Sapienza
Università di Roma

Binary Code Similarity Detection

Facoltà di Informatica

Corso di Laurea Magistrale in Cybersecurity

Candidate

Roberto Garzone
ID number 1991589

Thesis Advisor

Prof. Leonardo Querzoni

Academic Year 2023/2024

Thesis not yet defended

Binary Code Similarity Detection
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: robertogarzone0@gmail.com

Abstract

Nowadays, among the techniques that are gaining more and more ground in the world of cybersecurity, and in particular in the fields of malware analysis, vulnerability detection, copyright violation detection and bug detection, we find the techniques of Binary Code Similarity Detection (BCSD).
This thesis analyzes the three approaches on which BCSD techniques are based: text-based, logic-based and semantic-based, highlighting their use cases, such as SAFE or BinGo tools, advantages, such as the computation speed of fingerprinting and N-Gram techniques (text-based) or the robustness with respect to the most modern obfuscation techniques of BERT or GNN models (semantic-based), and disadvantages, such as the computational complexity of CFGs (logic-based) or the sensitivity to the syntactic structure of the binary code of text-based approaches.
After a careful analysis, these approaches are compared in order to highlight the problems of each of them and define which are the most effective future directions. The comparisons highlight that there is no ideal approach in an absolute sense, but rather an ideal approach for a specific scenario.

Contents

Chapter 1

Introduction

In this chapter we want to give an overview of Binary Code Similarity Detection (BCSD) techniques, the context in which they were developed and their main fields of application.
It is also intended to define the contribution of this paper to the existing literature and how it is structured.

1.1 Definition and Context

With the inexorable increasing of the diffusion of devices connected and the diversification of hardware architectures, it has become increasingly necessary to find similarities between different binary codes. These codes can be derived from the different sources or from the same source, compiled for different hardware architecture or with different compiler optimizers.
This condition creates a problem for the traditional detection methods. In order to surpass that problems are implemented Binary Code Similarity Detection (BCSD) techniques.
These techniques have proven, over time, to be a valuable direction for cybersecurity research.
BCSD techniques have proven useful in several areas: eliminating redundancy in code, identifying vulnerabilities in software, ensuring copyright compliance and analysing malware.

Eliminate code redundancy: The initial aim of BCSD techniques was to be able to identify, by studying the similarity of functions within a binary code, and eliminate redundant code within a piece of software. This brings benefits not only from the point of view of computational and maintenance cost, but also from the point of view of identifying vulnerabilities.

Detecting software vulnerabilities: as mentioned above, the presence of redundant code inevitably leads to vulnerabilities. BCSD techniques support developers in identifying software vulnerabilities, bringing about a significant improvement in software security.

Copyright Protection: In addition to improving their code, software development companies can leverage BCSD techniques to identify copyright violations for their products, thereby increasing the effectiveness of copyright protection.

Malware Analysis: Another use of BCSD techniques, which has made them such an important field of research, is in malware analysis activities. In malware analysis, BCSD techniques can be used to compare the code of the sample under analysis with that of known malware. This simplifies the identification of malware families and their general attack behaviour. BCSD techniques are also widely used for cloud security. In fact, the possibility of analysing uploaded code is crucial for the identification of potentially malicious code.

The execution process of BCSD techniques can be divided into three stages, as shown in fig. 1.1:

Code processing: the aim of this stage is to increase detection efficiency and accuracy by reducing code size and noise. For this purpose, irrelevant instructions are eliminated, code blocks are extracted and symbols are renamed.

Comparison unit generator: in order to be able to perform the code similarity calculation, it is necessary to convert the binary code into a comparable representation. Examples of such a representation are byte streams, hash values or feature vectors.

Similarity computation: the last stage is the actual similarity calculation operation. For this purpose, there are different methodologies, necessary due to the presence of different units of comparison, such as the calculation of vector distances, the comparison of sub-graphs or clustering algorithms.

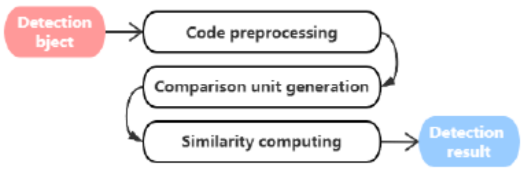


Figure 1.1. BCSD Processing

These techniques could be grouped into the following macro-categories of approaches, as shown in fig. 1.2:

text-based: This approach has the advantage of being able to operate directly on the binary code under analysis. This makes the implementation of the preprocessing and similarity calculation steps simpler than the other approaches. The problem with this approach is its inefficiency in the case of obfuscated code or code with renamed variables.

logic-based: This approach uses structured data such as sequences, trees or graphs to represent the data flow or control flow. The aim is to calculate the similarity of the code by identifying blocks of the same code with the same logic.

semantic-based: This approach uses natural language analysis or image recognition techniques, in order to learn the semantics of a program, through the use of ML (Machine Learning) or DNN (Deep Neural Networks) technologies.

This approach performs a comparison of embedded vectors. An operation that requires a significant use of computational resources and a large number of procedures. The complexity of this approach allows cross-instructions and different compiler optimisations to be performed.[1]

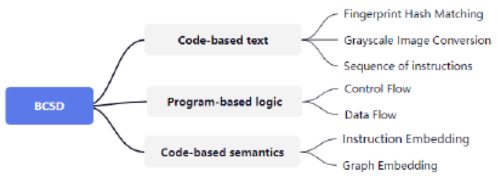


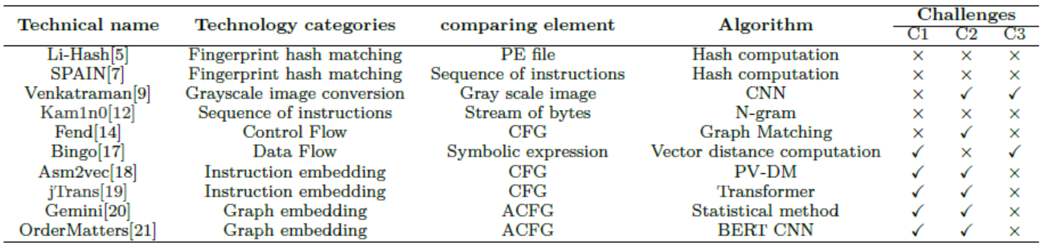
Figure 1.2. Classification of BCSD

Although a vast literature exists on the study and application of BCSD techniques, these often prove to be incomplete, inconsistent with the solution expressed in the papers they refer to or still only work for a specific dataset.

This condition leads to an inevitable difficulty in replicating the proposed techniques and comparing them with other techniques. Added to this, there is also the specificity of the contexts in which these techniques are applicable, such as objectives, degrees of detail, metrics used to represent results, or even variation in what is meant by similarity.

The combination of these issues has led to a fragmentation of the field of study and an increasing complexity of the techniques adopted; leading to the development of machine-learning-based techniques. By adopting this approach, it has been possible to increase the level of accuracy, effectiveness and efficiency of BCSD techniques, overcoming the main challenges faced: different compiler optimizations (C1), cross-architecture and cross-instruction analysis (C2), code obfuscation (C3). The table in fig.1.3 shows, for each main BCSD technique, which challenges it is able to overcome.

As can be seen from the table, BCSD techniques based on machine learning prove most successful in overcoming challenges.



Technical name	Technology categories	comparing element	Algorithm	Challenges		
				C1	C2	C3
Li-Hash[5]	Fingerprint hash matching	PE file	Hash computation	×	×	×
SPAIN[7]	Fingerprint hash matching	Sequence of instructions	Hash computation	×	×	×
Venkatraman[9]	Grayscale image conversion	Gray scale image	CNN	×	✓	✓
Kam1n0[12]	Sequence of instructions	Stream of bytes	N-gram	×	×	×
Fend[14]	Control Flow	CFG	Graph Matching	×	✓	×
Bingo[17]	Data Flow	Symbolic expression	Vector distance computation	✓	×	✓
Asm2vec[18]	Instruction embedding	CFG	PV-DM	✓	✓	×
jTrans[19]	Instruction embedding	CFG	Transformer	✓	✓	×
Gemini[20]	Graph embedding	ACFG	Statistical method	✓	✓	×
OrderMatters[21]	Graph embedding	ACFG	BERT CNN	✓	✓	×

Figure 1.3. C1 Compiler Optimization, C2 cross-architecture, C3 Obfuscation techniques, ✓means that the method can solve the challenge and ✗means that the method cannot.[1]

1.2 Contribution

The contribution of this paper is to provide a critical and in-depth analysis of existing Binary Code Similarity Detection (BCSD) techniques; focusing on how machine learning and deep learning technologies have influenced these techniques.

The focal points of this thesis are:

1. **Exploring basic and advanced BCSD techniques**: analysis of standard techniques, such as fingerprinting and graph matching, compared with more recent techniques based on semantic embedding, GNN (Graph Neural Network) and NMT (Neural Machine Translation) based models.

2. **Exploring the main challenges of the field**: analysis of the main problems that BCSD techniques face

3. **Evaluating the impact of Machine Learning techniques**: evaluation of the contribution made by machine learning techniques regarding the accuracy and effectiveness of BCSD techniques. The analysis includes techniques such as graph embedding, Siamese models and the use of architectures such as BERT[2] and Transformer[3].

4. **Propose future research directions**: based on the gaps in the current literature, we would like to suggest possible directions, such as the development of computationally more efficient methodologies or the integration and combination of static and dynamic techniques.

1.3 Importance of Binary Similarity in Cybersecurity

As stated earlier, BCSD techniques can also prove extremely useful in the field of cybersecurity. In this field, the most immediate application is certainly in malware analysis. This is because the constant proliferation of connected devices, such as smartphones and PCs, and the relentless growth in the use of IoT devices, have led to a significant increase in platform malware.

Such malware is compiled and distributed on different platforms, often appearing as slightly modified versions of the same code; with differences that can complicate analysis. In this regard, the use of techniques capable of verifying binary similarity and revealing malware variants with a similar structure proved necessary. The answer was found in BCSD techniques based on machine learning algorithms. An initial approach was made with techniques based on CFGs, which, however proved incapable of extrapolating all the structural features of the analyzed sample.

This problem was solved by combining Natural Language Processing (NLP) techniques with Graph Neural Networks (GNNs) techniques.

Through the NLP process approach, node problems were solved by simplifying the approach and improving the filtering of invalid nodes.

Three categories of analysis techniques exist in the current literature:

Graph matching-based solutions: based on graph matching, calculated by considering the distance of two CFGs as the similarity value of the corresponding pairs of features. Tools using this solution include BinGold[4], which extracts semantic information from CFGs and DFGs (Data Flow Graphs) and then calculates their similarity using graph matching algorithms, or BINGO[5], which introduces feature filtering prior to similarity calculation to eliminate irrelevant features.

Graph embedding-based solutions: This approach overcomes the problem of time and computational costs encountered in graph matching by converting graphs into multi-dimensional vectors. Among the main tools using this approach are Genius[6] and VulSeeker[7], both of which use the Structure2Vec[8] tool for conversion.

Deep learning-based solutions: this approach addresses the main problem of using graphs: namely that feature extraction must be manual, thus inevitably subject to bias. The use of Deep Learning techniques, such as Neural Networks, allow for autonomous and automatic feature extraction by using a learning-based approach to train the model in treating assembly instruction operands as tokens. Tools using this approach include SAFEsafe-malware and Asm2Vec[9].[10]

Another application of BCSD techniques concerns the identification of vulnerabilities within a code.

As we know, one of the most critical issues in software, whether open or closed source, is the presence of bugs. Such bugs can, in fact, lead to vulnerabilities such as memory corruption. Existing tools for scanning for bugs present obvious difficulties compared to the analysis of software used on multiple architectures due to two main problems. The first problem is having only the binary code of a piece of software available.

The second problem is having to identify bugs at the binary level for software compiled on multiple CPU architectures, such as ARM or x86. This poses a problem as different architectures have structural differences such as instruction sets, function offsets and function call conversions.

In order to overcome these issues, the solution was found in the use of BCSD techniques that, through a machine learning approach, are able to identify bug signatures within software that are potentially vulnerable to the bug identified in the signature itself.

In order to reduce the number of comparisons necessary between I/O pairs, the MinHash approach was preferred to verify the matching of code blocks, while with the CFG it was found to be possible to expand the check on the entire bug signature.[11]

1.4 Structure of the Paper

The thesis is structured around six chapters, each focusing on a fundamental aspect of Binary Code Similarity Detection (BCSD) techniques, emphasizing machine learning and deep learning approaches:

Chapter 1 - Introduction: this chapter introduces the concept of Binary Code Similarity Detection (BCSD), the techniques on which it is based, divided into macro-categories, and the problems they face.

Chapter 2 - Theoretical Foundations: this chapter analyses the fundamental concepts underlying BCSD techniques, such as the differences between binary code and source code, the importance of control flow and data flow analysis, and the main challenges involving BCSD.

Chapter 3 - State of the Art: This chapter reviews the existing literature on BCSD techniques; techniques that are divided into three categories: text-based approaches, logic-based approaches and semantic-based approaches. For each category and each technique, advantages and disadvantages are then discussed.

Chapter 4 - Challenges: this chapter discusses the main challenges to which BCSD techniques are subject: compiler optimisation, code obfuscation, cross-architecture and scalability of the techniques, and how the current literature attempts to solve them.

Chapter 5 - Future Perspectives: In this chapter, we focus on future directions that may lead to improvements in BCSD techniques. New techniques, the integration of static and dynamic code analysis, the use of more efficient models, large-scale datasets and the use of transfer learning techniques are analyzed in order to increase the generalization of BCSD processes.

Chapter 6 - Discussion and Conclusions: In this chapter, the previously analyzed topics are compared, showing the advantages, disadvantages and limitations of the individual approaches and techniques. Gaps in the current literature are also exposed. Finally, practical implications for BCSD techniques in the field of cybersecurity are presented.

Chapter 2

Theoretical Fundamentals

This chapter will address issues related to the development of knowledge to better understand Binary Code Similarity Detection (BCSD) techniques.
The topics addressed in this chapter start with an overview of Binary Code Similarity and an explanation of the differences between binary and source code, and end with a description of the main issues and challenges that BCSD techniques must overcome.

2.1 What Does Similarity Mean?

When talking about Binary Similarity, it's important to define which kind of similarity you mean. Indeed, there exist several variants of similarity concept, on the base of the assumption made on the source code. The most common definition on binary similarity is when the two binary codes become from the same source code, but it's not the only definition. A more complex definition is when the assumption admit the existence of similarity between binary codes becoming from different source codes. In that case, it's possible to identify some type of similarity bases:

Similarity based on patterns: the similarity definition is based on the context of the identification of codes or code blocks that perform the same operations. Definition based on the identification of similar patterns present if binary codes.

Similarity based on structures: the two codes are defined similar if they present the same flow structure, such as control flows or data flows. Techniques using this kind of similarity concept, search the similarity between binary codes analysing their logic.

Similarity based on behaviors: the most intuitive definition of similarity, it says that two codes are similar if the output is the same or if they follow the same process logic. The limit of this definition is about that, has demonstrated by the Turing's halting problem, it's impossible to implement an algorithm able to calculate, deterministically, if two programs are the same behavior. In order to surpass this limit, it's possible to use regression tests, that analyse program behavior in different scenarios, to create something that is near to the searched result.

2.2 Binary Code Similarity

Binary code similarity analysis is used to determine whether binary functions compiled from different source codes are similar or not.
A need that arose from the possibility of compiling the same source code for several architectures (x86, ARM, MIPS, etc.) and the presence of the same binary code within different existing programs or in several parts of the same program.
About the compilation on different architectures, it is known that these tend to have substantial structural differences, such as opcodes, operands, function calls and memory accesses, making it necessary to compare the similarity of the semantics of a known binary code with the semantics of an unknown code. For this purpose, David and Yahav calculate similarity on the basis of the number of overlapping code fragments in two sets of binary instructions. Huang et al., on the other hand, calculate the similarity between the lengths of the execution paths on CFGs (Control Flow Graphs).[12] As a result of the most recent research, the use of Graph Neural Networks based on CFGs has been shown to return more accurate binary code

similarity comparisons with significantly better performance. Among the tools using GNNs is Gemini[13], which is based on the manual extraction of features in order to calculate the similarity between blocks of features. Another approach is the one used by UFEBScite[14], which uses the RNN (Recurrent Neural Network) to be able to compute the semantics of basic blocks and convert CFGs into integrated graphs, supporting GNNs.

Considering two binary functions compiled for different platforms, the purpose of the BCS is to be able to define whether these functions can be associated with the same source code or not.

Despite the support provided by deep learning technology, achieving this goal presents some technical challenges:

Challenge 1: Instruction difference: instructions differ significantly between platforms. There are not only syntactic differences between binary codes generated from the same source code for different platforms, but also differences in feature extraction outputs.

Challenge 2: CFG difference: As far as CFGs are concerned, through disassemblers such as IDA, it can be seen that different platforms present differences in the number of basic blocks that make up the binary code, causing a different number of node states in the GNNs. This difference can cause a high number of false positives in the comparison results.

2.3 Differences between Binary and Source Code

Fundamental to understanding the process carried out by Binary Code Similarity Detection (BCSD) techniques is understanding what binary code actually is and how it differs from source code. The main difference between them is that the binary code is the result of the process of compiling the code of a program, called source code. The different abstraction level between binary and source code is the main reason why the calculation of similarity is made between binary codes. Indeed, source code, being written in a high-level language, presents a high level of abstraction; unlike binary code, which presents itself as a detailed set of instructions almost at the level of machine code. Binary code, instead, tends to present less information than source code and contributes to the optimization of the compilation process and management of instruction sets proposed by different architectures.
As shown in fig.2.1, the compilation process from source code to binary code is divided into five stages, or phases:

1. **Source code**: the starting point of the process is the source code. Written in a high-level language that allows easier implementation by the developer due to the use of a more human-like syntax and the use of a higher level of abstraction.

2. **Compilation**: this phase performs the translation of the source code from a high-level language to a set of instructions, defined on the basis of the target architecture of the compilation, written in a low-level language. This phase is particularly complex due to the sub-passages involved. In particular, there are three fundamental steps: lexical analysis, recognition of the symbols present in the source code, syntactic analysis, recognition of the instructions that define the behavior of the software, and semantic analysis, understanding the meaning of the analyzed instructions.

3. **Object Files**: a binary representation of the program, executable directly on the processor.[15]

4. **Linker**: is fundamental, as, at this stage, the object code is loaded into memory and subsequently executed, producing an executable file (.exe). **Binary program**: output of the linking phase, it is the final stage of the source code compilation process. In this. stage, an executable code, called binary code, is finally obtained.

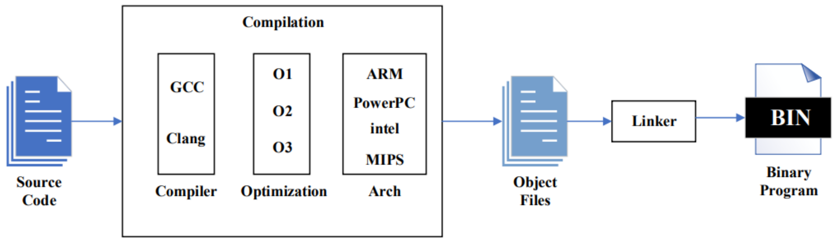


Figure 2.1. Compilation process

From the fig.2.1 it can be seen that the compilation phase has three elements. These elements, compiler, optimization and architecture, are fundamental to the entire compilation process.

Compiler: The same source code could be used for different versions of the same compiler or for different compilers. Conditions that may lead to the production of significantly different binary codes.

Compilation optimization options: during the compilation process, different compilation options may be adopted. These options, at different efficiency levels, simplify the binary code, for example by eliminating functions present in the source code but never used.

Target instruction architecture: Just as it is possible to compile source code with different compilers, or different optimizations, in the same way it is possible to compile the same source code for different platforms (such as x86, ARM or MIPS). Obviously, each platform brings with it specific structural characteristics, such as instruction sets, memory accesses and opcodes; hence, it is easy for binary codes compiled for different platforms to present significant structural differences.

2.4 Uses of Binary Similarity Detection

As seen above, Binary Code Similarity Detection (BCSD) techniques have their main applications in fields related to cybersecurity and software analysis. Fields that these techniques have significantly changed and revolutionized. BCSD techniques have, in fact, simplified the tasks of malware analysis, identification of code vulnerabilities and identification of copyright violations, creating the opportunity to perform these tasks by automatically comparing entire blocks of code.
The main revolution that these techniques brought to the world of analysis was the introduction of machine and deep learning technologies, which made cross-platform comparisons possible: that is, comparisons between blocks of binary code compiled for different platforms.

2.4.1 Bug Detection

One of the main challenges that cybersecurity activities are subject to is certainly the search for bugs within blocks of binary code compiled for different architectures (such as x86, ARM, MIPS), in order to identify potential known vulnerabilities.
To support this activity, the approach proposed by Pewny et al., in the paper "Cross-Architecture Bug Search in Binary Executables", comes to the rescue, an approach that is proposed as an innovative system for the search for bugs on cross-compiled binary codes.
The main problem that these approaches have to face are the structural and optimization differences of the compilers that the different platforms, such as x86 and ARM, require. Differences that can appear at the level of instructions, opcodes or even memory accesses.
The approach proposed by Pewny et al. is based on the use of CFGs (Control Flow Graphs), used to represent the logical structure of the code. The use of such graphs allows to work at a higher level of abstraction, overcoming the limits imposed by

the different architectures.

To overcome these limits, CFGs do not collect blocks of code, but the sequence of operations and control flows; in fact, ignoring what the actual instructions are and, therefore, the differences that the architectures impose.

In order to further increase the effectiveness of the comparison, this approach bases the calculation of similarity on hashing and graph matching techniques; thus, simplifying the identification of vulnerability patterns.

2.4.2 Malware Analysis

Another important field of application for BCSD techniques is certainly malware analysis.

In this field, BCSD techniques are mainly applied to identify malware belonging to the same family (for example Agent Tesla), variants of the same malware or the same typology (Trojan, Ransomware, etc...) based on the similarity of their behaviors.

The company Zynamics, with BinDiff[16], offers an effective tool capable of effectively detecting versions of the same malware for the same platform or for different platforms.

Using BCSD techniques based on Control Flow Graphs (CFGs), BinDiff is able to overcome the problems imposed by cross-platform compilation, which leads to significant structural differences in the binary code.

Instead, thanks to BCSD techniques based on signature-based detection, BinDiff is able to detect similarities even when the code is obfuscated.

This is possible because, even if the binary code of the sample, malware families generally exhibit the same behaviors and key features.

BinDiff is therefore able to isolate such elements and identify them within malware of the same family or different versions of the same malware.

The approach, aimed more at semantic and structural similarities of the code, rather than at individual instructions, leads BinDiff to increase the effectiveness of analysis activities, reducing the number of false negatives; providing analysts with an effective, scalable and precise solution.

This tool, due to its characteristics, can also be used in reverse engineering activities.

As regards malware analysis and reverse engineering activities, another valid approach is offered by tools like SAFE (Self-Attentive Function Embedding)[17], which is a valid technique for identifying the similarity of binary codes, using a machine learning model based on self-attention.

Self-attention is a method mainly used by techniques based on natural language processing (NLP), such as BERT or Transformer, for identifying relationships between words in the same sentence. In the context of BCS, this model is used to identify dependencies between instructions of a binary function, so that they, or the basic blocks of a binary function, can be assigned a specific weight, which will then be the subject of the similarity calculation.

Using this model, it is possible to improve the ability to identify the semantic characteristics of the code.

The operation of the model is divided into two phases:

> **Function embeddings**: binary functions are captured by vector embeddings, in which the structural and semantic information of the individual instructions is collected. The embeddings are then learned by the model using a structured neural network, to which self-attention mechanisms are applied, as shown in fig.2.2.
>
> **Similarity between functions**: once the model has finished the phase of learning the vector embeddings, it can start the phase of calculating the binary similarity between the learned vectors, as shown in fig.2.3.

The SAFE model, due to the high level of abstractions on which it places the calculation of the similarity of the binary code, is significantly robust with respect to current code obfuscation techniques and the problem of cross-platform compiling.

Similarly to SAFE, Asm2Vec uses the vector embedding technique, in order to represent the blocks of binary code as numerical vectors. These vectors are containers

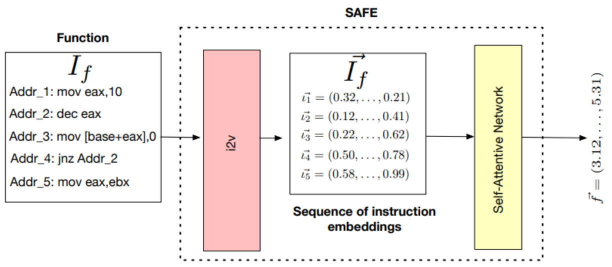

SAFE
Function
I_f
Addr_1: mov eax,10
Addr_2: dec eax
Addr_3: mov [base+eax],0
Addr_4: jnz Addr_2
Addr_5: mov eax,ebx
i2v
$\vec{I_f}$
$\vec{\iota_1} = (0.32, \ldots, 0.21)$
$\vec{\iota_2} = (0.12, \ldots, 0.41)$
$\vec{\iota_3} = (0.22, \ldots, 0.62)$
$\vec{\iota_4} = (0.50, \ldots, 0.78)$
$\vec{\iota_5} = (0.58, \ldots, 0.99)$
Sequence of instruction embeddings
Self-Attentive Network
$\vec{f} = (3.12, \ldots, 5.31)$

Figure 2.2. Architecture of SAFE. The vertex feature extractor component refers to the Unsupervised Feature Learning case

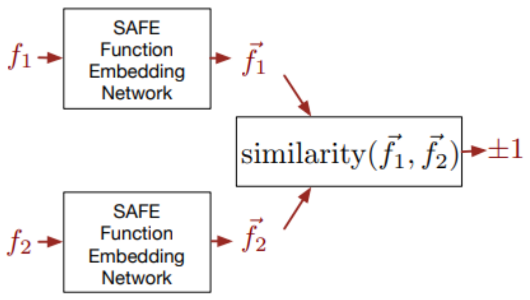

f_1
SAFE Function Embedding Network
$\vec{f_1}$
f_2
SAFE Function Embedding Network
$\vec{f_2}$
similarity$(\vec{f_1}, \vec{f_2})$
± 1

Figure 2.3. Binary Similarity calculation

of the structural and logical relations between the binary instructions.

The difference between the two techniques is due to the approach used. As highlighted in the paper "Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization", differently from the self-attentive approach of SAFE, Asm2Vec uses an approach based on Control Flow Graphs (CFGs) and Graph Neural Networks (GNNs).

The binary code is immediately converted into a CFG that describes the program execution flow through a series of nodes and arcs. The use of CFGs is due to the fact that malware of the same family or typology tend to present similar control flows. Vector embeddings are realized through the use of a Siamese neural network model. The resulting vectors, being the product of the processing of the CFGs, represent the behavior of the binary functions and not the functions themselves.

The vectors, output of this process, will be subject to the comparison calculation useful to determine the similarity of binary codes.

A peculiarity of this approach is the use of an unsupervised learning model; useful in the field of malware analysis, since malicious code is often subject to variations, such as obfuscation techniques, which hardly allow a precise annotation of the data sets. In fig.2.4 the entire work-flow of the Asm2Vec model is shown.

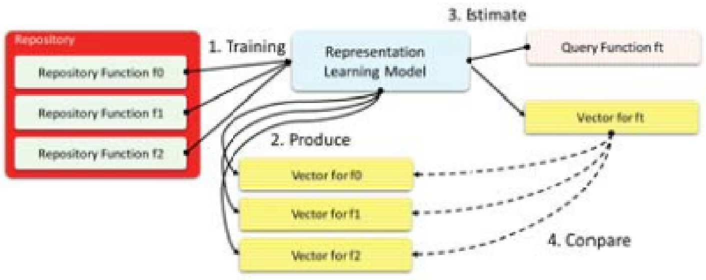


Figure 2.4. Asm2Vec Work-Flow

Chapter 3

State of Art

Binary Code Similarity Detection (BCSD) techniques, being fundamental for activities often related to cybersecurity (such as malware analysis, vulnerability detection or reverse engineering), are involved in a continuous evolution of the strategies on which they are based.
As previously mentioned, the final goal of these techniques is to identify the similarity between binary codes even if these are obfuscated, compiled for different architectures or otherwise.
To address these challenges, a large number of strategies have been developed on which to base BCSD techniques. Strategies can be grouped into three macro-categories:

- Text-based approaches
- Logic-based approaches
- Semantic-based approaches

3.1 Text-Based Approaches

As explained in the paper "Statistical Similarity of Binaries."[18], BCSD techniques that use text-based approaches focus the analysis on binary instructions, which are seen as text sequences.

The main goal of this approach is to be able to identify direct similarities, between different binary codes, or repetitive similarities, within the same binary code, in order to facilitate the search for bugs or malware variants.

BCSD techniques that are based on a text-based approach include **fingerprinting techniques** and **N-Gram methods**.

3.1.1 Fingerprinting Techniques

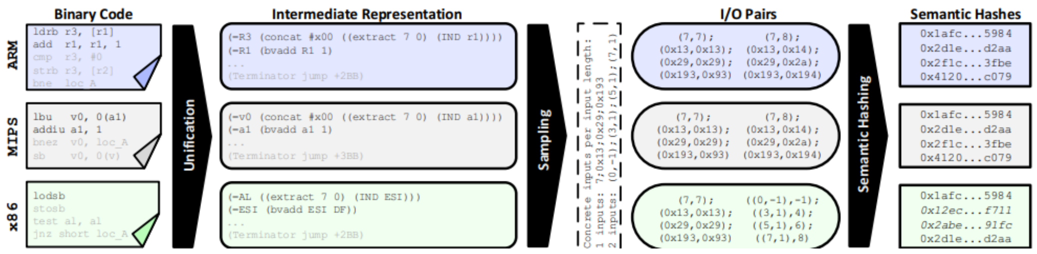
Among the most common text-based BCSD techniques are fingerprinting techniques. As the name suggests, these techniques base their process on the use of a digital fingerprint, or signature, representing a block of binary code.

This fingerprint is then compared with other previously collected fingerprints, in order to identify their similarity.

This makes fingerprinting techniques extremely useful for recognizing malware variants, where parts of the code have been altered, in order to elude more traditional signature-based comparators.

As explained in the paper "Statistical Similarity of Binaries.", fingerprinting techniques are typically based on cryptographic hashing algorithms, useful for providing a more compact representation of binary functions.

The hash, thus generated, is then compared with the other hashes present in the database in order to find a match. A positive outcome of the comparison identifies the similarity of two blocks of binary code. In fig.3.1 the process of transforming a binary function into a hash is shown.



```
ldrb r3, [r1]
add  r1, r1, 1
cmp  r3, #0
strb r3, [r2]
bne  loc_A
```

```
lbu   v0, 0(a1)
addiu a1, 1
bnez  v0, loc_A
sb    v0, 0(v)
```

```
lodsb
stosb
test al, al
jnz short loc_A
```

```
(←R3 (concat #x00 ((extract 7 0) (IND r1))))
(←R1 (bvadd R1 1))
...
(Terminator jump +2BB)
```

```
(←v0 (concat #x00 ((extract 7 0) (IND a1))))
(←a1 (bvadd a1 1))
...
(Terminator jump +3BB)
```

```
(←AL ((extract 7 0) (IND ESI)))
(←ESI (bvadd ESI DF))
...
(Terminator jump +2BB)
```

```
Concrete inputs per input length:
1 inputs: 7;0x13;0x29;0x193
2 inputs: (0,-1);(3,1);(5,1);(7,1)
```

```
      (7,7);          (7,8);
(0x13,0x13);   (0x13,0x14);
(0x29,0x29);   (0x29,0x2a);
(0x193,0x93)  (0x193,0x194)
```

```
      (7,7);          (7,8);
(0x13,0x13);   (0x13,0x14);
(0x29,0x29);   (0x29,0x2a);
(0x193,0x93)  (0x193,0x194)
```

```
      (7,7);    ((0,-1),-1);
(0x13,0x13);     ((3,1),4);
(0x29,0x29);     ((5,1),6);
(0x193,0x93)      ((7,1),8)
```

```
0x1afc...5984
0x2d1e...d2aa
0x2f1c...3fbe
0x4120...c079
```

```
0x1afc...5984
0x2d1e...d2aa
0x2f1c...3fbe
0x4120...c079
```

```
0x1afc...5984
0x12ec...f711
0x2abe...91fc
0x2d1e...d2aa
```

Figure 3.1. Translation form binary function to hash signature process

The paper also shows that, although particularly fast and effective, these techniques

have significant limitations. In fact, they are ineffective when binary codes are subjected to even the smallest changes, such as instruction reordering or the use of a different compiler optimization.
Changes that, although trivial, can completely alter the generated signature.
One of the first tools to use fingerprinting techniques was BinDiff, as seen previously.
Another application of fingerprinting-based techniques combines fingerprinting techniques with N-Grams methods to compare blocks of binary code, in order to identify known bugs.
The use of N-Grams is effective in cross-architecture analyses; thus allowing the identification of sequences of repeated binary instructions present within binary codes, even if these have been compiled for different architectures, such as x86 and ARM.[11]
In order to increase the effectiveness of fingerprinting-based techniques, reduce false positives and make them more robust to obfuscation techniques and the use of multiple compilation optimizers, they are often combined with other methodologies.
Among the most common, it is possible to identify graph matching and, more recently, techniques based on machine learning technologies.[19]

3.1.2 N-Gram Methods

Another technique widely used in the text-based approach for BCSD is the N-Gram.
An N-Gram can be defined as a sequence of N binary instructions extracted from a binary code.
At the basis of the N-Gram methodology, there are techniques based on NLP (Natural Language Processes); which, in fact, divide a text into sequences of N words, or N characters, in order to identify repeated patterns, similarly to what was seen for the techniques based on N-Gram. Fig.3.2 shows how an N-Gram is ideally structured.
As expressed in the paper "Opcode sequences as representation of executables for data-mining-based unknown malware detection."[20], the techniques based on N-methodology Gram are commonly used in identifying malware variants. This is

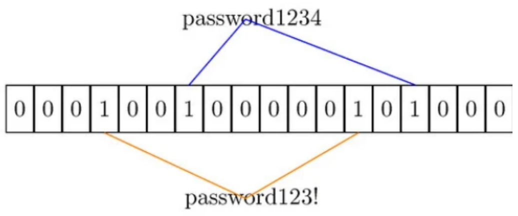


Figure 3.2. How an N-Gram is ideally represented

because, unlike fingerprinting-based techniques, N-Gram-based techniques overcome the recognition difficulties due to code reordering by analyzing not the sequence of binary instructions, but the binary code patterns.

Analysts can therefore generate an N-Gram from a binary code sample and compare it with the N-Grams previously collected and loaded into the database.

N-Gram-based techniques can also be used in cross-architecture contexts, since binary codes deriving from the same source code, although differing in structure and number of instructions between one architecture and another, have the same functional behavior. This aspect makes the N-Gram-based approach useful in analyzing the functional similarities of binary codes compiled with different methodologies.[11]

A significant problem with this approach is certainly the size of N, or the size of the N-Gram. This is because, by imposing a too small size of N, there is a risk of extrapolating only the most superficial details, causing a high number of false positives due to the presence of information that is too generic.

Similarly, by imposing a value of N that is too large, information with such a deep level of detail would be extrapolated that no similarity between variants of the same code would be found.[18]

A solution to this problem is represented by the integration of the methodology based on N-Grams with Machine Learning technologies.

This integration brings a significant optimization of the process both from the point of view of pattern extraction and from the point of view of comparison between the extracted patterns and those present in the database, significantly increasing the effectiveness, efficiency and accuracy of the entire process.

In addition to this, through an adequately trained model, it is able to define a value of N that is as balanced as possible for each context of analysis.[21].

The use of N-Gram-based techniques is scalable and fast in calculating similarity. This is because, once the N-Grams are generated, the comparison is based on the sequence match, rather than on more complex semantic comparisons. These advantages make the use of N-Grams functional in the case of comparative analysis on datasets of significant size.

The problem with this approach, as it was for fingerprinting, is due to the various possible compiler optimizations and code obfuscation; which lead to changes in the binary code, such as the insertion of unused functions, causing an increase in false negatives.

As seen, solutions to these limits are certainly the integrations with other analysis techniques, such as the match based on CFGs or on machine learning technologies.[22]

3.1.3 Advantages and Limitations

Text-based BCSD techniques, as previously described, have a number of advantages, such as simplicity of implementation and computational efficiency, but they also have several limitations such as sensitivity to compiler optimizations or the impossibility of analyzing obfuscated binary codes.

The advantages presented by this type of approach are the following:

Simplicity and speed of computation: text-based BCSD techniques are generally simpler to implement and require, in computation, a limited amount of resources.

Support for malware analysis: techniques based on N-Grams or on opcodes matching allow the rapid identification of recurring patterns within sections of binary code, simplifying the identification of malware variants.

Scalability for large datasets: Since they do not require a detailed understanding of the logic of the binary code, text-based techniques are ideal for analyzing datasets containing a large number of binary codes. In fact, they are often used for an initial comparison, prior to the use of more complex techniques.

The limitations of the text-based approach are analyzed below:

Sensitivity to compiler optimizations: Among the main limitations to which these techniques are subject, there is the unreliability with respect to compiler optimizations; that is, it is meant that the slightest modification to the binary code, such as the insertion of NOP (No OPeration) functions or the reformulation of the code, is enough to increase the number of false negatives in the comparisons.

Inability to capture program logic: Being an approach with a focus on syntax, rather than semantics, it focuses exclusively on instructions and not on program behavior. This leads to an inevitable inability to notice the similarity of codes that maintain the same behavior, but present different instructions.

Code obfuscation: As with compiler optimizations, code obfuscation techniques create alterations to the binary code, which increase the number of false negatives in comparisons of techniques based on this type of approach.

Cross-architecture comparison: compiling the same source code for different architectures, such as ARM or MIPS, leads to obtaining binary codes with structural differences, such as opcodes and instruction sets. These differences impact the number of false negatives in comparisons, as the text-based approach searches for the exact match of the opcodes.

3.2 Logic-Based Approaches

Unlike text-based approaches, which analyze binary code as a sequence of binary instructions, logic-based BCSD techniques analyze the execution flows of binary code.

Logic-based approaches base their analysis on the logic or data flows of a binary code, with the aim of representing the logic of the code. This focus allows for a more robust analysis of binary code than text-based approaches, overcoming the limitations of these due to code modifications.
Among the most common techniques that use this particular approach, there are techniques based on **Control Flow Graphs (CFGs)** or **Data Flow Graphs (DFGs)**.
CFGs analyze and extrapolate the execution flow of a program; collecting, in the nodes, the blocks of binary code that are connected to each other by a series of arcs that represent the logical structure of the function calls.
Differently, DFGs analyze the flow of data exchanged between blocks of binary code. The analysis, in this case, focuses more on how the variables are used and processed.
Techniques using logic-based approaches are often used in the field of malware analysis and vulnerability research.
In fact, these techniques are useful for analyzing the logical similarities in the internal structure of a binary code.
The wide use of these techniques in these fields is due to the fact that binary codes having the same control flow or data flow, and therefore the same logical structure, can present totally different functions and instructions. Examples of such conditions are cross-architecture contexts; where the structure of the binary code varies based on the architecture for which it was compiled, but its logic remains unchanged.

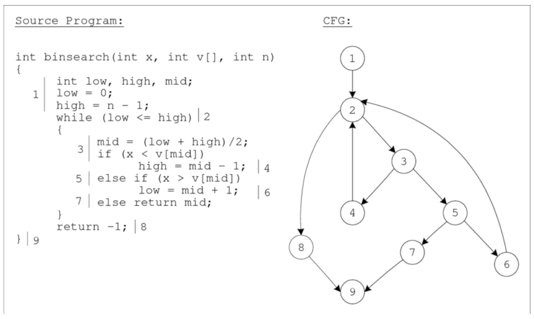
3.2.1 Control Flow Graph (CFG)

Techniques based on Control Flow Graphs (CFGs), are techniques focused on the representation of the control flow of a binary code through graphs.
Each graph is made up of two fundamental elements:

nodes: graphical representation of a block of binary code or basic block, or a sequence of instructions executed without conditional jumps.

arches: graphical representation of the control flow of the blocks of binary code. Each arch represents how the program passes from one block of binary code to another, from one node to another, based on the conditions specified in the starting block.

The fig.3.3 shows a representation of the graphs.



```
Source Program:

int binsearch(int x, int v[], int n)
{
    1 | int low, high, mid;
      | low = 0;
      | high = n - 1;
        while (low <= high) | 2
        {
            3 | mid = (low + high)/2;
              | if (x < v[mid])
                        high = mid - 1; | 4
            5 | else if (x > v[mid])
                        low = mid + 1;  | 6
            7 | else return mid;
        }
        return -1; | 8
} | 9
```

Figure 3.3. Example of Control Flow Graph.[23]

One of the main applications of techniques using CFGs is certainly the analysis of binary code for vulnerability research. Indeed, tools like Gemini[24] uses the CFG-based approach for this very purpose.

In this tool, the CFGs produced are used for the analysis of unusual control flows or potential exploit vectors, such as poorly managed unconditional jumps, which can cause Buffer Overflow.

In order to increase the effectiveness of CFGs on cross-architecture analyses, Gemini combines this approach with GNN (Graph Neural Network)-based techniques.

With the BinDiff tool, instead, we see the use of CFGs in the context of malware analysis. In fact, this approach allows not only to identify variants of known malware, but also to compare known malware with unknown malware and verify whether the sample under analysis is or is not a variant of a known malware.

The main advantage of CFGs is certainly their focus on the behavior of the binary code, rather than on the individual binary instructions. This focus allows, in fact, to identify the similarity of binary codes even in the presence of different compiler optimizations or code modifications and reorganizations, making CFGs robust against binary code obfuscation techniques.

The disadvantage is the computational complexity of processing comparisons between CFGs. This leads to a deficit regarding scalability both for the comparison between large graphs and for the comparison between a high number of collected graphs.

3.2.2 Data Flow Graphs (DFGs)

As seen previously, unlike CFGs, DFGs focus their analysis no longer on the execution flow, but on the flow of data that the blocks of binary code exchange.
As with CFGs, DFGs are also composed of two fundamental elements:

nodes: representing the data processing operations

arches: representing the flow of data between operations

The figure.3.4 shows some examples of DFGs representations.

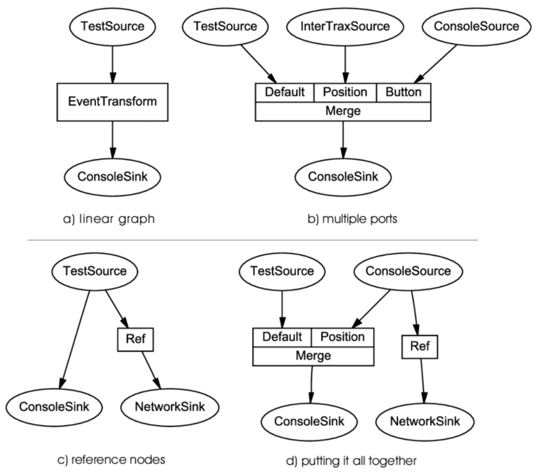


Figure 3.4. Visualizations of a data flow graphs as used in OpenTracker. (a) A linear flow. (b) A node with different input ports. (c) Fan-Out of output ports. (d) A complex example employing all features.[25]

We can therefore say, basically, that DFGs extrapolate how data moves and is processed within a binary code.
The work proposed by Zhou Y. et al.: "TaintFlow: A Practical Data Flow Tracking System for Detecting Taint-style Vulnerabilities"[26], shows how techniques with a DFGs-based approach can prove to be effective even with respect to the different

compiler optimizations. This is because, by analyzing the flow of data, and not the instructions that generate them, these techniques can easily identify which operations are redundant, but also which are NOPs, and therefore evasive functions.
Obviously, such efficiency finds wide use in the world of cybersecurity, where DFGs are mainly used to identify vulnerabilities related to memory management, such as buffer overflows, or related to those areas of binary code where data flow management is crucial.
With Vulseeker[7], the DFGs-based approach is used in order to identify cross-architecture vulnerabilities. In fact, this approach, as seen with TaintFlow, overcomes the limitations due to differences in instructions, whether caused by compilation on different architectures or by binary code obfuscation techniques, focusing the analysis on the data flow.
The main advantage of using DFGs is, certainly, the possibility of extrapolating the semantic behavior of the program under analysis, rather than the control flow; making this approach significantly useful in identifying vulnerabilities related to data management and code optimization. Furthermore, it is particularly useful for identifying correlations between data that could represent a vulnerability.
The problem with this approach is that it requires support to identify vulnerabilities in the code, such as poorly designed unconditional jumps. Furthermore, by performing a detailed analysis of the operations on the data, it presents a significant computational burden.

3.2.3 Advantages and Limitations

Among the advantages presented by logic-based techniques, the main ones that stand out are their robustness with respect to code obfuscation techniques, the ability to extrapolate the program logic and the applicability with respect to cross-architecture.
As for obfuscation techniques, as is known, these are used with the aim of creating alterations of the instructions within the binary code to elude detection systems.
Logic-based techniques, since they do not base their analysis directly on the instruc-

tions, but on the logical or data flow, are able to overcome the obstacles created by the obfuscation of the binary code and to be efficient even in the comparison of cross-architecture binary codes.

Another advantage is certainly the possibility of directly extrapolating the program execution logic. This ensures that, even if the binary codes are structured in a completely different way, if the behavior shown presents points in common, these will be identified. With this ability, it is possible to overcome the differences in abstraction, architecture and everything related to the code.

As text-based approaches were sensitive to changes, even slight ones, to the code structure, so logic-based approaches are sensitive to even minimal variations in the program logic. In fact, a small variation in the control flows, such as the introduction of conditions or cycles, or the variation in data flows, such as the processing of additional data, can cause a significant increase in false negatives.

Another limit is certainly the computational complexity that these approaches present. Complexity, from the temporal and memory point of view, due to the difficult management

3.3 Semantic-Based Approaches

With the advancement of technologies useful for the use of machine learning and deep learning techniques, the techniques related to BCSD (Binary Code Similarity Detection) have undergone a real revolution.

Through the use of approaches based on the use of such learning models of semantic representations of binary code, BCSD techniques have significantly increased the precision and effectiveness in identifying similarities in patterns hidden from more traditional approaches and reliability in the most disparate contexts.

The main objective of semantic-based approaches is certainly the extrapolation of functional relationships between binary instructions, rather than the instructions themselves or the logic of the program that contains them.

One of the first BCSD techniques based on this approach is the **embedding** technique which, as expressed in the paper presenting the tool Asm2Vec[9], converts the binary code into a series of numerical vectors; which can be compared in the vector space.

From the notions of natural language processing (NLP - Natural Language Process) derive the techniques based on **Siamese models** and **Machine Translation**. In both cases, they are approaches that, like text-based approaches, tend to see the binary code as a text, but use neural networks in order to identify the similarity between binary functions, overcoming obfuscation and cross-architecture techniques.[27]

Another relevant approach involves the use of **GNNs (Graph Neural Networks)**. Such an approach exploits CFGs techniques to obtain a structural representation of the code, integrating neural network techniques for more complex semantic relations. GNNs techniques, as demonstrated in the paper "SAFE: Self-Attentive Function Embeddings for Binary Similarity"[17], are particularly effective in comparing binary functions subjected to binary code obfuscation techniques or different compiler optimizations.

3.3.1 Embedding and Semantic Code Representations

Embedding techniques were actually the first step towards the revolution of Binary Code Similarity Detection (BCSD) techniques. Revolution that led to the development of techniques based on the semantic representation of binary code.

The approach on which embedding techniques are based mainly consists in the processing of binary functions in vector spaces. Such processing allows comparison not only between binary functions, or blocks of binary code, but also between semantic aspects of the binary code.

Such an approach allows to address and overcome the limitations imposed by binary code obfuscation techniques and by the structural differences of the same binary code, due to compilation for different architectures, by abstracting the binary code with a focus on the functional behavior of the individual binary instructions.

One of the most relevant early examples, which uses the embedding-based approach, is certainly Asm2Vec[9].

It was one of the main tools that contributed to the introduction of vector comparison. Asm2Vec extrapolates structural and semantic characteristics of binary functions, in order to generate a series of static embeddings; which will then be the object of comparisons.

As it is easy to imagine, this tool has found immediate use in the field of malware analysis with the aim of identifying variants of the same malware obfuscated or compiled for different architectures.

A more recent example of tools using an embedding-based approach is, certainly, the SAFE (Self-Attentive Function Embedding) model[28]. This tool uses a self-attention architecture to be able to learn the semantic representations of binary functions. In this way, being able to capture the dependencies between the individual functions and to assign them a specific weight, which is the subject of subsequent comparisons, it proves to be a significantly more precise tool than its predecessors.

As it has been possible to see, semantic-based binary code representations have a focus directed towards how the functions, within a binary code, interact with the context in which they are executed, rather than the mere comparison of the syntax of the binary instructions present in the code.

Such an approach allows to perform a particularly detailed analysis of the binary code, focused on functional blocks. It turns out to be particularly effective in patch and malware analysis; fields in which compilation differences and compiler optimizations make even graph-based techniques ineffective.

An example of the use of semantic representation is the BinSequence tool.

The primary purpose of this tool is binary code reuse detection. This goal is achieved by using fuzzy matching techniques, useful for comparing binary functions, with a focus on basic blocks, represented by the use of CFGs (Control Flow Graphs).

BinSequence has proven to be perfectly capable of analyzing binary code from large repositories.[29] The fig.3.5 shows the BinSequence work-flow.

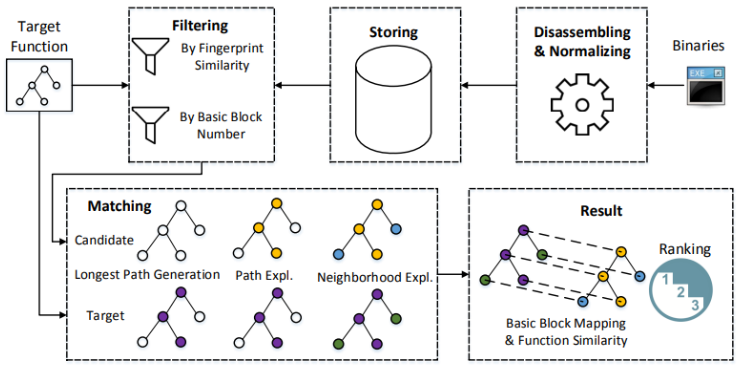


Figure 3.5. BinSequence work-flow

As seen, the main advantages provided by embedding techniques are the ability to extrapolate and represent semantic relations between binary functions and robustness in relation to obfuscation techniques and compiler optimizations.

Advantages that, as seen for Asm2Vec and SAFE, allow a significant increase in the precision of comparisons compared to text-based or logic-based techniques; make embedding-based techniques ideal for fields such as malware analysis, reverse engineering and vulnerability research.

However, these techniques are limited by the computational complexity required to process large amounts of data, useful for training models.

Another limitation is the quality of semantic representations, which depends on the quality of the data used in the training phase; a condition that can lead to having data that is not well labeled or in any case limited.

3.3.2 Siamese Models

Among the most effective neural network architectures used in the comparison of binary code similarity we certainly find Siamese Neural Networks (SNNs).

Initially used for facial or biometric recognition, today they are widely used in the field of BCSD.

As with all techniques based on machine learning or deep learning, SNN-based techniques process inputs, such as binary codes or binary functions, mapping them into vectors that will then be compared at a semantic level.

The architecture of SNNs involves the use of two identical neural networks, precisely Siamese, which, sharing the same information, are able to process, in parallel, different inputs.
Each network calculates a block of binary code, in the form of CFG or sequences of instructions, and generates a vector representation of it.
The most commonly used metrics for comparison purposes are distance metrics, such as Euclidean distance or cosine distance.
The three phases of the SNN workflow are detailed below:

Input: is the binary code, represented as a graph (CFG or DFG) or as a sequence of opcodes. The structural and functional characteristics are extracted from this, mapping an abstract representation of the functioning of the individual blocks of the binary code.

Feature extraction: in order to obtain a mapping, in a vector space, of the inputs, each sub-network extracts the relevant characteristics of the code, reducing its complexity. Since the networks share the same weights, the vector representation obtained is coherent. This simplifies the comparison between binary functions even if compiled with structural differences.

Comparison: the vector representations obtained are subject to comparisons based on distance. If the distance is less than a given threshold, the two compared binary codes are considered similar to each other.[30]

The figure.3.6 represents the process of processing the similarity of two inputs for techniques with an SNN-based approach.

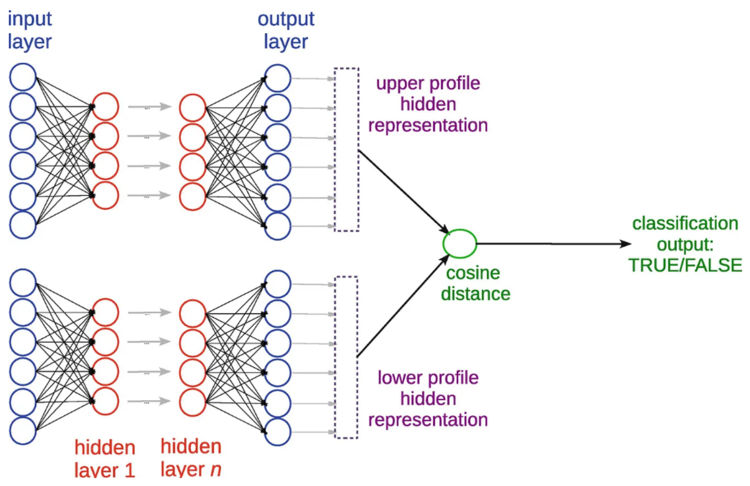

input layer
output layer
upper profile hidden representation
cosine distance
classification output: TRUE/FALSE
lower profile hidden representation
hidden layer 1
hidden layer *n*

Figure 3.6. Siamese Neural Network workflow

In the macro-field of BCSD, SNN-based techniques are applied on different fields of study.

One of the main applications of SNNs consists in the identification of the similarity of binary codes, in the presence of cross-platform compilation.[5]

The ability of these techniques to operate in cross-platform contexts is particularly useful in IoT analysis, where codes are compiled for hardware architectures that are totally different from each other; thus allowing a rapid and effective identification of malware variants.[31]

Techniques based on this approach are particularly efficient in detecting similarity, even in the face of code alteration.

Furthermore, unlike other semantic-based approaches, SNNs require little data for the model learning phases, as the focus is on the relationships between entities rather than on the single classification.

Even if the overall amount of data is lower than more traditional methods, it requires a high number of positive and negative data, a number that can cause computational complexities.

Another limitation is scalability, as calculating the distance between input pairs can become computationally expensive in proportion to the size of the dataset.

3.3.3 Neural Machine Translation (NMT)

The approach of models based on NMT (Neural Machine Translation) has its origins in the automatic translation of natural languages. A use case for this goal is certainly Google Translate[32].

Given the efficiency of these approaches in the recognition phases, they have also found wide use in different fields, including Binary Code Similarity Detection (BCSD) techniques.

The approach of BCSD techniques based on the NMT model is based on the representation of a source code as if it were a language and, consequently, the binary codes, generated for the different architectures, as if they were dialects originating from the same source language.

Such a vision makes it possible to translate such dialects into inputs useful for recognizing the similarity between binary functions that are structurally different from each other, but with similar functional behaviors.

This ability makes techniques based on NMT models particularly effective in cross-platform contexts.

The work of Zhu et al.[33], in addition to providing a overview of the origin and use of NMT-based models in BCSD techniques, provides an architectural description of the model.

In particular, NMT models for BCSD techniques feature an encoder-decoder architecture with an attention mechanism:

Encoder: in this phase, the binary code is transformed into a vectorial, semantic embedding, capable of abstracting the semantics of the code. The criticality of this consists in making the produced embedding independent of the compiler architecture or optimization.

Decoder: generates a new embedding, similarly to the encoder, for another function of the binary code and compares them. If the vectorial distance between the two embeddings falls within a given threshold, they are defined as similar to each other.

Attention mechanism: in order to avoid taking into account irrelevant variations of the code, this particular mechanism is applied. This mechanism makes it possible to extract a coherent block of binary code, eliminating parts of the block that do not add value.[34]

The figure3.7 shows a representation of the workflow of the NMT-based model.

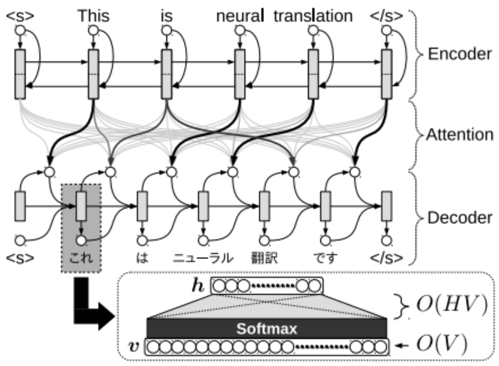


Figure 3.7. NMT model workflow.[35]

One of the main fields of application of this model is certainly vulnerability detection. Among the tools using this model, we can mention Bin2Vec[36], which uses the NMT model to address cross-platform analysis and analysis where the source code is not available.

In general, Bin2Vec creates a mapping of binary functions on multiple vectors, allowing a comparison at a level of abstraction that can ignore platform and compilation differences.

The entire architecture is based on the model NMT, thanks to which it can exploit a translation pipeline, useful, through the encoder, to process the code by translating it into a more abstract version that will be reworked by the decoder to obtain a comparable representation.

3.3.4 Graph Neural Network (GNN)

Graph Neural Networks (GNNs) are techniques widely used in BCSD, alongside techniques based on CFGs. GNNs are typically implemented in order to model CFGs, or other representations of the binary code, in order to extract the structural relations of the code under examination and therefore define its semantic aspects. In this way, they can be effective even in the face of cross-platform contexts and different compiler optimizations.

From an architectural point of view, as highlighted in the work of Yeming Gu et al.[37], GNNs are made up of two fundamental elements:

nodes: representations of binary instructions or functions. They can contain additional information, such as opcodes or conditional branches.

arches: define the dependencies between individual nodes, such as unconditional jumps and function calls.

Since they have to operate on flow graphs, whether they are data or control, GNN models, unlike other models or neural networks, require a convolution process on the graphs. This process allows you to perform aggregation operations on a node with its neighbors, in order to allow learning of their dependencies.

Techniques based on the GNN model are widely used in various sectors of cybersecurity.

One of these is certainly the detection of cross-platform similarities. In fact, as has already been seen for machine learning and deep learning technologies in general, GNN models allow the processing of binary code blocks at a level of abstraction that allows them to ignore the problems related to cross-platform compilation. In particular, GNN models, by processing flow graphs, can map the binary code into embedding representations that extract the semantic structure of the binary code itself. In this way, ensuring greater precision in the comparison outputs.

Another sector in which the GNN model is typically used is malware analysis. In fact, as highlighted in the work of Bingchang Liu et al.[38], the ability of such models to extrapolate the semantic relationship between blocks of binary code and provide a graphical representation of it, allows overcoming the problem due to elusive techniques, such as code obfuscation techniques.

Again, these are used in patch analysis and vulnerability detection. This is because, after the application of a patch to resolve a vulnerability, there may have been only a change in the code, without considering potential vulnerabilities at the functional level. The use of GNN models allows the analysis of the program's behavior before and after the patch application, effectively highlighting whether there are still vulnerable features in the code.[39]

Another advantage of this model is its scalability, as it can be trained on datasets of binary functions of significant size; also ensuring a more precise and reliable comparison.

The limits that this approach presents consist mainly in the computational complexity, as it is necessary to consider the cost of processing particularly complex graphs, or processed by blocks of binary code of significant size.

Another aspect to consider is the difficulty of processing long-range relations, that is, those present in two distant points in the binary code.

3.3.5 BERT Applications for Binary Similarity

The approach based on BERT (Bidirectional Encoder Representations from Transformers) was born as an integration of NLP techniques.

The effectiveness demonstrated in textual analysis has led, over time, to the use of techniques based on the BERT approach also in other environments, including Binary Code Similarity Detection (BCSD).

Given the possibility of approaching a bidirectional understanding of the text, it is able to perform cross-platform analysis or with different compiler optimizations. This ability allows a wide use of BCSD techniques based on BERT in the fields of malware analysis and vulnerability research.

First, the concept of bidirectional understanding of the text was introduced. This

term means that the BERT model is not limited to understanding a single word, or instruction in the case of BCSD techniques, but analyzes what comes before and after it in order to have a greater understanding of the context in which that word is found.
For this reason, this model is widely used in the web world, especially by Google, which has created its own version: GoogleBert, as seen in the work of Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.", a particularly useful tool not only for research, but also for market strategies, such as the SEO web positioning strategy.
The ability to understand the text in two directions, possessed by the BERT model, is due to the fact that the entire structure of the model is based on a Transformer-type architecture. In the application to BCSD techniques, the BERT model is able to define a mapping of binary instructions in a vector space, in which the semantic characteristics of the binary code are extrapolated.
Similarly to the architectures of the LSTM (Long Short Term Memory)[40] and NMT models, the BERT-based models present a self-attention mechanism that is applied, unlike the above-mentioned models, to increase the ability to understand the context in which an instruction is placed, eliminating dependencies on the order of sequence of the instructions.
The BERT model development process is based on three fundamental aspects:

Pre-training on binary code: after an initial training phase, on a significant amount of textual data, the BERT model is optimized for operation on binary code. This optimization occurs through the MLM (Masked Language Modeling) process of the binary code, which allows the model to learn the semantics of the code, independently of the hardware architecture. The MLM process consists of hiding some binary instructions in order to make the model capable of predicting them based on the understanding of the general context.

Fine-tuning for BCSD: once the pre-training phase has been completed, the model must be optimized to increase the effectiveness of the binary similarity calculation and the search for vulnerabilities. To this end, the fine-tuning phase allows the BERT model to better adapt to the binary code, significantly improving the accuracy in the comparisons of binary functions on differential architectures.

Self-attention mechanism: the focal point of the BERT model is the self-attention mechanism, with which the model is able to analyze the preceding and following text of a binary instruction.

One of the limits of the BERT model is certainly the computational complexity, due to the training phases that require a huge amount of data. Such a need can cause problems, from a computational point of view, in the case of large-scale comparisons. The discrepant training introduces a second limit. This is represented by the quality of the data and the close dependence of the model on it. The use of the BERT model in environments such as malware analysis and vulnerability research, both in cross-platform contexts, means that it is necessary for the training data to cover a wide range of structural aspects of each hardware architecture. In this way, the presence of false negatives in the comparisons is minimized.[41]

3.3.6 Advantages and Limitations

As seen, the use of Machine Learning and Deep Learning techniques has represented a turning point in the world of BCSD, revolutionizing the methodologies applied for the comparison of binary code and overcoming the limitations of techniques based on text and syntax of binary code, such as hash signatures and CFGs.
Below we want to highlight the advantages brought by these techniques:

Generalization ability: one of the main advantages brought by techniques based on semantics is certainly their ability to generalize the binary code, abstracting it and extracting its functional, or semantic, characteristics in order to significantly increase the reliability of comparisons on different architectures or with different code optimizations.

Robustness against obfuscation: being able to define similarities on the functional characteristics of the code, and not on the structural ones, semantic-based techniques, and in particular those based on Deep Learning, have proven to be able to overcome the limits imposed by binary code obfuscation techniques.

Automation of the feature extraction process: with the automation offered by semantic-based techniques, it has been possible to automate the process of extracting features relevant for comparisons on binary code, reducing the burden of manual work, present in more traditional comparison techniques.

Adaptability to specific tasks: through the fine-tuning methodology, it is possible to train the model on which semantic-based BCSD techniques are based to perform specific operations.

Obviously, these techniques also have limitations, due to the presence of a model that must necessarily be trained:

Computational complexity: models based on Machine Learning or Deep Learning technologies often require various training phases on a large amount of data. This causes a significant demand for computational resources, which affects the performance of the tool in the comparison phases of large sets of binary codes.

Needs for large datasets: another aspect related to datasets is the need to label the data, a task that can be particularly onerous, but also that of often not being able to find datasets that are varied enough to have a model trained enough to provide precise and effective comparison results.

Interpretability: semantic-based models present a complication in understanding which functional characteristics are influencing the work of the model itself.

Data Over-fitting: as previously mentioned, one of the problems related to datasets is having datasets with data as varied as possible. This is because having a model trained on data that is not diverse enough can cause an inability of the model to generalize to new data, preventing it from analyzing binary functions that differ from the training data.

Chapter 4

Challenges

BCSD techniques have gradually become increasingly indispensable in the field of cybersecurity. The use of these techniques in this field is the result of the possibility that they offer in being able to analyze and define the similarity of binary codes, even in the absence of a source code.
A capability that proves to be fundamental for a series of activities related to the world of cybersecurity such as malware analysis, vulnerability research and identification of copyright violations on the code.
Although these techniques are constantly improving, they face a series of challenges that undermine their precision, accuracy and reliability.
In this chapter we want to define the four main challenges that BCSD techniques must face and overcome.
Challenges that mainly concern the voluntary and non-voluntary variations that a binary code almost always tends to undergo, and that alter the structural and functional characteristics of the binary code itself.
These challenges can be defined as:

> **Compiler optimization**: each compiler applies different compilation optimizations to the source code, aimed at improving the performance of the resulting binary code. The problem is that such optimizations tend to make structural changes to the code, making it difficult to compare binary functions.

Binary Code Obfuscation: This challenge belongs to the voluntary alterations. These are alterations to the binary code, aimed at hiding the real logic of the program. This technique is typically performed by adding functions that have never been called, or by renaming existing ones with random strings.

Cross-Architecture: A source code must often be usable on different hardware architectures, such as MIPS, ARM and x86. The problem is that each architecture differs from the others in several aspects, such as the structure of the opcodes or the logic of memory accesses. This gives rise to binary codes, variants of the same source code, which are difficult to compare with each other.

Scalability and computational requirements: BCSD techniques often have to manage a significant quantity, or complexity, of binary codes to compare. This results in a significant time and computational burden, which BCSD techniques, in their evolution, must necessarily contain.

In the following sections, the individual challenges will be explored in greater depth.

4.1 Different Compiler Optimizations

The phase preceding the compilation of a source code consists in the choice of the compiler itself. This choice varies based on the optimizations that the compiler offers in relation to the context in which it is located.
The purpose of the choice phase is to obtain a binary code that is as efficient as possible for the context in which it will be executed.
The problem is that each optimizer often involves structural alterations to the binary code. Alterations that can also modify control and data flows, thus making the process of detecting the similarity of the binary code, perpetrated by BCSD techniques, significantly more complex.
To date, there are different types of compiler optimization. Among these, the most common are:

Speed optimization: in order to reduce computational complexity and, therefore, make binary code execution faster, the compiler eliminates redundant functions and replaces complex functions with computationally lighter alternatives.

Space optimization: in this case, the purpose of optimization is to minimize memory usage. To this end, this type of optimization compacts data as much as possible, eliminates variables and functions that are useless for execution, that is, those elements declared but never called in the code.

Function optimization: this type of optimization concerns the management of functions that are called only once within the entire code. In order to avoid the unnecessary occupation of memory space, due to the call of a separate function, the body of that function is directly integrated into the body of the caller.

Dead-code optimization: Similar to space optimization techniques, this type of optimization consists in eliminating all those functions that do not add meaning to the binary code. This approach is based on the elimination of code blocks indicating functions that have never been called or are not useful for the purposes of the code.[42]

Compiler optimization techniques, whatever the approach used, cause substantial structural alterations to the binary code. This represents a real challenge for BCSD techniques in having to compare binary codes, variants of the same source code, compiled with different optimizers. The pinnacle example of such alterations are certainly the approaches based on dead code optimization or space optimization, where code blocks are completely eliminated.
To face this challenge, text-based BCSD techniques, limited by this problem, have been replaced by logic-based techniques, namely CFGs or DFGs, and semantic-based techniques, based on Machine Learning or Deep Learning. Techniques that allow comparisons to be made independent of the syntactic structure of the code, and

therefore of the sequence of instructions, as they are based on logical characteristics, control or data flows, or semantic, or functional.
As highlighted by the work of Chen et al.[43], the most commonly used techniques to face this type of problem are based on CFGs, representations of the control flow of a program.
This is because, despite having syntactic differences, the execution flow of a program remains unchanged; therefore, with graphical representations of the execution progress it is possible to compare the similarity of binary codes.
As already seen with the Gemini tool, techniques based on the use of CFGs are often combined with machine learning techniques, such as GNNs, in order to significantly increase their performance and the accuracy of the analyses.

4.2 Code Obfuscation Techniques

Code obfuscation techniques are typically applied to prevent proper code analysis by creating alterations to the binary code. Such alterations cause significant problems for reverse engineering, decompilation, vulnerability research and, to date, binary code similarity detection.
In particular, the challenge that BCSD techniques face with code obfuscation techniques is to be able to recognize the similarities between an obfuscated binary code and an unaltered one.
First, from the evidence of the work of Ming et al.[44], let's define the different binary code obfuscation techniques:

Syntactic obfuscation: techniques that consist of altering the names of variables, functions and classes with random strings, making the code unreadable both manually and with automatic tools.

Control flow obfuscation: these are the most advanced binary code obfuscation techniques, specifically designed to elude code analysis using CFGs. Such techniques introduce noise into the code, such as false if-else statements, or split blocks of code in such a way as to alter not only the code structurally, but also the control flow logic.

Data obfuscation: techniques that elude DFGs-based methods, which tend to alter the data management in the binary code. In particular, they alter how data is represented within the binary code or how it is loaded into memory. This type of alteration is typically performed with more or less complex cryptographic techniques.

Although each technique operates on a specific aspect of the binary code, the goal is always the same: it alters part of the code in order to prevent its analysis, without changing its behavior.

The use of obfuscation techniques inevitably causes a significant decrease in the effectiveness and reliability of BCSD techniques, especially those based on text and logic.

This decrease is due to the fact that, as previously mentioned, binary code obfuscation techniques significantly alter the structure and the control and data flows of the binary code.

To address this challenge, techniques based on semantic approaches have been developed, or in any case on the embedding of the code in vector spaces.

Unlike more traditional approaches, these approaches, as seen for the Asm2Vec tool, tend to represent the binary code as a set of multi-dimensional vectors that contain parts of the behavior, semantics, and code itself. In this way, obfuscation techniques are rendered completely inefficient because they no longer look at the sequence of instructions, but at the behavioral aspects that the code shows during execution.

Another more classic approach is the use of GNNs that takes up the concept of graphical representation, but this time the nodes represent semantic characteristics of the code.

More recently, approaches based on the concept of self-attention applied to Transformer-based models have been developed. An example of this evolution is the CodeBERT[45] tool. This type of approach allows an analysis not only of the binary instruction, but also of the context in which it is placed. Therefore, the part preceding and following the analyzed instruction is also observed. In this way, it is possible to have a general view of the entire block of binary code.

4.3 Cross-Architecture Analysis

When a source code is compiled to run on different architectures, such as MIPS or ARM, the resulting binary codes show profound structural changes. These changes can affect the level of binary instructions, opcodes or memory accesses.
These differences tend to complicate BCSD operations, as binary variants of the same source code are completely different from a structural point of view of the binary code.
This challenge requires the use of advanced comparison techniques, in order to identify similarities between blocks of binary code.
As highlighted by the work of Ming et al.[46], possible solutions to this problem have been identified in graphical techniques: CFGs, DFGs, GNNs. In fact, these are approaches that represent binary functions as graphs representing control flows or data or semantics.
In this way, the similarity analysis is made totally independent from the syntactic structure of the binary code.
The introduction of the BinSim tool also highlights the fact that flow graphs alone may not be enough, as it is uncertain that the alterations only concern the syntax and do not also affect the logic.
It was consequently necessary to use much more complex and structured techniques. We are talking about GNN-based techniques, which, by abstracting the binary code to a set of functional characteristics, are able to focus on the recognition of behavioral patterns.

4.4 Scalability and Computational Requirements

Another challenge that BCSD techniques, especially those based on Machine Learning and Deep Learning technologies, have to face is the problem of scalability and computational complexity. This problem arises with the constant increase in the size of the binary code datasets that BCSD techniques must process in order to perform the necessary similarity comparisons. The challenge that arises is to manage datasets of significant size without affecting the reliability and precision of the comparisons. Particularly involved in this problem are certainly the techniques based on GNNs and Siamese networks. In fact, these techniques have a significant computational weight due to the need to train the models on a large amount of data or on particularly complex CFGs and DFGs. The work of Peng et al. "Fast Cross-Platform Binary Code Similarity Detection", already discussed above, proposes the use of a detection framework that integrates the use of NLP processes in the CFGs-based approach. In this way, an approach similar to that of the BERT model is maintained, but with a significantly lower computational burden.

As for Siamese models, these require a significant amount of resources in order to generate accurate embeddings of binary function pairs on a large scale. As suggested by Zuo et al. in the paper "Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs", a solution consists in using an approach based on pre-training the model, in order to reduce training times and, consequently, lighten the computational burden.

As for scalability and computational requirements for cross-platform binary code analysis, the work of Cochard et al.[47] proposes the use of neural networks based on CFGs representations, in order to improve scalability and efficiency in this type of analysis.

A general solution, regarding the scalability of the analysis techniques, is proposed with the introduction of the BinSequence tool. In the work of Huang et al., in fact, it is highlighted how a system that combines clustering techniques and pruning of the number of comparisons needed, in order to identify the similarity between binary codes, can significantly reduce the necessary analysis times and, consequently, the necessary computational resources.

Chapter 5

Future Perspectives

Binary Code Similarity Detection (BCSD) techniques are techniques characterized by constant and rapid evolution. Evolution that, today, is increasingly entering the world of Machine Learning and Deep Learning; continuing to improve in step with the improvement of these technologies and the increase in the availability of computational resources in systems.
A constant and rapid evolution is essential to be able to face the problems due to cross-architecture, code obfuscation techniques and cross-compiler optimization. Problems that continue to progress, becoming increasingly complex and articulated, managing to compile variants of binary codes that are increasingly different from each other.
To be able to cope with these problems, the literature, field of BCSD techniques, is exploring several new possible directions. Among these we certainly find: the integration between static and dynamic analysis, the development of increasingly efficient machine learning models, benchmarking and the use of large-scale datasets, the increasingly prevalent use of transfer learning and fine-tuning and the increase in robustness against code obfuscation techniques.
In this chapter we want to describe in detail these directions, which seem to be among the main directions currently being explored.

5.1 Static and Dynamic Analysis Integration

One of the main strategies that is being explored, to increase the efficiency and effectiveness of BCSD techniques, is certainly the integration between static and dynamic analysis. These types of analysis are already widely used in a large number of fields, but always with a mutually exclusive approach: either one or the other, with all the advantages and disadvantages that the individual types bring with them.
Static analysis consists in the analysis of a binary code without executing it. It is an analysis based on the structural characteristics of the code, such as control flows and opcodes. Features that allow the analyst to observe and study the structure and logic of the individual functions that make up the binary code.
Static analysis is often used in the field of malware analysis, because, thanks to disassemblers such as IDA or Ghidra, it allows the analysis of malicious binary codes that cannot be executed can be executed, or in any case without having to worry about any problems caused by them.
The limit of this type of analysis is the inability to understand the behavioral aspects of the binary code being analyzed, how it manages dynamic resources or the side effects that its behavior causes during execution.
On the other hand, dynamic analysis allows the study of the behaviors that the binary code shows during its execution, as well as the side effects caused. This allows the analyst to perform in-depth monitoring of the actions performed by the software and the interactions that it has with the system in which it is executed.
In all areas in which such analysis is carried out, with particular attention to malware analysis, the code under analysis must necessarily be executed in an isolated system, or sandbox, in order to prevent side effects from causing significant problems to the main system (host). The problem is that sandboxes can often be computationally expensive, not only for the virtual machine itself, but also for the necessary monitoring tools, which remain active for the entire duration of the execution.
The integration, and the combination, of these two analysis methodologies allow

you to exploit the advantages of both methodologies and reduce the limitations that they present.[48]

Tools such as BinSequence, or BinGold[4], are a first step in this direction. These tools, in fact, perform an analysis divided into two phases.

In a first phase they perform a static analysis process, with which they identify the most critical areas of the binary code.

They then start the second phase, characterized by a dynamic analysis process, with which they analyze and investigate the cause of certain behavioral anomalies.

This approach significantly reduces the number of false positives, characterizing the dynamic analysis, reducing, at the same time, the computational costs of the analysis processes and increasing their accuracy.

A future evolution in this sense could include the implementation of hybrid algorithms that can simultaneously perform static and dynamic analysis processes of a binary code, combined in a single model. In this way, the time needed for the analysis would be significantly reduced, without losing the different fundamental characteristics of the binary code on which the two analysis methodologies focus.

Another evolution could be the use of advanced sandboxes, integrated into the analysis tool itself. In this way it would be possible to perform the analysis on the host itself, without needing computational resources to manage a virtual machine that acts as an isolation and without having to worry about the integrity of the system on which the analysis takes place.

As highlighted previously, the integration of these types of analyses not only significantly improves the analysis capacity, but also allows a reduction in the necessary computational resources. This last aspect allows the execution of analyses even on datasets with a high number of binary codes, improving the scalability of the analysis tools.

5.2 More Efficient Machine Learning Models Development

BCSD techniques based on machine learning and deep learning models are already present in today's literature and are showing more than optimal results for effectiveness and accuracy of the analyses and overcoming the limits and challenges of the more traditional techniques: text-based and logic-based.

The main reason why these approaches are still uncommon is the computational weight that the analyses involve. A weight that requires a high number of computational resources and that limits these approaches in the scalability of the datasets. An example of this condition are the techniques based on GNNs (Graph Neural Networks); which, having to represent the semantic characteristics of the binary code in the form of CFGs, require a particularly significant amount of computational resources for the graph processing, comparison and even training phases on large datasets.

The evolutionary direction undertaken for techniques based on machine learning or deep learning algorithms consists precisely in trying to reduce and optimize the computational resources that the algorithms need, making them lighter and more scalable.

The work of Hamilton et al.: GraphSage[49], presents itself as an idea of solving the great limit to which semantic-based techniques are subject. The idea behind GraphSage is to reduce the computational complexity of graph-based algorithms, such as GNN, by extracting and comparing a portion of the graph at a time, allowing for computationally lighter, but equally effective and precise, binary code analysis.

Another solution, mainly focused on deep learning-based models, is the one proposed by Zhu et al.[50], which involves optimizing semantic-based models using pruning and quantization techniques.

As for the pruning process, this consists of removing irrelevant nodes, or weights, present in neural networks. In this way, it is possible to reduce the size of the model without compromising its effectiveness.

For the quantization process, the precision of the weights of the models is reduced, in order to reduce the computational load in the inference phase.

Another approach, as already seen previously, is the pre-trained models followed by a fine-tuning phase, or specialization. This type of approach is particularly effective because it allows to exploit the knowledge acquired, in the pre-training phase, on general, large-sized datasets and then apply it to specific contexts. The specialization of the model is done in the fine-tuning phase, with the use of smaller datasets specialized to a specific context.

An integration of these approaches is certainly the few-shot training, that is, a training model based on a reduced amount of labeled data.

From the point of view of computational infrastructures, the computational weight of the models in the training phase can be reduced by using GPU or TPU clusters on the cloud. Methods that would make large-scale training accessible, without impacting the resources of the system on which the analysis is performed.

5.3 Large Scale Dataset and Benchmarking

Among the main points of improvement of BCSD techniques that the current literature focuses on are the availability of large datasets and the improvement of benchmarking techniques.

As far as datasets are concerned, the scarcity of sufficiently diversified datasets containing high-quality data does not allow to train models that are robust enough to provide reliable comparative analyses.

Another problem caused by the scarcity of large public datasets is the impossibility of comparing the performances of different approaches, preventing a focus on the points that need to be improved.

The datasets currently available in this field are proprietary or in any case limited to specific application contexts. They often only cover the most common architectures, preventing effective generalization of models in new contexts.

In order to manage such a problem, initiatives such as BigCode[51][52] are starting to exist, whose aim is to create a dataset as varied as possible, containing data from public repositories, such as GitHub. The aim is to provide datasets varied enough to allow models to operate on increasingly different and complex contexts, improving their ability to handle scenarios of binary codes compiled with different optimizations or subject to code obfuscation techniques.

The limits on benchmarking, on the other hand, are represented by the fact that, currently, benchmark techniques evaluate the standard metrics of accuracy, precision and recall. Metrics that may not be sufficient to ensure that the complexity of BCSD techniques is adequately assessed.

One solution could be to introduce the robustness metric, to give an assessment with respect to code obfuscation techniques, and the cross-architecture metric, to evaluate the adaptability of BCSD techniques with respect to binary codes compiled for different architectures.

To be able to give a value to such metrics, one solution is to create ad hoc databases, such as the CodeXGLUE project[53], which will support models in the standardization of comparison methodologies between different approaches and in the development of increasingly complex techniques, capable of facing the evolution of elusive techniques.

5.4 Use of Transfer Learning and Fine-Tuning

Among the most observed directions, from the innovative point of view of BCSD techniques, is the use of transfer learning and fine-tuning techniques. Techniques that are already widely used in other fields where machine learning is widely used, such as NLP and computer vision processes.

The transfer learning technique consists in training the model on a large dataset to re-adapt it to a new function using smaller datasets.

Within the BCSD techniques, transfer learning could find wide use on models pre-trained on a specific hardware architecture, such as x86 or MIPS, to re-train them on a new architecture, such as ARM. In this way, the model is able to perform cross-architecture analysis with greater efficiency and a lower demand for necessary computational resources.[54]

An example of the use of the transfer learning technique can be found in tools such as SAFE (Self-Attentive Function Embedding). Tools that have already demonstrated the potential of such a technique in areas such as malware analysis and cross-platform vulnerability identification.

In particular, SAFE uses a self-attention-based architecture, with which it is able to extrapolate the relationships between different binary functions and learn increasingly general representations.

A tool of this type, through a fine-tuning process, could specialize in more specific tasks such as vulnerability analysis on specific architectures or cross-platform analysis.

Another technique of this line of thought is meta-learning, a technique that allows the model not only to learn how to manage a specific task, but also how to quickly adapt to ever-different tasks, with the smallest possible amount of data available.

Meta-learning is therefore a technique that could significantly reduce the time needed for the training phases, as well as the size of the datasets needed for the purpose, and at the same time increase the efficiency and flexibility of the analysis models to ever-changing scenarios. different.[55]

5.5 Improvement of Robustness against Code Obfuscation Techniques

As highlighted above, one of the main challenges that BCSD techniques face is binary code obfuscation techniques. These techniques are used to alter the structure of the binary code and, consequently, its logic, without modifying its functioning.

Code obfuscation techniques make analysis complex for text-based approaches, such as N-Grams, and logic-based approaches such as CFGs.

The means of overcoming this problem was the use of BCSD techniques based on machine learning and deep learning, which have proven to be more robust.

This is because models such as GNNs, Asm2Vec or SAFE, are able to extrapolate and represent the semantic characteristics of the code, focusing only on the behavioral patterns of the binary code in execution.

In order to strengthen even the most traditional techniques, models, such as BERT, based on the concept of self-attention and transformer, have been integrated. These models, widely applied in NLP processes, are used to allow the analysis algorithm to understand not only the single instruction, but the entire context in which it is placed; going, in fact, to overcome the syntactic limits of the algorithm itself. This integration, in fact, makes it possible for the algorithm to observe the coherence of the binary code block, focusing on parts that maintain the meaning, ignoring the alterations,

Beyond this integration, there are tools, such as BinDiff and Gemini, in which reverse engineering processes have been integrated that are useful for de-obfuscation of the binary code being analyzed. A process that allows the identification of the key elements of the code being analyzed even in scenarios of heavily obfuscated codes.

Seeing the analogy between the analysis of obfuscated binary codes and the processing of natural languages, it is possible to imagine a future integration of the most effective NLP techniques. This integration could prove particularly effective in supporting the improvement of the robustness of BCSD techniques in the face of binary code obfuscation techniques, cross-platform compilations, different compiler optimizations.

Chapter 6

Discussion and Conclusions

This chapter aims to present a detailed and in-depth analysis of the approaches, previously discussed, on which Binary Code Similarity Detection (BCSD) techniques can be based.

The aim of this chapter is to compare the different methodologies introduced in order to evaluate their strengths and weaknesses. This comparison will be carried out on the basis of some factors that represent real challenges for BCSD techniques: compiler optimizations, binary code obfuscation techniques, cross-architecture compilation.

Another aspect on which we want to focus are the gaps that, despite the enormous potential of use of BCSD techniques in the world of cybersecurity, prevent these techniques from being applied on a large scale, even outside the IT world.

As mentioned, the focal theme of this chapter is the comparison between text-based, logic-based and semantic-based approaches. Approaches underlying BCSD techniques. Each of these approaches will be analyzed on the basis of the limitations it must face, from the point of view of the syntax, logic and semantics of the code. The difficulties of text-based approaches when faced with a reordered or obfuscated code will be highlighted, as well as the limits of scalability and computational complexity of semantic-based approaches.

We also want to highlight the gaps in current literature on the BCSD topic. In fact, it is not new to deal with tools operating with BCSD techniques, performing in controlled scenarios, with specific datasets, but failing in more generic contexts. This replicability problem is another obstacle to the use of these techniques, especially in critical environments such as malware analysis.

6.1 Approaches Comparison

Throughout this document, the three approaches on which BCSD techniques are based have been explored and detailed: text-based, logic-based and semantic-based. These approaches have proven to be the evolution of each other, as each approach has solved problems of the previous one.
The advantages and disadvantages of each approach have been highlighted, which this paragraph aims to retrace and compare.
Starting from the text-based approach, as seen previously, it is a type of approach based on the syntactic structure of the binary code. Its focus is placed on the sequence of binary instructions, seeing the binary code as if it were a text file. In this way, the algorithms based on this approach have proven to be easy to implement and, requiring few computational resources, fast in comparisons. This allows them to quickly analyze even large datasets, proving to be scalable. The problem with this approach is the strong dependence on the syntax of the binary code. This means that simply reordering the instructions would distort the result of the comparisons. Such a problem makes the text-based approach ineffective when faced with binary codes compiled with different optimizations, cross-architecture compilations and code obfuscation techniques.
To solve these problems, logic-based approaches have been introduced. These approaches, rather than focusing on the sequences of binary instructions, are focused on the logic, even if static, of the function calls. The techniques based on logic-based approaches operate mainly on two types of graphical representations: control flows (CFG) and data flows (DFG). In this way, the dependence on the syntax of the binary codes is overcome.

This turns out to be more robust than the text-based approach, when faced with different compiler optimizations, cross-architecture scenarios and code obfuscation techniques. The problem now concerns the strict dependence on the program logic, which can be modified by introducing noise, or a logic that is irrelevant for the purposes of execution. Furthermore, having to manage not only simple strings, but more or less complex graphs, and requiring manual feature extraction, this approach also sees a strong increase in computational complexity and, consequently, a reduction in scalability with respect to large datasets.

To solve the problem of dependence on logic, machine learning and deep learning technologies have been introduced; giving rise to the semantic-based approach. A peculiarity of this approach is its ability to extrapolate the functional behavior of the program in execution.

This ability has solved every problem related to code or logic obfuscation, cross-architecture and cross-compiler scenarios.

In fact, being able to understand the general behavior of the binary code in execution phase, this approach is able to extract behavioral patterns that it can search for in other binary codes under analysis.

In order for this process to be effective and adequately reliable, it is necessary that the model is sufficiently trained. This sufficiency is often achieved by training with datasets of significant size.

Such training causes significant computational complexity, which often makes it impossible to perform analysis on datasets of equal size, thus reducing scalability.

Another problem is the need to have high-quality and diversified training data available. Datasets that, to date, tend to be scarce.

Another problem consists in the difficulty of interpretability, as it is difficult to understand the reasons why a model has made a certain choice.

Ultimately, after having analyzed in detail the pros and cons of each individual approach, we can conclude by saying that there is no ideal approach in an absolute sense. Rather, we can affirm the existence of an ideal approach in a specific application scenario.

6.2 Conclusions

The purpose of this document was to make a description as detailed as possible of the Binary Code Similarity Detection (BCSD) techniques.

A description that began by defining the general process of these techniques and their goal of identifying similarities between different binary codes or between blocks of code within the same program.

The applications of these techniques in the computer science world were analyzed, with particular attention to some cybersecurity environments: malware analysis, vulnerability detection, copyright violation detection, bug identification.

The approaches on which the BCSD techniques are based were analyzed: text-based, logic-based and semantic-based; showing the advantages and disadvantages of each approach, such as the calculation speed of text-based approaches or the computational complexity of logic-based approaches or, again, the ability to identify behavioral patterns of semantic-based approaches. For each approach, the main techniques for which they are applied have been detailed, such as: fingerprinting and N-Gram for text-based approaches; Control Flow Graphs (CFGs) and Data Flow Graphs (DFGs) for logic-based approaches; Graph Neural Networks (GNN) and Neural Machine Translation (NMT) for semantic-based approaches, highlighting, also in this case, advantages and disadvantages.

Some tools operating with BCSD techniques are described, such as SAFE, Asm2Vec or Gemini.

An accurate description of the main problems that BCSD techniques have to face and overcome, such as: binary code obfuscation techniques, scalability on datasets, different compiler optimizations and cross-architecture scenarios, has been made.

A focus has been given to the future directions towards which these techniques should evolve, such as the integration of the combination of static and dynamic analysis or the use of transfer learning techniques.

Key points of the approaches on which BCSD techniques are based have been highlighted, which have led to the understanding that there is no approach that is better than the others in an absolute sense, but there are approaches that perform better than others in certain scenarios.

Furthermore, evident gaps have been shown of the current literature on BCSD techniques that make them difficult to compare, as they are often not replicable except in controlled scenarios, with datasets specific to that context. Gaps that prevent an effective improvement of these techniques and, consequently, their effective use in areas where they could be particularly useful.

Bibliography

[1] Zhenshan Li, Hao Liu, Ruijie Shan, Yanbin Sun, Yu Jiang, and Ning Hu. Binary code similarity detection: State and future. In *2023 IEEE 12th International Conference on Cloud Networking (CloudNet)*, pages 408–412, 2023.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[4] Saed Alrabaee, Lingyu Wang, and Mourad Debbabi. Bingold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (sfgs). *Digital Investigation*, 18, 08 2016.

[5] Yucong Du, Pengcheng Li, and Shuyang Zhao. On the inductive blockwise alperin weight condition for the unipotent blocks of finite groups of lie type e_6, 2021.

[6] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 480–491, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 896–899, New York, NY, USA, 2018. Association for Computing Machinery.

[8] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data, 2020.

[9] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489, 2019.

[10] Jinxue Peng, Yong Wang, Jingfeng Xue, and Zhenyan Liu. Fast cross-platform binary code similarity detection framework based on cfgs taking advantage of nlp and inductive gnn. *Chinese Journal of Electronics*, 33(1):128–138, 2024.

[11] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724, 2015.

[12] He Huang, Amr M. Youssef, and Mourad Debbabi. Binsequence: Fast, accurate and scalable binary code reuse detection. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 155–166, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17. ACM, October 2017.

[14] Roberto Baldoni, Giuseppe Antonio Di Luna, Luca Massarelli, Fabio Petroni, and Leonardo Querzoni. Unsupervised features extraction for binary similarity using graph embedding neural networks, 2018.

[15] Linux Base. Object files. https://refspecs.linuxbase.org/elf/gabi4+/ch4.intro.html.

[16] Zynamics. Bindiff, 2023.

[17] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. Safe: Self-attentive function embeddings for binary similarity, 2019.

[18] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. *SIGPLAN Not.*, 51(6):266–280, jun 2016.

[19] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity, 2019.

[20] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G. Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Inf. Sci.*, 231:64–82, may 2013.

[21] Noam Shalev and Nimrod Partush. Binary similarity detection using machine learning. *Conference: the 13th Workshop*, pages 42–47, 10 2018.

[22] Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects (english version). *SSTIC*, 5, 01 2005.

[23] R. Al-Ekram and K. Kontogiannis. Source code modularization using lattice of concept slices. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, pages 195–203, 2004.

[24] Xiaojun Xu, Ruoyu Qiao, and Yinqian Zhang. Gemini: Learning effective binary code similarity detection with graph neural networks. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 324–335. ACM, 2017.

[25] Gerhard Reitmayr. *On Software Design for Augmented Reality.* PhD thesis, Institut 188 fur Software Technologie und Interaktive Systeme, 01 2004.

[26] Y. Zhou, X. Zhang, F. Qin, and Y. Zhou. TaintFlow: A Practical Data Flow Tracking System for Detecting Taint-style Vulnerabilities. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[27] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *Proceedings 2019 Network and Distributed System Security Symposium*, NDSS 2019. Internet Society, 2019.

[28] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2099–2116, Boston, MA, August 2022. USENIX Association.

[29] He Huang, Amr M. Youssef, and Mourad Debbabi. Binsequence: Fast, accurate and scalable binary code reuse detection. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 155–166, New York, NY, USA, 2017. Association for Computing Machinery.

[30] Zhengping Luo, Tao Hou, Xiangrong Zhou, Hui Zeng, and Zhuo Lu. Binary code similarity detection through lstm and siamese neural network. *EAI Endorsed Transactions on Security and Safety*, 8(29), 9 2021.

[31] Mohannad Alhanahnah, Qicheng Lin, Qiben Yan, Ning Zhang, and Zhenxiang Chen. Efficient signature generation for classifying cross-architecture iot malware. In *2018 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9, 2018.

[32] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation, 2016.

[33] Zeng Chen Xiaodong Zhu, Liehui Jiang. Cross-platform binary code similarity detection based on nmt and graph embedding. *Mathematical Biosciences and Engineering*, 18(4):4528–4551, 2021.

[34] Putelli Luca. *Attention Mechanism e Interpretabilità del Deep Learning per il Natural Language Processing in Ambito Biomedico*. PhD thesis, Università degli Studi di Brescia, 2021. Tesi di Dottorato.

[35] Yusuke Oda, Philip Arthur, Graham Neubig, Koichiro Yoshino, and Satoshi Nakamura. Neural machine translation via binary code prediction, 2017.

[36] Yuan Quan, Longfei Shi, Jialei Liu, and Jiazhi Ma. A novel bistatic joint radar-communication system in multi-path environments, 2020.

[37] Yeming Gu, Hui Shu, and Fan Hu. Uniasm: Binary code similarity detection without fine-tuning, 2023.

[38] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. α diff: Cross-version binary code similarity detection with dnn. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 667–678, 2018.

[39] Guangming Liu, Xin Zhou, Jianmin Pang, Feng Yue, Wenfu Liu, and Junchao Wang. Codeformer: A gnn-nested transformer model for binary code similarity detection. *Electronics*, 12(7), 2023.

[40] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation, 2015.

[41] Sunwoo Ahn, Seonggwan Ahn, Hyungjoon Koo, and Yunheung Paek. Practical binary code similarity detection with bert-based transferable similarity learning. In *Proceedings of the 38th Annual Computer Security Applications Conference*, ACSAC '22, page 361–374, New York, NY, USA, 2022. Association for Computing Machinery.

[42] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, mar 2000.

[43] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. SelectiveTaint: Efficient data flow tracking with static binary rewriting. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1665–1682. USENIX Association, August 2021.

[44] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. Loop: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 757–768, New York, NY, USA, 2015. Association for Computing Machinery.

[45] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[46] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. Binsim: trace-based semantic binary diffing via system call sliced segment equivalence checking. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 253–270, USA, 2017. USENIX Association.

[47] Victor Cochard, Damian Pfammatter, Chi Thang Duong, and Mathias Humbert. Investigating graph embedding methods for cross-platform binary code similarity detection. In *2022 IEEE 7th European Symposium on Security and Privacyc (EuroS&P)*, pages 60–73, 2022.

[48] Michael Ernst. Static and dynamic analysis: Synergy and duality. *WODA 2003: Workshop on Dynamic Analysis*, pages 24–27, 05 2003.

[49] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1025–1035, Red Hook, NY, USA, 2017. Curran Associates Inc.

[50] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression, 2017.

[51] Kisub Kim, Sankalp Ghatpande, Dongsun Kim, Xin Zhou, Kui Liu, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Big code search: A bibliography. *ACM Comput. Surv.*, 56(1), aug 2023.

[52] BigCode. Bigcode project. https://github.com/bigcode-project.

[53] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021.

[54] Jaekoo Lee, Hyunjae Kim, Jongsun Lee, and Sungroh Yoon. Transfer learning for deep learning on graph-structured data. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1), Feb. 2017.

[55] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks, 2017.

[56] Yaniv David and Eran Yahav. Tracelet-based code search in executables. *SIGPLAN Not.*, 49(6):349–360, jun 2014.

[57] Xia Bing Pang Jianmin Zhou Xin Shan Zheng Wang Junchao Yue Feng. Binary code similarity analysis based on naming function and common vector space. *Scientific Reports*, 2023.

[58] Welcome to Technology. Processo di compilazione. https://welcometothetechnology.altervista.org/processo-di-compilazione/, 2024.

[59] Zhongtang Zhang, Shengli Liu, Qichao Yang, and Shichen Guo. Semantic understanding of source and binary code based on natural language processing. In *2021 IEEE 4th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, volume 4, pages 2010–2016, 2021.