Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

# A survey of binary code representation technology

Taiyan WANG[1,2], Qingsong XIE[1,2], Lu YU[1,2], Zulie PAN[1,2], Min ZHANG[†‡1,2]

[1]*College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China*

[2]*Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation, Hefei 230037, China*

[†]E-mail: zhangmindy@nudt.edu.cn

Received Feb. 6, 2024; Revision accepted June 24, 2024; Crosschecked

**Abstract:** Binary analysis, as an important foundational technology, provides support for numerous applications in the fields of software engineering and security research. With the continuous expansion of software scale and the complex evolution of software architecture, binary analysis technology is facing new challenges. To break through existing bottlenecks, researchers have applied artificial intelligence (AI) technology to the understanding and analysis of binary code. The core lies in characterizing binary code, i.e., how to use intelligent methods to generate representation vectors containing semantic information for binary code, and apply them to multiple downstream tasks of binary analysis. In this paper, we provide a comprehensive survey of recent advances in binary code representation technology, and introduce the workflow of existing related research in two parts: binary code feature selection methods and binary code feature embedding methods. The feature selection section mainly includes two parts: definition and classification of features; and feature construction. Firstly, the abstract definition and classification of features are systematically explained, and secondly, the process of constructing specific representations of features is introduced in detail. In the feature embedding section, based on the different intelligent semantic understanding models used, the embedding methods are classified into four categories based on the usage of text embedding models and graph embedding models. Finally, we summarize the overall development of existing research and provide prospects for some potential research directions related to binary code representation technology.



**Key words:** Binary analysis; Binary code representation; Binary code feature selection; Binary code feature embedding

https://doi.org/10.1631/FITEE.2400088 **CLC number:**

## 1 Introduction

With the continuous progress of information technology, software systems have penetrated into many aspects of human lives, from daily communication and entertainment to study and work. In the process of development, the functional and performance requirements for software are constantly increasing, and the software landscape itself is becoming increasingly complex (Lu et al., 2023). In some scenarios, such as the process of vulnerability detection for Internet of Things (IoT) firmware, researchers do not have access to the source code; so, the binary code needs to be analyzed. Due to the lack of high-level semantic information (such as data types and structures) in the source code, a large number of code changes can be introduced in the compilation process to generate different codes, which leads to certain problems and challenges in the process of binary code analysis. Therefore, how to analyze and detect binary code efficiently and accurately has become a research hotspot in the field of software engineering and cybersecurity.

The object of binary analysis technology is binary program or software. Using different strategies to understand and analyze the binary code in the program can assist many applications in the field of cybersecurity. In the field of cybersecurity, analysis

‡ Corresponding author

and processing such as function boundary identification, function call convention recovery and control flow graph recovery of binary programs can help researchers to carry out reverse analysis well, thus supporting the vulnerability mining of software with no source (Yu et al., 2022) and assisting the optimization of binary code, code review, and automated testing, finally improving software quality.

With the rapid development of AI, researchers have begun to use machine learning models more often in binary analysis tasks to intelligently analyze and understand binary program (Haq and Caballero, 2021). The model uses massive code data to learn; it can automatically extract the feature patterns, so as to understand the code semantics and complete various binary analysis tasks set by researchers. In the process of application of an AI model, the binary code representation technology is particularly critical, and the representation converts the binary code into a form that is easy to process and analyze, so as to provide strong support for subsequent analysis and detection.

Binary code representation technology refers to the use of artificial intelligent models to process binary code as input, understand the program semantic information contained therein, and generate representation in vector form (Li et al., 2021b). Mapping binary code into numerical vector space can facilitate more practical and complex binary analysis downstream tasks, such as variable type and name prediction (Allamanis et al., 2020; Chen et al., 2020; David et al., 2020; Nitin et al., 2021; Pei et al., 2021; Zhang et al., 2021; Chen et al., 2022), function information recovery (Gao et al., 2021; Jin et al., 2022; Kim et al., 2023; Patrick-Evans et al., 2023; Yu et al., 2023), binary code similarity detection (Gao et al., 2021; Duan et al., 2020; Zhang et al., 2020; Li et al., 2021b; Liu, 2022; Peng et al., 2021; Ullah and Oh, 2022; Yang et al., 2022, 2023a; Ahn et al., 2022; Guo et al., 2022; Wang et al., 2022; Kim et al., 2022; Pei et al., 2023; Kim et al., 2023; Xu et al., 2023; Luo et al., 2023; Yang et al., 2023b; Wang et al., 2023a; Qasem et al., 2023), malicious code detection (David et al., 2016a; Chu et al., 2020; Vasan et al., 2020a,b; Xi-Dong1 et al., 2020; Qiao et al., 2021; Giaretta et al., 2021; Pham et al., 2021; Li et al., 2021a; Chaganti et al., 2022; Jinwei et al., 2023; Wu et al., 2023; Qixu et al., 2023; Yumlembam et al., 2023), and so on.

The purpose of this paper is to investigate the binary code representation technology, introduce the current relevant research progress, so as to provide reference for the code representation related research, and promote the representation-based intelligent binary analysis research. The binary code representation technology can be divided into two parts: binary code feature selection and binary code feature embedding. The binary code feature selection part systematically classifies and summarizes the abstract binary code features commonly selected by researchers and the concrete feature representations frequently constructed and used. In the section on binary code feature embedding, the model methods commonly used to characterize binary code in the existing research are introduced comprehensively. Finally, the trend and development of current research are summarized, and some potential research directions related to binary code representation are prospected.



## 2 Background

This chapter introduces the basic concepts in this paper. Firstly, we introduce the two concepts of binary code (which is the object of representation), and the concept of representation technology, and then introduces the binary code representation technology. Then, we give an overview of which the downstream tasks of binary analysis to which the technique can be applied to, and how to realize achieve specific applications.

### 2.1 Binary code and representation learning

In the field of software program analysis, binary code serves as a representation of a computer program, often referred to as the machine code. Binary code is essentially a set of instructions and data encoded in binary numbers, directly executable by the computer hardware. By contrast, source code represents programs written in high-level programming languages that are readable by humans. Before execution, the source code typically requires compilation and other transformations into binary code. However, reversing the process to recover the source code from binary code is challenging; typically, only a semantically similar pseudo-code can be reconstructed, as illustrated in Fig. 1. Consequently, in scenarios where in the source code is unavailable or

inaccessible, contemporary research centers on the analysis of binary code.

Representation technology serves as a pivotal technology in the field of AI. It encodes and represents data in such a way that computers can comprehend and process it (Yoshua et al., 2012; Xu, 2020). Representation learning refers to the implementation of representation techniques in a learnable manner, such as training on massive data using deep neural network (DNN) models, to achieve automatic extraction of data features. The input entails the original data of the representation object, while the output is the fixed-dimension representation vector mapped to the numerical vector space. These features can be utilized to accomplish classification, clustering, recommendation, and other downstream tasks.

Binary code representation refers to the process of extracting the features of the binary program and creating an embedded representation in the numerical vector space based on these features, thus obtaining the representation vector that can be numerically calculated, as shown in Fig. 2. Due to the ease of manipulating the vector, it better supports subsequent specific downstream tasks in software engineering and cybersecurity fields. The overall binary code representation learning process can be divided into two steps:

1. Binary code feature selection: By means of program analysis or by relying on expert knowledge, useful features are classified and constructed from the original code data. Features can be divided into two levels, namely abstract semantics and concrete feature representation forms. Different feature representation forms often contain different abstract semantics or multiple abstractions. The selected features can be transformed and optimized, and more prominent representations can be used to represent the features, thereby improving the quality and effect of subsequent embedded representations.

2. Binary code feature embedding: Using the selected code features as a basis, the input data are represented through embedding, ultimately generating a fixed-dimension representation vector within the unified numerical vector space. Embedding often serves as a "compression" operation, utilizing low-dimensional feature vectors or spaces to describe higher-dimensional feature vectors or spaces containing redundant information. This reduces computational load and mitigates the phenomenon of overfitting. However, some information may be lost during compression, leading to information loss. After embedding the binary code, it is highly unlikely that the original binary code can be reversibly restored. Nevertheless, the generated embedding vector provides a more concise representation of the binary code characteristics.

### 2.2 Application representation in downstream tasks

Downstream tasks are practical and specific applications of upstream technologies or tools. In the field of machine learning, a downstream task refers to the actual classification, detection, and other tasks that are applied to a pre-trained model with parameters trained on generic tasks and data under the pre-training and fine-tuning paradigm. It is essential to utilize the specific data of the downstream task in application and adjust the pre-trained model according to the task. Subsequently, the model is retrained to fine-tune the model parameters, enabling a better application of the pre-trained model to the downstream task.

The downstream tasks in binary analysis comprise a range of specific undertakings that leverage binary code analysis technology. These encompass tasks such as variable type and name prediction, function information recovery, binary code similarity detection, and malicious code detection, among others. The utilization of binary code representation technology offers a more nuanced depiction of these downstream tasks within the context of machine learning applications. For instance, they can be framed as classification or regression tasks, requiring only a designated amount of task-relevant data. By fine-tuning the parameters of a pre-trained model for various tasks, superior outcomes can be attained. The workflow is depicted in Fig. 3.

To simplify tasks such as binary analysis for classification, regression, and clustering, we can leverage the pre-training fine-tuning paradigm for machine learning models. This approach relies on mature code representation models to design the output layer tailored to the specific task. Subsequently, we utilize task-specific data for training, as exemplified in the Task-1 and Task-2 processes depicted on the left side of Fig. 3. During the training phase, it is possible to freeze the parameters in the binary code representation model as the backbone network. We

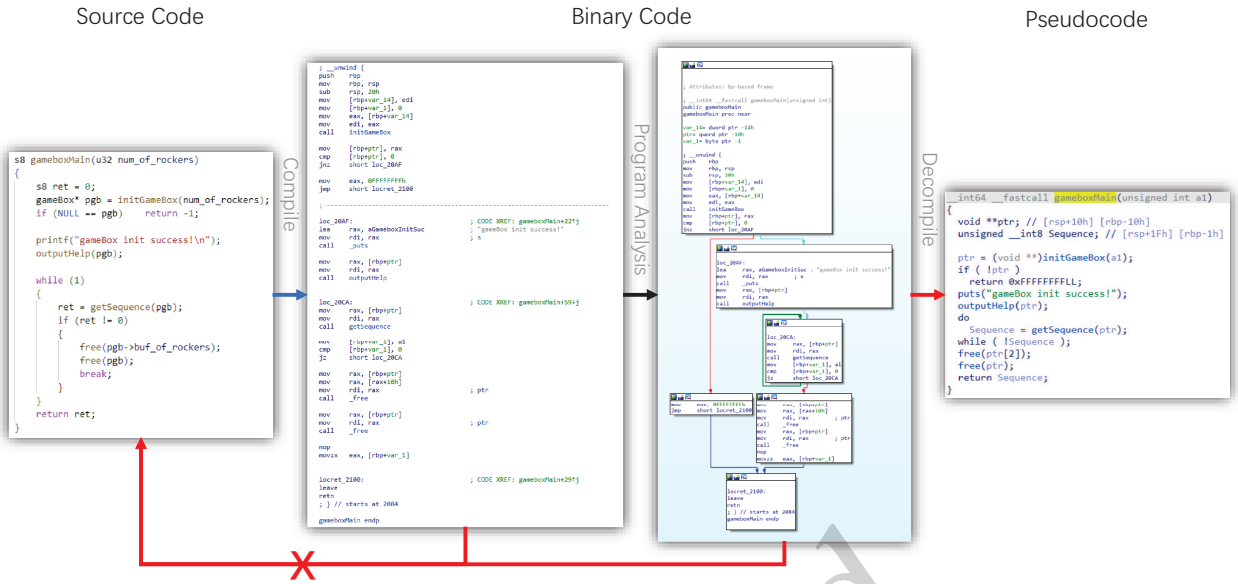


**Fig. 1 Conversion between source, binary and pseudo code.**

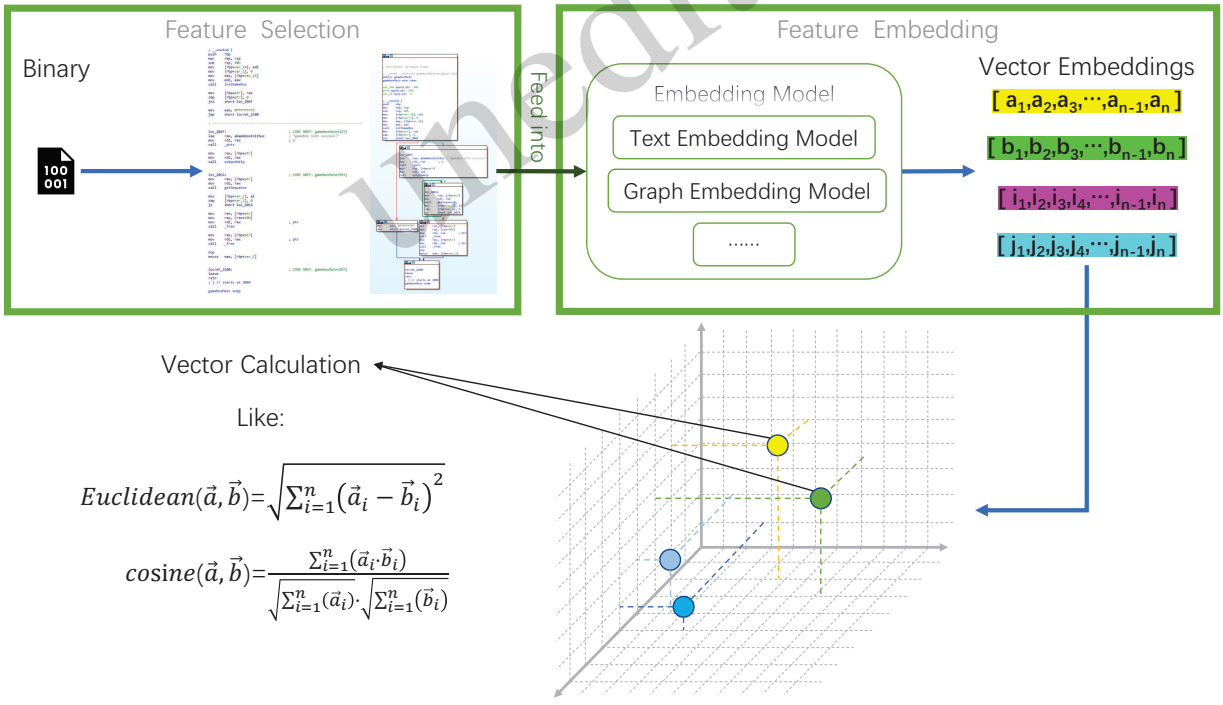


**Fig. 2 Binary code representation procedure.**

can then use a limited amount of task-specific data to conduct supervised or unsupervised training on the task-specific output layer. This approach enhances the comprehension of the representation model for binary code, ultimately leading to superior performance on corresponding tasks. This paradigm is

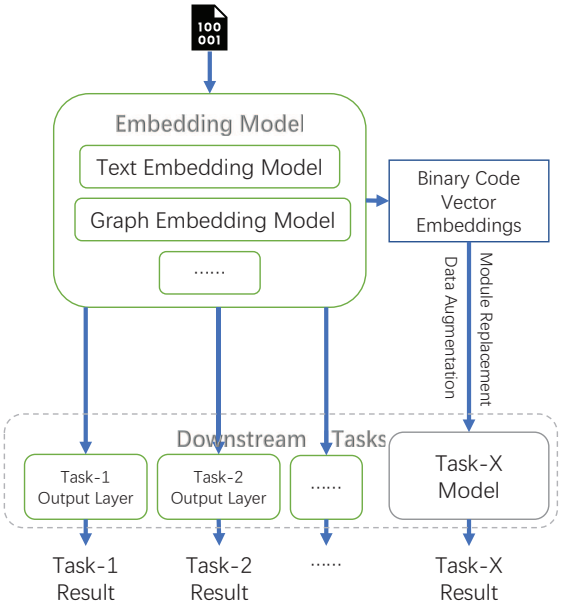


**Fig. 3 Workflow of using binary code representation in binary analysis downstream tasks.**

capable of performing any functions that fall under the categories of classification, regression, and clustering. The quality of the upstream embedding representation greatly affects the performance of these tasks. This is because the pre-trained embedding model provides a foundational understanding of semantics, and the weights in the output layer of the downstream task are refined with the assistance of the embedding representation and task-specific data.

Another paradigm is to integrate embedding models into off-the-shelf methods without altering the original workflow or routine, in order to address complex domain-specific tasks in the binary analysis field. The common approach is to replace certain inside critical components with binary code representation, as exemplified in "Task-X" on the right side of Fig. 3. As an example, researchers replace the manually designed graph node attributes used in Gemini (Xu et al., 2017) with the instruction embedding generated by the model PalmTree (Li et al., 2021b) for better binary code similarity detection. This kind of approach is followed by the development of a task-specific model, which is described in detail in Section 4.3. In this manner, the upstream embedding models will produce representations that contain more comprehensive and nuanced information, and the enriched information allows for more accurate and sophisticated downstream processing, enhancing the overall performance and capabilities of the system in tasks. Another example is for approaches containing code embedding models, researchers can upgrade the internal models to an advanced binary code embedding model, such as substituting Bidirectional Encoder Representation from Transformers (BERT) for one-hot, Word2Vec, long- and short-term memory networks (LSTM), and other code encoding models in binary analysis methods such as EKLAVYA (Chua et al., 2017) and DeepVSA (Guo et al., 2019). This will enhance performance due to the increased information content and improved semantic understanding capabilities.

## 3 Binary code feature selection

The crux of binary code representation lies in selecting its semantic features. Binary code can yield diverse abstract semantics, which must be displayed and utilized in conjunction with specific concrete representation forms. Consequently, it is imperative to devise distinct or concrete representation forms tailored to different types of semantic features or to devise unified forms by integrating various types of features. A high-quality feature representation can effectively convey the semantics of binary code, thus facilitating the comprehension and learning abilities of intelligent models with regard to these semantics. This chapter delves into the frequently selected abstract semantic features and the conventional feature representations utilized in current methods.

### 3.1 Binary code feature classification

Binary code encompasses a range of features at various levels, and different representation methods focus on capturing distinct levels of code characteristics. The abstract features selected in existing methods fall into the following categories, with different approaches considering one or several of these for their investigations.

#### 3.1.1 Syntax feature

Existing research first converts binary code into the form of instruction code, which is a textual representation. This allows for analysis from the perspectives of syntax. Syntactic analysis (parsing) is one of the most important technologies in natural language processing (NLP). It aims to analyze the

grammatical structure, components, and dependencies between words in a sentence. It is the basis for parsing, semantic interpretation, dialog understanding, and machine translation applications. During analysis, specific features are extracted to reflect the lexical and grammatical features of the binary code.

#### 3.1.2 Statistical feature

Through reverse engineering and program analysis, binary code can be expressed in various granular forms, including instructions, basic blocks, functions, and component modules. A complete functional component module comprises one or more functions, each function consists of one or more basic blocks; a basic block contains several instructions. Consequently, a binary code can be viewed as a whole that is composed of individual elements under a specific granularity division. Researchers can utilize mathematical statistics to describe the overall usage of binary code based on individual characteristics, such as quantitative characteristics and attribute characteristics.

#### 3.1.3 Textual feature

In reality, binary code primarily comprises machine instructions and binary-based data resources. Machine instructions carry out distinct operations on computer equipment and data resources, making the semantic features of instructions crucial for binary code representation. As machine instructions can be translated into comprehensible assembly language instructions, binary code can also be processed and understood in textual form. This allows for the characterization of semantics.

#### 3.1.4 Control flow feature

When machine instructions are executed in a computer, they are fetched and executed one by one by the central processor. The order of instruction fetching does not always align with the order of instructions in the binary code. The execution of the next instruction is contingent upon the operation of the previous instruction and the memory state of the program. For instance, conditional jump instructions dictate the selection of the next instruction based on register values, indicating that instructions are not executed sequentially. The actual order and manner of program instruction execution is referred to as control flow, and this layer of semantic characteristics is intricately linked to the operational logic of binary code.

#### 3.1.5 Data flow feature

The data flow feature in binary code uses the concept of data flow from the realm of program analysis, referring to the flow of data throughout the program code. This encompasses both the flow sequence and the specific data processing steps. The data flow feature offers another layer of description regarding the operational logic of binary code.

#### 3.1.6 Symbolic feature

Machine instructions in binary code can be symbolically represented in mathematical logic, which involves treating part of the data as symbolic values and expressing the instructions as expressions composed of these symbolic values and logical symbols. By utilizing symbolization, the instructions in binary code can undergo logical reasoning through symbolic operations, enabling formal reasoning and proofs. As a more rigorous representation, symbolic feature can capture data dependence and condition checks, and describe binary code from the perspective of mathematical logic.

#### 3.1.7 Function call feature

Most existing methods for characterizing function-level code focus on the internal features of the function, making it challenging to distinguish between functions with similar code structures. Function call information, such as the characteristics of the caller and the callee function, offers a higher-level perspective, potentially enhancing representation accuracy in specific scenarios.

#### 3.1.8 Data resource

In addition to machine instructions, data resources also play a crucial role in binary code. These include the string information and numerical type information stored in the data segment of the program, as well as the key values present in the code segment and that control the running logic. Sometimes, string resources contain helpful information text, allowing the functionality of the program to be described in natural language form. Numerical type resources, on the other hand, reflect the config-

uration of program code and running logic, making them suitable for use as semantic features.

#### 3.1.9 Dynamic feature

The semantics described earlier belong to the static semantic features of binary code. However, the execution semantics of binary code are intricately linked to the actual running state of the machine and cannot be solely determined through static analysis. Therefore, capturing semantics during the dynamic execution of the code can provide a more realistic representation of the code functional characteristics.

### 3.2 Binary code feature construction

The information from a single type of abstract feature is relatively limited, so existing research focuses on the combined characterization of multiple types of abstract features. For different types of abstract semantic features that need to be considered, different concrete representations, such as text sequences, adjacency matrices, multidimensional vectors, etc.and so on, are needed to reasonably display semantics on the one hand, and facilitate the process of learning and understanding semantics by intelligent models to learn and understand semantics on the other hand. This section classifies the feature representations used in each research method, and introduces the methods of constructing specific feature representations commonly used in existing methods.

#### 3.2.1 Text sequence

For the semantic features of instruction text, we can utilize text sequence representations, which are primarily categorized into three groups: assembly instruction text sequences, intermediate language (IL) text sequences, and custom instruction sequences.

The assembly instruction text sequence refers to the sequential assembly instructions obtained from disassembling binary code, and the complete textual sequence is regarded as the representation of the instruction text, and then it can be characterized by NLP related methods. Typical methods used in this domain include: SAFE (Massarelli et al., 2019a), InnerEye (Zuo et al., 2018), Asm2Vec (Ding et al., 2019), cross-arch-instr-model (Redmond et al., 2018), MIRROR (Zhang et al., 2020), PalmTree (Li et al., 2021b), BinDiffNN (Ullah and Oh, 2022), Binshot (Ahn et al., 2022), etc.

The IL (or IR) text sequence refers to the conversion of machine instructions in binary code into an intermediate language representation, which is then treated as a representation of the instruction text. Since assembly instructions vary significantly across different architectures (such as x86, x64, ARM, AARCH64, MIPS, etc.), IL such as LLVM IR used by the compiler LLVM (Lattner and Adve, 2004), VEX used by the dynamic analysis framework Valgrind (Nethercote and Seward, 2007), Microcode in IDA (Hex-rays, 2024) and ESIL in Radare2 (Radare2, 2024), etc. can unify the representation of binary code across different architectures into a unified syntax environment, to a certain extent, reducing the semantic interference caused by different architectures in the representation process. Representative methods include: OSCAR (Peng et al., 2021), BinFinder (Qasem et al., 2023), etc.

The custom instruction sequence involves modifying the original text sequence by building upon the sequential assembly instruction text sequence and IL text sequence. This is achieved through data flow-based slicing and arrangement of instructions based on their importance, emphasizing specific levels of semantics. To better represent data flow characteristics, researchers have devised a custom instruction sequence called Strands, which comprises all the instructions calculated with respect to a specific variable within a code block. The representative methods include Esh (David et al., 2016b), GitZ (David et al., 2017), Zeek (Shalev and Partush, 2018), FirmUP (David et al., 2018), etc. MKIS (Li et al., 2020) uses fuzzing to build the input of dynamic execution of the program, considers the sequence of API calls together with the key values that remain unchanged in multiple executions, constructs the new execution sequence of the function as a representation, and subsquent matching based on the key values. Trace information and symbolic constraint semantics in dynamic execution can also be used for instruction sequence construction. For example, researchers define the continuous part of short Trace in the actual execution of code as Tracelet, which is expressed in combination with symbolic constraints, thus reflecting the dynamic execution and symbolic characteristics in the text. Representative methods include TRACY (David and Yahav, 2014), BinGo (Chandramohan et al., 2016), sem2vec (Wang

et al., 2023a), etc. Trex (Pei et al., 2023) also extracts information from Trace in dynamic execution, deletes the text of instructions that have not been executed, and uses real data in dynamic execution to replace the memory address and register in the original instruction text to enrich the semantics, thus reflecting the dynamic functional features. jTrans (Wang et al., 2022) specifically describes the control flow feature of the instruction, and replaces the target address of the jump instruction with the token serial number of the target instruction. DiEmph (Xu et al., 2023) uses a dual-pronged approach for importance assessment. It bases the importance of corresponding instructions on classification performance by monitoring changes in embedded representation results during instruction modification. Additionally, it defines the importance of instructions in semantic aspects through program analysis. Based on these two types of importance, the method screens out and removes instructions that are prone to causing distribution bias from the dataset.

#### 3.2.2 Numerical statistical feature

Statistical features are the description of binary code from the perspective of mathematical statistics, including numerical-type and attribute-type features. Due to the widespread use of machine learning models, researchers tend to extract numerical type features and leverage mature machine learning models for analysis based on massive data sources.

Numerical statistical features embody statistical characteristics in numerical format. These features encompass counts of specific types of instructions, representations of basic blocks, and even the number of instructions throughout the execution of code, which reflect dynamic characteristics. Table 1 presents a list of commonly utilized numerical statistical features.

The methods that use numerical statistical vectors as the ultimate feature representation include Patchecko (Sun et al., 2020) and TikNib (Kim et al., 2023). Given the diverse forms of feature representation, some techniques combine numerical statistical features with other concrete representations. For instance, numerical statistical features of a code unit can serve as node attributes in a topology diagram, enabling the comprehensive representation of all code units in this diagram format (see the section on "Topology graph structure"). Examples of such methods include Genius (Feng et al., 2016), Gemini (Xu et al., 2017), VulSeeker (Gao et al., 2018a), VulSeeker-Pro (Gao et al., 2018b), and BinSeeker (Gao et al., 2021). Among these, Genius and Gemini share the same eight features shown on the left side of Table 2, while VulSeeker, VulSeeker-Pro, and BinSeeker share the eight features displayed on the right side of Table 2.

#### 3.2.3 Topology graph structure

In the field of program analysis, the semantics of program control flow, data flow and function invocations are primarily represented through topological graphs such as control flow graph, data flow graph, function call graph and program dependency graph. There are three types of topology structure used in the existing methods: attribute control flow graph, abstract syntax tree, and custom topology structure.

The Attribute control flow graph (ACFG) builds upon the control flow graph obtained through program analysis. It integrates instruction text embedding vectors and numerical statistical features to enrich the semantic information of graph nodes. Initially, as proposed by researchers, the ACFG used numerical statistical features as node attributes of the control flow graph, resulting in the topology graph structure depicted in Fig. 4. Works such as Genius and Gemini are representative examples that utilize this feature representation. Subsequently, some researchers enhanced the ACFG by incorporating data flow graph information, reflecting data flow characteristics. This enhancement involved adding edges from the data flow graph to enrich the edge information of the control flow graph. Notable works in this vein include VulSeeker, VulSeeker-Pro, and BinSeeker. With the advent of advanced text embedding techniques, various embedding methods for program instructions have emerged. The embedding outcomes of all instructions within a basic block can also be represented as properties of that basic block. Methods such as GMN (Li et al., 2019), GraphEmb (Massarelli et al., 2019b), OrderMatters (Yu et al., 2020), DeepBinDiff (Duan et al., 2020), Codee (Yang et al., 2022), and VulHawk (Luo et al., 2023) are representative examples in this domain.

Abstract syntax trees (ASTs), a prevalent representation in source-level program analysis, are infrequently used in binary program analysis due to the

**Table 1 Numerical statistical characteristics commonly used in functions**

| Feature description (static) | Feature description (dynamic) |
|---|---|
| Number of constants value in the function | Number of binary-defined function calls during execution |
| Number of strings in the function | Minimal stack depth during execution |
| Number of instruction in the function | Maximal stack depth during execution |
| Size of local variables in bytes | Average stack depth during execution |
| Various flags associated with a function, e.g., FUNC NORET, FUNC FAR. | Standard deviation stack depth during execution |
| Number of import functions | Number of executed instruction |
| Number of code references from this function | Number of executed unique instruction |
| Number of function calls from this function | Number of call instruction |
| Size of the function | Number of arithmetic instruction |
| Minimal number of instruction for basic block | Number of branch instruction |
| Maximal number of instruction for basic block | Number of load instruction |
| Average number of instruction for basic block | Number of store instruction |
| Standard deviation of number of instruction for basic block | Maximal number of frequency of the executed same branch instruction |
| Minimal size of basic block | Maximal number of frequency of the executed same arithmetic instruction |
| Maximal size of basic block | Number of accessing heap memory space |
| Average size of basic block | Number of accessing stack memory space |
| Standard deviation of size of basic block | Number of accessing library memory space |
| Number of basic block for each function | Number of accessing anonymous mapping memory space |
| Number of edge of among basic blocks for each function | Number of accessing others part memory space |
| Function cyclomatic complexity = Edges - Nodes + 2 | Number of library function calls during execution |
| Normal block type of function basic block | Number of system calls during execution |
| Block ends with indirect jump | |
| Return block type of function basic block | |
| Conditional return block type of function basic block | |
| Noreturn block type of function basic block | |
| External noreturn block (does not belong to the function) | |
| External normal block type of function basic block | |
| Block passes execution past the function end | |
| Minimal number of call instruction of each basic block | |
| Maximal number of call instruction of each basic block | |
| Average number of call instruction of each basic block | |
| Standard deviation of call instruction of basic block | |
| Total number of call instruction of the function | |
| Minimal number of arithmetic instruction of each basic block | |
| Maximal number of arithmetic instruction of each basic block | |
| Average number of arithmetic instruction of each basic block | |
| Standard deviation of arithmetic instruction of each basic block | |
| Total number of arithmetic instruction of the function | |
| Minimal number of arithmetic FP instruction of each basic block | |
| Maximal number of arithmetic FP instruction of each basic block | |
| Average number of arithmetic FP instruction of each basic block | |
| Standard deviation number of arithmetic FP instruction of each basic block | |
| Total number of arithmetic FP instruction of the function | |
| Minimal number of betweeness centrality | |
| Maximal number of betweeness centrality | |
| Average number of betweeness centrality | |
| Standard deviation number of betweeness centrality | |
| How many node the betweeness centrality is zero | |



**Table 2 Function statistical features often used as node attribution**

| Genius and Gemini | VulSeeker and BinSeeker |
|---|---|
| String constants | No. of stack operation instructions |
| Numeric constants | No. of arithmetic instructions |
| No. of transfer Instructions | No. of logical instructions |
| No. of calls | No. of comparative instructions |
| No. of instructions | No. of library function calls |
| No. of arithmetic Instructions | No. of unconditional jump instructions |
| No. of offspring | No. of conditional jump instructions |
| Betweenness | No. of generic instructions |

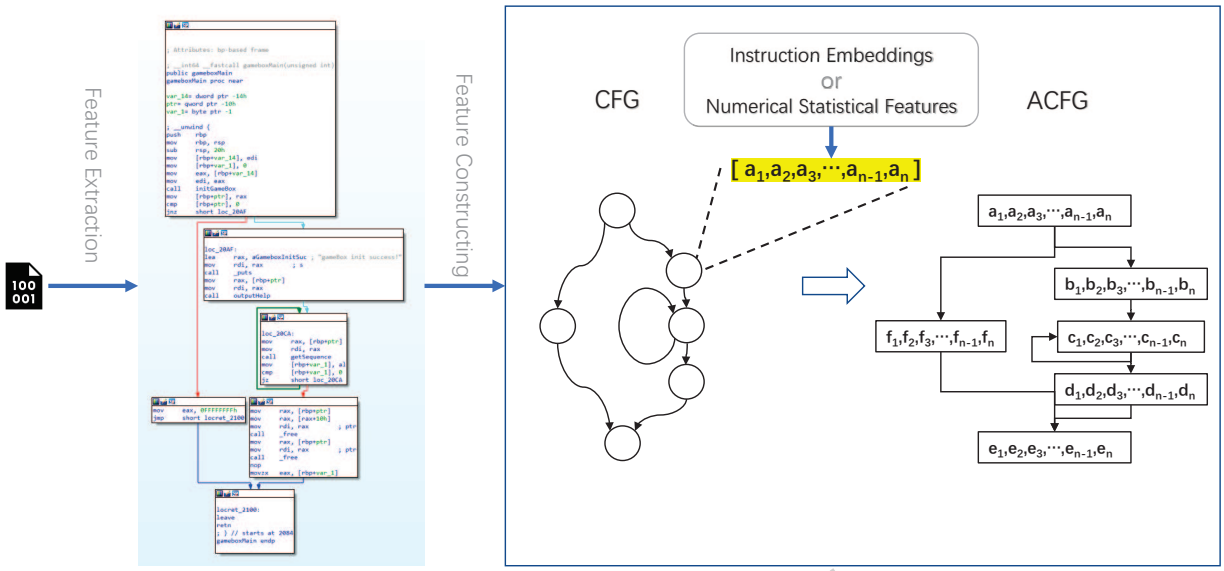


**Fig. 4 ACFG construction process.**

challenges in recovering syntactic structures, such as variable type information, from binary code. Consequently, researchers often opt to leverage tools that elevate binary code to the IL level, upon which they construct ASTs. Notably, ASTs can yield more consistent representation outcomes than control flow diagrams in cross-architecture scenarios. Representative methods that capitalize on this approach include Asteria (Yang et al., 2023a) and Asteria-Pro (Yang et al., 2023b).

In addition to the commonly used representation forms in program analysis such as control flow graph (CFG), data flow graph (DFG) and AST, existing methods further enhance semantic features by designing customized topology diagram structures. For instance, $\alpha$Diff (Liu et al., 2018) represents binary code bytes as "0" and "1" according to a 100x100x1 dimension. If the dimension is insufficient, it pads with zeros, and if it exceeds, it truncates. This arrangement of the original byte stream follows a grid topology, resulting in a "01" matrix resembling an image pixel map. This representation form, often utilized in malicious code detection (Jinwei et al., 2023), lacks interpretability. On the other hand, BCSD (Liu, 2022) defines four types of key instruction: function call, comparison, return, and memory storage. Using symbolic execution, it generates symbolic values at these key instructions, and constructs a novel graph structure for nodes. BMM (Guo et al., 2022) redefines the data flow graph, whereby nodes represent instructions determining the data flow direction, and edges signify data dependencies between instructions, reflecting data flow characteristics. XBA (Kim et al., 2022) extends nodes based on the control flow diagram, abstracting binary files into custom binary disassembly graphs (BDGs). These graphs have nodes representing basic blocks, external functions, and strings, with relational edges indicating relationships such as jumps, calls, and accesses. This approach encompasses control flow and richer binary file context information. BinUSE (Wang et al., 2023b) applies the under-constrained symbolic execution (USE) technique (Ramos and Engler, 2015) to identify the external function call points of a function and uses them as nodes to generate new subgraphs as features.

### 3.2.4 Function call sequence

The semantic of function call features can be effectively represented through the sequence of function calls, a feature frequently utilized in malicious code detection. Specifically, the list of API function calls obtained via static analysis, along with the actual sequence of API function calls during dynamic execution, can provide comprehensive insights into code functionalities and function call details. Representative methods that apply this representation include $\alpha$Diff, Asteria-Pro, and BinFinder, among

others.

#### 3.2.5 String values and constant values

The data resource information within a binary code predominantly comprises string and numerical constants, which are often treated as supplementary feature representations in current approaches. For instance, numerical constants are directly used as feature vector values, while string values serve as criteria for similarity matching. Methods that incorporate this representation include VulHawk (Luo et al., 2023) and BinFinder, among others.

#### 3.2.6 Dynamic trace data

Characterizing dynamic execution features involves leveraging trace data generated during the dynamic execution of code. These data encompass memory access details, input, and output interactions. To comprehensively test the functionality of the code, it is essential to construct diverse initial inputs and apply techniques such as fuzzing and simulated execution. These methods enable the capture of actual memory access patterns during runtime. Some notable approaches for representing code functionality through input/output pairs are VulSeeker-Pro, BinSeeker, Trex, and sem2vec.

### 3.3 Summary

This chapter systematically classifies and summarizes the abstract semantic features that researchers consider in the feature selection stage, which are mainly divided into nine categories. The specific representation of binary code features is generated by the combination of one or more of the nine features, which can be roughly divided into six types, namely: textual sequence form, numerical statistical feature form, topological graph structure form, function call sequence form, string values and constant values form, and dynamic trace data form. In Table 3, the code features selected by various methods in existing research, as well as the specific representation forms of final use, are summarized. According to the specific representation form of the binary code features extracted, present researches select the appropriate embedding representation methods to achieve binary code representation.

## 4 Binary code feature embedding

With the increasing amount of data in the field of binary analysis, learnable intelligent semantic understanding models can achieve a deeper comprehension of the semantics by leveraging vast datasets, thus generating high-quality embedded representations for superior performance in downstream tasks. Depending on the various forms of feature representation, distinct intelligent representation models can be chosen to capture semantic information in unique manners. This chapter introduces the feature embedding representation component of existing binary code representation methods, focusing on two primary and targeted embedding representation models: method utilizing text-embedding model, method utilizing graph-embedding model, method integrating text embedding model and graph embedding models, other embedding methods.

### 4.1 Methods utilizing text-embedding models

Given the textual and sequential nature of code, text embedding models commonly used in NLP can be employed to comprehend code text from a linguistic perspective. Before feeding manually constructed text features into NLP models, tokenization and normalization must be performed as the preliminary step for encoding, and to address the out-of-vocabulary (OOV) problem. Using the instruction “mov rax, qword [rsp+0x58]” as an example, we will examine how various methods handle it. A simple method used by Asm2Vec (Ding et al., 2019) is to split an instruction into its opcode and operands, thus dividing it into “mov” and “rax, qword [rsp+0x58]”. However, the vocabulary size may explode due to the expansive value space of constants and literals in operands, which can result in encountering unknown tokens during inference that were not in the training dataâĂŤthis is known as the OOV problem. PalmTree (Li et al., 2021b) adopts a more fine-grained tokenization, splitting the instruction into “mov”, “rax”, “qword”, “[”, “rsp”, “+”, “0x58”, and “]”. To alleviate the OOV problem caused by strings and constant numbers, PalmTree uses the special token “[str]” to replace strings and “[addr]” for large constants. If the constants are relatively small, they may contain critical information and should be preserved as individual tokens. In this way, the large value space ($2^{32}$ possible tokens for four bytes) can be

**Table 3 Selection of code feature in existing research works**

| Name | Binary code feature category | | | | | | | | | Form of characterization |
|---|---|---|---|---|---|---|---|---|---|---|
| | Syntax feature | Textual feature | Statistical feature | Control flow feature | Data flow eature | Symbolic feature | Function call feature | Data resource | Dynamic feature | |
| **Genius(Feng et al., 2016)** | | | X | X | | | | | | ACFG |
| **Gemini(Xu et al., 2017)** | | | X | X | | | | | | ACFG |
| **$\alpha$Diff(Liu et al., 2018)** | | | | | | | X | | | Raw Bytes |
| **VulSeeker(Gao et al., 2018a)** | | | X | X | X | | | | | LSFG |
| **VulSeeker-pro(Gao et al., 2018b)** | | | X | X | X | | | | X | LSFG, Dynamic Trace |
| **Zeek(Shalev and Partush, 2018)** | | X | | X | X | | | | | Strands |
| **SAFE(Massarelli et al., 2019a)** | | X | | | | | | | | Assembly |
| **GMN(Li et al., 2019)** | | | | X | | | | | | ACFG |
| **InnerEye(Zuo et al., 2018)** | | X | | | | | | | | Assembly |
| **cross-arch-instr-model(Redmond et al., 2018)** | | X | | | | | | | | Assembly |
| **GraphEmb(Massarelli et al., 2019b)** | | X | | X | | | | | | ACFG |
| **Asm2Vec(Ding et al., 2019)** | | X | | | | | | | | Assembly |
| **OrderMatter(Yu et al., 2020)** | | X | | X | | | | | | ACFG, CFG |
| **Patchecko(Sun et al., 2020)** | | | X | | | | | | X | Numerical Statistical Feature |
| **MKIS(Li et al., 2020)** | | X | | X | | | | | X | Customized Instruction Sequence |
| **DeepBinDiff(Duan et al., 2020)** | | X | | X | | | | | | ACFG |
| **MIRROR(Zhang et al., 2020)** | | X | | | | | | | | Assembly |
| **BCSD(Liu, 2022)** | | | | X | | X | | | | Customized Graph Structure |
| **PalmTree(Li et al., 2021b)** | | X | | | X | | | | | Assembly |
| **Asteria(Yang et al., 2023a)** | X | | | | | | X | | | AST |
| **OSCAR(Peng et al., 2021)** | | X | | | | | | | | IL Text |
| **BinDiffNN(Ullah and Oh, 2022)** | | X | | | | | | | | Assembly |
| **Codee(Yang et al., 2022)** | | X | | X | | | | | | Assembly, ACFG |
| **BinSeeker(Gao et al., 2021)** | | | X | X | X | | | | X | LSFG, Dynamic Trace |
| **Binshot(Ahn et al., 2022)** | | X | | | | | | | | Assembly |
| **BMM(Guo et al., 2022)** | | X | | X | X | | | | | CFG, CG, DFG |
| **jTrans(Wang et al., 2022)** | | X | | X | | | | | | Customized Instruction Sequence |
| **XBA(Kim et al., 2022)** | | X | | X | | | X | X | | Customized Graph Structure |
| **Trex(Pei et al., 2023)** | | X | | | X | | | | X | Customized Instruction Sequence |
| **TikNib(Kim et al., 2023)** | | | X | | | | | | | Numerical Statistical Features |
| **DiEmph(Xu et al., 2023)** | | X | | | | | | | | Customized Instruction Sequence |
| **VulHawk(Luo et al., 2023)** | | X | | X | | | X | X | | ACFG |
| **Asteria-Pro(Yang et al., 2023b)** | X | | | | | | X | X | | AST |
| **sem2vec(Wang et al., 2023a)** | | X | | X | | X | X | | X | Dynamic Trace, Function Call Sequence |
| **BinFinder(Qasem et al., 2023)** | | X | | | | | X | X | | IL Text, Function Call Sequence, Constants |



mitigated. Other research works use similar methods of tokenization to process assembly instructions or IL sequences, with the granularity that falls between the two above-mentioned approaches.

In the current research landscape, models such as Word2Vec (Mikolov et al., 2013), recurrent neural network (RNN), and LSTM have been predominantly used for the embedded representation of instruction text formal features in their initial stages. However, following the introduction of the Transformer (Vaswani et al., 2017) model structure, network architectures constructed using Transformer Encoder and Transformer Decoder have become increasingly prevalent in characterizing the features of binary code.

SAFE (Massarelli et al., 2019a) uses the skip-gram pattern of the Word2Vec model to predict the context of a given input word for instruction semantic learning and representation, thus building the instruction embedding model i2v. In the i2v model, each function is regarded as a document and each assembly instruction as a word; then, the function in-

struction semantics are learned based on the perception of context, and the instruction representation is generated. SAFE then takes an embedded vector sequence of all instructions in a function as the input and uses an RNN combined with an an attention mechanism to complete a fixed-length vectorization of the function-level code. This routine illustrates the basic workflow of using embedding models for function-level binary code representation.

InnerEye (Zuo et al., 2018) also uses the skip-gram pattern of Word2Vec model to learn instruction semantics. After generating instruction representation, InnerEye uses the LSTM model to obtain the embedded representation vector of basic blocks. Based on this, it can represent code snippets (multiple basic blocks) to improve the flexibility of the representation granularity, and embed sequential input more effectively using an LSTM model.

The cross-arch-instr-model (Redmond et al., 2018) is extended based on the continuous bag of words (CBOW) mode of the Word2Vec model, i.e., it predicts the words in the specified position given the context, and uses the joint learning method to semantically align the instructions in x86 and ARM architecture, so as to learn the semantic correlation between instructions in different architectures. Thus, an instruction level embedding vector is generated. It first uses linear instruction alignment and instruction opcode classification to pair instructions under different architectures one by one with the same semantics, and then uses CBOW model to predict the corresponding instructions under another architecture according to the context instructions of the corresponding instructions under one architecture and the corresponding instructions under another architecture. It provides a new idea of alignment between cross-architecture instructions, although the definition of instruction semantics alignment lacks authoritative support.

Asm2Vec (Ding et al., 2019) uses the NLP model of PV-DM to learn assembly instructions, which regards each assembly instruction as a sentence, and separates operand and opcode as a word unit (token). Asm2Vec has a much finer tokenization granularity than SAFE and InnerEye, as mentioned before.

BinDiffNN (Ullah and Oh, 2022) directly deals with assembly instructions, and designs an attention-based embedding neural network (ABENN), a twin model based on the attention mechanism, which processes the encoded assembly instructions using a fully connected network and add attention to the different tokens in the instruction. ABENN classifies whether functions are completely similar or only partially similar, with an attention mechanism that helps ignore small structural changes due to basic block rearrangement and focus on real changes in code semantics. The embedding model with attention mechanism is the core design of this method, and it defines scenarios for completely similar and partially similar cases, which is realistic in practical application.

MIRROR (Zhang et al., 2020) is based on the idea of NMT to translate x86-ARM inter-architecture instructions, so as to characterize inter-architecture instructions. MIRROR maps x86 and ARM instruction sets to the same vector space according to instruction semantics, so that semantic representation can be learned and used for similarity detection. The model is built based on transformer, and the training stage is divided into two parts to improve the model effect: first, the x86 instruction embedding model is trained separately, and then, when the ARM instruction embedding model is trained, the x86 instruction embedding model is fine-tuned to achieve the representation in the same vector space. This method aligns the semantics of cross-architecture binaries entirely at the embedding representation level using NMT, rather than at the instruction level using definitions such as the cross-arch-instr-model, thereby granting the model greater ability and potential to comprehend instruction semantics.

OSCAR (Peng et al., 2021) converts source code and binary files into the LLVM IR form and builds a model based on the transformer structure for semantic understanding and representation of LLVM IR, so that similarity detection can be carried out among source code, binary code, and binary and source codes respectively. Trex (Pei et al., 2023) also uses transformer to build hierarchical models for characterization of the instruction input combined with data flow information in dynamic execution. DiEmph (Xu et al., 2023) improves the methods found in existing studies which use Transformer and other complex deep learning models, considers the importance of instructions for downstream tasks, deletes unimportant instructions in the training data, and fine-tunes the model to solve the prob-

lem of instruction distribution bias. This indicates a trend toward using the transformer, which demonstrates superior performance in understanding semantics. Another trend involves embedding based on IL text and customized sequences, which contain more abundant information to characterize binaries.

PalmTree (Li et al., 2021b) uses the Bidirectional encoder representations from transformers (BERT) (Devlin et al., 2019) pre-training model in NLP to learn the representation of assembly instructions by modifying word segmentation methods and pre-training tasks to fit x86 assembly instructions. According to the instruction sequence and data dependency, two pre-training tasks are designed, which can effectively take into account the logic relationship between the assembled instructions in the program and the data dependency relationship between the instructions, make full use of massive data to extract and characterize the corresponding semantic features, and perform data fine-tuning and evaluation on a variety of binary analysis downstream tasks. The experiment proves that, based on the massive binary file data that can be collected on the Internet at present, using the paradigm of pre-training and fine tuning can obtain good application results. PalmTree has introduced BERT to binary code representation, and it has cleverly focused on designing pre-training tasks to enhance binary code embedding.

jTrans (Wang et al., 2022) builds a BERT-like model based on the transformer structure and uses masked language modeling (MLM) to represent and learn skip relationships between instructions in binary code, i.e., mask the destination address parts of skip instructions. The model then makes predictions about the tokens at this location. The model obtained through pre-training can finally obtain a higher success rate of jump relationship prediction, and after fine-tuning the model on the task of similarity detection, the generated representation vector can achieve a higher accuracy of similarity comparison and CVE vulnerability detection. It skillfully incorporates the learning of instruction jump semantics into the pre-training task, thereby achieving the integration of structural information with textual information.

BinShot (Ahn et al., 2022) used BERT to build Siamese twin neural network for similarity detection. The combination of the basic model and twin neural network to achieve the structure itself is not new, but it focuses on the design of distance calculation function and loss function and finally chooses to use the weighted variance distance calculation method and binary cross-entropy loss function. They are used to replace the cosine distance and contrast loss function, respectively, and good results are obtained in the experiment even if it is simple in design.

### 4.2 Methods utilizing graph-embedding models

As researchers pay increasing attention to the structural features of binary code, numerous methods are being applied to embed the topological formal features. According to semantic information such as control flow and data flow, the feature representation in the form of corresponding topological graph is constructed. Then, the graph-embedding model based on node-embedding technology and graph neural network is used to embed the feature and generate the feature vector. Node-embedding methods typically sample a specific number of nodes and primarily focus on the format of node attributes, whereas graph neural networks require the input to be standardized in terms of both node and edge formats. The input format for the corresponding graph-embedding models typically consists of an adjacency matrix or the triple form representing edge relationships.

Gemini (Xu et al., 2017) uses the graph embedding model structure2vec (Dai et al., 2016) to generate the vector form embedding representations for ACFG of binary code. structure2vec models the structured data as pairwise Markov random field (Lafferty et al., 2001), and then finds the fixed point iteration formula in the variational inference process. It is then converted into a fixed point iteration of embedding, which is often used to embed the code graph structure. Based on Gemini, VulSeeker (Gao et al., 2018a) extracted data dependencies, constructed DFG, combined it into ACFG, and finally formed labeled semantic flow graph (LSFG). With reference to structure2vec, a deep neural network (DNN) was constructed to carry out the graph structure embedding characterization. VulSeeker-pro (Gao et al., 2018b) and BinSeeker (Gao et al., 2021) also use custom DNN for embedded characterization of LSFG. Compared with VulSeeker, BinSeeker improves the representation of dynamic exe-

cution features and is conducive to further analysis. This series of research has employed methods ranging from standard node embedding to graph neural networks, progressively enhancing the ability to represent topological structures.

Graph matching network (GMN) (Li et al., 2019) uses the cross-subgraph attention mechanism between different graphs to determine whether the two input function graph structures are similar from end to end, thus guiding the generation of embedding vectors for two graphs, rather than using a single graph embedding model to generate embedding vectors for each piece of code. After large-scale experimental evaluation by the Cisco Talos team (Marcelli et al., 2022), it is found that the generated representation quality is outstanding, and it can achieve high accuracy when applied to binary code function ACFG similarity calculation. This method focuses solely on the graph matching link, and it has achieved a good effect in similarity detection. Incorporating binary code-related knowledge may further improve the representation task.

Since both Asteria (Yang et al., 2023a) and Asteria-Pro (Yang et al., 2023b) focus on the characterization of the feature representation of an abstract syntax tree AST, both use the Tree-LSTM (Tai et al., 2015) model which is consistent with the tree structure of AST for vectorized representation. Each unit in Tree-LSTM is similar to LSTM, but the update of vector and state depends on the state of all the sub units related to it, and there are multiple forgetting gates, which can selectively obtain information from the sub nodes. Experimental results also show the effectiveness of this model to represent AST. This work combines the AST form of binary code with the Tree-LSTM model well, while as another novel representation of binary code topology besides CFG, AST differs from CFG in both structure and semantics, which can affect the application of downstream tasks.

XBA (Kim et al., 2022) uses the graph convolutional network (GCN) (Kipf and Welling, 2016) to learn node semantics to generate representations, and aligns the nodes of the graph according to node semantics for the custom graph structure: binary disassembly graphs (BDGs). Good results are achieved in the downstream tasks of graph matching. The customized topological structure representation of binary code has an effect similar to that of the customized text sequence, namely enriching it with additional semantic information.

### 4.3 Methods integrating text-embedding and graph-embedding models

Recent research has begun to integrate the aforementioned two types of embedding representation methods. Here is the primary workflow. Initially, a text-embedding model is used to characterize the features of binary code instruction text and generate corresponding representation vectors. Subsequently, these vectors are seamlessly integrated into the topological structure features via node attributes, such as the construction of ACFG. Finally, a graph-embedding model is utilized to generate representation vectors for the topological structure feature representation, serving as the ultimate representation of the binary code. This entire process is graphically depicted in Fig. 5.

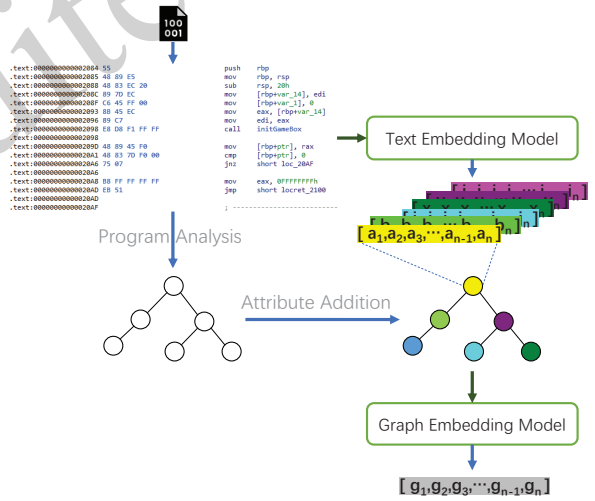


**Fig. 5 Combination of text-embedding and graph-embedding models.**

GraphEmb(Massarelli et al., 2019b) obtains the embedded representation vector of each assembly instruction on the basis of the i2v instruction embedding model proposed by SAFE(Massarelli et al., 2019a), and aggregate all the instruction embedding vectors in a basic block into the representation of a basic block. Then, using the graph embedding method proposed in Gemini, the graph structure is used as the main body of function-level representation, and node attributes are replaced with corresponding basic block representations to generate

ACFG. Finally, structure2vec is used for function embedding characterization of ACFG. This method adheres to the basic paradigm, and its performance is highly dependent on the performance of the two embedded models within it.

OrderMatters (Yu et al., 2020) uses a scheme similar to GraphEmb, first using the BERT model for instruction embedding and building ACFG and then using message passing neural network (MPNN) (Gilmer et al., 2017) to aggregate the properties of the basic block nodes in the graph structure. At the same time, it cleverly combines the representation of the adjacency matrix, and uses CNNs to represent the structure of the grid topology graph, so as to ignore the influence of node order. Finally, the representation vectors generated by the two models are spliced, and multi-layer perception (MLP) is used to learn the representation that is more conducive to the downstream task. This method characterizes the order of the adjacency matrix from the perspective of convolution, offering a novel approach. Although it predominantly captures the structural attributes of the topological graph, the model as a whole starts to exhibit a multi-modal nature.

DeepBinDiff (Duan et al., 2020) uses the CBOW mode of the Word2Vec model similar to that in the cross-arch-instr-model (Redmond et al., 2018) for instruction text embedding, treating each instruction as a word. Each basic block containing several instructions is treated as a sentence, thus performing basic block vector embedding. After that, TADW (Text-associated DeepWalk algorithm) (Yang et al., 2015), a graph node-embedding method, is used to conduct node sampling on inter-procedural control flow diagrams (inter-procedural CFG, ICFG). Finally, a sequence of embedded representation vectors of the sampled base blocks is used as the representation of each function. Because it uses node embedding techniques, the method is capable of representing binary code at the basic block level, function level, and even across the entire program, similar to InnerEye (Zuo et al., 2018). However, the accuracy of the representation heavily depends on the quality of the instruction-level embedding.

Codee (Yang et al., 2022) uses the graph embedding algorithm node2vecWalk (Grover and Leskovec, 2016) for random walk sampling of instruction sequences, and uses skid-gram combined with negative sampling technology for embedding characterization of operands and opcodes. The embedded representation vector of all instructions contained in a basic block is directly combined with the feature vector, which is represented together with CFG as the basic block semantic information. Based on accelerated attributed network embedding (AANE) (Huang et al.) and large-scale information network embedding (LINE) (Tang et al., 2015), The loss function is solved based on the alternating direction method of multipliers (ADMM) algorithm to generate the basic block representation. Finally, the function is characterized in the form of tensor, and tensor singular value decomposition (tSVD) is used for tensor compression, which can capture the main features in the feature vector of the function and ignore the noise information. In this method, binary codes at the instruction level, basic block level, and function level are represented and transformed level by level using tensor data structures. Tensor decomposition techniques are then applied to compress the features, endowing the method with a robust mathematical foundation. Furthermore, this introduces a novel perspective on how to adapt techniques from the AI domain to this specific field.

VulHawk (Luo et al., 2023) uses RoBERTa (Liu et al., 2019) to generate instruction text embedding vectors, which are also combined into ACFG, and uses GCN to obtain semantic representation of function graph structure. For the obtained results, feature vectors are further extracted from basic blocks, string constants, import functions and other information, and a self-constructed feed-forward neural network is utilized to characterize and compute the similarity for the next step. This method also follows the basic paradigm, but is aided by binary provenance information and correspondingly uses different adaptor (Houlsby et al., 2019) weights for representation, thus achieving leading advantages on multiple metrics.

In BMM (Guo et al., 2022), it is proposed to merge the semantics of control flow graph, data flow graph and call graph, use BERT to embed both instructions and operands at the same time, and use gated graph neural network (GGNN) (Li et al., 2015) to merge and represent the generated ACFG graph structure, so as to solve binary analysis tasks at the program and function level together using one model. sem2vec (Wang et al., 2023a) uses USE to extract symbolic constraints for tracelets extracted

from binary code (David and Yahav, 2014), and uses RoBERTa to train masked language models to calculate the embedded representation of output information from symbolic execution. Finally, a graph neural network combining GGNN and Set2Set (Vinyals et al., 2015) is used to compute the embedding vector and aggregate all the results from Tracelets as a function embedding representation at the CFG level. These methods reflect the trend that, by enriching the representation of text-level and topology graph-level features and by applying more advanced models, their performance can be significantly improved.

### 4.4 Other methods

Apart from text embedding and graph embedding models, there exist alternative methods for embedding representations of binary code. However, these approaches are not considered mainstream, and they are briefly introduced in the following part.

Genius (Feng et al., 2016) performs spectral clustering on ACFG of binary code, and uses cluster results as coding basis to make a codebook, so as to represent binary code with cluster category coding of ACFG. The clustering method is used to encode categories. Only when the number of categories of scene data is sufficient and the category distribution of the training and test sets is consistent can good results be achieved, so there are limitations in scene generalization. Therefore, the subsequent methods (see Section 4.2 and 4.3 for details) mostly use graph neural networks to embed and represent the graph structure for ACFG, to improve the quality of characterization and expand the application scenarios.

$\alpha$Diff (Liu et al., 2018) is based on the machine code in the form of the original byte stream, which is arranged into a gridded topological structure representation, similar to the pixel map of the image, which can then be characterized using CNNs to generate embedded vectors. This kind of method, which represents binary code as image form and then uses image processing domain model for representation processing, is more popular in the field of malicious code detection, and experiments have proved that it can achieve good results in downstream tasks, but whether the feature level is associated with the real semantics of binary code is still uncertain.

Zeek (Shalev and Partush, 2018) divides the binary code into multiple strands (David et al., 2016b) of instruction sequences and hashes each strand to form features in the form of vectors. The neural network composed of an input layer, two fully connected networks as the hidden layer, and a Softmax output layer is used to learn and represent the feature vectors. The semantics of binary code are precisely characterized through the use of meticulously constructed features (strands). However, the hashing phase reduces the information through compression, and the embedding model is not as well-designed as it could be. By applying more powerful text embedding models, these aspects can be enhanced to boost overall performance.

Patchecko (Sun et al., 2020) combines static and dynamic features to characterize binary code, in the form of 48 numerical statistical features extracted from static analysis and 21 numerical statistical features extracted from dynamic execution, including the maximum, minimum, and average stack frame depth, the number of various instructions executed and the number of memory slots accessed at different locations. After concatenating the extracted static and dynamic statistical features, the neural network obtained by stacking six linear layers is input to carry out feature learning and finally generate the representation vector. The dynamic analysis used in this method can offer more accurate semantic information regarding program execution, although it may incur some performance overhead.The idea of using the patch comparison method to assist vulnerability detection has also been adopted in many studies.

Through experimental tests, BinFinder (Qasem et al., 2023) has located a series of binary program features that are not sensitive to techniques such as code obfuscation and compiler optimization, including numerical features such as the number of target functions called and the number of callers, and list type features such as library function call sequence and special VEX IR instruction sequence. The twinning neural network is formed by the three-layer sensing set, and the feature characterization and similarity detection are carried out. Selecting features that are resilient to code obfuscation and compilation optimization is the correct approach to achieve feature selection in complex compilation scenarios. However, advanced representation embedding techniques should also be applied to achieve good representation effects.

### 4.5 Summary

According to the specific representation of features extracted from binary code in Section 3, we can choose an appropriate embedding model for feature embedding. In this section, we introduce four categories of embedding methods, which are classified based on the two primary embedding models: text embedding and graph embedding. These categories comprise the basic text-embedding model-based method, the graph-embedding model-based method, the method integrating text-embedding and graph-embedding models, and other types of embedding representation methods. The specific feature-embedding techniques utilized by each method are outlined in Table 4.

From an overarching perspective, existing research has consistently pursued cutting-edge methods in AI, successfully adapting and transferring these techniques to suitable scenarios, thereby achieving impressive practical outcomes.

## 5 Prospects

The research on binary code representation technology has reached a considerable level of maturity, effectively supporting downstream tasks in binary program analysis. During the feature selection and extraction phase, the exploration of feature definition, classification, and construction has attained a state of stability. Furthermore, as embedded representation models in the field of AI undergo rapid advancement, the study of binary code representation is progressively embracing state-of-the-art models. These contemporary models are being refined to incorporate enhancements that are intimately linked to code semantics, thereby being more suitable for binary code representation tasks.

On the basis of the existing work, aligning with prominent research areas in recent years, the potential research directions are prospected as follows.

### 5.1 Binary code representation based on multi-modal fusion

Multi-modal fusion technology is a multi-modal data processing method, which aims to fuse data of different modalities to obtain more comprehensive, accurate and reliable information. In the real world, three-dimensional entities can be described by various modalities such as text, image and sound, while binary codes can also be described from the perspectives of text sequence, topological graph structure and other forms suck as dynamic execution trace. The features of different representation forms can be regarded as different modalities. Therefore, multi-modal fusion technology can be effectively applied in the domain of multi-modality, particularly in the context of binary code representation. By leveraging this, we can generate more comprehensive representation vectors that encapsulate rich feature information.

A significant aspect of research in the multi-modal domain centers on devising effective interaction modes among diverse modalities. When it comes to the representation of binary code, it is crucial to consider the relationships among various features. Additionally, exploring whether there exists semantic overlap and complementarity among these features is equally important. By doing so, we can identify the most effective interaction mode and achieve seamless integration of these features, ultimately enhancing their collective utility.

### 5.2 Large language model based binary code understanding and representation

Since the overnight sensation of ChatGPT in early 2023, the evolution of large models has been rapid and remarkable. Today, both domestic and foreign developers have embraced open-source and closed-source large models, and the corresponding products have gradually made their way into personal mobile phones, personal computers, and other terminal devices to offer services. The significant increase in model parameters not only enables more efficient utilization of massive datasets but also plays a pivotal role in data intelligence. Furthermore, emerging capabilities such as logical inference and profound understanding, including grokking (Liu et al., 2022; Power et al., 2022), have expanded the applications of these models into real-world scenarios.

With the support of large-scale computing power, the utilization of large models for code understanding and representation, drawing upon extensive binary code data, represents a promising research avenue with immense growth potential. At present, the predominant application scenarios for large models include natural language text comprehension and

**Table 4 Embedding methods of code features in existing research works**

| Name | Feature embedding methods | | Others |
|---|---|---|---|
| | Text embedding model | Graph embedding model | |
| Genius(Feng et al., 2016) | | | Spectral clustering |
| Gemini(Xu et al., 2017) | | structure2vec | |
| $\alpha$Diff(Liu et al., 2018) | | CNN | |
| VulSeeker(Gao et al., 2018a) | | Customized DNN | |
| VulSeeker-pro(Gao et al., 2018b) | | Customized DNN | |
| Zeek(Shalev and Partush, 2018) | | | Customized layers |
| SAFE(Massarelli et al., 2019a) | Word2Vec(Skip-Gram) | | |
| GMN(Li et al., 2019) | | Graph Matching Network | |
| InnerEye(Zuo et al., 2018) | Word2Vec(Skip-Gram) | | |
| cross-arch-instr-model(Redmond et al., 2018) | Word2Vec(CBOW) | | |
| GraphEmb(Massarelli et al., 2019b) | Word2Vec(Skip-Gram) | structure2vec | |
| Asm2Vec(Ding et al., 2019) | PV-DM | | |
| OrderMatter(Yu et al., 2020) | BERT | CNN + MPNN | MLP |
| Patchecko(Sun et al., 2020) | | | Customized layers |
| DeepBinDiff(Duan et al., 2020) | Word2Vec(CBOW) | TADW | |
| MIRROR(Zhang et al., 2020) | Transformer | | |
| PalmTree(Li et al., 2021b) | BERT | | |
| Asteria(Yang et al., 2023a) | | Tree-LSTM | |
| OSCAR(Peng et al., 2021) | Transformer | | |
| BinDiffNN(Ullah and Oh, 2022) | ABENN | | |
| Codee(Yang et al., 2022) | Word2Vec(Skip-Gram) | AANE + LINE | |
| BinSeeker(Gao et al., 2021) | | Customized DNN | |
| Binshot(Ahn et al., 2022) | BERT | | |
| BMM(Guo et al., 2022) | BERT | GGNN | |
| jTrans(Wang et al., 2022) | BERT | | |
| XBA(Kim et al., 2022) | | GCN | |
| Trex(Pei et al., 2023) | Transformer | | |
| DiEmph(Xu et al., 2023) | Transformer | | |
| VulHawk(Luo et al., 2023) | RoBERTa | GCN | |
| Asteria-Pro(Yang et al., 2023b) | | Tree-LSTM | |
| sem2vec(Wang et al., 2023a) | RoBERTa | GCN | |
| BinFinder(Qasem et al., 2023) | | | Customized layers |



generation, the advancement of high-level programming language code capabilities, and the accomplishment of image matching and mutual generation in conjunction with multi-modal technology. Notably, there exists a notable absence of relevant work in the training and application of large-scale binary code data corpora. Therefore, exploring and researching in this direction offers the potential to assess the limitations and boundaries of large models for binary code understanding.

### 5.3 Binary code representation combining source code and annotation information

The binary code solely contains the functional semantic information of the program, while the source code and corresponding natural language annotation information matching with the binary code can provide richer and higher-level information such as design considerations for analysis (Zhang et al., 2022). The source code and annotation information are also easily accessible data in the era of big data.

Previous studies have been conducted to match the source code with the binary code compiled based on it, as well as achieve similarity detection by: mining the common features between the two code forms (Guo et al., 2023), converting uniformly to binary code or IL level (Ji et al., 2021; Peng et al., 2021), using multi-modal fusion and cross-modal matching (**?**) and other methods. When representing binary code, it is beneficial to integrate its corresponding source code for a more holistic understanding. This approach allows for the description of binary code from two distinct levels: functional characteristics and design ideas. By considering both, a more com-

prehensive representation of the binary code can be achieved.

### 5.4 Interpretable binary code representation methods

Since binary code representation technology needs to apply intelligent semantic understanding models, there is always a need for theoretical proof in the interpretability of models, and the interpretability of models is also an important research direction. Research on interpretability involves the significance of features, the internal structure of the model, the interpretability of the model output, etc. Central questions include the following: Which features are genuinely effective? Which model structure better facilitates feature understanding? And, does the vector space representation generated by the model carry meaningful implications? Addressing these questions is integral to enhancing the overall comprehensibility and effectivenesss of binary code representation within intelligent semantic understanding frameworks.

### 5.5 Research on downstream tasks of binary analysis

Binary analysis encompasses a vast array of tasks, and numerous software engineering studies have resorted to AI techniques, particularly representation learning, as viable solutions (Hou et al., 2023; Wang et al., 2023c). At present, similarity detection of binary code stands out as a domain that seamlessly integrates multiple representation techniques. By extracting and characterizing the features of binary code, the representation vector can directly calculate the similarity degree through the distance calculation formula in vector space, so as to measure the similarity at the semantic or functional level. Furthermore, there are many studies using machine learning models to directly implement end-to-end tasks, such as variable name and type recovery, control flow diagram improvement, instruction disassembly, etc., which can be carried out.

## 6 Conclusion

This paper provides a comprehensive survey of recent advances in binary code representation technology. First, we introduce the concept of binary code representation and discuss its correlation with downstream tasks in binary analysis. Subsequently, categorize the existing research workflow into two fundamental components: binary code feature selection methods and binary code feature embedding methods. For feature selection, we systematically explain the definition and classification of features, and detail the methodology for constructing representations of these features. For feature embedding, we classify the embedding methods into four distinct categories based on the usage of text embedding models and graph embedding models. Finally, we summarize the overall development of existing research and provide prospects for some potential research directions.

### Contributors

Taiyan WANG designed the research. Taiyan WANG, and Qingsong XIE drafted the manuscript. Qingsong XIE processed the data. Lu YU helped organize the manuscript. Zulie PAN and Min ZHANG revised and finalized the paper.

### Conflict of interest

All the authors declare that they have no conflict of interest.

### References

Ahn S, Ahn S, Koo H, et al., 2022. Practical binary code similarity detection with bert-based transferable similarity learning. Proceedings of the 38th Annual Computer Security Applications Conference, New York, NY, USA, p.361âĂŞ374, https://doi.org/10.1145/3564625.3567975

Allamanis M, Barr ET, Ducousso S, et al., 2020. Typilus: neural type hints. Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. https://doi.org/10.1145/3385412.3385997

Chaganti R, Ravi V, Pham TD, 2022. Deep learning based cross architecture internet of things malware detection and classification. *Computers & Security*, 120:102779.

Chandramohan M, Xue Y, Xu Z, et al., 2016. Bingo: cross-architecture cross-os binary search. Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016, p.678-689. https://doi.org/10.1145/2950290.2950350

Chen L, He Z, Mao B, 2020. Cati: Context-assisted type inference from stripped binaries. 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), p.88-98. https://doi.org/10.1109/DSN48063.2020.00028

Chen Q, Lacomis J, Schwartz EJ, et al., 2022. Augmenting decompiler output with learned variable names and types. 31st USENIX Security Symposium (USENIX Security 22), p.4327-4343.

Chu Q, Liu G, Zhu X, 2020. Visualization feature and cnn based homology classification of malicious code. *Chinese Journal of Electronics*, 29(1):154-160.

Chua ZL, Shen S, Saxena P, et al., 2017. Neural nets can learn function type signatures from binaries. 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, p.99-116, https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/chua

Dai H, Dai B, Song L, 2016. Discriminative embeddings of latent variable models for structured data. Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48, p.2702âĂŞ2711.

David Y, Yahav E, 2014. Tracelet-based code search in executables. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, p.349âĂŞ360. https://doi.org/10.1145/2594291.2594343

David Y, Partush N, Yahav E, 2016a. Statistical similarity of binaries. *SIGPLAN Not*, 51(6):266âĂŞ280. https://doi.org/10.1145/2980983.2908126

David Y, Partush N, Yahav E, 2016b. Statistical similarity of binaries. Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, p.266âĂŞ280. https://doi.org/10.1145/2908080.2908126

David Y, Partush N, Yahav E, 2017. Similarity of binaries through re-optimization. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, p.79âĂŞ94. https://doi.org/10.1145/3062341.3062387

David Y, Partush N, Yahav E, 2018. Firmup: Precise static detection of common vulnerabilities in firmware. *SIGPLAN Not*, 53(2):392âĂŞ404. https://doi.org/10.1145/3296957.3177157

David Y, Alon U, Yahav E, 2020. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1âĂŞ28. https://doi.org/10.1145/3428293

Devlin J, Chang M, Lee K, et al., 2019. BERT: pre-training of deep bidirectional transformers for language understanding. Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers), p.4171-4186.

Ding SHH, Fung BCM, Charland P, 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. 2019 IEEE Symposium on Security and Privacy (SP), p.472-489. https://doi.org/10.1109/SP.2019.00003

Duan Y, Li X, Wang J, et al., 2020. Deepbindiff: Learning program-wide code representations for binary diffing. *Proceedings 2020 Network and Distributed System Security Symposium*, . https://api.semanticscholar.org/CorpusID:195063875

Feng Q, Zhou R, Xu C, et al., 2016. Scalable graph-based bug search for firmware images. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, New York, NY, USA, p.480âĂŞ491, https://doi.org/10.1145/2976749.2978370

Gao H, Cheng S, Xue Y, et al., 2021. A lightweight framework for function name reassignment based on large-scale stripped binaries. Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, New York, NY, USA, p.607âĂŞ619. https://doi.org/10.1145/3460319.3464804

Gao J, Yang X, Fu Y, et al., 2018a. Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary. Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, New York, NY, USA, p.896âĂŞ899, https://doi.org/10.1145/3238147.3240480

Gao J, Yang X, Fu Y, et al., 2018b. Vulseeker-pro: enhanced semantic learning based binary vulnerability seeker with emulation. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, NY, USA, p.803âĂŞ808, https://doi.org/10.1145/3236024.3275524

Gao J, Jiang Y, Liu Z, et al., 2021. Semantic learning and emulation based cross-platform binary vulnerability seeker. *IEEE Transactions on Software Engineering*, 47(11):2575-2589. https://doi.org/10.1109/TSE.2019.2956932

Giaretta L, Lekssays A, Carminati B, et al., 2021. Limnet: Early-stage detection of iot botnets with lightweight memory networks. Computer Security – ESORICS 2021, Cham, p.605-625.

Gilmer J, Schoenholz SS, Riley PF, et al., 2017. Neural message passing for quantum chemistry. International conference on machine learning, p.1263-1272.

Grover A, Leskovec J, 2016. node2vec: Scalable feature learning for networks. Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining, p.855-864.

Guo W, Mu D, Xing X, et al., 2019. DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis. 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, p.1787-1804, https://www.usenix.org/conference/usenixsecurity19/presentation/guo

Guo X, Cai R, Yin X, et al., 2023. Searching open-source vulnerability function based on software modularization. *Applied Sciences*, 13(2). https://doi.org/10.3390/app13020701

Guo Y, Li P, Luo Y, et al., 2022. Exploring gnn based program embedding technologies for binary related tasks. Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, New York, NY, USA, p.366âĂŞ377. https://doi.org/10.1145/3524610.3527900

Haq IU, Caballero J, 2021. A survey of binary code similarity. *ACM Comput Surv*, 54(3). https://doi.org/10.1145/3446371

Hex-rays, 2024. Ida pro, .

Hou X, Zhao Y, Liu Y, et al., 2023. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:230810620*, .

Houlsby N, Giurgiu A, Jastrzebski S, et al., 2019. Parameter-efficient transfer learning for NLP. Proceedings of the 36th International Conference on Machine Learning, 97:2790-2799, https://proceedings.mlr.press/v97/houlsby19a.html

Huang X, Li J, Hu X. Accelerated Attributed Network Embedding. In: Proceedings of the 2017 SIAM International Conference on Data Mining (SDM). https://doi.org/10.1137/1.9781611974973.71

Ji Y, Cui L, Huang HH, 2021. Buggraph: Differentiating source-binary code similarity with graph triplet-loss network. Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security, New York, NY, USA, p.702âĂŞ715. https://doi.org/10.1145/3433210.3437533

Jin X, Pei K, Won JY, et al., 2022. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, New York, NY, USA, p.1631âĂŞ1645. https://doi.org/10.1145/3548606.3560612

Jinwei W, Zhengjia C, Xue X, et al., 2023. Review of malware detection and classification visualization techniques. *Chinese Journal of Network and Information Security*, 9(5):1. https://doi.org/10.11959/j.issn.2096-109x.2023064

Kim D, Kim E, Cha S, et al., 2023. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Transactions on Software Engineering*, 49(04):1661-1682. https://doi.org/10.1109/TSE.2022.3187689

Kim G, Hong S, Franz M, et al., 2022. Improving cross-platform binary analysis using representation learning via graph alignment. Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, New York, NY, USA, p.151âĂŞ163. https://doi.org/10.1145/3533767.3534383

Kim H, Bak J, Cho K, et al., 2023. A transformer-based function symbol name inference model from an assembly language for binary reversing. Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, New York, NY, USA, p.951âĂŞ965. https://doi.org/10.1145/3579856.3582823

Kipf T, Welling M, 2016. Semi-supervised classification with graph convolutional networks. *ArXiv*, abs/1609.02907. https://api.semanticscholar.org/CorpusID:3144218

Lafferty JD, McCallum A, Pereira FCN, 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. Proceedings of the Eighteenth International Conference on Machine Learning, San Francisco, CA, USA, p.282âĂŞ289.

Lattner C, Adve V, 2004. Llvm: a compilation framework for lifelong program analysis & transformation. International Symposium on Code Generation and Optimization, 2004 CGO 2004, p.75-86. https://doi.org/10.1109/CGO.2004.1281665

Li C, Shen G, Sun W, 2021a. Cross-architecture intemet-of-things malware detection based on graph neural network. 2021 International Joint Conference on Neural Networks (IJCNN), p.1-7. https://doi.org/10.1109/IJCNN52387.2021.9533500

Li X, Qu Y, Yin H, 2021b. Palmtree: Learning an assembly language model for instruction embedding. Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, New York, NY, USA, p.3236âĂŞ3251, https://doi.org/10.1145/3460120.3484587

Li Y, Wang B, Hu B, 2020. Semantically find similar binary codes with mixed key instruction sequence. *Information and Software Technology*, 125:106320. https://doi.org/10.1016/j.infsof.2020.106320

Li Y, Tarlow D, Brockschmidt M, et al., 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:151105493*, .

Li Y, Gu C, Dullien T, et al., 2019. Graph matching networks for learning the similarity of graph structured objects. Proceedings of the 36th International Conference on Machine Learning, 97:3835-3845, https://proceedings.mlr.press/v97/li19d.html

Liu B, Huo W, Zhang C, et al., 2018. *alpha*diff: cross-version binary code similarity detection with dnn. Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, New York, NY, USA, p.667âĂŞ678, https://doi.org/10.1145/3238147.3238199

Liu Y, Ott M, Goyal N, et al., 2019. Roberta: A robustly optimized bert pretraining approach. *ArXiv*, abs/1907.11692. https://api.semanticscholar.org/CorpusID:198953378

Liu Z, 2022. Binary code similarity detection. Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering, p.1056âĂŞ1060, https://doi.org/10.1109/ASE51524.2021.9678518

Liu Z, Kitouni O, Nolte NS, et al., 2022. Towards understanding grokking: An effective theory of representation learning. Advances in Neural Information Processing Systems, 35:34651-34663.

Lu Y, Yu L, Zhao J, 2023. Survey of software vulnerability mining methods based on machine learning. *Information Countermeasure Technology*, 2(2).

Luo Z, Wang P, Wang B, et al., 2023. Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search, https://api.semanticscholar.org/CorpusID:257501992

Marcelli A, Graziano M, Ugarte-Pedrero X, et al., 2022. How machine learning is solving the binary function similarity problem. 31st USENIX Security Symposium (USENIX Security 22), Boston, MA, p.2099-2116.

Massarelli L, Di Luna GA, Petroni F, et al., 2019a. Safe: Self-attentive function embeddings for binary similarity. Detection of Intrusions and Malware, and Vulnerability Assessment, Cham, p.309-329.

Massarelli L, Luna GAD, Petroni F, et al., 2019b. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. *Proceedings 2019 Workshop on Binary Analysis Research*, . https://api.semanticscholar.org/CorpusID:160018518

Mikolov T, Chen K, Corrado G, et al., 2013. Efficient estimation of word representations in vector space. 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings.

Nethercote N, Seward J, 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, p.89âĂŞ100. https://doi.org/10.1145/1250734.1250746

Nitin V, Saieva A, Ray B, et al., 2021. Direct : A transformer-based model for decompiled identifier renaming. NLP4PROG.

Patrick-Evans J, Dannehl M, Kinder J, 2023. Xfl: Naming functions in binaries with extreme multi-label learning. 2023 IEEE Symposium on Security and Privacy (SP), p.2375-2390.

Pei K, Guan J, Broughton M, et al., 2021. Stateformer: fine-grained type recovery from binaries using generative state modeling. Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, NY, USA, p.690âĂŞ702. https://doi.org/10.1145/3468264.3468607

Pei K, Xuan Z, Yang J, et al., 2023. Learning approximate execution semantics from traces for binary function similarity. *IEEE Transactions on Software Engineering*, 49(4):2776-2790. https://doi.org/10.1109/TSE.2022.3231621

Peng D, Zheng S, Li Y, et al., 2021. How could neural networks understand programs? Proceedings of the 38th International Conference on Machine Learning, 139:8476-8486, https://proceedings.mlr.press/v139/peng21b.html

Pham DP, Marion D, Mastio M, et al., 2021. Obfuscation revealed: Leveraging electromagnetic signals for obfuscated malware classification. Proceedings of the 37th Annual Computer Security Applications Conference, New York, NY, USA, p.706âĂŞ719. https://doi.org/10.1145/3485832.3485894

Power A, Burda Y, Edwards H, et al., 2022. Grokking: Generalization beyond overfitting on small algorithmic datasets. *arXiv preprint arXiv:220102177*, .

Qasem A, Debbabi M, Lebel B, et al., 2023. Binary function clone search in the presence of code obfuscation and optimization over multi-cpu architectures. Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, New York, NY, USA, p.443âĂŞ456. https://doi.org/10.1145/3579856.3582818

Qiao Y, Zhang W, Du X, et al., 2021. Malware classification based on multilayer perception and word2vec for iot security. *ACM Trans Internet Technol*, 22(1). https://doi.org/10.1145/3436751

Qixu L, Jiaxi L, Ze J, et al., 2023. Survey of artificial intelligence based iot malware detection. *Journal of Computer Research and Development*, 60(10):2234-2254. https://doi.org/10.7544/issn1000-1239.202330450

Radare2, 2024. radare2, .

Ramos DA, Engler D, 2015. Under-Constrained symbolic execution: Correctness checking for real code. 24th USENIX Security Symposium (USENIX Security 15), Washington, D.C., p.49-64.

Redmond K, Luo L, Zeng Q, 2018. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. *ArXiv*, abs/1812.09652. https://api.semanticscholar.org/CorpusID:56895273

Shalev N, Partush N, 2018. Binary similarity detection using machine learning. Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, New York, NY, USA, p.42âĂŞ47, https://doi.org/10.1145/3264820.3264821

Sun P, Garcia L, Salles-Loustau G, et al., 2020. Hybrid firmware analysis for known mobile and iot security vulnerabilities. 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), p.373-384. https://doi.org/10.1109/DSN48063.2020.00053

Tai KS, Socher R, Manning CD, 2015. Improved semantic representations from tree-structured long short-term memory networks. Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), Beijing, China, p.1556-1566, https://aclanthology.org/P15-1150

Tang J, Qu M, Wang M, et al., 2015. Line: Large-scale information network embedding. Proceedings of the 24th International Conference on World Wide Web, Republic and Canton of Geneva, CHE, p.1067âĂŞ1077, https://doi.org/10.1145/2736277.2741093

Ullah S, Oh H, 2022. Bindiffnn: Learning distributed representation of assembly for robust binary diffing against semantic differences. *IEEE Transactions on Software Engineering*, 48(9):3442-3466. https://doi.org/10.1109/TSE.2021.3093926

Vasan D, Alazab M, Venkatraman S, et al., 2020a. Mthael: Cross-architecture iot malware detection based on neural network advanced ensemble learning. *IEEE Transactions on Computers*, 69(11):1654-1667. https://doi.org/10.1109/TC.2020.3015584

Vasan D, Alazab M, Wassan S, et al., 2020b. Imcfn: Image-based malware classification using fine-tuned convolutional neural network architecture. *Computer Networks*, 171:107138.

Vaswani A, Shazeer N, Parmar N, et al., 2017. Attention is all you need. Proceedings of the 31st International Conference on Neural Information Processing Systems, Red Hook, NY, USA, p.6000âĂŞ6010.

Vinyals O, Bengio S, Kudlur M, 2015. Order matters: Sequence to sequence for sets. *arXiv preprint arXiv:151106391*, .

Wang H, Qu W, Katz G, et al., 2022. jtrans: jump-aware transformer for binary code similarity detection. Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, New York, NY, USA, p.1âĂŞ13, https://doi.org/10.1145/3533767.3534367

Wang H, Ma P, Wang S, et al., 2023a. sem2vec: Semantics-aware assembly tracelet embedding. *ACM Trans Softw Eng Methodol*, 32(4). https://doi.org/10.1145/3569933

Wang H, Ma P, Yuan Y, et al., 2023b. Enhancing dnn-based binary code function search with low-cost equivalence checking. *IEEE Transactions on Software Engineering*, 49(1):226-250. https://doi.org/10.1109/TSE.2022.3149240

Wang J, Huang Y, Chen C, et al., 2023c. Software testing with large language model: Survey, landscape, and vision. *arXiv preprint arXiv:230707221*, .

Wu CY, Ban T, Cheng SM, et al., 2023. Iot malware classification based on reinterpreted function-call graphs. *Comput Secur*, 125(C). https://doi.org/10.1016/j.cose.2022.103060

Xi-Dong1 L, Zhe-Min D, Ye-Kui Q, et al., 2020. Malicious code classification method based on deep forest. *Journal of Software*, 31(5):1454. https://doi.org/10.13328/j.cnki.jos.005660

Xu M, 2020. Understanding graph embedding methods and their applications. *SIAM Rev*, 63:825-853. https://api.semanticscholar.org/CorpusID:229180918

Xu X, Feng S, Ye Y, et al., 2023. Improving binary code similarity transformer models by semantics-driven instruction deemphasis. New York, NY, USA, p.1106âĂŞ1118. https://doi.org/10.1145/3597926.3598121

Xu X, Liu C, Feng Q, et al., 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, New York, NY, USA, p.363âĂŞ376, https://doi.org/10.1145/3133956.3134018

Yang C, Liu Z, Zhao D, et al., 2015. Network representation learning with rich text information. Proceedings of the 24th International Conference on Artificial Intelligence, p.2111âĂŞ2117.

Yang J, Fu C, Liu XY, et al., 2022. Codee: A tensor embedding scheme for binary code search. *IEEE Transactions on Software Engineering*, 48(7):2224-2244. https://doi.org/10.1109/TSE.2021.3056139

Yang S, Dong C, Xiao Y, et al., 2023a. Asteria-pro: Enhancing deep learning-based binary code similarity detection by incorporating domain knowledge. *ACM Trans Softw Eng Methodol*, 33(1). https://doi.org/10.1145/3604611

Yang S, Dong C, Xiao Y, et al., 2023b. Asteria-pro: Enhancing deep learning-based binary code similarity detection by incorporating domain knowledge. *ACM Trans Softw Eng Methodol*, 33(1). https://doi.org/10.1145/3604611

Yoshua B, Aaron CC, Pascal V, 2012. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35:1798-1828. https://api.semanticscholar.org/CorpusID:393948

Yu SY, Achamyeleh YG, Wang C, et al., 2023. Cfg2vec: Hierarchical graph neural network for cross-architectural software reverse engineering. 2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), p.281-291.

Yu YC, Gan ST, Qiu JY, et al., 2022. Binary code similarity analysis and its applications on embedded device firmware vulnerability search. *Journal of Software*, 33(11):4137-4172.

Yu Z, Cao R, Tang Q, et al., 2020. Order matters: Semantic-aware neural networks for binary code similarity detection. Proceedings of the AAAI conference on artificial intelligence, 34(01):1145-1152.

Yumlembam R, Issac B, Jacob SM, et al., 2023. Iot-based android malware detection using graph neural network with adversarial defense. *IEEE Internet of Things Journal*, 10(10):8432-8444. https://doi.org/10.1109/JIOT.2022.3188583

Zhang X, Sun W, Pang J, et al., 2020. Similarity metric method for binary basic blocks of cross-instruction set architecture. Workshop on Binary Analysis Research.

Zhang Y, Huang C, Zhang Y, et al., 2022. Combo: Pre-training representations of binary code using contrastive learning. *ArXiv*, abs/2210.05102. https://api.semanticscholar.org/CorpusID:252816102

Zhang Z, Ye Y, You W, et al., 2021. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. 2021 IEEE Symposium on Security and Privacy (SP), p.813-832. https://doi.org/10.1109/SP40001.2021.00051

Zuo F, Li X, Zhang Z, et al., 2018. Neural machine translation inspired binary code similarity comparison beyond function pairs. *ArXiv*, abs/1808.04706. https://api.semanticscholar.org/CorpusID:52004699