BinStruct: Binary Structure Recovery Combining Static Analysis and Semantics

Yiran Zhang*, Zhengzi Xu†§, Zhe Lang‡, Chengyue Liu*, Yuqiang Sun*, Wenbo Guo*, Chengwei Liu*¶, Weisong Sun* and Yang Liu*¶
*College of Computing and Data Science, Nanyang Technological University, Singapore
yiran002@e.ntu.edu.sg, chengyue001@e.ntu.edu.sg, suny0056@e.ntu.edu.sg, honywenair@gmail.com, chengwei.liu@ntu.edu.sg, weisong.sun@ntu.edu.sg, yangliu@ntu.edu.sg
†Imperial Global Singapore, Imperial College London, Singapore
z.xu@imperial.ac.uk
‡Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
langzhe@iie.ac.cn
¶China-Singapore International Joint Research Institute, Guangzhou, China

***Abstract*—Binary reverse engineering is foundational to various tasks such as malware analysis and vulnerability detection. Traditional binary analysis tools mainly operate at the function level. However, modern software has grown significantly in size, with binaries often containing thousands of functions. Without understanding how these functions are organized into higher-level structures, it becomes difficult to effectively support downstream analysis tasks. Analysts must examine thousands of functions separately, making the process time-consuming and error-prone. Despite these challenges, current research on recovering the higher-level structure of binaries remains limited.**

To bridge this gap, we propose BinStruct, a novel binary structure recovery framework that recovers both file and module structures from binaries. BinStruct first identifies the file structure by combining data reference patterns, function calls, and semantic understanding from Large Language Models. Then, inspired by software architecture recovery in source code analysis, BinStruct identifies modules by clustering the recovered files using consensus between structural dependency and semantic similarity. Evaluation on 121 real-world stripped binaries demonstrates that BinStruct outperforms state-of-the-art techniques in both file and module recovery accuracy, while requiring only 7.42s and 34.46s on average to recover file and module structures, respectively. Case studies on Libxml2 and PredatorTheStealer demonstrate BinStruct's effectiveness on security tasks like attack surface analysis and malware investigation.

***Index Terms*—Reverse Engineering, Binary Analysis, Program Comprehension**

I. Introduction

Understanding binary programs is a fundamental challenge in reverse engineering, underpinning critical tasks such as malware analysis [1], third-party library identification [2], [3], and vulnerability detection [4]–[8]. Traditional binary analysis tools focus primarily on individual function recovery [9] and identification of function or variable names [10]–[12]. However, these function-level tools fall short when analyzing modern software systems, which are composed of numerous interdependent components. In such large binaries, analysts are often left with the daunting task of inspecting thousands of disassembled functions in isolation, lacking the higher-level context necessary for scalable and effective comprehension.

In contrast, research on the source code level often addresses software comprehension challenges through Software Architecture Recovery (SAR). These techniques uncover module-level organization, where each module encompasses multiple related source files and represents a distinct functionality within the system [13]–[16]. Recent work has demonstrated that semantic understanding significantly improves the granularity and quality of recovered modules [17]. By revealing inter-module relationships and architectural boundaries, SAR tools help developers maintain and navigate large codebases more efficiently.

Despite the success of SAR in source code, its application to binary analysis is less explored. Only a few works have explored structure recovery beyond the function level. For instance, DeLink [18] attempts to reconstruct source file boundaries using structural features. However, it stops at the file level and does not capture higher-level abstractions such as modules, making the recovered structure still difficult to interpret. Furthermore, DeLink relies solely on structural features and coarse-grained similarity metrics, which limit its accuracy and semantic fidelity. BCD [19] proposes identifying modules in binaries by clustering functions based on caller–callee dependencies. However, the resulting clusters often closely resemble individual files rather than true architectural modules. In addition, BCD treats each function independently, without considering file boundaries, often assigning adjacent functions that originate from the same source file to different modules. This violates the architectural principle that modules should be composed of coherent source files, resulting in fragmented and semantically ambiguous groupings. Modx [2] builds on the observation that functions from the same source file are often placed together in binaries, a principle later supported by empirical findings from [20]. It incorporates spatial locality into the clustering process. However, this spatial information is only measured as raw function distance in the binary, contributing minimally to final clustering results.

§Zhengzi Xu is the corresponding author

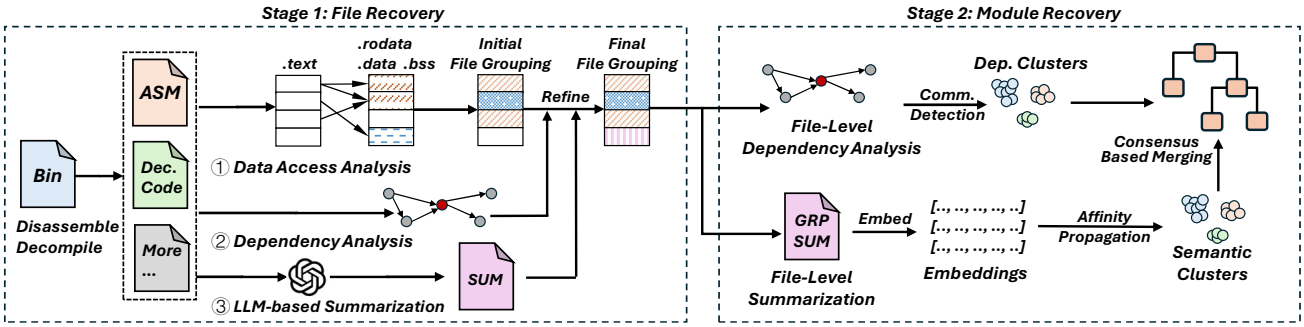


Fig. 1. Overview of our approach

Consequently, ModX also tends to produce groupings that align more with file-level structures rather than architectural modules. ChatCPS [21] similarly attempts to recover binary structure, but it adopts the same clustering algorithm as BCD and therefore inherits its limitations.

These limitations can be attributed to two fundamental challenges in binary architectural recovery. First **C1**, stripped binaries contain no explicit indicators of source file boundaries, making structural inference difficult. Second **C2**, most semantic information, such as comments and function names, is lost during compilation and stripping, removing critical cues for understanding program intent.

To address these challenges and recover more human-understandable structures from binaries, we propose BinStruct, an LLM-enhanced framework for recovering both file and module structure from stripped binary files. BinStruct first reconstructs the source file structure by analyzing data references and function call relationships. It begins by forming initial groupings based on shared data access patterns, expands these groups using call graph information, and then refines boundaries using a LLM. The LLM applies a bottom-up summarization strategy to infer semantic similarities that cannot be captured through structure alone. After file recovery, BinStruct performs module-level reconstruction by clustering the recovered files based on both structural dependencies and semantic similarity. These two clustering results are merged using a consensus-based strategy. An LLM is then used to resolve ambiguous cases, resulting in final module groupings that better reflect the software's original architectural design. Experimental results show that BinStruct outperforms existing binary recovery tools in both file and module reconstruction, with average improvements of 5.38%-17.09% and 5.1-25.7 percentage points, respectively. Remarkably, BinStruct achieves performance comparable to or even better than source-level SAR tools, despite operating on stripped binaries with significantly less available information. Then, an ablation study reveals that each component of file recovery and module recovery can effectively contribute to the final performance. Moreover, a cost analysis shows that BinStruct can recover file and module structures in 7.42 and 34.46 seconds respectively, with an average cost of only 0.167 USD per binary on average when using GPT-4.1-nano. Finally, two case studies on real-world binaries demonstrate BinStruct's effectiveness in security-related tasks, such as attack surface detection and malware analysis. Our key contributions include:

- We propose a novel framework, BinStruct, that integrates structural analysis with LLM-guided refinement to recover file and module structure from stripped binaries.
- We develop a comprehensive binary structure recovery approach that recovers files by combining data reference patterns, function calls, and LLM's semantics.
- We evaluate BinStruct on 121 real-world binaries and show that it outperforms existing binary structure recovery tools on both file and module recovery.
- We demonstrate BinStruct's potential practical value through case studies on attack surface detection and malware analysis.

II. Methodology

The overview of our methodology is shown in Figure 1. Our approach consists of two main parts. In the first part, we recover the file structure through three steps. First, we create initial groups of functions based on data reference patterns. Second, we expand these groups based on function similarity measured by function calls. Finally, we apply LLMs to refine the group structure based on its semantic understanding to form the final recovered file structure. In the second part, we first recover the module structure within the binary using two clustering methods: structural dependency analysis and semantic similarity analysis. We then combine these two clustering results based on their consensus and use LLMs to resolve conflicts, which forms the final module structure.

A. File Recovery

The source file structure recovery process mainly consists of 3 parts. In the first part, we use the data reference patterns to identify initial groups of functions that are from the same source file. In the second part, we use the function call graph to expand the grouping. Finally, we use LLM to finalize the source file recovery by resolving the ambiguous functions based on previous structural features.

Before introducing the recovery algorithm, we will first introduce the typical compilation process of x86 binaries to facilitate the understanding of function and data locality.

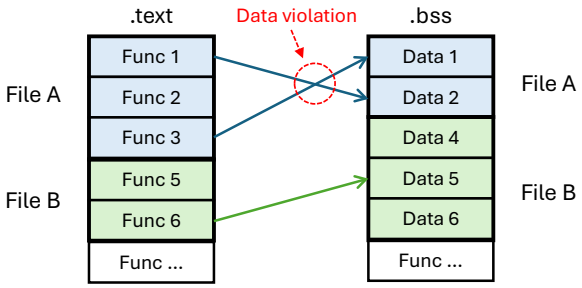
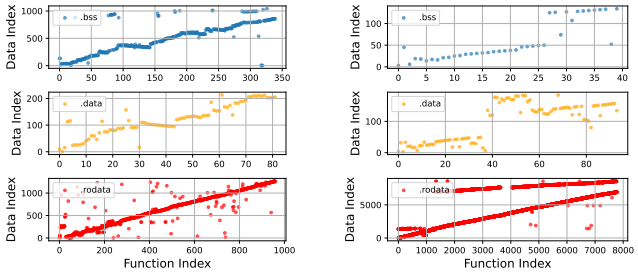


Fig. 2. Illustration of file locality and data reference violation



(a) Bash (b) Sqlite3

Fig. 3. The relation between the function index and the referenced data index.

1) x86 Binaries and File Locality

An executable file on x86 platforms is assembled with several sections, each storing different types of data [22]. Executable code is stored in the `.text` section, initialized global read-write variables are stored in the `.data` section, and uninitialized global variables are stored in the `.bss` section. The `.rodata` section stores read-only data, such as string literals and constant variables. When a program is compiled, the linker merges identically named sections from all input object files. The merging follows the order in which files are passed to the linker, unless overridden by a linker script. This process produces the final layout in the executable file. Importantly, by default, the linker will try to keep functions and global variables that are defined in the same source file close to each other in the final executable [23].

Therefore, functions and data in binaries are typically organized as illustrated in Fig. 2. First, functions or data from the same source file are placed together within their respective sections. Second, the file ordering is consistent across all sections, i.e. `.text` section and the three data sections (`.data`, `.bss`, and `.rodata`).

2) Initial Grouping with Data Reference

In this section, we first present our core idea by temporarily making two strong assumptions. Then, we explain how to apply this idea to real-world scenario when these two assumptions do not apply.

Core Idea. To help understand our core idea, we first assume that 1) the file order of the *.text* segment (i.e., the segment that contains user-defined functions) and data segments (i.e., the *.data*, *.bss* and *.rodata* segments) are the same and 2) a function only accesses data from its own file. Under this assumption, if a function f_1 reference data d_1 and f_2 reference data d_2, and we have $addr(f_1) < addr(f_2)$ but $addr(d_1) \geq addr(d_2)$ (or vice versa), then f_1 and f_2 must come from the same source file. To explain this, we show an example in Fig. 2. Consider two functions from different files: f_1 from File A and f_5 from File B. Under our assumptions, f_1 can only access data from File A, and f_5 can only access data from File B. Since File A comes before File B, we expect f_1 to have a smaller address than f_5. Meanwhile, data from File A also have a smaller address. This means that for functions from different files, their address order should match the address order of their accessed data. However, this does not hold for function from the same file. Therefore, if the function address order is different from the data address order (which we will refer as a *data violation* in the following paper), then the two functions must come from the same source file.

However, these two assumptions do not always hold in practice. Next, we explain how we handle the binaries without making these strong assumptions.

On the File Order. Based on this observation, we now need to revisit the two assumptions. For the first assumption that the file order of *.text* and data segments are the same, it is the default behavior as aforementioned, which is held for most of the real-world binaries. And, for those does not follow this assumption, we can detect them and refuse the corresponding data section.

To Illustrate the detection, we show two examples in Fig. 3. We scattered all the data reference for two projects. For each point, its x-axis shows the index of functions (in *.text*) with access to the regarding data segment, and y-axis shows the index of data, both sorted by the address. Fig. 3(a) shows the common scenario: the function and data indexes are highly correlated and forms a line, with a small portion of cross-file data scattered around. In this case, the spearman correlation between the function indexes and their accessed data indexes are very high (0.804-0.932 for *bash*). And for Sqlite3 in Fig. 3, where the data reference on *.rodata* does not follow such pattern, the correlation is much lower (0.359). Therefore, before using data reference to measure the similarity, we first measure the correlation between the functions index and their referenced data indexes. We will only adopt the data reference to measure the similarity if the correlation is higher than 0.5.

On the Cross File Data Access. Our second assumption is that functions only access data within their own source file. Since this assumption is not realistic, we address it in two steps. First, we filter out some cross-file data references. Second, we present our recovery method that works even if cross-file data reference exists.

First, we focus on filtering cross-file data references. As shown in Figure 3(a), most data access points lie near the diagonal line, which means functions access data close to their positions. Points far from this line indicate that functions are accessing data far away from them, which likely represent cross-file data references. Therefore, we can eliminate these outliers to filter obvious cross-file data references.

To filter these outliers, we first use linear regression to find the best-fit line. We then apply the interquartile range (IQR) method [24] to detect outliers. We compute the first quartile (Q1) and third quartile (Q3) of the residuals. The IQR is defined as IQR $= Q3 - Q1$. Any data point with a residual larger than $Q3{+}1.5{\times}$IQR is marked as an outlier and removed.

In theory, removing all cross-file data accesses would allow us to group functions by data violations easily. However, since all debug information is unavailable, stripped binary files provide no direct way to distinguish cross-file from within-file data accesses. This makes complete removal of cross-file accesses nearly impossible, even after filtering outliers. To address this challenge, we do not assume that functions with data violations always belong to the same file. Instead, we view such violations as strong hints that the two functions are likely originated from the same source file. In this way, even if cross-file data accesses still exist, their influence are significantly reduced.

Then the task can be modeled as a graph problem where functions are nodes and edges connect function pairs with data violations. Each edge suggests these functions probably came from the same file, and we need to identify the best partition of the functions. This transforms our task into community detection, which identifies the best partitioning result with more intra-group edges and fewer inter-group edges. For community detection, the Girvan-Newman modularity [25] is a standard metric for assessing clustering quality, which is defined as:

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta\left(C_i, C_j\right) \tag{1}$$

where A_{ij} is the adjacency matrix element, k_i is the degree of node i, m is the total number of edges in the graph, and $\delta(C_i, C_j)$ is 1 only if nodes i and j are in the same community.

However, our problem still has some differences from typical community detection. Functions from the same source file appear consecutively in the binary. Therefore, recovery means finding file boundaries rather than arbitrary function clusters. Standard community detection algorithms ignore this spatial constraint and can produce suboptimal results. Therefore, instead of using standard community detection algorithms, we developed a dynamic programming (DP) algorithm to find boundaries that can maximize the Girvan-Newman modularity.

In our DP algorithm, $dp[i]$ represents the highest modularity when partitioning the first i functions. We build the DP table as follows. For each position i, we test all possible starting points k for the final segment. This segment includes functions from position k to position $i-1$. We compute each segment's modularity contribution $\Delta Q(k, i-1)$. The recurrence relation becomes $dp[i] = \max_k(dp[k] + \Delta Q(k, i-1))$. Since source files contain limited functions, to improve efficiency, we cap the maximum segment length at 20% of total functions in BinStruct. After building the DP table, we trace back through stored split points to find the optimal partition.

This DP algorithm allows us to identify the optimized file boundaries based on data reference similarity. However, due to possibly limited data references, many functions cannot be grouped with this algorithm. To include these functions, we will next extend the grouping using function call relationships.

3) Expand Recovery Result Based on Function Call

As function calls happen more commonly inside the same file than across files [18], they can also be used as a similarity measure between functions. Though the accuracy of this function call measure is usually lower than the data violation, the advantage of function calls is that they can usually cover nearly all functions. Instead of treating all function calls equally, due to the locality characteristic of files, intuitively, the farther the caller and callee, the lower the possibility for the function call to be in the same file. Therefore, we assign the following weight for the function call:

$$W(f_{src}, f_{dst}) = \frac{1}{distance(f_{src}, f_{dst})} \tag{2}$$

Then we use the same DP algorithm as detailed previously to identify the file boundaries that can maximize the modularity. The only difference is that the function groups identified by data references are treated as pre-defined groups, which means the functions in the same groups are treated as a single item and will not be separated during the DP process.

4) Final Refinement with LLMs

Our method based on data references and function calls can identify file boundaries, but it sometimes makes mistakes. The main problem occurs with functions near file boundaries. When two functions from different but adjacent files are located close to each other and have dependencies or data relations, our method may incorrectly group them into the same file.

To address these boundaries that are hard to determine using only structural information, we use semantic understanding to resolve them. First, we summarize these binary functions to understand their semantic meaning. Then, based on this understanding, we decide the final boundaries using both the function summaries and the structural relations of the boundary functions. Since this decision requires strong semantic understanding to process multiple information sources, we use LLMs for this task rather than smaller models.

Bottom-up Summarization. To help the LLM understand each function better, we provide it with context about the functions that the target function calls. To this end, we use a bottom-up approach to summarize functions in order. First, we identify functions that do not call any other functions in our analysis scope. We summarize these leaf functions first since they have no dependencies. Then, we move up the call hierarchy until all functions are summarized. Due to space limitations, we refer readers to our website [26] for the detailed prompts used in this process.

Adjustment with LLM. We use the recovered semantic information to fix minor grouping mistakes around file boundaries. To reduce potential LLM hallucination, we first identify suspicious functions that may be incorrectly grouped. For

all functions located at cluster boundaries, we check they have structural relations (i.e. function calls or data violations) with both their current cluster and the adjacent cluster. If a boundary function has connections to both clusters, we mark it as suspicious. For each suspicious function, we provide the LLM with: 1) the function's own summary and decompiled code, and 2) summaries of functions in both its current cluster and the adjacent cluster. We then ask the LLM whether the function is more related to one cluster than the other. To further reduce hallucination risk, we ask the same question twice with different random seeds. Only if both responses confirm that the function belongs better with the adjacent cluster do we move it. We repeat this process for all suspicious boundary functions until all resolved.

B. Module Recovery

After recovering the file structure, we aim to identify the module structure of the binary. Module recovery presents different challenges than file recovery. Unlike functions within a file, files from the same module are not necessarily placed together in the binary. This means we cannot use data access patterns and locality features that worked for file recovery. However, module recovery has an advantage. Since each file contains many functions, we can extract more reliable semantic information from files than from individual functions, which can serve as a feature for clustering.

Previous studies on source code [27], [28] show that both structural dependencies between files and semantic similarity can identify modules effectively. Since both approaches have proven useful but perform differently depending on the project, we use a consensus-based strategy. We first apply structural and semantic clustering independently, then combine their results to determine the final module structure.

1) Clustering with Structural Dependencies

To cluster files based on structural dependencies, we first convert the function call graph into a file dependency graph. We do this by merging all functions that belong to the same recovered file. A simple approach would be to set the edge weight between two files as the number of function calls between them. However, as reported in [17], different function calls have different levels of architectural importance. Therefore, we follow their setup to weight function calls according to Inverse PageRank (IPR) [29].

IPR is a variant of PageRank [30] that measures the importance of each function. Unlike PageRank, IPR is designed to find influential nodes that can access many other nodes, which can be more important for the architectural modules. IPR is calculated as:

$$IPR(f_i) = d \sum_{f_j \in P(f_i)} \frac{IPR(f_j)}{in\_degree(f_j)} + \frac{1-d}{N} \tag{3}$$

where $f_1 \sim f_N$ are the functions, N is the total number of functions, $P(f_i)$ are the direct successors of f_i, and d is the damping factor set to default value 0.85.

We define the weight of each function call as the product of the IPR values of its source and destination functions. The dependency weight between two files is then the sum of all function call weights between them:

$$W(F_a, F_b) = \sum_{\substack{f_i \in F_a, f_j \in F_b \\ (f_i, f_j) \in E}} (IPR(f_i) \times IPR(f_j)) \tag{4}$$

where F_a, F_b are files, f_i, f_j are functions, and E represents the function calls. Finally, we apply the Louvain community detection algorithm [31] with default parameters to cluster the files based on this weighted dependency graph, as it efficiently handles weighted graphs and automatically determines the optimal number of clusters.

2) Clustering with Semantic Features

To measure semantic similarity between files, we first use an LLM to generate a summary for each file based on its function summaries. We then convert these file summaries into embeddings to enable similarity computation. According to a recent comparative study [32], the all-MiniLM-L6-v2 model from Sentence Transformers [33] outperforms other embedding models. Therefore, we use this model to generate sentence embeddings. We then create a similarity matrix for the recovered files based on the cosine similarity between their embedding vectors.

To cluster files using this similarity matrix, we apply the affinity propagation algorithm [34], which automatically determines the optimal number of clusters without requiring this parameter to be predefined.

C. Merging Clustering Results

To merge the structural and semantic clustering results, we use a consensus-based approach. We first identify file groups that belong to the same cluster in both methods by building a consensus graph. The nodes represent files, and we add edges between file pairs that are clustered together by both methods. We then find connected components in this graph to identify consensus groups.

For files not in consensus groups, we process them in two phases. First, we handle files that have partial consensus (same cluster in one method but different clusters in the other method). For each such file, we use an LLM to decide its placement based on similarity to existing consensus groups. We provide the LLM with: 1) the summary of the candidate file, 2) summaries of existing consensus groups, and 3) function call relations between the file and existing groups.

We guide the LLM through a structured decision process: first, compare the file with existing groups from both semantic and structural perspectives; second, identify if the file has significantly stronger similarity to one or several groups; third, if yes, choose the most similar group, otherwise create a new standalone group if the file has a clearly distinct purpose, or assign it to the most relevant existing group.

After each file placement, we regenerate the summary of any updated group to maintain accuracy. We refer readers to our website [26] for detailed prompts.

Finally, we process remaining files without any consensus using the same LLM-based approach. The process concludes

when all files are assigned to clusters, forming the final module structure.

III. Evaluation

A. Research Questions

In this section, we aim the answer the following RQs.
RQ1: What is the quality of the recovered file structure?
RQ2: What is the quality of the recovered modules structure?
RQ3: How each part of our design contributes to BinStruct?
RQ4: What is the time and token cost of BinStruct?
RQ5: What is the real-world application of BinStruct?

B. Datasets

We collected data from two types of sources. First, we reused datasets from previous studies, including 105 binaries from ModX [2] and 14 binaries from BCD [19], resulting in a total of 119 binaries. Second, we adopted the dataset used in a recent study [17], which includes nine software projects. However, seven of these projects were either Java-based, where the compiled output is not comparable to traditional native binaries, or could not be successfully compiled. Therefore, we included only two projects: Libxml2 (version 2.4.22) and Bash (version 4.2), each successfully compiled into a binary. In total, our evaluation dataset consists of 121 binaries.

C. Implementation Details

We utilize IDA Pro [35] to disassemble the binaries and extract the necessary features. For the language model, we use gpt-4.1-nano-2025-04-14 [36], a cost-efficient yet effective model. All binaries used in the evaluation are compiled using GCC 7.5 [37] with optimization level O2, targeting the x86-64 architecture. Each binary is initially compiled with the -g flag to include debug information, which is used to extract ground-truth data for evaluation. The binaries are then stripped using the strip command to remove the debug information. All experiments, whether conducted using BinStruct or baseline methods, are performed on the stripped binaries.

D. RQ1: Accuracy of File Recovery Evaluation

1) Baseline Selections

First, we include DeLink [18], which, to the best of our knowledge, represents the state-of-the-art in source file structure recovery. Following the evaluation setup of DeLink, we also include two binary modularization tools, BCD [19] and ModX [2], as baselines.

2) Evaluation Metrics

First, we follow the setup of [18] to use the Jaccard Coefficient (JC) [38], Fowlkes and Mallows Index (FMI) [39], and Rand Index (RI) [40], which are defined as:

$$JC = \frac{a}{a+b+c}, FMI = \sqrt{\frac{a}{a+b} \cdot \frac{a}{a+c}}, RI = \frac{a+d}{a+b+c+d} \tag{5}$$

where a is the number of function pairs that are from the same source file and correctly grouped into the same file, b is the number of function pairs that are from the same source file but incorrectly grouped into different files, c is the number of function pairs that are from different source files but incorrectly grouped into the same file, d is the number of function pairs that are from different source files and correctly grouped into different files. These three metrics are widely used metrics in clustering problems. However, RI usually has a limited dynamic range, so we also adopted its improved version Adjusted Rand Index (ARI) [41], which is defined as:

$$ARI = \frac{RI - E(RI)}{1 - E(RI)} \tag{6}$$

where $E(RI)$ is the expected value of the Rand Index with random inputs.

TABLE I
Accuracies for File Recovery Results

Technique	JC	FMI	RI	ARI
ModX	20.32	31.81	78.40	18.55
BCD	40.06	56.90	83.73	46.22
DeLink	48.72	64.28	86.31	54.92
BinStruct	56.67	71.46	90.94	64.31
Impv.	16.31%	11.17%	5.38%	17.09%

3) File Recovery Results and Discussions

The accuracy of BinStruct and the baselines is presented in Table I. As shown, BinStruct achieves the best performance across all evaluation metrics. It outperforms the strongest baseline by 5.35% to 17.09% in accuracy, depending on the metric used. Among the baselines, DeLink yields the most competitive results, while BCD and ModX perform worse.

The primary limitation of BCD and ModX is that they are not designed to recover source file structures and therefore do not enforce the locality constraint of functions. In both BinStruct and DeLink, a source file is assumed to consist of a contiguous block of functions. In contrast, BCD and ModX generate groupings without this constraint, often clustering together functions that clearly originate from different files. This significantly reduces their accuracy in file structure recovery. DeLink's main limitation lies in its handling of data references. It only considers functions accessing the same data as similar, and overlooks data reference violations, as discussed in Section II-A, which can also serve as strong indicators of file-level similarity. This omission contributes to its lower performance.

Although BinStruct achieves higher scores than all baseline techniques, it is not without error. We identify two major types of mistakes. First, in cases where two adjacent source files are highly similar in both structural and semantic aspects, BinStruct sometimes merges them into a single group. However, combining such closely related files has limited negative impact on code understanding. Second, for very large source files containing hundreds of functions, BinStruct often splits them into multiple groups. This occurs because these files tend to contain several function clusters that are independently cohesive in both structure and semantics. As a result, BinStruct occasionally over-segments these large files.

TABLE II
WELCH'S T-TEST BETWEEN PROJECTS WITH FILE ORDERING PRESERVED OR NOT

	JC	FMI	RI	ARI
Mean (preserved)	0.5646	0.7156	0.9104	0.6419
Mean (not preserved)	0.5919	0.7010	0.8959	0.6584
P-value	0.3712	0.3405	0.8335	0.5229

Answering RQ1: BinStruct outperforms the state-of-the-art file structure recovery techniques by 5.38%-17.09% on average across 4 metrics.

4) Reliance on the Two Assumptions

As described in Section II-A2, in the first step of our file recovery (i.e., recovering file based on the data reference), our methodology is designed to address two temporary: 1) functions and data appear in the same file order in binaries; 2) functions may access data from other files. Though our methodology works even when these assumptions do not hold, it is still important to investigate two questions: 1) How often do these assumptions hold in our dataset? 2) How does the accuracy change when these assumptions do not hold?

To conduct this investigation, we first need to identify the original source file for each function and data element. For functions, this information can be obtained from debug information before stripping. However, for data elements, the binary provides no source file information even with debug information available. To solve this problem, we examined all strings in the binary and identified the source file for strings that appear in only one file. Since determining the source location of remaining data elements is nearly impossible, we base our analysis on these strings with known source files.

We first tested the assumption that file ordering is preserved for both functions and data. Using the ground-truth file orders, we found that 9 out of 121 binaries in our dataset do not follow this assumption. In these binaries, the file order for functions and data are different. To test whether these 9 binaries show different accuracy compared to the remaining ones, we performed a Welch's t-test. Table II shows the results. Since all p-values are greater than 0.05, the difference between the two accuracy distributions is not statistically significant.

Next, we analyzed cross-file data accesses in our dataset. Unlike file ordering, the assumption that functions only access data from the same file never holds in our dataset. All 121 binaries contain at least one function that accesses data defined in other files. On average, 6.37% of identified data accesses are cross-file, while the remaining 93.63% are within the same file. Since all binaries have cross-file data accesses, we examined whether more cross-file accesses would damage our results. We calculated the Spearman correlation between accuracy metrics and the percentage of cross-file data accesses. Table III shows the results. Based on the correlation values and p-values, file recovery accuracy has no meaningful correlation with the percentage of cross-file data accesses.

TABLE III
SPEARMAN CORRELATION BETWEEN FILE RECOVERY ACCURACY AND CROSS-FILE DATA ACCESS PERCENTAGE

	JC	FMI	RI	ARI
Corr.	0.0552	-0.0035	0.0192	-0.0554
P-value	0.5546	0.9703	0.835	0.5532

TABLE IV
ACCURACIES FOR BINARY MODULARIZATION TOOLS (ABBREVIATIONS: M-MOJOFM, A-A2A, C-$c2c_{cvg}$, R-ARI, J-$a2a_{adj}$)

Tool	Bash					Libxml2					Groff					Libcrypto				
	M	A	C	J	R	M	A	C	J	R	M	A	C	J	R	M	A	C	J	R
ModX	48	75	0	34	4	20	70	0	37	2	54	80	0	43	10	25	74	0	34	3
BCD	77	80	2	22	6	73	82	0	28	18	70	82	0	29	8	64	83	5	32	17
DeLink	81	78	2	18	4	79	82	6	32	21	66	82	0	35	8	74	80	3	26	10
BinStruct	82	88	10	53	35	70	93	43	68	59	89	90	14	66	37	84	84	16	40	26
Impv.	1	8	8	19	29	-9	11	37	31	38	19	8	14	23	27	10	1	11	6	9

E. RQ2: Accuracy of Module Recovery Evaluation

1) Ground Truth Knowledge Construction

Unlike file structures, the ground truth for module structures cannot be directly extracted and typically requires extensive manual annotation [42]. As a result, such ground-truth data is relatively scarce. We collected two projects from previous SAR studies (*Libxml2* and *Bash*). For these two project, we directly adopt their labeled module structures as ground truth. For the remaining dataset, we seek projects with a well-defined folder organization, where each top-level directory corresponds to a distinct module with a clear responsibility. Among the 119 remaining binaries, we identify two such projects. The first is *OpenSSL*: its *libcrypto.so* component is structured such that each folder corresponds to a cryptographic algorithm, forming a natural module boundary. The second is *Groff*, whose source code is neatly organized into subdirectories like src, libs, and utils, each reflecting a distinct functionality. For these two projects, we adopt their folder structures as the ground truth for module organization without any modifications. In total, we obtain four ground-truth module structures for our evaluation.

2) Evaluation Metrics

To assess the similarity between the recovered module structures and the ground truth, we adopt five evaluation metrics following recent studies [17]: MoJoFM [43], $a2a$ [44], $c2c_{cvg}$ [44], $a2a_{adj}$ [17], and the Adjusted Rand Index (ARI), which is also used to evaluate file recovery results. MoJoFM [43] measures similarity based on the edit distance between two clustering results, defined as follows:

$$MoJoFM(A,B) = 1 - \frac{mno(A,B)}{\max(mno(\forall A,B))} \quad (7)$$

where A and B represent two clusterings, and $mno(A,B)$ denotes the minimum number of Move or Join operations required to transform clustering A into clustering B.

$a2a$ [44] is another distance-based metric defined using the edit distance between two clusterings:

$$a2a(A,B) = 1 - \frac{mto(A,B)}{aco(A) + aco(B)} \quad (8)$$

where $mto(A, B)$ represents the minimum number of operations required to transform clustering A into clustering B, and $aco(A)$ denotes the number of operations needed to construct clustering A from an empty clustering.

$c2c_{cvg}$ [44] measures similarity based on the number of similar clusters shared between two clusterings:

$$c2c_{cvg}(A, B) = \frac{|simC(A, B)|}{|A|} \quad (9)$$

where $simC(A, B)$ denotes the set of clusters in A that have a corresponding similar cluster in B. Following [17], we set the similarity threshold to 0.66 in our evaluation.

$a2a_{adj}$ [17] is designed to address the limited variation issue observed in the original $a2a$ metric. It is defined as follows:

$$a2a_{adj} = 1 - \alpha \frac{mto_m(A, B)}{mto_m^{max}(A, B)} - \beta \frac{mto_{ar}(A, B)}{aco(A) + aco(B)} \quad (10)$$

where $mto_m(A, B)$ is the movement cost, $mto_{ar}(A, B)$ is the add/remove cost, and $mto_m^{max}(A, B)$ is the maximum possible movement cost. The coefficients α and β balance the contribution of each term.

3) Baseline Selections

For module recovery, we compare our approach against both binary-level module recovery tools and source code-level software architecture recovery tools. For binary module recovery, we reuse the baselines from RQ1, which include BCD, ModX, and DeLink. For source code-level module recovery, we follow the experimental setup of Zhang et al. [17] and implement five representative tools. ACDC [28] is a pattern-based clustering algorithm that identifies subgraphs by recognizing dominator nodes and their corresponding dominated nodes. Bunch [13] employs a search algorithm to explore the modular solution space, aiming to maximize modular quality (MQ), which evaluates the modular structure by measuring high cohesion within modules and low coupling between modules. LIMBO [15] is a hierarchical clustering algorithm based on the information bottleneck optimization principle, making it well-suited for handling large and complex software systems. WCA [14] assigns different weights to various types of dependencies for architecture recovery, enabling architecture views to be generated at different levels of abstraction. We include two of its variants, WCA_UEM and WCA_UEMNM, which employ different clustering strategies. SARIF [17] is a recent technique that integrates file dependencies, code text, and code structure to effectively recover software architecture through detailed dependency analysis and topic extraction.

4) Module Recovery Results and Discussions

RQ2.1 Comparison to Binary Level Baselines: For the module-level ground truths, each corresponds to a group of source files. In our scenario, we transform these file groupings into function groupings, where each group of functions consists of all functions originating from the source files within the same ground-truth module. We then compare the recovered groupings with this ground-truth grouping. The results are presented in Table IV. The last row of the table reports the absolute metric value percentage point (pp.) [45]improvement against the best result from the three baselines. We use pp. to measure improvement instead of relative percentage increase because some baseline scores, especially for the $c2c_{cvg}$ metric, are zero.

TABLE V
ACCURACIES FOR SOURCE CODE MODULARIZATION TOOLS.

Tool	Bash					Libxml2					Groff					Libcrypto				
	M	A	C	J	R	M	A	C	J	R	M	A	C	J	R	M	A	C	J	R
ACDC	91	86	11	47	22	32	83	40	28	8	73	89	0	57	27	53	85	14	38	19
Bunch	75	84	12	35	14	42	86	20	38	28	84	88	14	47	22	31	82	6	27	11
LIMBO	81	83	8	34	13	64	90	27	56	40	76	84	4	44	11	33	81	0	20	10
SARIF	**91**	**92**	**29**	**66**	**49**	**80**	**94**	**64**	**75**	**70**	85	**95**	**40**	**75**	**56**	60	**90**	**50**	**59**	**54**
WCA_UE	63	84	9	29	13	38	85	25	34	14	69	92	25	59	12	12	78	0	13	1
WCA_NM	66	84	6	31	14	58	89	43	53	43	69	92	25	59	12	15	79	0	15	1
BinStruct	82	88	10	53	35	70	93	43	68	59	**89**	90	14	66	37	**84**	84	16	40	26

As shown in the table, across 20 metric values evaluated on 4 projects, BinStruct achieves the best score in 19 out of 20 cases. We also compute the average scores for each tool over the four projects. On average, BinStruct outperforms the best baseline by 5.1 to 25.7 pp. across the various metrics.

The only metric where BinStruct performs worse than the baselines is the MoJoFM score for the Libxml2 project. As noted in [17], MoJoFM tends to favor results with many smaller groups. For Libxml2, the ground truth consists of only 13 modules, whereas DeLink and BCD cluster the functions into 54 and 34 groups respectively. Our result groups the functions into 14 groups, closer to the ground truth but fewer than the baselines. This difference likely explains the lower MoJoFM score. Notably, the other four metrics, which do not exhibit a bias toward the number of groups, show significantly higher scores for BinStruct.

> **Answering RQ2.1:** BinStruct outperforms the other three binary modularization tools. On average, it achieves 5.1 to 25.7 pp. higher scores across the five evaluation metrics.

RQ2.2 Comparison to Source Code Level Baselines: Besides the binary modularization tools, we also compared our tool against SAR tools. These tools are applied to the corresponding source code repositories of the binaries to produce source-level module structures, which group source files into modules. We then map this file-level grouping to the corresponding binary functions and evaluate the recovered module structure against the ground truth. The metric results are presented in Table V.

In the table, we highlight the top two performing tools in green and bold the best-performing tool. BinStruct ranks first on 2 out of 20 (10%) total scores and achieves second place on 13 out of 20 (65%) scores across the five metrics. In most cases, BinStruct's performance is only slightly behind SARIF. The averaged performance across the four projects is summarized in Table VI. As shown, BinStruct achieves the highest score for MoJoFM and ranks second for the remaining four metrics.

TABLE VI
AVERAGED ACC. FOR SOURCE MODULARIZATION TOOLS AND BINSTRUCT.

Tool	MoJoFM	a2a	c2c	a2a_adj	ARI
ACDC	62.18	85.74	16.13	42.72	18.87
Bunch	57.88	85.03	13.02	36.92	18.57
LIMBO	63.40	84.52	9.99	38.80	18.52
SARIF	78.88	**92.61**	**45.76**	**68.72**	**57.44**
WCA_UE	45.44	84.64	14.77	33.55	10.06
WCA_NM	51.87	85.88	18.53	39.41	17.38
BinStruct	**81.08**	88.48	20.69	56.89	39.13

TABLE VII
ABLATION RESULTS FOR FILE RECOVERY

Stage	JC	FMI	RI	ARI	Coverage
Data Ref.	0.7418	0.8464	0.9266	0.7694	0.6342
Func. Call	0.5568 (-24.94%)	0.7071 (-16.45%)	0.9078 (-2.02%)	0.6337 (-17.64%)	0.9841 (+55.17%)
LLM	0.5667 (+1.77%)	0.7146 (+1.06%)	0.9094 (+0.18%)	0.6431 (+1.49%)	/ /

Although our performance is slightly lower than SARIF, it is important to note that SARIF recovers module structures based on source code, which contains significantly more semantic information than binaries. Additionally, SARIF incorporates the folder structure of the repository as an input feature, which closely aligns with the ground truth module boundaries in our dataset. In contrast, such structural information is unavailable in binaries and thus inaccessible to BinStruct. Despite these limitations, BinStruct still outperforms the remaining five SAR tools on average, demonstrating its strong capability in recovering module structures from binaries alone.

> **Answering RQ2.2:** BinStruct achieved the highest averaged score for MoJoFM and the second best score for the remaining 4 metrics compared to the 6 SAR tools operating on source code. The best performing source code tool SARIF takes advantages of folder structures, which is not available in binary.

F. RQ3: Ablation Study

In this RQ, we aim to explore how each part of our design contribute to the final result.

RQ3.1 File Recovery: As described in Section II, the file recovery process comprises three main stages: initial grouping based on data violations, expansion using function call relationships, and refinement using LLMs. We evaluated the accuracy and summarized the results in Table VII.

In the first stage, BinStruct identifies initial groupings by analyzing data references. As shown in the table, this stage achieves notably high accuracy. However, due to the limited availability of data references, only a subset of functions can be grouped. Specifically, 63.42% of functions in our dataset are grouped during this stage with high precision.

In the second stage, function call relationships are used to distinguish between different files. This approach offers broader coverage, as function calls are prevalent and span 98.41% of all functions. However, the accuracy of this stage is relatively lower because cross-file function calls, which may introduce false positives, are significantly more frequent than cross-file data references.

In the final stage, an LLM is employed to make minor adjustments that refine file boundaries. As detailed in Section II-A4, we first identify those suspicious boundaries, then use LLMs to refine the boundaries. Across the 121 binaries in our dataset, a total of 325 suspicious boundaries were identified. As shown in Table VII, after the refinement by LLMs, the accuracy achieved a modest improvement by 1.49% for ARI. Though the overall metric improvement is modest due to the limited number of suspicious boundaries, it can effectively fix these identified boundaries. Among the 325 suspicious boundaries, 30.77% were refined into extract file boundaries, and 79.69% showed increased accuracy after refinement. This result indicate that LLM can effectively address those suspicious boundaries.

TABLE VIII
ABLATION RESULTS FOR MODULE RECOVERY

Approach	MoJoFM	a2a	c2c	a2a_adj	ARI	Coverage
Dependency	61.43	86.79	3.70	49.02	31.18	100%
Semantic	68.55	87.42	6.48	52.15	30.73	100%
Consensus	89.52	91.77	35.61	62.30	51.08	47.71%
Final	81.08	88.48	20.69	56.89	39.13	100%

> **Answering RQ3.1:** For file recovery, the data reference stage accurately identifies initial groupings, covering 63.42% of functions. The function call stage achieves broader coverage, reaching 98.41% of functions, though with lower accuracy. The LLM refinement improves 79.69% of suspicious file boundaries.

RQ3.2 Module Recovery: For module recovery, we evaluate three key intermediate results: the dependency-based clustering, the semantic-based clustering, and their consensus. Each of these is compared against the ground-truth clustering, and the averaged accuracy for each step is presented in Table VIII.

As shown in the table, both the dependency-based and semantic-based clustering individually yield significantly lower scores than the final result. In contrast, the consensus of the two, defined as the file groups formed consistently by both clustering methods, achieves notably higher accuracy, although it covers only 47.71% of the files on average. This finding suggests that consensus clustering offers a reliable foundation for the LLMs to incrementally complete the module structure by integrating the remaining files.

> **Answering RQ3.2:** Clustering based solely on dependencies or semantics results in lower accuracy. In contrast, the consensus groups are much more accurate and can cover 47.71% of files on average.

TABLE IX
AVERAGED TIME CONSUMPTION FOR BINARY STRUCTURE RECOVERY

Tools	ModX	DeLink	BCD	BinStruct		
				File	Module	Total
Time(s)	170.25	20.78	44.99	7.42	34.46	41.88

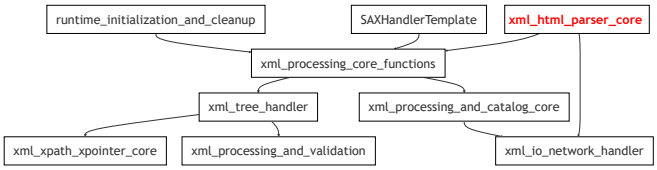


Fig. 4. Recovered Module Structure of Libxml2

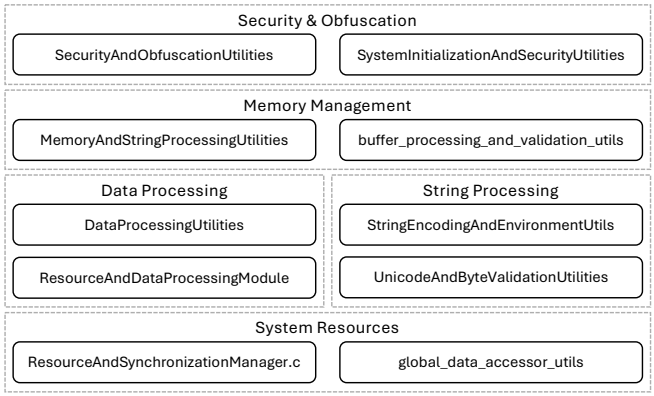


Fig. 5. Recovered Module Structure of PredatorTheStealer

G. RQ4: Performance and Cost Evaluation

We first present the time required to recover binary structure using BinStruct and baseline methods in Table IX. As shown, BinStruct is faster than ModX and BCD in total recovery time but slower than DeLink. However, it is important to note that DeLink only performs file-level recovery. When comparing only the file recovery stage, BinStruct is significantly faster than DeLink, reducing time consumption by 64.29%. Moreover, among the 41.88 seconds of total processing time, 27.19 seconds (64.92%) are spent on querying LLMs, while the remaining components of our algorithm account for only 14.69 seconds on average.

We also analyzed the total number of tokens consumed by the LLM component. On average, recovering a single binary involved 1.44 million prompt tokens and 57.65 thousand completion tokens. Based on OpenAI's pricing at the time of our experiments, the average cost to analyze one binary was approximately $0.167.

> **Answering RQ4:** BinStruct requires 7.42 seconds and 34.46 seconds on average to recover file and module level structures, respectively, with a cost of approximately $0.167 per binary.

H. RQ5: Applications

In this section, we demonstrate the potential real-world security applications of BinStruct. One key application is identifying attack surfaces directly from binary code. Figure 4 shows the recovered module structure of Libxml2 (single-function modules are omitted for clarity). Without this module-level view, security analysts must manually inspect all 1,150 functions to locate potential attack surfaces, making the process time-consuming and error-prone.

With the module structure recovered by BinStruct, analysts can more efficiently pinpoint security-critical components. For example, the identification of a parser module (*xml_html_parser_core*) immediately highlights a likely attack surface, since parsers are commonly exposed to untrusted input and frequently contain vulnerabilities. To validate this observation, we examined 142 CVEs associated with Libxml2 from the CVE website [46]. Our analysis revealed that 62 vulnerabilities (43.66%) were linked to functions within this parser module. This finding suggests that the module structure produced by BinStruct can effectively guide vulnerability auditing by narrowing the focus to high-risk areas.

Another application of BinStruct is malware analysis. We tested our tool on PredatorTheStealer [47], a known malware sample from Vx-underground [48]. This binary contains 6,407 functions, which BinStruct grouped into 25 modules. Figure 5 highlights the 10 largest modules and their recovered names.

Even though various code obfuscation techniques were applied, BinStruct was able to uncover suspicious structural patterns. For example, the recovered structure reveals several modules dedicated to security-related and obfuscation functionalities like sophisticated XOR transformations and anti-debugging mechanisms, which are uncommon in legitimate software. Moreover, this large binary spends unusual high efforts on basic tasks like memory and string operations. Meanwhile, it lacks the high-level features that would normally use these operations. These characteristics indicate that the binary is highly likely to be malicious.

BinStruct also helps to understand the behavior of malware. According to reports [49], the primary malicious activity of PredatorTheStealer is stealing user data. The recovered module structure confirms this by showing 1) a *global_data_accessor* module responsible for collecting data, 2) several data processing modules that handle different types of stolen information like environment variables and encoded strings, and 3) the *SecurityAndObfuscationUtilities* module conceals these activities. This clear correspondence between the recovered modules and the known behavior of the malware demonstrates the potential of BinStruct to assist analysts in understanding malware functionality.

> **Answering RQ5:** Two case studies demonstrate the potential of BinStruct on downstream tasks like attack surface detection and malware analysis.

IV. Threats to Validity

The first threat is the limited number of ground truth module structures. Due to high manual effort of labeling reliable ground truths [50], we can only obtain limited reliable ground truths. To mitigate this, we obtained two more ground truths based on the folder structure. Moreover, though cannot be measured by metrics, we published the module recovery result for all 121 binaries on our website for public verification [26]. The second threat is the potential hallucination of LLMs. To mitigate this, we carefully provided necessary structural information and contexts to lower the chance for hallucination. Moreover, strategies like chain-of-thought, double confirmation, etc. are also applied to further mitigate the hallucination. The third threat is the potential inaccuracy of IDA Pro, which serve as the foundation for many of our analysis. A potential solution is to combine with dynamic analysis to further increase the accuracy. The fourth threat relates to handling obfuscated binaries. Most real-world obfuscations work at code or IR level before linking. Since file ordering happens at link-time, these techniques do not affect file ordering or BinStruct's effectiveness. The case study on PredatorTheStealer (as detailed in section III-H) confirms this by showing that BinStruct can handle binaries with common obfuscations like dynamic API loading and randomization. However, post-link-time obfuscations like packing or encryption disable static analysis on the original binary. While unpacking techniques could be applied first to address these obfuscations, we emphasize that BinStruct is designed to analyze regular binaries rather than highly protected ones.

V. Related Work

Binary Comprehension. Attempts have also been made to recover the semantics of binary files. Dire [10] has been proposed to recover the variable names of X84-64 binaries with neural networks. Chen et al. [11] uses Transformer-based models to recover the variable names and data types of C/C++ binaries. Transfer learning has also been applied to recover the variable names of binaries [12]. In addition, works like Debin [51], punstrip [52], NFRE [53] and SymLM [54] have also been proposed to recover the function symbols. With the invention of large language models (LLMs), LLMs show great potential in this task. Hu et al. [9] proposed DeGPT, a framework that uses LLMs to optimize the output of disassemblers and decompilers, making them more human-readable. Tan et al. [55] proposed LLM4Decompile, a framework that uses LLMs to refine the outputs of disassemblers, and translate them into decompiled source code. These works mainly focus on the recovery of variable names and data types, instead of the understanding of the overall structure of the binary files.

Module & File Structure Recovery. The software module recovery tools are to automatically recover software architectures from their implementations. These tools are often referred as software architecture recovery tools. Bunch [13] clusters source files of software systems by maximizing the modularity quality (MQ) of the dependency graph. WCA [14] is a hierarchical clustering technique often used in software clustering scenarios. LIMBO [15] applies information theory concepts to software clustering. DAGC [16] is similar to Bunch but optimizes its search space. MCA and ECA [56] incorporate multi-objective optimization into software clustering. The Cooperative clustering technique (CCT) [57] is a consensus-based approach utilized in software clustering. Mohammadi et al. [58] use existing knowledge in the dependency graph to develop a neighborhood tree that guides clustering. FCA [59] clusters software systems by performing operations on the dependency matrix. It has good scalability and can cluster very large software systems within a reasonable amount of time. Unlike these techniques that rely on static dependencies, Xiao et al. [60] demonstrate that dynamic dependencies offer certain advantages. SARIF [17] enhances recovery by combining dependencies, folder structures, and textual information.

Efforts have also been made to recover the file or module structures on binary files. DeLink [18] recovers the file structure of binary files based on structured features. ModX [2] recovers modules level information from binary files, with fast unfolding Louvain algorithm [31]. And BCD [19] is a file structure recovery tool that constructs a weighted directed graph to recover file structures from binary files. These works mainly focus on syntactic level information and program structure, without the understanding of the semantics and functionalities of the binary files.

VI. Conclusion

We presented BinStruct, an LLM-enhanced framework that recovers the structures from stripped binaries. Our evaluation shows that BinStruct outperforms existing binary recovery tools by 17.09% and 25.07% for file and module recovery respectively, while matching source-level SAR tools despite working with stripped binaries. Through the accurate recovery result, BinStruct can provide practical understandings for downstream tasks like security analysis.

VII. Data Availability

The code, data and prompts are available on our website: https://sites.google.com/view/binstruct/home [26].

Acknowledgments

This research is supported by the National Research Foundation, Singapore, and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG4-GC-2023-008-1B); by the National Research Foundation Singapore and the Cyber Security Agency under the National Cybersecurity R&D Programme (NCRP25-P04-TAICeN); and by the Prime Minister's Office, Singapore under the Campus for Research Excellence and Technological Enterprise (CREATE) Programme. Any opinions, findings and conclusions, or recommendations expressed in these materials are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore, Cyber Security Agency of Singapore, Singapore.

REFERENCES

[1] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 94–109.

[2] C. Yang, Z. Xu, H. Chen, Y. Liu, X. Gong, and B. Liu, "Modx: binary level partially imported third-party library detection via program modularization and semantic matching," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1393–1405.

[3] D. Zhang, P. Luo, W. Tang, and M. Zhou, "Osldetector: Identifying open-source libraries through binary analysis," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1312–1315.

[4] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu, "Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search." in *NDSS*, 2023.

[5] J. Tian, W. Xing, and Z. Li, "Bvdetector: A program slice-based binary code vulnerability intelligent detection system," *Information and Software Technology*, vol. 123, p. 106289, 2020.

[6] H. Yan, S. Luo, L. Pan, and Y. Zhang, "Han-bsvd: a hierarchical attention network for binary software vulnerability detection," *Computers & Security*, vol. 108, p. 102286, 2021.

[7] Y. Xu, Z. Xu, B. Chen, F. Song, Y. Liu, and T. Liu, "Patch based vulnerability matching for binary programs," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 376–387.

[8] L. Li, S. H. Ding, Y. Tian, B. C. Fung, P. Charland, W. Ou, L. Song, and C. Chen, "Vulanalyzer: Explainable binary vulnerability detection with multi-task learning and attentional graph convolution," *ACM Transactions on Privacy and Security*, vol. 26, no. 3, pp. 1–25, 2023.

[9] P. Hu, R. Liang, and K. Chen, "Degpt: Optimizing decompiler output with llm," in *Proceedings 2024 Network and Distributed System Security Symposium*, vol. 267622140, 2024.

[10] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, "Dire: A neural approach to decompiled identifier naming," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 628–639.

[11] Q. Chen, J. Lacomis, E. J. Schwartz, C. Le Goues, G. Neubig, and B. Vasilescu, "Augmenting decompiler output with learned variable names and types," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4327–4343.

[12] K. K. Pal, A. P. Bajaj, P. Banerjee, A. Dutcher, M. Nakamura, Z. L. Basque, H. Gupta, S. A. Sawant, U. Anantheswaran, Y. Shoshitaishvili *et al.*, ""' len or index or count, anything but v1": Predicting variable names in decompilation output with transfer learning," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 4069–4087.

[13] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 1999, pp. 50–59.

[14] O. Maqbool and H. A. Babri, "The weighted combined algorithm: A linkage algorithm for software clustering," in *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.* IEEE, 2004, pp. 15–24.

[15] P. Andritsos, P. Tsaparas, R. J. Miller, and K. C. Sevcik, "Limbo: Scalable clustering of categorical data," in *International Conference on Extending Database Technology*. Springer, 2004, pp. 123–146.

[16] S. Parsa and O. Bushehrian, "A new encoding scheme and a framework to investigate genetic clustering algorithms," *Journal of Research and Practice in Information Technology*, vol. 37, no. 1, pp. 127–143, 2005.

[17] Y. Zhang, Z. Xu, C. Liu, H. Chen, J. Sun, D. Qiu, and Y. Liu, "Software architecture recovery with information fusion," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1535–1547.

[18] Z. Lang, Z. Xu, X. Chen, S. Lv, Z. Song, Z. Shi, and L. Sun, "Delink: Source file information recovery in binaries," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1009–1021.

[19] V. Karande, S. Chandra, Z. Lin, J. Caballero, L. Khan, and K. Hamlen, "Bcd: Decomposing binary code into components using graph-based clustering," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 393–398.

[20] L. Jiang, J. An, H. Huang, Q. Tang, S. Nie, S. Wu, and Y. Zhang, "Binaryai: Binary software composition analysis via intelligent binary source code matching," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[21] S. Arasteh, P. Jandaghi, N. Weideman, D. Perepech, M. Raghothaman, C. Hauser, and L. Garcia, "Trim my view: An llm-based code query system for module retrieval in robotic firmware," *arXiv preprint arXiv:2503.03969*, 2025.

[22] "Elf - osdev wiki," https://wiki.osdev.org/ELF, 2025.

[23] "Using ld, the gnu linker - bfd," https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_chapter/ld_5.html#SEC30, 2025.

[24] F. M. Dekking, *A Modern Introduction to Probability and Statistics: Understanding why and how*. Springer Science & Business Media, 2005.

[25] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proceedings of the national academy of sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[26] Anonymous, "Our website," https://sites.google.com/view/binstruct/home, 2025.

[27] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 552–555.

[28] V. Tzerpos and R. C. Holt, "Acdc: an algorithm for comprehension-driven clustering," in *Proceedings Seventh Working Conference on Reverse Engineering*. IEEE, 2000, pp. 258–267.

[29] Z. Gyongyi, H. Garcia-Molina, and J. Pedersen, "Combating web spam with trustrank," in *Proceedings of the 30th international conference on very large data bases (VLDB)*, 2004.

[30] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[31] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.

[32] M. Nimisha, T. Shamitha, and G. Mishra, "Comparative analysis of embedding models for keyphrase extraction: A keybert-based approach," in *2023 4th IEEE Global Conference for Advancement in Technology (GCAT)*. IEEE, 2023, pp. 1–6.

[33] "sentence tramsformer," https://www.sbert.net/, 2024.

[34] B. J. Frey and D. Dueck, "Clustering by passing messages between data points," *science*, vol. 315, no. 5814, pp. 972–976, 2007.

[35] Hex Rays, "Hex rays - state-of-the-art binary code analysis solutions," https://hex-rays.com/ida-pro/, 2025.

[36] OpenAI, "gpt-4.1-nano-2025-04-14," Apr. 2025, language model developed by OpenAI. Version as of 2025-04-14.

[37] GCC Development Community, *Using the GNU Compiler Collection (GCC)*, 7th ed., Free Software Foundation, 2019, the GNU Compiler Collection, version 7.5.0. Documentation available at https://gcc.gnu.org/onlinedocs/gcc-7.5.0/gcc.pdf.

[38] G. W. Milligan, S. C. Soon, and L. M. Sokol, "The effect of cluster size, dimensionality, and the number of clusters on recovery of true cluster structure," *IEEE transactions on pattern analysis and machine intelligence*, no. 1, pp. 40–47, 1983.

[39] E. B. Fowlkes and C. L. Mallows, "A method for comparing two hierarchical clusterings," *Journal of the American statistical association*, vol. 78, no. 383, pp. 553–569, 1983.

[40] W. M. Rand, "Objective criteria for the evaluation of clustering methods," *Journal of the American Statistical association*, vol. 66, no. 336, pp. 846–850, 1971.

[41] L. Hubert and P. Arabie, "Comparing partitions," *Journal of classification*, vol. 2, no. 1, pp. 193–218, 1985.

[42] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic, "Obtaining ground-truth software architectures," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 901–910.

[43] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.* IEEE, 2004, pp. 194–203.

[44] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural change in open-source software systems," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 235–245.

[45] Wikipedia contributors, "Percentage point — Wikipedia, The Free Encyclopedia," 2025, [Online; accessed 31-May-2025]. [Online]. Available: https://en.wikipedia.org/wiki/Percentage_point
[46] The MITRE Corporation, "CVE Search Results for libxml2," https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=libxml2, 2025.
[47] vxunderground, "Win32.PredatorTheStealer.c.7z - MalwareSourceCode," https://github.com/vxunderground/MalwareSourceCode/blob/main/Win32/Stealers/Win32.PredatorTheStealer.c.7z.
[48] "Vx-underground," https://vx-underground.org.
[49] NHS Digital, "Cyber Alert CC-2970: Predator Trojan," https://digital.nhs.uk/cyber-alerts/2019/cc-2970, NHS Digital, Tech. Rep. CC-2970, March 2019, last accessed 31 May 2025.
[50] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, and R. Kroeger, "Measuring the impact of code dependencies on software architecture recovery techniques," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 159–181, 2017.
[51] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1667–1680.
[52] J. Patrick-Evans, L. Cavallaro, and J. Kinder, "Probabilistic naming of functions in stripped binaries," in *Proceedings of the 36th Annual Computer Security Applications Conference*, 2020, pp. 373–385.
[53] H. Gao, S. Cheng, Y. Xue, and W. Zhang, "A lightweight framework for function name reassignment based on large-scale stripped binaries," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 607–619.
[54] X. Jin, K. Pei, J. Y. Won, and Z. Lin, "Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1631–1645.
[55] H. Tan, Q. Luo, J. Li, and Y. Zhang, "Llm4decompile: Decompiling binary code with large language models," *arXiv preprint arXiv:2403.05286*, 2024.
[56] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 264–282, 2010.
[57] R. Naseem, O. Maqbool, and S. Muhammad, "Cooperative clustering for software modularization," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2045–2062, 2013.
[58] S. Mohammadi and H. Izadkhah, "A new algorithm for software clustering considering the knowledge of dependency between artifacts in the source code," *Information and Software Technology*, vol. 105, pp. 252–256, Jan. 2019.
[59] N. Teymourian, H. Izadkhah, and A. Isazadeh, "A Fast Clustering Algorithm for Modularization of Large-Scale Software Systems," *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1451–1462, Apr. 2022.
[60] C. Xiao and V. Tzerpos, "Software clustering based on dynamic dependencies," in *Ninth European Conference on Software Maintenance and Reengineering*. IEEE, 2005, pp. 124–133.