginally different requests under the same server state, these responses are likely to be roughly similar but with specific variations. This resemblance facilitates traditional statistical analysis, such as sequence alignment, to uncover hidden nuances for format analysis. In this process, requests are analyzed by active probing, and responses are analyzed using statistical analysis. To synthesize the results of both analyses for further improvement, DYNPRE performs a mapping based refinement that simultaneously recognizes the message types.

On-the-Fly Message Rewriting. Following the message rewrite rules $\varUpsilon$, the dynamic inference module establishes an adaptive message rewriting mechanism that extracts the value of session-specific identifiers and adjusts the corresponding byte regions in the requests during live interaction. Specifically, (i) for a response α from the server, if α serves as the source of a particular session-specific identifier, this submodule extracts and stores the corresponding value for future use. For example, as shown in Figure 2 and Table I, [44..51] in message ④ is the source of the *SessionID* identifier and its value should be recorded for subsequent request rewriting; (ii) for a request β to be sent, if β contains references to certain session-specific identifiers (e.g., [44..51] in $\textcircled{n_1}$ in Table I), this submodule adjusts the relevant byte regions in β using the recorded values and associated constraints before transmission.

Message Probing. We analyze the format of each request M in $\varGamma$ using byte-level flip modifications. This is based on three insights: (i) The smallest field unit in most protocols is the byte, so byte-level field identification is appropriate. To our knowledge, all existing tools use bytes as the basic unit for field identification. (ii) Modifying content, rather than adding or deleting, preserves the overall message structure and can provide deeper semantics. For example, for a shallow sanity check such as length field (e.g., $M[1..3]$ and $M[8..9]$ in Figure 1), a request constructed by modifying some other byte in M can pass this check. (iii) Since the semantics of the responses are opaque to us, we focus on detecting semantic changes across varied modifications. Thus, we assign a value that markedly differs from the original to maximize semantic exposure. Having obtained the field boundary, we can obtain further information by applying random modifications at the field level and observing the corresponding responses.

Moreover, given the server's stateful nature, when analyzing a request M, DYNPRE first employs the preceding requests in $\varGamma$ along with the established message rewriting mechanism to drive the server into the state receptive to M. This ensures richer feedback on the variants of M.

Request Analysis. To facilitate the illustration, we first introduce the following definitions. Let M be a request in the network trace $\varGamma$, M° represents its corresponding response in $\varGamma$. $\varGamma_{[:M]}$ denotes the sequence of requests in $\varGamma$ that precede M. For illustration, consider Figure 2. If the request (i.e., M) is $\textcircled{n_1}$, its corresponding response M° is $\textcircled{n_2}$, and the set $\varGamma_{[:M]}$ contains the requests ① and ③. Besides, we use $M_{[i]}$ to denote the probe request constructed by flipping the i-th byte of M and $M_{[a..b]}$ to represent the probe request constructed by randomly modifying the a-th to b-th bytes in M. Further, we define $Q\langle M_{[i]}\rangle$ as the output from the function SENDRECV. This function establishes a new session with the server, sends the requests $\varGamma_{[:M]}$ to drive the server into the appropriate state, and sends $M_{[i]}$, taking the corresponding response to $M_{[i]}$'s as a return value. Notably, the message rewriting is applied to each request to ensure session-specific identifiers' validity.

$$Q\langle M_{[i]}\rangle = \textsc{SendRecv}(\varGamma_{[:M]}, M_{[i]})$$

For a byte in M at index i, i.e., $M[i]$, we define its response pool $R[i]$ as the set of responses to the probe request $M_{[i]}$, i.e., $R[i] = \{Q\langle M_{[i]}\rangle\}^n$. Note that the responses in $R[i]$ might vary, as the server may respond differently to identical requests due to the session-specific identifiers or random fields. To further explore these potential variations, we send the crafted request $M_{[i]}$ to the server multiple times to elicit these random nonces. Based on this, we introduce $mask[i]$, which denotes exhibiting differences across the responses in $R[i]$. We have:

$$mask[i] = \{j \mid \exists r_1, r_2 \in R[i], r_1[j] \neq r_2[j]\}$$

Algorithm 2 gives an overview of the request analysis. For a request M, it outputs the inferred format $\mathbb{L}$. Each field in $\mathbb{L}$ is represented as a tuple $M[s..e] \diamond T$, indicating that the field lies in the bytes indexed from s to e, and its type is T.

Algorithm 2: Request Analysis

```
Input  : $M$ - request message under analysis
Output: $\mathbb{L}$ - inferred message format
1  Algorithm
2      $\mathbb{L} \leftarrow []; s \leftarrow 0$
3      $\textsc{InitWithEmptySet}(R); \textsc{InitWithEmptySet}(mask)$
4      $R[0] = \{Q\langle M_{[0]}\rangle\}$
5      for $0 < i < |M|$ do
6          $R[i] = \{Q\langle M_{[i]}\rangle\}^2$
7          $mask[i] = \textsc{CalculateMask}(R[i])$
8          $R[s].append(Q\langle M_{[s]}\rangle)$
9          $mask[s] = \textsc{CalculateMask}(R[s])$
10         if (not $\textsc{Compatible}(mask[i], mask[s])$) or
             $\textsc{Discrepant}(R[i], R[s], mask[s])$ then
11             $e = i - 1$
12             $T = \textsc{InferType}(\{Q\langle M_{[s..e]}\rangle\}^n, M^\circ)$
13             $\mathbb{L}.append(M[s..e] \diamond T)$
14             $s = i$
15     return $\mathbb{L}$
```

For Algorithm 2 in detail, we sequentially analyze each byte in M and determine whether it is the start of a new field. We use s to denote the start byte of the current field. Initially, we set s to zero, since the zeroth byte is always the start of the first field (Line 2), and initialize its response pool $R[0]$ (for $M_{[0]}$) for further use (Line 4). We then iterate the remaining bytes in M by comparing their semantics with that of byte s. For a byte at index i, we first initialize its response pool, $R[i]$, with the two responses to $M_{[i]}$ by sending $M_{[i]}$ twice, which enabling the calculation of $mask[i]$ (Lines 6-7). Then, we employ a similar method to acquire an additional response for $R[s]$ (Line 8) and update the associated $mask[s]$ (Line 9). This ensures the time interval between the first and last responses in $R[s]$ encompasses the period between the first and last responses in $R[i]$, which is designed to eliminate the effects of random fields like timestamps (e.g., the *SystemTime* field in Figure 2) in the responses.

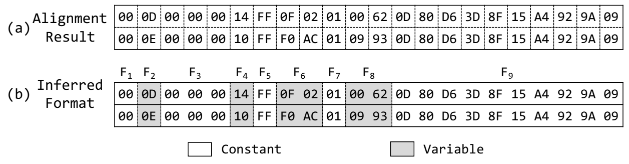
To check whether byte i and byte s are semantically identical, we use a two-step check: first, we compare their masks (Line 10). The $mask[i]$ is expected to be a subset of the $mask[s]$ since the latter covers a longer period; otherwise, it indicates that the random fields exhibited in $R[i]$ are different from those in $R[s]$, and thus the responses in $R[i]$ and $R[s]$ are not semantically identical. We then compare their corresponding response pools, i.e., $R[s]$ and $R[i]$, without considering the regions in $mask[s]$ (Line 10). A simple comparison of the content may miss the differences caused by their position differences instead of semantic differences. For example, Figure 4 shows the initial response M° for the request M in Figure 1 and two responses for $M_{[10]}$ and $M_{[11]}$. However, these two bytes belong to the same field, i.e., the *CreditsCharge* field. Since the server will return a response that contains the same value for the *CreditsCharge* field as the request, separately flipping these two bytes can cause the server to return responses with different content. However, these responses are semantically identical. Therefore, the differences introduced by their different locations should be ignored. We achieve this by observing the value propagation between the request and response under request modification. In this example, if the 10th byte in the request M is flipped, the 10th byte in its corresponding response, i.e., $Q\langle M_{[10]}\rangle$, will also be flipped. This is also the case for the 11th byte. These two bytes are consecutive in M, and we can conclude they are semantically identical. Following the above steps, if the check fails, we argue that byte s and byte i belong to different fields and byte i is the start of a new field (Lines 11-14).

$$\begin{aligned}
M^\circ : &\quad \texttt{00000050fe534d424000}\underline{\texttt{0000}}\texttt{00000000} \\
Q(M_{[10]}) : &\quad \texttt{00000050fe534d424000}\underline{\texttt{ff}}\texttt{0000000000} \\
Q(M_{[11]}) : &\quad \texttt{00000050fe534d42400000}\underline{\texttt{ff}}\texttt{00000000}
\end{aligned}$$

Fig. 4: A comparison of the first 16 bytes of the responses to the original and the modified requests.

For the newly identified field $M[s..e]$, we identify its semantics by determining basic field type information, i.e., whether a field is constant or variable, given our lack of prior protocol knowledge. (Line 12). This design is consistent with similar work like Netzob [6] and Netplier [53]. We achieve this by randomly modifying the value of this field in the request to obtain multiple responses, i.e., $\{Q\langle M_{[s..e]}\rangle\}^n$. Then, using a semantic check similar to the one above, we check whether all these responses are semantically identical and different from the initial response M°. If so, the field is constant, otherwise it is variable. This is based on the observation that a constant field takes a fixed value, and modifying it will incur the same error response or no response. In contrast, a variable field can take different values, and modifying it can cause diverse messages. For example, the *ProtocolID* field in Figure 1 is constant, and assigning other values to it can always result in no response. However, the *Command* field is a variable field whose valid values are taken from a finite set. Assigning different values to it can lead to different scenarios.

Response Analysis. Unlike the request, which the client actively sends, the response is usually received passively and is hard to probe. We use a statistical method to analyze the response. To analyze the response M°, we use the additional responses obtained from modifying each byte of M, i.e., the responses in R. In practice, the byte-level modifications (which are tiny) to the request M can yield many responses of the same type as M°, and these responses have slightly different details. Alignment-based approaches work well to uncover hidden nuances when clusters of messages of the same type are well-defined. For example, Figure 5(a) shows the alignment results for two messages. We can infer the format by merging consecutive bytes that are either identical or variable, as shown in Figure 5(b), and obtain the primary field types simultaneously. Due to a lack of protocol knowledge, we filter R and retain these responses with the same length as M°, i.e., $R' = \{r \mid r \in R, |r| = |M^\circ|\}$, since these responses are more likely to be of the same type as M°. We then apply the alignment algorithm to R' to infer the format of M°.



```
(a) 00 0D 00 00 00 14 FF 0F 02 01 00 62 0D 80 D6 3D 8F 15 A4 92 9A 09
    00 0E 00 00 00 10 FF F0 AC 01 09 93 0D 80 D6 3D 8F 15 A4 92 9A 09

(b) 00 0D 00 00 00 14 FF 0F 02 01 00 62 0D 80 D6 3D 8F 15 A4 92 9A 09
    00 0E 00 00 00 10 FF F0 AC 01 09 93 0D 80 D6 3D 8F 15 A4 92 9A 09
```

Fig. 5: An example of the inferred format for two messages derived from the alignment results.

Mapping Based Refinement. In the steps above, we analyze the formats of the requests and responses independently. However, the formats of these messages often correlate, as evidenced by the mapping relationships of their field values. Specifically, protocol messages typically have a consistent header structure, and some fields in this header exhibit a one-to-one mapping relationship between the requests and responses. For instance, SMB2 requests and responses share the same header structure shown in Figure 1. The *ProtocolID* field specifies the protocol version and is constant in both requests and responses. The *Command* field identifies the message type, and its request value corresponds to its response value. Based on these observations, we explore the mapping relationships between the requests and responses to search for their common fields and refine the overall format, coupling their results to achieve improvement.

To achieve this, we first examine the field that appears most frequently in all inferred message formats. Assume its index range is $[s, e]$. We then determine whether this field exhibits a correlation property in the network trace $\mathit{\Gamma}$, i.e., looking for an injective function satisfying: $f(M[s..e]) \to M^\circ[s..e], \forall (M, M^\circ) \in \mathit{\Gamma}$. If so, we refine the request and response formats by treating this field as a common field. We repeat these steps until no common field can be found. Meanwhile, during this process, we consider the first common field found to be the message type field of the protocol. This is based on the observation that the request and response message types have an intense correspondence, as illustrated in the SMB2 session in Figure 2.

C. Implementation

We implement a prototype of DYNPRE using Python 3. It consists of two components: the session-specific identifier detector and the dynamic inference module. The session-specific identifier detector implements Section III-A's recursive analysis and validation algorithm. It derives a set of message rewrite rules in JSON format with the attributes listed in Table I, facilitating manual inspection and further extension. The dynamic inference module uses these rules to establish the on-the-fly message rewriting and implements the dynamic inference algorithm from Section III-B. In particular, the response analysis submodule is built upon the sequence alignment algorithm implemented in Netzob [6]. Besides, we implement DYNPRE$^-$, a version without the mapping based refinement strategy from Section III-B, specifically to evaluate the strategy's effectiveness.

IV. EVALUATION

In this section, we evaluate DYNPRE to answer the following four research questions:

RQ1 How does DYNPRE's performance compare to state-of-the-art tools on static datasets of different sizes for public protocols? (Section IV-B)

RQ2 Does DYNPRE demonstrate superiority when providing other state-of-the-art tools with datasets enhanced by dynamic interaction? (Section IV-C)

RQ3 Do the session-specific identifier detector and the refinement strategy contribute to the effectiveness of DYNPRE? (Section IV-D)

RQ4 How effective is DYNPRE in reverse engineering real-world proprietary protocols? (Section IV-E)

A. Experiment Setup

Subjects. Table II shows the protocols selected for evaluation. We selected them by referring to prior research [53], [9] and considering various characteristics. Regarding ownership, S7comm, SMB, and SMB2 are proprietary, initially closed protocols, while the others are open protocols. Concerning content, HTTP is a textual protocol, whereas DNS and TFTP are mixed binary and textual protocols, and the remaining are binary protocols. Regarding field composition, NTP features fixed field lengths, while the others exhibit variable lengths.

Moreover, since DYNPRE employs session-specific identifier detection for dynamic interaction initialization and utilizes server responses for analysis, we selected protocols with session-specific identifiers and varying response granularity to evaluate the effectiveness and scalability of DYNPRE. Among the selected protocols, SMB, SMB2, HTTP, BGP, and TFTP contain session-specific identifiers. The scenarios in SMB and SMB2 are pretty complicated. Within a single session, multiple fields can serve as session-specific identifiers, each with different lifetimes, as illustrated in Figure 2. Although these two protocols are related, they differ in their fields, so we chose both for evaluation. Regarding response granularity, we selected MQTT and AMQP because of their specific characteristics. MQTT provides several levels of Quality of Service (QoS) to achieve different objectives. Under different QoS levels, the granularity of responses varies. We introduce two instances of MQTT, represented as MQTT-QoS1/QoS2, to measure the impact of message granularity on the performance of each tool. Besides, the AMQP server rarely responds to a client's publish request after connection establishment, resulting in a relatively small response sample.

For the protocol server under learning, we utilize off-the-shelf server utilities provided by typical open-source protocol projects or publicly accessible services (e.g., the public NTP time server), as shown in Table VIII. To simulate real-world usage scenarios, we prepared the message dataset using various client utilities provided by each project to interact with the protocol server while capturing the network traces. Meanwhile, for each protocol, we prepared datasets of 10, 100, and 1000 messages to measure the impact of dataset size on the performance of each tool. Further details are given in Appendix B.

Compared Tools. We select five state-of-the-art protocol reverse engineering tools widely used in academia and industry as baselines, including Netzob [6], Netplier [53], FieldHunter [3], BinaryInferno [9], and Nemesys [22]. Their approaches are diverse. Netzob and Netplier represent the alignment-based approach. They use different message clustering algorithms. The former employs message similarity based clustering, while the latter uses keyword based clustering. Nemesys adopts a heuristic approach that identifies field boundaries based on the bit-level congruence of successive byte pairs. FieldHunter identifies fields by leveraging statistical characteristics specific to general fields, such as *hostID*. BinaryInferno ensembles several field boundary detectors and applies heuristics to identify fields. All these tools are publicly available [4], [33], [32], [51], except for FieldHunter, for which we use a public re-implementation version [52].

Metrics. We evaluate the effectiveness of each tool in terms of format inference and message type inference, two critical tasks in protocol reverse engineering. Format inference allows for understanding transmitted messages, and message type inference is the basis for the subsequent protocol automata construction [5]. We use *tshark* [50] for message format ground truth. Based on this, we obtain the truth of the message type by manually inspecting the official protocol documents.

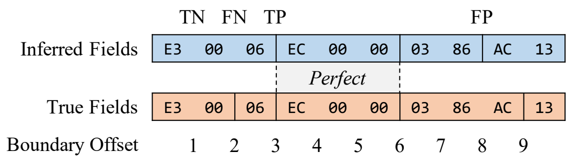


Fig. 6: Illustration of the format inference metrics.

Metrics for format inference. We employ metrics from recent work [53], [9] and divide them into two categories, i.e., one- and two-dimensional metrics, categorized by whether they are from a byte or field perspective. (i) The one-dimensional metric evaluates each byte boundary individually as either a field boundary or not (i.e., a binary classification). The inferred result for a message M is a binary vector V of length $|M|-1$, with each element signifying a byte boundary's status. For instance, Figure 6 shows a message with nine adjacent byte boundaries. The inferred result identifies the 3rd, 6th, and 8th byte boundaries as field boundaries, i.e., $V = [0, 0, 1, 0, 0, 1, 0, 1, 0]$. Moreover, for the ground truth, we define a $positive$ as a true field boundary and a $negative$ as a byte boundary within a true field. Based on these concepts, we represent each adjacent byte boundary in the inferred format using the four outcomes of the binary classification, i.e., true negative (TN), false negative (FN), true positive (TP), and false positive (FP). For the inferred format in Figure 6, the outcomes are $[TN, FN, TP, TN, TN, TP, TN, FP, FN]$. (ii) The two-dimensional metric assesses both byte boundaries of a field, where a field is considered *perfect* if it exactly matches the ground truth. Thus, in Figure 6, the inferred field with two boundaries indexed 3 and 6 is *perfect*.

Based on the concepts above, we use two one-dimensional metrics, *accuracy* and *F1-score* (a combination of *precision* and *recall*), and a two-dimensional metric, *perfection*, for evaluation. They are calculated as Formula 1-5. As discussed in Section II, incorrect format inference – such as incorrect partitioning of a true field, treating multiple true fields as one, or redistributing portions of bytes from one field to other fields – can confuse protocol understanding and affect various subsequent analyses, such as interpreting relational constraints between fields like length constraints. Therefore, *perfect* inference of field boundaries is essential for evaluating the performance of protocol reverse engineering tools.

$$Accuracy = \frac{\#\ Correctly\ Inferred\ Boundaries\ (TP+TN)}{\#\ All\ Byte\ Boundaries\ (TP+TN+FP+FN)} \tag{1}$$

$$Precision = \frac{\#\ Inferred\ True\ Field\ Boundaries\ (TP)}{\#\ Inferred\ Field\ Boundaries\ (TP+FP)} \tag{2}$$

$$Recall = \frac{\#\ Inferred\ True\ Field\ Boundaries\ (TP)}{\#\ True\ Field\ Boundaries\ (TP+FN)} \tag{3}$$

$$F1\text{-}score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{4}$$

$$Perfection = \frac{\#\ Perfectly\ Inferred\ Fields}{\#\ True\ Fields} \tag{5}$$

Metrics for message type inference. Different tools employ varied methods for message type inference. Netzob uses a similarity based clustering approach, while DynPRE, Netplier, and FieldHunter identify the fields that represent the message type. For a uniform comparison, we treat message type inference as a clustering problem, i.e., each group of messages of the same type corresponds to a cluster, in line with similar work [53]. We evaluate clustering performance using widely accepted metrics: *homogeneity* and *completeness* [42]. *Homogeneity* measures how well each cluster contains only messages of a single type, while *completeness* measures how well all messages of the same type are assigned to the same cluster. These metrics derive from conditional entropy analysis. Specifically, for a message set of size N, n_t denotes the number of messages of type t, n_k represents messages assigned to cluster k, and $n_{t,k}$ indicates messages of type t within cluster k. The entropy of message types $E(T)$ and the conditional entropy of types considering clustering $E(T|K)$ are defined in Formula 6. The cluster entropy $E(K)$ and the conditional entropy of clusters given the message type $E(K|T)$ are symmetrically established. From this, the *homogeneity* ($\mathcal{H}$), *completeness* ($\mathcal{C}$), and their harmonic mean *V-measure* ($\mathcal{V}$) are calculated as in Formula 7.

$$E(T) = -\sum_t \frac{n_t}{N} \log\left(\frac{n_t}{N}\right), \quad E(T|K) = -\sum_{t,k} \frac{n_{t,k}}{N} \log\left(\frac{n_{t,k}}{n_k}\right) \tag{6}$$

$$\mathcal{H} = 1 - \frac{E(T|K)}{E(T)}, \quad \mathcal{C} = 1 - \frac{E(K|T)}{E(K)}, \quad \mathcal{V} = \frac{2 * \mathcal{H} * \mathcal{C}}{\mathcal{H} + \mathcal{C}} \tag{7}$$

B. Comparison with Prior Work on Static Dataset

Given that the existing tools perform analysis on static network traces, we first compare DynPRE against these tools using static datasets (as described in Section IV-A). DynPRE starts with the same datasets but performs dynamic interaction to gather more insights and samples for analysis.

Format Inference Results. Table II shows the performance of each tool across different protocols and dataset sizes, and Figure 7 plots the average performance for each tool.

DynPRE substantially outperforms the other tools, achieving the upper bound of both one-dimensional (i.e., accuracy and F1-score) and two-dimensional (i.e., perfection) metrics across all dataset sizes. On average, DynPRE achieves an accuracy of 0.83, an F1-score of 0.70, and a perfection score of 0.50 across the three dataset sizes, compared to (0.66, 0.35, 0.15) for Netzob, (0.77, 0.33, 0.16) for Netplier, (0.73, 0.36, 0.15) for FieldHunter, (0.70, 0.36, 0.15) for BinaryInferno, and (0.75, 0.55, 0.31) for Nemesys. Notably, on the most important metric – perfection (as discussed in Section IV-A), DynPRE's score is 3.3×, 3.1×, 3.3×, 3.3×, and 1.6× of Netzob, Netplier, FieldHunter, BinaryInferno, and Nemesys, respectively, enabling more precise downstream analysis.

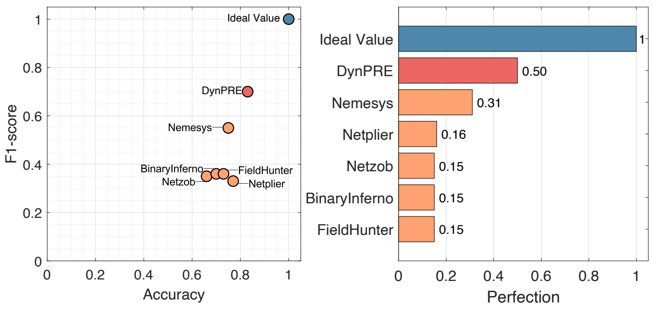


Fig. 7: Plots of average accuracy, F1-score, and perfection for each tool across different protocols and dataset sizes. Results closer to ground truth (i.e., *Ideal Value*) are better.

For the DNS protocol, Nemesys exhibits notable performance, attributed to its strategy of exploiting typical patterns of value changes in messages as a heuristic for identifying field boundaries. This approach is especially effective for DNS, which features mixed binary and text fields, thereby making value changes between fields more obvious and facilitating boundary identification. This scenario also explains the superior performance of Nemesys in analyzing NTP.

DynPRE's performance for the AMQP protocol is not consistently the best due to the lack of server response after connection establishment. This means that DynPRE cannot extract detailed semantic information from server responses and is limited to observing basic server behaviors, such as disconnections, for coarse-grained semantic insights. Nevertheless, DynPRE achieves the highest perfection on the 1000-message dataset and ranks second on the 10- and 100-message datasets, highlighting the importance of semantic information in protocol reverse engineering. Experiments on different instances of the MQTT protocol also demonstrate this. DynPRE performs better on MQTT-QoS2 than on MQTT-QoS1, attributed to more reliable message delivery in the former. This reliability provides more feedback for constructed requests, allowing DynPRE to capture richer semantics. Despite challenges with coarse-grained responses, DynPRE remains

TABLE II: Format inference results of each tool on various protocols with different dataset sizes. Bold indicates the best.

Protocol	#msg	DynPRE			Netplier			BinaryInferno			Netzob			Nemesys			FieldHunter		
		Acc.	*F1*	*Perf.*	*Acc.*	*F1*	*Perf.*	*Acc.*	*F1*	*Perf.*	*Acc.*	*F1*	*Perf.*	*Acc.*	*F1*	*Perf.*	*Acc.*	*F1*	*Perf.*
IEC61850-MMS	10	0.68	**0.59**	**0.23**	**0.69**	0.28	0.08	0.63	0.37	0.08	0.52	0.27	0.03	0.62	0.46	0.14	0.66	0.37	0.07
S7comm	10	**0.76**	**0.72**	**0.33**	0.72	0.48	0.24	0.49	0.33	0.04	0.52	0.40	0.11	0.66	0.61	0.23	0.56	0.49	0.13
Modbus	10	0.70	**0.75**	**0.54**	**0.78**	0.56	0.38	0.67	0.67	0.25	0.51	0.52	0.13	0.50	0.47	0.21	0.67	0.67	0.25
MQTT-QoS1	10	**0.98**	**0.94**	**0.82**	0.65	0.07	0.13	0.62	0.41	0.26	0.60	0.45	0.32	0.74	0.56	0.22	0.63	0.52	0.26
MQTT-QoS2	10	**0.99**	**0.96**	**0.89**	0.62	0.03	0.07	0.51	0.46	0.29	0.49	0.44	0.32	0.69	0.61	0.28	0.51	0.46	0.29
AMQP	10	0.75	0.58	0.37	0.78	0.56	0.32	0.71	0.38	0.12	0.65	0.28	0.16	**0.79**	**0.65**	**0.45**	0.66	0.21	0.00
SMB2	10	**0.85**	**0.58**	**0.24**	0.77	0.38	0.05	0.79	0.35	0.03	0.73	0.43	0.10	0.72	0.36	0.07	0.80	0.27	0.00
SMB	10	**0.82**	**0.55**	**0.35**	0.70	0.37	0.19	0.77	0.32	0.07	0.76	0.34	0.08	0.71	0.41	0.13	0.81	0.41	0.18
HTTP	10	0.89	**0.79**	**0.71**	0.68	0.11	0.12	**0.97**	0.00	0.00	0.91	0.50	0.50	0.69	0.61	0.60	0.88	0.10	0.10
NTP	10	0.71	0.44	0.21	**0.85**	0.05	0.00	0.75	**0.54**	**0.27**	0.78	0.08	0.00	0.77	0.35	0.14	0.81	0.18	0.00
DNS	10	0.62	0.46	0.27	0.79	0.36	0.08	0.76	0.26	0.00	0.72	0.42	0.13	0.79	**0.64**	**0.41**	**0.80**	0.38	0.14
BGP	10	**0.93**	**0.73**	**0.56**	0.87	0.26	0.09	0.90	0.62	0.52	0.79	0.24	0.13	0.84	0.55	0.27	0.90	0.62	0.52
TFTP	10	**0.92**	**0.77**	**0.66**	0.81	0.23	0.00	0.67	0.00	0.00	0.71	0.37	0.00	0.80	0.73	0.58	0.81	0.00	0.00
Average-10		**0.82**	**0.68**	**0.48**	0.75	0.29	0.14	0.71	0.36	0.15	0.67	0.36	0.15	0.72	0.54	0.29	0.73	0.36	0.15
IEC61850-MMS	100	**0.67**	**0.49**	**0.18**	**0.67**	0.30	0.09	0.63	0.35	0.08	0.57	0.30	0.04	0.62	0.44	0.12	**0.67**	0.41	0.08
S7comm	100	**0.80**	**0.75**	**0.41**	0.69	0.48	0.24	0.61	0.48	0.14	0.47	0.29	0.07	0.64	0.58	0.19	0.58	0.47	0.12
Modbus	100	0.71	**0.65**	**0.46**	0.72	0.50	0.10	0.58	0.48	0.13	0.56	0.46	0.12	0.58	0.49	0.31	**0.75**	0.64	0.34
MQTT-QoS1	100	**0.99**	**0.94**	**0.83**	0.66	0.09	0.16	0.62	0.40	0.25	0.62	0.48	0.33	0.78	0.56	0.25	0.62	0.40	0.25
MQTT-QoS2	100	**0.99**	**0.97**	**0.91**	0.63	0.06	0.08	0.48	0.45	0.29	0.48	0.48	0.33	0.72	0.61	0.29	0.48	0.45	0.29
AMQP	100	**0.78**	**0.65**	0.44	**0.78**	0.56	0.37	0.68	0.22	0.13	0.60	0.19	0.13	**0.78**	0.64	**0.45**	0.68	0.22	0.00
SMB2	100	**0.88**	**0.67**	**0.33**	0.79	0.38	0.03	0.79	0.34	0.02	0.67	0.40	0.07	0.70	0.29	0.05	0.80	0.27	0.00
SMB	100	**0.86**	**0.60**	**0.33**	0.73	0.46	0.19	0.76	0.45	0.13	0.67	0.38	0.10	0.70	0.46	0.10	0.79	0.51	0.22
HTTP	100	0.93	**0.87**	**0.83**	**1.00**	0.47	0.47	0.99	0.00	0.00	0.89	0.29	0.28	0.91	0.71	0.72	0.87	0.00	0.00
NTP	100	0.72	0.42	0.17	**0.85**	0.09	0.00	0.81	**0.61**	0.18	0.76	0.12	0.00	0.79	0.39	**0.32**	0.81	0.18	0.00
DNS	100	0.62	0.50	0.22	**0.86**	0.43	0.12	0.74	0.33	0.00	0.74	0.49	0.11	0.80	**0.60**	**0.32**	0.59	0.24	0.11
BGP	100	**0.97**	**0.85**	0.74	0.93	0.05	0.02	0.96	**0.85**	**0.80**	0.85	0.40	0.25	0.91	0.64	0.33	0.96	**0.85**	**0.80**
TFTP	100	**0.91**	0.79	0.71	0.76	0.25	0.00	0.43	0.00	0.00	0.62	0.21	0.01	0.90	**0.88**	**0.82**	0.78	0.00	0.00
Average-100		**0.83**	**0.70**	**0.51**	0.77	0.32	0.14	0.70	0.38	0.17	0.65	0.34	0.14	0.76	0.56	0.33	0.72	0.36	0.17
IEC61850-MMS	1000	0.68	**0.48**	**0.19**	**0.80**	0.30	0.09	0.59	0.26	0.05	0.51	0.23	0.01	0.65	**0.48**	0.10	0.61	0.31	0.05
S7comm	1000	0.78	**0.73**	**0.39**	**0.79**	0.50	0.28	0.53	0.39	0.05	0.52	0.36	0.11	0.61	0.54	0.18	0.58	0.50	0.12
Modbus	1000	0.74	**0.70**	**0.49**	**0.76**	0.51	0.09	0.58	0.50	0.14	0.60	0.50	0.20	0.58	0.50	0.28	0.69	0.61	0.23
MQTT-QoS1	1000	**0.99**	**0.94**	**0.83**	0.70	0.33	0.34	0.79	0.65	0.50	0.98	0.87	0.67	0.78	0.53	0.23	0.79	0.50	0.25
MQTT-QoS2	1000	**1.00**	**0.97**	**0.92**	0.69	0.39	0.25	0.48	0.45	0.29	0.48	0.50	0.33	0.72	0.61	0.29	0.48	0.45	0.29
AMQP	1000	**0.80**	**0.67**	**0.50**	0.77	0.52	0.38	0.68	0.23	0.13	0.64	0.20	0.13	0.79	**0.67**	0.49	0.68	0.23	0.00
SMB2	1000	**0.92**	**0.70**	**0.35**	0.79	0.38	0.03	0.79	0.33	0.00	0.62	0.38	0.08	0.72	0.31	0.06	0.82	0.36	0.05
SMB	1000	**0.87**	**0.67**	**0.37**	0.76	0.45	0.15	0.70	0.37	0.12	0.63	0.36	0.08	0.67	0.43	0.07	0.78	0.47	0.17
HTTP	1000	0.99	**0.93**	**0.93**	**1.00**	0.91	0.91	0.99	0.00	0.00	0.86	0.14	0.14	0.88	0.58	0.58	0.97	0.00	0.00
NTP	1000	0.73	**0.43**	0.18	**0.83**	0.08	0.00	0.77	0.42	0.09	0.76	0.12	0.00	0.81	**0.43**	**0.32**	0.81	0.18	0.00
DNS	1000	0.60	0.49	0.22	0.74	0.29	0.11	0.74	0.33	0.00	0.74	0.56	0.11	**0.84**	**0.66**	**0.43**	0.71	0.31	0.11
BGP	1000	**0.91**	**0.69**	**0.49**	0.90	0.12	0.05	0.89	0.59	0.45	0.78	0.00	0.00	0.83	0.54	0.28	0.89	0.59	0.45
TFTP	1000	**0.95**	0.88	0.84	0.61	0.04	0.00	0.43	0.00	0.00	0.58	0.22	0.03	0.94	**0.91**	**0.86**	0.78	0.00	0.00
Average-1000		**0.84**	**0.72**	**0.51**	0.78	0.37	0.21	0.69	0.35	0.14	0.67	0.34	0.15	0.76	0.55	0.32	0.74	0.35	0.13

effective, as evidenced by its superior performance on MQTT-QoS1 and its capability with the less responsive AMQP.

Moreover, to demonstrate DynPRE's scalability with limited input data, we utilize common statistical measures, including mean, median, and percentiles, to present the results for the 10-message dataset using box plots, as depicted in Figure 11 in Appendix C. DynPRE shows impressive perfection, surpassing Nemesys, the runner-up in mean perfection. These statistical visualizations demonstrate DynPRE's capacity to handle small inputs, highlighting its practical utility.

Message Type Inference Results. Among the selected tools, Nemesys and BinaryInferno focus on protocol format inference and do not support message type inference. Therefore, we use the remaining three tools for comparison.

Figure 8 plots the average V-measure for each tool on different protocols, and the detailed results are provided in Table XI in Appendix C. On average, DynPRE achieves a V-measure of 0.94, while Netzob, Netplier, and FieldHunter achieve 0.72, 0.73, and 0.83, respectively. Of the 13 protocol instances (including two instances for MQTT), DynPRE achieves 100% V-measure in 9 instances, while Netzob, Netplier, and FieldHunter achieve this in only 1, 1, and 3 instances, respectively. Because these methods use static analysis of network traces, they face the challenge of discerning deep semantic information to identify message types. Instead, DynPRE uses dynamic analysis to obtain semantic information, enabling more perfect field identification. It also uses the semantic-aware refinement strategy to improve format results further and infer message types, allowing it to identify true type fields in the messages more effectively.

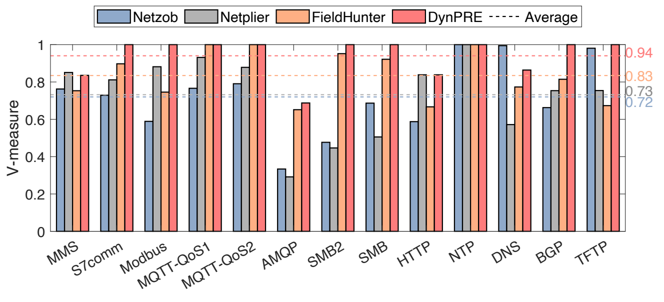


Fig. 8: Plot of average V-measure for each tool on various protocols across different dataset sizes (higher is better).

TABLE III: Average results for format and message type inference of DYNPRE compared to other tools on the datasets enhanced with S_{DYNPRE} or S_{BooFuzz}, with their original results shown in brackets. Underlined for decreases, bold for best.

	DYNPRE	**Netplier**	**BinaryInferno**	**Netzob**	**Nemesys**	**FieldHunter**
Accuracy	**0.84**	$\underline{0.77}$, 0.78 (0.78)	0.71, 0.69 (0.69)	0.70, 0.70 (0.67)	0.76, 0.77 (0.76)	$\underline{0.71}$, 0.74 (0.74)
F1-score	**0.72**	0.44, 0.44 (0.37)	$\underline{0.34}$, $\underline{0.29}$ (0.35)	0.42, 0.42 (0.34)	0.56, 0.57 (0.55)	$\underline{0.27}$, $\underline{0.34}$ (0.35)
Perfection	**0.51**	0.25, 0.28 (0.21)	0.14, $\underline{0.13}$ (0.14)	0.23, 0.23 (0.15)	0.34, 0.35 (0.32)	$\underline{0.09}$, 0.14 (0.13)
V-measure	**0.88**	0.62, 0.68 (0.62)	-	0.72, 0.72 (0.66)	-	$\underline{0.79}$, $\underline{0.79}$ (0.81)

Netzob outperforms the other tools in analyzing the DNS protocol. Upon inspection, we discovered that DNS samples typically contain many messages of the same type and exhibit high similarity – a feature intrinsic to DNS functionality. Using a message similarity based clustering approach, Netzob successfully recognizes and groups these messages into a cluster, thereby boosting its V-measure. Instead, other tools identify the type field for message type inference and use the byte as the minimal field unit. Such an approach is not optimal for DNS, which uses several bits (rather than bytes) to denote its type – replacing the type field with alternative multiple bytes leads to incorrect clustering and a lower V-measure. However, most protocols use the byte as the minimum field unit and use type fields instead of message content similarity to identify message types. Tools like DYNPRE, Netplier, and FieldHunter work on this principle and achieve an overall better V-measure than Netzob.

C. Comparison with Prior Work on Enhanced Dataset

To further enrich our comparative analysis, we provide the five compared tools with datasets enhanced by dynamic interaction. Specifically, we enrich the initial 1000-message dataset S_{initial} by providing two types of additional datasets: (i) the additional message samples derived by DYNPRE, denoted as S_{DYNPRE}, and (ii) the additional message samples derived by a protocol fuzzer – BooFuzz [20], denoted as S_{BooFuzz}. For S_{BooFuzz} in detail, we initialize BooFuzz with the 1000-message dataset S_{initial}, let it interact with the protocol server in a setup identical to that of DYNPRE, and capture the traffic during this interactive phase to obtain additional samples. For BooFuzz initialization, we use the message order recorded in the input network traces to construct the session graph for BooFuzz, and define the message structure by treating each recorded message as a byte sequence [21], since we are format agnostic when reverse engineering the protocol. In addition, since BooFuzz performs mutations randomly, the messages it generates are not necessarily semantically valid. To avoid confusing the compared tools, we filter the traffic and retain only the server responses for the extension, as the responses are legitimate messages. This treatment is consistent with DYNPRE, which adds only the server responses to the dataset for statistical analysis. Meanwhile, for each protocol, we restrict the size of S_{BooFuzz} to be the same as S_{DYNPRE} to ensure a fair comparison.

Results. Table III shows the average performance of the compared tools when provided two enhanced datasets, compared to their original performance and the performance of DYNPRE. Detailed results are given in Tables IX and X in Appendix C. To ensure that the results after enhancement are comparable to the original ones, we calculate only the average performance of the compared tools on the S_{initial} part, even when they are provided with additional datasets. From the results, Netplier and Netzob showed slightly more improvement than the other three tools on the enhanced datasets. This is because they employ alignment-based approaches, and providing more samples for alignment often leads to better results. However, certain tools perform worse on the enhanced datasets, such as BinaryInferno's F1-score and FieldHunter's V-measure. This can be attributed to the inherent uncertainty of the underlying statistical analysis. Overall, the performance of the static tools on the S_{DYNPRE} enhanced dataset and the S_{BooFuzz} enhanced dataset show a slight difference, and their improvements over the original results both remain limited, with DYNPRE consistently outperforming them. The main reason lies in their different strategies for exploring interactive traffic. DYNPRE explores by correlating the modification operations with the server feedback, thus ensuring precise inference of byte semantics. In contrast, the other tools rely exclusively on statistical analysis that explores byte-value distribution and similarity. This is less accurate because bytes with identical or different values do not inherently imply semantic similarity or difference. This gap between the semantics of the field and its statistical characteristics limits the performance of the static tools and cannot be overcome by providing additional samples.

D. Module Evaluation

In our proposed approach, the accurate detection of session-specific identifiers is critical, as the derived set of message rewrite rules forms the basis of the subsequent dynamic inference. Meanwhile, DYNPRE's refinement strategy is designed to improve the overall format accuracy. In this section, we evaluate the effectiveness of these two modules.

TABLE IV: Number of $\underline{\text{(sources / references)}}$ of derived message rewrite rules for protocols with session-specific identifiers

Protocol	**10 messages**	**100 messages**	**1000 messages**
HTTP	1 / 3	3 / 42	7 / 477
SMB	2 / 9	3 / 149	9 / 410
SMB2	3 / 8	12 / 163	87 / 1093
BGP	1 / 1	1 / 1	4 / 8
TFTP	3 / 3	15 / 39	105 / 339

Session-Specific Identifier Detection Accuracy. For an input network trace $\varGamma_o$, we use a dynamic approach to measure the correctness of the derived message rewrite rules $\varUpsilon$ by checking whether we can successfully replay $\varGamma_o$. Specifically, we replay $\varGamma_o$ using the on-the-fly message rewriting configured with $\varUpsilon$ (as shown in Section III-B) and obtain the replayed $\varGamma_r$. We then check whether $\varGamma_r$ is identical to $\varGamma_o$, excluding byte regions associated with the sources or references documented in $\varUpsilon$. If they are identical, the derived $\varUpsilon$ is correct. Otherwise,

TABLE V: The inferred message format of the protocol used in different devices and the behaviors triggered by the messages generated from this format (with behavior-determining bytes in bold, * means the protocol employs the session-specific identifiers).

Device	Behaviors of Input Messages	Message Format	# Triggered Behaviors
Yeelight LED Screen Light Bar Pro	Turn On	V(18) **V(9)** C(13) **V(2)** V(3) V(6) V(7) C(2)	Turn On, Brighten, Turn Off, Dim
	Brighten	V(18) **V(10)** C(12) **V(2)** V(2) C(6) V(7) C(2)	
Philips Hue Bridge	Create Group	**V(7)** C(28) V(15) **V(9)** V(57) C(1) V(11) V(7) V(1) V(3) **V(36)**	Set Name, Create Group, Output Name, Delete Group
	Set Name	**V(57)** V(57) C(1) V(7) V(1) V(5) V(5) V(1) V(1) V(2) **V(21)**	
Broadlink Smart Plug	Turn Off	V(32) V(4) C(2) C(2) V(2) C(10) C(1) V(1) V(2) V(4) V(2) **V(26)**	Turn On, Turn Off
Xiaomi Mijia Smart Camera*	Turn On	C(12) V(4) V(9) V(7) V(21) C(9) V(4) V(1) V(2) V(3) C(5) **V(2)** C(4)	Turn On, Turn Off
Tplink Router	Add Forbidden Domain	C(2) C(10) C(12) C(13) C(10) V(15) C(11) C(6) C(3) **V(13)** C(14) **V(3)** C(2)	Add Forbidden Domain, Clear Forbidden Domains, Output Forbidden Domains

we need to investigate whether the discrepancy between $\mathit{\Gamma}_r$ and $\mathit{\Gamma}_o$ stems from random nonces, whose values also vary between different sessions. We manually check the semantics of the relevant regions by consulting the protocol specification. If the discrepancy is solely due to random nonces, we can conclude that the derived $\mathit{\Upsilon}$ is accurate. We repeat the above steps three times to confirm the correctness of the derived constraint for each rewrite rule, thus eliminating chance factors.

Following the procedure above, we confirm that DynPRE accurately identifies all potential session-specific identifiers in the network traces within the dataset. Specifically, among the subject protocols, DynPRE detects five protocols that contain session-specific identifiers, i.e., HTTP, SMB, SMB2, BGP, and TFTP. Table IV summarizes the message rewrite rules derived for their identifiers. By manually inspecting the results, we find the derived message rewrite rules consistent with the respective protocol specifications. In the HTTP, TFTP, and BGP protocols, all detected session-specific identifiers belong to the same type of field, i.e., the *Cookie*, *DestinationPort*, and *MyAS* fields, respectively. Conversely, the scenarios in the SMB and SMB2 protocols are more complicated. The detected session-specific identifiers in these protocols include various fields, including the *SessionID*, *TreeID*, and *ServerGuid* fields. Furthermore, their lifetime within the session varies, demonstrating the ability of our method to handle complex situations.

Effectiveness of the Refinement Strategy. This strategy aims to improve the overall format results and to infer message field types, with the latter being evaluated in Section IV-B. We evaluate the format inference results by comparing DynPRE and DynPRE$^-$ (a variant without the refinement strategy). Table VII in Appendix C shows the format inference results of DynPRE$^-$. With the help of the refinement strategy – which synthesizes the results from dynamic analysis through interactions and static analysis on increased samples – DynPRE achieves an average improvement of 0.05 in accuracy, 0.10 in F1-score, and 0.13 in perfection compared to DynPRE$^-$, demonstrating the effectiveness of the refinement strategy. Overall, DynPRE$^-$ achieves an accuracy of 0.78, an F1-score of 0.60, and a perfection of 0.37, which still significantly outperforms the state-of-the-art tools, demonstrating the effectiveness of our proposed dynamic inference approach.

E. Proprietary Protocol Analysis

In this section, we apply DynPRE to proprietary protocols used in IoT devices. However, evaluating protocol reverse engineering tools on unknown protocols is challenging due to their proprietary nature, which makes it difficult to obtain specifications and establish ground truth (as performed in Sections IV-B and IV-C). Considering the protocol's application, we design a three-step evaluation process: (1) *Input message construction.* First, we activate various behaviors on the target IoT device via the official APP. Meanwhile, we capture the network traffic during activation, label the messages corresponding to each behavior, and use them as input for DynPRE. (2) *Protocol reverse engineering.* We then use DynPRE to infer the message format associated with each behavior through dynamic interaction with the device. (3) *Application of the inferred formats.* To evaluate the results, we use the inferred formats to generate messages and send them to the device. We then observe whether these newly generated messages can (i) trigger the same behavior as the original input messages and (ii) trigger additional behaviors beyond those triggered by the input messages.

Following the described procedure, we select five widely used IoT devices with diverse functionalities for evaluation, including a light, two controllers (a bridge and a smart plug), an IP camera, and a router. Table V provides their detailed information and the corresponding analysis results. Some devices use encrypted communication, and we decrypt them using the device's private key when adapting DynPRE to focus our study on protocol reverse engineering. The second to fourth columns summarize the information for each evaluation step. As indicated in the second column, we activate different behaviors according to the device type, such as turning on or brightening the light, creating a new control group for the bridge, and adding a forbidden domain for the router. The third column presents the inferred message format for each activated behavior. We show only the type and length for each field as `FieldType(FieldLength)`, where 'C' denotes constant fields and 'V' variable fields. Notably, DynPRE detects a session-specific identifier in the protocol used by the Xiaomi camera, demonstrating its capacity to handle intricate scenarios in real-world protocols.

For format evaluation, we generate new messages based on the inferred formats and send them to the device. Following the strategy used in protocol fuzzers [14], we employ a structure-aware message generation approach that integrates two levels of operations: (i) random field content modification – specifically, we leave constant field values unchanged, while for variable fields we consider existing values from network traces as well as random content; and (ii) random field deletion. The fourth column of Table V shows the behaviors triggered by the newly generated messages. We observe that these messages not only initiate the original behaviors triggered by the input messages but can also trigger additional behaviors beyond the originals. These include turning off and dimming the light, deleting the control group in the bridge, removing the forbidden domain in the router, and so on. In summary, based on the original 7 types of behaviors initiated on the selected devices, the newly generated messages can trigger 15 different behaviors, demonstrating the validation of the formats inferred by DynPRE. For further study, we analyze the bytes determining the behaviors triggered by the new messages, as shown in bold in the third column. We find that these bytes are usually command fields or fields carrying command arguments, and that these bytes are all identified as variable fields by DynPRE. Below, we provide two case studies.

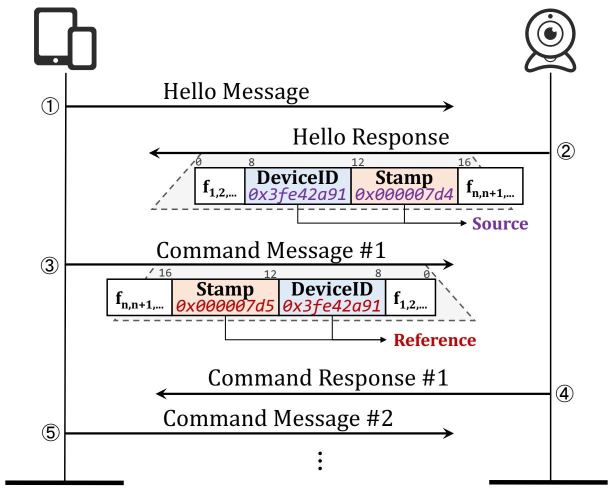


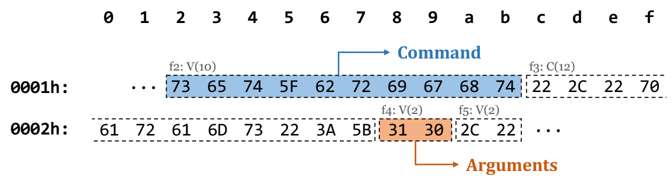
Fig. 9: The usage of the session-specific identifiers in the Xiaomi smart camera.

Case Studies. Figure 9 presents the startup session of the Xiaomi smart camera. The startup session is initiated by a `Hello Message` sent from the client to the device (①). In response, the device sends a `Hello Response` containing two session-specific identifiers, the *DeviceID* and *Stamp* fields (②). The *Stamp* field is an increasing counter designed to prevent replay attacks and to trace the session. Specifically, if the device's response message assigns a value x to this field, the subsequent request message should present a value of $x + 1$ in this field. If not, the device will reject the request. This rule also applies to subsequent request/response pairs throughout the session, including ④ and ⑤. For this identifier, DynPRE generates a rule presented in the first row of Table VI. The rule states that the source is bytes $M[12..15]$ in message ②, with references including bytes $M[12..15]$ in message ③ under constraint $y = x + 1$. The third rule is similarly derived. The *DeviceID* field serves as a unique identifier for the device. The device will reject any request message containing a mismatched *DeviceID*. This field is not always categorized as a session-specific identifier. Consider the following two scenarios: (i) If the device on which the input network traces are captured and the device from which DynPRE learns are the same, then the *DeviceID* field is not deemed a session-specific identifier because the device accepts the *DeviceID* value recorded in the input network traces. (ii) If the device on which the input network traces are captured and the device from which DynPRE learns are different, the *DeviceID* field is recognized as a session-specific identifier. In this scenario, DynPRE must rewrite the *DeviceID* field in the probe request messages to the appropriate value provided in response ②. DynPRE successfully detects these identifiers in our experiment and derives the corresponding message rewrite rules, as shown in Table VI. Based on this, DynPRE establishes on-the-fly message rewriting to dynamically adjust the values of these fields in the probe requests, which prevents early rejection of the probe requests and enables more profound semantic feedback for accurate protocol analysis.

TABLE VI: The derived message rewrite rules for the Xiaomi camera startup session.

No.	Source	References	Constraint
1	②: [12..15]	③: [12..15]	$y = x + 1$
2	②: [8..11]	③: [8..11], ⑤: [8..11]	$y = x$
3	④: [12..15]	⑤: [12..15]	$y = x + 1$

Figure 10 shows a snippet of a Yeelight light's brightening message in hex format. We highlight its behavior-determining bytes, which correspond to the second and fourth inferred fields as shown in Table V. The precise field identification of DynPRE allows us to observe the correlation between the change in the field's values and the light's behaviors, assisting us in determining the fields' semantics. The second field, in the blue background, is the command field. Different values of this field can instruct the light to perform different behaviors, including turning on/off and brightening/dimming the light shown in our experiment. The fourth field, in the orange background, carries the command arguments. For the command to set the brightness, this field carries the target brightness that a user wants to set. In our experiment, manipulating this field allowed us to change the brightness of the light directly. Besides, successfully detecting this field allows us to fuzz the light effectively by assigning abnormal values to the field for attack vectors such as integer overflow.



```
        0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
0001h:  ...   73 65 74 5F 62 72 69 67 68 74 22 2C 22 70
0002h:  61 72 61 6D 73 22 3A 5B 31 30 2C 22 ...
```

Fig. 10: A snippet of a Yeelight light's brightening message.

V. Discussion

A. Performance Overhead

Since DynPRE performs dynamic analysis by interacting with the server, it may not be faster than tools that only perform static analysis on network traces. However, considering that the protocol reverse engineering is generally a one-time effort and affects many downstream applications, a slightly longer execution time for better results is a worthwhile trade-off, as errors in the results can propagate and accumulate through the application chain. In our experiments, for the largest dataset size of 1000 messages, DYNPRE takes an average of 259 minutes to reverse engineer a protocol, with an average of 50344 connection attempts and 3976443 exchanged messages with the protocol server. This overhead is acceptable because DYNPRE provides a more accurate understanding of the target protocol. The detailed overhead statistics are given in Table XII in Appendix C.

Since DYNPRE requires interaction with the server, its execution time is affected by the response time of the protocol server. Low server throughput or denied requests after frequent connections can hinder the connection frequency of DYNPRE, thus affecting the analysis efficiency. We can employ static tools, e.g., those compared tools shown in Section IV-A, for initial analysis and screening to migrate this. On the one hand, the static analysis helps to identify obvious features, allowing us to skip analyzing unnecessary bytes. For instance, in the SMB2 protocol, the *ProtocolId* field ($M[4..7]$ in Figure 1) in the header is consistently set to "`\xFESMB`" to denote this protocol, so we can avoid analyzing these bytes in each message. On the other hand, by analyzing the initial format derived by static tools, we can determine byte regions of interest, i.e., regions likely to contain essential fields like command fields, and use DYNPRE to dynamically re-analyze these specific byte regions with high precision.

B. *Impact of Active Probing*

DYNPRE analyzes protocols using active probing. However, as the protocol server is stateful, the probing process may change its internal state, affecting the subsequent exploration process. DYNPRE addresses this by establishing a new session with the server for each probe, as most network protocols maintain states separately for different sessions. Yet, problems arise with protocols that have global states shared across sessions. For instance, in IoT protocols with global device states, previous requests like power on/off can affect the validity of subsequent requests, potentially reducing feedback quality. However, DYNPRE can still get feedback on message parsing, as servers typically parse before state checking. Besides, it is beneficial to control the server where possible; analyzers can reset its state via restarts or APP control, mitigating the effects of previous exploration and providing more accurate results.

VI. RELATED WORK

A. *Protocol Reverse Engineering*

Program Analysis Based Approach. These techniques utilize dynamic program analysis methods such as taint analysis to trace the program execution of given well-formed messages [23], [8], [11] or perform static program analysis on the source code [44], [43]. Despite their high accuracy, they require access to the source code or binary, which may not be feasible in practice. Instead, DYNPRE requires only network access to the server and can be conducted in a black-box setting. Meanwhile, unlike the dynamic program analysis based on well-formed messages, DYNPRE employs an intelligent approach to modify well-formed messages to reveal diverse server behaviors for semantic information extraction.

Network Trace Based Approach. These techniques analyze static network traces to mine features such as message bytes [22], sequences [53], [6], and common field semantics [3], [9]. Some techniques, such as Netzob [6] and Netplier [53], employ an alignment-based approach that classifies messages into distinct clusters and summarizes cluster-specific structures based on alignment [12]. These techniques rely on high-quality network traces containing diverse messages to recognize differences between clusters and data commonalities between messages within each cluster. Obtaining such traces is often difficult in practice due to the uneven distribution of messages in real conversations. DYNPRE uses alignment to analyze responses and addresses this limitation by actively generating many slightly different requests to obtain multiple response variants that are generally similar but with different details, providing natural clustering and more granular insight into the format. DYNPRE also incorporates a dynamic approach to obtaining semantics from the server for analysis.

Some work also incorporates semantic information into statistical analysis to infer protocol formats. Discoverer [12] uses token type patterns provided by the analyst to tokenize bytes using simple semantics, e.g., textual or binary, and then clusters messages based on the tokenized results. FieldHunter [3] uses the general semantics and features of specific common fields, e.g., message length or host ID, and identifies fields by establishing a statistical correlation between these types and byte values. BinaryInferno [9] uses an ensemble of detectors to search for semantically meaningful explanations using serialization patterns. The semantics used by these techniques are based on prior knowledge and are still inferred through the statistical analysis of the static messages, which may face challenges in capturing semantics partially reflected in the supplied value or in dealing with message content with noise. Instead, DYNPRE uses dynamic inference to obtain more precise semantics from interactions and identify fields by automatically recognizing semantic changes.

B. *Protocol Fuzzing*

Protocol fuzzing is a widely adopted dynamic testing technique that continuously generates and sends messages to the server while monitoring for anomalies [56], [36], [25], [57], [10]. Due to the highly complex protocol structure and interaction logic, effective fuzzing requires knowledge of the protocol model, including the acceptable message formats and ordering [20], [14], [24], [46], [26]. However, constructing protocol models requires significant manual effort and expertise. Protocol reverse engineering can automate this process, especially for protocols without specifications or source code. WEIZZ [15] identifies field boundaries by analyzing the dependencies between input bytes and comparison instructions, which requires the source code or binary. Pulsar [17] performs a simple combination of protocol reverse engineering and fuzzing. It statically analyzes the input network traces to infer the protocol model and uses it to initialize fuzzing. Pulsar does not recognize and utilize semantic information during dynamic fuzzing to improve the protocol model, so it faces the same challenges as existing approaches. Instead, DYNPRE utilizes the interactive capability of the server to obtain more semantic information and samples for analysis and can provide a more accurate protocol format and state machine than existing approaches, thus improving the effectiveness of downstream applications like fuzzing.

VII. Conclusion

In this paper, we present DynPRE, a network trace based protocol reverse engineering tool that is fully automatic and requires no prior knowledge of the protocol under analysis. Rather than relying exclusively on the statistical analysis of the input network traces, DynPRE integrates a dynamic approach that intelligently interacts with the protocol server to extract insightful information, allowing for a more accurate understanding of the underlying protocol structure and semantics. Our experiments show that DynPRE substantially outperforms state-of-the-art tools in field identification and message type inference. Furthermore, the successful application of DynPRE to real-world proprietary protocols showcases its versatility and potential to address challenges in security applications.

Acknowledgement

We thank the anonymous reviewers for their constructive feedback and suggestions. This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000) and NSFC Program (No. 62022046, 92167101, U1911401, 62021002).

References

[1] 8051Enthusiast. (2023) delsum: A reverse engineer's checksum toolbox. https://github.com/8051Enthusiast/delsum.

[2] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley, "Your exploit is mine: Automatic shellcode transplant for remote exploits," *2017 IEEE Symposium on Security and Privacy (SP).*

[3] I. Bermudez, A. Tongaonkar, M. Iliofotou, M. Mellia, and M. M. Munafò, "Automatic protocol field inference for deeper protocol understanding," *2015 IFIP Networking Conference*, 2015.

[4] binaryinferno. (2023) Implementation of binaryinferno. https://github.com/binaryinferno/binaryinferno.

[5] G. Bossert, "Exploiting semantic for the automatic reverse engineering of communication protocols." 2014.

[6] G. Bossert, F. Guihéry, and G. Hiet, "Towards automated protocol reverse engineering using semantic information," *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014.

[7] J. Caballero, P. Poosankam, C. Kreibich, and D. X. Song, "Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering," in *ACM CCS*, 2009.

[8] J. Caballero, H. Yin, Z. Liang, and D. X. Song, "Polyglot: automatic extraction of protocol message format using dynamic binary analysis," in *ACM CCS 2007.*

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

[10] Y. Chen, F. Ma, Y. Zhou, Y. Jiang, T. Chen, and J. Sun, "Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model," in *IEEE Symposium on Security and Privacy (SP)*, 2023.

[11] P. M. Comparetti, G. Wondracek, C. Krügel, and E. Kirda, "Prospex: Protocol specification extraction," *2009 30th IEEE Symposium on Security and Privacy.*

[12] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic protocol reverse engineering from network traces," in *USENIX Security Symposium*, 2007.

[13] eclipse. (2023) Eclipse mosquitto - an open source mqtt broker. https://github.com/eclipse/mosquitto.

[14] M. Eddington. (2023) Peach fuzzing platform. https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce.

[15] A. Fioraldi, D. C. D'Elia, and E. Coppa, "Weizz: automatic grey-box fuzzing for structured binary formats," *29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.

[16] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager, "Combining model learning and model checking to analyze tcp implementations," in *Computer Aided Verification: 28th International Conference, CAV 2016.*

[17] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *Security and Privacy in Communication Networks*, 2015.

[18] M. E. Hoque, O. Chowdhury, S. Y. Chau, C. Nita-Rotaru, and N. Li, "Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs," *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).*

[19] M. Isberner, F. Howar, and B. Steffen, "The open-source learnlib - a framework for active automata learning," in *International Conference on Computer Aided Verification*, 2015.

[20] jtpereyda. (2023) BooFuzz: Network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz.

[21] jtpereyda. (2023) BooFuzz Quickstart. https://boofuzz.readthedocs.io/en/stable/user/quickstart.html?highlight=session.connect#quickstart.

[22] S. Kleber, H. Kopp, and F. Kargl, "Nemesys: Network message syntax reverse engineering by analysis of the intrinsic structure of individual messages," in *WOOT @ USENIX Security Symposium*, 2018.

[23] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution," in *NDSS 2008.*

[24] Z. Luo, J. Yu, F. Zuo, J. Liu, Y. Jiang, T. Chen, A. Roychoudhury, and J. Sun, "Bleem: Packet sequence oriented fuzzing for protocol implementations," *2023 32nd USENIX Security Symposium.*

[25] Z. Luo, F. Zuo, Y. Jiang, J. Gao, X. Jiao, and J. Sun, "Polar: Function code aware fuzz testing of ICS protocol," *ACM Trans. Embed. Comput. Syst.*, 2019.

[26] Z. Luo, F. Zuo, Y. Shen, X. Jiao, W. Chang, and Y. Jiang, "ICS protocol fuzzing: Coverage guided packet crack and generation," *ACM/IEEE Design Automation Conference (DAC)*, 2020.

[27] R. Marcovich, O. Grumberg, and G. Nakibly, "Pise: Protocol inference using symbolic execution and automata learning," *Proceedings 2023 Workshop on Binary Analysis Research.*

[28] K. L. McMillan and L. D. Zuck, "Formal specification and testing of QUIC," *ACM Special Interest Group on Data Communication*, 2019.

[29] Microsoft, "Server message block (smb) protocol versions 2," 2023, https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-smb2/5606ad47-5ee0-437a-817e-70c366052962.

[30] msoulier. (2023) Pure python tftp library. https://github.com/msoulier/tftpy.

[31] mz automation. (2023) Official repository for libiec61850, the open-source library for the iec 61850 protocols. https://github.com/mz-automation/libiec61850.git.

[32] NetPlier. (2023) Netplier: Probabilistic network protocol reverse engineering from message traces. https://github.com/netplier-tool/NetPlier.

[33] netzob. (2023) Netzob: Protocol reverse engineering, modeling and fuzzing. https://github.com/netzob/netzob.

[34] osrg. (2023) Bgp implemented in the go programming language. https://github.com/osrg/gobgp.

[35] M. L. Pacheco, M. von Hippel, B. Weintraub, D. Goldwasser, and C. Nita-Rotaru, "Automated attack synthesis by extracting finite state machines from protocol specification documents," in *2022 IEEE Symposium on Security and Privacy (SP).*

[36] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNET: A greybox fuzzer for network protocols," *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST).*

[37] RabbitMQ. (2023) Open source rabbitmq: core server and tier 1 (built-in) plugins. https://github.com/rabbitmq/rabbitmq-server.

[38] RFC 6101, "The secure sockets layer (ssl) protocol version 3.0," Website, https://datatracker.ietf.org/doc/html/rfc6101.

[39] RFC 6265, "Http state management mechanism," Website, https://datatracker.ietf.org/doc/html/rfc6265.

[40] RFC 793, "Transmission control protocol," Website, https://www.ietf.org/rfc/rfc793.txt.

[41] RFC 959, "File transfer protocol (ftp)," Website, https://datatracker.ietf.org/doc/html/rfc959.

[42] A. Rosenberg and J. Hirschberg, “V-measure: A conditional entropy-based external cluster evaluation measure,” in *Conference on Empirical Methods in Natural Language Processing*, 2007.

[43] Q. Shi, J. Shao, Y. Ye, M. Zheng, and X. Zhang, “Lifting network protocol implementation to precise format specification with security applications,” *ACM SIGSAC CCS*, 2023.

[44] Q. Shi, X. Xu, and X. Zhang, “Extracting protocol format as state machine via controlled static loop analysis,” *Usenix Security*, 2023.

[45] stephane. (2023) A modbus library for linux, mac os, freebsd and windows. https://github.com/stephane/libmodbus.git.

[46] Synopsis. (2023) Defensics fuzz testing. https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html.

[47] T. S. Team. (2023) Samba is the standard windows interoperability suite of programs for linux and unix. https://gitlab.com/samba-team/samba.

[48] thekelleys. (2023) Implementation of the dns protocol. https://thekelleys.org.uk/dnsmasq/doc.html.

[49] tplink. (2023) Tp-link ax5400 pro web management page. https://www.tp-link.com/us/support/download/archer-ax5400-pro/#Firmware.

[50] TShark. (2023) Tshark - dump and analyze network traffic. https://www.wireshark.org/docs/man-pages/tshark.html.

[51] vs uulm. (2023) Network message syntax analysys. https://github.com/vs-uulm/nemesys.

[52] vs-uulm. (2023) Re-implementation of the protocol reverse engineering approach fieldhunter. https://github.com/vs-uulm/fieldhunter.

[53] Y. Ye, Z. Zhang, F. Wang, X. Zhang, and D. Xu, “Netplier: Probabilistic network protocol reverse engineering from message traces,” *Proceedings 2021 Network and Distributed System Security Symposium*.

[54] J. Yen, T. L’evai, Q. Ye, X. Ren, R. Govindan, and B. Raghavan, “Semi-automated protocol disambiguation and code generation,” *Proceedings of the 2021 ACM SIGCOMM Conference*.

[55] yerseg. (2023) Implementation of the s7comm protocol. https://github.com/yerseg/s7comm_investigation.

[56] F. Zuo, Z. Luo, J. Yu, T. Chen, Z. Xu, A. Cui, and Y. Jiang, “Vulnerability detection of ICS protocols via cross-state fuzzing,” *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 2022.

[57] F. Zuo, Z. Luo, J. Yu, Z. Liu, and Y. Jiang, “PAVFuzz: State-sensitive fuzz testing of protocols in autonomous vehicles,” *ACM/IEEE Design Automation Conference (DAC)*, 2021.

Appendix

A. DynPRE*’s Scalability for Checksum Mechanism used in Protocols*

DynPRE constructs probe messages by modifying the original messages in the input network traces. However, the checksum mechanism used in some protocols can render this strategy ineffective because the modified messages may fail checksum verification, making it difficult to induce in-depth semantics from the server. This mechanism is widely used in transport-layer protocols like TCP but is less common in application-layer protocols, which are the primary targets of protocol reverse engineering. To improve the scalability of DynPRE with potential targets that use checksum mechanisms, we integrate DynPRE with an existing tool, *delsum* [1], and utilize our on-the-fly message rewriting mechanism. Specifically, DynPRE first employs *delsum* to reverse engineer the checksum algorithm. Then, DynPRE automatically generates the corresponding message rewrite rules to allow the checksum to be recalculated before the message is sent. Based on this, DynPRE can perform normal analysis on the target protocol. We evaluate the effectiveness of this approach on the ICMP protocol, which uses a checksum mechanism. The results show that DynPRE can successfully reverse engineer the checksum algorithm and correctly establish message rewriting for the checksum field. We also evaluate the format inference performance. The results show that DynPRE achieves the best perfection of 0.66, compared to the best result of 0.41 for the other tools. The superiority of DynPRE on ICMP is consistent with the results shown in Section IV-B, demonstrating the scalability of DynPRE with protocols that use checksum mechanisms.

B. Experimental Dataset

Table VIII shows detailed information on the selected open-source protocol projects and constructed dataset. This includes information on the protocol server under learning, the client utilities used for building the dataset, and the number of message types within the constructed dataset. The encryption or authentication mechanisms of the selected utilities are disabled since this is beyond the scope of our work. To facilitate a fair comparison, the selected server and client utilities (cf. the third and fourth columns) all come from off-the-shelf utilities that are either included in the project or operating system or publicly available services on the Internet (e.g., the public NTP time server).

To enhance the diversity of message types, we try to use as many existing clients as possible, and to manipulate the call parameters of each client to trigger various interaction scenarios. The fifth column shows the number of message types in the dataset, which is obtained based on the message parsing results of *tshark* [50] and the official protocol documents.

C. Detailed Results

This section presents the detailed performance results of DynPRE, DynPRE$^-$, and the state-of-the-art tools.

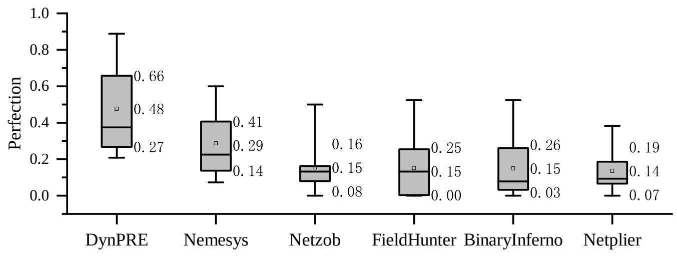


Fig. 11: The box plot of mean, median, and percentiles of the perfection on the 10-message dataset for each tool.

TABLE VII: Format inference results of DynPRE$^-$ on various protocols with different dataset sizes.

Protocol	10 messages			100 messages			1000 messages		
	Acc.	F1	Perf.	Acc.	F1	Perf.	Acc.	F1	Perf.
IEC61850-MMS	0.63	0.34	0.14	0.66	0.37	0.16	0.65	0.34	0.14
S7comm	0.69	0.54	0.24	0.70	0.54	0.24	0.69	0.53	0.23
Modbus	0.60	0.64	0.33	0.64	0.55	0.30	0.68	0.61	0.32
MQTT-QoS1	0.78	0.76	0.48	0.82	0.78	0.50	0.82	0.77	0.50
MQTT-QoS2	0.79	0.78	0.55	0.82	0.80	0.58	0.83	0.80	0.58
AMQP	0.75	0.53	0.34	0.78	0.64	0.42	0.79	0.67	0.45
SMB2	0.83	0.44	0.14	0.86	0.56	0.24	0.86	0.63	0.32
SMB	0.77	0.54	0.32	0.79	0.54	0.30	0.84	0.64	0.33
HTTP	0.86	0.75	0.68	0.91	0.81	0.77	0.97	0.94	0.92
NTP	0.67	0.41	0.21	0.69	0.40	0.17	0.68	0.41	0.18
DNS	0.72	0.46	0.23	0.71	0.46	0.17	0.71	0.46	0.17
BGP	0.92	0.54	0.44	0.93	0.62	0.48	0.91	0.66	0.45
TFTP	0.82	0.64	0.38	0.84	0.62	0.49	0.90	0.72	0.64
Average	0.76	0.57	0.35	0.78	0.59	0.37	0.79	0.63	0.40

TABLE VIII: Information on the selected open-source protocol project and constructed dataset for evaluation. We use off-the-shelf servers and clients for dataset construction and evaluation.

Protocol	Project	Server Under Learning	Clients for Dataset Construction	Message Types
IEC61850-MMS	libiec61850 [31]	server_example_basic_io/server_example_basic_io	iec61850_client_example_async/client_example_async iec61850_client_example_control/client_example_control iec61850_client_example_array/client_example_array iec61850_client_example_log/client_example_log iec61850_client_example1/client_example1 iec61850_client_example2/client_example2 iec61850_client_example_no_thread/client_example_no_thread iec61850_client_example_reporting/client_example_reporting	10
S7comm	s7comm_investigation [55]	server	client	4
Modbus	libmodbus [45]	tests/bandwidth-server-many-up	tests/random-test-client	14
MQTT-QoS1	Mosquitto [13]	src/mosquitto	client/mosquitto_sub client/mosquitto_pub	4
MQTT-QoS2	Mosquitto [13]	src/mosquitto	client/mosquitto_sub client/mosquitto_pub	6
AMQP	RabbitMQ [37]	rabbitmq/sbin/rabbitmq-server	pika (a python library)	5
SMB2	Samba [47]	bin/smbd	bin/smbclient	28
SMB	Samba [47]	bin/smbd	bin/smbclient	32
HTTP	Router Web [49]	usr/bin/httpd	Chrome (web browser)	5
NTP	-	pool.ntp.org (public NTP service)	ntptrace (a Linux utility in ntp package)	2
DNS	Dnsmasq [48]	src/dnsmasq	nslookup (a Linux utility in Dnsutils package)	4
BGP	gobgp [34]	gobgpd	gobgpd	3
TFTP	tftpy [30]	bin/tftpy_server.py	atftp (a Linux utility)	4

TABLE IX: Format inference results of compared tools on $\underline{(S_{\text{BooFuzz}},\ S_{\text{DYNPRE}})}$ enhanced datasets ("-" indicates timeout).

Protocol	Netplier			BinaryInferno			Netzob			Nemesys			FieldHunter		
	Accuracy	*F1-score*	*Perfection*	*Accuracy*	*F1-score*	*Perfection*	*Accuracy*	*F1-score*	*Perfection*	*Accuracy*	*F1-score*	*Perfection*	*Accuracy*	*F1-score*	*Perfection*
IEC61850-MMS	0.70, 0.68	0.45, 0.37	0.23, 0.07	0.61, 0.55	0.29, 0.19	0.10, 0.00	0.58, 0.58	0.22, 0.22	0.05, 0.04	0.70, 0.59	0.56, 0.38	0.22, 0.07	0.60, 0.57	0.30, 0.20	0.05, 0.00
S7comm	0.76, 0.76	0.55, 0.53	0.35, 0.21	0.58, 0.58	0.48, 0.49	0.13, 0.14	0.61, 0.63	0.53, 0.54	0.23, 0.24	0.62, 0.70	0.56, 0.65	0.17, 0.29	0.58, 0.58	0.50, 0.50	0.12, 0.12
Modbus	0.81, 0.81	0.65, 0.65	0.31, 0.33	0.66, 0.76	0.50, 0.71	0.22, 0.47	0.42, 0.48	0.24, 0.40	0.13, 0.13	0.58, 0.58	0.50, 0.50	0.28, 0.28	0.54, 0.55	0.33, 0.36	0.09, 0.11
MQTT-QoS1	0.75, 0.75	0.37, 0.45	0.34, 0.34	0.74, 0.79	0.37, 0.50	0.26, 0.25	0.98, 0.98	0.87, 0.87	0.67, 0.67	0.74, 0.78	0.45, 0.54	0.24, 0.23	0.78, 0.63	0.49, 0.18	0.24, 0.09
MQTT-QoS2	0.80, 0.80	0.61, 0.61	0.50, 0.50	0.47, 0.47	0.00, 0.00	0.00, 0.00	0.98, 0.98	0.87, 0.87	0.83, 0.83	0.72, 0.72	0.61, 0.61	0.29, 0.29	0.72, 0.57	0.57, 0.25	0.28, 0.13
AMQP	0.83, 0.82	0.56, 0.53	0.35, 0.26	0.68, 0.72	0.23, 0.40	0.13, 0.13	0.64, 0.64	0.47, 0.33	0.26, 0.13	0.83, 0.79	0.72, 0.67	0.55, 0.49	0.68, 0.68	0.23, 0.23	0.00, 0.00
SMB2	0.68, 0.77	0.38, 0.42	0.08, 0.09	0.79, 0.79	0.33, 0.37	0.00, 0.05	-, -	-, -	-, -	0.74, 0.71	0.36, 0.32	0.06, 0.06	0.82, 0.82	0.36, 0.36	0.05, 0.05
SMB	0.79, 0.76	0.44, 0.45	0.16, 0.12	0.71, 0.74	0.34, 0.37	0.13, 0.08	0.65, 0.59	0.42, 0.37	0.14, 0.11	0.67, 0.67	0.40, 0.41	0.09, 0.10	0.76, 0.76	0.43, 0.43	0.13, 0.13
HTTP	1.00, 1.00	0.91, 0.88	0.91, 0.88	0.99, 0.99	0.00, 0.00	0.00, 0.00	-, -	-, -	-, -	0.88, 0.88	0.58, 0.58	0.58, 0.58	0.97, 0.97	0.24, 0.00	0.24, 0.00
NTP	0.76, 0.63	0.17, 0.36	0.14, 0.36	0.66, 0.70	0.27, 0.42	0.00, 0.09	0.78, 0.78	0.24, 0.28	0.14, 0.18	0.80, 0.81	0.40, 0.41	0.27, 0.32	0.79, 0.72	0.00, 0.13	0.00, 0.00
DNS	0.76, 0.73	0.49, 0.30	0.16, 0.06	0.81, 0.77	0.40, 0.36	0.22, 0.22	0.74, 0.71	0.56, 0.53	0.00, 0.11	0.92, 0.91	0.82, 0.82	0.62, 0.62	0.75, 0.74	0.34, 0.32	0.15, 0.11
BGP	0.90, 0.90	0.12, 0.12	0.05, 0.05	0.89, 0.89	0.59, 0.59	0.45, 0.45	0.78, 0.78	0.00, 0.00	0.00, 0.00	0.83, 0.83	0.54, 0.54	0.28, 0.28	0.89, 0.89	0.59, 0.59	0.45, 0.45
TFTP	0.61, 0.61	0.04, 0.04	0.00, 0.00	0.43, 0.43	0.00, 0.00	0.00, 0.00	0.58, 0.58	0.22, 0.22	0.03, 0.03	0.94, 0.94	0.91, 0.91	0.86, 0.86	0.78, 0.78	0.00, 0.00	0.00, 0.00
Average	0.78, 0.77	0.44, 0.44	0.28, 0.25	0.69, 0.71	0.29, 0.34	0.13, 0.14	0.70, 0.70	0.42, 0.42	0.23, 0.23	0.77, 0.76	0.57, 0.56	0.35, 0.34	0.74, 0.71	0.34, 0.27	0.14, 0.09

TABLE X: Message type inference results of compared tools on $\underline{(S_{\text{BooFuzz}},\ S_{\text{DYNPRE}})}$ enhanced datasets ("-" indicates timeout).

Protocol	Netplier			Netzob			FieldHunter		
	Homogeneity	*Completeness*	*V-measure*	*Homogeneity*	*Completeness*	*V-measure*	*Homogeneity*	*Completeness*	*V-measure*
IEC61850-MMS	0.52, 0.51	0.89, 0.73	0.66, 0.60	0.70, 0.70	1.00, 1.00	0.82, 0.82	0.52, 0.52	0.74, 0.74	0.61, 0.61
S7comm	0.75, 0.55	0.78, 0.85	0.76, 0.67	0.83, 0.50	0.54, 1.00	0.66, 0.67	0.50, 0.83	1.00, 0.54	0.67, 0.66
Modbus	0.27, 0.27	1.00, 1.00	0.43, 0.43	0.51, 0.52	0.43, 0.43	0.47, 0.47	0.27, 0.27	0.33, 0.33	0.30, 0.30
MQTT-QoS1	0.98, 0.98	0.67, 0.29	0.79, 0.45	0.99, 0.99	1.00, 1.00	0.99, 0.99	1.00, 1.00	1.00, 1.00	1.00, 1.00
MQTT-QoS2	0.75, 0.75	0.55, 0.55	0.63, 0.63	0.75, 0.75	1.00, 1.00	0.86, 0.86	1.00, 1.00	1.00, 1.00	1.00, 1.00
AMQP	0.04, 0.03	0.74, 0.80	0.08, 0.06	0.04, 0.04	0.74, 0.94	0.08, 0.08	0.83, 0.83	0.53, 0.53	0.65, 0.65
SMB2	0.33, 0.81	0.92, 0.43	0.48, 0.29	-, -	-, -	-, -	0.96, 0.96	0.88, 0.88	0.92, 0.92
SMB	1.00, 1.00	1.00, 1.00	1.00, 1.00	0.30, 0.30	1.00, 1.00	0.46, 0.46	1.00, 1.00	1.00, 1.00	1.00, 1.00
HTTP	0.74, 0.74	1.00, 1.00	0.85, 0.85	-, -	-, -	-, -	1.00, 1.00	0.56, 0.56	0.72, 0.72
NTP	1.00, 1.00	1.00, 1.00	1.00, 1.00	1.00, 1.00	1.00, 1.00	1.00, 1.00	1.00, 1.00	1.00, 1.00	1.00, 1.00
DNS	1.00, 1.00	0.96, 1.00	0.98, 1.00	0.96, 0.96	1.00, 1.00	0.98, 0.98	0.96, 1.00	1.00, 0.96	0.98, 0.98
BGP	0.57, 0.57	1.00, 1.00	0.73, 0.73	0.89, 0.89	1.00, 1.00	0.94, 0.94	0.57, 0.57	1.00, 1.00	0.73, 0.73
TFTP	0.55, 0.55	0.28, 0.28	0.37, 0.37	0.51, 0.51	1.00, 1.00	0.68, 0.68	0.51, 0.51	1.00, 1.00	0.68, 0.68
Average	0.65, 0.68	0.83, 0.76	0.68, 0.62	0.68, 0.65	0.88, 0.94	0.72, 0.72	0.78, 0.81	0.85, 0.81	0.79, 0.79

TABLE XI: Message type inference results of each tool on various protocols with different dataset sizes ($\mathcal{H}$ for *homogeneity*, $\mathcal{C}$ for *completeness*, and $\mathcal{V}$ for *V-measure*).

Protocol	**#msg**	**DynPRE**			**Netplier**			**Netzob**			**FieldHunter**		
		$\mathcal{H}$	$\mathcal{C}$	$\mathcal{V}$	$\mathcal{H}$	$\mathcal{C}$	$\mathcal{V}$	$\mathcal{H}$	$\mathcal{C}$	$\mathcal{V}$	$\mathcal{H}$	$\mathcal{C}$	$\mathcal{V}$
IEC61850-MMS	10	1.00	1.00	1.00	1.00	1.00	1.00	0.57	0.41	0.48	1.00	0.60	0.75
S7comm	10	1.00	1.00	1.00	0.55	0.85	0.67	0.78	0.69	0.73	0.90	0.56	0.69
Modbus	10	1.00	1.00	1.00	0.51	0.88	0.64	0.52	0.88	0.66	1.00	0.88	0.94
MQTT-QoS1	10	1.00	1.00	1.00	1.00	1.00	1.00	0.79	1.00	0.88	1.00	1.00	1.00
MQTT-QoS2	10	1.00	1.00	1.00	1.00	1.00	1.00	0.70	1.00	0.82	1.00	1.00	1.00
AMQP	10	1.00	1.00	1.00	0.39	0.71	0.50	0.59	0.56	0.57	0.96	0.52	0.67
SMB2	10	1.00	1.00	1.00	0.34	1.00	0.51	0.45	0.77	0.57	1.00	0.88	0.94
SMB	10	1.00	1.00	1.00	0.34	1.00	0.51	0.86	0.93	0.89	0.93	0.87	0.90
HTTP	10	0.64	1.00	0.78	0.64	1.00	0.78	1.00	0.52	0.68	1.00	0.72	0.84
NTP	10	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
DNS	10	1.00	0.58	0.73	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.30	0.46
BGP	10	1.00	1.00	1.00	1.00	1.00	1.00	0.42	1.00	0.59	1.00	0.78	0.88
TFTP	10	1.00	1.00	1.00	0.71	1.00	0.83	1.00	1.00	1.00	0.41	1.00	0.59
Average-10		0.97	0.97	0.96	0.73	0.96	0.80	0.74	0.83	0.76	0.94	0.78	0.82
IEC61850-MMS	100	0.81	1.00	0.90	0.81	1.00	0.90	1.00	0.93	0.97	0.81	1.00	0.90
S7comm	100	1.00	1.00	1.00	1.00	1.00	1.00	0.75	0.71	0.73	1.00	1.00	1.00
Modbus	100	1.00	1.00	1.00	1.00	1.00	1.00	0.58	0.70	0.64	1.00	1.00	1.00
MQTT-QoS1	100	1.00	1.00	1.00	1.00	1.00	1.00	0.94	0.68	0.79	1.00	1.00	1.00
MQTT-QoS2	100	1.00	1.00	1.00	1.00	1.00	1.00	0.74	0.83	0.78	1.00	1.00	1.00
AMQP	100	1.00	1.00	1.00	0.18	0.77	0.29	0.23	0.68	0.35	0.83	0.51	0.63
SMB2	100	1.00	1.00	1.00	0.24	1.00	0.39	0.27	0.96	0.42	1.00	1.00	1.00
SMB	100	1.00	1.00	1.00	0.34	1.00	0.51	0.49	0.90	0.63	0.95	0.79	0.86
HTTP	100	0.79	1.00	0.88	0.79	1.00	0.88	1.00	0.38	0.55	1.00	0.29	0.45
NTP	100	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
DNS	100	0.97	0.80	0.88	0.97	0.29	0.44	0.97	1.00	0.99	0.97	0.80	0.88
BGP	100	1.00	1.00	1.00	1.00	0.80	0.89	0.57	1.00	0.72	1.00	0.80	0.89
TFTP	100	1.00	1.00	1.00	0.55	1.00	0.71	1.00	1.00	1.00	0.55	1.00	0.71
Average-100		0.97	0.98	0.97	0.76	0.91	0.77	0.73	0.83	0.74	0.93	0.86	0.87
IEC61850-MMS	1000	0.52	0.74	0.61	0.52	0.89	0.66	0.94	0.77	0.85	0.52	0.74	0.61
S7comm	1000	1.00	1.00	1.00	0.75	0.78	0.76	0.75	0.70	0.73	1.00	1.00	1.00
Modbus	1000	1.00	1.00	1.00	1.00	1.00	1.00	0.53	0.43	0.48	0.27	0.33	0.30
MQTT-QoS1	1000	1.00	1.00	1.00	0.98	0.67	0.79	0.99	0.46	0.63	1.00	1.00	1.00
MQTT-QoS2	1000	1.00	1.00	1.00	0.75	0.55	0.63	0.75	0.79	0.77	1.00	1.00	1.00
AMQP	1000	0.03	1.00	0.06	0.04	1.00	0.08	0.04	0.60	0.08	0.83	0.53	0.65
SMB2	1000	1.00	1.00	1.00	0.28	1.00	0.44	0.28	0.97	0.44	0.96	0.88	0.92
SMB	1000	1.00	1.00	1.00	0.37	0.79	0.50	0.41	0.77	0.53	1.00	1.00	1.00
HTTP	1000	0.74	1.00	0.85	0.74	1.00	0.85	1.00	0.36	0.53	1.00	0.56	0.72
NTP	1000	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
DNS	1000	1.00	0.96	0.98	0.98	0.16	0.27	1.00	1.00	1.00	1.00	0.96	0.98
BGP	1000	1.00	1.00	1.00	0.55	0.28	0.37	0.51	1.00	0.68	0.51	1.00	0.68
TFTP	1000	1.00	1.00	1.00	0.57	1.00	0.73	0.89	1.00	0.94	0.57	1.00	0.73
Average-1000		0.87	0.98	0.88	0.66	0.78	0.62	0.70	0.76	0.66	0.82	0.85	0.81

TABLE XII: DynPRE's runtime overhead and number of exchanged messages on different protocols for the dataset of 1000 messages.

Protocol	**Runtime Overheads**		**Exchanged Messages**
	Time (*min*)	Connections	
IEC61850-MMS	208	55854	2397585
S7comm	211	22565	5646067
Modbus	127	23738	4528680
MQTT-QoS1	52	28019	7044752
MQTT-QoS2	66	16231	4027658
AMQP	313	38785	242574
SMB2	691	122718	9849747
SMB	511	114542	4341266
HTTP	255	92106	192358
NTP	336	76800	2213180
DNS	28	20042	4089829
BGP	328	27443	3439450
TFTP	241	15630	3680608
Average	259	50344	3976443