Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2025.071552

ARTICLE

ProRE: A Protocol Message Structure Reconstruction Method Based on Execution Slice Embedding

Yuyao Huang, Hui Shu and Fei Kang*

Key Laboratory of Cyberspace Security, Ministry of Education, Zhengzhou, 450001, China

*Corresponding Author: Fei Kang. Email: kfminnie@163.com

Received: 07 August 2025; Accepted: 15 October 2025; Published: 12 January 2026

ABSTRACT: Message structure reconstruction is a critical task in protocol reverse engineering, aiming to recover protocol field structures without access to source code. It enables important applications in network security, including malware analysis and protocol fuzzing. However, existing methods suffer from inaccurate field boundary delineation and lack hierarchical relationship recovery, resulting in imprecise and incomplete reconstructions. In this paper, we propose ProRE, a novel method for reconstructing protocol field structures based on program execution slice embedding. ProRE extracts code slices from protocol parsing at runtime, converts them into embedding vectors using a data flow-sensitive assembly language model, and performs hierarchical clustering to recover complete protocol field structures. Evaluation on two datasets containing 12 protocols shows that ProRE achieves an average F1 score of 0.85 and a cophenetic correlation coefficient of 0.189, improving by 19% and 0.126% respectively over state-of-the-art methods (including BinPRE, Tupni, Netlifter, and *QwQ-32B-preview*), demonstrating significant superiority in both accuracy and completeness of field structure recovery. Case studies further validate the effectiveness of ProRE in practical malware analysis scenarios.

KEYWORDS: Protocol reverse engineering; program slicing; code embedding; hierarchical clustering

1 Introduction

Protocol Reverse Engineering (PRE) reconstructs protocol functionality by parsing the structure and semantics of communication messages, with increasingly widespread applications. In network security, PRE is used to analyze malware communication mechanisms and traffic characteristics [1], helping security researchers understand attackers' command and control protocols. In IoT security, where numerous devices use proprietary protocols lacking documentation, PRE becomes essential for discovering security vulnerabilities. In threat detection, it facilitates understanding and monitoring of unknown applications' network behavior. As protocols become increasingly encrypted and complex, traditional manual analysis has become inadequate, making automated PRE techniques crucial.

Message structure recovery is one of the most fundamental tasks in PRE, providing crucial support for further inferring protocol semantics and state machines. Its primary goal is to reconstruct the message structure of communication protocols without access to source code. Depending on the application scenario, methods are typically classified into Network Trace-based (NetT-based) [2,3] and Execution Trace-based (ExeT-based) [4–8] approaches. NetT-based methods parse field formats through statistical analysis of captured traffic data, but often perform poorly with insufficient traffic samples. Our research focuses on ExeT-based methods, which achieve relatively accurate recovery by dynamically tracing message processing

without requiring large amounts of traffic. However, existing ExeT-based methods face significant limitations in inference effectiveness. Two key challenges persist:

- *First, unreasonable boundary delineation methods lead to inaccurate recovery*. Field division relies on precise byte representation forms, yet existing methods typically employ only low-level features as approximations, such as program operator [4] or basic block [5] sequences. Without high-level abstraction of protocol semantics, these approaches are prone to errors when representing complex programs.
- *Second, absence of hierarchical relationship recovery results in incomplete message structures*. Multiple relationships exist among message fields [6], yet existing methods focus solely on sequential relationships, overlooking parallel and nested relationships. This limitation prevents complete restoration of message structures and comprehensive evaluation metrics.

To address these challenges, we propose ProRE, a novel method for **Pro**tocol field structures **Re**construction based on program execution slice embedding. ProRE precisely extracts code execution slices from the protocol parsing process at runtime, converts them into high-dimensional semantic vectors using a data flow-sensitive assembly language model, and introduces hierarchical clustering algorithms to iteratively aggregate these vectors, thereby completely recovering protocol field structural relationships including nesting and parallelism. This method achieves precise characterization of field boundaries by mapping program semantics to vector space, and reconstructs the protocol's inherent structure through hierarchical clustering trees, significantly improving recovery accuracy and completeness.

We summarize our contributions as follows:

- We propose a protocol message structure reconstruction method based on execution slice embedding, integrating protocol reverse analysis and program semantic analysis through code slice embedding and hierarchical clustering to address inaccurate and incomplete field recovery challenges.
- We develop a data flow-sensitive assembly language model that converts execution code slices into embeddings, achieving precise characterization of protocol field semantics.
- We introduce hierarchical clustering to protocol field recovery tasks for the first time, which can not only divide sequential fields but also restore structural relationships. We also propose a structured evaluation method using the cophenetic correlation coefficient for comprehensive protocol structure assessment.
- We design and implement the ProRE. Evaluation on two datasets containing 12 protocols shows that ProRE achieves an average F1 score of 0.85 and a cophenetic correlation coefficient of 0.189, improving by 19% and 0.126% respectively over state-of-the-art methods (including BINPRE, TUPNI, NETLIFTER, and *QwQ-32B-preview*), demonstrating significant superiority in both accuracy and completeness of field structure recovery. Case study shows that ProRE effectively applies to practical malware analysis scenarios, successfully identifying *Duke* steganographic data and *Mirai* botnet protocol message structures. We also open-source this work to facilitate future research.

The remainder of this paper is organized as follows. Section 2 reviews related work in code embedding and protocol field recovery, highlighting existing gaps. Section 3 presents background knowledge and motivates our approach through concrete examples. Section 4 details our methodology, including execution code slicing, slice embedding, and hierarchical clustering techniques. Section 5 describes the implementation details. Section 6 presents comprehensive experimental evaluation, including performance comparisons, ablation studies, and case studies on real malware. Section 7 discusses time consumption, generalization capability, and limitations. Finally, Section 8 concludes the paper and outlines future research directions.

2 Related Work

This section reviews existing approaches in code embedding and protocol field recovery, examining their techniques and limitations to establish the context for our proposed method.

2.1 Code Embedding

Code embedding is a program representation method that achieves semantic abstraction by converting program code into numerical vectors, commonly applied in code similarity detection and vulnerability analysis. These methods treat code as natural language, learning embedding representations by training deep learning models. The most intuitive embedding method is one-hot encoding [9,10], which uses fixed-dimension vectors to map instructions, but limited by abstraction capability, cannot fully represent syntactic and semantic information. Some works [11–14] analogize instructions and basic blocks to words and sentences, respectively, constructing code embedding vectors through representation learning with strong generality. However, they cannot capture semantics of specific code fragments and lack abstraction of data flow relationships.

This paper constructs a data flow-sensitive assembly language model to convert slice code of protocol processing into embedding vectors, better representing execution semantics of communication programs and supporting protocol analysis.

2.2 Protocol Field Recovery

Protocol Reverse Engineering (PRE) infers network protocol specifications from traffic or program binaries [15,16], enabling applications like traffic classification [17] and protocol vulnerability detection [4,5]. Field structure recovery is one of the core tasks, typically divided into two categories based on application scenarios. NetT-based methods [2,3] infer message formats through statistical analysis of traffic. However, their accuracy depends on rich, high-quality traffic, which is difficult to obtain in practice. ExeT-based methods [4–8] do not rely on traffic data, identifying field boundaries by dynamically analyzing communication program execution traces and clustering semantically similar bytes. However, their recovery results remain inaccurate or incomplete.

BinPRE [4], the current state-of-the-art, approximates field semantics through operator sequences extracted during protocol parsing. However, these low-level features fail to capture complex program semantics, leading to frequent field over-segmentation when execution sequences differ significantly. Tupni [7] employs taint analysis for data tracking but cannot handle nested field structures, limiting its applicability to modern protocols. AIFORE [5] utilizes basic block analysis for rapid processing but lacks the semantic depth required for accurate boundary detection. AutoFormat [6] combines instruction addresses with function call stacks to achieve comprehensive tracing, yet suffers from excessive computational overhead and tendency toward over-segmentation. Netlifter [18] attempts static analysis to avoid runtime costs but misses critical dynamic execution paths essential for accurate protocol reconstruction.

We improve field structure recovery performance through: (1) embedding code slices as representations in semantic space; (2) applying hierarchical clustering algorithms to restore message formats.

3 Background and Motivation

In this section, we introduce the underlying background, key concepts, and the motivation that drives our proposed approach.

3.1 Problem Description

We consider the following scenario: a security analyst needs to determine the protocol message format of a network communication program. They cannot obtain source code or protocol specification descriptions in any form. They possess a binary executable without source code or debug symbols that can be correctly disassembled and decompiled. This program can run on a dynamic instrumentation platform to record message exchanges with communication peers. While our method is architecture-agnostic, we assume the program runs on Windows x86 for evaluation purposes.

3.2 Background

Protocol Reverse Engineering

Protocol reverse engineering infers protocol specifications by analyzing network traffic or program execution behavior without protocol documentation. Protocol specifications typically contain three core elements: *Message Format*, *Protocol Semantics*, and *State Machine*. Message format defines the structured organization of messages, including field boundaries, types, and hierarchical relationships; protocol semantics describes the meaning and processing logic of each field; state machine characterizes the protocol dynamic behavior and state transition rules. This paper focuses specifically on message format recovery.

Program Slicing

Program slicing is a program analysis technique that extracts statement subsets related to specific computations from programs, such as protocol parsing portions in communication programs. For a given slicing criterion (typically a program location and variable), a program slice contains all statements that may affect that variable's value. In dynamic program slicing, slices are computed based on specific execution paths, enabling more precise capture of actual data dependencies. Dynamic slicing typically follows these steps: (1) Execute the program and record execution traces; (2) Construct dynamic data dependency graphs; (3) Collect relevant statements by traversing the dependency graph backward from the slicing criterion.

Code Representation Learning

Code representation learning maps program code into continuous vector space such that semantically similar code fragments are proximate in vector space. Early methods primarily encoded programs based on structural features (such as abstract syntax trees and control flow graphs). Recently, inspired by natural language processing advances, researchers have applied deep learning techniques to code understanding tasks. The Transformer architecture demonstrates excellent performance in code representation learning due to its powerful sequence modeling capabilities. Through pre-training tasks (such as masked language models), models learn code syntactic structures and semantic patterns.

3.3 Motivation

Existing protocol field recovery methods face significant challenges in accurately identifying field boundaries and recovering hierarchical structures. To illustrate these limitations, we analyzed message field parsing results on an example communication program. Fig. 1 demonstrates how current state-of-the-art methods perform, where BINPRE [4] represents the latest ExeT-based approach and AUTOFORMAT [6] serves as a classic baseline.

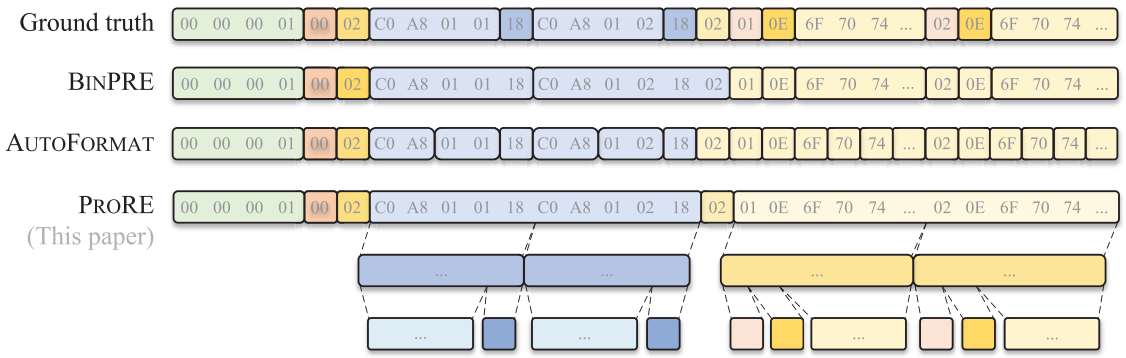

Figure 1: Message parsing results of existing methods on communication sample

*Challenge 1 (***C1***): Existing methods employ inadequate boundary delineation approaches, resulting in inaccurate recovery.*

Field division is typically considered a message byte clustering process [4], where bytes with similar semantics merge into protocol fields. Existing methods approximate these semantics through low-level program features during protocol parsing, such as instruction addresses, function call stacks, with data reference records [6], or operator sequences [4]. However, since these features cannot accurately reflect program semantics, errors frequently occur. As shown in Fig. 1, when execution sequences corresponding to different bytes of the same field differ significantly, excessive clustering and field over-segmentation may result, such as bytes 7–10 and 12–15 recovered by AutoFormat. Conversely, highly similar execution sequences may cause under-segmentation errors, such as bytes 10 and 11 recovered by BINPRE.

Recent advances in code pre-trained language models [11–13] provide opportunities for precisely representing program semantics. In this paper, ProRE designs a data flow-sensitive assembly language model to convert program traces into embedding vectors as abstract byte representations. This approach better abstracts program semantic features, thereby improving field division accuracy.

*Challenge 2 (***C2***): Existing methods lack hierarchical relationship recovery, resulting in incomplete message structures.*

Multiple relationships exist among message fields [6], yet existing methods focus solely on sequential relationships, ignoring parallel and nested relationships. This limitation prevents complete restoration of message structures and comprehensive evaluation metrics. As shown in Fig. 1, neither BINPRE nor AutoFormat recover the nested message structure. They divide all bytes into field clusters at the same level, creating difficulties for inferring field semantics and understanding message structures.

To address this challenge, we introduce hierarchical clustering methods to message field recovery for the first time. ProRE gradually restores messages inherent structure by iteratively clustering fields. As shown in Fig. 1, it completely extracts nested results (light blue and light yellow fields) and parallel structures (dark blue and dark yellow fields). Additionally, we introduce structured evaluation methods for field recovery, providing comprehensive assessment of message parsing results.

We have also analyzed existing similar works, as shown in Table 1, and they generally have the problems involved in the above two challenges. Among them, although TUPNI supports the recovery of some cross-field dependencies and NETLIFTER can partially extract the structural relationships existing in decompiled code, they are not as complete and accurate as ProRE.

Table 1: Summary of existing approaches and the challenges

Method	Venue	C1	C2
BINPRE [4]	CCS'24	✗	✗
TUPNI [7]	CCS'08	✗	◯
AIFORE [5]	USENIX SEC'23	✗	✗
AUTOFORMAT [6]	CCS'08	✗	✗
NETLIFTER [18]	CCS'23	✗	◯
PRORE	(This paper)	✓	✓

Note: ✓: resolved, ✗: unresolved, ◯: partially resolved.

4 Methodology

Our methodology addresses the fundamental challenges in protocol field recovery through a three-phase approach that progressively transforms program execution traces into structured protocol representations. Fig. 2 presents the overall framework of PRORE, which consists of three integrated phases: execution code slicing (Section 4.1), slice embedding (Section 4.2), and hierarchical clustering (Section 4.3).

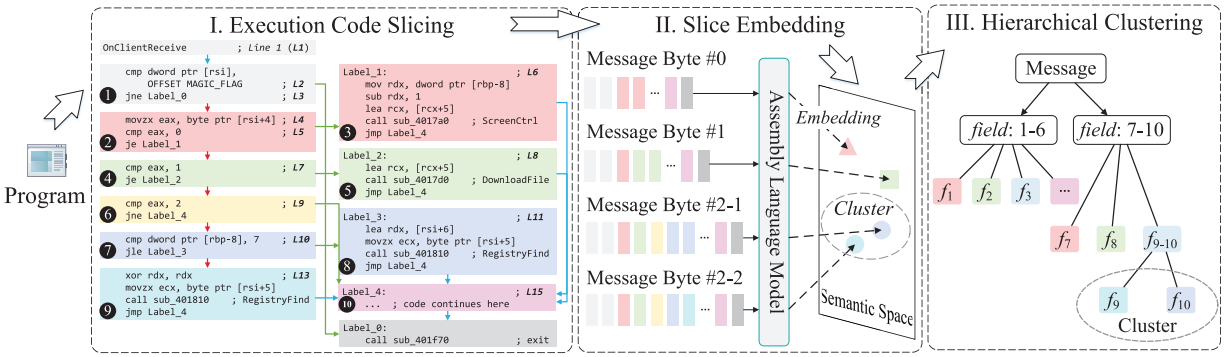

Figure 2: Overall framework of ProRE. It takes a communication program as input and outputs inferred protocol field structures via three main phases

4.1 Execution Code Slicing

Formally, for communication program P, we define the following:

Definition 1 (Message Byte-Guided Code Slice): *Given a slicing criterion $\theta = \langle f_0, i, M \rangle$, where f_0 is a reception point in P related to the i-th byte of received message M ($1 \le i \le |M|$), this criterion specifies extracting all basic blocks with data flow dependencies on the i-th byte starting from f_0. We denote the set of basic blocks satisfying θ as the message byte-guided code slice S_i.*

For clarity, we describe the message reception process by default; the technical principles for the sender remain consistent. To accurately extract code fragments from the protocol parsing process, we perform hybrid static-dynamic analysis on the communication program, then extract code slices as shown in Fig. 3. Specifically, we first record disassembled instruction sequences indexed by address on the static analysis engine, along with basic blocks (BBLs) and functions information. To separate slice sets S corresponding to different message bytes, PRORE executes a code slice extraction algorithm, as shown in Algorithm 1.

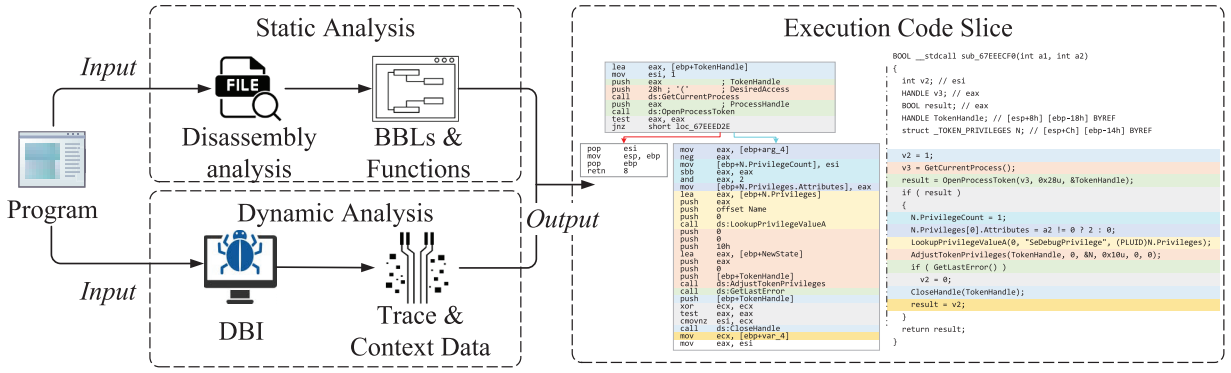

Figure 3: Slice extraction process. Colors identify processing code for different message bytes

Algorithm 1: Code slice extraction

$S \leftarrow \varnothing$, Cache $\leftarrow \varnothing$
L, context $\leftarrow$ Execute(P, M, f_0)
function ExtractDataFlow (l, ctx)
    $t \leftarrow$ InitTemplate()
    if t in Cache **then**
        return t
    end if
    SetContext(ctx)
    for $i = 1$ **to** $|l|$ **do**
        $t[i] \leftarrow$ GetTemplate$(l[i])$
        Cache.update(t)
    end for
    return t
end function
for $i = 1$ **to** $|M|$ **do**
    $S_i \leftarrow \varnothing$
    for $j = 1$ **to** $|L|$ **do**
        $t \leftarrow$ ExtractDataFlow(L [j], context [j])
        $S_i \leftarrow S_i +$ ExtractSlice(t, i, θ)
    end for
end for
return S

Slice extraction accuracy directly affects subsequent analysis quality. Traditional static slicing methods often produce excessive code due to path explosion and pointer alias analysis difficulties. While dynamic slicing offers greater precision, it requires balancing execution efficiency and coverage. We adopt a hybrid strategy: collecting necessary runtime information through lightweight online instrumentation, then performing detailed data flow analysis offline. This design ensures both analysis accuracy and minimal impact on program execution.

Specifically, ProRE first performs Dynamic Binary Instrumentation (DBI) on the program, recording execution traces and context data after receiving messages. Then, ProRE traverses basic blocks on the

execution path, extracting code slices based on data flow templates. Templates record data propagation relationships during each basic block's execution. To minimize impact on actual program execution, ProRE's template generation occurs offline and employs caching mechanisms to prevent repeated analysis. We introduce a caching mechanism based on the following observation: a significant portion of data flow analysis time is spent analyzing internal instructions of known APIs, whose data flow propagation relationships are well-documented. Based on these specifications, we manually extracted and compiled 1684 API semantic summaries from commonly used standard libraries, containing information about parameter variable types and data flow propagation directions. This approach alleviates unnecessary overhead in data flow analysis.

4.2 Slice Embedding

Recent advances in code pre-trained language models [11–13] provide new opportunities for precisely representing program semantics. These models learn not only syntactic structures but also highly abstract semantics by converting code into vector representations. Inspired by this progress, we design a data flow-sensitive assembly language model as shown in Fig. 4, which represents byte semantics by converting code slices into embedding vectors to better support field recovery. The fundamental assumption is that a mapping exists from program code space to abstract semantic space that equivalently expresses program semantics in vector space. We approximate this semantic mapping through an attention mechanism-based Transformer encoder-decoder model.

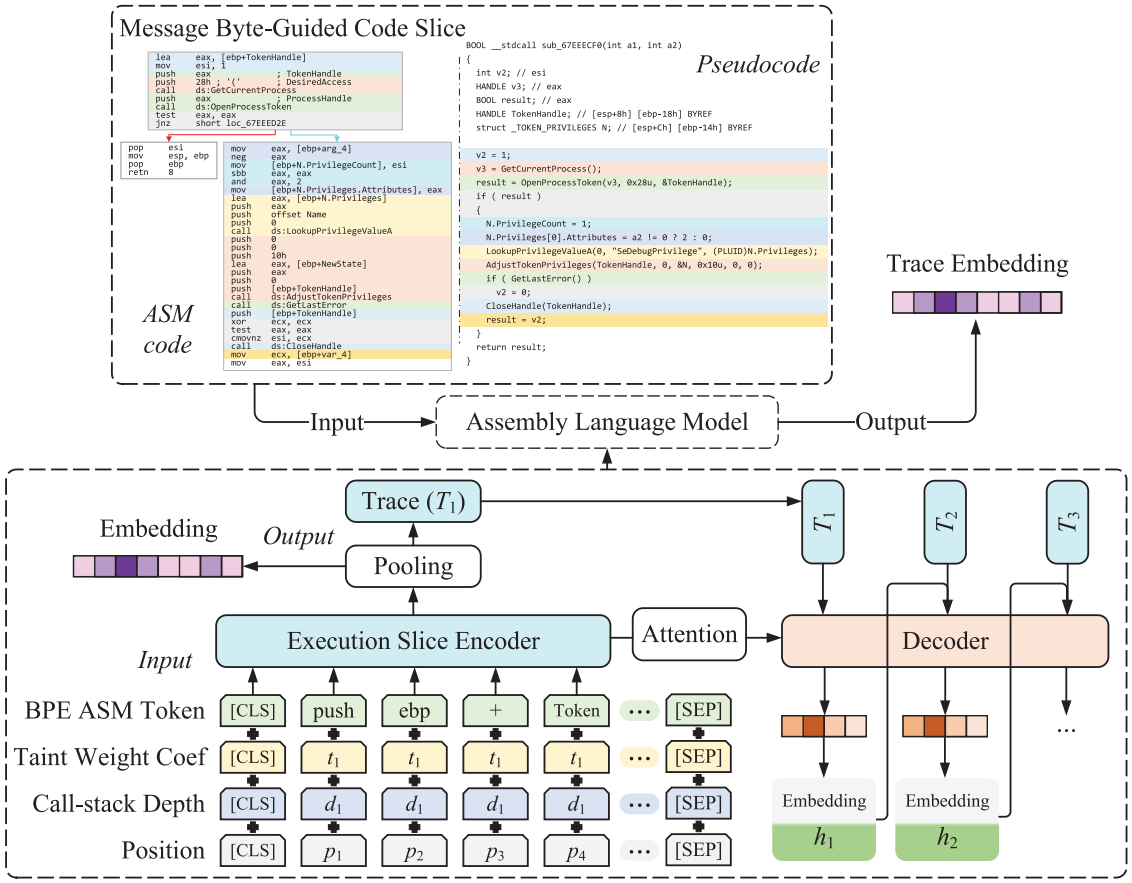

Figure 4: Architecture of the data flow-sensitive assembly language model

Building code embedding models that meet protocol reverse analysis requirements presents challenges. Unlike general code language models, communication program analysis requires greater focus on message processing, particularly data flow relationships between instructions, which existing methods have not achieved. Therefore, we construct a data flow-sensitive assembly language model with specialized input encoding layers and training strategies.

Input Encoding Layer

As shown in Fig. 4, the model's input comprises four encoding components. *BPE Assembly (ASM) Token* employs byte pair encoding to tokenize assembly instructions and builds a 30 K-token vocabulary. *Taint Weight Coefficient* calculates each instruction's impact weight on message bytes based on taint analysis results, enabling the model to focus on code fragments closely related to message processing while excluding irrelevant segments. This coefficient is calculated from the intersection size between memory regions read/written by instructions and message input taint regions. *Call Stack Depth* helps the model understand instructions relative positions in the program, as code at the same hierarchical depth typically exhibits closer semantic relevance. We set the program entry point as the initial stack bottom and return instructions relative offset on the function call stack. *Position* encodes instructions absolute order. To balance training efficiency and effectiveness, we set the maximum sequence length to 1024 tokens, with exceeding instruction tokens truncated.

Training Tasks

We enhance the model's awareness of program semantics in message processing through three pre-training tasks.

- **Masked Language Model (MLM):** This task captures assembly instruction syntax fundamentals. Following standard MLM settings [19], we randomly select 15% of tokens for masking: 80% replaced by [MASK] tokens, 10% replaced by random tokens, and 10% unchanged. By predicting masked original tokens, the model learns syntactic structures and contextual semantics of assembly instructions. The loss function is:

$$\mathcal{L}_{MLM} = -\sum_{i \in M} \log P(x_i | x_{\setminus i})$$

where M denotes masked positions, x_i represents the token at position i in the sequence, and $x_{\setminus i}$ represents all tokens except position i.

- **Context Window Prediction (CWP):** This task captures program control dependencies by predicting whether two instructions co-occur within the same control branch's influence range. For instructions i_a and i_b, we calculate their shortest path distance in the control flow graph. If the distance is less than predefined window size w, they are marked as positive samples; otherwise negative. Through binary classification, the model learns control flow dependencies between instructions:

$$\mathcal{L}_{CWP} = -\sum_{(i_a, i_b)} [y \log p + (1 - y) \log(1 - p)]$$

where $y \in \{0, 1\}$ indicates whether they share the same window, and p is the model's predicted probability.

- **Def-Use Prediction (DUP):** This task explicitly models data dependencies between instructions. Fig. 5 shows a decompiled code fragment of *Mirai* attack message parsing, where "`len`" is a masked message field from which we extract Def-Use relationships on its data dependency graph. For statement i that defines or uses a memory location, we extract code with data dependencies. We enhance the model's attention to message processing code by adding a specialized "Dataflow Seeing Matrix" atop the model's

original attention matrix. Attention weights are assigned based on data dependency proximity: code with direct message data dependencies receives weight 1.0, while connected adjacent code receives 0.2 to radiate attention. During training, we adopt a contrastive learning framework: for positive samples (ground truth Def-Use pairs), the model outputs high similarity; for negative samples (randomly sampled pairs), the model outputs low similarity. The loss function is:

$$\mathcal{L}_{DUP} = \sum_{(i,j)\in\mathcal{D}^+} \max(0, \gamma - s(h_i, h_j)) + \sum_{(i,k)\in\mathcal{D}^-} \max(0, s(h_i, h_k) - \gamma)$$

where $\mathcal{D}^+$ and $\mathcal{D}^-$ are positive and negative sample sets, s is the similarity function, γ is the margin parameter, and h is the hidden layer representation of instruction i. Attention matrix weights are directly added for computation.

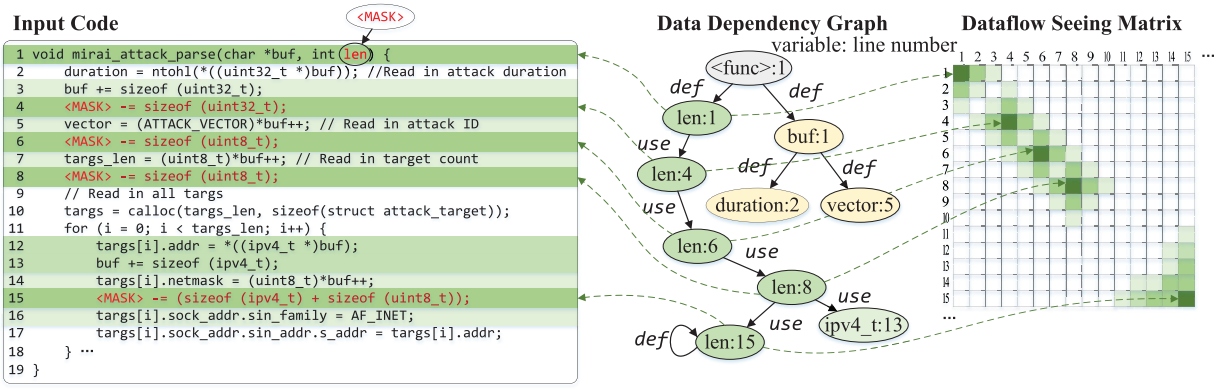

Figure 5: Def-Use prediction task with dataflow seeing matrix. Green positions indicate code fragments receiving model focus, with darker colors indicating higher attention weights

The total training objective combines three tasks as a weighted sum:

$$\mathcal{L}_{total} = \lambda_1 \mathcal{L}_{MLM} + \lambda_2 \mathcal{L}_{CWP} + \lambda_3 \mathcal{L}_{DUP}$$

where λ_1, λ_2, λ_3 are trainable parameters. The model training process takes assembly sequences corresponding to functions as samples, inputs them through the encoding layer, executes pre-training tasks, and learns assembly language syntax and semantics to provide higher-quality semantic representations. During inference, the model takes program execution code slices as input and outputs corresponding embedding representation vectors.

4.3 Hierarchical Clustering

Considering the inherent hierarchical structure of protocol messages [6], we represent fields using sets and express field nesting relationships through set inclusion. Our core insight is that the process of deconstructing nested fields naturally reflects message structure. Processing traces of different sub-fields within the same parent field exhibit greater similarity than those across different parent fields. By comparing semantic similarities between different bytes, we can restore message nested structure. To achieve field structure recovery, we introduce hierarchical clustering methods.

Hierarchical clustering is an unsupervised learning method that organizes data by constructing nested cluster hierarchies. It begins with each data point as a separate cluster, iteratively merging the most similar cluster pairs until all data points belong to a single cluster. This method offers several advantages: (1) no

requirement to pre-specify cluster numbers; (2) ability to reveal multi-scale data structures; (3) generated dendrograms intuitively display data hierarchical relationships. In protocol analysis scenarios, hierarchical clustering particularly suits capturing nested protocol field structures.

ProRE's hierarchical clustering process is described in Algorithm 2. The core concept builds hierarchical structures by iteratively merging field classes with the closest semantics. Taking slice embeddings representing byte semantics as input, it performs clustering by calculating distances at different levels to recover field structures.

Algorithm 2: Field structure recovery based on hierarchical clustering

$m \leftarrow |S|, k \leftarrow m$
for $i = 1$ **to** m **do**
  $r_i \leftarrow \delta(S_i)$ ▷ Compute semantic embedding for byte i
end for
$R_m \leftarrow \{C_1, C_2, \ldots, C_m\}$, where $C_i = \{r_i\}, i = 1, 2, \ldots, m$
while $k > 1$ **do**
  Compute the distance $d(C_i, C_j)$ between any two classes C_i and C_j
  $(C_p, C_q) \leftarrow \underset{C_i, C_j \in C, i \neq j}{\operatorname{argmin}}\ d(C_i, C_j)$ ▷ Find closest clusters
  $C_{new} \leftarrow C_p \cup C_q$ ▷ Merge clusters
  $R_{k-1} \leftarrow (R_{k-1} \backslash \{C_p, C_q\}) \cup C_{new}$ ▷ Update cluster set
  $k \leftarrow k - 1$
end while
$T \leftarrow \{R_1, R_2, \ldots, R_m\}$
return $\{T\}$

Definition 2 (Abstraction Extraction Function): *For element p in the program concrete domain and element r in the semantic abstract domain, the abstraction function is:*

$$\delta(p) = r, (p \in P, r \in R) \tag{1}$$

where r represents the abstract semantic representation of p under Galois connection.

For code slice S_i corresponding to the i-th message byte, ProRE employs the assembly language model as the abstraction extraction function δ to compute embedding r_i. We then formalize field structure recovery as a hierarchical clustering process on slice embeddings. Let the complete set of m slice embedding representations be: $\Omega = \{r_1, r_2, \cdots, r_m\}$.

Definition 3 (n–cluster): *Let L^n be a set consisting of n arbitrary subsets C of Ω ($C_i \subseteq \Omega$, $i = 1, \cdots, n$). If:*

$$\begin{cases} C_i \neq \varnothing \\ \bigcup_{i=1}^{n} C_i = \Omega \\ C_i \cap C_j = \varnothing, i \neq j, j = 1, 2, \cdots, n \end{cases} \tag{2}$$

then L^n is an n-cluster of Ω. For any two clusters L^p and L^q of Ω, where $p > q$, if each class in L^p is a subset of some class in L^q, then L^p is nested in L^q, denoted as $L^p \sqsubseteq L^q$.

Our goal is to generate the *field cluster tree T*, represented as an m-leaf binary tree with multiple levels L_i, each level consisting of one or more field classes. As shown in Fig. 6, field cluster tree $T = \{L_i\}, i = 1, \cdots, 5$,

where $L_5 \sqsubseteq L_4 \sqsubseteq L_3 \sqsubseteq L_2 \sqsubseteq L_1$. We calculate semantic similarity between different bytes through the following distance as the basis for clustering:

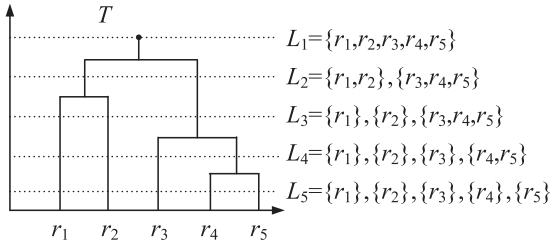

Figure 6: Field cluster tree

Definition 4 (Semantic Distance):

$$d_{C_i,C_j} = \frac{\sum_{x \in C_i} \sum_{y \in C_j} d_{xy}}{|C_i||C_j|} \tag{3}$$

The hierarchical clustering process starts from m isolated field classes, each class mapping to single byte semantics. Then, it iteratively merges the closest classes in a bottom-up manner until reaching the root node. Specifically, in each iteration, based on the distance calculated by Eq. (3), the two closest classes among the current k field classes are merged to form new $k-1$ field classes. The final output T contains field structures with complete hierarchical relationships.

5 Implementation

ProRE uses IDA Pro [20] and Intel Pin [21] as static and dynamic analysis engines, respectively. The assembly language model consists of a 12-layer Transformer encoder-decoder. Each layer contains standard multi-head self-attention mechanisms and feed-forward neural network (FFN) modules. The attention mechanism uses 12 attention heads, and the FFN layer uses GELU activation function. To enhance generalization capability, we apply Dropout (rate = 0.1) in attention and FFN layers. During pre-training, we use FP16 mixed precision training for acceleration with a batch size of 256. The learning rate peak is set to 5e-4 with linear warmup. We set $\lambda_1 = 1.0$, $\lambda_2 = 0.5$, $\lambda_3 = 0.5$.

6 Evaluation

Our evaluation aims to answer the following research questions:

- **RQ1:** What is the overall performance of ProRE in field structure recovery?
- **RQ2:** How does ProRE compare with other baseline methods?
- **RQ3:** Are the implementations of ProRE's modules beneficial for improving effectiveness?
- **RQ4:** How effective is ProRE in real-world tasks?

6.1 Setup

This section introduces the datasets, baseline methods, and metrics for experimental evaluation.

6.1.1 Datasets

We selected evaluation datasets from both common communication programs and malicious samples based on the following principles: (1) diverse protocol types; (2) message structures ranging from simple to complex; (3) real-world applicability with broad representativeness.

Common Protocol Dataset ($\mathcal{D}_{com}$)

We curated a common protocol set $\mathcal{D}_{com}$ containing 6 protocols, including common Ethernet protocols Ethernet, FTP, TFTP, HTTP, and industrial control protocols Modbus, S7comm, as shown in Table 2.

Table 2: Details of dataset $\mathcal{D}_{com}$

Protocol	Project	Total field	Message types
Ethernet	OpENer	37	Register Session; Unregister Session; Send RR Data; List Identity.
FTP	LightFTP	36	USER; PASS; PWD; LIST; RETR; STOR; DELE; MKD; RMD; PASV; QUIT.
TFTP	tftp-hpa	5	ACK; DATA.
HTTP	miniweb	97	CURL PARAM; CURL PATH; CURL -H; CURL -L; CURL -I.
S7comm	snap7	28	Job: Setup communication; Job: Read Var; Job: Write Var; Userdata: Request.
Modbus	freemodbus	60	Read Coils; Read Discrete Inputs; Read Holding Registers; Read Input Registers; Write Single Coil; Write Multiple Coils; Write Multiple Registers Job: Setup communication; Job: Read Var; Job: Write Var; Userdata: Request.

Malicious Protocol Dataset ($\mathcal{D}_{mal}$)

Existing research lacks protocol benchmarks for malware, so we manually created a sample set from six popular malware families to evaluate ProRE in malicious protocol analysis, as shown in Table 3. Among these, *Gh0st* uses a custom binary protocol, while *Mirai* implements lightweight IoT protocols. *Cobalt Strike* and *Sliver* represent modern C2 frameworks with complex protocol designs. The samples range from simple single-field protocols (*Sliver*) to complex multi-layered structures (*Mythic*) with multiple message types. All selected families are actively deployed in real attacks, with publicly available samples and verifiable protocol behaviors.

Table 3: Details of dataset $\mathcal{D}_{mal}$

Malware family	Message types	Lines of code
Gh0st	20	7301
Mirai	3	5113
Cobalt Strike	31	2934
Sliver	1	119
Revenant	5	1810
Mythic	3	892

6.1.2 Baselines

We compare ProRE with following methods:

- BINPRE [4]: The state-of-the-art ExeT-based method. We supplemented its support for floating-point extension instructions and ported it to the Windows platform for broader analysis applicability.
- TUPNI [7]: A classic work in state-of-the-art ExeT-based protocol reverse analysis methods, using an open-source reimplementation [4].
- NETLIFTER [18]: Aims to lift protocol implementation code to BNF protocol format. For fair comparison, we used decompiled code fragments parsing protocols as input and wrote targeted stub code to drive NETLIFTER, also performing unified conversion of its original BNF output.
- *QwQ-32B-preview* [22]: We use the latest open-source large reasoning model (LRM) as a representative of LLM-based inference methods. For fair comparison, we carefully designed prompts for field structure recovery tasks, specifying the correspondence between fields and variables.

We did not consider more well-known ExeT-based methods [6] as their performance has been surpassed by state-of-the-art work [4]. Due to fundamental differences in application scenarios and technical principles, NetT-based methods [2,3] are also not within our comparison scope.

6.1.3 Metrics

We use the following metrics for evaluation.

Slice

For two code slices, let the relatedness score (Re) be the Spearman coefficient [23] calculated from their embedding vector cosine similarity and manual judgment results. Let the categorization score (Ca) be the clustering purity of slice embedding vectors in the labeled sample set; the coherence score (Co) be the accuracy of nearest neighbor code slices calculated from embedding vectors on manually determined similar samples.

Field Recovery

We use macro-F1 score and cophenetic correlation coefficient (c) to evaluate performance. For a message containing k fields, the precision, recall, and F1 score of the i-th field are:

$$prc(i) = \frac{TP_i}{TP_i + FP_i},\ rec(i) = \frac{TP_i}{TP_i + FN_i},\ f1(i) = \frac{2 \cdot prc(i) \cdot rec(i)}{prc(i) + rec(i)} \tag{4}$$

where TP_i is correctly classified bytes in field i, FP_i is misclassified bytes not belonging to field i, and FN_i is missed bytes in field i. The macro-F1 score for all fields is:

$$F1 = \sum_{i=1}^{k} \sigma_i f1(i) \tag{5}$$

with $\sigma_i = 1$.

We propose using the cophenetic correlation coefficient (c) to measure consistency between recovered and actual format structures:

$$c = \frac{\sum_{i<j}(d_{ij} - \bar{y})(z_{ij} - \bar{z})}{\sqrt{\sum_{i<j}(d_{ij} - \bar{y})^2 \sum_{i<j}(z_{ij} - \bar{z})^2}} \tag{6}$$

where d_{ij} and z_{ij} are distances in the compressed distance matrix (Y) and linkage matrix (Z), respectively, $\bar{y}$ and $\bar{z}$ are their averages. c close to 1 indicates good consistency; close to −1 indicates poor consistency. We use SciPy [24] to calculate Z, Y, and c.

6.2 Performance

The field structure recovery performance on the common protocol dataset $\mathcal{D}_{com}$ is described in Table 4. The results show that ProRE achieves excellent performance on the common protocol set, with an average F1 of 0.89 and coefficient c of 0.229, significantly exceeding all baseline methods. Particularly on text protocols with complex nested structures like FTP and HTTP, ProRE's advantages are more pronounced. For example, on FTP protocol, ProRE achieves F1 of 0.90 and coefficient c of 0.115, while BINPRE only achieves 0.63 and 0.068. This is mainly due to our hierarchical clustering method's ability to effectively capture protocol nesting relationships. TUPNI shows consistently lower performance (average F1 of 0.70), primarily due to its reliance on taint analysis without semantic abstraction. Its performance particularly degrades on text protocols like HTTP (F1 of 0.43) where field boundaries are less distinct. NETLIFTER, despite its static analysis approach, achieves moderate results on binary protocols (Modbus F1 of 0.91) but struggles with complex text protocols due to missing dynamic execution paths. On binary protocols like Modbus and S7comm, performance differences between methods are relatively small because these protocols have relatively simple structures with clear field boundaries. However, even in these cases, ProRE maintains performance comparable to the best methods. Notably, on OpENer, an Ethernet protocol developed for I/O devices, although BINPRE matches ProRE in F1, ProRE still leads by 44% in coefficient c, demonstrating our method advantage in identifying field structures.

Table 4: Performance comparison of ProRE on dataset $\mathcal{D}_{com}$ (best in bold)

Protocol	ProRE		BINPRE		TUPNI		NETLIFTER		*QwQ-32B-preview*	
	***F*1**	***c***	***F*1**	***c***	***F*1**	***c***	***F*1**	***c***	***F*1**	***c***
Modbus	0.96	**0.219**	**0.98**	0.209	0.88	0.073	0.91	0.118	0.91	0.189
S7comm	0.88	**0.304**	0.88	0.258	**0.91**	0.090	0.90	0.214	0.89	0.170
Ethernet	**0.86**	**0.270**	**0.86**	0.188	0.57	0.066	0.55	0.010	0.51	0.028
FTP	**0.90**	**0.115**	0.63	0.068	0.55	0.024	0.52	0.107	0.53	−0.023
HTTP	**0.73**	**0.092**	0.42	0.026	0.43	0.009	0.38	−0.011	0.32	−0.102
TFTP	**1.00**	**0.374**	0.92	0.315	0.85	0.110	0.84	0.117	0.84	0.201
Average	**0.89**	**0.229**	0.78	0.177	0.70	0.062	0.68	0.092	0.67	0.077

The performance on the malware protocol dataset $\mathcal{D}_{mal}$ is summarized in Table 5. Overall, ProRE achieves an average $F1/c$ of 0.82/0.149, 5%–35%/0.103–0.145 higher than baseline tools. While ProRE cannot perfectly recover all message fields, except for samples like *Sliver* containing a single long field, it performs best. Due to significant differences in processing instructions between the beginning and ending bytes of *Sliver* messages, ProRE's byte-level slicing may produce errors. ProRE performs particularly well on *Gh0st* and *Cobalt Strike*, with average $F1$ scores 9% higher than BINPRE, because these samples use numerous API calls, making BINPRE's semantic computation more error-prone. For *Sliver* with only one field, our method's over-segmentation of its internal structure results in lower coefficient c than BINPRE. However, for samples with two or more structural layers (43% of the evaluation dataset), our method achieves higher coefficient c, particularly for *Mirai* and *Mythic* with many nested fields. For *Revenant*, all methods show

relatively lower F1 scores (0.41–0.62) due to its use of custom obfuscation techniques that complicate semantic analysis. Compared to ProRE, netlifter has lower average $F1/c$ because it relies on static analysis and misses dynamic call paths. Notably, despite powerful code understanding capabilities of *QwQ-32B-preview*, its performance on complex tasks like protocol field structure recovery remains limited, with average $F1/c$ of 47%/0.004; its average precision exceeds recall, indicating a cautious strategy focused on reducing false positives rather than inferring more fields.

Table 5: Performance comparison of ProRE on dataset $\mathcal{D}_{mal}$ (best in bold)

	ProRE		binpre		tupni		netlifter		*QwQ-32B-preview*	
	***F*1**	***c***	***F*1**	***c***	***F*1**	***c***	***F*1**	***c***	***F*1**	***c***
Gh0st	**0.80**	**0.124**	0.72	0.117	0.67	0.065	0.61	0.008	0.42	−0.048
Mirai	**0.99**	**0.208**	0.96	0.020	0.80	0.042	0.68	0.045	0.42	0.028
Cobalt Strike	**0.90**	**0.142**	0.81	−0.018	0.74	−0.024	0.71	−0.023	0.51	0.026
Sliver	**0.72**	−0.017	0.70	**0.005**	0.68	−0.003	0.68	−0.011	0.54	−0.024
Revenant	**0.62**	**0.083**	0.60	0.004	0.54	−0.006	0.47	−0.012	0.41	−0.014
Mythic	**0.89**	**0.352**	0.82	0.021	0.63	0.203	0.54	0.128	0.51	0.053
Average	**0.82**	**0.149**	0.77	0.025	0.68	0.046	0.62	0.023	0.47	0.004

To evaluate generalization capability on unknown malware, Table 6 presents the performance comparison between ProRE and baseline methods across 5 unknown samples. These samples were selected from the public platform [25], with initial disclosure dates after June 2025. Sample types were identified using the commercial detection platform [26], and we manually analyzed the communication protocol message structures. Results demonstrate that ProRE maintains robust performance on unknown samples, achieving an average F1 score of 0.78 and cophenetic correlation coefficient of 0.127, outperforming all baseline methods by 6%–28% and 0.066–0.129, respectively. Even for complex protocols with nested encryption layers such as BlackCat, ProRE successfully identifies field boundaries through semantic understanding of data transformation operations. This indicates that ProRE generalizes effectively to unknown malware protocols. However, tupni shows the steepest performance decline on unknown samples (average F1 of 0.66), indicating poor adaptability to novel protocol patterns. netlifter's static analysis approach results in highly variable performance, from moderate success on structured protocols like 8Base (F1 of 0.73) to near failure on encrypted protocols like BlackCat (F1 of 0.52). The *QwQ-32B-preview* model exhibits the poorest generalization (average F1 of 0.50), suggesting that its training on general code corpora provides insufficient protocol-specific knowledge.

6.3 Ablation Study

We analyze the code slicing module, assembly language model, and clustering algorithm to evaluate their impact on overall performance.

Code Slicing Module

For code slice extraction, we evaluate through field structure recovery $F1$ scores on input data under different parameters. We set the current slice as S_0 and dynamic execution path as L_0. Define the following two operations. (1) Path extension (+): Merge the original path L_0 with its k-hop neighborhood basic blocks to form a new slice set S_{+k}. Here, k is the extension coefficient. (2) Path reduction (–): Remove basic blocks from the original path L_0 whose node indices are multiples of path length divided by k. Specifically, remove

nodes v_i satisfying $i \bmod \lceil \frac{|L_0|}{k} \rceil = 0$. Here, k is the reduction coefficient. The (+) operation extends the slice path; as k increases, the number of basic blocks increases. Its upper limit covers the entire control flow graph (CFG) traversed by L_0. Conversely, the (–) operation reduces the slice path; as k increases, the number of basic blocks decreases. When k equals path length $|L_0|$, no basic blocks remain. As shown in Fig. 7, after evaluating different extension/reduction coefficients $k \in \{-5, -4, -3, -2, -1, 0, +1, +2, +3, +4, +5\}$ on the benchmark dataset, we calculated the average $F1$ score of fields under each setting. Results show that the current slice configuration yields the highest average $F1$ score for field structure recovery.

Table 6: Performance comparison on unknown malware samples (best in bold)

	ProRE		BINPRE		TUPNI		NETLIFTER		*QwQ-32B-preview*	
	F1	***c***	***F1***	***c***	***F1***	***c***	***F1***	***c***	***F1***	***c***
BlackCat	**0.71**	**0.098**	0.64	0.042	0.58	0.021	0.52	−0.008	0.41	−0.032
LockBit	**0.82**	**0.152**	0.75	0.068	0.71	0.055	0.66	0.031	0.53	0.018
Akira	**0.85**	**0.171**	0.79	0.094	0.72	0.048	0.68	0.022	0.55	0.009
BlackBasta	**0.68**	**0.089**	0.61	0.015	0.54	−0.012	0.48	−0.025	0.38	−0.041
8Base	**0.84**	**0.124**	0.82	0.087	0.76	0.064	0.73	0.052	0.61	0.035
Average	**0.78**	**0.127**	0.72	0.061	0.66	0.035	0.61	0.014	0.50	−0.002

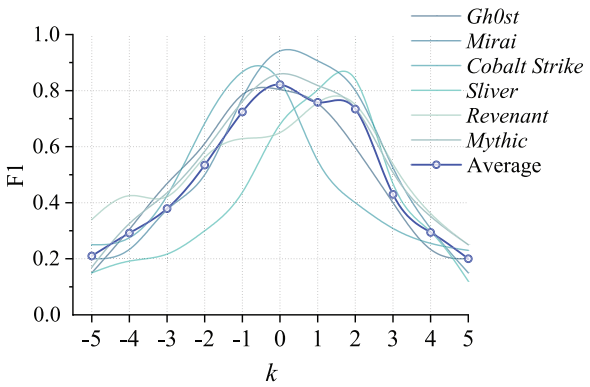

Figure 7: k-hop data flow slices

Assembly Language Model

To comprehensively evaluate current slice performance, we generate embedding vectors for instruction, basic block, and trace-level code, respectively. Results in Fig. 8 show that, ProRE assembly language model maintains relatively stable embedding quality across different granularities. Among them, basic block-level achieves higher average relatedness score of 0.88, indicating slice embeddings at this granularity are closer to actual results; trace-level has higher coherence score, reflecting better generalization capability in semantic expression for program execution records of certain length, but inferior to the other two granularities in relatedness and categorization; instruction-level has higher categorization score, reflecting more accurate and concentrated semantic expression.

To compare performance of different assembly language models, we selected PALMTREE [13], ASM2VEC [12], SAFE [11], and raw instruction sequences as instances of abstraction extraction function δ.

Specifically, PALMTREE abstracts semantics through data and control flow, ASM2VEC can handle code obfuscation problems, and SAFE performs well on programs including malware. For raw instruction sequences, we use method from BINPRE [4] to calculate similarity. Samples are grouped by execution path length: A (1–100), B (101–300), and C (301–700). Average F1 scores for each model are shown in Table 7 and Fig. 9.

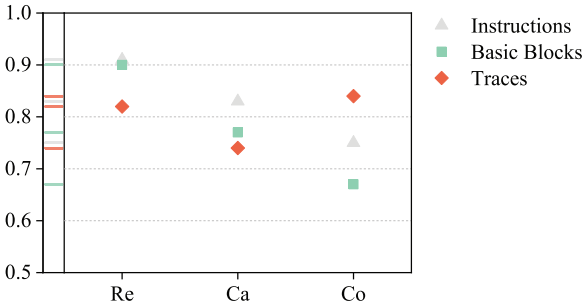

Figure 8: Assembly language model evaluation on ProRE

Table 7: Evaluation on assembly language models (best in bold)

Group	Sample quantity proportion	Average F1				
		PALMTREE	ASM2VEC	SAFE	Raw instruction	PRORE
A	47.54%	0.88	0.89	0.89	0.69	**0.91**
B	29.51%	0.78	0.79	0.74	0.66	**0.80**
C	22.95%	0.66	0.62	0.60	0.36	**0.69**

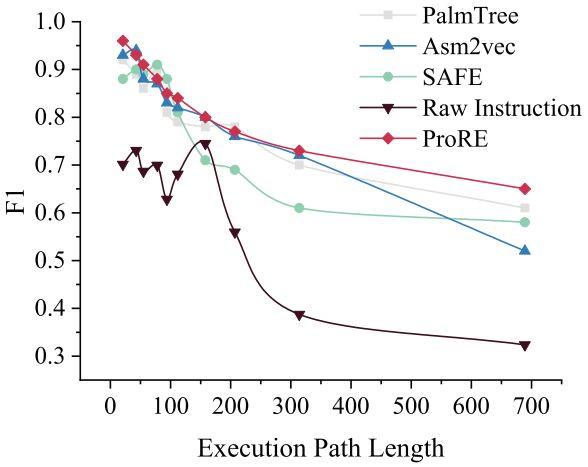

Figure 9: Comparison with other assembly language models

Results show that when protocol processing paths are short (not exceeding 100), F1 scores for field recovery are similar across different assembly language models. As path length increases, F1 scores gradually decrease, but PRORE still maintains the highest average F1 of 0.83. When path length exceeds 300, PRORE performs more stably, with average F1 scores 13% higher than other models. Average F1 scores for PALMTREE, SAFE, and ASM2VEC are 80.4%, 80.7%, and 78.6%, respectively. Their limitations are: (1) PALMTREE and SAFE

cannot handle library functions, losing partial semantic information. (2) ASM2VEC is limited by context window and cannot capture long-distance semantic relationships. Using raw instruction sequences lacks sufficient semantic abstraction capability, having the worst field structure recovery ability compared to assembly models, with F1 scores 19.3% lower on average.

Clustering Algorithm

To validate the effectiveness of hierarchical clustering in protocol field structure recovery, we compared it with various classical clustering algorithms, including Needleman-Wunsch [27], K-means, DBSCAN, and spectral clustering. We adapted these algorithms for protocol field recovery tasks by using byte-level slice embeddings as input.

The results in Table 8 demonstrate that hierarchical clustering provides significant advantages in protocol field recovery. While Needleman-Wunsch achieves reasonable F1 scores through sequence alignment, it cannot capture nested field relationships, resulting in significantly lower cophenetic correlation coefficients (0.087 vs. 0.229 on $\mathcal{D}_{com}$). K-means requires a predefined number of clusters, and even when using the actual field count, its structure recovery performance remains poor (c = 0.012 on $\mathcal{D}_{com}$). DBSCAN struggles with fields of varying density in the embedding space, while spectral clustering, though capable of capturing some nonlinear relationships, cannot preserve the inherent hierarchical structure of protocol messages. In contrast, hierarchical clustering achieves both accurate field boundary detection (higher $F1$) and structural relationship preservation (higher c), with its core advantage being the natural representation of multi-level field relationships through the clustering tree.

Table 8: Clustering algorithm comparison for field structure recovery (best in bold)

Algorithm	$\mathcal{D}_{com}$		$\mathcal{D}_{mal}$		Hierarchy recovery	Time (ms)
	$F1$	c	$F1$	c		
Hierarchical Clustering (Ours)	**0.89**	**0.229**	**0.82**	**0.149**	✓	124
Needleman-Wunsch	0.81	0.087	0.75	0.041	✗	892
K-means	0.73	0.012	0.68	−0.008	✗	67
DBSCAN	0.76	0.034	0.71	0.018	✗	103
Spectral Clustering	0.78	0.056	0.73	0.029	✗	215

Note: ✓: Supported; ✗: Not Supported.

6.4 Case Study

We select two representative samples to validate PRORE in real-world malware analysis.

6.4.1 Duke Steganographic Protocol

Duke [28] samples use LSB (Least Significant Bit) algorithm to extract steganographic data from bitmap images. Fig. 10 shows key code fragments of the sample processing image data: (a) shows regular bitmap data reading portion, (b) shows LSB steganographic data extraction portion. We generate code slice embeddings for each bit of the image and perform visual analysis. As shown in Fig. 11, specific positions in each group of three bytes, such as bits 7–8 of the first byte, bits 6–8 of the second byte, and bits 6–8 of the third byte, show obvious semantic differences from other bits. These positions exactly correspond to steganographic data bits extracted by the LSB algorithm (Fig. 12). Experimental results show that PRORE accurately identified execution semantic differences when malicious code processes different data types through slice embedding, recovering steganographic protocol structures.

```
ElementCount = v19 * v20;
v16 = 0;
j_memset(v17, 0, sizeof(v17));
v14 = 0;
j_memset(v15, 0, sizeof(v15));
fread(&v16, 3u, v19 * v20, Stream);
for ( i = 0; i < v20; ++i )
{
  for ( j = 0; j < v19; ++j )
  {
    sub_4113E8(&v16 + 0x300 * i + 3 * j, &v14 + 0x300 * i + 3 * j);
    v3 = &v14 + 0x300 * i + 3 * j;
    v4 = &v16 + 0x300 * i + 3 * j;
    *(_WORD *)v4 = *(_WORD *)v3;
    v4[2] = v3[2];
  }
}
```

(a)

```
v9 = j_unknown_libname_2(v19 * v20);
v8 = &v16;
v7 = 0;
for ( k = 0; k < v20; ++k )
{
  for ( l = 0; l < v19; ++l )
  {
    v6 = ((*v8 & 3) << 6) | ((v8[1] & 7) << 3 ) | v8[2] & 7;
    v8 += 3;
    *(_BYTE *)(v7 + v9) = v6;
    ++v7;
  }
}
```

(b)

Figure 10: *Duke* malware code fragments

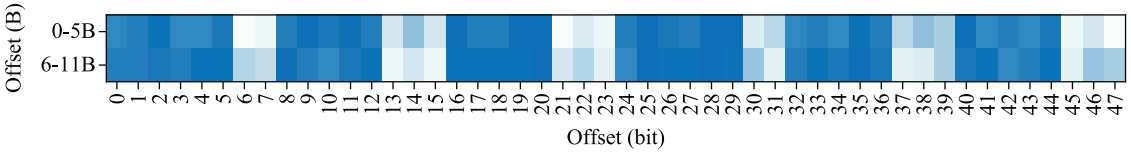

Figure 11: *Duke* code slice embedding visualization

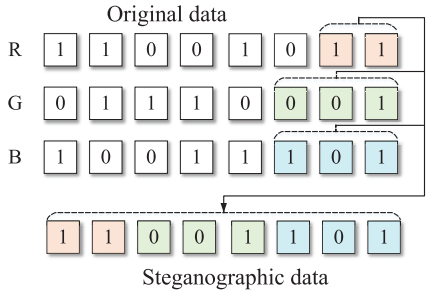

Figure 12: LSB algorithm process

6.4.2 *Mirai* Credential Download Protocol

Mirai [17] is a botnet program that implements covert attacks by infecting and controlling large numbers of IoT devices. A sample from this family (MD5: 3df80916a0d54cdf5eb3d476b4ae176d) has a credential download protocol containing 40 bytes of data payload. Fig. 13 shows the hexadecimal representation of the protocol message. We extract code slice embeddings corresponding to each byte (see Table 9) and apply hierarchical clustering algorithm to generate the field cluster tree. Analysis results show that the message has clear field boundaries at offsets 0–3, 4–5, 6–21, 22–37, and 38–39 bytes, completely matching the actual field structure (Fig. 14). Notably, fields 6–21, 22–37 and bytes 4–5, 38–39 are relatively close in semantic distance, corresponding to username and password data in credentials, respectively, demonstrating ProRE's ability to capture similarities in message processing logic.

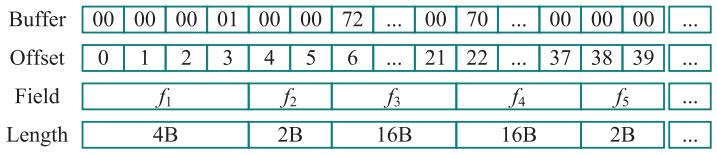

Figure 13: *Mirai* credential download protocol message structure

Table 9: *Mirai* credential download protocol slice representation

Message byte offset	Slice embedding	Mnemonic
0	[10.1, 11.2, −2.4, . . ., 5.7, −1.3, −8.4]	o_0
1	[10.6, 10.5, −3.1, . . ., 5.7, −1.2, −7.2]	o_1
2	[10.5, 11.1, −2.9, . . ., 3.5, −0.9, −7.1]	o_2
⋮	⋮	⋮
39	[−1.5, 12.3, 8.9, . . ., −0.9, −1.8, 10.1]	o_{39}

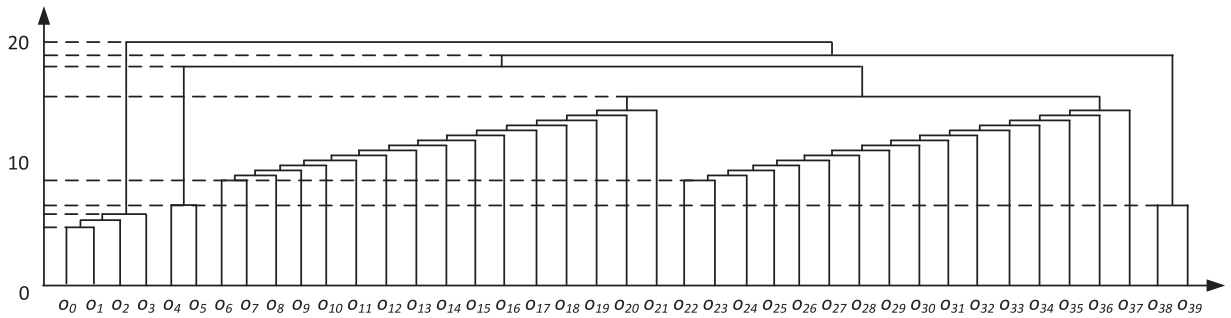

Figure 14: *Mirai* protocol field tree

7 Discussion

In this section, we discuss the time consumption and generalization ability of ProRE, encrypted message processing, and its limitations.

Time Consumption

ProRE's main computational overhead comes from: (1) dynamic execution trace collection, (2) code slice embedding computation, and (3) hierarchical clustering process. In our experiments, processing a 40-byte message takes an average of 3.2 s, with slice extraction accounting for 43.0%, embedding computation 35.6%, and clustering 21.4%. While this overhead is higher than simple static analysis methods, considering the significant improvement in analysis accuracy, this trade-off is acceptable. Additionally, ProRE adopts an online instrumentation execution plus offline data flow simulation design, separating necessary instrumentation code for tracking execution from data flow analysis code, reducing runtime analysis burden.

Generalization Capability

Our method has been validated on x86 architecture, but its core ideas can be extended to other architectures. The assembly language model can be adapted by retraining on the target instruction set, while slice extraction and hierarchical clustering algorithms are architecture-agnostic. Future work could explore cross-architecture transfer learning to reduce training costs on new architectures.

Encrypted Message

Most malware adopts encrypted protocols for communication. Similar to previous work [8,29], ProRE begins analysis from identified unencrypted message buffers to bypass the impact of encryption/decryption functions on data flow analysis accuracy. ProRE captures the data propagation path of raw message buffers from the network, and when standard cryptographic API calls exist in the path, it updates the analysis starting point f_0 to the pre-encryption or post-decryption address. We consider this approach feasible [30], as ProRE's inherent dynamic binary instrumentation framework supports this analysis.

Limitations

Despite ProRE achieving excellent performance, several limitations remain that we plan to address in future work: *First*, for protocols containing only single long fields (like *Sliver*), byte-level slice extraction may lead to over-segmentation. This can be mitigated by implementing adaptive granularity analysis that dynamically adjusts the slicing unit based on preliminary field length estimation. We are exploring multi-scale slicing approaches that combine byte, word, and block-level analysis. *Second*, when protocols use complex encryption or obfuscation techniques, execution slices may not accurately reflect true field boundaries. To address this, we plan to integrate symbolic execution techniques to reason about data transformations and develop encryption-aware slicing algorithms that can identify and handle cryptographic boundaries. *Third*, the current instrumentation engine [21] is limited by applicable architectures and platforms. We are developing a platform-agnostic intermediate representation layer that can abstract away architecture-specific details, enabling ProRE to support multiple binary analysis platforms [31–33] without significant modifications.

8 Conclusion

This paper proposes ProRE, a protocol message structure reconstruction method based on execution slice embedding. Addressing the shortcomings of existing methods in field boundary division and hierarchical relationship recovery, we design three key techniques: (1) execution slice extraction based on data flow dependencies to precisely capture protocol parsing processes; (2) a data flow-sensitive assembly language model to achieve high-quality vector representation of program semantics; (3) hierarchical clustering algorithm to completely recover protocol nested structures. Evaluation on a dataset containing 12 protocols shows that ProRE achieves an average F1 score of 0.85 and cophenetic correlation coefficient of 0.189, improving by 19% and 0.126 respectively over state-of-the-art baseline methods (including BINPRE, TUPNI, NETLIFTER,

and *QwQ-32B-preview*), demonstrating significant superiority in both accuracy and completeness of field structure recovery. Case studies further validate ProRE's effectiveness in practical malware analysis.

ProRE enables security analysts to rapidly understand unknown protocols in malware analysis, reducing analysis time from days to hours. The hierarchical structure recovery capability provides crucial insights for vulnerability assessment, as nested field relationships often indicate potential parsing vulnerabilities. Furthermore, the method's success on encrypted protocols like Duke demonstrates its applicability to modern malware that employs sophisticated evasion techniques. Organizations can integrate ProRE into their threat intelligence pipelines to automatically extract protocol specifications from captured malware samples, enhancing their defensive capabilities.

Several promising research directions emerge from this work. First, extending ProRE to handle stateful protocol analysis would enable complete protocol state machine recovery. Second, developing cross-architecture transfer learning techniques could reduce the training overhead when adapting to new processor architectures. Third, integrating ProRE with fuzzing frameworks could enable structure-aware protocol fuzzing for vulnerability discovery. Finally, investigating the use of large language models to generate human-readable protocol documentation from recovered structures could bridge the gap between automated analysis and human understanding.

Acknowledgement: Not applicable.

Funding Statement: The authors received no specific funding for this study.

Author Contributions: The authors confirm contribution to the paper as follows: Conceptualization, Hui Shu; methodology, Yuyao Huang; software, Yuyao Huang; validation, Yuyao Huang; investigation, Fei Kang; writing—original draft preparation, Yuyao Huang; writing—review and editing, Hui Shu, Fei Kang; supervision, Fei Kang; funding acquisition, Hui Shu. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: Available at: https://github.com/Mal-PRE/ProRE (accessed on 12 October 2025).

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest to report regarding the present study.

References

1. Wang S, Sun F, Zhang H, Zhan D, Li S, Wang J. EDSM-based binary protocol state machine reversing. Comput Mater Contin. 2021;69(3):3711–25. doi:10.32604/cmc.2021.016562.
2. Luo Z, Liang K, Zhao Y, Wu F, Yu J, Shi H, et al. DynPRE: protocol reverse engineering via dynamic inference. In: Network and Distributed System Security (NDSS) Symposium 2024; 2024 Feb 26–Mar 1; San Diego, CA, USA. p. 1–18.
3. Chandler J, Wick A, Fisher K. BinaryInferno: a semantic-driven approach to field inference for binary message formats. In: Network and Distributed System Security (NDSS) Symposium 2023; 2023 Feb 27–Mar 3; San Diego, CA, USA. p. 1–12.
4. Jiang J, Zhang X, Wan C, Chen H, Sun H, Su T. BinPRE: enhancing field inference in binary analysis based protocol reverse engineering. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security; 2024 Oct 14–18; Salt Lake City, UT, USA. p. 3689–703.
5. Shi J, Wang Z, Feng Z, Lan Y, Qin S, You W, et al. AIFORE: smart fuzzing based on automatic input format reverse engineering. In: 32nd USENIX Security Symposium (USENIX Security 23); 2023 Aug 9–11; Anaheim, CA, USA. p. 4967–84.

6. Lin Z, Jiang X, Xu D, Zhang X. Automatic protocol format reverse engineering through context-aware monitored execution. In: Network and Distributed System Security (NDSS) Symposium 2008; 2008 Feb 10–13; San Diego, CA, USA. Vol. 8, p. 1–15.
7. Cui W, Peinado M, Chen K, Wang HJ, Irun-Briz L. Tupni: automatic reverse engineering of input formats. In: Proceedings of the 15th ACM Conference on Computer and Communications Security; 2008 Oct 27–31; Alexandria VA, USA. p. 391–402.
8. Caballero J, Poosankam P, Kreibich C, Song D. Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In: Proceedings of the 16th ACM Conference on Computer and Communications Security; 2009 Nov 9–13; Chicago IL, USA. p. 621–34.
9. Raff E, Barker J, Sylvester J, Brandon R, Catanzaro B, Nicholas CK. Malware detection by eating a whole EXE. In: The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, 2018 Feb 2–7; New Orleans, LA, USA. Washington, DC, USA: AAAI Press; 2018. p. 268–76.
10. Guo W, Mu D, Xing X, Du M, Song D. DEEPVSA: facilitating value-set analysis with deep learning for postmortem program analysis. In: 28th USENIX Security Symposium (USENIX Security 19); Santa Clara, CA, USA: USENIX Association; 2019. p. 1787–804.
11. Massarelli L, Di Luna GA, Petroni F, Baldoni R, Querzoni L. Safe: self-attentive function embeddings for binary similarity. In: Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019; Cham, Switzerland: Springer; 2019. p. 309–29.
12. Ding SH, Fung BC, Charland P. Asm2vec: boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: 2019 IEEE Symposium on Security And Privacy (sp); 2019 May 19–23; San Francisco, CA, USA. p. 472–89.
13. Li X, Qu Y, Yin H. Palmtree: Learning an assembly language model for instruction embedding. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security; 2021 Nov 15–19; New York, NY, USA: Association for Computing Machinery. p. 3236–51.
14. Wang H, Qu W, Katz G, Zhu W, Gao Z, Qiu H, et al. Jtrans: jump-aware transformer for binary code similarity detection. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis; 2022 Jul 18–22; New York, NY, USA: Association for Computing Machinery. p. 1–13.
15. Huang Y, Shu H, Kang F, Guang Y. Protocol reverse-engineering methods and tools: a survey. Comput Commun. 2022;182:238–54.
16. Kleber S, Maile L, Kargl F. Survey of protocol reverse engineering algorithms: decomposition of tools for static traffic analysis. IEEE Commun Surv Tutor. 2019;21(1):526–61. doi:10.1109/comst.2018.2867544.
17. Antonakakis M, April T, Bailey M, Bernhard M, Bursztein E, Cochran J, et al. Understanding the mirai botnet. In: 26th USENIX Security Symposium; 2017 Aug 16–18; Vancouver, BC, Canada: USENIX Association. p. 1093–110.
18. Shi Q, Shao J, Ye Y, Zheng M, Zhang X. Lifting network protocol implementation to precise format specification with security applications. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security; 2023 Nov 26–30; Copenhagen, Denmark. p. 1287–301.
19. Devlin J, Chang MW, Lee K, Toutanova K. Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers); Rock Hill, CA, USA: ACL; 2019. p. 4171–86.
20. Hex-Rays. IDA Pro: a powerful disassembler, decompiler and a versatile debugger; 2025 [Internet]. [cited 2025 Jan 27]. Available from: https://hex-rays.com/ida-pro.
21. Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, et al. Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation; 2005 Jun 12–15; Chicago, IL, USA. p. 190–200.
22. Team Q. QwQ: reflect deeply on the boundaries of the unknown; 2024 [Internet]. [cited 2025 Jan 27]. Available from: https://qwenlm.github.io/blog/qwq-32b-preview/.

23. Schnabel T, Labutov I, Mimno D, Joachims T. Evaluation methods for unsupervised word embeddings. In: Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing; 2015 Sep 17–21; Lisbon, Portugal. p. 298–307.
24. Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, et al. SciPy 1.0: fundamental algorithms for scientific computing in python. Nature Meth. 2020;17:261–72. doi:10.1038/s41592-020-0772-5.
25. VirusShare. VirusShare.com is a repository of malware samples to provide security researchers, incident responders, forensic analysts, and the morbidly curious access to samples of live malicious code; 2025 [Internet]. [cited 2025 Aug 21]. Available from: https://virusshare.com.
26. VirusTotal. Analyse suspicious files, domains, IPs and URLs to detect malware and other breaches, automatically share them with the security community; 2025 [Internet]. [cited 2025 Aug 21]. Available from: https://www.virustotal.com.
27. Likic V. The Needleman-Wunsch algorithm for sequence alignment. Lecture given at the 7th Melbourne Bioinformatics Course. Australia: Bi021 Molecular Science and Biotechnology Institute, University of Melbourne; 2008.
28. 42 PANU. Report title about duke malware; 2025 [Internet]. [cited 2025 Aug 6]. Available from: https://unit42.paloaltonetworks.com/tag/duke-malware/.
29. Wang Z, Jiang X, Cui W, Wang X, Grace M. Automatic reverse engineering of encrypted messages. In: Computer Security ESORICS 2009: 14th European Symposium on Research in Computer Security; 2009 Sep 21–23; Saint-Malo, France. Cham, Switzerland: Springer; 2009. p. 200–15.
30. Meijer C, Moonsamy V, Wetzels J. Where's Crypto?: automated identification and classification of proprietary cryptographic primitives in binary code. In: 30th USENIX Security Symposium (USENIX Security 21); 2021 Aug 11–13; Vancouver, BC, Canada. p. 555–72.
31. DynamoRIO. Dynamic instrumentation tool platform; 2025 [Internet]. [cited 2025 Jan 27]. Available from: https://dynamorio.org/.
32. Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM SIGPLAN Notices. 2007;42(6):89–100.
33. Triton. Triton is a dynamic binary analysis library; 2025 [Internet]. [cited 2025 Jan 27]. Available from: https://triton-library.github.io/.