Cybersecurity

RESEARCH

Open Access

Unleashing the power of pseudo-code for binary code similarity analysis

Weiwei Zhang[1], Zhengzi Xu[2], Yang Xiao[3,4] and Yinxing Xue[1*]

Abstract

Code similarity analysis has become more popular due to its significant applicantions, including vulnerability detection, malware detection, and patch analysis. Since the source code of the software is difficult to obtain under most circumstances, binary-level code similarity analysis (BCSA) has been paid much attention to. In recent years, many BCSA studies incorporating AI techniques focus on deriving semantic information from binary functions with code representations such as assembly code, intermediate representations, and control flow graphs to measure the similarity. However, due to the impacts of different compilers, architectures, and obfuscations, binaries compiled from the same source code may vary considerably, which becomes the major obstacle for these works to obtain robust features. In this paper, we propose a solution, named UPPC (Unleashing the Power of Pseudo-code), which leverages the pseudo-code of binary function as input, to address the binary code similarity analysis challenge, since pseudo-code has higher abstraction and is platform-independent compared to binary instructions. UPPC selectively inlines the functions to capture the full function semantics across different compiler optimization levels and uses a deep pyramidal convolutional neural network to obtain the semantic embedding of the function. We evaluated UPPC on a data set containing vulnerabilities and a data set including different architectures (X86, ARM), different optimization options (O0-O3), different compilers (GCC, Clang), and four obfuscation strategies. The experimental results show that the accuracy of UPPC in function search is 33.2% higher than that of existing methods.

Keywords: Binary code similarity, Machine learning, Software security, Pseudo-code

Introduction

Open source libraries are widely adopted in software development cycles, which improves the development efficiency and reduces the costs (Laguë et al. 1997). They also result in a large number of code duplicates and clones in different software (Alrabaee et al. 2020). Source code level code auditing and clone detection tools (Fang et al. 2020; Zhang et al. 2019) are developed to find the usage of open source components in the software. However, in the real world, the source code of the software is often difficult to obtain, which makes these tools impractical to search clones in off-the-shelf software. To analysis, the program in binary format, closed-source binary code similarity analysis (BSCA) has been proposed. It becomes the key technique to address security-related issues in open source components at binary level, such as patch analysis (Xiao et al. 2021; Dullien and Rolles 2005; Gao et al. 2008), vulnerability searching (Xu et al. 2017b; Eschweiler et al. 2016), code plagiarism detection (Wang et al. 2009; Luo et al. 2014; Liu et al. 2006), and etc.

In retrospect, the applications and challenges of BSCA have led to the proposal of a variety of BSCA tools based on different types of code representation, including text-based (David et al. 2017), program logic-based (Chandramohan et al. 2016; David et al. 2016), CFG-based (Lindorfer et al. 2012; Luo et al. 2014), etc. Recently, the latest machine learning (ML) methods have significantly

*Correspondence: yxxue@ustc.edu.cn

[1] School of Computer Science and Engineering, University of Science and Technology of China, Hefei, China
Full list of author information is available at the end of the article

enhanced the capabilities of BSCA (Peng et al. 2021; Ding et al. 2019; Massarelli et al. 2019; Yu et al. 2020b).

However, performing binary semantic analysis directly on assembly code features or control flow graph (CFG) features is challenging because different architectures have different assembly codes and obfuscation changes the CFG of functions. It hinders the understanding of program semantics by deep learning models (Haq and Caballero 2021). Therefore, to eliminate differences in assembly code between architectures, existing approaches (Peng et al. 2021; Luo et al. 2019) use deep learning techniques to learn function semantics from intermediate representation (IR) features, which are platform-independent and more abstract than assembly code. Furthermore, Singh (2021) found that combining a compiler with specific optimization options to compile the source code into a binary file and then extracting the corresponding binary pseudo-code was more beneficial for code classification and code clone detection. The reason is that deep learning-based approaches are known to have impressive success in source code clone detection (Fang et al. 2020; Zhang et al. 2019; Alon et al. 2019) and the pseudo-code is similar to source code, which can be extracted from a binary executable by decompiler tools. *However, as far as we have reviewed, there is no BCSA work using pseudo-code to extract features and match functions.*

To this end, we propose a deep learning-based binary code similarity measurement tool, named UPPC (Unleashing Power of Pseudo-code), which leverages pseudo-code to extract the semantic representation of functions. Inspired by existing work (Chandramohan et al. 2016; Peng et al. 2021; Ding et al. 2019; Luo et al. 2019; Singh 2021), UPPC selectively inlines key functions (Chandramohan et al. 2016; Ding et al. 2019) to recover the full semantics of functions. Then, it extracts code features and string features from the pseudo-code and combines them with existing *deep pyramid convolutional neural networks* (Johnson and Zhang 2017) (DPCNN) to capture the full function semantics and compute semantic embedding vectors. This semantic embedding vector can then be used to efficiently match semantically similar functions for tasks such as vulnerability detection and function search.

Compared to previous binary-level analysis methods our approach has the following advantages. (1) Pseudo-code is platform-independent: Binary instructions or assembly code have different architectures (e.g. X86 and ARM architectures), while pseudo-code is generated with higher abstraction to unify the instructions. Therefore, by converting programs to pseudo-code, differences between architectures can be eliminated, enabling more accurate cross-architecture code analysis. (2) Pseudo-code contains richer syntactic information: Pseudo-code is similar to source code in that it contains semantic features such as variable definitions, data structure definitions, strings, etc. (3) Pseudo-code contains richer semantic information: by converting binary to pseudo-code, more information about the logical structure of the code can be recovered, and pseudo-code is closer to natural human language, allowing the function of the program to be easily expressed.

To evaluate the effectiveness of our approach, we evaluated UPPC on an open-source data set (Kim et al. 2020) containing different architectures (X86, ARM), different optimization options (O0-O3), different compilers (GCC, Clang), and four obfuscation strategies. The experiments show that UPPC outperforms existing tools in identifying semantically similar functions across all the aspects. Moreover, to explore the applications of UPPC, we conducted vulnerability search experiments on a publicly available data set (David et al. 2016). UPPC correctly found 95.1% of vulnerabilities compared to existing tools SAFE (Massarelli et al. 2019) and Gemini (Xu et al. 2017b), which could only find 68.9% and 59.0% of vulnerabilities. Finally, we conduct the ablation experiments to show that the accuracy of semantic similarity function matching can be significantly improved by selectively inlining key functions and using pseudo-code and string features.

Overall, we have made the following contributions:

- We present a new approach to matching semantically similar functions in closed-source software: we learn binary function feature representations from pseudo-code using deep learning. Pseudo-code is more abstract than intermediate representation (IR) and assembly code, which is similar to source code and contains more semantic features.
- We develop a semantic learning model UPPC for function pseudo-code. UPPC first captures the complete function semantics through function inlining, learns key semantic information about the function from the pseudo-code and string features, and obtains a semantic embedding vector for the function.
- We demonstrate that UPPC is more resilient than existing deep learning-based BCSA methods. In BCSA tasks, such as function similarity matching, function search, and vulnerability detection, UPPC has better accuracy than existing methods.

Preliminaries

In this section, we first introduce background knowledge about function semantic similarity. Then, we explain the problems that existing methods for function similarity detection may encounter with a simple example. Finally, we describe the method we use and its benefits.

Code similarity

Code similarity detection can be divided into source code similarity detection and binary code similarity detection, which fall into two main categories: syntactic similarity and semantic similarity (Haq and Caballero 2021; Walker et al. 2020; Bellon et al. 2007). The syntactic similarity in source code refers to code fragments with similar text, while in binary code it refers to sequences of similar instructions. Semantic similarity in source and binary code refers to code that is functionally similar. In general, semantic similarity refers to whether the code being compared has a similar effect, while syntactic similarity refers to the similarity of the code representation. However, in most cases, the source code of the software is often difficult to obtain and therefore binary code similarity detection techniques are more widely used in a wide range of scenarios. Traditional binary similarity detection methods can be broadly analysed to include dynamic (Zhang et al. 2006; Newsome and Song 2005; King 1976; Pei et al. 2020). and static (Xiao et al. 2021; David et al. 2017; Chandramohan et al. 2016) analysis, and are now more often combined with deep learning (Peng et al. 2021; Ding et al. 2019; Pei et al. 2020; Duan et al. 2020) for binary similarity analysis.

Technical challenges

In contrast to source code similarity detection, different binary codes compiled from the same source code can differ significantly due to differences in architecture, compile, compilation options, security compilation options, etc. Therefore, while the high-level idea seems simple enough, the following challenges are required to implement binary similarity detection:

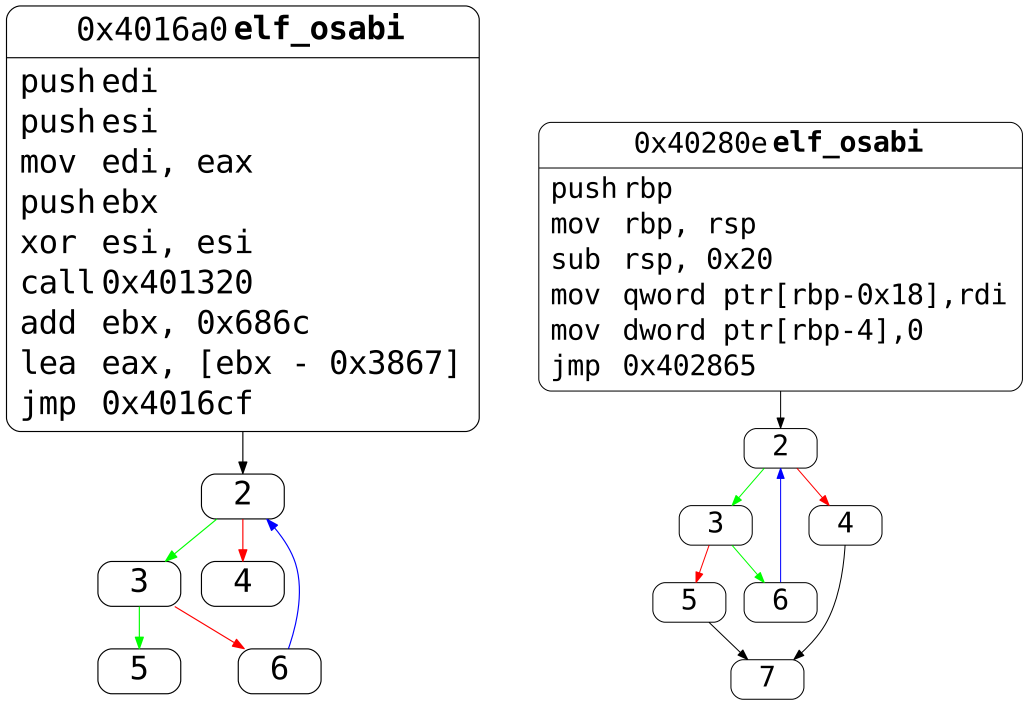
(1) *Key information loss* When we compile source code into binary, information such as function names, variable names, data structure definitions, variable type definitions, comments, and other information that helps to understand the intent of the code is lost.
(2) *Cross-compiler* The binaries compiled with different compilers are different because different compilers use different optimization algorithms when compiling, and the same compiler continues to optimize and improve the compilation algorithm during iterations.
(3) *Cross-compilation optimization* The compiler can produce completely different binaries depending on the optimization technique used, for example, GCC offers almost a hundred optimization options, any level of optimization will result in changes to the structure of the code such as merging and eliminating branches, eliminating common sub-expressions, etc.
(4) *Cross-architecture* Instructions are different between different architectures. Different architectures have their own recognizable instruction sets, and different instruction sets correspond to different assembly codes.
(5) *Code protection* Code protection techniques, such as code obfuscation, compression pack, encryption pack, etc., can lead to complex code structures and unrecognizable execution sequences.

Research motivation

We use motivating examples to illustrate the problems encountered with existing assembly code and CFG based approaches, and the advantages of our pseudo-code-based approach.

Figure 1 gives the CFG for the function elf_obf in Binutils, compiled into a 32-bit program and a 64-bit program using the same compiler, GCC, but with different compilation options, O0 and O2, under the same architecture. Although it is clear from Fig. 8a, b that the function uses a loop structure, there are differences in the number of basic blocks (6 and 7 respectively) and the amount of assembly code in each basic block in the two CFG graphs. Gemini (Xu et al. 2017b) was created by extracting basic block attributes (numerical constants, number of transfer instructions, number of calls, etc.) to construct an attribute control flow graph (ACFG), which poses a challenge for Gemini in terms of code similarity detection. SAFE (Massarelli et al. 2019) directly considers sequences of instructions in binary functions and models them as a natural language. As shown in Fig. 8a, b, 64-bit and 32-bit assembly instructions, in the number of basic registers (64-bit has 16 basic registers, and 32-bit only 8), the naming of registers (32-bit registers start with *e*, and 64-bit registers use *r*), function transfer, etc. are different, so directly using assembly instructions as text sequences will also affect the accuracy of the model.

In contrast, we can use the decompiler tool to obtain binary pseudo-code, which is very similar to the source code, pseudo-code snippets #2 and #3 correspond to Fig. 8a, b respectively. As shown in Fig. 2 the pseudo-code has a more uniform style than the binary code, and the pseudo-code for both 64 and 32-bit programs is similar to the source code (#1 in Fig. 2). In addition, the pseudo-code retains more semantic features and is more syntactically uniform. Thus, if we had access to



```
0x4016a0 elf_osabi
push edi
push esi
mov  edi, eax
push ebx
xor  esi, esi
call 0x401320
add  ebx, 0x686c
lea  eax, [ebx - 0x3867]
jmp  0x4016cf
```

```
0x40280e elf_osabi
push rbp
mov  rbp, rsp
sub  rsp, 0x20
mov  qword ptr[rbp-0x18],rdi
mov  dword ptr[rbp-4],0
jmp  0x402865
```

(a) X86-32-O2 (b) X86-64-O0

Fig. 1 Motivating example: the control flow graph and assemble code for the elf_osabi. The code block 0x4016a0 in **a** and the code block 0x40280e in **b** give the different assembly code blocks. The code blocks 2, 3, 4, 5, 6 and 7 in **a** and **b** represent the different code blocks respectively. Note that 2, 3, 4, 5, 6 in **a** and 2, 3, 4, 5, 6 in **b** are not the same block

```
#1 elfedit-elf_osabi
source code
static int elf_osabi
(const char *osabi)
{
  unsigned int i;
  for (i = 0; i <
ARRAY_SIZE (osabis);
i++)
    if (strcasecmp
(osabi, osabis[i].name)
== 0)
      return
osabis[i].osabi;
  error (_("Unknown
OSABI: %s\n"), osabi);
  return -1;
}
```

```
#2 x86-32-O2-elfedit-elf_osabi
pseudocode
int __usercall elf_osabi@<eax>
(const char *a1@<eax>)
{   const char *v1;
    int v2; const char *i;
    char *v5; v1 = a1;v2 = 0;
    for(i="none";;i=off_7DA4[2*v2])
    {  if(!strcasecmp(v1,i))
         return osabis[2*v2];
       if(++v2==16)
         break; }
    v5=dcgettext(0,"Unknown
OSABI:%s\n",5);
    error(v5, (char)v1);
    return -1;
}
```

```
#3 x86-64-O0-elfdit-elf_osabi
pseudocode
signed __int64 __fastcall
elf_osabi(const char *a1)
{   char v2;
    unsigned int i;
    for(i=0;i<=0xF;++i)
    {
if(!strcasecmp(a1,off_207B88[2
*i]))
            return*((unsigned
int*)&osabis+4*i);
    }
    v2=(unsigned
__int64)gettext("Unknown OSABI:
%s\n");
    error(v2);
    return 0xFFFFFFFFLL;
}
```

Fig. 2 Motivating example: the elf_osabi source code and pseudo-code

the corresponding pseudo-code, we would not need to consider the challenges posed by different compilers, compilation optimization options, and instruction architectures. This observation led us to explore the feasibility of extracting binary pseudo-code for binary code similarity analysis.

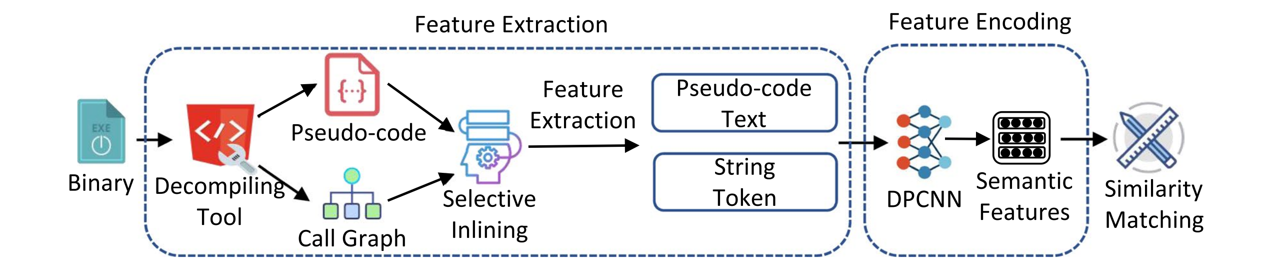


Fig. 3 An overview of our system

To answer this question, we propose and implement a pseudo-code encoding model, which enables encoding of pseudo-code and its application to binary code similarity analysis tasks.

Methodology

In this section, we present the workflow and implementation details of our proposed pseudo-code-based binary similarity detection tool, UPPC, which uses deep neural networks to convert pseudo-codes into vectors and apply them to binary code similarity analysis.

System overview

The goal of UPPC is to obtain embedding vectors of binary functions and to perform binary code similarity analysis. Figure 3 shows the overview of UPPC, which contains two main steps. The *Feature Extraction* step takes binary as an input and uses a decompilation tool to extract the pseudo-code and function call graph. Based on extraction result, UPPC extracts function features (i.e., the pseudo-code *Text* feature and string *Token* feature) from the pseudo-code with selective inlining. The *Feature Encoding* step feeds the extracted features into a deep neural network to obtain the function's semantic embedding vector. The vector can be used for tasks such as function similarity detection, function search, and vulnerability matching, which will be discussed in detail later.

Feature extraction

We extract the function's feature in five steps as explained in the next five subsections respectively.

Decompiling

Like most existing approaches (Eschweiler et al. 2016; Hu et al. 2013; Jang et al. 2013; Egele et al. 2014; Pewny et al. 2015), we apply a powerful binary analysis tool *IDA Pro 7.5* to process binary files, use plugin *IDA Python* to extract features, and analyze binary files automatically. With the programming interface provided by *IDA Python*, we can easily extract the function pseudo-code as well as the function call graph (CG) from the binary file. The most important point is that the analysis of binary files can be done offline as a preprocessing process.

Pseudo-code extraction

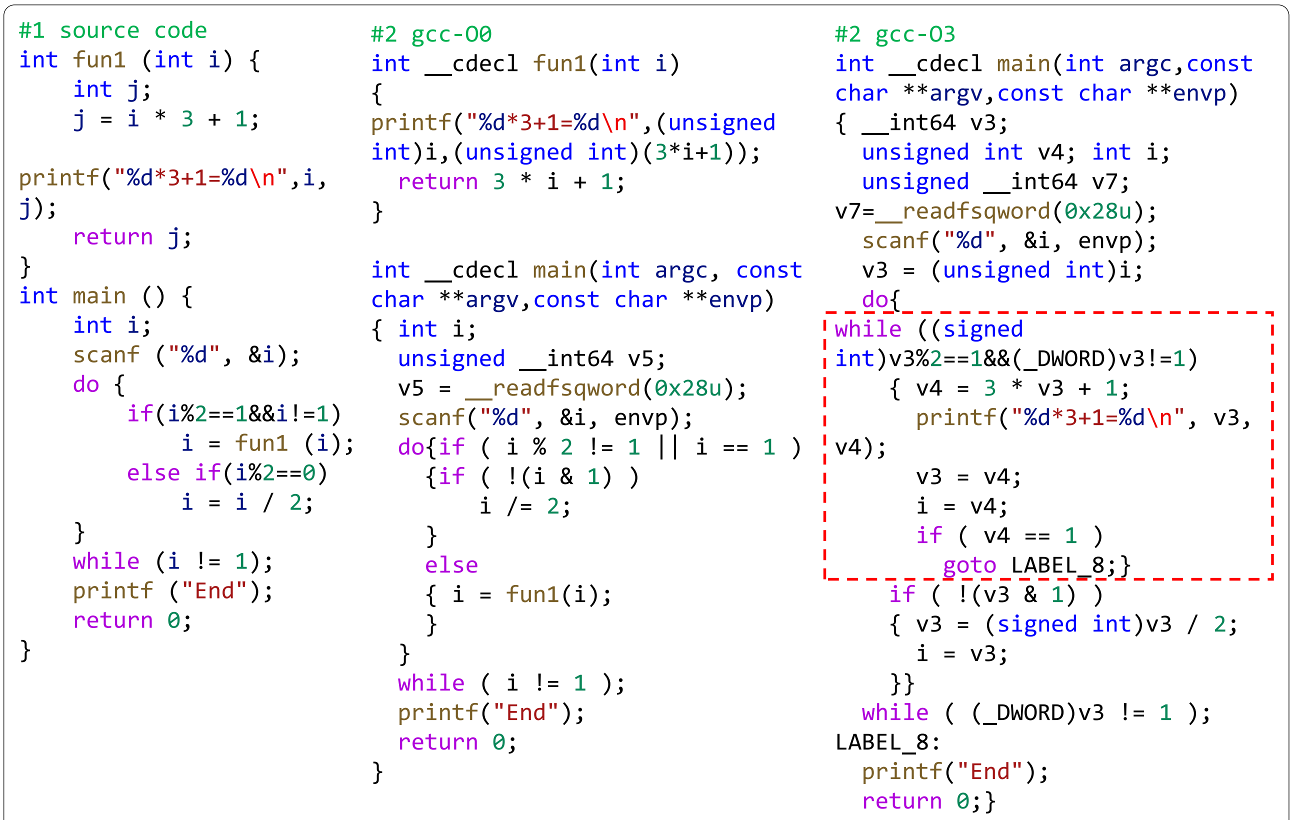
As we have previously described, pseudo-code is similar to source code and has rich semantic and syntactic features. As shown in Fig. 2, pseudo-code is more abstract than assembly code and intermediate code and is platform-independent. Using *IDA Pro*, we can easily extract the pseudo-code corresponding to the functions in the binary file.

Call graph generation

A call graph (CG) is a graph representing the invocation relationships between methods (functions) throughout a program. The nodes in the graph are methods, the edges represent invocation relationships, the starting node of an edge is called the caller, and the destination point represents the callee. The use of function call graphs allows inter-procedural analysis of a program, which helps to understand the complete logic of the program and to understand the complete semantics of the function.

Selective function inlining

Function inlining is a compiler optimization technique that replaces function call instructions with the body of the calling function and is generally used for functions that can be executed quickly (Chandramohan et al. 2016). Function inlining improves the speed of the program by avoiding the overhead associated with function calls (e.g. parameter passing, return value passing), but function inlining modifies the CFG structure of the program, so function inlining is also one of the challenges of binary code similarity search. Most existing approaches consider information about individual functions in isolation, and the called function is not considered part of the semantics of the calling function, which can result in some semantic information being lost when executing binary code similarity analysis.



```
#1 source code
int fun1 (int i) {
    int j;
    j = i * 3 + 1;

printf("%d*3+1=%d\n",i,
j);
    return j;
}
int main () {
    int i;
    scanf ("%d", &i);
    do {
        if(i%2==1&&i!=1)
            i = fun1 (i);
        else if(i%2==0)
            i = i / 2;
    }
    while (i != 1);
    printf ("End");
    return 0;
}
```

```
#2 gcc-O0
int __cdecl fun1(int i)
{
printf("%d*3+1=%d\n",(unsigned
int)i,(unsigned int)(3*i+1));
  return 3 * i + 1;
}

int __cdecl main(int argc, const
char **argv,const char **envp)
{ int i;
  unsigned __int64 v5;
  v5 = __readfsqword(0x28u);
  scanf("%d", &i, envp);
  do{if ( i % 2 != 1 || i == 1 )
    {if ( !(i & 1) )
        i /= 2;
    }
    else
    { i = fun1(i);
    }
  }
  while ( i != 1 );
  printf("End");
  return 0;
}
```

```
#2 gcc-O3
int __cdecl main(int argc,const
char **argv,const char **envp)
{ __int64 v3;
  unsigned int v4; int i;
  unsigned __int64 v7;
v7=__readfsqword(0x28u);
  scanf("%d", &i, envp);
  v3 = (unsigned int)i;
  do{
while ((signed
int)v3%2==1&&(_DWORD)v3!=1)
    { v4 = 3 * v3 + 1;
      printf("%d*3+1=%d\n", v3,
v4);
      v3 = v4;
      i = v4;
      if ( v4 == 1 )
        goto LABEL_8;}
    if ( !(v3 & 1) )
    { v3 = (signed int)v3 / 2;
      i = v3;
    }}
  while ( (_DWORD)v3 != 1 );
LABEL_8:
  printf("End");
  return 0;}
```

Fig. 4 The function inline example

Existing solutions inline all user-defined functions, which can lead to an explosion in function code size (Chang et al. 1992; Wang et al. 2015). However, not all functions are closely related to the function calling them. BinGo (Chandramohan et al. 2016) proposes a selective inlining technique in binary similarity comparisons that captures the full function semantics by inlining related libraries and user-defined functions. FCDetector (Fang et al. 2020) uses a similar technique in source code similarity detection, extracting function caller-callee graphs to analyze the caller-callee relationships between methods. Asm2vec (Ding et al. 2019) uses the PV-DM algorithm in the NLP field to represent semantic and structured information in binary and uses the function inlining method proposed by Bingo to analyze functions.

We use function inlining methods similar to BinGo, Asm2vec, and FCDetector for static analysis. Bingo uses function call patterns to guide function inlining, it inlines all library functions, and we do not consider library functions when inlining functions. Bingo makes recursive inline calls when inlining functions, and here we do the same as Asm2vec, we only consider first-order inlining, which makes the caller's functions more statically similar. The in-degree of a node in the CG indicates the number of times the function has been called. Furthermore, we only inline functions whose in-degree is equal to 1 ($indegree(f) = 1$), disregard inline library functions, functions with an in-degree of 1 are more likely to be inlined at compile time.

As shown in the Fig. 4, the same source code (#1 in Fig. 4) compiled with different compilation optimisation options will result in pseudo-code with different CG structures (#2 and #3 in Fig. 4), the higher optimisation option O3 will inline the *func1* function called in the *main* function when compiled (red box in Fig. 4 #3), the optimization option O0 does not. The *main* function in code snippet #2 in Fig. 4 calls four functions (*__readfsqword, scanf, func1, printf*), but only *func1* is not a library function, and *func1* is called only once in the entire binary, so we will inline *func1* into the *main* function. Furthermore, we will only consider first-order inlining and will not consider continuing to recursively inline the other functions called in *func1*. With function inlining,

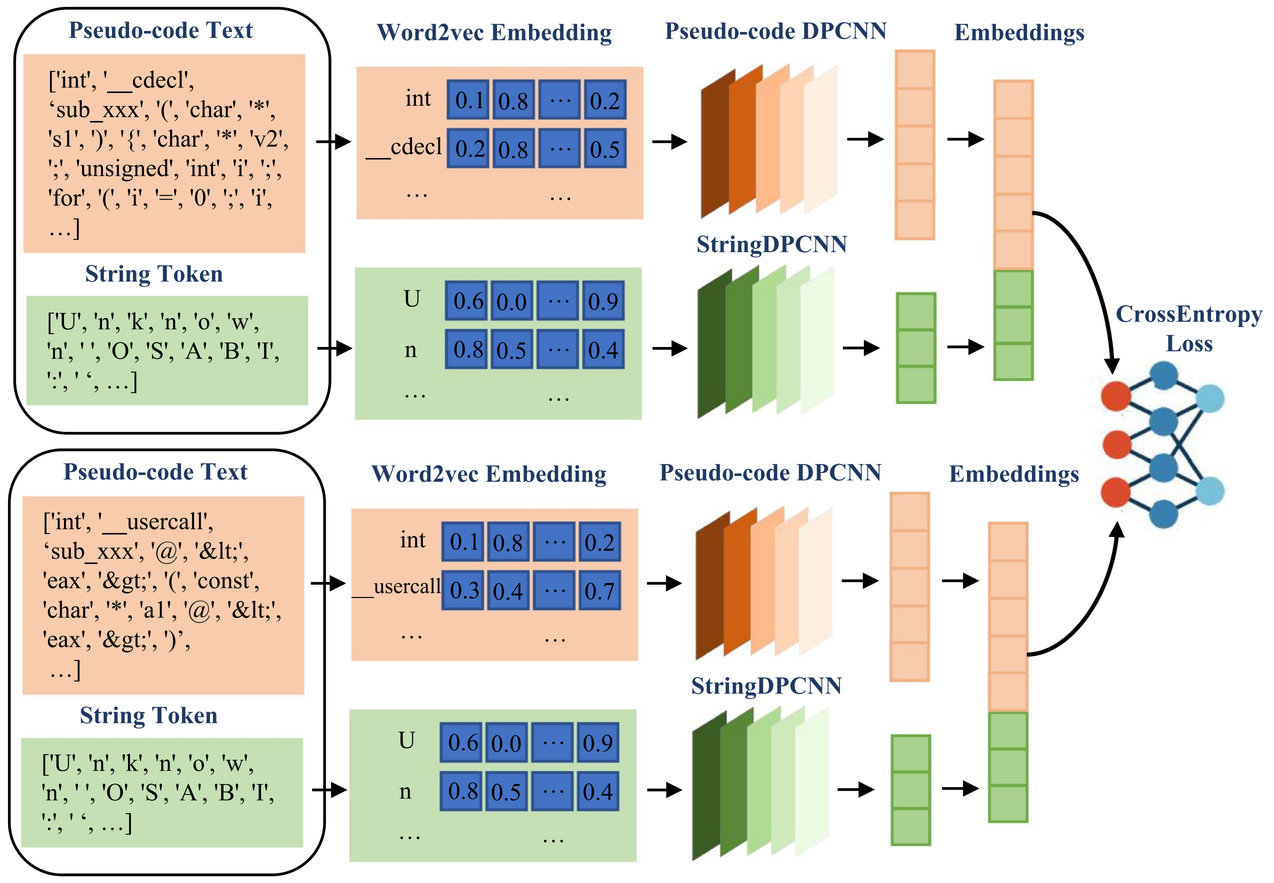

Pseudo-code Text
['int', '__cdecl', 'sub_xxx', '(', 'char', '*', 's1', ')', '{', 'char', '*', 'v2', ';', 'unsigned', 'int', 'i', ';', 'for', '(', 'i', '=', '0', ';', 'i', …]
String Token
['U', 'n', 'k', 'n', 'o', 'w', 'n', ' ', 'O', 'S', 'A', 'B', 'I', ':', ' ', …]
Word2vec Embedding
int 0.1 0.8 ⋯ 0.2
__cdecl 0.2 0.8 ⋯ 0.5
U 0.6 0.0 ⋯ 0.9
n 0.8 0.5 ⋯ 0.4
Pseudo-code DPCNN
StringDPCNN
Embeddings
CrossEntropy Loss
Pseudo-code Text
['int', '__usercall', 'sub_xxx', '@', '&lt;', 'eax', '&gt;', '(', 'const', 'char', '*', 'a1', '@', '&lt;', 'eax', '&gt;', ')', …]
String Token
['U', 'n', 'k', 'n', 'o', 'w', 'n', ' ', 'O', 'S', 'A', 'B', 'I', ':', ' ', …]
Word2vec Embedding
int 0.1 0.8 ⋯ 0.2
__usercall 0.3 0.4 ⋯ 0.7
U 0.6 0.0 ⋯ 0.9
n 0.8 0.5 ⋯ 0.4
Pseudo-code DPCNN
StringDPCNN
Embeddings

Fig. 5 The Siamese framework for function encoding

the *main* function in code snippet #2 will be more similar to the *main* function in #3.

Feature extraction

As shown in the Fig. 3, For the pseudo-code extracted by the decompiler, we use Txl[1] to parse it and extract the corresponding pseudo-code *Text* information and string information from it. Since pseudo-code has natural language properties like source code (Hindle et al. 2016), we treated the pseudo-code as a *Text* sequence without considering the structural features in the code, and our experiments showed that the overall structural features of the code could be learned by a global deep convolutional network. We also did not normalize the pseudo-code because previous work (Singh 2021) has shown that some features in the source code have been smoothed out after the source code has been compiled, and features such as variable names and variable definitions have also been normalized by the decompilation tool processing. Finally, we found that string features in the decompiled code are also important for understanding function semantics, so we extracted the string features separately, converted them into *Token* sequences, and used a deep learning network to determine the similarity between two strings.

Feature encoding

The general framework of our proposed deep learning model for decompiling function encoding is shown in Fig. 5, which takes as input a sequence of pseudo-codes *Text* and a sequence of strings *Token*. We use the DPCNN network to extract the semantic features of the string *Token* and the pseudo-code *Text*. It is important to note that the parameters of the DPCNN differ in different networks. Finally, we concatenate together the semantic feature vectors of the string and the pseudo-code to obtain the semantic embedding vector of the whole function.

We used the *Siamese* network architecture to train a deep learning model for pseudo-code similarity detection. The *Siamese* network is commonly used to measure

[1] Txl: Txl is a unique programming language specifically designed to support computer software analysis and source transformation tasks. https://www.txl.ca/.

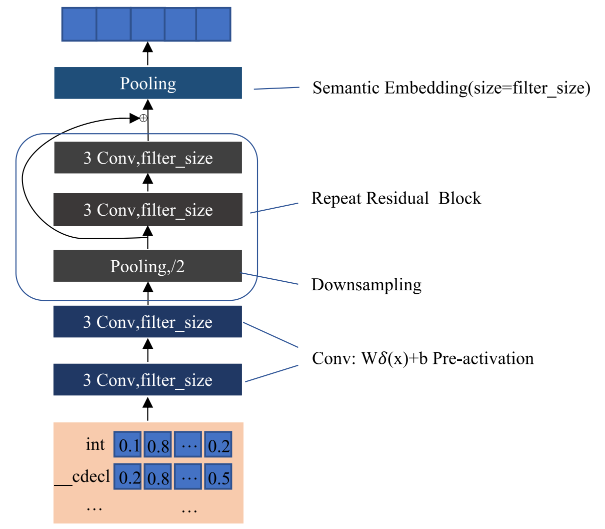


Fig. 6 The DPCNN model for extracting code and string semantics

the similarity of two inputs, and the *Siamese* network has the same configuration between the two networks, i.e. the same parameters and weights, and the parameters are updated simultaneously on both subnets during training. It is well known that *Siamese* networks are widely used in existing work (Fang et al. 2020; Xu et al. 2017b; Massarelli et al. 2019) on code similarity detection.

We use similar and dissimilar code pairs as input to our model to train the *Siamese* network. In the construction of the code pairs, we consider codes compiled from the same source code to be similar and codes compiled from different source codes to be dissimilar. Since the final output of our model has only two cases, similar and dissimilar, we choose cross-entropy as our loss function. For each pair of inputs, we predict similarity with probability p and dissimilarity with probability $1-p$, so the loss function is:

$$\begin{aligned} L &= \frac{1}{N}\sum_i L_i \\ &= \frac{1}{N}\sum_i -\left[y_i \cdot \log\left(p_i\right) + \left(1-y_i\right)\cdot \log\left(1-p_i\right)\right] \end{aligned}$$

y_i: denotes the label of sample i, positive class is 1, negative class is 0, and p_i denotes the probability that sample i is predicted to be in a positive class.

Our encoding model for code *Text* sequences and string *Token* sequences is shown in Fig. 6. We use the Deep Pyramid Convolutional Neural Networks (Johnson and Zhang 2017) (DPCNN) network to embed the feature. DPCNN is the first word-level, a widely effective convolutional neural network for the semantic classification of text. Compared to other deep learning models, DPCNN is a low-complexity word-level deep convolutional neural network (CNN) that can effectively model long-term dependencies in text and is suitable for capturing semantic dependencies in pseudo-code sequences.

For the input pseudo-code text embedding vector features, we first input the sequence to a region embedding layer. The result of a convolution layer containing a multi-size convolution filter is called a region embedding, meaning an embedding resulting from a set of convolution operations on a text region/fragment (e.g. 3gram). As different from the original model, we use a model that preserves word order, i.e. we set up a set of two-dimensional convolution kernels of $size = 3 * D$ to convolve 3-grams (D is the word embedding size). After two equal-length convolutional layers, the embedding vectors were used as input to several iterative DPCNN blocks, each with one downsampling pooling layer and two equal-length convolutional layers, and one downsampling pooling layer. Finally, we use a global average pooling layer to generate the final semantic representation of the function.

Similarity matching

In practice, we use the trained DPCNN network to encode the pseudo-code of the function to obtain the semantic embedding vector of the function, and use this vector for the task of function similarity detection. In more detail, when comparing the similarity of two functions f_1,f_2, the vectors $\overrightarrow{f_1}$, $\overrightarrow{f_2}$ of the two functions are obtained by using the same function embedding network DPCNN and these vectors are then compared using the cosine similarity as a distance metric with the following equation:

$$\text{similarity}\left(\overrightarrow{f}_1, \overrightarrow{f}_2\right) = \frac{\sum_{i=1}^{n}\left(\overrightarrow{f}_1[i]\cdot\overrightarrow{f}_2[i]\right)}{\sqrt{\sum_{i=1}^{n}\overrightarrow{f}_1[i]}\cdot\sqrt{\sum_{i=1}^{n}\overrightarrow{f}_2[i]}} \quad (1)$$

where $\overrightarrow{f}_1[i]$ indicates the i-th component of the vector $\overrightarrow{f_1}$

Evaluation

In this section, we demonstrate experimentally the effectiveness of our method UPPC and prove that our tool has better accuracy than existing tools. Our purpose is to investigate the following four research questions (RQs) through experimental evaluation:

- *RQ1* How accurate is UPPC in matching semantically similar functions in different architectures and optimizations?
- *RQ2* How accurate is UPPC in performing function searches with different architectures, optimization options, compilers, and obfuscations?
- *RQ3* How accurate is UPPC in vulnerability search tasks?
- *RQ4* How can string features and function inlining help improve the accuracy of UPPC semantically similar function matching?

Implementation and setup

Baseline

There have been many previous works using deep learning for binary code similarity analysis including Gemini (Xu et al. 2017b), Genius (Feng et al. 2016), Asm2Vec (Ding et al. 2019), TREX (Pei et al. 2020), Codee (Yang et al. 2021), Order Matter (Yu et al. 2020b), CodeCMR (Yu et al. 2020a), SAFE (Massarelli et al. 2019), OSCA (Peng et al. 2021), etc. However, many of the tools are not open source and we do not have access to the source code of these tools. The results of experiments with deep learning-based tools are highly data-dependent, so we used three tools with access to source code, Asm-2Vec (Ding et al. 2019), Gemini (Xu et al. 2017b) and SAFE (Massarelli et al. 2019), as the baseline for our experiments.

Metrics

We choose precision (P), *Recall* (R) and F1 measure (F1) as the evaluation of function matching, where *TP* predict positive class as positive, *FN* predict positive class as negative, *FP* predict negative class as positive, and *TN* predict negative class as negative, we calculate P, R and F_1 as follows:

$$P = \frac{TP}{TP + FP} \quad R = \frac{TP}{TP + FN} \quad F_1 = \frac{2PR}{P + R}$$

In the similarity function match task, we use cosine similarity to measure the similarity between two function embedding vectors, the cosine similarity takes any real number between -1 and 1, and we use the receiver operation characteristic curve (ROC) to measure the false positives/true positives of the model under different thresholds. the area under the ROC curve (AUC) is used to measure the accuracy of the model embedding vector, the larger the AUC, the better the model accuracy. In function search tasks, we use *Top-K* accuracy to measure model accuracy, *Top-K* is also known as emphRecall@K. *Top-K* accuracy is used to calculate the proportion of correct results among the top K results with the highest probability among the predicted results.

Table 1 The total number of functions in the OpenSSL dataset and the decompile time

Total functions	Train function	Test function	Train pair	Test pair	Decompile time
4056	3244	812	119258	29620	320 (min)

Training details

We construct our dataset in a similar way to Gemini and SAFE, where our data contains two types: similar and dissimilar, where binary functions compiled from the same source code are similar and otherwise considered dissimilar so that during data pre-processing we consider all functions with the same name to be similar (except for the *main* function) and otherwise not. We extracted all functions from the binary and partitioned the data set by function name at a rate of 80% and 20% so that there was no intersection between the training and test data sets during the partitioning process. For training, we trained 10 epochs, set the small batch size to 128, used the Adam (Kingma and Ba 2014) optimization algorithm, and set the learning rate to 0.001.

Experimental settings

We conduct all the experiments on an AMAX computing server. It has two 2.1GHz 24-core CPUs, four NVIDIA GeForce RTX 2080ti GPUs, and 384G memory.

Function semantic classification

In this section, we attempt to answer *RQ1* with experimental results. We generate the dataset in a similar way to Gemini. Compile OpenSSL-1.0.1f and OpenSSL-1.0.1u to different architectures (x86, ARM, MIPS) using GCC-7.5 combined with different optimization options (O0-O3). We use P, R, F_1 as measures for model training and the same *ROC* as Gemini to measure UPPC semantic classification accuracy.

Table 1 summarises the total number of functions we have used for training in OpenSSL. We keep the function name information at compile-time, so we can sort them by the function name, with the same function names belonging to the same type. We extracted a total of 4056 functions from OpenSSL, each with approximately 24 functions (a combination of 2 binary version, 3 architectures and 4 optimization options), which may be less than 24 due to optimizations such as function inlining. We split the data into training and testing datasets by function name, which ensures that the same function does not appear in both training and testing datasets. When constructing data pairs, functions with the same name

Table 2 UPPC uses semantic classification results from different deep learning models and their combinations

Dataset-pseudo-codeModel-StringModel	P	R	F1	Training time (min)
OpenSSL-DPCNN-DPCNN	**0.953**	**0.953**	**0.953**	**154**
OpenSSL-LSTM-DPCNN	0.943	0.943	0.943	267
Openssl-Transformer-DPCNN	0.865	0.848	0.846	279
OpenSSL-DPCNN-LSTM	0.950	0.950	0.950	237

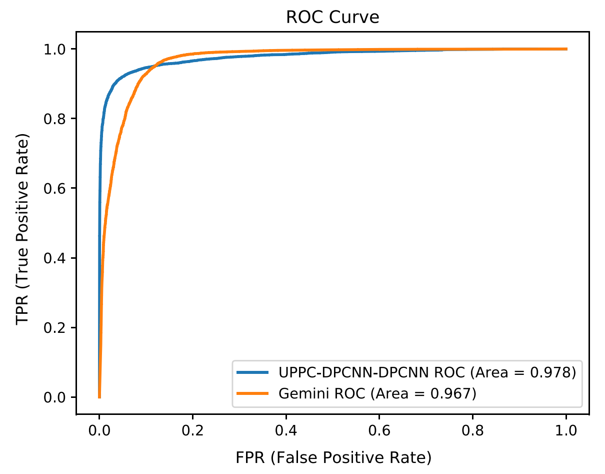


Fig. 7 ROC curves for UPPC, Gemini on the test set

are similar and functions with different names are dissimilar. As shown in the Table 1, it takes about 320 min to extract and decompile the pseudocode for 24 OpenSSL binaries, which are about 68.2 MB in size, and about 13 min for a single Binary file.

UPPC mainly contains pseudo-code *Text* embedding models and string *Token* embedding models, and here we consider the results of DPCNN (Johnson and Zhang 2017) , Transformer (Vaswani et al. 2017) and LSTM (Hochreiter and Schmidhuber 1997) and their combinations. Table 2 shows the semantic classification results for different deep learning models and their combinations, where the pseudo-code generates an embedding vector size of 128 and the string embedding vector size is 64. As can be seen from Table 2, the best results for function semantic classification (F1:0.953) are obtained when the DPCNN model is used on both pseudo-code and string, so we use this model in all subsequent experiments. Transformer-based accuracy is low (F1:0.846) because our limited resources make it difficult for us to train Transformer models with large parameters. In addition, training the DPCNN-based model is the fastest, and it takes about 154 min to train 10 Epochs.

PAs Fig. 7 shows the ROC of different models, our model achieves better results than Gemini , in the best case, the UPPC-DPCNN-DPCNN model achieves an AUC of 0.978, compared to 0.967 for Gemini. It should be noted that the AUC of Gemini is obtained after we re-divided the data set according to the ratio of 80% and 20% and trained 100 epochs, which took about 486 min to retrain the model. The experimental results show that UPPC outperforms Gemini in terms of code similarity detection across architectures and optimization options.

Now, we answer *RQ1* UPPC matches similar functions in different architectures and optimizations more accurately than the existing tools, UPPC has better sense classification performance and can distinguish between similar and dissimilar binary functions.

Function search

In this section, we will answer *RQ2*, the function search accuracy of UPPC in real-world software. We evaluate the function search accuracy of UPPC under different architectures, compilers, compilation options, and code obfuscation, using *Recall*@1 as a measure of search accuracy.

We evaluated UPPC by extracting seven common items with a high number of functions from the binary code similarity analysis benchmark provided by Kim et al. (2020). We cleaned up the dataset we used to avoid incorrect or biased results, filtered short functions with less than 10 lines of assembly code and pseudo-code, and selected only functions in the code (.text) segment, as functions in other segments may not contain valid binary code. Finally, we obtained the number of functions per item as shown in Table 3, we trained a new model with this function search dataset. We use Asm2Vec as the baseline for our experiments. The authors have integrated Asm2Vec into Kam1n0[2] and there is no need to consider the training of the Asm2Vec model during function search, as Asm2Vec is an unsupervised self-learning model. For our experiments we configured Asm2Vec according to the suggested parameters.

Cross-compile search

In this experiment, we tested the similarity code search function of UPPC between different compilers Clang-7.0 and GCC7.3.0, and between different versions of the same compiler Clang7.0 and Clang4.0. In our experiments, we generated X86-64 bit binaries with the same optimization options (O0) with different compilers. The data we used are shown in Table 3, the total number of

[2] Kam1n0 is a scalable assembly management and analysis platform: https://github.com/McGill-DMaS/Kam1n0-Community.

Table 3 Number of functions per project after filtering

			libgmp	tar	gawk	libunistring	gcal	binutils	xorriso	Total functions
GCC-7.3.0	X86-64	O0	110	867	833	487	478	1669	2073	6517
Clang-4.0	X86-64	O0	110	765	913	483	476	1688	2183	6618
Clang-7.0	ARM-64	O0	116	954	887	496	484	1905	2179	7021
	X86-32	O0	99	855	847	498	475	1529	2124	6427
	X86-64	**O0**	**110**	**764**	**913**	**483**	**476**	**1687**	**2166**	**6599**
		O2	52	548	590	418	461	1156	1850	5075
		O3	53	543	586	417	461	1158	1846	5064
		BCF	136	944	1059	558	493	1996	2522	7708
		SUB	112	766	928	483	476	1711	2199	6675
		FLA	133	886	1047	507	485	1905	2406	7369
		ALL	143	970	1097	562	485	2049	2586	7892

Table 4 Function search results, using the *Recall*@1 metric

Software	Tools	Compile		Optimization		Arch		Obfuscation			
		Clang&Gcc	Clang& 7.0&4.0	O0&O3	O2&O3	X86&ARM	X86 32&64	BCF	SUB	CFF	ALL
libgmp -6.1.2	UPPC	**0.944**	**1.000**	**0.968**	**1.000**	**0.936**	**0.990**	**0.804**	**1.000**	**0.873**	**0.836**
	Asm2Vec	0.550	0.667	0.448	0.816	–	0.504	0.580	0.797	0.779	0.568
tar-1.30	UPPC	**0.948**	**0.993**	**0.865**	**0.989**	**0.736**	**0.956**	**0.808**	**0.990**	**0.93**	**0.876**
	Asm2Vec	0.850	0.585	0.258	0.726	–	0.496	0.476	0.557	0.501	0.355
gawk -4.2.1	UPPC	**0.902**	**0.987**	**0.654**	**0.979**	**0.828**	**0.939**	**0.736**	**0.978**	**0.897**	**0.797**
	Asm2Vec	0.506	0.561	0.305	0.499	–	0.478	0.507	0.552	0.523	0.402
libunistring -0.9.10	UPPC	**0.693**	**0.930**	**0.749**	**0.983**	**0.623**	**0.834**	**0.538**	**0.899**	**0.649**	**0.536**
	Asm2Vec	0.233	0.302	0.137	0.335	–	0.250	0.228	0.302	0.229	0.177
gcal-4.1	UPPC	**0.356**	**0.969**	**0.557**	**0.963**	**0.566**	**0.820**	**0.708**	**0.962**	**0.881**	**0.697**
	Asm2Vec	0.198	0.550	0.204	0.624	–	0.251	0.488	0.533	0.447	0.357
binutils -2.30	UPPC	**0.893**	**0.999**	**0.739**	**0.975**	**0.768**	**0.927**	**0.679**	**0.983**	**0.879**	**0.788**
	Asm2Vec	0.710	0.756	0.369	0.683	–	0.651	0.665	0.735	0.669	0.465
xorriso -1.4.8	UPPC	**0.828**	**0.993**	**0.682**	**0.963**	**0.808**	**0.914**	**0.641**	**0.957**	**0.797**	**0.690**
	Asm2Vec	0.696	0.768	0.505	0.515	–	0.680	0.680	0.764	0.681	0.484
Average	UPPC	**0.795**	**0.982**	**0.745**	**0.979**	**0.752**	**0.911**	**0.702**	**0.967**	**0.844**	**0.746**
	Asm2Vec	0.535	0.598	0.318	0.600	–	0.473	0.518	0.607	0.547	0.401
Total Average	UPPC:**0.842**					Asm2Vec:0.510					

functions obtained by Clang-7.0_X86-64_O0, Clang-4.0_X86-64_O0, and GCC-7.3.0 _X86-64_O0 are 6599, 6618, and 6517 respectively, the total number of functions makes little difference.

As shown in the *Compiler* column in Table 4, the average result of UPPC on different compilers (0.795) is significantly better than Asm2Vec (0.535) , and UPPC achieve better results on different versions of the same compiler (Clang 7.0, Clang 4.0) with 0.982. The experimental results show that UPPC is more robust in binary similarity search across compilers, achieving a *Recall* of 0.795 even on different compilers.

In addition, it is clear from the experimental results that the binary codes obtained from different compilers are more different and harder to identify. the code *Recall* for both tools on different compilers is lower than the *Recall* on different versions of the same compiler.

Cross-optimization search

In this experiment, we use Clang 7.0 on the same architecture (X86-64) with different compilation optimization strategies (O0, O2, O3) to explore UPPC's ability to function section in different compilation options.

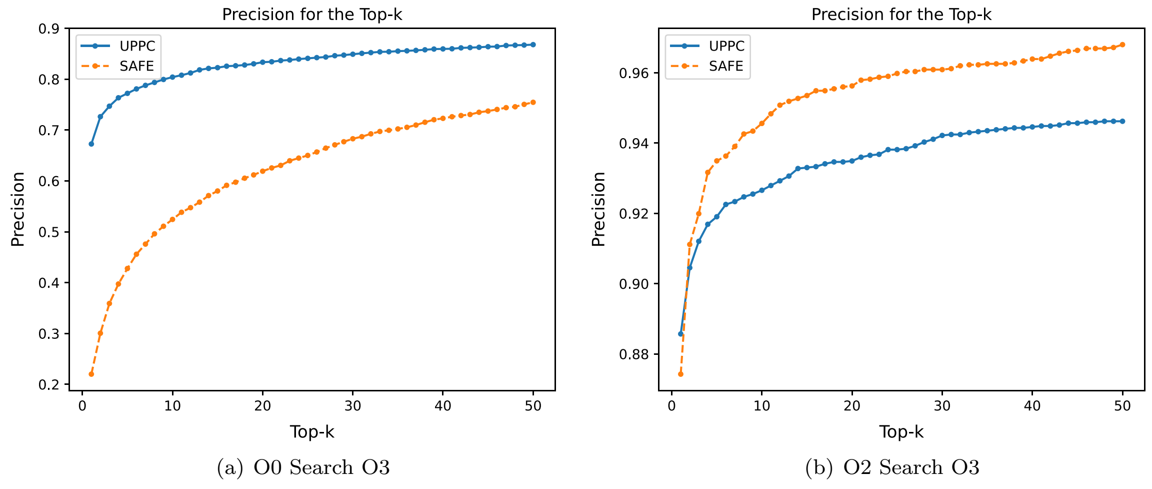


Fig. 8 Precision of openssl-1.1.1-pre8 search for Top-K, K ≤ 50

The data generated by the different compilation options are shown in Table 3. For the same compiler and architecture (Clang-7.0_X86_64), the different compilation options yield different numbers of functions (O0:6599, O2:5075, O3:5064). Higher optimization levels will result in fewer functions, in addition higher optimization levels contain all optimization strategies of lower levels, as more optimization techniques are applied, more functions are inlined, so higher optimizations levels get fewer functions. Therefore, when the optimization strategies are quite different (such as O0, O3), the detection of the similarity function is more difficult.

The results are shown in the *Optimization* column of Table 4. We searched O3 with O0 and O2, and UPPC achieved a better *Recall*@1 in both cases. When searching for O0 with O3, UPPC's *Recall*@1 is significantly better than Asm2Vec, with *Recall* 0.427 higher than Asm2Vec. Asm2Vec has difficulty detecting code fragments with large differences due to compilation optimization, and they only have good *Recall* in the case of small differences like O2&O3 with a *Recall* of 0.600.

Furthermore, we retrained our UPCC model with the X86_64 dataset in Table 3 and OpenSSL-1.1.1-pre8 compiled with GCC-5.4 (same as SAFE). Then, we use SAFE and UPPC to conduct code search experiments on the OpenSSL-1.1.1-pre8 cross-compile option dataset (the number of functions is O0:4471, O2:4002, O3:3933), and the experimental results are shown in Fig. 8. As shown in Fig. 8, on the O0 O3 search dataset with large differences, the similar code searchability of UPPC is significantly higher than that of SAFE. The Top-1 of UPCC and SAFE are 0.673 and 0.220, respectively, and the Top-50 are 0.868 and 0.758, respectively. However, on the O2 O3 search dataset with less variance, the similar code searchability of UPPC is only slightly higher than SAFE on Top-1, and the Top-1 of UPCC and SAFE are 0.886 and 0.874, respectively.

The experimental results show that it is more difficult to detect when there are large differences in the optimization strategies. Furthermore, as can be seen from the average rows of Table 4, apart from code obfuscation, compilation optimization (O0&O3) has the greatest impact on function search in UPPC, with a low *Recall*@1 of 0.745.

Cross-architecture search

In this experiment, we used Clang 7.0 with the O0 optimization option to compile and generate binaries for different architectures (X86-64, ARM-64) and different bits of the same architecture (X86-64, X86-32) to explore UPPC's ability to search for similar functions in different architectures. As shown in Table 3, the difference between the number of functions of the same architecture (Clang-7.0_X86-64_O0:6599, Clang-7.0_X86-32_O0:6427) is smaller than the number of functions of different architectures (Clang-7.0_ARM-64_O0:7021).

The search results on different architectures are shown in the *Architecture* column of Table 4. UPPC achieves better results on different architectures and different bits of the same architecture, with mean Recal@1 values of 0.752 and 0.911 respectively. Asm2Vec does not support cross-architecture code search, so we only tested the results of Asm2Vec on different bits of the same architecture, the Recal@1 of SAFE is lower in these case (0.473).

The experimental results in Table 4 show that similar code searches between different architectures are more difficult and UPPC achieves better accuracy in the more difficult cross-architecture similar code searches.

Table 5 Real-world vulnerability analysis results (Top-k)

CVE	Vulnerable function	Total	UPPC/true positives (k)	Asm2Vec/true positives (k)	SAFE/true positives (k)	Gemini/true positives (k)
2014-0160	dtls1_proc- ess_heartbeat	15	14	**15**	12	7
2014-6271	initialize_sh- ell_variables	9	**9**	7	6	4
2015-3456	fdctrl_handle_drive_s- pecification_command	6	**6**	**6**	4	2
2014-9259	configure	7	**7**	**7**	2	4
2014-7169	parse_and_execute	3	**3**	**3**	1	3
2014-4877	snmp_usm_pass- word_to_key_sha1	7	**7**	**7**	5	6
2014-4877	ftp_syst	7	**5**	**5**	**5**	3
2015-6826	ff_rv34_decod- e_init_thread_copy	7	**7**	**7**	**7**	**7**
Total		61	**58**	57	42	36
*Recall*Rate			**0.951**	0.934	0.689	0.590
Embedding Times (s)			**172**	1980	2341	1652

Code obfuscation search

To search for obfuscated code, we extracted obfuscated X86-64 bit binaries compiled with Clang 7.0 and Obfuscator-LLVM (O-LLVM) from the dataset. O-LLVM uses three different obfuscation techniques and their combinations: Bogus Control Flow (BCF), Control Flow Flattening (CFF), and Instruction Substitution (SUB). In addition, *ALL* (BCF+CFF+SUB) means that all of the above obfuscation options are used. The number of functions obtained after obfuscation is shown in Table 3. Compared to the O0 (6599) optimization option, obfuscation increases the number of functions, with the largest increase in functions after all obfuscation operations are applied (BCF:7709, SUB:6675, FLA:7369, ALL:7892). The reason is that obfuscation changes the control structure of assembly instructions, one function may be split into several, so functions become numerous.

Obfuscation can severely modify the structure of a program and code searches can be more difficult. We evaluated the performance of UPPC on the obfuscated binaries and the results are shown in the Obfuse column of Table 4. UPPC achieves the best *Recall@*1 (0.967) on SUB and decompiling with equivalent instruction substitution yields a similar pseudo-code, so SUB has the least impact on UPPC. BCF changes the logic of the program, the subgraph structure of the program, and the structure of the disassembled code, so of the three obfuscations, BCF has the greatest impact on UPPC (0.702). Of course, when a combination of all three obfuscations is used together, it also changes the control structure of the program, CFG features, etc., making the program more different, so all obfuscations used together have the impact on UPPC, with *Recall@*1 0.746

As can be seen from the Obfuse column of Table 4, Among the three obfuscation methods, BCF has the greatest impact on Asm2Vec, and the function search Recall@1 is the lowest (0.518). Asm2Vec extracts the assembly instruction sequence in the CFG by random walk, and adding garbage code will have a greater impact on the extracted sequence. Interestingly, like the results in OSCA (Peng et al. 2021), we found in our experiments that the test results of Asm2Vec are lower than those in the paper. Since the author does not disclose its source code, but only provides an integrated testing tool, we cannot conduct a detailed analysis.

Now, we answer *RQ2* Overall, UPPC (0.842) outperformed the existing tools Asm2Vec (0.510) in terms of function search, with code search regarding obfuscation being more difficult, followed by cross-compiler options, then cross-architecture, and cross-compiler code similarity search being the least difficult.

Vulnerability search

In this section we experimentally evaluate the ability of UPPC to find vulnerable functions on a real-world dataset, answering *RQ3*.

In this experiment, we use the vulnerability dataset provided by David et al. (2016), which contains eight CVE vulnerability functions and some normal functions. The data is compiled with the compilers GCC, Clang, and ICC in combination with different compilation options, resulting in different function variants. We evaluate the accuracy of the model using *Top-K*, where *K* equals the total number of relevant vulnerabilities, e.g. Heartbleed has 15 variants and we calculate the percentage of true positives in Top-15. We test the accuracy of the model on the pretrained model as in the experimental setup of SAFE and Gemini.

The details of each vulnerability and the results of our experiments are shown in Table 5. We filtered out functions with less than 10 lines of assembly code and pseudo-code and ended up with 2769 usable functions,

Table 6 A dataset for evaluating the impact of each component of the UPPC on semantic classification

	Train data					**Test data**		
Dataset	sg3utils	findutils	usbutils	coreutils	utillinux	binutils	inetutils	diffutils
Binary count	504	32	16	832	720	118	208	32
Total count	2104					358		
Pairs	119272					19177		

of which 61 were functions related to vulnerability and another 2708 were functions without vulnerability. In the experiments, vulnerable and normal functions were used as targets and randomly selected vulnerable functions were used as queries, returning the true positive *Top-k* of the query.

Both UPPC and Asm2Vec can achieve better recall rates on this vulnerability dataset, 95.1% and 93.4%, respectively, because both UPPC and Asm2Vec support the identification of similarity functions under different obfuscations, different compilation optimizations, and different compilers. Compared with SAFE, UPPC achieves poor results on wget, only finding 5 vulnerabilities, and missing one relevant vulnerability on Heartbleed. In addition to this, UPPC achieved an excellent *Recall* rate (100%) on the other 6 vulnerabilities. SAFE achieved an excellent *Recall* rate (100%) only on FFmpeg, same performance on wget as UPPC, and was worse than UPPC in all other cases. Gemini has the lowest accuracy, Gemini can only correctly recall 59.0% of the vulnerabilities, while the recall accuracy of UPPC, Asm2Vec, and SAFE are 95.1%, 93.4% and 69.8%, respectively. In addition, as shown in the *Embedding Times* row of the Table 5, UPPC took the least total time to obtain the vector of 2769 functions, just 172 seconds.

Now, we answer *RQ3* UPPC also achieves 95.1% accuracy when searching for vulnerabilities in real-world application scenarios, with a better vulnerability *Recall* than existing tools.

Ablation study

In this section, we experimentally answer *RQ4* How can string features and function inlining help improve the accuracy of UPPC semantically similar function matching. We explored the impact of the individual components used in UPPC and their combinations on semantic classification, and we used P, R, and F_1 as metrics for our evaluation.

The same library code may be shared between different binaries under the same project, as the library code may be statically linked to the binaries during the compilation process, which will affect the results of our experiments. We need to ensure that there is no bias in the dataset and

Table 7 The semantic classification results of UPPC using different features and their combinations

Features and methods			**P**	**R**	**F1**
Psudocode	**Function Inline**	**String**			
√			0.722	0.746	0.723
√	√		0.854	0.848	0.848
√	√	√	**0.928**	**0.928**	**0.928**

that there is no overlap between the training set and the testing set. Therefore, we extracted the corresponding *main* functions from the different binaries under each project for our dataset. The *main* functions represent the main functions of the binaries and are implemented differently in the different binaries, so we can ensure that there were differences between our data.

The datasets we used are shown in Table 6. We compiled the different projects into X86-64 bit binaries using the compilers GCC and Clang combined with the compilation optimization options (O0-O3). We divided the datasets into two groups by project, and the binaries for each group are shown in Table 6, thus ensuring that there is no overlap between the training and test sets in the division.

The effect of string features and function inlining on UPPC is shown in Table 7, where all components work in UPPC's favor, improving its accuracy in performing semantic classification tasks. When only pseudo-code features are used, semantic classification is not as good, with an F1 of only 0.732. From *RQ2* we learn that as more optimization techniques are applied, more functions are inlined and therefore fewer functions are produced, so using function inlining to capture the function logic can improve the accuracy of the model (F1:0.848). Finally, using additional string features is more helpful to understand the semantics of the program, and using both function inlining and string features can achieve the best classification accuracy, with the highest F1 (0.923).

Now, we answer RQ3 *RQ4* All components are beneficial to UPPC and improve its accuracy to perform semantic binary classification tasks, and we design UPPC to better capture the semantic features of the code.

Related work

Binary code similarity detection has always been one of the hot topics in the field of network security, and binary code similarity detection is also the basis of binary analysis (Haq and Caballero 2021). The binary similarity is often used as a auxiliary analysis technique. It is widely used in tasks such as vulnerability detection (Xu et al. 2017b; Eschweiler et al. 2016; Chandramohan et al. 2016; Feng et al. 2016; Liu et al. 2018; Gao et al. 2018; Huang et al. 2017; David et al. 2016; David and Yahav 2014), malware identification (Hu et al. 2013; Jang et al. 2013; Hu et al. 2009; Bruschi et al. 2006; Cesare et al. 2013) and patch analysis (Xiao et al. 2021; Dullien and Rolles 2005; Gao et al. 2008; Hu et al. 2016; Xu et al. 2017a; Kargén and Shahmehri 2017) etc.

Several binary similarity detection tools have been proposed, and Sæbjørnsen et al. (2009), one of the pioneers of binary code search, proposed a framework for binary code clone detection using a function modeling technique based on normalized grammars (i.e. normalized operands). We classify binary similarity detection methods into machine learning-based and traditional methods.

*Traditional methods*Jiang et al. (2020) propose a binary optimization framework, IMPTO, to re-optimise lifted code to mitigate the impact of optimization. Tang et al. (2020) proposed LibDX, a platform-independent and fully automated system for detecting reused libraries in binary files. Hu et al. (2018) propose a semantic-based hybrid approach to detecting binary clone functions. The execution of the function is simulated, semantic features are extracted during the execution, and semantic features are used for similarity comparison. David et al. (2018) propose a static, precise, and scalable technique for finding CVEs (Common Vulnerabilities and Exposures) is stripped firmware images. Luo et al. (2014) combine strict program semantics with fuzzy matching based on the longest common subsequence to propose a binary-oriented, confusion-resistant method for comparing binary code similarity. David et al. (2017) decompose binary into comparable fragments and use a compiler optimizer to convert them into a canonical, normalized form so that equivalent fragments can be found by simple syntactic comparison. Kargén and Shahmehri (2017) propose a new method based on aligned binary code tracking using dynamic time warping and information retrieval techniques for binary similarity analysis. Chandramohan et al. (2016) use selective function inlining techniques to capture the complete function semantics by inlining related libraries and user-defined functions. A scalable and powerful dichotomous search engine, Bingo, supporting various architectures and operating systems was implemented. David and Yahav (2014) decompose functions into tracelets: continuous, short, partial traces performed to calculate the similarity between functions. Pewny et al. (2015) convert binary code into intermediate representations to generate assignment formulas with input and output variables, sample-specific inputs to observe the I/O behavior of basic blocks, and thus grasp the function semantics and solve the problem of incomparable instruction sets between different CPUs. Structural similarity in binary code can be represented on different graphs [e.g. control flow graph (CFG), call graph (CG)]. Most structural similarity methods examine changes in graph isomorphism, which involve methods such as K-subgraph matching, path similarity, and graph embedding. KKMRV2005 (Kruegel et al. 2005) proposes to divide a graph into k subgraphs, where each subgraph contains only k connected nodes.

ML-based methods Gemini (Xu et al. 2017b) proposed the use of graph embedding as a machine learning concept to do binary code similarity analysis. Inspired by the great success of deep learning in natural language understanding, there is a growing body of exploratory work on understanding programming languages by incorporating code structures into Deep neural networks. Yang et al. (2021) propose an unsupervised tensor embedding scheme Codee that uses an NLP-based neural network to generate semantic-aware token embeddings, learning semantic information about instructions and control flow structure information to generate basic block embeddings. All basic block embeddings are used in the function to obtain a variable-length function feature that allows efficient code search. Pei et al. (2020) learn the execution semantics explicitly and automatically from micro traces of binary functions and migrate the learned knowledge to match semantically similar binary functions. Yu et al. (2020b) used the improved BERT model to convert the function CFG nodes into semantic embedding vectors, then used the improved MPNN model to obtain the semantic structure embedding vectors of CFG, and then used the CNN model to obtain the sequential embedding vectors of CFG nodes, and finally integrated the two as the final function embedding vectors for function similarity measurement. Relying on code semantic information and program-wide control flow information to generate basic block embeddings, Duan et al. (2020). propose an unsupervised program-wide code representation learning technique to perform code similarity analysis. Zuo et al. (2018) used the technique of (Neural Machine Translation) NMT: using word embeddings to represent instructions and then LSTM to encode instruction embeddings and instruction dependencies, and proposed a new neural network-based tool for basic block similarity comparison, INNEREYE. Liu et al. (2018) proposed a solution to the binary cross-version search problem with a deep neural network (DNN), using three semantic features, namely intra-functional, inter-functional, and inter-module features, using a DNN to extract the intra-functional binary features. Li et al. (2019) used graph neural network generation for binary function similarity matching, and calculated the similarity of graph embeddings through matching based on cross-graph attention;

Although state-of-the-art deep learning-based binary code similarity detection tools have shown impressive achievements, their effectiveness relies heavily on well-labeled training data, and deep learning-based models act as a black box, and experimental results are lacking some degree of interpretability. Furthermore, the performance of BCSA trained with one dataset may not be as effective in detecting similarity codes from another dataset.

Singh (2021) propose a technique for clone detection using compiler optimization. They compiled the source code into a binary executable by optimizing it with the compiler optimization option and then converted it into decompiled code by a decompiler tool. They found that the compilation optimization smoothed out high-level features between different source codes, thus making the programs more similar in structure for the same task and more conducive to code classification and code clone detection. Our work differs from theirs in that we extract the binary decompiled code for binary code similarity detection.

Discussion

In this section, we discuss the limitations and potential solutions to our work, as well as ideas that can still be explored in the future.

A limitation of our work is that our tools are based on decompiled pseudo-code, benefiting from the work of countless researchers working on decompiling. The pseudo-code we use is extracted using the IDA Pro tool, and although IDA Pro is widely used for binary security research, the version of IDA Pro we are currently using only supports decompilation for a limited number of architectures (X86,ARM,MIPS, etc.), imposing a significant limitation on our work. At present, the main processor in PC is X86, MIPS is widely used in embedded devices, and ARM is widely used in mobile field, ARM, MIPS and X86 architecture are the three architectures with the highest market share. Therefore, UPPC is still suitable for most closed source software, UPPC is highly extensible, and we believe that as decompilation tools become more powerful, UPPC will be able to support more architectures.

Currently, in UPPC we only consider pseudo-code features of binary functions, string features, and function inlining methods, which has some limitations. In the future, we will further consider intermediate representation features, assembly code features, constant features, control flow graph features, etc. Although we have only considered some of these features so far, the accuracy has exceeded existing tools and a large number of function semantic embeddings can be generated quickly using the available features.

Conclusion

In this paper, we present an accurate and robust approach to binary function similarity analysis: UPPC, which uses function inlining to obtain semantic embedding representations of functions from binary function pseudo-codes and strings and uses them for semantically similar function identification. The main innovation of our work is the use of deep pyramidal convolutional neural network (DPCNN) to learn the semantic features of binary decompiled function pseudo-codes and strings, and then to obtain a semantic vector of the entire function and use it for code matching, vulnerable searching, and other binary similarity analysis tasks. Our evaluation shows that UPPC has better semantic classification accuracy and better recall in searching for functions across compilers, architectures, optimization options, and obfuscation. In the future, we will explore additional binary program analysis tasks such as software composition analysis (SCA) on pseudo-code.

Acknowledgements
The authors would like to thank all the anonymous reviewers for their useful comments and suggestions.

Authors' contributions
All authors read and approved the final manuscript.

Authors' information
Yinxing Xue is now a research professor in the Department of Computer Science and Technology, University of Science and Technology of China (USTC). He has published nearly 40 papers as lead author in top level international journals and conferences, such as journals TIFS, TSE, TOSEM, A conferences CCS, USSENIX security, ICSE, FSE, ASE, ISSTA, etc., and has served as an external reviewer for 30 conferences or seminars.

Funding
Not applicable.

Availability of data and materials
All code and data can be found at: https://github.com/UnPPCode/uppc.

Declarations

Competing interests
The authors declare no competing interests.

Author details
[1]School of Computer Science and Engineering, University of Science and Technology of China, Hefei, China. [2]School of Computer Science and Engineering, Nanyang Technological University, Singapore, Singapore. [3]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. [4]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China.

Received: 21 December 2021 Accepted: 8 March 2022
Published online: 01 December 2022

References

Alon U, Zilberstein M, Levy O, Yahav E (2019) code2vec: learning distributed representations of code. Proc ACM Program Lang 3(POPL):1–29

Alrabaee S, Debbabi M, Shirani P, Wang L, Youssef AM, Rahimian A, Nouh L, Mouheb D, Huang H, Hanna A (2020) Binary analysis overview

Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E (2007) Comparison and evaluation of clone detection tools. IEEE Trans Softw Eng 33(9):577–591

Bruschi D, Martignoni L, Monga M (2006) Detecting self-mutating malware using control-flow graph matching. In: International conference on detection of intrusions and malware, and vulnerability assessment. Springer, pp 129–143

Cesare S, Xiang Y, Zhou W (2013) Control flow-based malware variant detection. IEEE Trans Depend Secure Comput 11(4):307–317

Chandramohan M, Xue Y, Xu Z, Liu Y, Cho CY, Tan HBK (2016) Bingo: Cross-architecture cross-os binary search. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, pp 678–689

Chang PP, Mahlke SA, Chen WY, Hwu W-MW (1992) Profile-guided automatic inline expansion for c programs. Softw Pract Exp 22(5):349–369

David Y, Yahav E (2014) Tracelet-based code search in executables. ACM Sigplan Not 49(6):349–360

David Y, Partush N, Yahav E (2016) Statistical similarity of binaries. ACM SIGPLAN Not 51(6):266–280

David Y, Partush N, Yahav E (2017) Similarity of binaries through re-optimization. In: Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation, pp 79–94

David Y, Partush N, Yahav E (2018) Firmup: precise static detection of common vulnerabilities in firmware. ACM SIGPLAN Notices 53(2):392–404

Ding SH, Fung BC, Charland P (2019) Asm2vec: boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: 2019 IEEE symposium on security and privacy (SP). IEEE, pp 472–489

Duan Y, Li X, Wang J, Yin H (2020) Deepbindiff: learning program-wide code representations for binary diffing. In: Network and distributed system security symposium

Dullien T, Rolles R (2005) Graph-based comparison of executable objects (English version). Sstic 5(1):3

Egele M, Woo M, Chapman P, Brumley D (2014) Blanket execution: dynamic similarity testing for program binaries and components. In: 23rd $\{$USENIX$\}$ security symposium ($\{$USENIX$\}$ security 14), pp 303–317

Eschweiler S, Yakdan K, Gerhards-Padilla E (2016) discovre: efficient cross-architecture identification of bugs in binary code. In: NDSS, vol 52, pp 58–79

Fang C, Liu Z, Shi Y, Huang J, Shi Q (2020) Functional code clone detection with syntax and semantics fusion learning. In: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, pp 516–527

Feng Q, Zhou R, Xu C, Cheng Y, Testa B, Yin H (2016) Scalable graph-based bug search for firmware images. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pp 480–491

Gao D, Reiter MK, Song D (2008) Binhunt: Automatically finding semantic differences in binary programs. In: International conference on information and communications security. Springer, pp 238–255

Gao J, Yang X, Fu Y, Jiang Y, Sun J (2018) Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary. In: 2018 33rd IEEE/ACM international conference on automated software engineering (ASE). IEEE, pp 896–899

Haq IU, Caballero J (2021) A survey of binary code similarity. ACM Comput Surv (CSUR) 54(3):1–38

Hindle A, Barr ET, Gabel M, Su Z, Devanbu P (2016) On the naturalness of software. Commun ACM 59(5):122–131

Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780

Huang H, Youssef AM, Debbabi M (2017) Binsequence: fast, accurate and scalable binary code reuse detection. In: Proceedings of the 2017 ACM on Asia conference on computer and communications security, pp 155–166

Hu X, Chiueh T-c, Shin KG (2009) Large-scale malware indexing using function-call graphs. In: Proceedings of the 16th ACM conference on computer and communications security, pp 611–620

Hu X, Shin KG, Bhatkar S, Griffin K (2013) Mutantx-s: scalable malware clustering based on static features. In: 2013 $\{$USENIX$\}$ annual technical conference ($\{$USENIX$\}$ $\{$ATC$\}$ 13), pp 187–198

Hu Y, Zhang Y, Li J, Gu D (2016) Cross-architecture binary semantics understanding via similar code comparison. In: 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER), vol 1. IEEE, pp 57–67

Hu Y, Zhang Y, Li J, Wang H, Li B, Gu D (2018) Binmatch: a semantics-based hybrid approach on binary code clone analysis. In: 2018 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 104–114

Jang J, Woo M, Brumley D (2013) Towards automatic software lineage inference. In: 22nd $\{$USENIX$\}$ security symposium ($\{$USENIX$\}$ security 13), pp 81–96

Jiang J, Li G, Yu M, Li G, Liu C, Lv Z, Lv B, Huang W (2020) Similarity of binaries across optimization levels and obfuscation. In: European symposium on research in computer security. Springer, pp 295–315

Johnson R, Zhang T (2017) Deep pyramid convolutional neural networks for text categorization. In: Proceedings of the 55th annual meeting of the association for computational linguistics (Volume 1: Long Papers), pp 562–570

Kargén U, Shahmehri N (2017) Towards robust instruction-level trace alignment of binary code. In: 2017 32nd IEEE/ACM international conference on automated software engineering (ASE). IEEE, pp 342–352

Kim D, Kim E, Cha SK, Son S, Kim Y (2020) Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. arXiv:2011.10749

King JC (1976) Symbolic execution and program testing. Commun ACM 19(7):385–394

Kingma DP, Ba J (2014) Adam: a method for stochastic optimization. arXiv:1412.6980

Kruegel C, Kirda E, Mutz D, Robertson W, Vigna G (2005) Polymorphic worm detection using structural information of executables. In: International workshop on recent advances in intrusion detection. Springer, pp 207–226

Laguë B, Proulx D, Mayrand J, Merlo EM, Hudepohl J (1997) Assessing the benefits of incorporating function clone detection in a development process. In: 1997 Proceedings international conference on software maintenance. IEEE, pp 314–321

Li Y, Gu C, Dullien T, Vinyals O, Kohli P (2019) Graph matching networks for learning the similarity of graph structured objects. In: International conference on machine learning. PMLR, pp 3835–3845

Lindorfer M, Di Federico A, Maggi F, Comparetti PM, Zanero S (2012) Lines of malicious code: insights into the malicious software industry. In: Proceedings of the 28th annual computer security applications conference, pp 349–358

Liu C, Chen C, Han J, Yu PS (2006) Gplag: detection of software plagiarism by program dependence graph analysis. In: Proceedings of the 12th ACM SIGKDD international conference on knowledge discovery and data mining, pp 872–881

Liu B, Huo W, Zhang C, Li W, Li F, Piao A, Zou W (2018) $\alpha $diff: cross-version binary code similarity detection with DNN. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, pp 667–678

Luo L, Ming J, Wu D, Liu P, Zhu S (2014) Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, pp 389–400

Luo Z, Wang B, Tang Y, Xie W (2019) Semantic-based representation binary clone detection for cross-architectures in the internet of things. Appl Sci 9(16):3283

Massarelli L, Di Luna GA, Petroni F, Baldoni R, Querzoni L (2019) Safe: Self-attentive function embeddings for binary similarity. In: International conference on detection of intrusions and malware, and vulnerability assessment. Springer, pp 309–329

Newsome J, Song DX (2005) Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In: NDSS, vol 5, pp 3–4 (**Citeseer**)

Pei K, Xuan Z, Yang J, Jana S, Ray B (2020) Trex: learning execution semantics from micro-traces for binary similarity. arXiv:2012.08680
Peng D, Zheng S, Li Y, Ke G, He D, Liu T-Y (2021) How could neural networks understand programs? arXiv:2105.04297
Pewny J, Garmany B, Gawlik R, Rossow C, Holz T (2015) Cross-architecture bug search in binary executables. In: 2015 IEEE symposium on security and privacy. IEEE, pp 709–724
Sæbjørnsen A, Willcock J, Panas T, Quinlan D, Su Z (2009) Detecting code clones in binary executables. In: Proceedings of the eighteenth international symposium on software testing and analysis, pp 117–128
Singh S (2021) Leveraging compiler optimization for code clone detection. In: Proceedings of the 33rd international conference on software engineering and knowledge engineering
Tang W, Luo P, Fu J, Zhang D (2020) Libdx: a cross-platform and accurate system to detect third-party libraries in binary code. In: 2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 104–115
Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. Adv Neural Inf Process Syst 30
Walker A, Cerny T, Song E (2020) Open-source tools and benchmarks for code-clone detection: past, present, and future trends. ACM SIGAPP Appl Comput Rev 19(4):28–39
Wang X, Jhi Y-C, Zhu S, Liu P (2009) Behavior based software theft detection. In: Proceedings of the 16th ACM conference on computer and communications security, pp 280–290
Wang M, Yin H, Bhaskar AV, Su P, Feng D (2015) Binary code continent: finer-grained control flow integrity for stripped binaries. In: Proceedings of the 31st annual computer security applications conference, pp 331–340
Xiao Y, Xu Z, Zhang W, Yu C, Liu L, Zou W, Yuan Z, Liu Y, Piao A, Huo W (2021) Viva: Binary level vulnerability identification via partial signature. In: 2021 IEEE international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 213–224
Xu Z, Chen B, Chandramohan M, Liu Y, Song F (2017a) Spain: security patch analysis for binaries towards understanding the pain and pills. In: 2017 IEEE/ACM 39th international conference on software engineering (ICSE). IEEE, pp 462–472
Xu X, Liu C, Feng Q, Yin H, Song L, Song D (2017b) Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pp 363–376
Yang J, Fu C, Liu X-Y, Yin H, Zhou P (2021) Codee: a tensor embedding scheme for binary code search. IEEE Trans Softw Eng
Yu Z, Zheng W, Wang J, Tang Q, Nie S, Wu S (2020a) Codecmr: cross-modal retrieval for function-level binary source code matching. Adv Neural Inf Process Syst 33:3872–3883
Yu Z, Cao R, Tang Q, Nie S, Huang J, Wu S (2020b) Order: matters Semantic-aware neural networks for binary code similarity detection. In: Proceedings of the AAAI conference on artificial intelligence, vol 34, pp 1145–1152
Zhang X, Gupta N, Gupta R (2006) Pruning dynamic slices with confidence. ACM SIGPLAN Not 41(6):169–180
Zhang J, Wang X, Zhang H, Sun H, Wang K, Liu X (2019) A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE). IEEE, pp 783–794
Zuo F, Li X, Young P, Luo L, Zeng Q, Zhang Z (2018) Neural machine translation inspired binary code similarity comparison beyond function pairs. arXiv:1808.04706

Publisher's Note