

DOI: 10.15514/ISPRAS-2025-37(4)-6



Extraction of Functionality from Binary Code

^{1,2} A.A. Ilina, ORCID: 0009-0002-6727-8050 <ilina@ispras.ru>

¹ Sh.F. Kurmangaleev, ORCID: 0000-0002-0558-2850 <kursh@ispras.ru>

¹ Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

² Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. Semantic code analysis is an important but time-consuming process used in many areas of programming. The purpose of this work is to study a method for automating the semantic analysis of binary code, which is based on dividing software into semantic kernels using partial traces of execution or subgraph extraction from call graph and highlighting their functionality.

Keywords: semantic analysis; static analysis; large language models.

For citation: Ilina A.A., Kurmangaleev Sh.F. Extraction of Functionality from Binary Code. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 4, part 1, 2025, pp. 97-110. DOI: 10.15514/ISPRAS-2025-37(4)-6.

Извлечение функциональности из бинарного кода

^{1,2} А.А. Ильина, ORCID: 0009-0002-6727-8050 <ilina@ispras.ru>

¹ Ш.Ф. Курмангалеев, ORCID: 0000-0002-0558-2850 <kursh@ispras.ru>

¹ Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

² Московский государственный университет имени М.В. Ломоносова,
Россия, 119991, Москва, Ленинские горы, д. 1.

Аннотация. Семантический анализ кода – важный, но трудоемкий процесс, используемый во многих областях программирования. Целью данной работы является изучение метода автоматизации семантического анализа бинарного кода, который основан на разделении программ на семантические ядра с использованием частичных трасс выполнения или выделения подграфов графа вызовов и выделения их функциональности.

Ключевые слова: семантический анализ; статический анализ; большие языковые модели.

Для цитирования: Ильина А.А., Курмангалеев Ш.Ф. Извлечение функциональности из бинарного кода. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 97–110 (на английском языке). DOI: 10.15514/ISPRAS-2025-37(4)-6.

1. Introduction

In the modern world, software audit for certification purposes is becoming increasingly important. When conducting an audit on a large amount of code, it is essential to understand the functionality of the software under study.

However, there are often limitations:

1. The customer may not be willing to provide access to the source code for analysis due to the fact that sections of the code are proprietary.
2. Not all functionality of the code is documented.

Due to these limitations, code analysis can be time-consuming, and there is a need for automated methods to extract program functionality. The constraints define the object of analysis: it is binary code.

There are many different approaches to extracting a description of the functionality implemented in binary code. These include semantic pattern mining, semantic code clone mining and methods based on the application of neural networks. However, existing methods have limitations due to the need to keep databases of patterns or programs up to date for comparison, the need for debugging information or other facilities, and the amount of time required.

This paper will investigate a method based on static and dynamic code analysis and the use of large language models, which will allow the extraction of functionality from binary code without the need for debugging information or pattern bases. The following sections will provide a more detailed description of existing solutions, an overview of the method, comparison with analogues, experimental results, and applicability of the method to real-world problems.

2. Related Work

One method of extracting functionality from binary code involves the identification of semantic patterns. CAPA [1] is a well-known tool for identifying malicious code that relies on a similar database of patterns referred to as "rules". CAPA facilitates the identification of the semantic characteristics and functionality of binary code without necessitating full reverse-engineering. The tool employs a mapping process that translates low-level code attributes into high-level actions, such as 'makes a network connection', 'performs encryption' and 'code injection'. The primary benefit of this approach is that it enables lightweight analysis of the program. However, the approach is inherently limited by the finite size of the rule base and the laborious process of creating such rules for complex algorithms.

Another method of functionality extraction is the search for semantic code clones. By comparing the program under study with code that has been shown to have known functionality, the semantics of program fragments can be determined. Modern methods for finding semantic clones of binary code include various approaches such as symbolic execution, graph methods, use of intermediate representations (IR), as well as methods based on vector representations and machine learning. Tools such as BinDiff [2] and BinHunt [3] leverage heuristics and graph representation of code to compare its structure; however, these tools often encounter challenges when dealing with high levels of code optimization and obfuscation. In contrast, SAFE [4], which utilizes function vectorization, has exhibited both high accuracy and expeditious execution times, particularly in the context of comparing functions compiled with disparate compilers and on disparate architectures. However, even the most advanced methods, such as Gemini [5] and Asm2Vec [6], encounter limitations when dealing with heavily obfuscated code. The utilization of semantic code clone mining is warranted in scenarios where the objective is the retrieval of particular functionalities; however, the limitations that emerge in the context of a semantic pattern database also pertain to this approach.

The following discussion will proceed by way of a further examination of methodologies associated with the implementation of neural networks. The article "How Far Have We Gone in Stripped Binary

Code Understanding Using Large Language Models" [7] reviews tools intended for semantic analysis of binary code using large language models and also creates a dataset to test the quality of these tools' performance. The article describes the sequence of steps most commonly used in the studied tools, which includes disassembling, decompiling, and neural network analysis stages. The authors note that the most common tasks in semantic analysis at the moment include function name recovery (tools for this task include NERO [8], NFRE [9], and SymLM [10]), and natural language generation for binary code segments (BinT5 [11] and HexT5 [12]). As outlined in the aforementioned article, all methods described involve a decompilation process. The employment of large language models on decompiled code has been demonstrated to yield results that surpass the accuracy of alternative methods by 10%. The utilization of language models on decompiled code demonstrates considerable promise for the purpose of learning binary code. However, the granularity of the current tasks (tools performing function-level analysis) may not be sufficient in some cases. Many binary files contain several thousand functions, and even with correctly recovered function names and generated descriptions, analyzing a binary file can take a long time. As a result, it is necessary to select semantic kernels (code areas) that implement a certain functionality in the binary code being studied.

The authors of the tool XRefer [13, 14] perform Gemini-powered cluster analysis for the binary file based on the metrics contained in the code. Like CAPA, XRefer is designed to analyze malware. It is based on the use of large language models and no longer has the constraints of a rule base. The tool can highlight any functionality. But, as mentioned, XRefer uses code metrics to analyze semantics, which may not be enough to recover functionality.

This article will delve into the process of cluster analysis which is equivalent to semantic kernels analysis, focusing on the method applied to decompiled code.

3. Method Overview

The proposed methodology in this paper is designed to determine the functionality of binary code by dividing it into sections called semantic kernels. The method is comprised of the following phases: first, sources for semantic kernels (functions in callgraph) are selected; second, semantic kernels are extracted and decompiled; third, a query is sent to a large language model for all kernels; and fourth, the functionality of the kernels is determined. The sequence of steps and the tools utilized in each step are illustrated in Fig. 1.

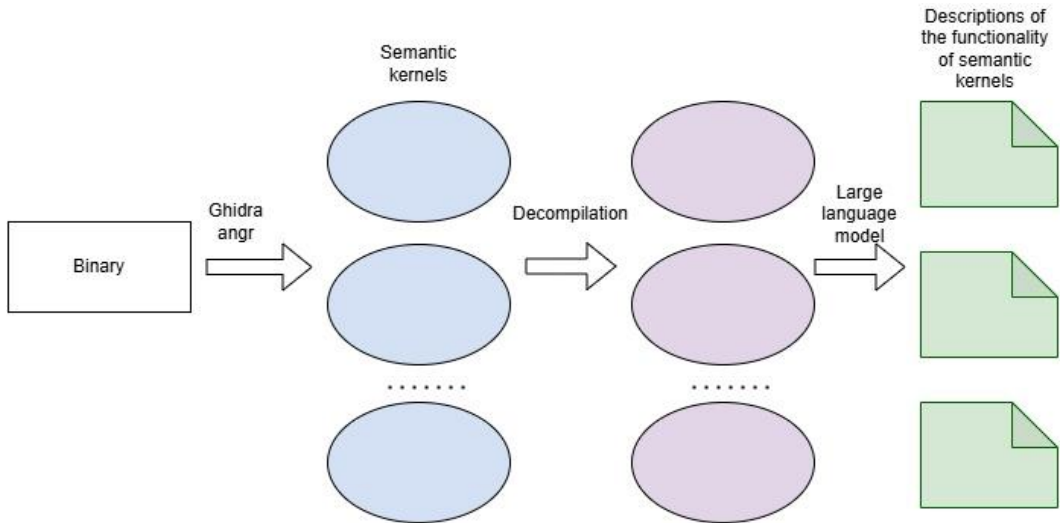


Fig. 1. General scheme of the method.

3.1 Code Division into Semantic Kernels

The goal of this stage is to reduce the number of objects required for binary code analysis by dividing the software into parts that implement specific semantics called semantic kernels. We use two types of semantic kernels.

Let's introduce their definitions:

- 1) Semantic kernel is the tree of fixed depth extracted from program's call graph, whose vertex is selected function.
- 2) Semantic kernel is the set of tracelets (partial traces of executions) that start from selected function.

In order to accomplish code division, it is necessary to select sources from which semantic kernels will be created. The selection of these sources is typically determined heuristically. Functions can be utilized as initial points for the generation of semantic kernels. In this context, semantic kernels can be considered as subgraphs of the call graph, with the vertices representing the selected sources. Fig. 2 shows semantic kernels highlighted in different colors, each with a specific functionality. For example, kernel 1 may be responsible for implementing a network protocol, kernel 2 may be responsible for reading files of a certain format, kernel 3 may be responsible for error handling, etc.

In this paper, we choose large functions with a large number of branches, strings, and comparisons as sources for semantic kernels. These functions often contain user data processing and are semantically independent from each other.

We use the Ghidra disassembler [15] to compute the metrics required for source selection.

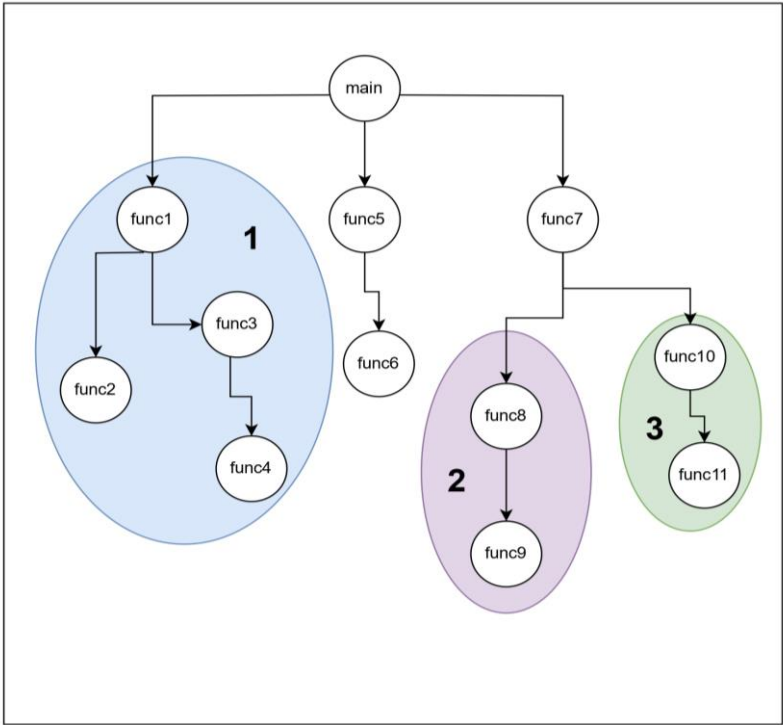


Fig. 2. Example of dividing the code into semantic kernels.

Code division into kernels is achieved in two ways as it listed before:

1. Trees of fixed depth (ranging from 2 to 3) are chosen from the call graph, whose vertices are the selected sources.

2. Angr [16] is used to generate artificial execution traces starting at the selected sources. The process of generating trails is as follows. First, the entry point of the selected function is taken to create a semantic kernel. Then instructions are taken for this entry point, which can be executed next in the control flow. The procedure is then continued for the instructions received in the previous step. Thus, the result is a set of possible execution paths starting in the function.

These techniques are employed due to limitations related to the context size of the models, as an entire call graph or large subgraphs cannot be processed by a single query.

3.2 Defining the Functionality of Semantic Kernels

The next step in the method is to define the functionality of extracted semantic kernels using a large language model. The semantic kernels obtained in the previous step are represented as binary code. In order to simplify the processing of kernels, decompilation is performed using the Ghidra. The presence of debugging information is not necessary for the proposed method to function.

A language model was chosen based on the following criteria:

1. Benchmark results. Models that successfully analyze code are of interest.
2. Local execution. This criterion ensures the security of data transmitted to the model, as it reduces the likelihood of sensitive information being leaked.
3. Output in required format. Since the tool requires an automated run and output generation, it needs the ability to convert model output into a specific format (JSON was chosen).
4. Training on data containing code samples. Given that the task involved analyzing code sections, a natural criterion was the choice of models trained on source code datasets.

According to the listed conditions, the Qwen 2.5 Coder model [17] was selected and run locally.

To determine the functionality included in the semantic kernel, algorithms are extracted from decompiled code, allowing us to understand the basic semantics of the investigated code snippet. When generating a query for the model, it was necessary to determine the level of detail in the query. Common types of queries, such as "Analyze given code snippets and identify all algorithms implemented: {CODE SNIPPET}" led to uninformative answers. A very detailed query listing a large number of algorithm types led to a longer response time, which was inefficient and unacceptable for the large number of objects being investigated. The most appropriate query is shown in Fig. 3.

3.3 Working Process

Fig. 4 presents the pseudo-code of the algorithm proposed in the paper.

4. Results

4.1 Quality Assessment of the Functionality Extraction

To evaluate the proposed method, we used TheAlgorithms repository [18], which contains implementations of various well-known algorithms in C and C++. The corresponding binary files for the algorithms were obtained from the repository. To ensure the purity of the experiment, we removed debugging information from the binary files. The algorithms in the repository were categorized. The distribution of algorithms in repository by category is presented in Fig. 5. Categories with fewer than two algorithms were excluded from consideration. For all binary files from selected categories, we performed division into semantic kernels in two ways: by extracting execution traces and by extracting subtrees from the call graph of binary. The resulting decompiled kernels were sent to a language model. The names of the algorithms in the model's answer may not exactly match the names of the algorithms whose implementations are presented in binary files. To speed up the testing process, we compare the lists of algorithms in the model output with the expected answer using st-

codesearch-distilroberta-base [19] embeddings. The testing procedure is illustrated in Fig. 6. After the results were filtered based on the threshold value, a manual validation was performed and the True Positive rate was calculated. The results for semantic kernels extracted using angr (Table 1). The best results of the method were achieved for the categories client server, hash, and searching and sorting. In most cases, the algorithms were recognized correctly. However, the minimum TP percentages (for math and misc) are related to the limitations of the angr: the generated execution traces were not long enough to fully detect the algorithms.

Prompt

Analyze the given code traces and identify all algorithms implemented in them from types: Combinatorial algorithms, Computational mathematics, Computational science, Computer science, Information theory and signal processing, Software engineering, Database algorithms, Distributed systems algorithms, Networking, Operating systems algorithms, Protocol algorithms.

If no specific algorithms are identified, respond with 'No specific algorithms identified'.

Your response should only contain the dict of algorithms by types or the 'No specific algorithms' message.

Answer must be in form {"Combinatorial algorithms": [algorithm1, ...], "Computational mathematics": [algorithm1, ...], "Computational science": [algorithm1, ...], "Computer science": [algorithm1, ...], "Information theory and signal processing": [algorithm1, ...], "Software engineering": [algorithm1, ...], "Database algorithms": [algorithm1, ...], "Distributed systems algorithms": [algorithm1, ...], "Networking": [algorithm1, ...], "Operating systems algorithms": [algorithm1, ...], "Protocol algorithms": [algorithm1, ...]}

Fig. 3. Prompt for model.

Algorithm 1 Semantic Kernels Extraction with Descriptions

```

1: Procedure EXTRACTSEMANTICKERNELS(binary, threshold, kernel_type)
2:   metrics ← calculateMetrics(binary)
3:   kernel_starts ← extractStartFunctionsByMetrics(metrics, threshold)
4:   kernels ← ∅
5:   for func in kernel_starts do
6:     if kernel_type == "angr" then
7:       kernel ← extractAngrKernel(func)
8:     else if kernel_type == "subgraph" then
9:       kernel ← extractSubgraphKernel(func)
10:    end if
11:    kernels.add(kernel)
12:  end for
13:  kernel_descriptions ← getKernelDescriptions(kernels)
14:  return kernel_descriptions
15: End Procedure

```

Fig. 4. Pseudocode of proposed method.

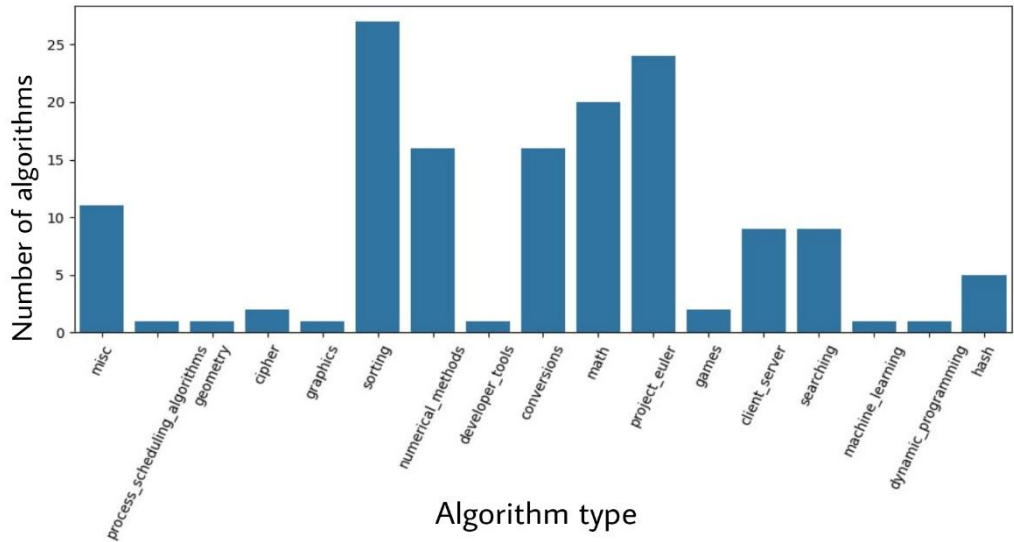


Fig. 5. The Algorithms repository structure.

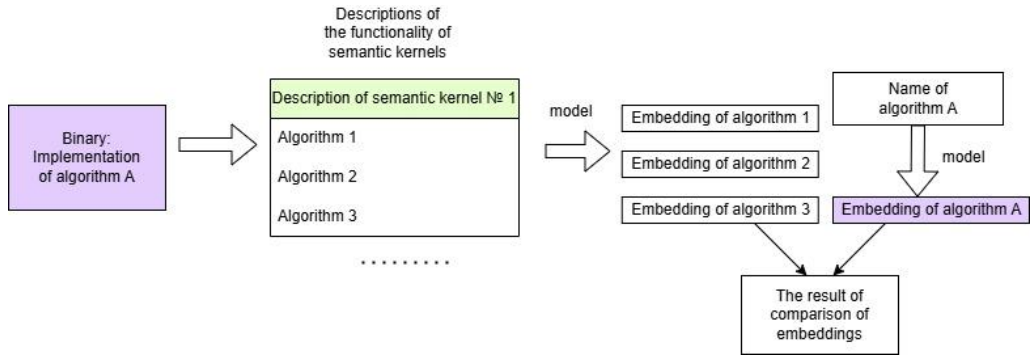


Fig. 6. Testing scheme.

Table 1. Results of using angr for semantic kernels extraction.

Category of algorithms	True Positive (%)
client_server	77,78
conversions	56,25
games	50
hash	80
math	45
misc	45,45
numerical_methods	50
searching	77,78
sorting	81,48

Similar testing was also conducted to extract semantic kernels using fixed-depth trees. The results are presented in Table 2. In some cases, the language model did not recognize the expected algorithms. However, in most cases, the low percentages are due to the fact that the names of the expected algorithms have a specific look (such as `remote_command_exec_udp_server`). And the response of

the model (UDP, for example) turned out to be not similar enough to the response string to be included in True positive.

Table 2. Results of using subtrees for semantic kernels extraction.

Category of algorithms	True Positive (%)
client_server	50
conversions	57,14
games	66,67
hash	83,33
math	52,17
misc	68,75
numerical_methods	44,44
searching	100
sorting	97,06

4.2 Application to Large Code Volume Analysis

As part of our research, we tested our tool on binary file of Microsoft Access 97 DBMS [20] containing 10,539 functions with no debugging information. The size of the binary is 2.9 Mb. The analysis by the Ghidra took 5 hours, the analysis using a large language model lasted an hour. This resulted in the formation of 256 clusters, each with their own functionality:

- 1) Work with databases: configuration, query processing, search, and indexes.
- 2) User interface.

Additionally, we found a cluster containing the "Magic Eight Ball" game, which was embedded as an "easter egg" by the software creators.

It was found after a series of additional prompts. The first prompt was used to extract algorithms from the software documentation. After that, the second prompt was used to compare the results of our tool with the expected algorithms in the documentation, creating a difference. This process resulted in 30 clusters being extracted. Some of these clusters contained false positives, but some also had undocumented features. To reduce the number of false positive clusters, a third prompt was used to check the generated answers. After filtering, only 5 clusters remained. These clusters were then manually checked. The remaining clusters are shown in Fig. 7.

Thus, functionality was found that was not typical for the software under study. In the future, the search for undocumented features can also be automated using queries to large language models. Fig. 8 shows the button that trigger the "Magic Eight Ball" game launch. Fig. 9 shows a function that contains a decompiled implementation of the game.

4.3 Limitations

When processing large sequences or a high volume of queries, the quality of responses may decrease. This can occur if the model loses context or is unable to process large amounts of information effectively, leading to incomplete responses. False positives may also arise as a result of hallucinations. Choosing a specialized model with a larger context size or pre-training may help reduce errors.

4.4 Comparison to Other Language Models

As mentioned earlier, the model selection criteria included the ability to run the model locally and whether the output matched the specified format in the query. We used TheAlgorithms sample to compare models that met the requirements, resulting in true positive values (see Table 3).

The testing process was performed on model CodeLlama [21] too. The output of CodeLlama for a large number of queries was not in the expected format. Qwen2.5_Coder_32B_Instruct proved to be the most suitable model for the task.

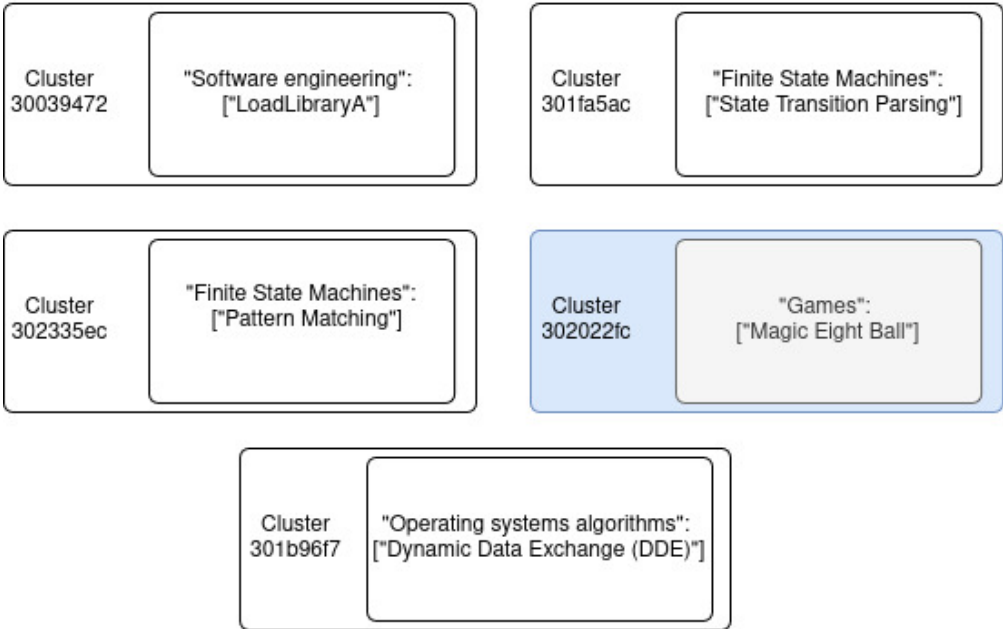


Fig. 7. Prompt sequence for undocumented features extraction.




Fig. 8. "Magic Eight Ball" in Microsoft Access 97.

4.5 Comparison to XRefer

As previously stated, XRefer employs code metrics to analyze clusters. In cases where these metrics contain sufficient information to describe the clusters (e.g., functions with numerous string constants reflecting the semantics of the code under study), cluster analysis yields satisfactory results. For

instance, the vlandhcpd [23] analysis resulted in the following set of clusters that effectively demonstrate the program semantics (see Fig. 10, DHCP Packet Processor functionality is detected). However, when the code contains few string constants but consists of a set of mathematical operations, the method is less applicable. This is demonstrated in the example of the code implementing the calculation of factorial, where XRefer did not reveal the expected functionality. As illustrated in Fig. 11, the function responsible for factorial computation does not contain string constants, leading to the failure of XRefer to recognize any functionality.



```
2 void FUN_302022fc(void)
3
4 {
5     int iVar1;
6     HMODULE hModule;
7     FARPROC pFVar2;
8     HRSRC hResInfo;
9     HGLOBAL hResData;
10    LPVOID pvVar3;
11
12    iVar1 = FUN_30025860((int *)0x0);
13    hModule = LoadLibraryA("winmm.dll");
14    if ((HMODULE)0x1f < hModule) {
15        pFVar2 = GetProcAddress(hModule,"sndPlaySoundA");
16        if (pFVar2 != (FARPROC)0x0) {
17            hResInfo = FindResourceA(DAT_302c3518,(LPCSTR)0x320,&DAT_3028d3b0);
18            if (hResInfo != (HRSRC)0x0) {
19                hResData = LoadResource(DAT_302c3518,hResInfo);
20                if (hResData != (HGLOBAL)0x0) {
21                    pvVar3 = LockResource(hResData);
22                    if (pvVar3 != (LPVOID)0x0) {
23                        (*pFVar2)(pvVar3,7);
24                    }
25                    FreeResource(hResData);
26                    FreeLibrary(hModule);
27                    FUN_300b332d((uint *)(&PTR_s_Most_likely_3028d1f8)[iVar1 % 0x12],
28                        (uint *)"The Magic Eight Ball says:",(HWND)&DAT_00000040,0,0,(uint *)0x0,0);
29                }
30            }
31        }
32    }
33    return;
34 }
```

Fig. 9. Function from cluster with “Magic Eight Ball” in Ghidra.

Table 3. Model comparison.

Model	True Positive (%)
Phi3min_f16 [22]	55,56
Qwen2.5_Coder_32B_Instruct	77,78

The method proposed in the paper identified the required functionality in both vlandhcpd and factorial computation examples. This was due to analyzing the code in decompiled form. This form allows us to find code patterns that are not tied to specific metrics.

5. Conclusion

The use of large language models for semantic analysis of binary code can significantly speed up the process of understanding the functionality of a program under study. By transitioning to the level of decompiled code, large language models can help with this process. This method can be used for extracting semantic kernels from binary code, which can then be applied in software certification tasks, as well as for studying large volumes of binary code.

```
This function serves following roles in clusters:
-----
As root node in:
  • cluster.id.0005 - DHCP Packet Processor

[-] DIRECT XREFS
-----
Lookup successful                                0x52ea
Lookup of MAC %x:%x:%x:%x:%x:%x in VLAN %d    0x5283
ERROR: Unknown DHCP Option Tag: %d             0x51cd
DHCP options:                                  0x5131
Server Identifier: %d.%d.%d.%d                 0x511b
Malformed len in server identifier tag: Should be 0x5041
MSG TYPE: %s                                    0x5014
MSG TYPE: %d                                    0x4fe5
Lease: %ld                                       0x4f92
Requested IP: %d.%d.%d.%d                      0x4f45
Malformed len in requested IP tag: Should be 4 but 0x4e6b
Hostname: %s                                    0x4e31
DNS Server %d: %d.%d.%d.%d                     0x4dba
Malformed len in DNS tag: Should be a multiple of 0x4cb2
Malformed len in DNS tag: Should 4 or greater but 0x4c7d
Router %d: %d.%d.%d.%d                         0x4c36
Malformed len in router tag: Should be a multiple 0x4b2e
Malformed len in router tag: Should 4 or greater b 0x4af9
Subnet Mask: %d.%d.%d.%d                       0x4acc
Malformed len in subnet mask tag: Should be 4 but 0x49f2
%d.%d.%d.%d                                     0x4896
Optionslen: %d                                  0x4822
UNKNOWN                                         0x5006
DHCPDISCOVER                                   0x5006
DHCPOFFER                                       0x5006
DHCPREQUEST                                    0x5006
```

Fig. 10. Cluster analysis by XRefer on vlandhcpd.

```
1 int64 __fastcall func1(int a1)
2 {
3     if ( a1 )
4         return a1 * func1((unsigned int)(a1 - 1));
5     else
6         return 1LL;
7 }
```

XRefer:

[0x11e9] NO FUNCTION CONTEXT AVAILABLE

Fig. 11. Cluster analysis by XRefer on factorial computation.

References

[1]. capa: Automatically Identify Malware Capabilities. Mandiant. Google Cloud Blog. [Online]. Available at: <https://cloud.google.com/blog/topics/threat-intelligence/capa-automatically-identify-malware-capabilities>, accessed 07.04.2025.

[2]. zynamics.com – BinDiff. [Online]. Available at: <https://www.zynamics.com/bindiff.html>, accessed 07.04.2025.

[3]. D. Gao, M. K. Reiter, and D. Song, BinHunt: Automatically Finding Semantic Differences in Binary Programs in Information and Communications Security, vol. 5308, L. Chen, M. D. Ryan, and G. Wang, Eds. In Lecture Notes in Computer Science, vol. 5308. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 238–255. doi: 10.1007/978-3-540-88625-9_16.

- [4]. L. Massarelli, G. A. D. Luna, F. Petroni, L. Querzoni, and R. Baldoni, SAFE: Self-Attentive Function Embeddings for Binary Similarity, Dec. 19, 2019, arXiv: arXiv:1811.05296. doi: 10.48550/arXiv.1811.05296.
- [5]. X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Oct. 2017, pp. 363–376. doi: 10.1145/3133956.3134018.
- [6]. S. H. H. Ding, B. C. M. Fung, and P. Charland, Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA: IEEE, May 2019, pp. 472–489. doi: 10.1109/SP.2019.00003.
- [7]. X. Shang et al., How Far Have We Gone in Binary Code Understanding Using Large Language Models. In 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME), Flagstaff, AZ, USA: IEEE, Oct. 2024, pp. 1–12. doi: 10.1109/ICSME58944.2024.00012.
- [8]. Y. David, U. Alon, and E. Yahav, Neural Reverse Engineering of Stripped Binaries using Augmented Control Flow Graphs. *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 1–28, Nov. 2020, doi: 10.1145/3428293.
- [9]. 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, Virtual Denmark: ACM, Jul. 2021, pp. 607–619. doi: 10.1145/3460319.3464804.
- [10]. 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, Los Angeles CA USA: ACM, Nov. 2022, pp. 1631–1645. doi: 10.1145/3548606.3560612.
- [11]. A. Al-Kaswan, T. Ahmed, M. Izadi, A. A. Sawant, P. Devanbu, and A. van Deursen, Extending Source Code Pre-Trained Language Models to Summarise Decompiled Binaries. 2023, arXiv. doi: 10.48550/ARXIV.2301.01701.
- [12]. J. Xiong, G. Chen, K. Chen, H. Gao, S. Cheng, and W. Zhang, HexT5: Unified Pre-Training for Stripped Binary Code Information Inference. In 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), Luxembourg, Luxembourg: IEEE, Sep. 2023, pp. 774–786. doi: 10.1109/ASE56229.2023.00099.
- [13]. mandiant/xrefer: FLARE Team’s Binary Navigator. [Online]. Available at: <https://github.com/mandiant/xrefer>, accessed 21.02.2025.
- [14]. XRefer: The Gemini-Assisted Binary Navigator | Google Cloud Blog. [Online]. Available at: <https://cloud.google.com/blog/topics/threat-intelligence/xrefer-gemini-assisted-binary-navigator>, accessed 21.02.2025.
- [15]. Ghidra Software Reverse Engineering Framework. Available at: <https://github.com/NationalSecurityAgency/ghidra>, accessed 31.01.2025.
- [16]. Y. Shoshitaishvili et al., SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA: IEEE, May 2016, pp. 138–157. doi: 10.1109/SP.2016.17.
- [17]. QwenLM/Qwen2.5: Qwen2.5 is the large language model series developed by Qwen team, Alibaba Cloud. [Online]. Available at: <https://github.com/QwenLM/Qwen2.5>, accessed 21.02.2025.
- [18]. The Algorithms. [Online]. Available at: <https://github.com/TheAlgorithms>, accessed 21.02.2025.
- [19]. flax-sentence-embeddings/st-codesearch-distilroberta-base. Hugging Face. [Online]. Available at: <https://huggingface.co/flax-sentence-embeddings/st-codesearch-distilroberta-base>, accessed 21.02.2025.
- [20]. Microsoft Access – Wikipedia. [Online]. Available at: https://en.m.wikipedia.org/wiki/Microsoft_Access, accessed 21.02.2025.
- [21]. codellama/CodeLlama-13b-Instruct-hf. Hugging Face. [Online]. Available at: <https://huggingface.co/codellama/CodeLlama-13b-Instruct-hf>, accessed 21.02.2025.
- [22]. microsoft/Phi-3-mini-4k-instruct. Hugging Face. [Online]. Available at: <https://huggingface.co/microsoft/Phi-3-mini-4k-instruct>, accessed 21.02.2025.
- [23]. aheck/vlandhcpd: VLAN aware DHCP server which listens on a trunk port. [Online]. Available at: <https://github.com/aheck/vlandhcpd>, accessed 21.02.2025.

Информация об авторах / Information about authors

Анна Александровна ИЛЬИНА – студентка магистратуры ВМК МГУ, лаборант ИСП РАН. Сфера научных интересов: статический анализ бинарного кода, символьное выполнение, применение больших языковых моделей.

Anna Aleksandrovna ILINA – a graduate student at the CMC MSU and a laboratory assistant at the ISP RAS. Area of her scientific interests: static binary code analysis, symbolic execution, the use of large language models.

Шамиль Фаимович КУРМАНГАЛЕЕВ – кандидат физико-математических наук, руководитель направления разработки автономных систем и технологий для создания безопасного ПО в ИСП РАН.

Shamil Faimovich KURMANGALEEV – Cand. Sci. (Phys.-Math.), head of the development of autonomous systems and technologies for creating secure software at the ISP RAS.

