International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 9
***Article Received:** 25 July 2023 **Revised:** 12 September 2023 **Accepted:** 30 September 2023*

An Inclusive Report on Robust Malware Detection and Analysis for Cross-Version Binary Code Optimizations

S. Poornima[1], R. Mahalakshmi[2]
[1]Department of CSE, Presidency University, Bangalore
Email: poornima.s@presidencyuniversity.in
[2]Professor, Department of CSE, Presidency University, Bangalore
Email: mahalakshmi@presidencyuniversity.in

Abstract: Numerous practices exist for binary code similarity detection (BCSD), such as Control Flow Graph, Semantics Scrutiny, Code Obfuscation, Malware Detection and Analysis, vulnerability search, etc. On the basis of professional knowledge, existing solutions often compare particular syntactic aspects retrieved from binary code. They either have substantial performance overheads or have inaccurate detection. Furthermore, there aren't many tools available for comparing cross-version binaries, which may differ not only in programming with proper syntax but also marginally in semantics. This Binary code similarity detection is existing for past 10 years, but this research area is not yet systematically analysed. The paper presents a comprehensive analysis on existing Cross-version Binary Code Optimization techniques on four characteristics: 1. Structural analysis, 2. Semantic Analysis, 3. Syntactic Analysis, 4. Validation Metrics. It helps the researchers to best select the suitable tool for their necessary implementation on binary code analysis. Furthermore, this paper presents scope of the area along with future directions of the research.

Keywords: Binary code analysis, Cross version Optimization, Anti-Malware Systems.

I. Introduction:

To create new software, programmers reuse existing code. Finding the source code from another project and using it for their own purposes is a widespread practise among them [1]. In order to speed up the development process, inexperienced developers will even copy and paste code examples from the Internet. Deep consequences for software security and privacy result from this trend. When a developer copies a problematic function from an already-existing project, the problem will persist even after the original developer fixes it. The corporation may also be charged with infringing an open-source licence, such as the GNU General Public Licence (GPL), if a developer for a commercial software company unintentionally uses library code from an open-source project [2].

Unfortunately, employing a similarity analysis to identify such issues in binary code is not always easy, especially when the source code is not accessible. This is due to the absence of high-level abstractions in binary code, such as data types and functions. Determining whether a memory cell represents an integer, a text, or another data type, for instance, cannot be done just by looking at the binary code. Furthermore, [3], [4] it is extremely difficult to pinpoint exact function boundaries in the first place. Binary code similarity detection (BCSD) is the process of determining whether two binary functions are similar. Code plagiarism detection [32, 33, 43], malware family and lineage analysis [2, 26, 28] are only a few of the numerous applications in which it is crucial. When using BCSD on pre-patch and post-patch binaries, further applications include the analysis of 1-day (i.e., patched) vulnerabilities [5] and the summary of vulnerability patterns [53]. Additionally, it might be applied to cross-architecture bug finding [16, 17, 52] when BCSD is applied to known bugs and target applications. However, BCSD confronts a number of difficulties.

Security practitioners face an increasing need to quickly detect similar functions directly in binaries across multiple platforms, e.g., x86, ARM, or MIPS. Only recently, researchers have started to tackle the problem of cross-platform binary code similarity detection [16, 18, 31]. These efforts propose to extract directly from binary code various robust platform-independent features for each node in the control flow graph to represent a function.Cross-optimization binaries are produced by several compiler optimisations first. Second, cross compiler binaries are produced by compilers using various strategies (such as register allocation). Third, cross-architecture binaries are produced when source code is generated on various platforms (e.g., using various instruction sets). Despite having differing grammatical patterns, these binaries are semantically similar. On the other side, cross-version binaries may result from the source code itself

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 9
***Article Received:** 25 July 2023 **Revised:** 12 September 2023 **Accepted:** 30 September 2023*

changing over time (such as through patches). These binaries are similar by nature since they share a common root. However, their semantics and syntactic structures differ slightly. Existing solutions could somewhat handle these BCSD issues, but they struggle with cross-version binaries.

II. Related Works:

A Traditional binary code similarity detection

Many methods that are similar to binary coding have been developed, taking into account its applications and difficulties. Traditional techniques typically use Binhunt8 and iBinHunt9. Multi_MH11, discovRE12, BinClone10, etc. To find the semantic differences, BinHunt used a novel graph isomorphism technique, symbolic execution, and theorem proving. Deep taint was used by iBinHunt to spot semantic differences in control flow between programmes, however it was inefficient and had high overhead. BinClone represents a chunk of binary code as a bit-vector and uses hashing to extract a fixed-length value from a variable-length instruction sequence in order to compute similarity. The first cross-architecture binary code search technique was Multi_MH. It used input and output semantics to index functions. DiscovRE employs a backtracking approach to fix inaccurate matches and used the basic block's call and arithmetic instruction counts as features. These techniques, however, take a lot of time and are challenging to use with numerous function pairings. A tool named Esh13 and its sequel GitZ14 were created by David et al. This has a high degree of accuracy for large-scale detection. Bingo, a scalable and reliable binary search engine described by Chandramohan et al.15, caught the whole function semantics by inlining relevant libraries and user-defined functions. They have the drawback that dynamic analysis must work in conjunction with static analysis and that the cross-optimization option scenario has a high false positive rate.

B Deep Learning based cross-version Binary Code Similarity Detection

End-to-end detection methods and multi-stage detection methods are two categories of deep learning-based methodologies. The primary techniques for end-to-end detection include Dif16, Asm2vec17, CodeCMR18, etc. These techniques use instructions or raw bytes to directly extract features, avoiding manually chosen features. The features of the multi-stage detection techniques include both feature selection and encoding. Tey confirmed that SimInspector's similarity detection accuracy rate is roughly 6% higher than Gemini's. A prototype system called INNEREYE was put into use by Zuo et al.22 for extensive binary code analysis.Tey encoded categorical features using embeddings for natural language processing (NLP). A hybrid model called BinDeep was proposed by Tian et al.23. BinDeep employed RNN to determine the specific categories of two functions and siamese neural networks to determine how similar two functions were to one another. Even though embedding has the ability to learn features automatically, it offers no information regarding the learning content. A bug search method called Genius was proposed by Feng et al.19, which initially employed embedding vectors for feature selection. Gemini is a cutting-edge neural network-based method for similarity identification that was put forth by Xu et al. Gemini embeds a function's control flow graph (CFG) using a graph embedding paradigm. SimInspector, a similar detection technique put forth by Zhu et al.21, is based on neural machine translation (NMT) and graph embedding. Massarelli et al.24 proposed the SAFE architecture, a revolutionary design that delivers great performance by directly extracting function characteristics from assembly instructions. Based on this article, Tey added several application scenarios and published it in 2021. Due to the fact that SAFE architecture does not have to worry about the computational costs associated with creating or modifying control flow graphs, it has a significant speed advantage. Numerous architectures and stripped binaries. However, there are still some areas that want improvement. According to their model, the latter terms have a bigger influence on the outcomes than the former ones. It will be less effective and have a negative effect when used to capture the semantics of the entire function. Figure.1 Demonstrates the malware classification approaches available and can be utilized.

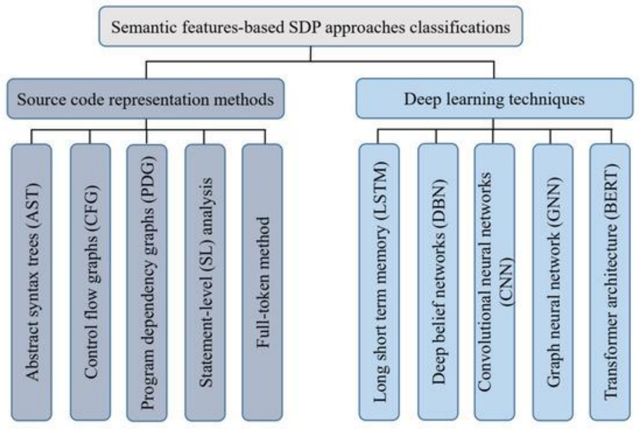
Semantic features-based SDP approaches classifications
Source code representation methods
Deep learning techniques
Abstract syntax trees (AST)
Control flow graphs (CFG)
Program dependency graphs (PDG)
Statement-level (SL) analysis
Full-token method
Long short term memory (LSTM)
Deep belief networks (DBN)
Convolutional neural networks (CNN)
Graph neural network (GNN)
Transformer architecture (BERT)

Fig.1 Malware Classification Approaches

C. Cross Version Binary Code Optimization: Overview

Many studies have been conducted recently that have two primary orientations and focus on cross-platform BCSD. The intersection of two given programmes' associated strand sets (such as code slices tracelets, tokens, and expressions) can be used to compare two programmes by representing them with various platform-independent strands and measuring how

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 9
***Article Received:** 25 July 2023 **Revised:** 12 September 2023 **Accepted:** 30 September 2023*

similar they are. Another approach is to extract various robust platform-independent properties from the control flow graph (CFG) vertices and compare them using graph matching. The most common scenario is detecting the similarity of a pair of binary functions coming from different platforms (such as x86, ARM, or MIPS).

Identifying if any two binary blocks are the same, similar, or equivalent is the goal of binary code similarity detection (BCSD). For malware detection [1] and vulnerability hunting, BCSD approaches are employed. The main challenge with BCSD concerns is that binary codes can be produced using various compilers, architectures, and versions of codes. When source code is compiled using various compiler algorithms, cross-compiler binaries are created. Cross-platform binaries are created by combining source codes with multiple instruction sets. When the source codes are updated and patched over time, cross-version binaries are produced [2]. If the functionality of cross-platform and cross-compiler binaries is the same but their syntactic features differ, then they are considered to be semantically similar. As they are compiled on the same platforms with the same root, the cross version binaries are comparable. Contrasting syntactic and semantic characteristics may still be present in the cross-version binaries. The methods for solving BCSD issues that are currently accessible rely on binary functions.

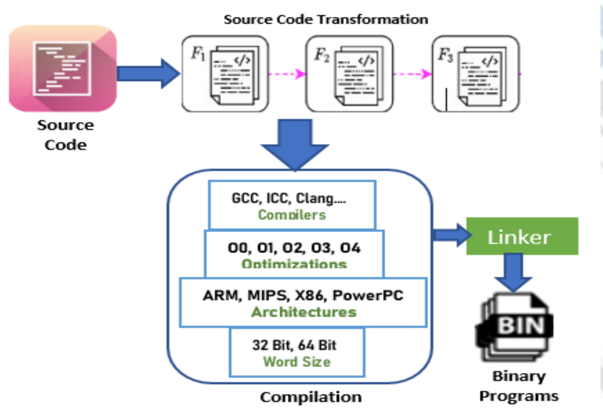

Fig.2 Cross version Binary Code Similarity Detection

Asm2Vec [18], INNEREYE [19] and SAFE [20] explore many new methods to compute the embedding vector of binary functions. Asm2Vec [18] employs representation learning to construct a feature vector for assembly code and provides more robustness to code obfuscation and compiler optimizations. INNEREYE [19] utilizes word embedding and LSTM to automatically capture the semantics and dependencies of instructions and solves the BCSD problem among basic blocks. SAFE [20] proposes a new architecture for computing binary function embedding from disassembled binaries and get better performance. Besides, Alpha-Diff [21], which is one of the state-of-the-art solutions to solve cross-version BCSD problems, represents the raw bytes of functions as images and uses the Siamese convolutional neural network to compute the similarity of functions.

III. Binary Analysis Approached in Cross – Versions BCSA:

The main challenge of cross-platform binary code similarity detection is choosing a proper code representation that not only can eliminate the influence of different instruction sets under different platforms to ensure the detection accuracy, but also facilitates efficient detection. According to code representations and matching algorithms, the existing BCSD methods can be classified into three categories: Structural level embedding, syntactic level embedding, semantic level embedding.

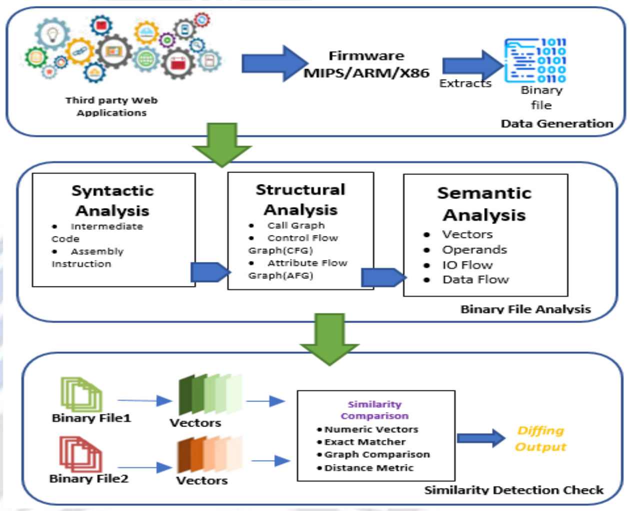

Figure.3 Binary Code Analysis Approaches

At a high level, CV-BCSA performs four major steps as described below:

(S1) Syntactic Analysis. Given a piece of binary code, one can parse it to produce an abstract syntax tree (AST) or disassembly of the code, often known as an intermediate representation (IR) [36]. This stage involves parsing source code into an AST, which is equivalent to the syntax analysis in conventional compiler theory. If the input code is a complete binary file, we first divide it into pieces based on the file type.

(S2) Structural Analysis. The control structures built into the binary code that were not immediately accessible from the syntactic analysis phase (S1) are analysed and recovered in this step. This stage specifically entails recovering the call graphs (CGs) and control-flow graphs (CFGs) in the binary code [37], [38]. Any property of these control structures can be used as a feature once the control-structural information has been collected. This process is distinguished from semantic analysis.

(S3) Semantic Analysis. The underlying semantics of the binary can be determined using conventional programme analyses, such as data-flow analysis and symbolic analysis, on the binary using the control-structural knowledge received from S2. This step allows for the generation of features that describe complex programme semantics, such as the flow of register values into different programme points. Along with the semantic data, the features acquired from S1–S2 can also be improved.

A. SCOPE & PAPER SELECTION

The scope of our assessment of the state-of-the-art must be clearly defined in order to keep it focused and manageable. Overall, the biggest limitation is that we only concentrate on binary code comparisons. The following four restrictions are then introduced by this restriction:

1. Source-to-source and source-to-binary similarity techniques, as those in [43] and [44], are not included since they call for access to the source code.
2. Avoided behavioural comparisons based only on how a programme interacts with its surroundings via system calls or OS API calls.
3. Excluded techniques like file hashes (e.g., [51]), fuzzy hashes (e.g., [52], [53]), and signature-based approaches (e.g., [54], [55]) that treat binary code as a series of raw bytes with no structure.
4. Papers that merely use commercial binary code similarity tools as a first step towards their goal is considered in priority.

Paper selection. First, all papers published in the last 20 years at 14 of the top conferences for software engineering and computer security—IEEE S&P, ACM CCS, USENIX Security, NDSS, ACSAC, RAID, ESORICS, ASIACCS, DIMVA, ICSE, FSE, ISSTA, ASE, and MSR—were thoroughly reviewed to find candidate papers. Not all pertinent binary code similarity techniques, particularly early ones, have been published in those forums. We performed comprehensive searches in specialised search engines, such as Google Scholar, using terms related to binary code similarity and its applications, such as code search, binary diffing, and bug search, in order to find potential candidate articles in other venues. Additionally, we carefully looked over the candidate papers' references to look for any additional papers we could have overlooked. More than one hundred candidate papers were found throughout this exploration. The next step was to study each potential article to see if it presented a binary code similarity approach that complied with the aforementioned scope restrictions. In the end, we identified the 61 binary code similarity research works, whose approaches are systematized.

B Key Assumptions from Past Research: Analysis

When reviewing the literature, it was discovered that the majority of methods heavily rely on the semantic properties gleaned from (S3), presuming that these features shouldn't vary between compilers or target architectures. But none of them explicitly explains why such intricate semantics-based analyses are required. They ignore the specific justifications for their tactics and only concentrate on the final outcomes.

In fact, this serves as the primary driving force behind our investigation. There might be basic aspects that we have missed, despite the fact that the majority of existing approaches concentrate on complicated studies. Depending on the target architecture and compiler, effective presemantic features, for instance, could be able to outperform semantic features. Due to the lack of a full investigation on these well-known qualities, it is possible that they have not been properly evaluated against the appropriate benchmark.

Additionally, current research presupposes the accuracy of the underlying binary analysis framework, such as IDA Pro [95], which is in fact the most widely used tool, as displayed in the rightmost column of Table 2. CFGs generated using those technologies, nevertheless, might be fundamentally flawed. For instance, they could omit some crucial basic building components, which would have a negative impact on the accuracy of BCSA features.

There have been several attempts to increase the accuracy of both analyses, which makes (S1) and (S2) demanding research challenges in and of themselves. For instance, deconstructing binary code is an intractable task [96], and developing a practical binary lifter that is both accurate and efficient is extremely difficult. Recovering control-flow edges for indirect branches [102] and identifying functions from binaries [3], [4], [96], [98], [99], [100], [101] are continuing active study areas.

IV. Experimental Setup:

Binary code similarity detection was experimented on a system with 128 GB SSD hard discs and 32 GB of RAM. Additionally, our token-level embedding generation method, Asm2Vec model, Gemini model, Safe model, and DeepBinDiff model are employed in the comparative studies on a server with two 3.2 GHz CPUs and two NVIDIA GeForce GTX rtx5000 graphics cards.

The statistics for our datasets show that 97% of assembly functions have fewer than 200 basic blocks, and 98.8% of basic blocks have fewer than 30 instructions. Most CFGs in our data sets are small graphs with a few nodes, as opposed to other big networks with millions of nodes, like the social network. Because the lengths of basic blocks in CFGs are also brief, we embed a basic block as

Tabel.2: Summary of related works

Year	**Prior Works**	**Binaries**	**Dataset Used**	**Architecture**				**Compiler**								**Optimization**			
				X68	**X86**	**ARM**	**MIPS**	**GCC9.5**	**GCC 0.4**	**GCC1.4**	**GCC 12**	**GCC 13**	**Clang 4**	**Clang 5**	**Clang 6**	**O0**	**O1**	**O2**	**O3**
2016	Genius	7,848	·	·	√	·	·	·	·	·	·	·	·	·	·	O	·	O	·
	Bingo	5,145	o	·	·	√	·	√	√	·	√	·	√	·	·	·	O	O	·
	Mocki-ngbird	80	●	√	·	√	·	·	·	·	√	·	·	√	·	O	·	·	O
	BinDNN	2,072	o	√	√	√	·	·	·	√	√	·	·	√	·	·	O	·	·
2017	Binsign	31	●	√	√	√	·	√	√	√	·	√	·	·	√	O	·	O	·
	Gemini	18,269	o	√	·	·	√	·	·	√	·	√	·	√	√	O	O	·	·
	BinSim	1,062	o	·	√	·	·	√	·	√ √	·	·	√	√	·	O	O	·	·
	BinSequence	1,718	●	√	√	·	√	·	√	√	·	·	√	√	·	O	·	O	·
2018	BinGo	5,145	o	√	·	√	·	√	·	√	·	√	·	√	·	O	O	O	·
	BinMatch	82	o	√	√	v	·	·	√	·	√	·	√	√	√	O	·	O	O
	BinArm	2,628	o	√	·	√	·	·	√	·	√	·	√	√	√	·	O	·	·
	αDiff	69,989	o	·	√	·	√	·	√	√	·	√	·	·	·	O	O	O	·
	VulSeeker	10,513	·	√	·	√	·	·	√	·	·	√	·	√	√	O	O	·	·
2019	Asm2Vec	68	●	√	√	·	√	·	·	√	·	√	·	√	√	O	·	O	·
	SAFE	5,001	·	√	√	·	√	·	√	√	·	·	√	√	·	·	O	O	·
	InnerEye	844	·	√	·	√	·	√	·	√	·	√	·	√	√	O	·	·	O
	FuncNet	180	●	√	√	v	·	·	√	·	√	·	√	√	√	·	O	·	·
2020	DeepBinDiff	2,206	o	√	·	√	·	·	√	·	√	·	√	√	√	O	·	O	·
	Patchek	2,108	·	√	√	·	√	·	√	√	·	·	√	√	·	O	O	O	·
	BinKit	2,42,876	●	·	·	√	·	√	√	·	√	·	√	·	√	O	·	O	·
	Deep Dual SD	4,132	●	√	·	√	·	·	·	·	√	·	·	√	√	·	O	O	·
	Trex	2,172	o	√	√	√	·	·	·	√	√	·	·	√	√	O	·	·	O
	FastSpec	1,098	●	√	√	√	·	√	√	√	·	√	·	·	√	·	O	·	·
2021	EnBinDiff	28,093	o	√	·	·	√	·	·	√	·	√	·	√	√	O	O	O	O
	BinDiffNN	79	o	√	√	·	√	·	√	√	·	·	√	√	√	O		O	
	Codee	1,534	o	√	·	√	·	√	·	√	·	√	·	√	√	O	O	O	O
	Asteria	2,098	o	√	√	v	·	·	√	·	√	·	√	√	√	O	O	O	O
	PalmTree	987	●	√	·	√	·	·	√	·	√	·	√	√	√	·	O	·	·
2022	JTrans	2,167	o	·	·	√	·	√	√	·	√	·	√	·	√	O	·	O	O
	XBA	4,324	o	√	·	√	·	√	·	√	√	·	√	√	√	O	O	·	O

O – Indicates partial dataset available ● –Dataset Not Available · - Not supportable and available
√ - Models supporting defined architecture and compiler
O – Optimizations availability

a value in the basic block embedding procedure, setting d = 1. By vectorising the fundamental block embedding matrix by columns—we set n1 = 200 with a function feature vector is produced. We build n2 = 1, 333 binary libraries for our data sets, and a binary file can only have n3 = 1000 assembly functions.

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 9
***Article Received:** 25 July 2023 **Revised:** 12 September 2023 **Accepted:** 30 September 2023*

Table.2 Benchmark Works utilized for BCSA

Related Works	Binaries	Source	Embedding	Architecture
Asm2Vec	1,209	API calls, memory objects	One hot Encoding	Siamese
αDiff	3,087	Dissembled Text	Word2Vec	RNN
Vulseeker	10,512	Control Flow Graph (ACFG)	Structure2Vec	Feed Forward
Genius	7,848	Bytes	Word2Vec	CNN
BinDiff	876	Dissembled Text	Context Vectors	LSTM
Gemini	18,269	Control Flow Graph (ACFG)	Structure2Vec	CNN+LSTM
Safe	5,001	Dissembled Text	Word2Vec	BiRNN, Siamese
DeepBinDiff	2,206	Instruction metadata	Word2Vec	ANN

A. Dataset:

We constructed our evaluation on datasets created for earlier binary analysis evaluations to support reproducibility. We specifically started with the BinKit corpus [36], which is based on all GNU software packages that are readily available. 53 software packages are included in BinKit, which was built using five different GCC versions (v4.9.4, v5.5.0, v6.4.0, v7.3.0, and v8.2.0) and four different clang versions (v1.0, v2.0, v3.0). The original corpus is made up of various separate datasets that use the -fno-inline, -fPIE, -Os, and -flto compiler options. We included new datasets (CFI) and more latest GCC and Clang versions (GCC v.9.4.0, GCC v11.2.0, Clang v9.0, Clang v13.0) to the corpus.

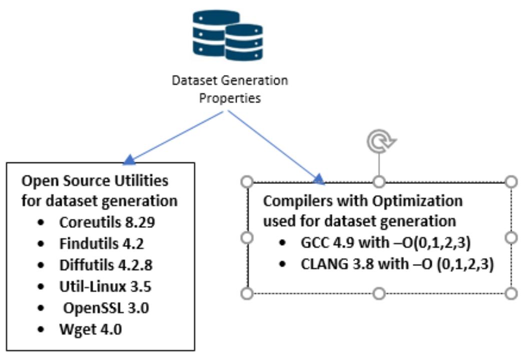
Dataset Generation Properties
Open Source Utilities for dataset generation
Coreutils 8.29
Findutils 4.2
Diffutils 4.2.8
Util-Linux 3.5
OpenSSL 3.0
Wget 4.0
Compilers with Optimization used for dataset generation
GCC 4.9 with –O(0,1,2,3)
CLANG 3.8 with –O (0,1,2,3)

Figure.4 Dataset gathered

B. Evaluation Criteria's:

The following five parameters are used as evaluation indicators in this article to quantitatively evaluate the effectiveness of the detection model network. The calculation formula is shown below:

1. PSNR

In image analysis, Peak Signal-to-Noise Ratio (PSNR) is the maximum value between the power of a signal and corrupting noise. In our case, this ratio measures how close an estimator image is to the estimated image. It is expressed in terms of the logarithmic decibel scale:

$$PSNR = 10log(\frac{C^2}{MSE})$$

Where C ^{2} is the largest possible value of individual pixels (the difference between the foreground and background), and the mean square error (MSE) is defined by the equation below:

$$MSE = \frac{\sum_{x=1}^{M}\sum_{y=1}^{N}(A(x,y) - B(x,y))^2)}{MN}$$

Where $\bar{A}$is the mean value of one array and $\bar{B}$ is the mean of another, and M and N are the dimensions of the arrays.

2. NRM

The relationship between elements that have been incorrectly classified and all other elements in the class is represented numerically by the Negative Rate Metric (NRM). It is the average of two false negative rates (NRFN, false negative rate) and false positive rates (NRFP, false positive rate).

$$NRM = \frac{NR_{FN} + NR_{FP}}{2}$$

$$NR_{FN} = \frac{FN}{TP + FN}$$

$$NR_{FP} = \frac{FP}{TN+FP}$$

A higher NRM indicates a worse mismatch between two classifiers.

NCC:

Normalized Cross Correlation is often used for comparing multidimensional arrays and is defined by the following equation:

$$NCC = \frac{\sum_{m=1}^{M}\sum_{n=1}^{N}(A(m,n) - \bar{A})(B(m,n) - \bar{B})}{\sqrt{\sum_{m=1}^{M}\sum_{n=1}^{N}(A(m,n) - \bar{A})^2 \sum_{m=1}^{M}\sum_{n=1}^{N}(B(m,n) - \bar{B})^2}}$$

Where M and N are the arrays' dimensions, $\bar{A}$ represents one array's mean value and $\bar{B}$ represents another.

V. Discussions:

We now briefly discuss a number of significant issues identified in the prior literature and present exploration stemming from these issues. The issues to be considered are

1. Not any study employs the same standard of measurement for evaluation, and the methods used to assess the effectiveness of the strategies used in the papers are very different.
2. Only a handful of the research make their source code and data available, which makes it extremely challenging to duplicate or improve on previous findings. In addition, the majority of articles employ manually selected and error-prone ground truth data for their evaluation.
3. Finally, state-of-the-art methods for BCSA currently emphasise the extraction of semantic information using sophisticated analysis methods.

The Discussions listed below are logical extensions of these issue identifications. Notably, we only partially address several of the open-ended questions.

OB1: Inconvenient Truths of Ground Truth for Binary Analysis
Binary analysis research involves automatic analysis of executable binaries and a transformation of those binaries into some intermediate representation that allows for complex analysis. How successfully binary analysis tools and approaches map to a ground truth is frequently used to gauge their efficiency. From the literature survey, it was found that not all ground facts are created equally. In order to evaluate tools and procedures with confidence, there must be surety that agreement on definitions of ground truth is in conformance. This study challenges the binary analytic community to examine the concept of ground truth carefully. Since there is a transition to train deep learning models, which are only as valuable as the accuracy of the ground truth throughout the training, this becomes even more crucial.

OB2: Basic BCSD not focusing on Cross Architectural features
The task of finding similarities between binary functions that are not present in the corresponding source code is known as binary code similarity detection (BCSD). It has been extensively applied to make key security analyses of many kinds in software engineering easier. The method of identifying binary code similarity is difficult due to the intricacy of programme compilation. A solid vector representation of binary code serves as the foundation for the most logical binary similarity detector. Few BCSD methods, though, are appropriate for constructing vector representations for comparing the similarities between binaries, which may differ not only in semantics but also in structures. Additionally, the current methods, which primarily rely on manual feature engineering to create feature vectors, do not account for the connections between instructions, Since it fails to consider cross optimization, cross compiler and cross architectural features.

OB3: Newer BCSD architectures can be developed by analysing failure instances of traditional BCSD
Since most current works rely on opaque machine learning algorithms, they rarely analyse their failure instances. However, our objective is to adopt a straightforward and understandable model to learn from mistakes and collect knowledge for future study. As a result, we carefully reviewed failure cases using our interpretable method and discovered three prevalent failure causes that the prior literature had mostly missed. First, it is true that traditional binary analysis techniques produce inaccurate results. Second, various compiler backend for the same architecture may differ greatly from one another. Third, the same function has code fragments that are particular to different architectures. The above mentioned views can be interpreted to develop effective BCSD system.

VI. An Inclusive report on Binary Code Similarity Detection:

We concentrate on malware attacks in our assessment of neural function boundary detection. The name of these attacks comes from the fact that no knowledge of the model-under-test's (MUT's) internal weights or structural details is assumed. Because malware attacks are dynamic, they have the advantage of not requiring a thorough knowledge of MUTs; rather, all that is required is the capacity to execute queries and track the outcomes. Malware detection techniques may, however, miss latent vulnerabilities that a white-box adversarial search like projected gradient descent (PGD) [44] may otherwise find because the search process is unguided by model knowledge. In this regard, the outcomes of our research ought to be seen as a lower constraint on the MUTs' susceptibility. To find and take advantage of function boundary misclassifications when inferring from binary programmes, we build a universal malware vulnerability search approach. Input generation, ground truth creation, training and inference, and misclassification analysis are the first four stages of the search process.

Generate Input: We compile a corpus of benchmark programme source code S in the first stage. A variety of compiler toolchains T are used to compile each benchmark, and each one has an attack configuration C that consists of a number of compiler flags and code changes. We obtain a benchmark binary corpus B made up of n distinct compiles of

S using each toolchain and attack configuration tuple (T,C), given n = |T | compilers.

Generation of Ground Truth: Each option makes sure that debugging information is produced while also guarding against the removal of symbol information from compiled binaries. In order to create a ground truth mapping F (1), or a function that labels each byte of code in each binary as either a function start, a function end, or neither, we can post-process each binary and use the information from these sources.

Misclassification Analysis: In the last phase, for each MUT we process its misclassification sets $E + m$, $E- m$ to identify attack inputs Am that can reliably produce function boundary misclassifications in arbitrary binary programs. For each model, we rank-order misclassifications from highest to lowest incidence in order to achieve this. As seeds for an adversarial search, the ranked attack inputs are then each successively injected into specific functions of B to create a mutant corpus B m. The MUT m is then used to perform inference on B m in a second attack validation cycle to make sure the targeted functions duplicate the intended misclassifications.

A. Limitations and future work:

The Binary Code Similarity Detections frameworks experimented still has certain drawbacks in addition to all of its benefits. First, function-level presemantic features are extracted in this research using the disassembly tool IDA; thus, the accuracy of the IDA analysis findings determines the dependability of function-level presemantic features. IDA, however, has the potential to produce inaccurate outcomes, such as incomplete CFG generation and wrong function border division. The error analysis is left as presemantic features extracted by disassembly tools for future research, given that IDA is extensively used in related work and the aim of this thesis is to employ the attention mechanism to fuse presemantic information to yield functional semantic features.

Second, Experiments shows the performance (TOP-1 = 0.1614, the lowest of all testing scenarios) will be greatly impacted by the wide variation between optimisation levels (O0-O3). This is due to the fact that high optimisation level intra- and inter-procedural structures differ greatly from low optimisation level structures. However, note that no related study achieves higher performance that depends on function structural information.

Third, only intra-procedural techniques like BCF, FLA, and SUB are taken into consideration; the LLVM is used in this research to test the experimental performance under cross-obfuscation. These techniques barely affect CG features, while they significantly influence CFG features. On the other hand, the obfuscation technique is more intricate in real-world situations and involves altering the function call relationship through inter-procedural Malware detection. We leave the investigation of Cross Version Binary Code Similarity detection techniques for future research because the majority of current studies do not support Cross-Platform programs.

VII. Conclusion and Future Work:

In this study, traditional BCSA techniques are explained by integrating the attention mechanism with interpretable features. To acquire semantic features of functions (i.e. features with attention information), we first extract the presemantic features of binary functions and then use the attention method for feature fusion. Lastly, the model is trained on these semantic features using the Siamese network. Using two datasets, we train distinct models for various compilation parameters in order to identify commonalities between function pairs and evaluate function search efficiency. Interestingly, we explored that frameworks has to generate semantic features which can withstand changes in architecture, optimisation level, obfuscation, and compiler, and can even surpass cutting-edge techniques. The main challenge of cross-platform binary code similarity detection is choosing a proper code representation that not only can eliminate the influence of different instruction sets under different platforms to ensure the detection accuracy but also facilitates efficient detection.

REFERENCES:

[1] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. 2019. A Systematic Review on Code Clone Detection. IEEE Access 7 (2019), 86121–86144.

[2] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing Memory Accesses in x86 Executables. In Compiler Construction, Evelyn Duesterwald (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 5–23.

[3] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to recognize functions in binary code. In 23rd USENIX Security Symposium (USENIX Security 14). 845–860.

[4] Oren Barkan, Edan Hauon, Avi Caciularu, Ori Katz, Itzik Malkiel, Omri Armstrong, and Noam Koenigstein. 2021. Grad-sam: Explaining transformers via gradient self-attention maps. In Proceedings of the 30th ACM International Conference on Information & Knowledge Management. 2882–2887.

[5] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. Clone detection using abstract syntax trees. In Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). 368–377.

[6] Leyla Bilge, Davide Balzarotti, William Robertson, Engin Kirda, and Christopher Kruegel. 2012. Disclosure: detecting botnet command and control servers through large-scale netflow analysis. In Proceedings of the 28th Annual Computer Security Applications Conference. ACM.

[7] BinDiff 2022. zynamics BinDiff. https://www.zynamics.com/bindiff.html

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. Advances in neural information processing systems 33 (2020), 1877–1901.

[9] Dong-Kyu Chae, Jiwoon Ha, Sang-Wook Kim, BooJoong Kang, and Eul Gyu Im. 2013. Software plagiarism detection: a graph-based approach. In Proceedings of the 22nd ACM international conference on Information & Knowledge Management. 1577–1580.

[10] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: Cross-Architecture Cross-OS Binary Search (FSE 2016). Association for Computing Machinery, New York, NY, USA, 678–689. https://doi.org/10.1145/2950290.2950350

[11] Hila Chefer, Shir Gur, and Lior Wolf. 2021. Transformer interpretability beyond attention visualization. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 782–791.

[12] Coreutils 2022. Coreutils - GNU core utilities.

[13] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical Similarity of Binaries. In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (Santa Barbara, CA, USA) (PLDI
’16). Association for Computing Machinery, New York, NY, USA, 266–280.

[14] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In 2019 IEEE Symposium on Security and Privacy (SP). 472–489.

[15] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing.

[16] S. Ducasse, M. Rieger, and S. Demeyer. 1999. A language independent approach for detecting duplicated code. In Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM’99). ’Software Maintenance for Business Change’ (Cat. No.99CB36360). 109–118.

[17] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In Proceedings of the 23rd USENIX Conference on Security Symposium (San Diego, CA) (SEC’14). USENIX Association, USA, 303–317.

[18] A. Jayachandran and R. Dhanasekaran, Brain tumor detection using fuzzy support vector machine classification based on a texton co-occurrence matrix, Journal of imaging Science and Technology 57(1) (2013), 10507-1–10507-7(7).

.

[19] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary. Association for Computing Machinery, New York, NY, USA, 896–899.

[20] GNU. 2022. gcc-9. Retrieved Feb 16, 2023.

[21] Yikun Hu, Hui Wang, Yuanyuan Zhang, Bodong Li, and Dawu Gu. 2021. A Semantics-Based Hybrid Approach on Binary Code Similarity Comparison. IEEE Transactions on Software Engineering (TSE) 47, 6 (June 2021), 1241–1258.

[22] Y. Hu, Y. Zhang, J. Li, H. Wang, B. Li, and D. Gu. 2018. BinMatch: A SemanticsBased Hybrid Approach on Binary Code Clone Analysis. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE Computer Society, Los Alamitos, CA, USA, 104–114.

[23] IDA Pro 2022. A powerful disassembler and a versatile debugger.

[24] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. 2014. Hulk: Eliciting malicious behavior in browser extensions. In 23rd USENIX Security Symposium (USENIX Security 14).

[25] Chariton Karamitas and Athanasios Kehagias. 2018. Efficient features for function matching between binary executables. In 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). 335–345.

[26] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. 2015. Cutting the gordian knot: A look under the hood of ransomware attacks. In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer.

[27] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2022. Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering and Lessons Learned. IEEE Transactions on Software Engineering (2022), 1–23.

[28] Raghavan Komondoor and Susan Horwitz. 2001. Using Slicing to Identify Duplication in Source Code. In Static Analysis, Patrick Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 40–56.

[29] J. Krinke. 2001. Identifying similar code with program dependence graphs. In Proceedings Eighth Working Conference on Reverse Engineering. 301–309.

[30] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. aDiff: Cross-Version Binary Code Similarity Detection with DNN. Association for Computing Machinery, New York, NY, USA, 667–678.

[31] Shengzhong Liu, Franck Le, Supriyo Chakraborty, and Tarek Abdelzaher. 2021. On exploring attention-based explanation for transformer models in text classification. In 2021 IEEE International Conference on Big Data (Big Data). IEEE, 1193–1203.

[32 A. Jayachandran and R. Dhanasekaran, Automatic detection of brain tumor in magnetic resonance images using multi texton histogram and support vector machine, International Journal of Imaging Systems and Technology 23(2) (2013), 97–103..

[33] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017.Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection. IEEE Transactions on Software Engineering 43, 12 (2017), 1157–1177. [34] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama.2018. MODE: Automated Neural Network Model Debugging via State

Differential Analysis and Input Selection. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Lake Buena Vista, FL, USA) (ESEC/FSE 2018). Association for Computing Machinery, New York, NY, USA, 175–186.

[35] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio,Mohamad Mansouri, and Davide Balzarotti. 2022. How machine learning is solving the binary function similarity problem. In USENIX 2022, 31st USENIX Security Symposium, 10-12 August 2022, Boston, MA, USA, Usenix (Ed.). Boston.

[36] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. CoRR abs/2103.11626 (2021).arXiv:2103.11626 .

[37] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2018. SAFE: Self-Attentive Function Embeddings for Binary Similarity.

[38] Tomás Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. CoRR abs/1310.4546 (2013). arXiv:1310.4546 http://arxiv.org/abs/1310.4546

[39] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic Disassembly. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). 1187–1198.

[40] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Tracebased Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In 26th USENIX Security Symposium (USENIX Security 17). USENIX Association, Vancouver, BC, 253–270.

[41] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. 2019.TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97), Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 4901–4911.

[42] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. Sok: All you ever wanted to know about x86/x64binary disassembly but were afraid to ask. In SP. IEEE, 833–851.

[43] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. 2021. Xda: Accurate, robust disassembly with transfer learning. In NDSS. The 1117 Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis ISSTA '23, July 17–21, 2023, Seale, WA, USA Internet Society.

[44] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. Trex: Learning Execution Semantics from Micro-Traces for Binary Similarity.

[45].Jesu Prabhu A and Jayachandran, A, "Mixture Model Segmentation System for Parasagittal Meningioma Brain Tumor Classification based on Hybrid Feature Vector" Journal of Medical System, vol 42, issues 12, 2018.

[46]. Jayachandran, A and R.Dhanasekaran ,(2017) 'Multi Class Brain Tumor Classification of MRI Images using Hybrid Structure Descriptor and Fuzzy Logic Based RBF Kernel SVM' , Iranian Journal of Fuzzy system , Volume 14, Issue 3, pp 41-54 , 2017.

[47] Nina Poerner, Hinrich Schütze, and Benjamin Roth. 2018. Evaluating neural network explanation methods using hybrid documents and morphosyntactic agreement. In Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 340–350.

[48] PyTorch 2023. An open source machine learning framework that accelerates the path from research prototyping to production deployment. https://pytorch.org

[49] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[50] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. OpenAI blog 1, 8 (2019), 9.

[51] rev.ng. 2023. Rethink Binary Analysis. Retrieved Feb 16, 2023 from https://rev.ng

[52] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. In 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). 1157–1168.

[53] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-cam: Visual explanations from deep networks via gradient-based localization. In Proceedings of the IEEE international conference on computer vision. 618–626.

[54] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (Phoenix, AZ, USA) (HASP '19). Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages.

[55] Ridwan Salihin Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. 2021. Automated Patch Transplantation. ACM Trans. Softw. Eng.Methodol. 30, 1, Article 6 (dec 2021), 36 pages.

[56] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In 24th USENIX Security Symposium (USENIX Security 15). 611–626.

[57] Guanhong Tao, Shiqing Ma, Yingqi Liu, Qiuling Xu, and Xiangyu Zhang. 2020. TRADER: trace divergence analysis and embedding regulation for debugging recurrent neural networks. In ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 986–998.

[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones,Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. CoRR abs/1706.03762 (2017). arXiv:1706.03762 .

[59] Jesse Vig. 2019. A Multiscale Visualization of Attention in the Transformer Model. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations. 37–42.

[60] Andrew Walker, Tomas Cerny, and Eungee Song. 2020. Open-Source Tools and Benchmarks for Code-Clone Detection: Past, Present, and Future Trends. SIGAPP Appl. Comput. Rev. 19, 4 (jan 2020), 28–39.

[61] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. JTrans: Jump-Aware Transformer for Binary Code Similarity Detection. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (Virtual, South Korea) (ISSTA 2022). Association for Computing Machinery, New York, NY, USA, 1–13.

[62] Shuai Wang and Dinghao Wu. 2017. In-Memory Fuzzing for Binary Code Similarity Analysis. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (Urbana-Champaign, IL, USA) (ASE 2017). IEEE Press, 319–330.

[63] Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code. Proc. ACM Program. Lang. 3, POPL, Article 62 (jan 2019), 30 pages. https://doi.org/10.1145/3290375

[64] Yuting Wang, Xiangzhe Xu, Pierre Wilke, and Zhong Shao. 2020. CompCertELF: Verified Separate Compilation of C Programs into ELF Object Files. Proc. ACM Program. Lang. 4, OOPSLA, Article 197 (nov 2020), 28 pages

[65] Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023. Artifact for DiEmph.

[66] Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023. Supplementary Material. Retrieved May 27, 2023

[67] Xi Xu, Qinghua Zheng, Zheng Yan, Ming Fan, Ang Jia, and Ting Liu. 2021. Interpretation-Enabled Software Reuse Detection Based on a Multi-level Birthmark Model. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). 873–884.

[68] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. SPAIN: Security Patch Analysis for Binaries towards Understanding the Pain and Pills. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). 462–472.

[69] Yapeng Ye, Zhuo Zhang, Qingkai Shi, Yousra Aafer, and Xiangyu Zhang. 2022. D-ARM: Disassembling ARM Binaries by Lightweight Superset Instruction Interpretation and Graph Modeling. In 2023 IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, 728–745.

[70] Wei You, Zhuo Zhang, Yonghwi Kwon, Yousra Aafer, Fei Peng, Yu Shi, Carson Harmon, and Xiangyu Zhang. 2020. PMP: Cost-effective Forced Execution with Probabilistic Memory Pre-planning. In 2020 IEEE Symposium on Security and Privacy (SP). 1121–1138.

[71] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. Proceedings of the AAAI Conference on Artificial Intelligence 34, 01 (Apr. 2020), 1145–1152.