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

Review:

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 Mar. 21, 2025

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 research in two parts, i.e., binary code feature selection methods and binary code feature embedding methods. The feature selection section includes mainly two parts: definition and classification of features, and feature construction. First, the abstract definition and classification of features are systematically explained, and second, 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:** TP312

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 YL 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

‡ Corresponding author
ORCID: Taiyan WANG, https://orcid.org/0009-0007-0533-4404; Min ZHANG, https://orcid.org/0000-0002-6654-7610
© Zhejiang University Press 2025

cybersecurity. In the field of cybersecurity, analysis and processing such as function boundary identification, function call convention recovery, and control flow graph recovery of binary programs can help researchers carry out reverse analysis well, thus supporting the vulnerability mining of software without source (Yu YC et al., 2022) and assisting the optimization of binary code, code review, and automated testing, finally improving software quality.

With the rapid development of artificial intelligence (AI), researchers have begun to use machine learning models more often in binary analysis tasks to intelligently analyze and understand a 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 AI models to process binary code as input, understand the program semantic information contained therein, and generate representation in a vector form (Li XZX et al., 2021). Mapping binary code into the 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 LG et al., 2020; David et al., 2020; Nitin et al., 2021; Pei et al., 2021; Zhang Z et al., 2021; Chen QB et al., 2022), function information recovery (Gao H et al., 2021; Jin et al., 2022; Kim H et al., 2023; Patrick-Evans et al., 2023; Yu SY et al., 2023), binary code similarity detection (Duan et al., 2020; Zhang XC et al., 2020; Gao J et al., 2021; Li XZX et al., 2021; Liu ZA, 2021; Peng et al., 2021; Yang SG et al., 2021, 2023; Ahn et al., 2022; Guo YX et al., 2022; Kim G et al., 2022; Ullah and Oh, 2022; Wang H et al., 2022; Yang J et al., 2022; Kim D et al., 2023; Luo et al., 2023; Pei et al., 2023; Qasem et al., 2023; Wang HJ et al., 2023b; Xu XZ et al., 2023), and malicious code detection (David et al., 2016; Chu et al., 2020; Lu XD et al., 2020; Vasan et al., 2020a, 2020b; Giaretta et al., 2021; Li CF et al., 2021; Pham et al., 2021; Qiao et al., 2021; Chaganti et al., 2022; Liu QX et al., 2023; Wang JW et al., 2023; Wu et al., 2023; Yumlembam et al., 2023).

The purpose of this paper is to investigate the binary code representation technology and introduce the current relevant research progress, so as to provide reference for the code representation 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 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 section introduces the basic concepts in this paper. First, we introduce the two concepts of binary code and the concept of representation technology, and then introduce the binary code representation technology. Then, we give an overview of the downstream tasks in binary analysis to which the technique can be applied, and how to 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 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 them (Bengio et al., 2013; Xu MJ, 2021). 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 used 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, using 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

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

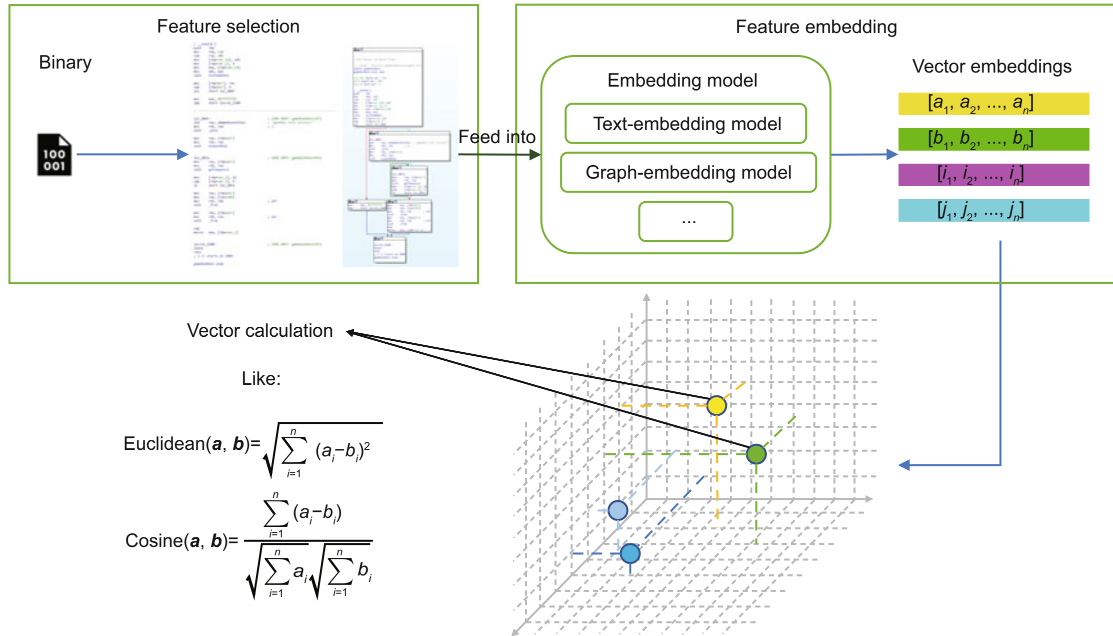

Fig. 2 Binary code representation procedure

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 use the specific data of the downstream task in the 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

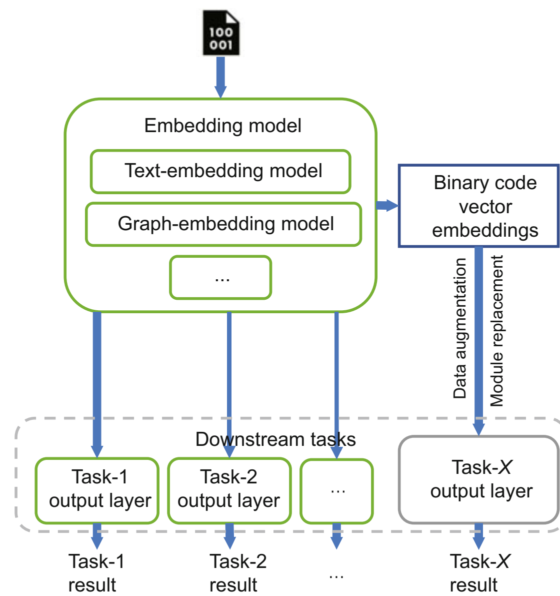

Fig. 3 Workflow of using binary code representation in binary analysis downstream tasks

classification, regression, and clustering, we can leverage the pre-training and 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 use 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 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 the corresponding tasks. This paradigm is capable of performing any function that falls 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, to address complex domain-specific tasks in the binary analysis field. The common approach is to replace certain internal 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 XJ et al., 2017) with the instruction embedding generated by the model PalmTree (Li XZX et al., 2021) 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, and 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 encoding, Word2Vec, long- and short-term memory network (LSTM), and other code encoding models in binary analysis methods such as EKLAVYA (Chua et al., 2017) and DEEPVSA (Guo WB 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 used 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 semantics. This section delves into the frequently selected abstract semantic features and the conventional feature representations used 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 perspective 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 use 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 control flow is the actual order and manner of program instruction execution, 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 using 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 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 configuration 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 is 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, and multi-dimensional vectors, are needed to reasonably display semantics on one hand, and facilitate the process of learning and understanding semantics by intelligent models to learn and understand semantics on the other hand. This subsection 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 use 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 text 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., 2019b), InnerEye (Zuo et al., 2019), Asm2Vec (Ding et al., 2019), cross-arch-instr-model (Redmond et al., 2019), MIRROR (Zhang XC et al., 2020), PalmTree (Li XZX et al., 2021), $\text{BinDiff}_{\text{NN}}$ (Ullah and Oh, 2022), and BinShot (Ahn et al., 2022).

The IL (or intermediate representation, 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, and MIPS), 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 (https://hex-rays.com/IDA-pro/), and ESIL in Radare2 (https://rada.re/n/radare2.html) 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) and BinFinder (Qasem et al., 2023).

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., 2016), GitZ (David et al., 2017), Zeek (Shalev and Partush, 2018), and FirmUp (David et al., 2018). MKIS (Li YC et al., 2020) uses fuzzing to build the input of dynamic execution of the program, considers the sequence of application programming interface (API) calls together with the key values that remain unchanged in multiple executions, constructs a new execution sequence of the function as a representation, and subsequently performs 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 a 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), and sem2vec (Wang HJ et al., 2023b). Trex (Pei et al., 2023) 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 H 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 XZ 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 used numerical statistical features.

The methods that use numerical statistical vectors as the ultimate feature representation include PatchEcko (Sun et al., 2020) and TikNib (Kim D 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 Section 3.2.3). Examples of such methods include Genius (Feng et al., 2016), Gemini (Xu XJ et al., 2017), VulSeeker (Gao J et al., 2018a), VulSeeker-Pro (Gao J et al., 2018b), and BinSeeker (Gao J 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 Topological graph structure

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

ACFG builds upon the CFG 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, ACFG used numerical statistical features as node attributes of the CFG, resulting in the topological graph structure depicted in Fig. 4. Works such as Genius and Gemini are representative examples that use this feature representation. Subsequently, some researchers enhanced the ACFG by incorporating DFG information, reflecting data flow characteristics. This enhancement involves adding edges from the DFG to enrich the edge information of the CFG. 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 graph matching network (GMN) (Li YJ et al., 2019), GraphEmb (Massarelli et al., 2019a), Order Matters (Yu ZP et al., 2020b), DeepBinDiff (Duan et al., 2020), Codee (Yang J et al., 2022), and VulHawk (Luo et al., 2023) are representative examples in this domain.

AST, a prevalent representation in source-level program analysis, is infrequently used in binary program analysis due to the 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 SG et al., 2021) and Asteria-Pro (Yang SG et al., 2023).

Table 1 Numerical statistical characteristics commonly used in functions

Feature description (static)	Feature description (dynamic)
Number of constant values in the function	Number of binary-defined function calls during execution
Number of strings in the function	Minimal stack depth during execution
Number of instructions 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 and FUNC FAR	Standard deviation of the stack depth during execution
Number of import functions	Number of executed instructions
Number of code references from this function	Number of executed unique instructions
Number of function calls from this function	Number of call instructions
Size of the function	Number of arithmetic instructions
Minimal number of instructions for a basic block	Number of branch instructions
Maximal number of instructions for a basic block	Number of load instructions
Average number of instructions for a basic block	Number of store instructions
Standard deviation of the number of instructions for a basic block	Maximal frequency of executing the same branch instruction
Minimal size of the basic block	Maximal frequency of executing the same arithmetic instruction
Maximal size of the basic block	Number of times accessing the heap memory space
Average size of the basic block	Number of times accessing the stack memory space
Standard deviation of size of the basic block	Number of times accessing the library memory space
Number of basic blocks for each function	Number of times accessing the anonymous mapping memory space
Number of edges among basic blocks for each function	Number of times accessing other part memory space
Function cyclomatic complexity=number of edges−number of nodes+2	Number of library function calls during execution
Normal block type of the function basic block	Number of system calls during execution
Block ends with an indirect jump	
Return block type of the function basic block	
Conditional return block type of the function basic block	
Noreturn block type of the function basic block	
External noreturn block (does not belong to the function)	
External normal block type of the function basic block	
Number of basic blocks executed beyond the function boundary	
Minimal number of call instructions of each basic block	
Maximal number of call instructions of each basic block	
Average number of call instructions of each basic block	
Standard deviation of the number of call instructions of each basic block	
Total number of call instructions of the function	
Minimal number of arithmetic instructions of each basic block	
Maximal number of arithmetic instructions of each basic block	
Average number of arithmetic instructions of each basic block	
Standard deviation of the number of arithmetic instructions of each basic block	
Total number of arithmetic instructions of the function	
Minimal number of arithmetic FP instructions of each basic block	
Maximal number of arithmetic FP instructions of each basic block	
Average number of arithmetic FP instructions of each basic block	
Standard deviation of the number of arithmetic FP instructions of each basic block	
Total number of arithmetic FP instructions of the function	
Minimal value of betweenness centrality	
Maximal value of betweenness centrality	
Average value of betweenness centrality	
Standard deviation of the value of betweenness centrality	
Number of nodes with betweenness centrality being zero	

FP: float-point

In addition to the commonly used representation forms in program analysis such as CFG, DFG, and AST, existing methods further enhance semantic features by designing customized topology diagram structures. For instance, αDiff (Liu BC et al., 2018) represents binary code bytes as 0 and 1 according to the $100 \times 100 \times 1$ dimension. If the dimension is insufficient, it pads with zeros, and if it exceeds the dimension limit, it truncates. This arrangement of the original byte stream follows a grid topology, resulting in a "0–1" matrix resembling an image pixel map. This representation form, often used in

Table 2 Function statistical features often used as node attribution

Genius and Gemini	VulSeeker, VulSeeker-Pro, and BinSeeker
String constants	Number of stack operation instructions
Numeric constants	Number of arithmetic instructions
Number of transfer instructions	Number of logical instructions
Number of calls	Number of comparative instructions
Number of instructions	Number of library function calls
Number of arithmetic instructions	Number of unconditional jump instructions
Number of offspring	Number of conditional jump instructions
Betweenness	Number of generic instructions

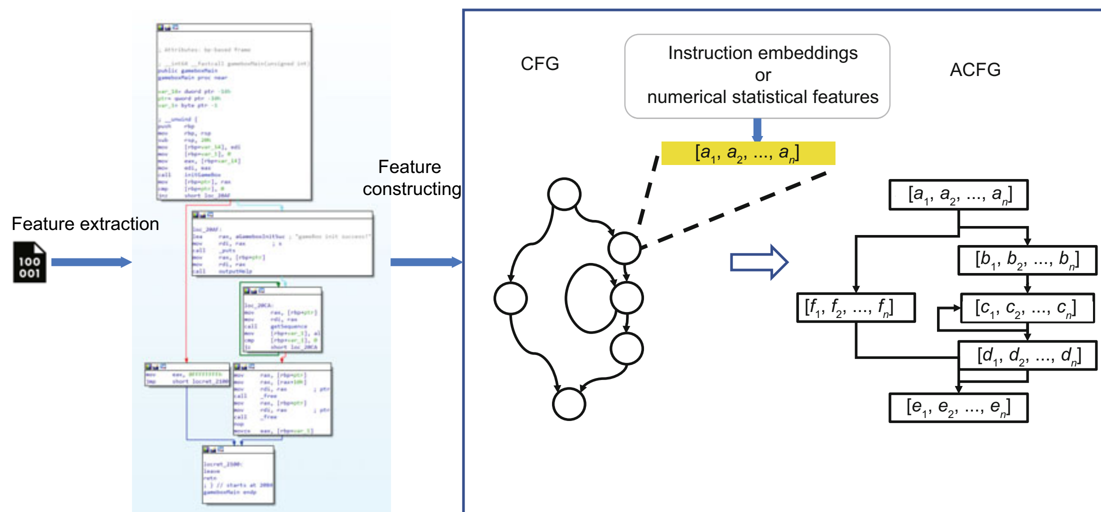

Fig. 4 ACFG construction process

malicious code detection (Wang JW et al., 2023), lacks interpretability. On the other hand, BCSD (Liu ZA, 2021) 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 YX et al., 2022) redefines the DFG, whereby nodes represent instructions determining the data flow direction, and edges signify data dependencies between instructions reflecting data flow characteristics. XBA (Kim G 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 HJ et al., 2023a) 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 semantics of function call features can be effectively represented through the sequence of function calls, a feature frequently used 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 αDiff, Asteria-Pro, and BinFinder.

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.

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 subsection systematically classifies and summarizes the abstract semantic features that researchers consider in the feature selection stage, which are divided mainly 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 divided roughly into six types, namely text 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 research selects 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 section introduces the feature embedding representation component of existing binary code representation methods, focusing on two primary and targeted embedding representation models: methods using text-embedding models, methods using graph-embedding models, methods integrating text-embedding and graph-embedding models, and other embedding methods.

4.1 Methods using 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 textual features into NLP models, tokenization and normalization must be performed as the preliminary step for encoding, 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 data—this is known as the OOV problem. PalmTree (Li XZX et al., 2021) 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 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

Table 3 Selection of code features in existing research works

Name	Binary code feature category									Form of characterization
	Syntax feature	Textual feature	Statistical feature	Control flow feature	Data flow feature	Symbolic feature	Function call feature	Data resource	Dynamic feature	
Genius (Feng et al., 2016)			✓	✓						ACFG
Gemini (Xu XJ et al., 2017)			✓	✓						ACFG
αDiff (Liu BC et al., 2018)							✓			Raw bytes
VulSeeker (Gao J et al., 2018a)			✓	✓	✓					LSFG
VulSeeker-Pro (Gao J et al., 2018b)			✓	✓	✓				✓	LSFG, dynamic trace
Zeek (Shalev and Partush, 2018)		✓		✓	✓					Strands
SAFE (Massarelli et al., 2019b)		✓								Assembly
GMN (Li YJ et al., 2019)				✓						ACFG
InnerEye (Zuo et al., 2019)		✓								Assembly
cross-arch-instr-model (Redmond et al., 2019)		✓								Assembly
GraphEmb (Massarelli et al., 2019a)		✓		✓						ACFG
Asm2Vec (Ding et al., 2019)		✓								Assembly
Order Matters (Yu ZP et al., 2020b)		✓		✓						ACFG, CFG
PatchEcko (Sun et al., 2020)			✓						✓	Numerical statistical feature
MKIS (Li YC et al., 2020)		✓		✓					✓	Customized instruction sequence
DeepBinDiff (Duan et al., 2020)		✓		✓						ACFG
MIRROR (Zhang XC et al., 2020)		✓								Assembly
BCSD (Liu ZA, 2021)				✓		✓				Customized graph structure
PalmTree (Li XZX et al., 2021)		✓			✓					Assembly
Asteria (Yang SG et al., 2021)	✓						✓			AST
OSCAR (Peng et al., 2021)		✓								IL text
BinDiff$_{NN}$ (Ullah and Oh, 2022)		✓								Assembly
Codee (Yang J et al., 2022)		✓		✓						Assembly, ACFG
BinSeeker (Gao J et al., 2021)			✓	✓	✓				✓	LSFG, dynamic trace
BinShot (Ahn et al., 2022)		✓								Assembly
BMM (Guo YX et al., 2022)		✓		✓	✓					CFG, call graph, DFG
jTrans (Wang H et al., 2022)		✓		✓						Customized instruction sequence
XBA (Kim G et al., 2022)		✓		✓			✓	✓		Customized graph structure
Trex (Pei et al., 2023)		✓			✓				✓	Customized instruction sequence
TikNib (Kim D et al., 2023)			✓							Numerical statistical features
DiEmph (Xu XZ et al., 2023)		✓								Customized instruction sequence
VulHawk (Luo et al., 2023)		✓		✓			✓	✓		ACFG
Asteria-Pro (Yang SG et al., 2023)	✓						✓	✓		AST
sem2vec (Wang HJ et al., 2023b)		✓		✓		✓	✓		✓	Dynamic trace, function call sequence
BinFinder (Qasem et al., 2023)		✓					✓	✓		IL text, function call sequence, constants

stages. However, following the introduction of the Transformer (Vaswani et al., 2017) model structure, network architectures constructed using the Transformer encoder and Transformer decoder have become increasingly prevalent in characterizing the features of binary code.

SAFE (Massarelli et al., 2019b) 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 instruction semantics is 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 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., 2019) 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., 2019) 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 the x86 and ARM architectures, 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 the CBOW model to predict the corresponding instructions according to the context instructions of 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 the distributed memory model of paragraph vectors (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.

$\text{BinDiff}_{\text{NN}}$ (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 adds attention to 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 the 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 XC et al., 2020) is based on the idea of neural machine translation (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's 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 low-level virtual machine (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 codes, binary codes, and binary and source codes. 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 XZ 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 problem of instruction distribution bias. This indicates a trend toward using Transformer, which demonstrates superior performance in understanding semantics. Another trend involves embedding based on IL text and customized sequences, which contains more abundant information to characterize binaries.

PalmTree (Li XZX et al., 2021) uses the 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 pre-training and fine-tuning paradigm 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 H 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 common vulnerability and exposure (CVE) 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) uses 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 using 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 the 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 XJ 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 a 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 J et al., 2018a) extracts data dependencies, constructs DFG, combines it into ACFG, and finally forms labeled semantic flow graph (LSFG). With reference to structure2vec, a DNN was constructed to carry out the graph structure embedding characterization. VulSeeker-Pro (Gao J et al., 2018b) and BinSeeker (Gao J et al., 2021) also use a custom DNN for embedded characterization of LSFG. Compared with VulSeeker, BinSeeker improves the representation of dynamic execution 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.

GMN (Li YJ 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 SG et al., 2021) and Asteria-Pro (Yang SG et al., 2023) focus on the characterization of the feature representation of an abstract syntax tree AST, they 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 the vector and state depends on the state of all the subunits related to it, and there are multiple forgetting gates, which can selectively obtain information from the subnodes. Experimental results show the effectiveness of this model in representing 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 G 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: 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 the 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 used 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.

GraphEmb (Massarelli et al., 2019a) obtains the embedded representation vector of each assembly instruction on the basis of the i2v instruction embedding model proposed by SAFE (Massarelli et al., 2019b), and aggregates 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 the 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.

Order Matters (Yu ZP et al., 2020b) uses a scheme similar to GraphEmb, first using the BERT model for instruction embedding and building ACFG and then using a 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 convolutional neural networks (CNNs) to represent

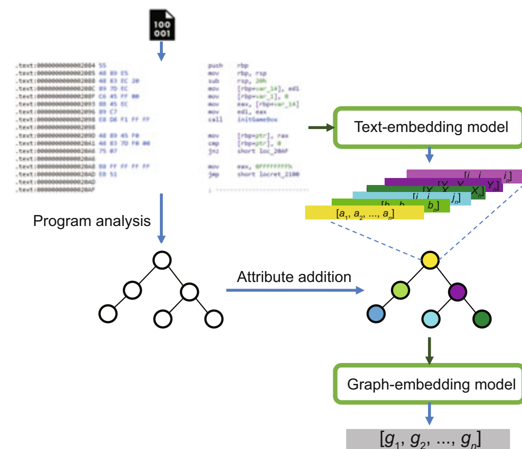

Fig. 5 Combination of text-embedding and graph-embedding models

the structure of the grid topological 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., 2019) 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, the text-associated deepwalk (TADW) algorithm (Yang C 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., 2019). However, the accuracy of the representation heavily depends on the quality of the instruction-level embedding.

Codee (Yang J et al., 2022) uses the graph-embedding algorithm node2vec (Grover and Leskovec, 2016) for random walk sampling of instruction sequences, and uses Skip-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 the accelerated attributed network embedding (AANE) (Huang et al., 2017) 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 a 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 YH et al., 2019) to generate instruction text-embedding vectors, which are 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 used to characterize and compute the similarity for the next step. This method follows the basic paradigm, but is aided by binary provenance information and correspondingly uses different adaptor weights for representation (Houlsby et al., 2019), thus achieving leading advantages on multiple metrics.

In BMM (Guo YX et al., 2022), it is proposed to merge the semantics of CFG, DFG, and call graph, use BERT to embed both instructions and operands at the same time, and use a gated graph neural network (GGNN) (Li YJ et al., 2015) to merge and represent the generated ACFG structure, so as to solve binary analysis tasks at the program and function level together using one model. sem2vec (Wang HJ et al., 2023b) uses the USE technique 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 topological 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 a 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 Sections 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.

αDiff (Liu BC 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 an image form and then uses an 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., 2016) 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 is 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 the 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 inputted 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 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 twin 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 achieving feature selection in complex compilation scenarios. However, advanced representation embedding techniques should 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 basic text-embedding model-based methods, graph-embedding model-based methods,

the methods integrating text-embedding and graph-embedding models, and other types of embedding representation methods. The specific feature-embedding techniques used 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 existing work, aligning with prominent research areas in recent years, the potential research directions are prospected.

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

Table 4 Embedding methods of code features in existing research works

Method	Feature-embedding method		
	Text-embedding model	Graph-embedding model	Others
Genius (Feng et al., 2016)			Spectral clustering
Gemini (Xu XJ et al., 2017)		structure2vec	
αDiff (Liu BC et al., 2018)		CNN	
VulSeeker (Gao J et al., 2018a)		Customized DNN	
VulSeeker-Pro (Gao J et al., 2018b)		Customized DNN	
Zeek (Shalev and Partush, 2018)			Customized layers
SAFE (Massarelli et al., 2019b)	Word2Vec (Skip-Gram)		
GMN (Li YJ et al., 2019)		Graph matching network	
InnerEye (Zuo et al., 2019)	Word2Vec (Skip-Gram)		
cross-arch-instr-model (Redmond et al., 2019)	Word2Vec (CBOW)		
GraphEmb (Massarelli et al., 2019a)	Word2Vec (Skip-Gram)	structure2vec	
Asm2Vec (Ding et al., 2019)	PV-DM		
Order Matters (Yu ZP et al., 2020b)	BERT	CNN+MPNN	MLP
PatchEcko (Sun et al., 2020)			Customized layers
DeepBinDiff (Duan et al., 2020)	Word2Vec (CBOW)	TADW	
MIRROR (Zhang XC et al., 2020)	Transformer		
PalmTree (Li XZX et al., 2021)	BERT		
Asteria (Yang SG et al., 2021)		Tree-LSTM	
OSCAR (Peng et al., 2021)	Transformer		
$\text{BinDiff}_{\text{NN}}$ (Ullah and Oh, 2022)	ABENN		
Codee (Yang J et al., 2022)	Word2Vec (Skip-Gram)	AANE+LINE	
BinSeeker (Gao J et al., 2021)		Customized DNN	
BinShot (Ahn et al., 2022)	BERT		
BMM (Guo YX et al., 2022)	BERT	GGNN	
jTrans (Wang H et al., 2022)	BERT		
XBA (Kim G et al., 2022)		GCN	
Trex (Pei et al., 2023)	Transformer		
DiEmph (Xu XZ et al., 2023)	Transformer		
VulHawk (Luo et al., 2023)	RoBERTa	GCN	
Asteria-Pro (Yang SG et al., 2023)		Tree-LSTM	
sem2vec (Wang HJ et al., 2023b)	RoBERTa	GCN	
BinFinder (Qasem et al., 2023)			Customized layers

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, and other forms such 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 exist 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 ZM 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 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 the corresponding natural language annotation information matching with the binary code can provide richer and higher-level information such as design considerations for analysis (Zhang YF et al., 2022). The source code and annotation information are 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 and achieve similarity detection, by mining the common features between the two code forms (Guo XX et al., 2023), converting uniformly to binary code or IL level (Ji et al., 2021; Peng et al., 2021), and using multi-modal fusion and cross-modal matching (Yu ZP et al., 2020a) 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 comprehensive 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, which is an important research direction. Research on interpretability involves the significance of features, the internal

structure of the model, and the interpretability of the model output. 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., 2024; Wang JJ et al., 2024). 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 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, and instruction disassembly, which can be carried out.

6 Conclusions

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, we categorize 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 paper. Qingsong XIE collected the literature and statistically analyzed the tables. Lu YU helped organize the paper. 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. Proc 38th Annual Computer Security Applications Conf, p.361-374. https://doi.org/10.1145/3564625.3567975

Allamanis M, Barr ET, Ducousso S, et al., 2020. Typilus: neural type hints. Proc 41st ACM SIGPLAN Conf on Programming Language Design and Implementation, p.91-105. https://doi.org/10.1145/3385412.3385997

Bengio Y, Courville A, Vincent P, 2013. Representation learning: a review and new perspectives. *IEEE Trans Patt Anal Mach Intell*, 35(8):1798-1828. https://doi.org/10.1109/TPAMI.2013.50

Chaganti R, Ravi V, Pham TD, 2022. Deep learning based cross architecture Internet of Things malware detection and classification. *Comput Secur*, 120:102779. https://doi.org/10.1016/j.cose.2022.102779

Chandramohan M, Xue YX, Xu ZZ, et al., 2016. BinGo: cross-architecture cross-OS binary search. Proc 24th ACM SIGSOFT Int Symp on Foundations of Software Engineering, p.678-689. https://doi.org/10.1145/2950290.2950350

Chen LG, He ZL, Mao B, 2020. CATI: context-assisted type inference from stripped binaries. Proc 50th Annual IEEE/IFIP Int Conf on Dependable Systems and Networks, p.88-98. https://doi.org/10.1109/DSN48063.2020.00028

Chen QB, Lacomis J, Schwartz EJ, et al., 2022. Augmenting decompiler output with learned variable names and types. Proc 31st USENIX Security Symp, p.4327-4343.

Chu QF, Liu GS, Zhu X, 2020. Visualization feature and CNN based homology classification of malicious code. *Chin J Electron*, 29(1):154-160. https://doi.org/10.1049/cje.2019.11.005

Chua ZL, Shen SQ, Saxena P, et al., 2017. Neural nets can learn function type signatures from binaries. Proc 26th USENIX Conf on Security Symp, p.99-116.

Dai HJ, Dai B, Song L, 2016. Discriminative embeddings of latent variable models for structured data. Proc 33rd Int Conf on Machine Learning, p.2702-2711.

David Y, Yahav E, 2014. Tracelet-based code search in executables. Proc 35th ACM SIGPLAN Conf on Programming Language Design and Implementation, p.349-360. https://doi.org/10.1145/2594291.2594343

David Y, Partush N, Yahav E, 2016. Statistical similarity of binaries. *ACM SIGPLAN Not*, 51(6):266-280. https://doi.org/10.1145/2980983.2908126

David Y, Partush N, Yahav E, 2017. Similarity of binaries through re-optimization. Proc 38th ACM SIGPLAN Conf on Programming Language Design and Implementation, p.79-94. https://doi.org/10.1145/3062341.3062387

David Y, Partush N, Yahav E, 2018. FirmUp: precise static detection of common vulnerabilities in firmware. *ACM SIGPLAN Not*, 53(2):392-404. 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. *Proc ACM Program Lang*, 4(OOPSLA):225. https://doi.org/10.1145/3428293

Devlin J, Chang MW, Lee K, et al., 2019. BERT: pre-training of deep bidirectional Transformers for language understanding. Proc Conf of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, p.4171-4186. https://doi.org/10.18653/v1/N19-1423

Ding SHH, Fung BCM, Charland P, 2019. Asm2Vec: boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. Proc IEEE Symp on Security and Privacy, p.472-489. https://doi.org/10.1109/SP.2019.00003

Duan Y, Li XZX, Wang JH, et al., 2020. DeepBinDiff: learning program-wide code representations for binary diffing. Network and Distributed Systems Security Symp, p.1-16. https://doi.org/10.14722/ndss.2020.24311

Feng Q, Zhou RD, Xu CC, et al., 2016. Scalable graph-based bug search for firmware images. Proc ACM SIGSAC Conf on Computer and Communications Security, p.480-491. https://doi.org/10.1145/2976749.2978370

Gao H, Cheng SY, Xue YX, et al., 2021. A lightweight framework for function name reassignment based on large-scale stripped binaries. Proc 30th ACM SIGSOFT Int Symp on Software Testing and Analysis, p.607-619. 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. Proc 33rd ACM/IEEE Int Conf on Automated Software Engineering, p.896-899. 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. Proc 26th ACM Joint Meeting on European Software Engineering Conf and Symp on the Foundations of Software Engineering, p.803-808. 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 Trans Softw Eng*, 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. Proc 26th European Symp on Research in Computer Security, p.605-625. https://doi.org/10.1007/978-3-030-88418-5_29

Gilmer J, Schoenholz SS, Riley PF, et al., 2017. Neural message passing for quantum chemistry. Proc 34th Int Conf on Machine Learning, p.1263-1272.

Grover A, Leskovec J, 2016. node2vec: scalable feature learning for networks. Proc 22nd ACM SIGKDD Int Conf on Knowledge Discovery and Data Mining, p.855-864. https://doi.org/10.1145/2939672.2939754

Guo WB, Mu DL, Xing XY, et al., 2019. DEEPVSA: facilitating value-set analysis with deep learning for postmortem program analysis. Proc 28th USENIX Conf on Security Symp, p.1787-1804.

Guo XX, Cai RJ, Yin XK, et al., 2023. Searching open-source vulnerability function based on software modularization. *Appl Sci*, 13(2):701. https://doi.org/10.3390/app13020701

Guo YX, Li PC, Luo YW, et al., 2022. Exploring GNN based program embedding technologies for binary related tasks. Proc 30th IEEE/ACM Int Conf on Program Comprehension, p.366-377. https://doi.org/10.1145/3524610.3527900

Haq IU, Caballero J, 2021. A survey of binary code similarity. *ACM Comput Surv*, 54(3):51. https://doi.org/10.1145/3446371

Hou XY, Zhao YJ, Liu Y, et al., 2024. Large language models for software engineering: a systematic literature review. https://doi.org/10.48550/arXiv.2308.10620

Houlsby N, Giurgiu A, Jastrzebski S, et al., 2019. Parameter-efficient transfer learning for NLP. Proc 36th Int Conf on Machine Learning, p.2790-2799.

Huang X, Li JD, Hu X, 2017. Accelerated attributed network embedding. Proc SIAM Int Conf on Data Mining, p.633-641. https://doi.org/10.1137/1.9781611974973.71

Ji YD, Cui L, Huang HH, 2021. BugGraph: differentiating source-binary code similarity with graph triplet-loss network. Proc ACM Asia Conf on Computer and Communications Security, p.702-715. https://doi.org/10.1145/3433210.3437533

Jin X, Pei KX, Won JY, et al., 2022. SymLM: predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. Proc ACM SIGSAC Conf on Computer and Communications Security, p.1631-1645. https://doi.org/10.1145/3548606.3560612

Kim D, Kim E, Cha SK, et al., 2023. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Trans Softw Eng*, 49(4):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. Proc 31st ACM SIGSOFT Int

Symp on Software Testing and Analysis, p.151-163. 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. Proc ACM Asia Conf on Computer and Communications Security, p.951-965. https://doi.org/10.1145/3579856.3582823

Kipf TN, Welling M, 2016. Semi-supervised classification with graph convolutional networks. Proc 5th Int Conf on Learning Representations.

Lafferty JD, McCallum A, Pereira FCN, 2001. Conditional random fields: probabilistic models for segmenting and labeling sequence data. Proc 18th Int Conf on Machine Learning, p.282-289.

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

Li CF, Shen GM, Sun W, 2021. Cross-architecture Internet-of-Things malware detection based on graph neural network. Proc Int Joint Conf on Neural Networks, p.1-7. https://doi.org/10.1109/IJCNN52387.2021.9533500

Li XZX, Qu Y, Yin H, 2021. PalmTree: learning an assembly language model for instruction embedding. Proc ACM SIGSAC Conf on Computer and Communications Security, p.3236-3251. https://doi.org/10.1145/3460120.3484587

Li YC, Wang BY, Hu BJ, 2020. Semantically find similar binary codes with mixed key instruction sequence. *Inform Softw Technol*, 125:106320. https://doi.org/10.1016/j.infsof.2020.106320

Li YJ, Tarlow D, Brockschmidt M, et al., 2015. Gated graph sequence neural networks. Proc 4th Int Conf on Learning Representations.

Li YJ, Gu CJ, Dullien T, et al., 2019. Graph matching networks for learning the similarity of graph structured objects. Proc 36th Int Conf on Machine Learning, p.3835-3845.

Liu BC, Huo W, Zhang C, et al., 2018. αDiff: cross-version binary code similarity detection with DNN. Proc 33rd IEEE/ACM Int Conf on Automated Software Engineering, p.667-678. https://doi.org/10.1145/3238147.3238199

Liu QX, Liu JX, Jin Z, et al., 2023. Survey of artificial intelligence based IoT malware detection. *J Comput Res Dev*, 60(10):2234-2254 (in Chinese). https://doi.org/10.7544/issn1000-1239.202330450

Liu YH, Ott M, Goyal N, et al., 2019. RoBERTa: a robustly optimized BERT pretraining approach. https://doi.org/10.48550/arXiv.1907.11692

Liu ZA, 2021. Binary code similarity detection. Proc 36th IEEE/ACM Int Conf on Automated Software Engineering, p.1056-1060. https://doi.org/10.1109/ASE51524.2021.9678518

Liu ZM, Kitouni O, Nolte N, et al., 2022. Towards understanding grokking: an effective theory of representation learning. Proc 36th Conf on Neural Information Processing Systems, p.34651-34663.

Lu XD, Duan ZM, Qian YK, et al., 2020. Malicious code classification method based on deep forest. *J Softw*, 31(5):1454. https://doi.org/10.13328/j.cnki.jos.005660

Lu YL, Yu L, Zhao JZ, 2023. Survey of software vulnerability mining methods based on machine learning. *Inform Counterm Technol*, 2(2):1-19 (in Chinese). https://doi.org/10.12399/j.issn.2097-163x.2023.02.001

Luo ZH, Wang PW, Wang BS, et al., 2023. VulHawk: cross-architecture vulnerability detection with entropy-based binary code search. Proc 30th Annual Network and Distributed System Security Symp. https://doi.org/10.14722/ndss.2023.24415

Marcelli A, Graziano M, Ugarte-Pedrero X, et al., 2022. How machine learning is solving the binary function similarity problem. Proc 31st USENIX Security Symp, p.2099-2116.

Massarelli L, Di Luna GA, Petroni F, et al., 2019a. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. Proc Workshop on Binary Analysis Research, p.1-11. https://doi.org/10.14722/bar.2019.23020

Massarelli L, Di Luna GA, Petroni F, et al., 2019b. SAFE: self-attentive function embeddings for binary similarity. Proc 16th Int Conf on Detection of Intrusions and Malware, and Vulnerability Assessment, p.309-329. https://doi.org/10.1007/978-3-030-22038-9_15

Mikolov T, Chen K, Corrado G, et al., 2013. Efficient estimation of word representations in vector space. Proc 1st Int Conf on Learning Representations.

Nethercote N, Seward J, 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. Proc 28th ACM SIGPLAN Conf on Programming Language Design and Implementation, p.89-100. 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. Proc 1st Workshop on Natural Language Processing for Programming, p.48-57. https://doi.org/10.18653/v1/2021.nlp4prog-1.6

Patrick-Evans J, Dannehl M, Kinder J, 2023. XFL: naming functions in binaries with extreme multi-label learning. Proc IEEE Symp on Security and Privacy, p.2375-2390. https://doi.org/10.1109/SP46215.2023.10179439

Pei KX, Guan J, Broughton M, et al., 2021. StateFormer: fine-grained type recovery from binaries using generative state modeling. Proc 29th ACM Joint Meeting on European Software Engineering Conf and Symp on the Foundations of Software Engineering, p.690-702. https://doi.org/10.1145/3468264.3468607

Pei KX, Xuan Z, Yang JF, et al., 2023. Learning approximate execution semantics from traces for binary function similarity. *IEEE Trans Softw Eng*, 49(4):2776-2790. https://doi.org/10.1109/TSE.2022.3231621

Peng DL, Zheng SX, Li YT, et al., 2021. How could neural networks understand programs? Proc 38th Int Conf on Machine Learning, p.8476-8486.

Pham DP, Marion D, Mastio M, et al., 2021. Obfuscation revealed: leveraging electromagnetic signals for obfuscated malware classification. Proc 37th Annual Computer Security Applications Conf, p.706-719. https://doi.org/10.1145/3485832.3485894

Power A, Burda Y, Edwards H, et al., 2022. Grokking: generalization beyond overfitting on small algorithmic datasets. https://doi.org/10.48550/arXiv.2201.02177

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. Proc ACM Asia Conf on Computer and Communications Security, p.443-456. https://doi.org/10.1145/3579856.3582818

Qiao YC, Zhang WZ, Du XJ, et al., 2021. Malware classification based on multilayer perception and Word2Vec for IoT security. *ACM Trans Int Technol*, 22(1):10. https://doi.org/10.1145/3436751

Ramos DA, Engler D, 2015. Under-constrained symbolic execution: correctness checking for real code. Proc 24th USENIX Conf on Security Symp, p.49-64.

Redmond K, Luo LN, Zeng Q, 2019. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. Proc Workshop on Binary Analysis Research, p.1-8. https://doi.org/10.14722/bar.2019.23057

Shalev N, Partush N, 2018. Binary similarity detection using machine learning. Proc 13th Workshop on Programming Languages and Analysis for Security, p.42-47. https://doi.org/10.1145/3264820.3264821

Sun PF, Garcia L, Salles-Loustau G, et al., 2020. Hybrid firmware analysis for known mobile and IoT security vulnerabilities. Proc 50th Annual IEEE/IFIP Int Conf on Dependable Systems and Networks, 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. Proc 53rd Annual Meeting of the Association for Computational Linguistics and 7th Int Joint Conf on Natural Language Processing, p.1556-1566. https://doi.org/10.3115/v1/P15-1150

Tang J, Qu M, Wang MZ, et al., 2015. LINE: large-scale information network embedding. Proc 24th Int Conf on World Wide Web, p.1067-1077. https://doi.org/10.1145/2736277.2741093

Ullah S, Oh H, 2022. BinDiff$_{\rm NN}$: learning distributed representation of assembly for robust binary diffing against semantic differences. *IEEE Trans Softw Eng*, 48(9):3442-3466. https://doi.org/10.1109/TSE.2021.3093926

Vasan D, Alazab M, Wassan S, et al., 2020a. IMCFN: image-based malware classification using fine-tuned convolutional neural network architecture. *Comput Netw*, 171:107138. https://doi.org/10.1016/j.comnet.2020.107138

Vasan D, Alazab M, Venkatraman S, et al., 2020b. MTHAEL: cross-architecture IoT malware detection based on neural network advanced ensemble learning. *IEEE Trans Comput*, 69(11):1654-1667. https://doi.org/10.1109/TC.2020.3015584

Vaswani A, Shazeer N, Parmar N, et al., 2017. Attention is all you need. Proc 31st Int Conf on Neural Information Processing Systems, p.6000-6010.

Vinyals O, Bengio S, Kudlur M, 2015. Order Matters: sequence to sequence for sets. Proc 4th Int Conf on Learning Representations.

Wang H, Qu WJ, Katz G, et al., 2022. jTrans: jump-aware Transformer for binary code similarity detection. Proc 31st ACM SIGSOFT Int Symp on Software Testing and Analysis, p.1-13. https://doi.org/10.1145/3533767.3534367

Wang HJ, Ma PC, Yuan YY, et al., 2023a. Enhancing DNN-based binary code function search with low-cost equivalence checking. *IEEE Trans Softw Eng*, 49(1):226-250. https://doi.org/10.1109/TSE.2022.3149240

Wang HJ, Ma PC, Wang S, et al., 2023b. sem2vec: semantics-aware assembly tracelet embedding. *ACM Trans Softw Eng Methodol*, 32(4):90. https://doi.org/10.1145/3569933

Wang JJ, Huang YC, Chen CY, et al., 2024. Software testing with large language model: survey, landscape, and vision. *IEEE Trans Softw Eng*, 50(4):911-936. https://doi.org/10.1109/TSE.2024.3368208

Wang JW, Chen ZJ, Xie X, et al., 2023. Review of malware detection and classification visualization techniques. *Chin J Netw Inform Secur*, 9(5):1 (in Chinese).

Wu CY, Ban T, Cheng SM, et al., 2023. IoT malware classification based on reinterpreted function-call graphs. *Comput Secur*, 125:103060. https://doi.org/10.1016/j.cose.2022.103060

Xu MJ, 2021. Understanding graph embedding methods and their applications. *SIAM Rev*, 63:825-853. https://doi.org/10.1137/20M1386062

Xu XJ, Liu C, Feng Q, et al., 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. Proc ACM SIGSAC Conf on Computer and Communications Security, p.363-376. https://doi.org/10.1145/3133956.3134018

Xu XZ, Feng SW, Ye YP, et al., 2023. Improving binary code similarity Transformer models by semantics-driven instruction deemphasis. Proc 32nd ACM SIGSOFT Int Symp on Software Testing and Analysis, p.1106-1118. https://doi.org/10.1145/3597926.3598121

Yang C, Liu ZY, Zhao DL, et al., 2015. Network representation learning with rich text information. Proc 24th Int Conf on Artificial Intelligence, p.2111-2117.

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

Yang SG, Cheng L, Zheng YC, et al., 2021. Asteria: deep learning-based AST-encoding for cross-platform binary code similarity detection. 51st Annual IEEE/IFIP Int Conf on Dependable Systems and Networks, p.224-236. https://doi.org/10.1109/DSN48987.2021.00036

Yang SG, Dong CP, Xiao Y, et al., 2023. Asteria-Pro: enhancing deep learning-based binary code similarity detection by incorporating domain knowledge. *ACM*

Trans Softw Eng Methodol, 33(1):1.
https://doi.org/10.1145/3604611

Yu SY, Achamyeleh YG, Wang CH, et al., 2023. CFG2VEC: hierarchical graph neural network for cross-architectural software reverse engineering. Proc IEEE/ACM 45th Int Conf on Software Engineering: Software Engineering in Practice, p.281-291.
https://doi.org/10.1109/ICSE-SEIP58684.2023.00031

Yu YC, Gan ST, Qiu JY, et al., 2022. Binary code similarity analysis and its applications on embedded device firmware vulnerability search. *J Softw*, 33(11):4137-4172. https://doi.org/10.13328/j.cnki.jos.006540

Yu ZP, Zheng WX, Wang JQ, et al., 2020a. CodeCMR: cross-modal retrieval for function-level binary source code matching. 34th Conf on Neural Information Processing Systems, p.1-3.

Yu ZP, Cao R, Tang QY, et al., 2020b. Order Matters: semantic-aware neural networks for binary code similarity detection. Proc 34th AAAI Conf on Artificial Intelligence, p.1145-1152.
https://doi.org/10.1609/aaai.v34i01.5466

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

Zhang XC, Sun WJ, Pang JM, et al., 2020. Similarity metric method for binary basic blocks of cross-instruction set architecture. Proc Workshop on Binary Analysis Research, p.1-12.
https://doi.org/10.14722/bar.2020.23002

Zhang YF, Huang C, Zhang YK, et al., 2022. Pre-training representations of binary code using contrastive learning. https://doi.org/10.48550/arXiv.2210.05102

Zhang Z, Ye YP, You W, et al., 2021. OSPREY: recovery of variable and data structure via probabilistic analysis for stripped binary. Proc IEEE Symp on Security and Privacy, p.813-832.
https://doi.org/10.1109/SP40001.2021.00051

Zuo F, Li XP, Zhang Z, et al., 2019. Neural machine translation inspired binary code similarity comparison beyond function pairs. https://arxiv.org/pdf/1808.04706