# Binary Function Similarities

A Thesis
Presented in Partial Fulfillment of the Requirements for the
Degree of Master Science
with a
Major in Computer Science
in the
College of Graduate Studies
University of Idaho
by
Le Phu Bui

Major Professor: James Alves-Foss, Ph.D.
Committee Members: Jia Song, Ph.D.; Michael Haney, Ph.D
Department Administrator: Terence Soule, Ph.D.

August 2021

## Authorization to Submit Thesis

This thesis of Le Phu Bui, submitted for the degree of Master of Science with a Major in Computer Science and titled "Binary Function Similarities" has been reviewed in final form. Permission, as indicated by the signatures and dates below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor: _____ _____ Date: _____
James Alves-Foss, PhD

Committee Members: _____ Date: _____
Jia Song, PhD

_____ Date: _____
Michael Haney, PhD

Department Administrator: _____ Date: _____
Terence Soule, Ph.D.

## Abstract

The purpose of comparing two or more binary functions on two executable files is to identify the similarities of binary code. We summarize the work of researchers who have studied a similar function between two binary executable files. We determine a list of metrics that uniquely identify each part of the function between two binary executable files. The study focuses on extracting binary for 32 and 64-bit Intel binaries running under Linux. The goal is to isolate the sequence of metrics for each binary function and measure the similarity between each set of metrics. We will run the experiments on two binary executable files built with the same software, use different compiler or compiler versions, and rely on the same library and various versions of libraries. The set of metrics will represent the attributes of the binary function behavior with resource and time usage, data requirement, availability of function data type, parameters, and numerous details include basic block features, Data Flow Graphs (DFGs), Control Flow Graphs (CFGs) of functions.


***Keywords— binary code analysis, binary functions, executable file, static analysis, dynamic analysis, control flow graphs, basic feature blocks.***

## Acknowledgements

I would like to thank Dr. Jim Alves-Foss for his guidance and support throughout this study, and especially for his confidence in me. I also would like to thank my committee members, Dr. Song, Dr. Haney for their time and support of this research.

## Dedication

I would like to thank my parents for their love and support throughout my life. Thank you both for giving me an amazing opportunity to study abroad in the US.

# Table of Contents

## List of Tables

## List of Figures

# 1 Introduction

Binary analysis is a form of code review that analyzes the binary executable of an application without visibility into the source code. The study shows the existing binary code fingerprint extraction techniques that use different methodologies. The results of binary code analysis help with the discovery of potential vulnerabilities in an application. Specifically, vulnerabilities analysis is a significant task of general software assurance practice. The analysis process decomposes the complete application's raw binaries, which is especially useful for the unavailable source code, but it is extraordinarily complex. The advantage of binary analysis is identifying the actual code running on the system directly, and binary function similarity is a case study. Binary function similarity is the comparison between two executable binary files to identify their similarities and differences. The proposed approach is challenging because binary function similarity widely explores the software application's detail in its implementation and design. This paper summarizes the work of other researchers who have studied a similar function and develops set of metrics that uniquely identify every single part between two binary executable files.

The binary code similarity comparison has been an essential factor in software engineering and security because it helps identify identical code fragments, binary auditing, bug search, vulnerability detection, etc. Many researchers have isolated their curriculums to find the most meaningful binary similarity. Precisely, the metrics and formulas measure the accuracy in the binary code between two executable files. Distance measures have been applied to evaluate the robustness of the research in various methodologies. This paper generally presents the binary function similarities survey that uses a binary static analysis approach to analyze function features. For each mechanism's probability theory, (1) Control-Flow Graph (CFG) and Data-Flow Graph (DFG) represent a program's control structure in binary static analysis; (2) the traceless-based matching approach addresses the problem of code search in executable files; and (3) Locality-Sensitive Hashing (LSH) uses several hashing functions to devise the main memory algorithm for finding the nearest neighbor search. We focus on determining the signification of binary function features and examining them in the sequence of experiments. Furthermore, we use an existing tool, JIMA [1], for the investigation. We validate the proposed metrics against other approaches on binary function similarity. The purpose is to calculate the distance between each set of metrics through test results on two binary executable files.

The binary static analysis offers root-cause approaches to vulnerabilities and identifies the code clones and function behaviors. This research will analyze the binary functions to determine whether there is a similar code from one-to-one functions between programs. The programs are binary executables that run under the Linux and target architecture of binary code to be the mainstream

architecture Intel x86-32bit. We will use JIMA [1] as the torch in our research to collect the data. Ideally, we will focus on gathering the function granularities at the assembler level and analyzing the instruction and register structure to support our research; next, we will distribute each function element into the sequence of metrics. We will apply various algorithms to measure the similarity by comparing those metric results.

Beyond the static analysis, this research will also discuss the data flow analysis to determine the possible values calculated from one point to another through the program. We will use reaching definition analysis to track the way values of register change in the scope of function, and we intend to take a basic block of each function. We expect to obtain a program's CFG and pick a basic block to identify a particular value assigned or loaded to each register flow through a function. We will track from the bottom to the top of the function to recover the register's initial declaration. However, we cannot be certain of this analysis method's suitability for our research until we run the experiments.

In general, we will create a couple of programs to verify the accuracy of the metrics before running our approach in the test suites. The evaluation covers the datasets on which test cases obtain the most accurate results. We will learn which metrics are valid along with the type of analyses. Therefore, we will pursue this research as a learning experience to address some challenges for the binary function similarities. Still, we are able to make novel contributions to the field.

In table 1, we summarize and compare twenty approaches and narrow down two fields: (1) techniques that include the static and dynamic analysis and (2) approach characteristics, which cover semantic, syntactic, and statistical features. Throughout the paper, we address two objectives: (1) Similarity between executable files with same compiler version and program has different source code version and (2) similarity between executable files with different compiler and program has same source code version. Those objectives cover the following cases:

Case 1: Function is identical in both versions.

Case 2: Function has been modified between the versions.

Case 3: Function is unique to the versions.

**Key contributions**

(1) Describes the crucial role of binary code analysis with cybersecurity threats.
(2) Specifies the granularities of binary function for isolating the metrics field.
(3) Compares the set of metrics for the binary function similarity.
(4) Creates a metric to demonstrate JIMA [1] for binary function similarity.
(5) Evaluates the setup and method research in the experiment.
(6) Outlines the directions for future research.

# 2 Overview

This section identifies the motivation for studying the binary function and provides an overview of the binary analysis approach and binary function similarity problem.

## 2.1 Motivation

Binary analysis entails threats assessment and vulnerability testing at the binary code level. The process of analyzing binary functions without source code availability helps solving many real-world scenarios. These scenarios present the different fragments of the software development process. Nowadays, software vulnerability is a priority because it is the root of many system failures. For identifying software vulnerability types, binary function analysis is necessary before implementing and designing any application. The technical analysis obtains insightful information of the program, which analyzes code and other components of the application within the context of the platform on which was built. Furthermore, the binary analysis solution enables organizations to inspect binary code at the machine level without any involvement from the vendor.

Numerous static code analysis tools identify the pattern in code and detect the potential security threats and issues in the quality of code. The analyzer can choose to automate the code analysis or manual analysis; either of them still enhances the culture of creating quality code. It is imperative to detect vulnerabilities before they ever make it into a finished application and to keep the critical data safe. The hybrid binary static analysis is a combination of the static and dynamic analysis techniques, and it provides better detection of errors and security vulnerabilities. It is the analyzer's responsibility to a device which methods use for the binary analysis, and there are huge advantages of the hybrid source and binary static analysis, as described below:

**Learn from comparing the same vulnerability:** Once the compiler has optimized the source code, the resulting binary code can reveal a different view of the detected potential attack. The process helps the analyzer understand the error to prevent the attack.

**Detect injected modified binaries and insider attacks:** A program's source is not its final state. The binary analysis exposes unwanted changes in the final executable. The analyzer can figure out the code injected into executables and download payloads to analyze for defects and vulnerabilities. Moreover, the malicious code added by inside attackers might be revealed before shipping to customers.

**Know the libraries and other binaries without source code:** The analyzer can analyze standard libraries and any other third-party libraries in an executable. The program might change the library version, or the pattern builds for some modules while compiling, beyond the developer's knowledge.

### 2.2 Binary Code Similarity

The analysis of binary function similarities leverages the software review to a new level; notably, it uncovers potential security holes. Without a source code available, it is essential to understand elements of the implementation, such as function data types, operations, etc., and how they might impact the execution flow. Moreover, comparing binary functions without source code availability will help solve many real-world scenarios. Binary function similarity compares the differences between binary parts. The analysis process addresses our first objective with two key components: CFG and DFG. Each of them is equally important for distinguishing parts identical through the programs. Also, the main characteristic of binary code similarity is a crucial factor that specifies the type such as identical, equivalent, and similar of the comparison [27]—the granularity of pieces of binary code being compared such as instructions, basic block, functions. The important thing is the number of input pieces that include one-to-one, one-to-many, and many-to-many [27]. The details of each character are discussed below:

**Type of comparison:** Firstly, two pieces of binary code will be identical if they have the same syntax. Also, when the hash is the same, the pieces are similar. Secondly, they are equivalent if they have the same semantic; for example, they indicate the same functionality represents each piece. Indeed, the process determines the binary code equivalence is excessive because it only can perform for the small pieces of code. Lastly, the two parts of binary code

are considered similar if their syntax, structure, or semantics are a friendly representation. The syntactic similarity approaches are the cheapest to compute, but they are the least robust [27]. Thus, the graph structure is considered the best method to discover the similarity. For instance, the structural similarity is comparing the graph representations of binary code that syntactic and semantic produce, such as CFG and DFG similarity. The essential element is that the control flow captures semantics and data flow of binary code, which helps to compare effectively.

**Granularities comparison:** The general granularities are basic blocks, function instructions, and the whole program. The approach determines the fraction of identical functions between both programs to compare whether two programs are matched. However, the granularity is not enough to assume the exact similarities because the programs might have compiled in different compilers in the various machine architectures. On the other hand, the granularity comparison needs to select the convenient element to conclude the binary code similarity.

**Several inputs comparison:** Comparing two or more binary code pieces indicates the scope of the binary code input. The intent of one-to-one is to take a source and a target of binary code and put them in a standard order to find the different levels. Specifically, they differentiate two consecutive or close versions for the same program to identify what was modified from source to target performance. The one-to-many approach take a source to many targets and obtains a similar or dissimilar binary code list. The many-to-many approach does not distinguish between source and target pieces. All the inputs are considered equal and comparable to each other. The approach is typically used to perform binary code clustering [27]—the output of similar pieces of binary code group as the clusters.

This paper mainly focuses on the comparison granularity of binary code, which is primary on binary function. Also, the comparison identifies the binary function similarity by applying the similar basic block, CFGs, and DFG.

# 3 Related Work

This section summarizes common approaches to finding binary function similarities. The processes of extracting features from binary functions are different for each study. The common methodologies are statics analysis, dynamic analysis, and machine learning, and they are used to develop the proposed prototype and system. Table 1 below summarizes the top twenty research articles.

| Prototype/Tool | Techniques and Approach Characteristics | | | | |
|---|---|---|---|---|---|
| | **Static Analysis** | **Dynamic Analysis** | **Semantic Similarity** | **Syntactic Similarity** | **Structural Similarity** |
| **JIMA [1]** | ✓ | × | ✓ | × | × |
| **Esh [2]** | ✓ | × | ✓ | × | × |
| **αdiff [3]** | ✓ | × | × | × | × |
| **Bindiff [4]** | ✓ | × | ✓ | ✓ | × |
| **discovRE[5]** | ✓ | × | × | × | ✓ |
| **MalwareDec[6]** | ✓ | ✓ | × | × | ✓ |
| **BLEX [7]** | ✓ | ✓ | ✓ | × | × |
| **BinGo [8]** | ✓ | × | ✓ | × | × |
| **Asm2Vec [9]** | ✓ | × | × | × | × |
| **BinHunt [10]** | ✓ | × | ✓ | × | ✓ |
| **iBinHunt [11]** | ✓ | ✓ | ✓ | × | × |
| **SAFE [12]** | ✓ | × | × | × | × |
| **VulSeeker s [13]** | ✓ | × | × | × | × |
| **INNEREYE [14]** | ✓ | × | × | × | × |
| **Gemini [15]** | ✓ | × | × | × | ✓ |
| **BinSign [16]** | ✓ | × | ✓ | × | × |
| **TRACY [17]** | ✓ | ✓ | ✓ | ✓ | × |
| **BINMATCH [18]** | ✓ | × | ✓ | × | × |
| **Kam1n0 [19]** | ✓ | × | ✓ | × | ✓ |
| **XMATCH [20]** | ✓ | × | ✓ | × | × |

**Techniques:** Static Analysis and Dynamic Analysis

**Approach Characteristics:** Semantic, Syntactic and Structural Similarities are approach.

✓ - Applied × - Not Applied

### 3.1 Cross Compiler and Architecture

In the statistical approach for measuring the similarity between two procedures, David et al. [2] created a tool called Esh to address the problem of finding a similar procedure in stripped binaries. The idea was to decompose applications into smaller comparable fragments and functions, define semantic similarity, and use statistical reasoning to isolate pieces to determine similarity. For procedure decomposition, they used standard CFGs and then developed a stand required to compute a particular variable's values in the instruction set. By providing the format definitions, which are the semantics of strand similarity, the design is used to calculate strand similarity using a program verifier. Furthermore, they proposed a quantitative similarity score to find a noise threshold [2] that transforms the quantitative method into a binary classifier by marking all pairs of procedures with a score above the hall as a match and the rest as a nonmatch. They estimated classifiers with the receiver operating characteristics (ROC) and the concentrated receiver operating characteristics (CROC). They then used Esh performance as a semantic tool to show the naïve use of programs verified to be infeasible.

With the same goal, INNEREYE [14] is a system protocol that learns from the successful Natural Language Processing (NLP) field to solve the cross-architecture binary code-similarity comparison problem. The approach regarded instructions as words and basic blocks as sentences. The purpose of the approach is to compare the meanings of sentences in different natural languages in the cross-architecture code-similarity comparison [14]. The design was a precise and efficient cross-(assembly)lingual basic block embedding model. It utilized word embedding and Long Short-Term Memory (LSTM) [3], which are frequently used in Neural Machine Translation (NMT) [14]. Moreover, INNEREYE successfully demonstrated that it is promising to approach binary analysis from language processing angles by adapting methodologies, ideas, and NLP techniques.

### 3.2 Deep Neural Network

Liu et al. [3] provided the cross-version binary code similarity detection problem. They analyzed two binaries from the same source code and focused on function matching, similarity scores, and difference identification [3]. They focused on the neural network to estimate the binary function semantic and raw bytes and define the connection between external modules imported in function

prototypes. They extracted those features classified as intra-function, inter-function, and inter-module [3]. Intra-function semantic features measured the distance between two similar functions (embedding) using a convolutional neural network (CNN). They extracted these features from the raw bytes of binary functions using the convolutional neural network (CNN). Inter-function semantic features identified an interactive relationship between tasks in the same binary and are represented by the CGs. Also, they extracted the in-degree and out-degree of a node in the CG as its inter-function feature. An inter-module semantic feature is a function that invokes a set of imported functions that also mean the inter-module feature. They took the intersection of two binaries from the imported functions and calculated the distance between the imported functions' binaries.

Gao et al. [13] proposed a similar approach with a neural network VulSeeker, a semantic learning-based vulnerability seeker for cross-platform binary. VulSeeker resulted in a higher accuracy and efficiency through the labeled semantic flow graph (LSFG) construction and semantic-aware deep neural network. VulSeeker determined whether the target binary contains a function similar to known vulnerabilities or not. In detail, LSFG has both a CFG and a DFG. This model is used to transform numerical features of basic blocks within the function into function semantics [13]. There are four major components of VulSeeker design: LSFG construction, block feature extraction, function semantics generation, and similarity calculation. Remarkably, VulSeeker has proposed eight types of basic block level features: number of stack operation instructions, number of arithmetic instructions, number of logical instructions, number of comparative instructions, number of library function calls, number of unconditional jump instructions, number of conditional jump instructions, and number of generic instructions.

Massarelli et al. [12] proposed Self-Attentive Function Embeddings (SAFE), a novel architecture for the embedding of functions based on a self-attentive neural network. SAFE has used an embedding model structured in two phases: the Assembly Instructions Embedding component, which transforms a sequence of assembly instructions in a series of vectors [12], and a self-attentive Neural Network, which transforms a sequence of vectors in a single embedding vector [12]. In the first phase, the main character is word2vec mode, which maps each instruction and trains the instruction embedding model. For the second phase, the self-attentive network has been proposed in different research methods that have applied bi-directional recurrent neural networks. The i2v model represents the assembly instruction embedding that has used assembly instruction as tokens, such as a single token that includes both the instruction mnemonic and the operands. SAFE is an architecture for computing

embeddings of functions in the cross-platform case that does not debug symbols and does need Control Flow Graph (CFG). Therefore, the approach has led to a considerable speed advantage. Furthermore, SAFE created a thousand embeddings per second on a mid-CTOS GPU [12], and it can factor the process of disassembling time for our end-to-end system (from binary file to function embedding) and process more than 100 functions per second [12].

On the same stage of the neural network-based approach, Xu et al. [15] proposed Gemini, which computes the embedding and isolates a numeric vector based on the control flow graph of each binary function. Then the similarity detection measured the distance between the embeddings for two functions. The key idea of this approach is code similarity embedding. It used the Siamese network to train the embedding network, a pre-trained model that can generate embedding for similarity detection. Firstly, the prototype designed the embedding mapping $\phi$ as a neural network. Instead of doing well on a particular predictive task, $\phi$ was trained to differentiate the similarity between two input attributed control flow graphs (ACFG). They mainly designed a Siamese architecture and embedded the graph embedding network Structure2vec [23]. The model requires a large amount of data on the ground truth; Gemini [15] implemented a task-independent pre-training model and a task-specific re-training model.

### 3.3 Syntactic and Semantic Static Analysis

Eschweiler et al. [5] addressed a new approach for searching a similar function in binary code efficiently. The approach started from a known bug and then searched for the same vulnerability in other binaries by identifying identical binaries' features. They emphasized the similarity between functions based on the corresponding CFG structure and employed an efficient pre-filter based on numeric attributes to quickly identify a small set of candidate functions [5]. The implementation is recognized as a set of code features divided into structural and numeric features. The structural features denote the structure of control flow inside a binary represented by its CFGs, while the features represent meta-information about binary function. The structural similarity presented the similarity metric based on the maximum common subgraph isomorphism (MCS) [5]. With the actual block distance, the authors created the parameter set for function distance. They included the number of arithmetic instructions, number of calls, number of instructions, number of logic instructions, number of transfer instructions, number of string constants, and numeric constants.

In another approach, Chandramohan et al. [8] proposed BinGo, which is a binary search engine that supports various architectures and operating systems. Technically, BinGo implemented a selective

inlining algorithm for capturing the complete semantic of the binary functions [8]. Inlining is a compiler technique to optimize the binaries for maximum speed and minimum size. The authors introduced the architecture and OS neutral function filtering process to narrow down the target function search space. Moreover, they generated function models agnostic to the underlying program structure via the length-variant partial traces [8].

### 3.4 Semantic Dynamic Analysis

With the CFGs for a specific case malware classification, Cesare and Xiang [5] narrowed down the potential sets of CFG by implementing a fixed size k-subgraph and then constructing a feature vector. This optimized efficiency when comparing data in two CFGs significantly and had not been used in previous studies. They demonstrated the set of control flow graphs for the identification of malware [5]. Also, they identified a set of code features divided into two categories: structural features and numeric features. The structural features denote the structure of control flow inside a binary represented by its CFG, while the numeric features represent meta-information about binary function. Significantly, the structural similarity presented the maximum common subgraph isomorphism for the similarity metric. The attributes of the metric sets were based on basic block distance that calculated the weighted sum over their respective feature [5].

Egle et al. [6] proposed a blanket execution to address the matching function binary. The blanket execution was designed to support semantic feature extraction using dynamic analysis. The implementation ensured every instruction was executed without violating the brand instruction semantics. There are seven binary code semantics extractors in the blanket execution. They are used to approximate the semantic of a function [6], and the algorithm implemented in a system called BLEX [6] operates on two inputs. The first input is a program binary, and the second input is an execution environment. The approach answered two questions: how the function is executed for feature collection, and what features are collected to be useful for semantic similarity comparison. Furthermore, BLEX evaluated the state-of-the-art binary comparison tool BinDiff [3].

### 3.5 Summary

Researchers proposed the theorem prover to compare the semantic similarities of binary functions. Still, their techniques are not suitable for large-scale code bases due to the high performance and complexity, the extensive computational resources, and their time-consuming nature. Additionally, the semantic data training used an existing neural network model whose build and original dataset the research has not elaborated. The dynamic programming algorithm was used to calculate the most

significant common subsequence between two sequences. Still, we need to give a linear independent path and find the track with the highest score in multiple directions of another genetic network.

We also learned that these approaches' techniques incorporated into the tool for analyzing the binary code. The binary analysis techniques share a common weakness of complexity in determining semantic and syntactic analysis. Likewise, the dynamic analysis did not fully prove the possibility that the tool or prototype would recover the binary code while inspecting the program. The dataset is not adequate for the machine learning analysis to reproduce the experiment or support the research's topics.

# 4 Approach

This section describes the challenges of binary static analysis and introduces the set of metrics that identify the binary function detail in various aspects. Each aspect is evaluated based on the number of instructions, the basic block level features, the CFG, and the DFG. We present a set of referenced metrics from the research, VulSeeker [12] and Bindiff [3], in Table IV-1 and IV-2. The metrics refer to JIMA [1] for characterizing the semantic likeness among parts of binary function in Table IV-3. Finally, Table IV-3 represents Tracelet features. All the metrics rely on static analysis.

## 4.1 Challenges

There are numerous challenges for doing the binary analysis without the source codes: binary optimization problem, complex compilers, and different calling convention analysis. We assume the target architecture of binary code to be the mainstream architecture x86-32 and x86-64 bit on Linux. More detailed information about the challenges of binary static analysis is provided below.

**No source codes:** Only analyzing an executable is difficult because the behavior of a function and the blueprint of a program are ambiguous. The typical binary level static includes disassembly, CFG, and DFG construction, but there is no standard structure for those approaches. Also, it is time-consuming because the analyzers manually conduct the set of procedures for gathering and validating only the beneficial granularities of binary function.

**A compiler optimization**: A compiler might generate imprecise code internally. Each compiler has different optimizations that could change the source code and the module's usage for compiling a program. It is even more complicated for the programming languages used to write a program because they do not provide explicit constructs. Thus, the compliant compilers do not have the exact information to preserve what the programmers purport for development. For instance, in C programming, the function's signature contains types of information about the process. During compilation, the information gets diluted or lost without notifying the programmers. Additionally, the compiler also determines which calling convention is used for all functions, but some compilers allow developers to set specific calling conventions on per-functions. Thus, the analyzers might have a difficult time spotting a calling function at the disassembly level. More details about calling conventions for Unix system can be found in the section bellow.

**The state-of-the-art disassemblers:** A static disassembler takes an executable file and processes it to disassemble it without running it. There are three static approaches: the linear sweep,

the recursive traversal, and the combined linear sweep with recursive traversal [26]. Each process has its difficulties in extracting features from binary functions. The linear sweep starts at the entry point of function and disassembles the instructions and the associated registers in the code section until it reaches the endpoint. The Unix objdump is used to adopt this approach. Still, it might mistreat data as code and produce an incorrect instruction sequence due to the type of architectures and the compliant compilers. The recursive traversal disassembles the instructions into the CFG of the code. Still, it is not easy to construct the full graph including indirect, unconditional jump, and call instructions because they occupy a tight spot on the content of register or memory location that is statically anonymous. If combined appropriately, linear sweep and recursive traversal can be beneficial, but the combination also carries deviant case analysis. The obfuscation of binary code effectively disassembles because it cannot deal with indirect jump or call instructions. Thus, the CFG created by the disassembler has not covered the entirety of the program, and it can cause the misrepresentation of analysis. Throughout the paper, we focus on the linear sweep with recursive traversal for the research. Other researchers have proposed standard techniques that modified those approaches, but they are beyond the scope of this paper.

**Calling conventions for Unix system calls:** Calling conventions act as a contract between subroutines and a function specifies a particular architecture and operating system interface. They describe how the arguments are passed to instructions, which registers are used to store values, how they may allocate local and global variables, and so forth. In x86 calling conventions, a function called another function that is a *caller* while a callee is a function called, and the currently executing function is a *callee*, but not a caller. The x86-32 bit has numerous calling prior to 64-bit processors. The following table illustrates the most common 32-bit calling conventions [27].

| Convention | Stack cleanup | Parameter passing |
|---|---|---|
| cdel | Caller | Pushes parameters on the stack in reverse order (right to left). |
| fastcall | Callee | The first two params store in registers, and the rest store on the stack with a reverse order. |
| stdcall | Callee | Pushes parameters on the stack in reverse order. |
| thiscall | Callee | The first param stores in ECX, and the rest store on the stack with a reverse order. |

The 32-bit architecture has 32 registers, while x64 extends x86's eight general-purpose registers to be 64-bit. There is a difference in the implementation of calling conventions, and it is also a challenge for the analyzers. A solid background is required to distinguish the type of registers used for system calls, the number of parameters on the stack, and the application binary interface (ABI) usage. The table below represents a comparison of the two architectures' significant calling conventions.

| **x86-32 bit** | **x86-64 bit** |
|---|---|
| The registers used for system call are - %ebx, %ecx, %edx, %esi, %edi, %ebp | The registers used for system call are - %rdi, %rsi, %rdx, %r10, %r8 and %r9 |
| Return values are stored in %eax register | Return values are stored in %rax register |
| Push instruction for passing parameters on the stack. If the number of arguments is more than six, then %ebx is needed to contain the memory location. | System-calls are limited to six arguments on the stack. The first six arguments are passed in registers, and the remaining arguments are passed on the stack. |
| In x86-32 parameters are passed on stack. Last parameter is pushed first on to the stack until all parameters are done, and then call instruction is executed. | First the parameters are divided into classes. The class of each parameter determines the pattern in which it is passed to the called function. |
| 32-bit int ABI is usable in 64-bit code. | 64-bit ABI calls cannot be used in 32-bit system. |

### 4.2 Metrics

The metric implies JIMA [1] characteristic analysis, which is comprehensive binary function in multiple phases. These phases are disassembly, exception handler analysis, jump pointer analysis, function detection, missing function detection, and terminal function call chain detection [19]. The process of function detection is a sort of list of possible function addresses. Each function initializes the start address to the end address. We will use JIMA to calculate the type of instructions and recognize program code constructs in assembly level. For instance, we want to discover C code constructs in assembly in the following attributes:

- Global vs Local Variable
- If statement
- Recognizing Loop
- Function call conventions
- Switch statements
- Disassembling Arrays
- Identifying structures
- Linked List data structure.

We utilize JIMA to extract features for each basic block; then we encode the features of each basic block for calculating the number of instructions. Research in the field has used VulSeeker [12], a semantic learning-based vulnerability seeker for cross-platform binary. The VulSeeker approach aims to determine whether the target binary contains a function like known vulnerabilities or not. The metric represents one of the four major components of VulSeeker design, block feature extraction, which can be extracted with a minor change under various implementation platforms. The basic block features are significant for comparing the binary function because they form the vertices and nodes in a CFG and DFG. The number of different instructions use to evaluate the binary function similarities.

| **Basic -block level features used by VulSeeker** | | |
|---|---|---|
| **Feature Name** | **Number of Features** | **Type of Feature Example** |
| Stack operation instructions | | Push, pop |
| Arithmetic instructions | | Add, sub |
| Logical instructions | | and, or |
| Comparative instructions | | test |
| Library function calls | | call printf |
| Unconditional jump instructions | | jmp |
| Conditional jump instructions | | jne, jb |
| Generic instructions | | mov, lea |

We take advantage of using the above basic block features to analyze and verify the similarity between two binary functions. We learn if two functions have the same instructions and number of instructions, they are equivalent and similar. Moreover, adding and swapping directions will lead the researcher to explore many exciting aspects.

The BinDiff [3] relies on the comparison of the components extracted from a binary's Call Graph (CGs) and Control Flow Graphs (CFGs). All these data identify the mapping of one binary function to another using each task's semantic features. Table 2 shows the abstract view of CG's and CFG's structural characteristics. The metric is used to study the syntactic and semantic differences and perform function matching and reduce graph isomorphism problems [20] between CGs and CFGs. The metric attributes are necessary for any research on binary function similarity. However, many questions about the use of distance metrics of Bindiff remain unanswered.

BinSign [16] presented a Tracelet features metric describing data constants, functionality tags, and Tracelet info such as the number of instructions, operands, imported functions, etc. The composition of CFG instructions captured the number of basic blocks and types of instructions in each block. Importantly, all features were extracted at two levels: global and tracelet features. Features described each basic block that were combined into tracelet features. Moreover, the CFG constructure was captured in the function’s fingerprint through the Tracelet features.

Additionally, the arrangement of CFG instructions stored the number of basic blocks and types of instructions in each block. All local, system, and API calls were categorized according to the system's execution outcome. We will apply Tracelet information that indicates the number of instructions, operands, code refers, function calls, and imported functions. The size of those features can be extracted at the disassembly level, but we might not capture all the function granularities.

| **Data Constants** | #Constants | | #Strings | |
|---|---|---|---|---|
| **Functionality Tags** | # API Tags | #Library Tags | #Mnemonic Groups | |
| **Tracelet Info** | #Instruction | #Operands | #Function Calls | #Imported Functions |

**BINSIGN – Tracelet Features Metric**

We extract the granularities of binary function on the .text section because it is where the main code of the program resides. Thus, it will frequently be the crucial focus of the binary analysis or reverse engineering effort. We also strip off the debugging symbols to reduce the size and give better performance than a non-stripped binary. Again, we intend to find the binary function similarities; thus, the bug search in the programs is beyond the scope of this paper.

The approach divides into three modules: assembly extraction, basic block level similarities, and metrics similarities. The assembly extraction conducts the programing code into the assembly code, which is basically binary code written in a form readable to human. By using JIMA, we can collect an object dump file, which represents a full assembly code of a program; however, we are more interested in analyzing jil file. Next, the basic block level similarities identify the binary code similarities between each binary function similarities, which represent the instruction types, assigned registers, and loaded registers. Lastly, the metrics similarities conclude the function similarities by comparing the features in various characteristics. Hypothetically, the list of metrics isolates from numerous field research studies in the field and logical ideas to analyze the binary function; therefore, we will run the sequence of experiments to validate those metrics and evaluate our approach.

# 5 Implementation

This section describes the experimental setup to study the binary function similarities and the proposed algorithm used to calculate the similarity scores between two binary functions.

## 5.1 Environment Setup

We use JIMA [1] to run the test for the metrics under the server:

- OS: Xubuntu kernel 5.8.0-53-generic x86_64 bits: 64
- CPU: 8 Cores, 64GB RAM Speed 2397 MHz, and 1Tb Hard drive.

We install Python 3.8.5 to run JIMA and other support scripts.

## 5.2 Algorithms

**Algorithm 1:** To predict the accuracy between the two functions, we determine the minimum number of subsequences of source that form a target. The algorithms are used to figure out the similarity score. Initially, the experiment follows the JIMA to generate the Jil file that records all the granularity of binary functions at the disassembly level. We pick a binary function in Program 1, and we decompose all the potential features into smaller comparable fragments. The fragments distribute into numerous metrics that represent basic block features count, DFG, CFG, and the C code constructs in the assembly. Each metric represents an object that contains all the attributes of the metric as the fields. We store fragments in a list of objects as the source.

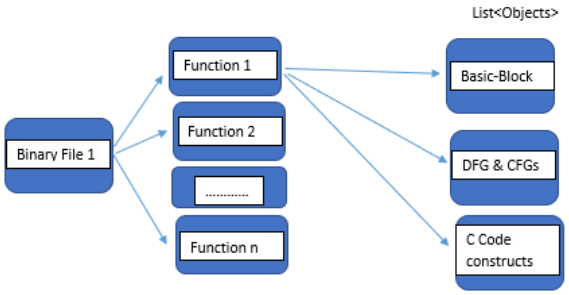


**Figure 5.1.a – Decomposing a program into the list of objects.**

In Program 2, we do the same approach; however, we conduct for all the functions and store them into a list of objects as the target. To predict the accuracy between two functions, we will discover the minimum number of subsequences of source that form a target.

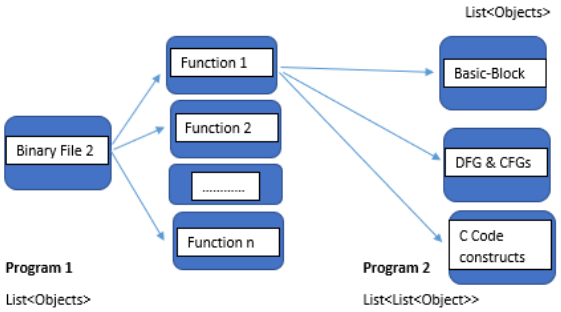


**Figure 5.1.b - Comparing an object in program 1 to the list of objects in program 2.**

**Algorithm 2:** We apply a reaching definition (RD) analysis that statically determines which definitions may reach a given point in code. The reaching definition is a type of data-flow analysis in which the data-flow confluence used is set union and the analysis is forward flow. The data-flow analysis is the process of collecting information about the way variables are used and defined in the program.

For each program point, we are interested in which assignment may have been made and not overwritten and when program execution reaches this point along some path. The data-flow equations used for a given basic block S in reaching definitions are:

$$REACH_{in}[S] = \bigcup_{p \in pred[S]} REACH_{out}[p]$$

$$REACH_{out}[S] = GEN[S] \cup (REACH_{in}[S] - KILL[S])$$

The set of reaching definitions going into S are all of the reaching definitions from S's predecessor, pred[S'] pred[S], which consists of all the basic blocks that come before S in the control-flow graph. The reaching definitions coming out of S are all reaching definitions of its predecessors minus those reaching definitions whose variable is killed by S plus any new definitions generated within S.

In this research, we assume a basic block is sized per address in a function that contains an assigned address, an instruction, registers, and the address of memory allocation. We make a pair "address, var" for the address where var represents register. For the dataflow, we discover which registers are sources of information for destination, which influences its value. There are two methods of implementation: direct flow and indirect flow in terms of workflow.

- Direct flow
    - We kill all data flows to the destination register.
    - We generate all sources from the instruction to the destination register. If those sources have data flows to them, we also add all of those to the destination register's flows.
- Indirect flow
    - We include all direct flows and keep track of what sets/unsets FLAGS register.
    - We add the FLAGS register to the data flow out of a conditional jump, but we also keep track of the target address of the jump register.
    - We need to use the FLAGS register as source if the FLAGS register is in the dataflow analysis.

RDs analysis determines the dataflow point, which is an important factor in identifying the similarity between two binary functions. We isolate the analysis metric with the column of a label, kill, and gen reaching definition. Then, we pick a basic block, which represents a binary function in a CFG, and extract the information for the entry points to exit points. We primarily target the operation instructions to create a direct data flow through a binary function. Thus, the label defines by the instruction address, each killed pair contains a source register to all the destination registers through per instruction, and each generated pair only contains a source and destination register at the instruction.

To coordinate JIMA and RDs, we generate a .jil file of a program and extract each function granularity into a dictionary list, where the key is the entry address of function, and the values contain a list of instruction details. The address of instruction is the key, and the register type and assignment source and destination are the values. We represent each function per a dictionary list use to find the similarity between functions. However, there are some characteristics that make the comparison effective such as the different compilers or usage of library versions. Thus, we consider the RD analysis to map the similar register assignment based on the operation instructions of each function.

### 5.3 Test Cases

We use the test cases to support the objectives and demonstrate JIMA tool ability to process the binary static and data flow analysis. Also, the test suite contains executable files strip and removes debug and symbol information. For the test suites, we analyze 516 stripped executable binaries, which are the common utility packages for the architecture x86-32 bit on Linux: binutils, coreutils, and findutils. Those packages were compiled with gcc in various optimization options O0-O3. We isolate the sequence of experiments to address the following study cases:

(1) If the compiler version is known, we look for distinguishing patterns in the output of the various compilers. We might be able to say if n executable was more likely produced with compiler A than with compiler B. We isolate the set of metrics and fill out the interesting characters we can extract and collect through semantic analysis methodologies.

(1.a) We experience two programs that have the same behavior but different implementation. They are compiled in the same architecture, Intel x86 32-bit, optimization o1, and compiler Gcc version 9.3.0: 1) Binary tree linear search (BTLS) program and 2) Binary tree search (BTS) program.

(1.b) Similar to Test Case 1.a, but we use Gcc version 10.2.0 to compile BTLS and BTS program, and then we compare them.

(1.c) We pick two programs that have the same blueprint, excepting different Gcc compiler version in the test suite.

(2) The two compilers, Gcc and Clang, are different. They may produce different output for the same input, and they may do so in some systematic way. One compiler likes to use a certain register to store something whenever it compiles a function call; another uses a unique pattern of instructions when storing a value in memory. That could result in some byte pattern being more common in the output of one compiler than in the output of another.

(2.a) We experience two programs that have the same source code but use different compilers. They are compiled in the same architecture, Intel x86 32-bit, and optimization o1: Binary tree search (BTS) program compiled with Gcc 9.3.0 and Clang 10.0.0.

(2.b) We pick a program compiled with Gcc and another compiled with Clang in the test suite.

For Test Cases 1.a, 1.b, and 2.a, we wrote two programs in C programming languages, then compiled them with referenced compilers to create binary executables files. We stripped away debug information and used JIMA to collect the function granularities for the static analysis. Therefore, the known source code of those test cases is used to measure the accuracy of the paper's approach. In Test Case 1.c and 2.c, we have the ground truth along with the program in the test suite.

Throughout the test cases, we expect to adjust the datasets to verify and increase the approach's accuracy in order to leverage the experiment to compare multiple functions between big programs. Also, we declare new features in JIMA to make the result more robust. For instance, we create a couple of scripts to extend JIMA features for extracting the assigned and load registers in each instruction. We strip off the instruction addresses and those registers to pass into the form of pairing registers and label that follow the reaching definition analysis.

# 6 Experiments

This section discusses the experiment and results for the metrics set and addresses some challenges that the study resolves. The main task is to figure out how to detect the binary function similarity between two executable files.

The analysis section demonstrates the process of our approach that includes the program source code, assembly code, and fulfilled data for metrics. We collect the data and present them in tables and figures for each test case. Ideally, we imply the outcome of the experiment to test if we have enough data to satisfy the objectives: (1) similarity between executable files with same compiler version and program has different source code version, and (2) similarity between executable files with different compiler and program has same source code version.

The discussion section recalls our research goals that have been tested by the achieved result in the analysis section. We provide a comprehensive study view, identify the challenges, and navigate the future direction research.

## 6.1 Analysis

**Basic block features:** The significant similarity analysis at the basic block level was executed for similar function pairs. Thus, we calculated the number of times each instruction represented in a binary function, which fills out the Vulseeker metric for basic-block level features. We applied this calculation for Test Case 1.a and b; then we used the outcome as an input to the Algorithm 1. Initially, we selected two main functions, linear search and binary search from the BTLS and BTS program. We elaborated the analysis process to obtain the outcome for each test case by comparing the source code, assembly code, and the number of each basic block feature.

```
/*Linear Search Tree*/

int linearSearch(int arr[], int n, int x) {

    int i;

    for (i = 0; i < n; i++)

        if (arr[i] == x)

            return i;

    return -1;

}
```

```
/*Binary Search Tree*/

int binarySearch(int arr[], int l, int r, int x)

{

    if (r >= l) {

        int mid = l + (r - l) / 2;

        if (arr[mid] == x)

            return mid;

        if (arr[mid] > x)

            return binarySearch(arr, l, mid - 1, x);

        return binarySearch(arr, mid + 1, r, x);

    }
```

**Figure 6-1. Comparing source code for Linear Search and Binary Search function.**

We demonstrated the comparison between the number of each instruction at the assembly level and the counted instructions JIMA extracted in the following Figure 6-1 for the linear search function of the BTLS program. JIMA captured all the important instructions to support the idea of identifying the similarities between two functions. We mapped one-to-one functions to understand whether they were created identically or uniquely. We collected the basic features of the linear search function to fill out VulSeeker metric. Similarly, we counted the instruction for the binary search function of the BTS program to isolate another VulSeeker metric. Particularly, we analyzed four functions distributed into two groups of gcc 9.3.0 and 10.2.0 that were used to compile linear search and binary search program, which are indicated in the column name in Table 2.

```
000011ed <linearSearch>:
   11ed:   endbr32
   11f1:   pushl  %ebp
   11f2:   movl   %esp,%ebp
   11f4:   subl   $0x10,%esp
   11f7:   calll  12f9 <__x86.get_pc_thunk.ax>
   11fc:   addl   $0x2dd8,%eax
   1201:   movl   $0x0,-0x4(%ebp)
   1208:   jmp    1229 <linearSearch+0x3c>
   120a:   movl   -0x4(%ebp),%eax
   120d:   leal   0x0(,%eax,4),%edx
   1214:   movl   0x8(%ebp),%eax
   1217:   addl   %edx,%eax
   1219:   movl   (%eax),%eax
   121b:   cmpl   %eax,0x10(%ebp)
   121e:   jne    1225 <linearSearch+0x38>
   1220:   movl   -0x4(%ebp),%eax
   1223:   jmp    1236 <linearSearch+0x49>
   1225:   addl   $0x1,-0x4(%ebp)
   1229:   movl   -0x4(%ebp),%eax
   122c:   cmpl   0xc(%ebp),%eax
   122f:   jl     120a <linearSearch+0x1d>
   1231:   movl   $0xffffffff,%eax
   1236:   leavel
   1237:   retl
```

```
000011ed <binarySearch>:
   11ed:   endbr32
   11f1:   pushl  %ebp
   11f2:   movl   %esp,%ebp
   11f4:   subl   $0x18,%esp
   11f7:   calll  1355 <__x86.get_pc_thunk.ax>
   11fc:   addl   $0x2dd8,%eax
   1201:   movl   0x10(%ebp),%eax
   1204:   cmpl   0xc(%ebp),%eax
   1207:   jl     1287 <binarySearch+0x9a>
   1209:   movl   0x10(%ebp),%eax
   120c:   subl   0xc(%ebp),%eax
   120f:   movl   %eax,%edx
   1211:   shrl   $0x1f,%edx
   1214:   addl   %edx,%eax
   1216:   sarl   %eax
   1218:   movl   %eax,%edx
   121a:   movl   0xc(%ebp),%eax
   121d:   addl   %edx,%eax
   121f:   movl   %eax,-0xc(%ebp)
   1222:   movl   -0xc(%ebp),%eax
   1225:   leal   0x0(,%eax,4),%edx
   122c:   movl   0x8(%ebp),%eax
   122f:   addl   %edx,%eax
   1231:   movl   (%eax),%eax
   1233:   cmpl   %eax,0x14(%ebp)
   1236:   jne    123d <binarySearch+0x50>
   1238:   movl   -0xc(%ebp),%eax
   123b:   jmp    128c <binarySearch+0x9f>
   123d:   movl   -0xc(%ebp),%eax
   1240:   leal   0x0(,%eax,4),%edx
   1247:   movl   0x8(%ebp),%eax
   124a:   addl   %edx,%eax
   124c:   movl   (%eax),%eax
   124e:   cmpl   %eax,0x14(%ebp)
   1251:   jge    126d <binarySearch+0x80>
   1253:   movl   -0xc(%ebp),%eax
   1256:   subl   $0x1,%eax
   1259:   pushl  0x14(%ebp)
   125c:   pushl  %eax
   125d:   pushl  0xc(%ebp)
   1260:   pushl  0x8(%ebp)
   1263:   calll  11ed <binarySearch>
   1268:   addl   $0x10,%esp
   126b:   jmp    128c <binarySearch+0x9f>
   126d:   movl   -0xc(%ebp),%eax
   1270:   addl   $0x1,%eax
   1273:   pushl  0x14(%ebp)
   1276:   pushl  0x10(%ebp)
   1279:   pushl  %eax
   127a:   pushl  0x8(%ebp)
   127d:   calll  11ed <binarySearch>
   1282:   addl   $0x10,%esp
   1285:   jmp    128c <binarySearch+0x9f>
   1287:   movl   $0xffffffff,%eax
   128c:   leavel
   128d:   ret
```

**Figure 6-2. Assembly code for Linear Search and Binary Search functions**

| Features Name | Instructions | Linear Search Gcc 9.3.0 | Linear Search Gcc 10.2.0 | Binary Search Gcc 9.3.0 | Binary Search Gcc 10.2.0 |
|---|---|---|---|---|---|
| No. of stack operation | Push, pop | 1 | 2 | 9 | 14 |
| No. of arithmetic | Add, sub | 4 | 1 | 11 | 5 |
| No. of logical | And, or | 0 | 0 | 0 | 0 |
| No. of comparative | test | 0 | 1 | 0 | 0 |
| No. of library function calls | Call printf | 0 | 0 | 0 | 2 |
| No. of unconditional jump | Jmp, | 4 | 4 | 6 | 5 |
| No. of conditional jump | Jne, jb | 2 | 3 | 3 | 3 |
| No. of generic | Mov, lea | 9 | 6 | 19 | 8 |

**Table 2. Counting No. of basic block features for Gcc 9.3.0 and 10.2.0.**

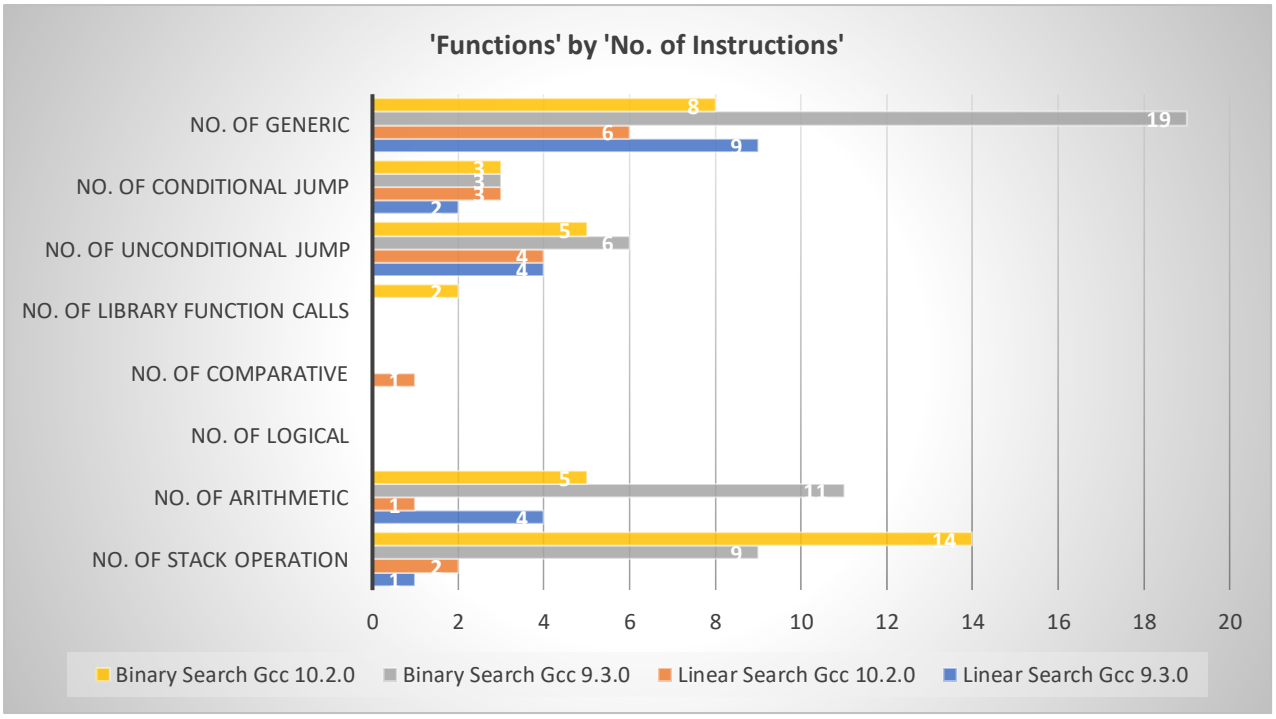


**Figure 6-3. Differing Number of basic block features for Linear Search and Binary Search**

Table 2 shows the difference between the number of instructions represented. The basic block level features used by VulSeeker indicate the linear search function of the BTLS program is not like the binary search function of the BTS program. As shown in Figure 6-2, we learned the number of condition jump instructions was same for three functions: linear search gcc 10.2.0, binary search gcc 9.3.0, and 10.2.0; likewise, the unconditional jump instructions were equal for the linear searches 9.3.0 and 10.2.0. The highest number of generic instructions fell into the binary search gcc 9.3.0.

For Test Case 2.a, we obtained the data for the differing number of basic blocks on compiler Gcc 9.3.0 and Clang 10.0.0, as illustrated in Table 3 below.

| Features Name | Instructions | Linear Search Gcc 9.3.0 | Linear Search Clang 10.0.0 | Binary Search Gcc 9.3.0 | Binary Search Clang 10.0.0 |
|---|---|---|---|---|---|
| No. of stack operation | Push, pop | 1 | 4 | 9 | 13 |
| No. of arithmetic | Add, sub | 4 | 0 | 11 | 4 |
| No. of logical | And, or | 0 | 0 | 0 | 0 |
| No. of comparative | test | 0 | 1 | 0 | 0 |
| No. of library function calls | Call printf | 0 | 0 | 0 | 1 |
| No. of unconditional jump | Jmp | 4 | 4 | 6 | 4 |
| No. of conditional jump | Jne, jb | 2 | 3 | 3 | 3 |
| No. of generic | Mov, lea | 9 | 5 | 19 | 7 |

**Table 3. Counting No. of basic block features for Gcc 9.3.0 and Clang 10.0.0**

We learned that the different compilers produced different numbers of basic block features. The number of unconditional and conditional jump instructs had singular range. The most dissimilar element was the numbers of arithmetic instructions, such as the clang consumed zero operator to compile linear search function.

Therefore, without source code and binary executable analysis, we cannot make assumptions for two functions that perform the same feature and complied with the same compiler and code optimization that are similar exactly. Specifically, the number of arithmetic instructions of linear search function is less than binary search function.

**Data Flow Analysis:** We tracked values across load/store instructions. Therefore, it is critical to know which definitions reach a certain load and vice versa. This information is provided by reaching definition analysis. We used data flow to gather information about the possible set of values calculated at various instructions that represent in a basic block of CFG. We determined a particular value assigned to a variable or a pointer, which might propagate through a function. To apply reaching definition analysis, we presented data flow for each function by labeling address, generate and kill pair consume source, and destination register. The set of generate and kill pair was used to conduct the entry and exit data flow through a function.

The table below describes the data for Test Cases 1.a and 1.b. Source registers are defined as memory load operations and the destination registers. Table 4 illustrated the pair of destination register and instruction addressed assigned at that register, where the instruction list excludes the operant instruction types.

| **Linear Search Gcc 9.3.0 - o1** | | **Linear Search Gcc 10.2.0 - o1** | |
|---|---|---|---|
| **Des. Register** | **Inst. Address** | **Des. Register** | **Inst. Address** |
| None | 0x000011ed | None | 0x000011ed |
| %ebp | 0x000011f1 | %ebx | 0x000011f1 |
| 12f9 | 0x000011f7 | %edx | 0x000011fe |
| 1229 | 0x00001208 | 121a | 0x00001200 |
| 1229 | 0x00001208 | 121a | 0x00001200 |
| 0x0(,%eax,4) | 0x0000120d | %ebx | 0x00001207 |
| %eax | 0x0000121b | 1218 | 0x0000120a |
| 1225 | 0x0000121e | 1218 | 0x0000120a |
| 1225 | 0x0000121e | %eax | 0x0000120f |
| 1236 | 0x00001223 | 1207 | 0x00001211 |
| 1236 | 0x00001223 | 1207 | 0x00001211 |
| 0xc(%ebp) | 0x0000122c | %ebx | 0x00001218 |
| 120a | 0x0000122f | None | 0x00001219 |
| 120a | 0x0000122f | 1218 | 0x0000121f |
| None | 0x00001237 | 1218 | 0x0000121f |

**Table 4. Destination register flow between Gcc 9.3.0 and 10.2.0 for linear search.**

We learned in the linear search Gcc 9.3.0, the register %ebp represented a function parameter that passed twice in the function at the instruction address 0x000011f1 and 0x0000122c.

### 6.2 Discussion

In practice, the binary static analysis methods share a weakness of difficulty determining whether a change in a binary function is a semantic change or a syntactic change. The process is up to the reverse engineer, who wants to spend time and resources determining that an identified change by one of these methods is not a functional change. In this research, at assembly level, we are interested in extracting the instructions containing a*rgs* registers, and we ignore the comment and call instructions. We compare the number of instructions on each function of each program to determine the similarity between them, but this approach is only valid if both programs compiled the same compiler. Thus, Algorithm 1 is out of scope for multiple compilers in binary function similarity research.

The number of basic block features is insufficient for figuring out the similarity between binary functions when different compilers compile the program. However, those features have a potential avenue, which would be to incorporate context further. We consider some structural approaches in the CG and CFG to match functions, but the analysis process does not always suffice. We believe it is possible to incorporate further another context like local variables, global variables, type of loop, and data reference variables that keep the analysis isolated and ensure that research is conducted properly.

The variety of test cases and methodologies used to evaluate our approach to the absence of source code makes it hard to perform a fair comparison and understand the benefits and limitations. Also, we believe there are great benefits from open datasets for benchmarking the objectives for isolating the similarity of binary functions. There is a need to handle obfuscation from source code to executable, beyond the change of compiler and compiler version. The explicit test cases for each dataset are fundamental for finding the binary function similarities and addressing any objectives for the research direction.

We use data flow analysis to identify function identical behavior. This is the main contribution of our research. Specifically, reaching definition analysis does not depend on the structure or the syntax of the instructions, only the effects of the structure and syntax on the data assigned from source to

destination register. We do not consider this to be a solution to identify the binary function similarity, yet it can be an additional approach to learn about binary function without source code.

Overall, we first identify the comparable binary functions, which are known for the type and version of compiler, the code optimization, and the compiled architecture. We focus on the disassembly code to generate the basic block for each function in a program, then collect the instruction types and reference registers to isolate the various metrics for comparing the similarity.

# 7 Conclusion

Binary function similarity delivers the ability to compare binary code and explore many different aspects of software vulnerability. In the field, there are various approaches to address the binary function. The evaluation of their outcome helps identify directions that future research can take in creating the binary function similarity metrics. The paper summarized different binary functions and code similarity approaches along two dimensions: the methodologies and characteristics evaluation. It discussed the advantages and limitations of different approaches and implementations. With this information, the set of metrics was developed to represent the comparison of binary functions. Eventually, the goal is to measure the accuracy of binary function metrics. The function granularities distinguish into the metrics by arrangement: (1) break down the binary task into the littler identical function, (2) characterize the semantic likeness among parts, and (3) utilize factual thinking to lift section closeness into the comparability between two executable files. Additionally, the set of metrics are continually modifying the best approach. There is also a need to develop a comparison of approaches that handle dataset and compiler options.

# Reference


[1] Jim Alves-Foss and Jia Song. 2019. Function boundary detection in stripped binaries. In Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC' 19).Association for Computing Machinery, New York, NY, USA, 84-96.

[2] Yaniv David, Nimrod Partush, and Eran Yahav "Statistical Similarity of Binaries" June 2016, ACM SIGPLAN.

[3] B. Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihur Piao, and Wei Zou, "alpha Diff: Cross-Version Binary Code Similarity Detection with DNN," 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), Montpellier, France, 2018.

[4] C. Karamitas and A. Kehagias, "Efficient features for function matching between binary executables," 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso.

[5] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padillay "discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code."

[6] S. Cesare and Y. Xiang, "Malware Variant Detection Using Similarity Search over Sets of Control Flow Graphs," 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications, Changsha, 2011

[7] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components," in 23rd USENIX Security Symposium, 2014.

[8] Mahinthan, Chandramohan & Xue, Yinxing & Xu, Zhengzi & Liu, Yang & Cho, Chia & Tan, Hee Beng Kuan. (, 2016). BinGo: cross-architecture cross-OS binary search.

[9] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization," 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2019

[10] D. Gao, M. K. Reiter, and D. Song, BinHunt: Automatically Finding Semantic Differences in Binary Programs. Springer Berlin Heidelberg, 2008.

[11] MING, Jiang; PAN, Meng; and GAO, Debin. iBinHunt: Binary Hunting with Inter-Procedural Control Flow. (, 2012). *Information Security and Cryptology - ICISC 2012: 15th International Conference, Seoul, Korea, November 28-30, 2012: Revised Selected Papers*.

[12] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. 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 2018).

[13] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. 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 2018).

[14] ] F. Zuo, X. Li, Z. Zhang, P. Young, L. Luo, and Q. Zeng, "Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs," in Network and Distributed System Security Symposium, 2019.

[15] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection," in ACM Conference on Computer and Communication Security. ACM, 2017.

[16] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W.-M. W. Hwu. Profile-guided automatic inline expansion for c programs. *Software: Practice and Experience*, 1992.

[17] *Fromkin, Victoria; Rodman, Robert; Hyams, Nina (2014). "Semantics: The Meanings of Language." An Introduction to Language (10th ed.). Boston, MA: Wadsworth, Cengage Learning*

[18] Irfan Ul Haq and Juan Caballero "A Survey of Binary Code Similarity" Sep 2019.

[19] J. Koret. Diaphora: A Free and Open Source Program Diffing Tool [Online]. Available: http://diaphora.re

[20] T. Dullien, R. Rolles," Graph-based comparison of Executable Objects," Proceedings of the Symposium sur la Security des Technologies del'Information et des Communications, 2005.

[21] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Sickinger, and Roopak Shah. 1993. Signature Verification Using A "Siamese" Time Delay Neural Network.

[22] Hanjun Dai, Bo Dai, and Le Song. 2016. Discriminative Embeddings of Latent Variable Models for Structured Data. In International Conference on Machine Learning.

[23] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. In ACM Conference on Computer and Communications Security (CCS'16)

[24] Lina Nouh, Ashkan Rahimian, Djedjiga Mouheb, Mourad Debbabi, Aiman Hanna. BinSign: Fingerprinting Binary Functions to Support Automated Analysis of Code Executables. 32th IFIP International Conference on ICT Systems Security and Privacy Protection (SEC), May 2017, Rome, Italy.

[25] David, Yaniv & Yahav, Eran. (, 2014). Tracelet-Based Code Search in Executables. ACM SIGPLAN Notices.

[26] Zeng, Bin. “Static Analysis on Binary Code.” (2012).