UNIVERSITY OF OULU

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Juho-Pekka Sipola

A FAST AND SCALABLE BINARY SIMILARITY METHOD FOR OPEN SOURCE LIBRARIES

Master's Thesis
Degree Programme in Computer Science and Engineering
January 2023

Sipola J. (2023) A Fast and Scalable Binary Similarity Method for Open Source Libraries. University of Oulu, Degree Programme in Computer Science and Engineering, 63 p.

ABSTRACT

Usage of third party open source software has become more and more popular in the past years, due to the need for faster development cycles and the availability of good quality libraries. Those libraries are integrated as dependencies and often in the form of binary artifacts. This is especially common in embedded software applications. Dependencies, however, can proliferate and also add new attack surfaces to an application due to vulnerabilities in the library code. Hence, the need for binary similarity analysis methods to detect libraries compiled into applications.

Binary similarity detection methods are related to text similarity methods and build upon the research in that area. In this research we focus on fuzzy matching methods, that have been used widely and successfully in text similarity analysis. In particular, we propose using locality sensitive hashing schemes in combination with normalised binary code features. The normalization allows us to apply the similarity comparison across binaries produced by different compilers using different optimization flags and being build for various machine architectures.

To improve the matching precision, we use weighted code features. Machine learning is used to optimize the feature weights to create clusters of semantically similar code blocks extracted from different binaries. The machine learning is performed in an offline process to increase scalability and performance of the matching system.

Using above methods we build a database of binary similarity code signatures for open source libraries. The database is utilized to match by similarity any code blocks from an application to known libraries in the database. One of the goals of our system is to facilitate a fast and scalable similarity matching process. This allows integrating the system into continuous software development, testing and integration pipelines.

The evaluation shows that our results are comparable to other systems proposed in related research in terms of precision while maintaining the performance required in continuous integration systems.

Keywords: binary similarity, code similarity, locality sensitive hashing, lsh, software composition analysis, sca

Sipola J. (2023) Nopea ja skaalautuva käännettyjen ohjelmistojen samankaltaisuuden tunnistusmenetelmä avoimen lähdekoodin kirjastoille. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 63 s.

TIIVISTELMÄ

Kolmansien osapuolten kehittämien ohjelmistojen käyttö on yleistynyt valtavasti viime vuosien aikana nopeutuvan ohjelmistokehityksen ja laadukkaiden ohjelmistokirjastojen tarjonnan kasvun myötä. Nämä kirjastot ovat yleensä lisätty kehitettävään ohjelmistoon riippuvuuksina ja usein jopa käännettyinä binääreinä. Tämä on yleistä varsinkin sulatetuissa ohjelmistoissa. Riippuvuudet saattavat kuitenkin luoda uusia hyökkäysvektoreita kirjastoista löytyvien haavoittuvuuksien johdosta. Nämä kolmansien osapuolten kirjastoista löytyvät haavoittuvuudet synnyttävät tarpeen tunnistaa käännetyistä binääriohjelmistoista löytyvät avoimen lähdekoodin ohjelmistokirjastot.

Binäärien samankaltaisuuden tunnistusmenetelmät usein pohjautuvat tekstin samankaltaisuuden tunnistusmenetelmiin ja hyödyntävät tämän tieteellisiä saavutuksia. Tässä tutkimuksessa keskitytään sumeisiin tunnistusmenetelmiin, joita on käytetty laajasti tekstin samankaltaisuuden tunnistamisessa. Tutkimuksessa hyödynnetään sijainnille sensitiivisiä tiivistemenetelmiä ja normalisoituja binäärien ominaisuuksia. Ominaisuuksien normalisoinnin avulla binäärien samankaltaisuutta voidaan vertailla ohjelmiston kääntämisessä käytetystä kääntäjästä, optimisaatiotasoista ja prosessoriarkkitehtuurista huolimatta.

Menetelmän tarkkuutta parannetaan painotettujen binääriominaisuuksien avulla. Koneoppimista hyödyntämällä binääriomisaisuuksien painotus optimoidaan siten, että samankaltaisista binääreistä puretut ohjelmistoblokit luovat samankaltaisien ohjelmistojen joukkoja. Koneoppiminen suoritetaan erillisessä prosessissa, mikä parantaa järjestelmän suorituskykyä.

Näiden menetelmien avulla luodaan tietokanta avoimen lähdekoodin kirjastojen tunnisteista. Tietokannan avulla minkä tahansa ohjelmiston samankaltaiset binääriblokit voidaan yhdistää tunnettuihin avoimen lähdekoodin kirjastoihin. Menetelmän tavoitteena on tarjota nopea ja skaalautuva samankaltaisuuden tunnistus. Näiden ominaisuuksien johdosta järjestelmä voidaan liittää osaksi ohjelmistokehitys-, integraatioprosesseja ja ohjelmistotestausta.

Vertailu muihin kirjallisuudessa esiteltyihin menetelmiin osoittaa, että esitellyn menetlmän tulokset on vertailtavissa muihin kirjallisuudessa esiteltyihin menetelmiin tarkkuuden osalta. Menetelmä myös ylläpitää suorituskyvyn, jota vaaditaan jatkuvan integraation järjestelmissä.

Avainsanat: käännettyjen ohjelmistojen samankaltaisuus, käännetyn lähdekoodin samankaltaisuus, sijainnille herkkä tiiviste, lsh, ohjelmistojen koostumuksen analyysi, sca

TABLE OF CONTENTS

FOREWORD

I would like to thank my thesis supervisors and work colleagues for the great guidance they have provided during my thesis project and my academic journey. I would also thank my family members for the support they have provided.

Oulu, January 10th, 2023

Juho-Pekka Sipola

LIST OF ABBREVIATIONS AND SYMBOLS

ACFG	Attributed Control Flow Graph
ALSH	Adaptive Locality Sensitive Hashing
AST	Abstract Syntax Tree
AUC	Area Under the Curve
BHB	Best-Hit-Broadening
CBOW	Continuous Bag of Words
CFG	Control Flow Graph
CG	Call Graph
COTS	Commercial-Off-The-Shelf
DFG	Data Flow Graph
FN	False Negative
FP	False Positive
ICFG	Inter-procedural Control Flow Graph
IR	Intermediate representation
ISA	Instruction Set Architecture
IoT	Internet of Things
LCS	Longest common sequence
LM	Language Modeling
LSH	Locality Sensitive Hashing
MDS	Multidimensional Scaling
MI	Mutual Information
NCD	Normalized Compression Distance
NLP	Natural Language Processing
OSS	Open Source Software
PV-DM	Paragraph Vector Distributed Memory
ROC curve	Receiver Operating Characteristic Curve
RTOS	Real Time Operating System
SBOM	Software Bill of Materials
SCA	Software Composition Analysis
SGD	Stochastic Gradient Descent
SVM	Support Vector Machine
TN	True Negative
TP	True Positive
Tree-LSTM	Long Short-Term Memory
kNN	K-Nearest Neighbor

1. INTRODUCTION

Binary similarity detection is a process where similarities between different binary executable files are computed. The similarity between different binaries can be computed using different methods in a deterministic way. These methods are often closely related to text similarity and graph-based similarity, therefore it often leverages many methods and models developed for these fields of research. Binary similarity detection has been studied extensively during the past three decades and it has been utilised for several different use cases from malware analysis to license violations [1, 2]. Similarity detection can leverage code graphs [3], assembler text [4], abstract code features [5] or machine-learned [6] based similarities present in different binaries. With these methods security of compiled software applications can be improved. Moreover, binary similarity research has enabled researchers to examine the content of binary executable files even if source code is not available. Thus allowing examination of applications' building blocks even if they have not published a software bill of materials (SBOM).

Binary similarity research has evolved rapidly in past years due to new techniques and improvements in machine learning and deep learning. Machine learning especially natural language processing (NLP) and different types of neural networks have been utilized to learn the semantics of binary disassembly [1] with relative success. These systems usually required a training stage to detect and classify open source libraries. This makes systems depending on machine learning models relatively slow or unscalable. To overcome this limitation, we propose a system that utilizes locality-sensitive hashing [7] for fast and robust binary similarity detection for open source software (OSS) libraries across different compilations of a binary.

The proposed method analyses features of functions compiled to different architectures, optimisations levels and compilers. From function disassembly feature vectors are extracted to determine semantic similarities across different functions. Feature vector contains 36 unique features extracted from binary functions. A minimal intermediate representation is used to further categorise different instructions into semantic groups to reduce the instruction diversity from different processor architectures. Feature vectors are converted into weighted hashes by utilising locality-sensitive hashing. Locality-sensitive hashing lowers the dimensionality of the matching problem by reducing the complexity of single similarity comparison from many-to-many to one-to-one. Locality-sensitive hashing allows set similarity methods and hamming distance to be used for computing similarities between binary samples.

The proposed method seemed to achieve comparable performance in matching times and matching accuracy to methods proposed in the literature. The proposed binary similarity system achieved 75 percent area under the curve with a sample set containing 5671 binary pairs. Furthermore, an average matching time of 120 seconds for binaries containing 1200 functions was observed. Moreover, the average matching time with a database containing 20000 hashes was 0.35 seconds. The accuracy and specificity of the system seem a bit lower than most methods studied in the literature. Lower accuracy is likely caused by computationally less intensive features in the feature vector and a smaller training set during feature weight optimisation. However, the matching performance of the system was comparable to or even better than the methods represented in the literature.

2. RELATED WORK

Binary similarity can be used for several different applications, traditionally binary similarity has been used for clone detection, malware analysis, patch detection, license violation detection, intelligent electronic devices vulnerability, vendor and third-party component detection [8]. Therefore, research on binary similarity has been evolving significantly over the years. In past years research has been focusing on machine learning and natural language-based solutions compared to solutions relying on signatures or graphs to determine the similarity of binaries. Systems leveraging different types of machine and deep learning methods can be trained to detect similarities and dissimilarities that may be difficult for humans to detect. Alternatively, signature or graph-based similarities can be used to detect and compare similarities between software applications.

Binary similarity can be measured based on different data points found in binary executable files. Data based binary similarity often relies on printable characters found from read-only sections of executable files whereas code based binary similarity relies on instructions found from executable code sections. Data extracted from read-only sections of the executable files like .data or .rodata in an ELF executable have been established as one of the most characteristic and reliable feature used in binary similarity detection. Data in these sections tend to retain similarity between different compilations and instruction set architectures. The read-only data can be further enhanced with information about the context in which data is encountered to improve the matching accuracy. The study indicates that the read-only data embedded with context information can be accurately used to detect open source libraries [9, 10]. However, data extracted from read-only sections is not usable in all situations as some libraries do not have any read-only data or have very little read-only data which makes other binary similarity detection methods necessary for certain executable libraries.

The similarity between two different pieces of software can be measured in several ways like shown by Haq et al. [1]. Measures often used for similarity are semantic and syntactic similarity which have been a relatively popular choice in past research alongside different variations of structural similarities utilising graph-based similarity derived from control flow graphs and call graphs. Furthermore, targeted platforms or instruction set architectures have varied a lot between different researches. The most of proposed implementations and methods in the literature support one or more common executable architectures like X86-64 or ARM.

Clone detection systems may be divided into four different categories based on the detection method used for similarity detection. Binary similarity detection methods are commonly categorised in the literature into the following categories

1. Textual Similarity
2. Lexical or Token Based Similarity
3. Syntactic Similarity
4. Semantic Similarity

Textual similarity compares identical code instructions, whereas lexical or token-based similarity compares only a subset of extracted tokens from instructions. These tokens can for example include unique or different variables and literal names. Syntactic similarity looks at the structure of a list of instructions and their relations. Semantic similarity compares the values and functional computation of code. [11, 12] These categories for clone detection can also be applied for other applications of binary similarity than clone detection. Different types of binary similarity detection systems regardless of the use case often rely on the same methods. Therefore, this categorisation can be broadened to other applications of binary similarity than clone detection. In this research, the main focus is on the syntactic and semantic similarities of binary executable files. As binaries might be compiled with different tools and compilation flags structure of binary may be different, hence structure, textual and lexical similarities are not feasible. Even though, in the literature, all these methods and their deviations have also been used with varying success [1, 13].

Binary similarity research can be divided into two distinct branches of research based on the methodology and mechanism used to detect similarities between binary files.

- Machine learning aided
- Non-machine learning aided

The main differences between these two categories usually are how the similarity matching is done and what types of mechanism is used to detect them. Machine-learning aided research leverage different machine-learning models to differentiate similar and dissimilar binaries and the non-machine learning aided methods usually utilise different graphs and features extracted from binaries to detect similarities. Non-machine learning aided solutions typically leverage more from expert knowledge of the field for binary similarity matching. This expert knowledge could include various aspects of binary similarity like a set of features [14, 5, 15, 2], analysing execution traces [16] or even conditional formulas [17]. Whereas machine learning aided solutions use machine learning models to learn similarities between binaries or binary features [18, 19, 20, 21]. Non-machine learning solutions might also use models usually associated with machine learning in other parts of the system like value optimisation.

2.1. Binary Similarity Research

Systems and implementations relying on machine learning for binary similarity comparison have been trending heavily in binary similarity research in past years, hence recent systems proposed in the literature that do not rely on machine learning have become more and more sparse. Machine learning aided systems typically gain additional accuracy with trained models that can detect semantic similarities across different architectures and compilations with a smaller amount of features. However, these types of systems do not scale as well as signature or graph-based systems in certain use cases. Therefore, implementations have been proposed that utilise different types of aggregated and labelled graphs [22, 23] or feature sets [15, 5, 24] extracted from binaries. In the literature non-machine learning aided methods and systems can be dived into two rough categories, graph-based and feature-based similarity detection solutions. The graph-based similarity solutions leverage from different types of graphs like control flow graphs, call graphs or data flow graphs [25, 26, 27, 28, 29, 3] or graphs derived from these graphs like attributed control flow graphs (ACFG) [23]. The graph-based similarity comparison is a known NP-complete problem [30, 28], thus methods that utilise graphs for binary similarity detection have implemented several different optimisations to overcome this problem from K-subgraphs [31, 1, 20] to locality-sensitive hashing [32, 33, 34, 14, 23] and different types of tree structures [25, 31] have also been used. These methods aim to lower the complexity and dimensionality of the matching problem to make similarity matching and detection faster.

In the feature-based similarity detection methods, a set of features is extracted from binary disassembly. These features can be graph-based features that leverage relationships observed in different graphs. Structural features and statistical features are also often seen in systems like these. Structural features capture the structure of observed binary and statistical features measure the statistical distribution of instructions in the binary disassembly [24, 5]. Feature-based binary similarity methods often have graph-based features, however these methods often utilise other similarity metrics like set similarity [5, 25] for similarity detection rather than complex graph-based similarity. These methods usually seem to have lower matching precision compared to the graph-based methods, however feature-based methods usually have higher performance.

To improve the precision of these systems dynamic execution can be used to capture the semantic behaviour of binary executable or filter function candidates [26, 35, 27], but this is an expensive operation and introduces overhead to the matching process. Thus, many systems have to balance between precision and matching performance.

2.2. Machine Learning Aided Binary Similarity Research

In the past couple of years, machine learning has been trending in the field of binary similarity. Machine learning provides a lot of powerful tools and methods to detect similarity between different binaries and even in some instances remove steps that have been nearly mandatory for binary similarity detection like microcode normalization into an intermediate representation. Machine learning techniques tend to reduce the need for expert knowledge of different binary architectures and instruction sets and might even remove a need for feature extraction from disassembly [36]. Machine learning models and deep learning methods like neural networks have been used for semantic similarity analysis of source code and binary disassembly.

A neural network can be used to learn the semantics of a program from source code or binary disassembly by using natural language processing methods. Natural language processing can be applied to binary code that is lifted into an intermediate representation. This can often mitigate the effects of different processor microcodes, compilers or high-level programming languages in similarity detection. Combined with natural language processing methods current machine learning models can learn and classify semantic similarities of source code and binary disassembly [37, 21, 36, 38, 35]. Furthermore, neural networks have been successfully used for classifying functions into semantic groups like into functions that perform encrypting or sorting alongside detection of vulnerable function search [39]. One commonality between many neural networks used for binary similarity detection is natural language processing and leveraging the latest achievements in that field of research.

Deep learning and machine learning techniques can also be applied to different graphs and feature sets or even have combinations of these methods [40, 19, 41]. These methods can either use these techniques to increase matching performance by filter matching candidates or to perform complex matching problems. Due to this, machine learning aided solutions tend to achieve high performance on matching tasks, but they are often limited by the mandatory model training process that has to be performed at the system startup which makes them less scalable compared to non-machine learning aided graph or feature-based systems.

3. A BINARY SIMILARITY MATCHING SYSTEM BASED ON LOCALITY SENSITIVE HASHING

In modern software development, third-party libraries are often used to substitute pieces of software developed in-house to speed up development cycles. These software libraries are often delivered to end users either in source code or in a compiled form, thus BOM of these libraries is not always known. This introduces a need for software composition analysis (SCA) [42] tools and methods to detect possible vulnerabilities and components used to produce software as early as possible. This is achieved by source code analysis or analysis of executable binary produced by a compiler. In this research, the focus is on binary similarity analysis.

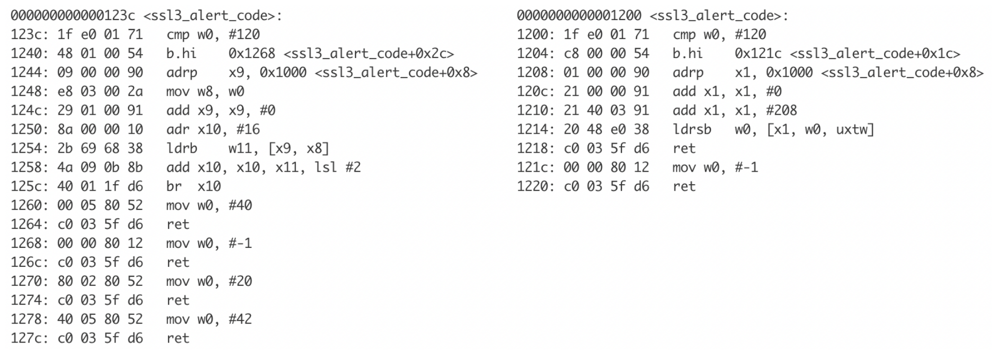
To detect vulnerable open source libraries as early as possible binary similarity analysis has to be located relatively early phase of software development like in a continuous integration pipeline where SCA tools are commonly used. This sets performance requirements for the proposed system as the matching mechanism has to be fast enough to not slow down the development process. Hence, in this research, we aim to implement a binary similarity analysis mechanism that is scalable in database size and is easy to deploy. Furthermore, it should possess a fast similarity matching against a set of known open-source libraries.

David et al. [30] stated in their research that methods known in the image similarity comparison known as similarity by composition can be also utilised in the field of binary similarity. Similarity by composition means that similar images should consist of similar regions. This can be applied to the binary similarity as similar executables should consist of similar functions or basic blocks. The challenging task is to determine when basic blocks or functions are considered to be similar or semantically equivalent.

Binary similarity can be divided into several categories, syntactically similar, syntactically same, semantically identical or exactly the same [43]. Function structure varies a lot between different compilations of the same source code without changing the core functionality. Therefore, the similarity of different binary executables has to be measured based on the semantic similarity of functions, graph similarity or other similarity measures as the functions are rarely identical or syntactically the same across different compilations.

There are currently several different commonly used compilers available like GCC and Clang which allow end users to compile their source code into a binary format. Semantically similar instructions and functions used to perform tasks might vary a lot between different compilers, optimisation levels and instruction sets. This variation is caused by several different reasons like arbitrary register use, different instruction selections that are semantically similar or a combination of instruction sequences [44]. This can be seen in the sample function in Figure 1. The sample function is compiled from the same source code and using the same compiler optimisation level O3 for both function samples, but the compilers used to compile the binary are different. In the left function sample Clang version 11.0.0 is used and on the right GCC version 10.2 is used. This makes semantic similarity-based matching necessary due to the volatile nature of the executable structure.

In past research, several solutions have been proposed to solve this problem like embedding analysis and models [39, 19] or different variants of fuzzy matching [5, 24]. These methods allow some variation in the instruction sequences extracted from the function body by different means. In this research fuzzy matching is utilised to allow a certain amount of changes to happen in a function instructions before they are deemed to be dissimilar.



```
000000000000123c <ssl3_alert_code>:
123c: 1f e0 01 71   cmp w0, #120
1240: 48 01 00 54   b.hi    0x1268 <ssl3_alert_code+0x2c>
1244: 09 00 00 90   adrp    x9, 0x1000 <ssl3_alert_code+0x8>
1248: e8 03 00 2a   mov w8, w0
124c: 29 01 00 91   add x9, x9, #0
1250: 8a 00 00 10   adr x10, #16
1254: 2b 69 68 38   ldrb    w11, [x9, x8]
1258: 4a 09 0b 8b   add x10, x10, x11, lsl #2
125c: 40 01 1f d6   br  x10
1260: 00 05 80 52   mov w0, #40
1264: c0 03 5f d6   ret
1268: 00 00 80 12   mov w0, #-1
126c: c0 03 5f d6   ret
1270: 80 02 80 52   mov w0, #20
1274: c0 03 5f d6   ret
1278: 40 05 80 52   mov w0, #42
127c: c0 03 5f d6   ret
```

```
0000000000001200 <ssl3_alert_code>:
1200: 1f e0 01 71   cmp w0, #120
1204: c8 00 00 54   b.hi    0x121c <ssl3_alert_code+0x1c>
1208: 01 00 00 90   adrp    x1, 0x1000 <ssl3_alert_code+0x8>
120c: 21 00 00 91   add x1, x1, #0
1210: 21 40 03 91   add x1, x1, #208
1214: 20 48 e0 38   ldrsb   w0, [x1, w0, uxtw]
1218: c0 03 5f d6   ret
121c: 00 00 80 12   mov w0, #-1
1220: c0 03 5f d6   ret
```

Figure 1. Differences between different compilers left Clang and right GCC

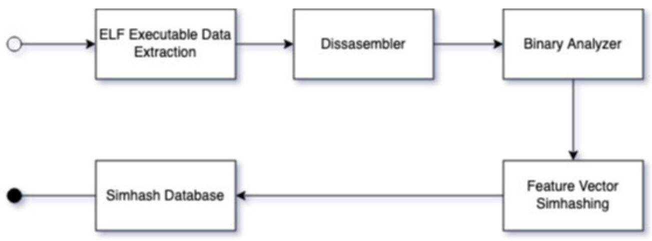


Figure 2. Simhashing pipeline

After an evaluation of past research and existing binary similarity methods, we have decided to utilise locality-sensitive hashing (LSH). LSH allows efficient and scalable fuzzy matching techniques to overcome differences in a binary executable structure inspired by research conducted by Dullien [24] and Wang et al. [5]. To measure the semantic similarity of a set of binaries, we use features in the binaries with LSH techniques as they provide a flexible and robust presentation of executable semantics.

The contribution of each executable feature to the final similarity is further adjusted in an offline process. This reduces runtime overhead compared to many purely machine learning based solutions by moving computationally complex operations from matching to a separate offline process. Furthermore, the solutions have to be horizontally scalable and possess good performance to ensure fast, accurate and expandable binary similarity matching in a real-world scenario like a continuous integration pipeline.

In Figure 2 the end-to-end pipeline of the proposed binary similarity analysis mechanism step by step is represented. The proposed binary similarity matching method can be divided into five general steps. The first step is executable metadata extraction, the second step is disassembling executable code and the third step is feature vector creation. The final two steps are the creation of locality-sensitive hashes from feature vectors and the matching of extracted hashes to a database of known locality-sensitive hashes.

3.1. Locality Sensitive Hashing

Like Marcelli et al. [45] stated in their research binary function similarity measurements can be divided into two categories.

Direct Similarity Comparison Approach rely on one-to-one similarity comparison over a set of instructions, function features or other data that can be extracted from a function after which the similarity of function pairs has to be determined by machine learning algorithms or other similar means.

Indirect Similarity Comparison Instructions or function features are converted from a higher dimension to a lower dimension by using methods like locality-sensitive hashing and comparing similarity through lower-dimensional comparison.

In the proposed system similarity is measured using aspects of both indirect and direct measurements as locality-sensitive hashing of function features is utilised. The indirect approach offers better scalability and less complex similarity comparison compared to the direct approach. Whereas direct similarity measurement provides more accurate similarity measurement, thus aspects of both are used in the similarity computation.

The proposed binary similarity analysis method leverages LSH for similarity matching. Locality-sensitive hashing is a type of fuzzy matching that allows matching or approximate nearest neighbour searches between input values. This is achieved by computing a hash of input value, thus projecting the input values from higher dimensional space into lower dimensional space. Moreover, locality-sensitive hashes are often stored into buckets or clusters of similar hash values which allows efficient one-to-many searches and grouping of same or near similar values compared to exact nearest neighbour search. Values in a hash bucket or cluster are deemed to be similar. Hence, having a high probability that values in the same cluster or bucket are closer to each other. Consequently, values that are deemed dissimilar have a low probability to land into the same hash bucket. [46, 47]

LSH aims for collisions of hash bits when comparing similar inputs. This property of hash functions is defined by collision probability. Collision probability is a feature of hash functions that describes the probability of two unique values producing the same output. This is not desired feature in cryptographic hash functions, however in LSH functions this is desired as LSH aims to produce the same output for similar input values. Therefore, in a binary similarity analysis system LSH allows changes to happen between input values before the resulting locality-sensitive hash changes significantly or the distance between two LSH values increases. This makes the system more resilient to changes happening in the function instructions between different compilations.

LSH chosen for similarity matching in the proposed system is simhash which was proposed the first time as early as 2002 by Charikar [48] and used successfully for binary similarity analysis [24, 5]. The efficiency of the simhash algorithm on large-scale near-duplicate detection systems was studied further by Henzinger et al. [49] in text-based similarity measurements. Results of these studies indicate that the simhash has better scalability compared to other locality-sensitive hashes like MinHash proposed by Broder in 1997 [50]. One of the most significant applications of this locality-sensitive hashing and especially simhashing technology has been near duplicate detection developed by Google for their search engine to detect duplicate web pages. The application developed by Google is similar to the system proposed in this research. Both utilise feature-based similarity detection with carefully selected feature sets. Hence, simhash with hamming distance can be used as a similarity detection mechanism [51]. However, the set of features in both of these systems are vastly different which makes the signature creation different in both cases.

In the common use case, the simhash is applied to sets of text, strings or n-grams where simhash is constructed from shingles of words or other text features. Shingles are generally a continuous subsequence of the source [52]. Shingles are then converted and normalised into a condensed presentation by using a hash function like SHA-1 or MD5 to achieve constant variable length over varying inputs. For the input normalisation, the hash function used is not required to be cryptographic, thus the performance of the hash function is the key criterion in the selection process.

The similarity between two simhash values is compared by computing a hamming distance between the two values. Hamming distance measures changes in the hash bits and gives the distance as differing bit values between the two input values. Thus, with simhash and hamming distance set of hash values can be grouped into sets of similar hashes by performing XOR operation over the values and counting differing bits [51, 24] i.e. computing the hamming distance of two hashes. Hamming distance computation is a relatively fast operation to perform on modern computer systems and it provides efficient computation of similar pairs as it relies on logical XOR operations between two integers [24, 51].

Algorithm 1. Simhash similarity comparison algorithm

```
Input  : Two Simhash values to be compared
Output: Boolean indicating if Input values are deemed to be similar
1 $Distance = hammingDistance(simhash1, simhash2)$
2 if $Distance < Threshold$ then
3     $Similar = True$
4 end
5 else
6     $Similar = False$
7 end
8 return $Similar$
```

Simhashes used in this research are 128-bits long which reduces the amount of false positive collisions, but they are not too long to decrease performance significantly. Hamming distances threshold value has to be chosen to determine similar and dissimilar hash values. Like demonstrated in Algorithm 1 hash pairs that have distances less than the chosen threshold value are determined to be similar and hash pairs with distances more than the threshold value are determined to be dissimilar. In past research threshold values have varied significantly, but for this research hamming distance threshold value 8 is used to determine similar and dissimilar hash pairs. The threshold was chosen as it produced more accurate results without causing too many false positive matches in an evaluation binary set.

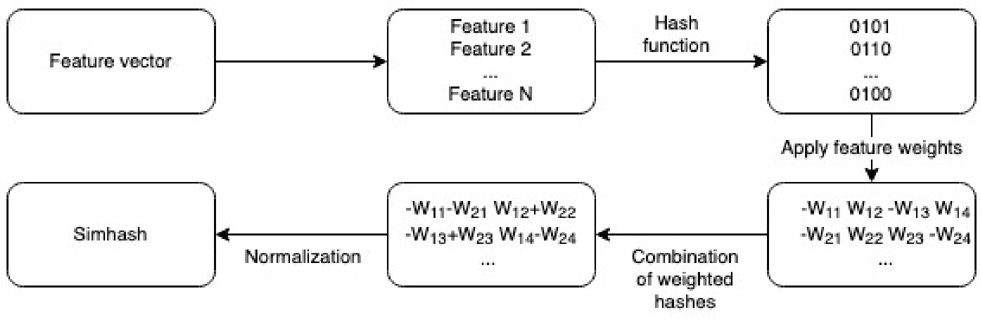


Figure 3. Weighted Feature vector simhash process

Simhash is constructed from a high dimensional feature vector by hashing feature values into static length byte arrays and joining hashed and weighted feature values together [5, 24]. Figure 3 demonstrates the process of converting a set of feature vectors into a set of simhash values described in the literature.

Feature vectors consist of numerical features extracted by analysing the structural and statistical distribution of instructions in a function and by analysing graphs derived from the function. Numeric values for each feature in a feature vector range from 0 to arbitrarily large numeric values. Hence, bit lengths for feature values can vary significantly. To overcome this issue, feature values have to be hashed to ensure static bit lengths. Once the feature values have been converted into desired bit length, feature values are converted from decimal or hexadecimal presentation into a binary presentation. Feature weights are applied to sequences of ones and zeros by adding a positive weight value to the sequence if the hash bit is one and a negative weight if the hash bit is zero. Weighted feature value sequences are combined into the final simhash value by adding up the weighted bit values from each feature value sequence into one by summing values from all feature binary value sequences with the same index value. Combined feature sequence values are then normalized by turning negative bit values in the sequence into zeros and positive bits in the sequence into ones. This allows simhash to be applied to a weighted feature vector while maintaining compliance with simhash specification. The final sequence of ones and zeros is joined together to construct the simhash of a feature vector.

3.2. Defining a Similarity Score

The similarity between two binary executable files has to be measured in a comparable way to ensure meaningful results. Typically different distance measures are used for data that consists of single points of measurement. The set similarity is used if comparable items consist of multiple points of measurement. In this research, measurement points consist of function hashes extracted from a binary which are compared against a set of known function hash sets, thus set similarity can be used for similarity measure between binaries. Set similarity usually computes similarity value based on set sizes, intersection and union between compared hash sets. Between different compilations of binary executable, some parts of binaries or binary functions might be excluded or included from the final executable. Hence, similar executable files might not have the same amount of function hashes which creates a need for fuzzy matching that allows the comparison of different-sized hash sets.

In the case of locality-sensitive hashing intersection between two hash sets consists of hashes that are deemed to be similar by comparing their hamming distance to the hamming distance threshold value. The similarity score between two different hash sets has to be computed in a way that the similarity score does not favour the larger or the smaller of the two hash sets by yielding a higher score value to either the larger or the smaller set when compared against each other. In state-of-the-art systems, several different similarity scores have been used like the Jaccard similarity coefficient. However, some of these similarity scores cannot be applied to the system proposed in this research as they do not take into account the varying size of hash sets.

The similarity between two binary executable files is typically defined by how much database reference is contained in a binary sample. The reference set in the divisor can be either the sample file or the database reference. We prefer the database reference as a divisor as we want to know how much of that reference is found in the sample. This is motivated again by the fact that a sample binary can contain a combination of different components code - all combined (linked) together into one file. Hence, the similarity would be lower the more components are combined into one file. However, we want to measure for each component in a file how many code parts (i.e functions) it contains as compared to the database reference.

After careful consideration of several different similarity scores that are available for the similarity score, the most promising ones are taken into for further evaluation. Similarity scores considered for the task are Tversky Index proposed by Amos Tversky in 1977 [53] and its symmetric variant proposed by Jimenez and Gelbukh [54], containment similarity [55] and Overlap coefficient similarity [56]. These similarity metrics fulfil all or most of the requirements set for a similarity metric that can be utilised in this system.

Symmetric Tversky Index is computed using Equation 3 where $a = min(|X-Y|,|Y-X|)$ and $b = \max(|X-Y|,|Y-X|)$. α and β values in the equation are used to give a weight for each set. The non-symmetric variant of the Tversky Index was left out of the evaluation as it is not suitable as a similarity metric. Containment similarity computes the size of the intersection between two sets and is divided by the size of the set used for a reference point as then in Equation 1. Overlap coefficient also known as Szymkiewicz–Simpson coefficient [56] is computed using Equation 2 where the intersection of two sets is divided with the size of the smaller set.

$$\text{Containment} = \frac{|A \cap B|}{|B|} \tag{1}$$

$$\text{Overlap Coefficient} = \frac{|A \cap B|}{min(|A|,|B|)} \tag{2}$$

$$\text{Tversky Index} = \frac{|X \cap Y|}{|X \cap Y| + \beta(\alpha a + (1-\alpha)b)} \tag{3}$$

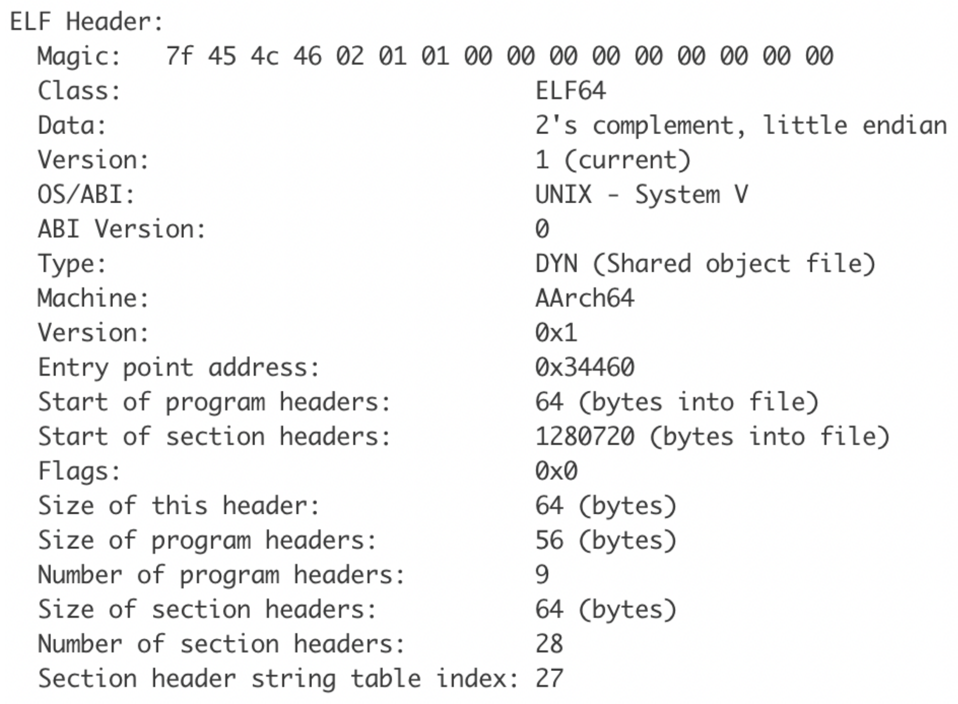
The similarity score produced by Tversky Index can be manipulated by changing the contribution of each set. This is achieved by adjusting α and β values. This makes two sets contribute more or less the final similarity scores depending on values set to α and β constants. After observing similarity scores across fixed evaluation binary set α value 0.2 and β value 0.8 was chosen for Tversky index constants. The values could be adjusted based on set sizes during the similarity computation, but this is ignored as it would make the comparison of vastly different-sized sets more complex and it would introduce unnecessary overhead in the similarity score computation. The Tversky Index score can adjust the bias towards the larger hash set, thus the smaller set contributes less to the final matching score. The containment similarity considers how much set A intersects with set B divided by the size of the set B. The contribution of each set and divisor are fixed in containment similarity and does not provide the flexibility offered by Tversky Index. The overlapping coefficient could also be a good choice, however it will favour the smaller one of two hash sets as the smaller set is chosen from two patterns. This would make the similarity score favour the smaller set and yield false positive matches when the executable contains a really small amount of functions and hence is not suitable for the similarity score for this research.

After evaluation of the available options for similarity metric score containment and symmetric Tversky Index proved to be two of the most promising choices. Containment similarity offers flexibility when comparing sets that are not constant in size. Whereas the Tversky index allows adjusting the contribution of sets, thus making it more flexible when the importance of two sets is not the same. From these options, Tversky Index proved to be the best fit for the similarity score for a system like this. Tversky Index provides flexibility by allowing changes to the contribution of each set compared to ensure more accurate matching.

3.3. Executable File Formats for Binaries

Over the years different types of executable formats have been created to serve different purposes, architectures and operating systems. Commonly used executable formats for open source libraries especially in the field of Internet of Things (IoT) applications are targeting specific hardware boards or real-time operating systems (RTOS). Hence, IoT applications are often designed to run on devices that have a restricted amount of resources available to them mainly amount of available memory or processing power. These limitations make more common executable formats and operating systems usually unsuitable for many of the most limited IoT boards and applications.

Often these systems utilise kernels optimised for certain tasks within the resource limitations introduced by the hardware of the board. Commonly used kernels for highly optimised systems are relatively lightweight Linux kernels and their derivatives or other real-time operating systems solutions like FreeRTOS. To optimise consumed resources as much as possible developers have often applied different executable formats and optimisations that might have been derived from the standard ELF executable specification. These stripped and optimised executable formats are often specific for certain boards or they utilise non-standard executable formats. Hence, they might lack function boundary information. This would create a need for custom-made function boundary detection that is out of the scope of this research. Function boundary detection would require methods like interprocedural control flow graph (ICFG) based solution proposed by Andriesse et. al [57] or with machine learning and signature-based solutions like ByteWeight [58]. These methods are used to detect function boundaries from unknown binary executable formats. Due to the limited documentation and requirement for function boundary detection optimised and non-standard ELF executable formats are left out of the scope of this research and the research focuses on ELF executable file format.



```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           AArch64
  Version:                           0x1
  Entry point address:               0x34460
  Start of program headers:          64 (bytes into file)
  Start of section headers:          1280720 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         28
  Section header string table index: 27
```

Figure 4. Example ELF file header

ELF executable file has a header that contains metadata about the executable that is used to detect the correct processor architecture for the executable as seen in Figure 4. From the ELF executable header metadata like offsets related to executable code sections are extracted. Code sections contain machine instructions of a program [59], thus containing the main functionality of a binary executable.

Metadata extracted from the ELF executable header consists of function offsets, sizes and names that are located in a symbol table. Furthermore, function thunks and trampolines are extracted from the corresponding jump tables for symbol relocation analysis for shared libraries that are described in ELF specification [59]. Trampolines are memory locations containing jump instructions to other sections in the executable code, thus creating indirect jump instructions. Thunks are function sub-routines inserted into the caller function during the linking time. Function relocation metadata is required to resolve shared library jumps and calls that may occur in the executable code section. With the function metadata extracted and normalised, executable instructions disassembly are mapped into corresponding functions that are extracted based on symbol information contained in the ELF header.

3.4. Disassembly Analysis

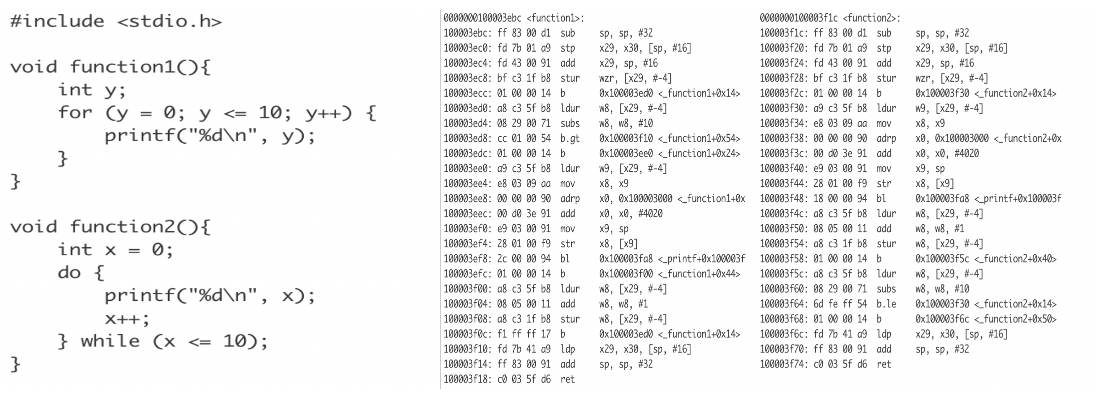
To analyse instructions and the semantic similarity of a function, binary has to be disassembled from the native binary presentation into a set of machine instructions for further analysis. To accomplish this task there are several out-of-the-shelf open source solutions available like Angr [60] or commercial-off-the-shelf (COTS) solutions like IDA Pro that perform binary disassembly into machine instructions and construction of graphs like control flow graphs (CFG) or call graphs (CG). Many existing binary similarity analysis solutions rely on these off-the-shelf disassembly analysing tools [1]. However, these solutions do not provide the features and performance required for the proposed system. Therefore, a new binary analysis framework has been created for this research.

In general, existing binary disassembling and analysis tools and solutions provide relatively good performance and flexibility over various executable formats for forensic analysis tools or plugins in most use cases. However, publicly available binary forensic platforms overall do not provide fast and comprehensive enough access to instruction extraction and classification process. By implementing our own analysis platform dependencies on third-party tools and libraries are reduced. Furthermore, only necessary parts for the proposed system are implemented and the platform can be easily expanded to support new executable format if needed. In summary, we chose to implement our own disassembler and feature extractor due to flexibility and scalability needs and the low footprint of our own implementation.

3.4.1. Instruction Set Architectures

Instruction set architecture (ISA) describes a set of instructions that can be executed by a certain processor. There are often major differences between two different executable ISAs and even between different versions of the same instruction set architecture may have a significant variation in the instruction set, presentation of instructions and semantics of instructions.

ISA can be considered a human-readable presentation of byte sequences used by different processor manufacturers and models to perform certain tasks and operations. Different ISAs have different models to present resources and operations available like registers or arithmetic operations. Therefore, functions that perform semantically similar tasks might have a vastly different presentation in disassembly format between different ISAs and semantically similar code segments. For example, like seen in Figure 5 where on the left is the source code for two semantically identical functions and on the right is the resulting disassembly for the same functions using the same compiler and compiler flags. Disassembled instructions have to be normalised into a format that is comparable across different ISAs and instruction set versions without loss of vital information.



```c
#include <stdio.h>

void function1(){
    int y;
    for (y = 0; y <= 10; y++) {
        printf("%d\n", y);
    }
}

void function2(){
    int x = 0;
    do {
        printf("%d\n", x);
        x++;
    } while (x <= 10);
}
```

```
0000000100003ebc <function1>:
100003ebc: ff 83 00 d1  sub     sp, sp, #32
100003ec0: fd 7b 01 a9  stp     x29, x30, [sp, #16]
100003ec4: fd 43 00 91  add     x29, sp, #16
100003ec8: bf c3 1f b8  stur    wzr, [x29, #-4]
100003ecc: 01 00 00 14  b       0x100003ed0 <_function1+0x14>
100003ed0: a8 c3 5f b8  ldur    w8, [x29, #-4]
100003ed4: 08 29 00 71  subs    w8, w8, #10
100003ed8: cc 01 00 54  b.gt    0x100003f10 <_function1+0x54>
100003edc: 01 00 00 14  b       0x100003ee0 <_function1+0x24>
100003ee0: a9 c3 5f b8  ldur    w9, [x29, #-4]
100003ee4: e8 03 09 aa  mov     x8, x9
100003ee8: 00 00 00 90  adrp    x0, 0x100003000 <_function1+0x
100003eec: 00 d0 3e 91  add     x0, x0, #4020
100003ef0: e9 03 00 91  mov     x9, sp
100003ef4: 28 01 00 f9  str     x8, [x9]
100003ef8: 2c 00 00 94  bl      0x100003fa8 <_printf+0x100003f
100003efc: 01 00 00 14  b       0x100003f00 <_function1+0x44>
100003f00: a8 c3 5f b8  ldur    w8, [x29, #-4]
100003f04: 08 05 00 11  add     w8, w8, #1
100003f08: a8 c3 1f b8  stur    w8, [x29, #-4]
100003f0c: f1 ff ff 17  b       0x100003ed0 <_function1+0x14>
100003f10: fd 7b 41 a9  ldp     x29, x30, [sp, #16]
100003f14: ff 83 00 91  add     sp, sp, #32
100003f18: c0 03 5f d6  ret
```

```
0000000100003f1c <function2>:
100003f1c: ff 83 00 d1  sub     sp, sp, #32
100003f20: fd 7b 01 a9  stp     x29, x30, [sp, #16]
100003f24: fd 43 00 91  add     x29, sp, #16
100003f28: bf c3 1f b8  stur    wzr, [x29, #-4]
100003f2c: 01 00 00 14  b       0x100003f30 <_function2+0x14>
100003f30: a9 c3 5f b8  ldur    w9, [x29, #-4]
100003f34: e8 03 09 aa  mov     x8, x9
100003f38: 00 00 00 90  adrp    x0, 0x100003000 <_function2+0x
100003f3c: 00 d0 3e 91  add     x0, x0, #4020
100003f40: e9 03 00 91  mov     x9, sp
100003f44: 28 01 00 f9  str     x8, [x9]
100003f48: 18 00 00 94  bl      0x100003fa8 <_printf+0x100003f
100003f4c: a8 c3 5f b8  ldur    w8, [x29, #-4]
100003f50: 08 05 00 11  add     w8, w8, #1
100003f54: a8 c3 1f b8  stur    w8, [x29, #-4]
100003f58: 01 00 00 14  b       0x100003f5c <_function2+0x40>
100003f5c: a8 c3 5f b8  ldur    w8, [x29, #-4]
100003f60: 08 29 00 71  subs    w8, w8, #10
100003f64: 6d fe ff 54  b.le    0x100003f30 <_function2+0x14>
100003f68: 01 00 00 14  b       0x100003f6c <_function2+0x50>
100003f6c: fd 7b 41 a9  ldp     x29, x30, [sp, #16]
100003f70: ff 83 00 91  add     sp, sp, #32
100003f74: c0 03 5f d6  ret
```

Figure 5. Semantically similar function pair's disassembly and source code

3.4.2. Intermediate Representation

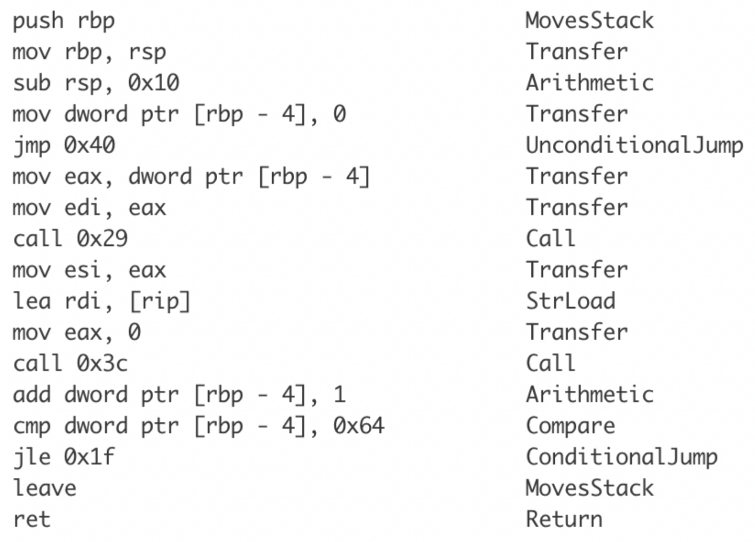
In an intermediate representation (IR) instructions are normalized into a universal format from architecture-specific instructions. IRs usually group semantically similar instructions into the same or similar IR expressions. This allows accurate comparison and optimisation of instruction and function semantics between different ISAs or binaries regardless of their original assembly instruction format or target CPU architecture.

There are several off-the-shell IR models developed to accomplish this task like VEX used by tools like Angr or LLVM IR used by the Clang compiler. These intermediate representations do a lot of analysis beyond the instruction normalization and semantic classification on lifted binary blobs like IF-ELSE block detection. These analysis steps are unnecessary for the proposed system as only instruction mnemonic normalization and semantic classification are needed. For this reason, a custom IR is created based on instruction mnemonics and registers. In this proposed IR only a minimal set of features provided by the more popular IR models are implemented as more complex analysis is not needed, thus creating unnecessary overhead in the binary lifting process.

Instructions are partially lifted into the IR by ignoring the more complex analysis of instruction operands. Figure 6 shows a sample of the proposed IR where single instruction may have been assigned to one or more semantic groups. Demonstrated by Tai et al. [61] only normalised sequences of instructions could be used for vulnerability detection. It does not provide good enough accuracy for the system proposed, thus further analysis of function instructions is required.

In the proposed IR instructions are grouped into categories listed in Table 1. Instructions are grouped into semantics groups based on their documented and observed behaviour. This ensures that semantically similar instructions with similar functionality across all supported architectures are grouped into the same semantic category. Architectures supported by the proposed IR language are X86-64, X86-32 ARM32, ARM64 and ARM Thumb instruction sets as they are more common CPU instruction sets used by modern processors.

Alongside mnemonic grouping, registers are extracted from the instruction operands. Instruction operands are extracted and further grouped into categories by their intended use as defined in architecture-specific calling conventions. The register



```
push rbp                                MovesStack
mov rbp, rsp                            Transfer
sub rsp, 0x10                           Arithmetic
mov dword ptr [rbp - 4], 0              Transfer
jmp 0x40                                UnconditionalJump
mov eax, dword ptr [rbp - 4]            Transfer
mov edi, eax                            Transfer
call 0x29                               Call
mov esi, eax                            Transfer
lea rdi, [rip]                          StrLoad
mov eax, 0                              Transfer
call 0x3c                               Call
add dword ptr [rbp - 4], 1              Arithmetic
cmp dword ptr [rbp - 4], 0x64           Compare
jle 0x1f                                ConditionalJump
leave                                   MovesStack
ret                                     Return
```

Figure 6. Intermediate representation of a sample function

categories used by the proposed IR are visible in Table 2. From the operands alongside register categories, source and target registers are tagged to make the operand analysis more accurate.

Table 1. Instruction semantic categories used in classification

Arithmetic	Data Transfer	Stack Operation
Bit Operation	Return	Call
Load	Store	Conditional Jump
String	Floating Point	Unconditional Jump
Interrupts	Processor	Compare

Table 2. Register categories used in classification

Callee Saved	Caller Saved	Temporary
Parameter	Local Variable	Stack Pointer
Frame Pointer	Instruction Pointer	Return Value
Immediate Value		

3.4.3. Basic Blocks and Function Boundaries

A basic block presents a sequence of instructions without branches, thus basic blocks may include one or more instructions that usually create a basis for binary similarity analysis methods and mechanisms. Basic blocks extracted from binary disassembly allow further extraction of functions and function boundaries. Basic blocks can be extracted from a binary utilising different methods, but all these methods follow a set of rules that are characteristic of a basic block.

The basic block extraction is presented in Algorithm 2 that follows rules [62] defined for basic block creation. In the algorithm, $isLeader$ function determines if an instruction belongs into categories that will end or start a new basic block like targets of jump instructions or instructions following a jump instruction. Basic blocks are extracted with two linear sweeps over the instruction set where the first sweep collects and stores all jump instruction target addresses and source addresses for each jump instruction. A jump instruction will end the current basic block and the target instruction of the jump will start a new basic block. Furthermore, call instructions might end the current basic block if the call target is a non-returning function call. The second sweep over function instructions will group instructions into basic blocks based on leaders and instructions that will end a basic block detected in the first sweep.

Linear sweep extracts and analyses instructions in a sequence one by one when encountered in the binary executable code sections. Another option for the basic block extraction is a recursive sweep, however it relies on CFG and it might leave certain code sections out if they are inserted into binary without using them. This might cause false negative matches which can be avoided with the linear sweep. However, the benefit of recursive sweep would be the simultaneous resolution of the control flow graph with basic blocks whereas with the linear sweep control flow have to be resolved separately. Accuracy gained with the linear sweep over the recursive sweep overcomes the performance gained with the recursive sweep. Hence, the linear sweep is selected over the recursive sweep.

Algorithm 2. Basic block creation algorithm

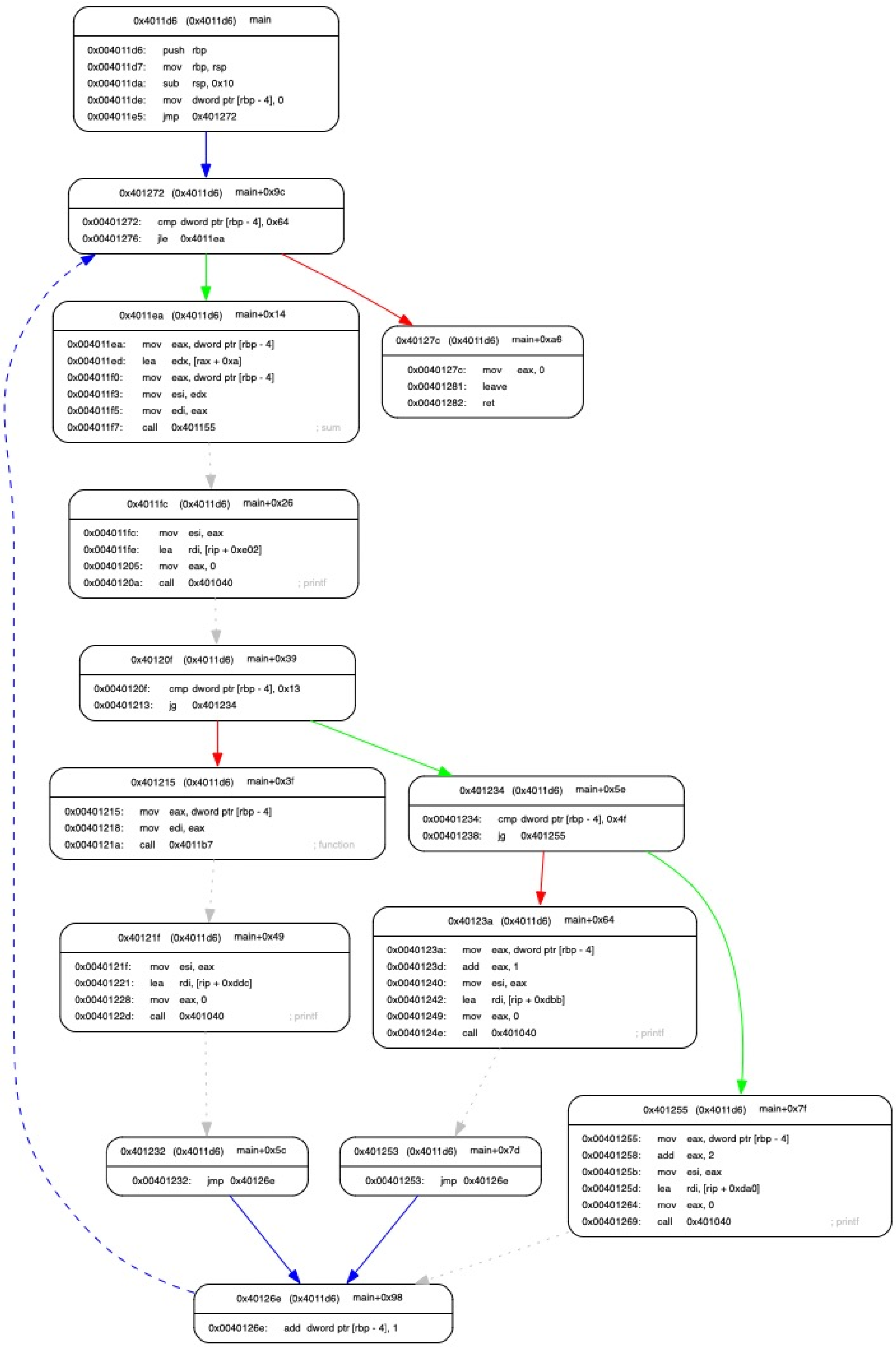
```
Input  : A list of function instructions
Output: A list containing all function basic block
$Leaders = Array$
for $i$ in instructions do
    if $isLeader(i) == True$ then
        append($Leaders$, $i$)
    end
end
$BasicBlocks = Array$
$BasicBlockInstructions = Array$
for $i$ in instructions do
    if $i \in Leaders$ then
        append($BasicBlocks$, $BasicBlockInstructions$)

        $BasicBlockInstructions = newArray$
        append($BasicBlockInstructions$, $i$)
    end
    else
        append($BasicBlockInstructions$, $i$)
    end
end
return $BasicBlocks$
```

3.4.4. Control Flow Graphs

CFG is a directed graph constructed from a set of nodes and edges like seen in Figure 7. CFG has an entry node which is the first node in the function's execution path and an exit node which is a node from which control transits outside the function. Usually, a node in a control flow graph is a basic block and an edge is a transition of control between control flow graph nodes. An edge presents transitions between nodes in the graph, thus creating a connection between nodes in a graph. The edges often consist of jumps and function calls. From the control flow graph relations between basic blocks can be analysed and used for similarity analysis.

There are also other types of graphs that are derived from the CFG. Usually, graphs that are derived from the CFG are CG and data flow graphs (DFG). CG represents call relations between functions and basic blocks in a binary, hence edges in a graph indicate a call instruction. DFG represents data dependencies across actions in a binary. In DFG edges represent data flow between nodes that data transformation functions.



```
0x4011d6   (0x4011d6)   main
0x004011d6:   push  rbp
0x004011d7:   mov   rbp, rsp
0x004011da:   sub   rsp, 0x10
0x004011de:   mov   dword ptr [rbp - 4], 0
0x004011e5:   jmp   0x401272

0x401272  (0x4011d6)   main+0x9c
0x00401272:   cmp  dword ptr [rbp - 4], 0x64
0x00401276:   jle   0x4011ea

0x4011ea  (0x4011d6)   main+0x14
0x004011ea:   mov   eax, dword ptr [rbp - 4]
0x004011ed:   lea   edx, [rax + 0xa]
0x004011f0:   mov   eax, dword ptr [rbp - 4]
0x004011f3:   mov   esi, edx
0x004011f5:   mov   edi, eax
0x004011f7:   call   0x401155              ; sum

0x40127c  (0x4011d6)   main+0xa6
0x0040127c:   mov   eax, 0
0x00401281:   leave
0x00401282:   ret

0x4011fc   (0x4011d6)   main+0x26
0x004011fc:   mov   esi, eax
0x004011fe:   lea   rdi, [rip + 0xe02]
0x00401205:   mov   eax, 0
0x0040120a:   call   0x401040              ; printf

0x40120f   (0x4011d6)   main+0x39
0x0040120f:   cmp  dword ptr [rbp - 4], 0x13
0x00401213:   jg   0x401234

0x401215  (0x4011d6)   main+0x3f
0x00401215:   mov   eax, dword ptr [rbp - 4]
0x00401218:   mov   edi, eax
0x0040121a:   call   0x4011b7              ; function

0x401234  (0x4011d6)   main+0x5e
0x00401234:   cmp  dword ptr [rbp - 4], 0x4f
0x00401238:   jg   0x401255

0x40121f   (0x4011d6)   main+0x49
0x0040121f:   mov   esi, eax
0x00401221:   lea   rdi, [rip + 0xddc]
0x00401228:   mov   eax, 0
0x0040122d:   call   0x401040              ; printf

0x40123a  (0x4011d6)   main+0x64
0x0040123a:   mov   eax, dword ptr [rbp - 4]
0x0040123d:   add   eax, 1
0x00401240:   mov   esi, eax
0x00401242:   lea   rdi, [rip + 0xdbb]
0x00401249:   mov   eax, 0
0x0040124e:   call   0x401040              ; printf

0x401255  (0x4011d6)   main+0x7f
0x00401255:   mov   eax, dword ptr [rbp - 4]
0x00401258:   add   eax, 2
0x0040125b:   mov   esi, eax
0x0040125d:   lea   rdi, [rip + 0xda0]
0x00401264:   mov   eax, 0
0x00401269:   call   0x401040              ; printf

0x401232  (0x4011d6)   main+0x5c
0x00401232:   jmp  0x40126e

0x401253  (0x4011d6)   main+0x7d
0x00401253:   jmp  0x40126e

0x40126e  (0x4011d6)   main+0x98
0x0040126e:   add  dword ptr [rbp - 4], 1
```

Figure 7. A Sample Control Flow Graph

3.5. Features Extraction from Disassembly

In past research, several different types of similarity measures for binary similarity systems have been used and proposed like graph, feature or semantic-based similarity measures. Similarity measure has to be chosen in a way that similarity changes as little as possible between similar samples and as much as possible between dissimilar samples. Hence, features have to be as static as possible and detectable between different compilations.

In this research combination of feature and semantic-based similarity has been chosen for similarity measure. Feature and semantic-based similarity measures provide good performance and flexibility for similarity comparison. As demonstrated by Kim et al. [2] carefully selected features can be a robust way to detect similar code between different compilers and architectures.

Binary similarity analysis purely based on semantic similarity could provide strong clues on the similarity of two binary executables, but comprehensive analysis of the semantic similarity between two binaries requires symbolic execution. Symbolic execution can be used to evaluate accurately similarities between binaries. Even though symbolic execution of binary executable files would give more accurate indications on the semantics of binaries the dynamic execution of functions introduces such high-performance overhead and would not be scalable enough to meet requirements set for the system that it will not be considered in this research.

3.5.1. Feature Granularities

In past research, many different granularities or units of observations for similarity comparison have been proposed from instruction tracalets to program-level similarity [1, 16, 63]. The granularity of observation is considered to be a single unit of measurement used for similarity comparison. The most commonly used granularities for a binary similarity analysis system have been either basic block-level or function-level analysis. In the basic block-level analysis similarity is measured based on the similarity between basic blocks. In function-level granularity, a function is considered a single unit of observation. A function is constructed from one or more basic blocks and it often contains a set of basic blocks that provide certain functionality or perform a certain task.

Function-level analysis significantly reduces the time required for analysis as it combines sets of basic blocks into one unit, but analysis based on basic blocks might be required when analysing differences between revisions of the same function or when increased accuracy for analysis is needed. In general, the basic block-level analysis would give a more accurate look into function or binary structure and the similarities between them, because changes in a function structure would be easier to detect at the basic block level. However, this would significantly downgrade the overall performance of the system, especially in large binaries which might have a huge amount of basic blocks and functions which make the control flow graph and amount of similarity comparisons grow rapidly. The basic block-level analysis would also significantly increase the amount of time spent in the database queries for each examined binary. Therefore, function-level analysis was chosen to be the granularity of similarity measures for the proposed system. Overall, the function-level analysis provides good enough performance without sacrificing too much accuracy in the binary similarity matching process.

The accuracy of function-level analysis could be improved with methods like conditional formulas. Conditional formulas can be extracted and applied to function-level similarity matching to archive a better presentation of the actual semantics of a function across different ISAs [17]. However, conditional formulas do not scale enough for larger binary executables as symbolic execution is required to resolve all function's inputs and outputs, hence conditional formulas are left out of this research.

Other less common units of observation used for similarity measurement like tracelets or in the past research quite popular graph-based similarity that use subgraph isomorphism to detect similarities between binaries can also be used to detect similarities between executables. Graph-based comparison can be based on different types of graphs like CFG, CG or DFG. These methods can be considered basic block-level granularity measurements as the commonly used control flow graphs or call graphs are constructed from basic blocks, but the focus is on the graph itself, not the basic block. These methods are ignored in this research as graph-based analysis is substantially slower than function matching using fuzzy matching techniques as graph matching is known to be an NP-complete problem.

3.5.2. Features Classification

Function features combined into feature vectors are used to represent a function. Feature vectors are created using a comprehensive analysis of function structure and instructions. Function analysis contains an analysis of different function attributes and relationships between basic blocks and other functions. The resulting feature vector is a collection of different features and feature types.

Features used in the proposed system are classified into two distinct feature types or categories like shown in the research conducted by Haq et al. [1].

Structural Features capture the structure of a function. These features are derived from control flow graphs and other similar graphs of a function. Structural features leverage from relations observed between functions or basic blocks observed in a graph.

Statistical Features capture the statistical distribution of function mnemonics or other quantifiable features. These features capture the amount and distribution of instructions groups in a function.

The first feature category is structural features which include structural features of code sections like the number of loops or arguments passed to a function. This category is chosen as the base structure of a function usually stays relatively static across different compilations and architectures. Structural features also utilise the CFG and capture information about relationships between nodes in a function. This gives robust information about the relation between basic blocks in a function that is used for similarity matching. However, sub-graph isomorphism [64] makes systems that rely purely on structural features relatively slow. Statistical features capture the distribution of different quantifiable values in a function like mnemonic groups. Statistical distribution of mnemonic groups of instructions can be generally used in the features selected to represent a function or semantics of a function like described by Alrabaee et. al [65].

Both structural and statistical features require normalization to be effective in a system targeting one or more ISAs. Mnemonics of a function are often altered between different compilations of the same open source library, which causes the importance of these features to be reduced if the system targets more than one architecture.

After careful examination of the literature and binary disassembly of similar and dissimilar functions, 36 features are chosen to represent a function. In Table 3 are listed all feature names, weights and categories that are extracted for each function in a binary. Most of the chosen features are statistical features. This is due to the low contribution of one individual statistical feature that can be observed from the low feature weights.

The feature set is further optimised during the machine learning process by observing the contribution of each feature to the final distance between predefined and labelled function pairs. Features that are observed to have zero or near zero weight relative to other feature weight values during the weight optimisation phase are removed from the feature vector. These features tend to have minimal impact on the final similarity. Features with really low feature weights will position all function pairs closer to each other regardless of their actual observed similarity which will increase false positive rates in the similarity matching.

3.6. Feature Enhancements

Function features give a robust representation of a function, however, different features have different impacts on the overall similarity of executable functions. This is caused by compilers and different ISAs that might change the structure of functions or code sections between different compilations of the same source code. To overcome this enhancements and optimisations for function features are required. Function feature enhancements are methods that are applied to functions or function features to mitigate or decrease changes affecting similarity across different compilations of libraries. This makes the matching more robust regardless of compiler, architecture or compiler options used in the compilation process.

3.6.1. Feature Weights

Features extracted from a binary do not have the same importance and they contribute different amounts to the similarity or dissimilarity of function pairs [24, 14]. Some features are more resilient across different compilations, thus they contribute more to the similarity of functions compared to features that change more when the compilation process is changed. Therefore, a unique weight for each feature is needed to combat these differences between binaries compiled with different compilers and compiler flags.

Table 3. Function Feature Names, Feature Categories and Feature Weights

Feature Name	**Structural**	**Statistical**	**Feature Weight**
# Arithmetic Instructions		X	0.0085
Average Instructions Per Block		X	0.0027
# Bit Operation Instructions		X	0.0019
# Call Instructions		X	0.0126
# Code Constants		X	0.0067
# Comparison Instructions		X	0.0090
# Data Transfer Instructions		X	0.0110
# External Function Calls		X	0.0127
# Floating Point Instructions		X	0.0015
# Function Called		X	0.0054
# Indirect Control Transitions		X	0.0106
# Instructions		X	0.0101
# Internal Function Calls		X	0.0054
# Jump Instructions		X	0.0011
Kurtosis		X	0.0207
# Load Instructions		X	0.0082
# Local Variable		X	0.0136
Max Instructions Per Block		X	0.0024
Mean		X	0.0099
Skewness		X	0.0093
# Stack Instructions		X	0.0069
Standard Deviation		X	0.0109
# Store Instructions		X	0.0005
# String Instructions		X	0.0012
Variance		X	0.0021
Z-Score		X	0.0200
Adjacency Matrix Size	X		0.0028
# CFG Nodes	X		0.0065
Betweenness	X		0.0400
CFG Edges	X		0.0089
Cyclomatic Complexity	X		0.0097
# Function Arguments	X		0.0056
Longest Path	X		0.0073
# Loops	X		0.0054
Shortest Path	X		0.0144
Stack Depth	X		0.0018

Features with a higher weight typically contribute more to the final similarity. Feature weights should also further accede hamming distance of similar function pairs and recede hamming distance of dissimilar function pairs. This is used to optimise feature weight in such a way that similar functions have a smaller hamming distance between them than dissimilar function pairs. The weight of each feature is optimised using offline machine learning process [24, 5]. This allows feature weights to be optimised in such a way that they are usable for all possible functions.

3.6.2. Feature Weights Optimisations

Feature weight optimisation is a computationally expensive operation, but it improves the matching precision significantly. In the proposed system machine learning process is setup in such a way that feature weights are optimised in a separated offline process from the binary similarity analysis. Therefore, the complexity of the machine learning process does not affect the matching process. Many different machine learning algorithms have been proposed for similar tasks in the literature. These algorithms utilise different techniques to find optimal values for features in feature vectors. From the available algorithm one that produces the most optimal feature weight has to be chosen.

One major limitation of machine learning systems like this is that if any aspect of the matching system has changed for example one feature is removed or a bug in the system has been fixed existing feature weights are rendered obsolete. So the weight training has to be redone each time the system is revised to achieve optimal matching accuracy.

Machine learning algorithms compared for the feature weight optimisation task usually consume labelled data. Consumed data have different types of function pairs, but usually they are categorised into similar and dissimilar function pairs. These function pairs are extracted from commonly used open source libraries. Libraries are chosen from different categories like networking or RTOS to ensure that feature weights are optimal for as many functions as possible regardless of the function's intended use. Libraries and their respective categories are visible in Table 4.

Sample libraries are compiled from the latest version available at the time of writing using two different compilers GCC and Clang and cross-compiling with GCC to ARM64. To increase variation in the binary sample set and to make detection more robust compiler optimisation levels from 0 to 3 are used for each compiler.

Sample matrix produced from the chosen set of libraries contained about 10000 unique function pairs. The sample matrix is divided into a training set and an evaluation set which contained 9000 and 1000 function pairs respectively. Function pairs in both training and evaluation sets are labelled as similar or dissimilar based on the source binary and function names. These labelled function pairs are passed to the machine learning algorithm for the feature weight optimisation process.

The machine learning algorithm has to optimise feature weights in such a way that similar function pairs produce a lower hamming distance between the simhashes than the dissimilar function pairs as it would eventually yield a higher set similarity score by growing set intersection between similar binary samples. In prior research, many different algorithms have been proposed for this task. Many linear models have been utilised like stochastic gradient descent (SGD) and algorithms derived from it [24]. Furthermore, different feature selection algorithms like Relief are proposed [5]. Inspired by these researches several linear and feature selection algorithms are tested and evaluated for the feature weight optimisation task.

Table 4. Open source libraries used in feature weight optimisations task

Library Name	**Library Category**
Alsa-lib	Audio
Portaudio	Audio
Gzip	Compression
Zlib	Compression
OpenSSL	Cryptography
WolfSSL	Cryptography
Evemu	I/O
Libuv	I/O
Libpng	Image
LibTIFF	Image
Blaspp	Mathematical
Libmemory	Memory
Asio	Networking
cURL	Networking
Libpcap	Networking
Nanomsg	Networking
Nghttp2	Networking
PicoTCP	Networking
cJSON	Parser
PicoHttpParser	Parser
D-Bus	Protocol
EmbeddedProto	Protocol
Nanopb	Protocol
Contiki	RTOS
Riot-OS	RTOS
ICU	String
SDS	String manipulation
BusyBox	Utility
JerryScript	Virtual machine
TinyVM	Virtual machine

Table 5. Machine learning algorithms compared for feature weight optimisation

Algorithm Name	**Classification**
Relief	Feature selection
ReliefF	Feature selection
MultiSURF*	Feature selection
Huber Regression	Linear Regression model
Stochastic Gradient Decent	Linear Regression model

Inspired by these researches, machine learning algorithms chosen for the evaluation are Huber regression [66] and SGD [67] which are linear models. From the available feature selection algorithms ones chosen for comparison are Relief [68] and ReliefF [69] algorithms and other variants of the Relief family of algorithms like MultiSURF* [70]. All machine learning algorithms and their classification that are considered for this research are visible in Table 5.

Feature selection algorithms are commonly used in the field of machine learning to reduce the amount of features in a high-dimensional dataset to scale down the dimensionality of classification problems. Usually feature selection algorithms required ranking or weighting of each feature in a dataset to outline the most important features. This feature is leveraged for feature weight optimisation. In machine learning systems linear models are often used to make predictions or estimates on data based on coefficients of values seen in the training dataset. These coefficients can then be used in feature weights as they have been fitted with a linear function based on the training set to estimate similar data.

Feature weight optimisation machine learning process is shown in Figure 8. Machine learning algorithms are compared against each other to evaluate the performance of each one in a feature weight optimisation task. The performance of a machine learning algorithm is determined by computing the average hamming distance between dissimilar and similar function pairs by creating simhashes from the function pairs in the evaluation set. A machine learning algorithm that produces feature weights that achieve the lowest average hamming distance between similar function pairs and the highest average distance between dissimilar function pairs is chosen for the feature weight optimisation task.

During the training process algorithms are guided in the correct direction by observing the hamming distance between function pairs. Dissimilar function pairs should recede and similar function pairs should accede to ensure the correctness of machine learning results otherwise resulting weights are ignored. Similarity score over sets of simhashes extracted from two different executable files was also considered as a performance metric for the machine learning algorithms, but this proved to be too vague for the machine learning process. Set similarity scores would mask minor changes in learning process performance evaluation, thus making performance evaluation less precis. Optimising weights over a hamming distance of function pairs yields better results as set similarity methods are applied to sets constructed based on set intersections determined by hamming distances. Therefore, optimising hamming distances improves similarity scores for similar executable binaries and is used to more accurately guide the machine learning process.

Dullien [24] proposed that the simhash values before the final feature value hash combination step could be used as feature values in a machine learning pipeline. After comparing both available options for feature weight optimisation feature values without hashing proved to have better performance and accuracy over the hashed feature values as they retained their distance to each other better during the training process compared to hashed feature values.

From the machine learning algorithm options compared for the feature weight optimisation task ReliefF variant of the Relief feature selection algorithm is chosen to compute the importance and weights of each feature as it produced the best performance on average hamming distance for samples in the dataset. ReliefF

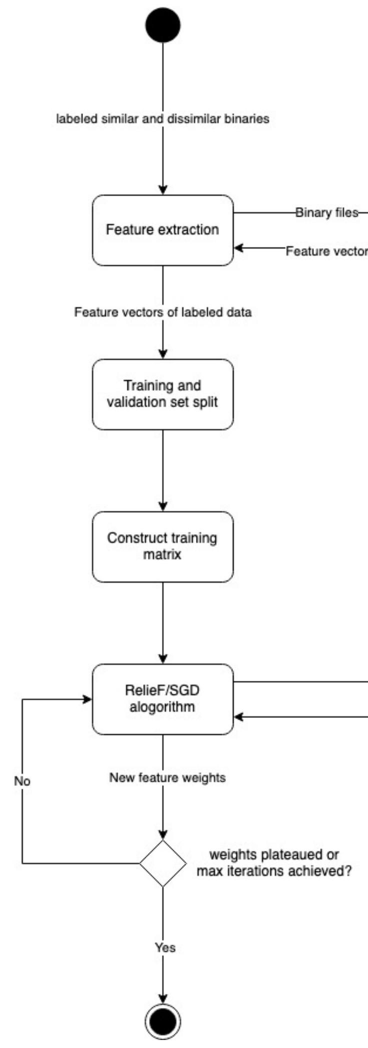

labeled similar and dissimilar binaries
Feature extraction
Binary files
Feature vectors
Feature Analyzer
Feature vectors of labeled data
Training and validation set split
function vector pair
feature weights
prehashed function values
Construct training matrix
RelieF/SGD alogorithm
No
New feature weights
weights plateaued or max iterations achieved?
Yes

Figure 8. Machine learning pipeline for feature weight optimisation task

produced the lowest hamming distance for similar function pairs and the highest hamming distance for dissimilar samples. The final feature weights produced by ReliefF machine learning algorithms for each feature are shown in Table 3.

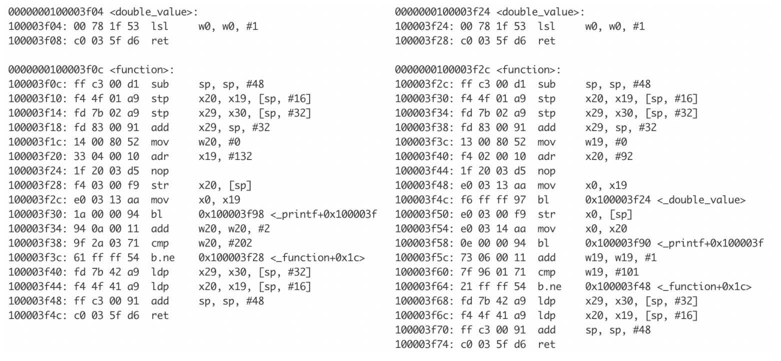
It was observed during the feature weight training process that the feature selection algorithms were overall faster to achieve the same or better average hamming distance between similar and dissimilar samples than linear models like SGD. The average hamming distance between similar and dissimilar function pairs achieved by the ReliefF algorithm was 11.4 after 1 iteration. The average hamming distance achieved by SGD between the pairs was 11.8 after 83 successful iterations. An iteration of the feature weight machine learning process is considered as successful when produced weights produced better hamming distance compared to the previous iteration. Due to the iterative nature of linear models, it is possible if given enough time and iterations linear models could produce better results than feature selection algorithms.

3.6.3. Function Inlining and Function Exclusion

Instructions used to perform a certain task are often optimised several different ways by compilers or by the developers themselves to reduce overhead during execution of instructions. This might lead into situations where functions are split into one or more sub-functions or separate functions are combined into one larger function. Combination of functions happens especially when greater compiler optimisations are used during the compilation process. Due to this, in certain situations function disassembly without further processing does not yield good enough presentation of the actual propose of the function. Hence, post-processing for functions that are observed to be split into several parts or combined together have to be done to ensure accurate comparison of function similarity.

In some instances, compilers reduce the overhead caused by function calls by inlining a callee function into the body of the caller function like seen in Figure 9 where the sample on the left has a higher optimisation level compared to the sample on the right. To improve the accuracy of matching some function in a binary has to be excluded completely from the matching process. In addition, some functions have to be included in the body of other functions to maintain the structural similarity of the function across different compilations of the function.

In function inlining, a body of the callee function is inserted into the caller function body and the function call is removed to reduce the amount of runtime overhead by removing a need for target function resolving and stack manipulation. However, in certain instances the caller function is transformed into a semantic clone of the callee function. To avoid this some heuristics have to be implemented to avoid blind function inlining.



```
0000000100003f04 <double_value>:
100003f04: 00 78 1f 53  lsl     w0, w0, #1
100003f08: c0 03 5f d6  ret

0000000100003f0c <function>:
100003f0c: ff c3 00 d1  sub     sp, sp, #48
100003f10: f4 4f 01 a9  stp     x20, x19, [sp, #16]
100003f14: fd 7b 02 a9  stp     x29, x30, [sp, #32]
100003f18: fd 83 00 91  add     x29, sp, #32
100003f1c: 14 00 80 52  mov     w20, #0
100003f20: 33 04 00 10  adr     x19, #132
100003f24: 1f 20 03 d5  nop
100003f28: f4 03 00 f9  str     x20, [sp]
100003f2c: e0 03 13 aa  mov     x0, x19
100003f30: 1a 00 00 94  bl      0x100003f98 <_printf+0x100003f
100003f34: 94 0a 00 11  add     w20, w20, #2
100003f38: 9f 2a 03 71  cmp     w20, #202
100003f3c: 61 ff ff 54  b.ne    0x100003f28 <_function+0x1c>
100003f40: fd 7b 42 a9  ldp     x29, x30, [sp, #32]
100003f44: f4 4f 41 a9  ldp     x20, x19, [sp, #16]
100003f48: ff c3 00 91  add     sp, sp, #48
100003f4c: c0 03 5f d6  ret
```

```
0000000100003f24 <double_value>:
100003f24: 00 78 1f 53  lsl     w0, w0, #1
100003f28: c0 03 5f d6  ret

0000000100003f2c <function>:
100003f2c: ff c3 00 d1  sub     sp, sp, #48
100003f30: f4 4f 01 a9  stp     x20, x19, [sp, #16]
100003f34: fd 7b 02 a9  stp     x29, x30, [sp, #32]
100003f38: fd 83 00 91  add     x29, sp, #32
100003f3c: 13 00 80 52  mov     w19, #0
100003f40: f4 02 00 10  adr     x20, #92
100003f44: 1f 20 03 d5  nop
100003f48: e0 03 13 aa  mov     x0, x19
100003f4c: f6 ff ff 97  bl      0x100003f24 <_double_value>
100003f50: e0 03 00 f9  str     x0, [sp]
100003f54: e0 03 14 aa  mov     x0, x20
100003f58: 0e 00 00 94  bl      0x100003f90 <_printf+0x100003f
100003f5c: 73 06 00 11  add     w19, w19, #1
100003f60: 7f 96 01 71  cmp     w19, #101
100003f64: 21 ff ff 54  b.ne    0x100003f48 <_function+0x1c>
100003f68: fd 7b 42 a9  ldp     x29, x30, [sp, #32]
100003f6c: f4 4f 41 a9  ldp     x20, x19, [sp, #16]
100003f70: ff c3 00 91  add     sp, sp, #48
100003f74: c0 03 5f d6  ret
```

Figure 9. Sample of function inlining in a function call

Inspired by past research [38, 19, 71] selective function inlining is implemented to improve the accuracy of matching across different executable optimisation levels. To keep the semantic similarity of functions, inlining is done selectively for functions that match Equation 4 and have a function in degree out degree ratio over the suggested threshold 0.01. This computes the ratio of CFG edges leaving the function to total CFG edges leaving and incoming to the function. Unlike Chandramohan et al. [71] proposed in their research that selective inlining is performed also for standard library functions, in this research selective function inlining is done only for user-defined functions. C/C++ standard library functions are ignored from function inlining to eliminate conversion of semantically and structurally dissimilar functions to more semantically and structurally similar counterparts. This would happen especially when the inlined C/C++ standard library functions are similarly sized or much bigger than the function they are inlined into.

$$ratio = \frac{outdegree_c}{outdegree_c + indegree_c} \tag{4}$$

Like demonstrated by Ding et al. [19] to avoid making semantically dissimilar functions appear more similar by inlining the callee function body into the caller function that has significantly fewer instructions than the callee function, inlining is only done when the caller function is large enough compared to the callee. To achieve this caller-to-callee function instructions ratio must be over 60 percent. Also, caller functions that have less than 10 instructions to include are assumed to be wrapper functions and they are included in the callee expansion process even if they do not meet other requirements. This avoids the inclusion of a larger function into smaller a function unless they are deemed to be wrapper functions that just call other functions with certain parameters. In these situations, callee expansion is desired even if the caller has fewer instructions than the callee.

There are several other function boundary obfuscation methods applied to function instructions like function outlining in which sections of a function are replaced with a function call or interleaving functions by merging function bodies into one or cloning functions [72]. However, these techniques are ignored in this research as they are not as common as function inlining and they affect the resulting similarity of function pairs less. These methods are also significantly harder to detect without full graph-based analysis of a function that introduces major computational complexity and thus decreases the performance of the system.

Although, features chosen to represent a function give a good overview of the function's semantics, there are still functions which have very little identifiable material. To prevent saturation of hash sets with function hashes that are similar to the C/C++ standard library functions some functions need to be excluded from the similarity comparison. Excluded functions have either very little identifiable material or are named similarly to standard library functions. Functions which meet these criteria are omitted from the matching process. Usually, standard C/C++ library functions are relatively static across different libraries and library versions to prevent API breaks, thus these functions yield more false positives in the matching. From a set of extracted functions ones with too few instructions or have too many instructions are filtered out based on predefined threshold values.

3.7. System Overview

The proposed binary similarity matching process can be divided into five distinct tasks and two sub-tasks or parallel tasks which are performed during the main task, but are notable enough to be mentioned separately. These tasks and sub-tasks are represented in Figure 10. These tasks are performed sequentially, once the previous step is finished data produced by the current step is passed to the next task for further analysis and processing.

The first task in the matching process is binary executable processing and metadata extraction. In this step function metadata which includes function offsets, names and relocation information is extracted from ELF executable files. Furthermore, from the ELF header information about ISA used by the executable is extracted to allow instructions to be disassembled correctly later on the process. Alongside function metadata and ISA, executable code section bytes are extracted and passed for the next step in the process which is disassembly analysis.

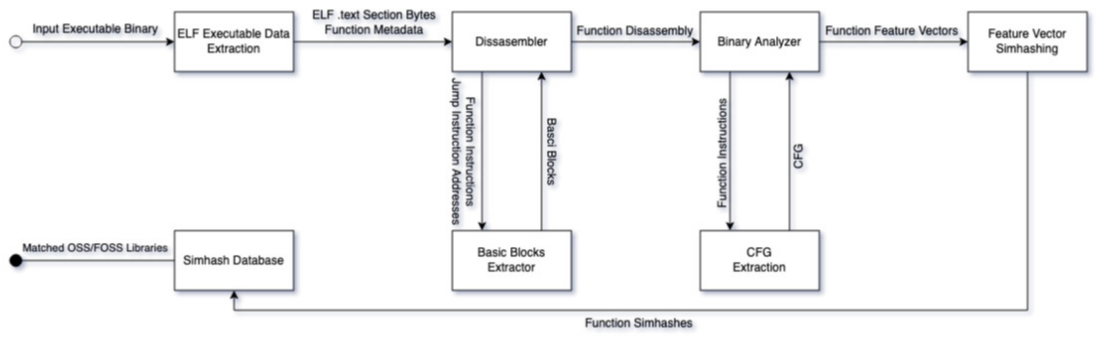


Figure 10. Proposed binary similarity matching process

In the second task of the process disassembly analysis task where code section bytes are lifted and normalised into the IR from code section bytes extracted in the previous task. The lifted and normalised instructions are then grouped into functions and basic blocks for further analysis. This is done with two linear sweeps over IR instructions for each binary code section extracted. The first sweep over instructions collects target addresses of all jump and branch instructions for the second sweep. During the first sweep code section instructions are also grouped into corresponding functions. The second sweep over the instructions creates basic blocks based on the algorithm shown in 2 while utilising addresses collected in the first sweep.

Once the second sweep over instructions is completed, functions are inlined based on functions' in and out degree and function size. Also, some functions are excluded from the binary similarity matching process if they are deemed to yield inaccurate results by being too small or large. After this extracted inlined functions and basic blocks are passed to the next step in the process.

In the third task feature vectors and CFG are created by analysing function instructions and basic blocks extracted in the previous step. Feature vectors are constructed for each function by analysing function structure, instruction distribution in the function and the function's structural features derived from the CFG.

In the fourth task, LSHs are created from feature vectors using feature weights produced by the offline machine learning task. Feature vectors are converted into the LSH using the process seen in the 3. LSH transforms a higher dimensional feature vector into a lower dimensional hash value that improves matching performance. Moreover, LSH ensures fuzziness in the matching process by allowing minor changes to occur in the feature vectors before resulting in hash value changes. LSH chosen for this research is the simhash [48]. Simhash is chosen as it provides a good performance in the set similarity matching tasks.

The fifth and final step in the binary similarity matching process is to match function simhashes extracted from a binary to the database of known hashes extracted from open source libraries. This is achieved by comparing the hamming distance between extracted simhashes and simhashes in the database belonging to different open source libraries. Hash pairs that have a hamming distance less than the chosen threshold 8 are chosen into the final set similarity comparison that computes the similarity value between hash sets. Symmetric Tversky index is used as the set similarity score with α value 0.2 and β value 0.8. Binary is deemed to be similar to a known open source library if the set similarity score between extracted hashes and known hashes is more than the predefined set similarity threshold value for a given open source library. The set similarity threshold value is different from the hamming distance threshold value as it defines the percentage value of how similar hash sets have to be.

3.8. Performance Optimisations

The performance of the matching system is one of the key aspects to consider when the system is applied to real-world scenarios like continuous integration pipelines of software supply chains. This sets hard requirements for the system's performance. For example, the system has to perform similarity matching for any given binary as fast as possible to reduce developers' downtime and overall time spent in the executable verification steps. In the literature, many different solutions have been proposed to improve the performance of binary similarity matching systems. However, these proposed performance optimisations are often very specific for the system in question. This often limits the generalisation of performance optimisation measures to other systems directly.

To address the performance issues we implemented our own executable analysis platform from a scratch. Our analysis platform performs only the analysis steps required by the proposed system for the executable. This reduces overhead caused by analysis tasks that are not necessary to evaluate the similarity of functions in the way proposed in this research. Furthermore, we have evaluated and detected the most performance-critical tasks in the matching system. We have decided to optimise the proposed system in these areas and tasks further. The most performance-critical tasks in the binary similarity analysis platform are simhash matching task, feature extraction and CFG creation.

3.8.1. Matching Optimisations

One of the biggest obstacles for systems that rely on a static set of signatures constructed from features, embeddings or hashes extracted from a binary is database queries when performing similarity matching. Queries are often linear searches of the entire signature database unless optimisations for database queries are done to reduce the number of queries for each matched library. This is achieved by reducing the amount of potential match candidates retrieved from the database. Alternatively, the database structure can be optimised in such a way that signature retrieval is performed more efficiently. In the literature, several solutions to solve similar issues have been proposed from splitting hash values into chunks [5] for faster hash comparison or nearest neighbour search [24] has been proposed. Another typical method for locality-sensitise hash algorithms is to construct hash buckets by computing N permutation of a hash. This provides a high probability that hashes belonging to the same bucket are similar and limits the amount of comparison needed. In this research to reduce the number of match candidates by utilising hash buckets is chosen. It is more convenient when using LSH as they tend to cluster similar hash values. Moreover, complex database structures often take a relatively large amount of time to create at the launch of the system which makes them undesirable for the proposed system.

To reduce the amount of required similarity score computation hash buckets of similar hashes are used. Hash values are stored in a bucket of N values by computing M permutations of each hash value bits depending on the chosen hamming distance threshold value. This is done during the database initialization phase therefore it is done only once over the life cycle of a matching system. Due to the design of LSH and simhash, similar function hashes have a high probability to end up in the same hash bucket during the matching phase. By leveraging this feature of LSH match candidates for each database query can be reduced significantly. Correct hash buckets based on permutations of input hash values are computed for each hash in the set of hashes extracted from the binary during the matching and only hashes in those hash buckets are taken into set similarity comparison. From these reduced hash sets similarity scores between compared binary and known open source libraries are computed. This speeds up the matching by reducing the time consumed on the worst case for linear search from $O(N^2)$ to $O(M^2)$ where N is the amount of hashes in the database and M is the size of a bucket. Matching function hashes extracted from binaries to the known database of hashes is a many-to-many problem. The complexity of this problem will increase when larger binaries are matched against the database of hashes.

3.8.2. Feature and Graph Extraction Optimisations

Structural features leveraging CFGs or ICFGs are often more expensive to extract compared to statistical features. Therefore, the extraction of structural features requires further optimisations. ICFGs would provide more context compared to the CFGs, but they would be computationally more complex. Hence, features extracted from ICFG are ignored in this research. Reduces the complexity of control flow graphs makes relationships between graph nodes inside functions faster to resolve.

Another option to improve the performance of CFG creation and instruction parsing is to reduce graphs into sets of key instructions and non-key instructions. These sets of key instructions can be then further classified into instructions that affect CFG and instructions that do not affect the graph [73]. Systems that perform symbolic execution would typically lose matching performance, but gain matching precision. Due to the requirement for computationally complex symbolic execution, these types of methods are ignored in this research.

4. EVALUATION

The accuracy and effectiveness of the proposed binary similarity matching system can be measured in several different ways. From the available options for evaluating the accuracy and effectiveness of the proposed binary similarity matching method, three different evaluation categories are chosen.

1. Function-to-Function similarity
2. File-to-File Similarity
3. Matching Threshold Comparison

These methods allow us to evaluate the accuracy of the system from single function pair matching to the comparison of a file-to-file matching accuracy across different matching thresholds. By utilising these evaluation methods each step in the binary similarity analysis process is evaluated individually.

Performance and accuracy evaluation of the proposed system could be further expanded into hamming distance threshold value comparison. However, in this research threshold value 8 is chosen by manual inspection of similarities across different binaries and based on past research [5, 24].

Matching time performance and scalability of the system are evaluated by collecting matching times for differently sized binary samples. With this effects of binary size on the matching times are evaluated. Moreover, the scalability of the proposed system is evaluated by comparing matching times when the size of the simhash database is changed. This is a key aspect of systems that require a large amount of supported library and comparably fast matching times.

4.1. Function-To-Function Similarity

To evaluate the accuracy and effectiveness of the feature weight optimisation task, distances between function simhashes are compared. This allows the evaluation of function-to-function similarity clustering characteristics. This is achieved by comparing distances between function simhashes created using feature vectors with and without feature weights. Function pairs compiled from the same source code using different compilers and compiler flags should have less distance between them compared to arbitrarily chosen dissimilar function pairs. Therefore, similar functions should create a cluster of functions.

Clustering characteristics of similar functions using feature weight 1 for every feature in the feature vectors are used as a baseline for the function-to-function evaluation. Using feature weights observed during the optimisation task compared to hash samples with feature weight 1 are used to evaluate the performance of function-to-function similarity. Hamming distances between each function hash in the data set are measured. Function simhashes are compared against each other which creates a one-to-one mapping between each data point. Therefore, to make data points easily comparable against each other, data points are scaled from 1-dimensional space to 2-dimensional space using Multidimensional scaling (MDS). During the scaling process

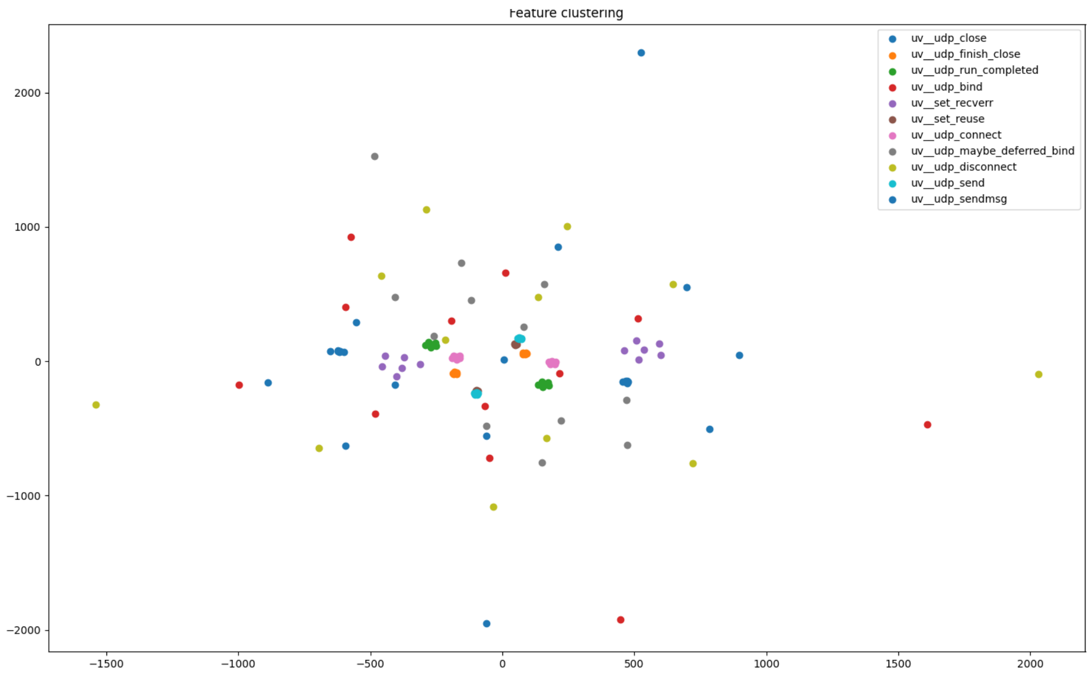


Figure 11. Similar function clustering without feature weights

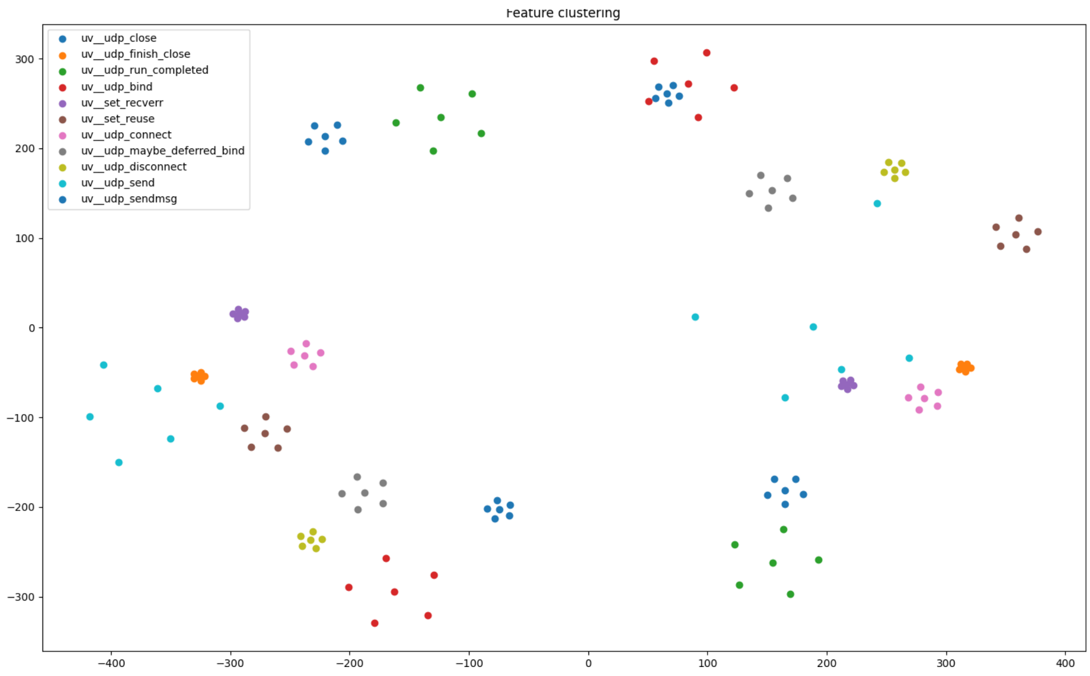


Figure 12. Similar function clustering with feature weights

relations between each hash value are observed. Based on the hash value relations 2 dimensional coordinates are created.

In Figure 11 are shown the clustering characteristics of functions with weight 1 for every feature in the feature vector. It can be observed that the functions do not create clear clusters or groups of clusters of similar functions. This could significantly increase false positive rates during matching as the function groups do not have clear separations between them. This would increase the possibility of dissimilar function hashes ending up in the same hash bucket.

In Figure 12 clustering characteristics of functions with optimised feature weights are shown. With the optimised feature weights similar functions create distinct clusters of similar functions. With the optimised feature weights similar functions cluster closer together than without feature weights. Therefore, hamming distances between similar function pairs are smaller compared to dissimilar function pairs, which makes the false positive rate overall lower for weighted function features.

Functions that are structurally or semantically similar tend to group closer to each other than functions that are semantically or structurally different. This makes semantically similar functions produce simhash values that have a smaller hamming distance between them. Functions imported from standard libraries could be the main cause for the false positive matches during the similarity scoring step. Even if standard library functions are excluded from the implemented function inlining process might be included in a function body by a compiler, especially at higher optimizations levels. This standard library inlining may cause similarities between otherwise dissimilar function pairs, thus introducing some false positive function match pairs.

4.2. File-To-File Similarity

The end-to-end matching accuracy of the proposed binary similarity method is measured by observing file-to-file similarity scores. File-to-file similarity comparison emulates a typical use case for a system like this, where one compilation of open source library is added into the database and matched against other similar and dissimilar binary compilations. For this comparison, 6 different open source libraries are chosen for similarity comparison which produces 36 different file-to-file matching pairs. Between all file pairs similarity scores are measured. This measures how similar the proposed method considers two executable files to be.

In Figure 13 similarity scores between different open source library samples compiled from common open source libraries are shown. Observed similarity scores seem relatively high for all samples. This indicates that the features selected do not represent functions in a binary with enough precision or that functions have similarities between them even if they are labelled as dissimilar. Furthermore, binary file samples or functions have a relatively large amount of standard library functions included in the binary file samples. Standard library functions that are inlined into a function body make functions seem more similar even if they are semantically dissimilar. This is typically caused by several different reasons. Our understanding of the reasons for this in the case of the proposed method is the inclusion of standard library functions into a set similarity computation and too permissive inclusion of functions into the compared functions sets. Small functions that have very few instructions often produce similar feature vectors to each other even if the semantics of those functions are completely different. This can be solved with more aggressive functions filtering or even with function outlining to exclude standard library functions from binaries complied with higher optimisation levels by separating sub-functions from the function body into their own functions. However, these improvements for matching precision are left for future research.

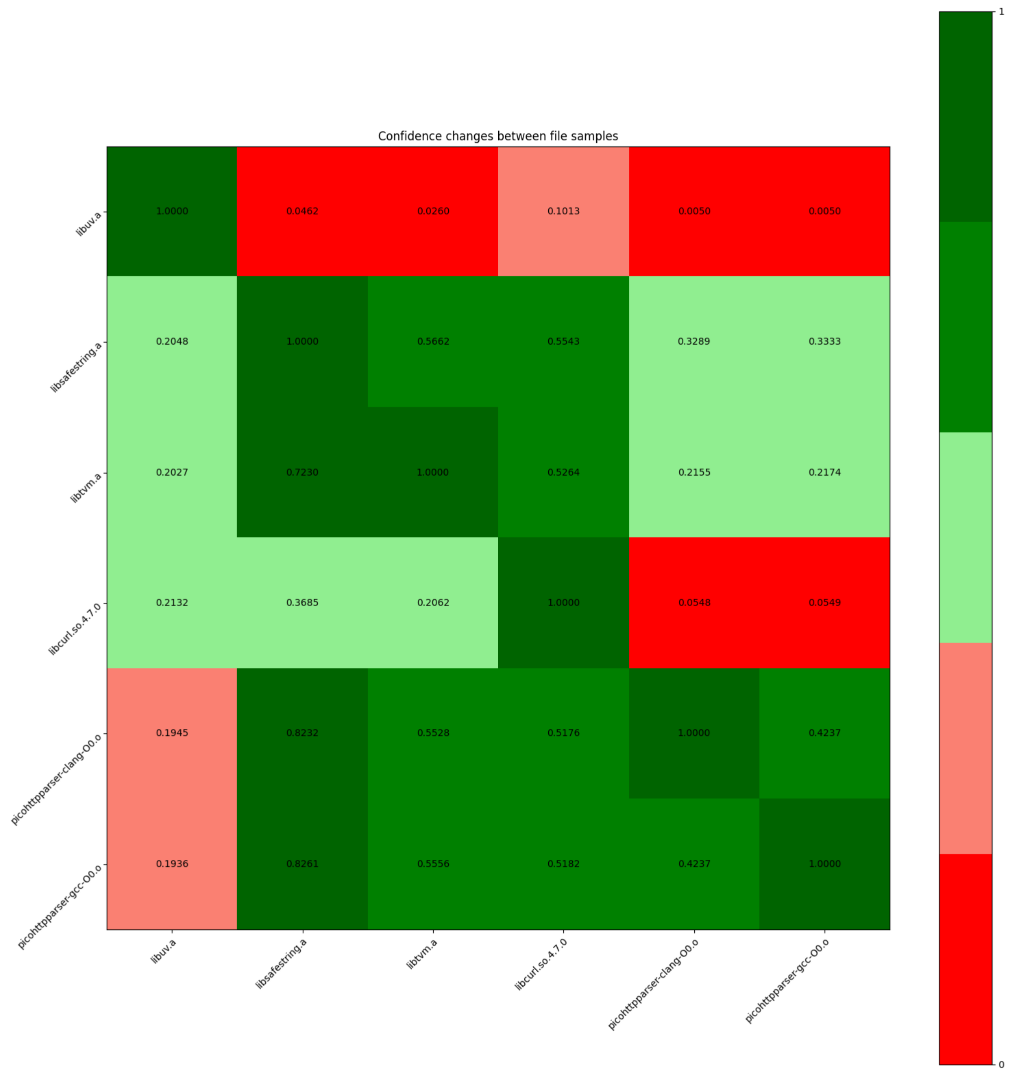


Figure 13. Binary similarity metric scores using Tversky Index

4.3. Matching Threshold Comparison

The general performance of the proposed method is evaluated by compiling a set of library samples using different compilers and compiler optimisations from different open source libraries available and comparing resulting similarity scores when the matching threshold is changed. Resulting set similarity scores across labelled binary samples are used to determine the truthfulness of predicted results. In the binary sample set binaries compiled from the same source code and libraries that have dependencies on other libraries in the data set are labelled to be similar otherwise they are labelled as dissimilar.

The performance of the system or method is measured by collecting information on a confusion matrix that contains total amounts of true positive (TP), false positive (FP), false negative (FN) and true negative (TN) matches. From derivatives of the confusion matrix performance of classifiers can be easily evaluated. Measures chosen to evaluate the performance of the proposed system are the receiver operating characteristic (ROC)

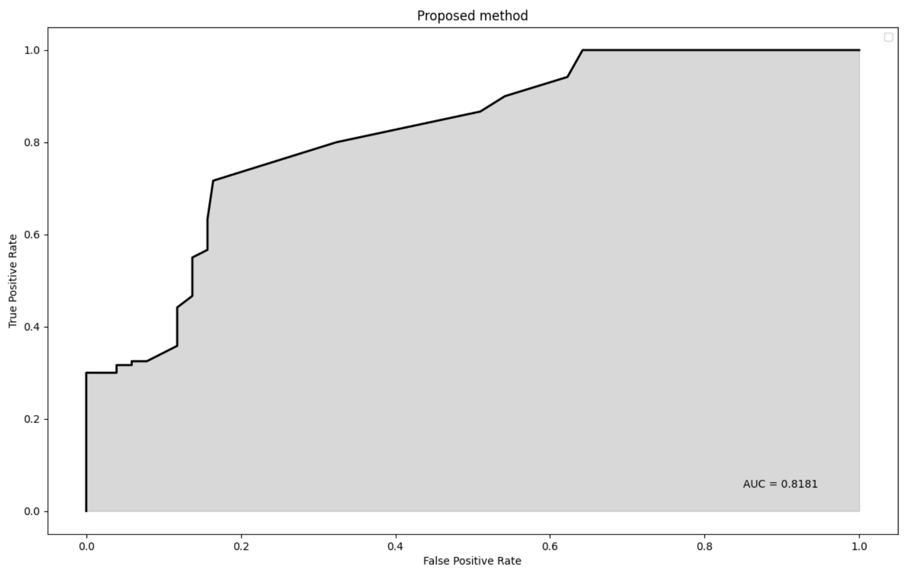


Figure 14. ROC curve and AUC achieved with a small hand-picked sample set

curve and area under the curve (AUC) [74]. These measures are chosen as they provide results that are as comparable to other solutions as possible. ROC curve is a derivative of the confusion matrix and it is used for many different applications, but in prediction and classification tasks it is often to measure the TP rate or recall and FP rate. AUC can be derived from the ROC curve and it is used to measure the performance of a classifier in a classification task, where the AUC indicates the probability of a classifier making the correct prediction. In the case of the proposed method, this means correct libraries are matched from arbitrary binary executables and libraries are not matched from binaries where they are not present. In other words, the proposed method should have a higher match ratio to libraries present in the binary in samples compared to the matching ratio to libraries that are not present in the binary when the similarity matching threshold is increased iteratively.

Figure 14 represents the ROC curve and AUC achieved by the proposed method using a small hand-picked sample set. The small hand-picked sample set contains mostly libraries included in the machine learning data set. Source code of libraries in the small hand-picked data set are compiled using GCC and Clang compilers using optimisation levels from O0 to O3. This produces in total 8 unique similar binary samples for each library. Libraries used in the hand-picked small sample set are shown in Table 6. The proposed method achieved an AUC of 0.8181 which indicated that around 82 percent of the time the proposed binary similarity matching method can correctly classify positive (similar) and negative (dissimilar) samples. The proposed method achieved comparable recall and specificity rates to other solutions that do not rely on machine learning for binary similarity, but it loses some accuracy compared to most machine learning aided solutions. However, as the amount of samples used in the hand-selected sample set is rather small larger more exhaustive analysis is needed.

Table 6. Open source libraries used in evaluation sample set

Library Name	Category
zlib	Compression
mbed TLS	Cryptography
OpenSSL	Cryptography
wolfSSL	Cryptography
libuv	I/O
Nanomsg	Networking
cURL	Networking
cJSON	Parser
safestringlib	String parsing

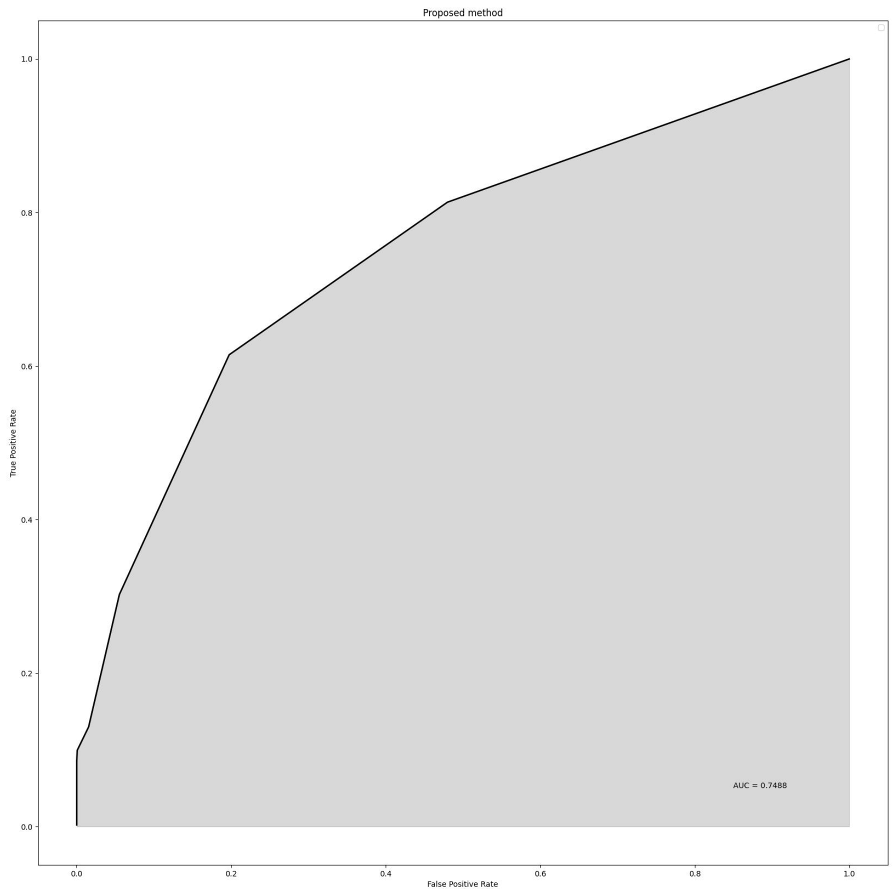


Figure 15. ROC curve and AUC of proposed method with larger sample set

Figure 15 shows ROC and AUC achieved by the proposed method with the larger sample set containing 5671 binary pairs. Binary pairs contained in the larger sample set are common C/C++ open source libraries from the same library categories as listed in Table 6. Samples in the larger sample set are compiled using Clang version 7.0.0 and GCC version 8.3.0. Furthermore, libraries are cross-compiled into ARM 32-bit using GCC version 10. Optimisation levels used for binaries in the larger data set vary from O0 to O3. AUC achieved by the proposed method in the larger data set is around 75 percent which is about 7 percent lower than with the small hand-picked sample set. Also, this seems a bit lower than most methods proposed in the literature. There can be several reasons which might cause this, but in our understanding, the reason for this is that the feature vectors do not make a clear enough distinction between semantically similar function pairs. For example, if two libraries have implemented JSON parsing functionality the function pairs are deemed to be similar even though libraries on the whole are not. Furthermore, many popular open source libraries have dependencies on other open source libraries. These dependencies are often included in the source code of the project which causes miss labelling in the data set during the feature weight optimisation task.

4.4. Matching Performance and Scalability Evaluation

The matching performance and scalability of the system are evaluated by comparing the matching times of different-sized libraries and comparing matching times when the database size is increased. Library size can be measured in several different ways, but for the performance evaluation of the proposed method library size is measured by function count in binary and executable section size. Moreover, the scalability of the system can be evaluated by comparing matching times when the database size is increased. The performance evaluation of the proposed method is conducted using a server equipped with Intel Xeon Gold 6248 and 1.5 Terabytes of RAM-memory.

Figure 16 shows the average matching times of the proposed method in the function of binary executable section size and function count. In the figure average matching times for certain code section sizes and function counts are computed. Code section sizes are normalised if a matched binary contains several ELF executable files like in the case many of statically compiled libraries. Code section size is normalised by computing a cumulative size of all executable code sections and the average size is computed based on the resulting total size. Binary matching times are compared against a static database of library hashes, thus the function count and code section size are the only attributes affecting the resulting matching times. The results show that the matching times seem to have a linear correlation with the number of functions. However, the matching times when code section size increases seem to increase linearly at first and after which they seem to level off. This would indicate that the binary code section size does not correlate directly with matching times. This can be due to several reasons, but the likely reason for this is that the complexity of CFG correlates with matching times which does not depend on code section size. Due to variations in the complexity of CFGs, matching times do not have a clear correlation with code section size after a certain point.

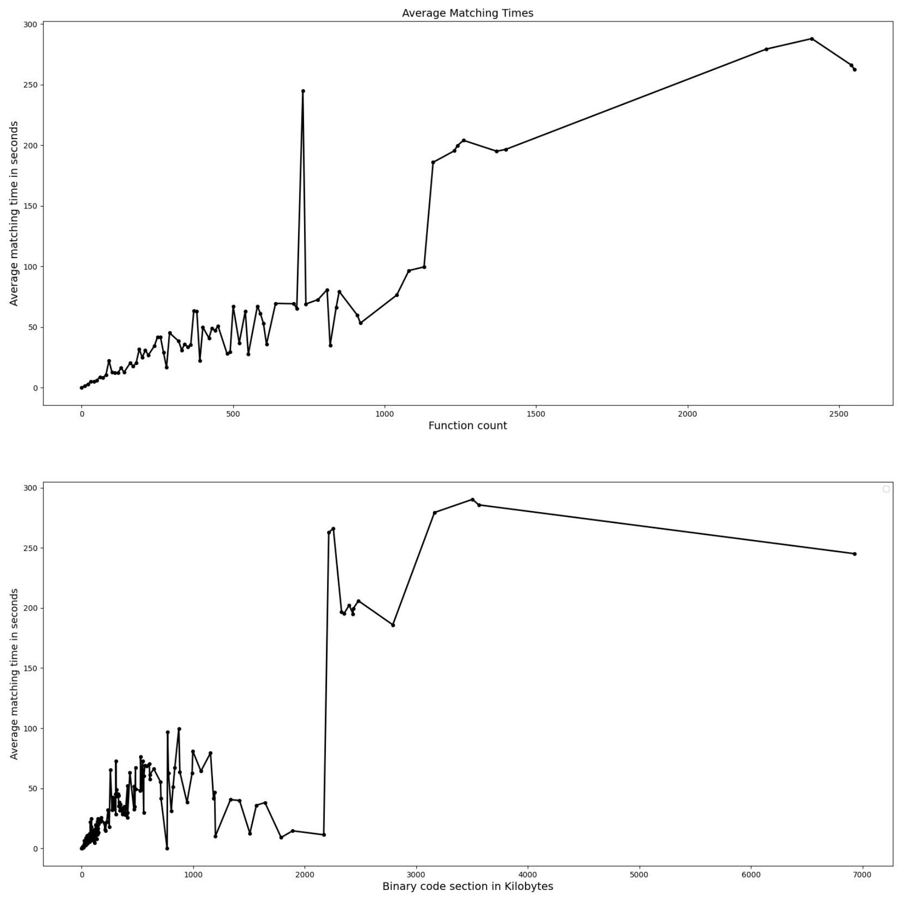


Figure 16. Matching times in function of binary function count and binary size

Figure 17 represents matching times achieved when the signature database size is changed. Database size is increased in increments of 50 hashes and average matching times for each database size are measured. Matching times are compared against a static set of hashes extracted from a binary executable file for every signature database size. Therefore, the only factor affecting achieved matching times is the size of the signature database. The matching times seem to increase quadratically when the signature database size is increased. This is caused by N amount of linear searches over hash buckets with size M during similarity score computation. Matching times with signature database containing in total about 20000 hashes is around 0.35 seconds. Matching times are relatively good in moderately sized signature databases, but it might come as a limiting factor when the signature database grows extremely large. Matching times could be improved by introducing a tree structure to the matching process where hash values are split into chunks. This would allow matching to bail out early if hashes are too different, hence improving matching times.

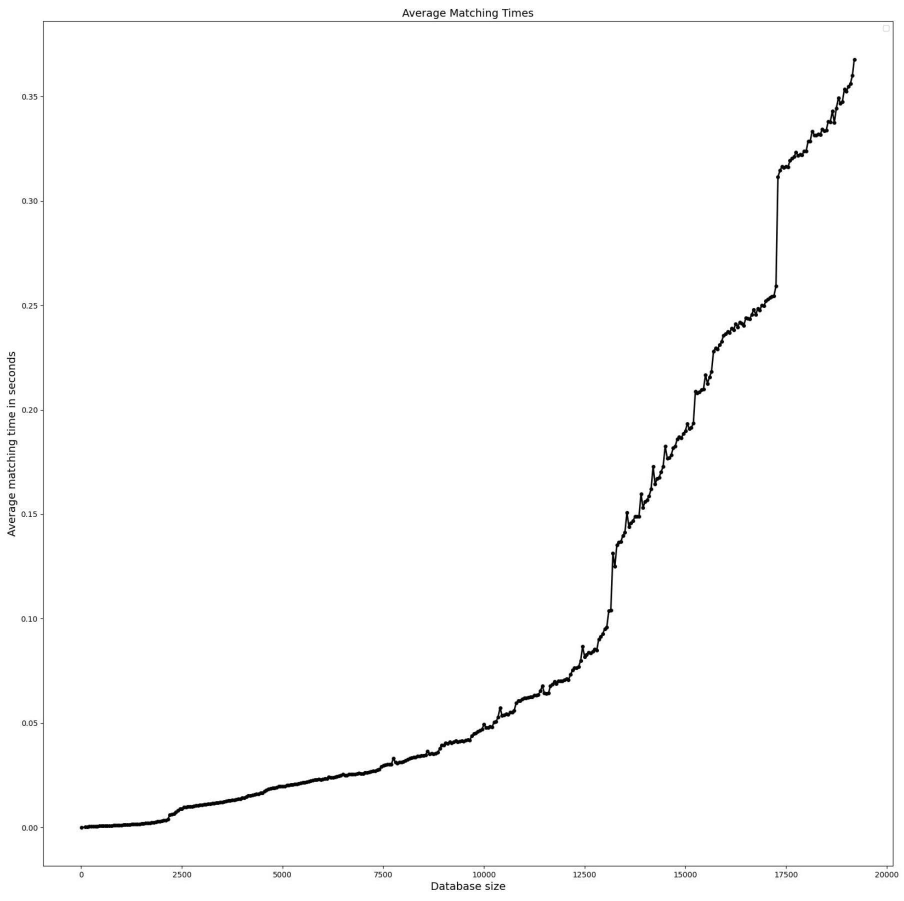


Figure 17. Matching times in function of signature database size

On average, the matching time for binary containing about 1000 functions is around 40 seconds and for binary containing about 1200 functions around 120 seconds which is relatively good. Matching times seem to increase linearly when the functions count increases, which is expected due to the linear time complexity of database signature matching. Furthermore, the proposed method achieved an average matching time of 0.35 seconds with the signature database containing about 20000 hashes in total. Matching times seem to increase linearly when the function count increases. This corresponds with the theoretical time complexity of $O(M^2)$ where M is the amount functions in the hash bucket. In future research, there are still ways to improve matching performance further as database matching seems to be the limiting factor. More selective filtering alongside function instruction count could be added. Also, tree structures could be introduced to reduce the time complexity of linear hash bucket searches. To reduce the number of required similarity computations pre-filters like basic block count comparison could be added to filter vastly different-sized functions from the comparison as they are unlikely to be similar.

5. DISCUSSION

Performance evaluation between different binary similarity matching systems is generally a complicated task even for tools aiming for similar goals [75]. Systems might target different types of executable or semantic groups of functions. Furthermore, systems might focus on certain ISAs and ignore others or systems might have widely different intended use cases like a semantic grouping of functions or malware detection. Due to these reasons, there are no generally accepted test data sets that would be widely used to test the performance of binary similarity methods. This makes the evaluation of binary similarity detection methods in a comparable way a complicated task. The method proposed in this research achieves similar performance to solutions examined in the literature. Furthermore, it meets the scalability and performance requirements set for the binary similarity-matching system. In general, machine learning aided solutions tend to have higher TP and TN rates and lower FP and FN rates compared to non-machine learning aided counterparts which may be the reason for the increased popularity of such systems in past years. Although, this comes with increased binary matching and processing times.

The system proposed in this research included several state-of-the-art methods for binary similarity analysis and accomplished relatively accurate and efficient open source component matching. The system seemed to achieve relatively good matching accuracy and performance with data sets containing multiple compilations of the same library version. However, compared to the other methods proposed in the literature proposed method achieved a bit lower accuracy and similar matching performance. Some of the possible research avenues were left out of the scope of the research. Research problems left out of the scope of this research included things like obscure and optimised executable file formats that are used especially in IoT applications. Moreover, similarity detection between major versions of open source libraries and compilers was left out of this research.

The evaluation of the proposed methods shows that the system achieves a comparable matching accuracy to other methods proposed in the literature. The proposed method achieved an AUC of around 75 percent with the larger evaluation set and keeps a relatively good TP to FP ratio in matching tasks. Hence, the method correctly detects known open-source libraries from an arbitrary binary executable in most cases. This is further demonstrated by the file-to-file similarity that simulates the typical use case of the proposed system where a compilation of the library is compared against other similar and dissimilar binary files. This visualises how similar the proposed method considers two executable files to be during the matching process. These values are relatively high, but the matching threshold can be adjusted on a library-to-library basis, thus high matching scores are not as harmful as lower ROC or AUC scores.

The system proposed achieved the goal of scalability and matching performance as the database of simhash signatures is easily expanded using pre-learned feature weights. With LSH complexity of the matching problem can be lowered. Moreover, it decreased amount of matching candidates in linear searches. Implementation of locality-sensitive hashing in binary similarity matching allows the matching to be conducted in a scalable manner. However, the matching accuracy was a bit lower than expected. This might be caused by several reasons, but the most probable reasons for the lower matching accuracy are misidentification of semantically similar functions that are structurally different.

5.1. Literature Review

In the literature many different use cases and methods for binary similarity detection have been proposed from the clone detection to malware analysis. These tasks can be accomplished using many different methods and techniques. In this section we give more comprehensive overview of the methods and systems proposed in the literature.

5.1.1. Binary Similarity Research

Wang et al. [5] demonstrated in their research that locality-sensitive hashing and especially simhash with a carefully selected feature set can be used to detect similarity between different binaries. Locality-sensitive hashes with basic block-level comparison can be used to detect similar vulnerable functions from firmware binaries. The set of vulnerable functions is matched and filtered by using a feature based on simhash and is further analysed by using symbolic execution on the basic block level on similar function pair to detect if a function extracted from an executable is patched or not. Furthermore, Dullien [24] demonstrated in his research that a binary similarity detection system based on locality-sensitive hashing can be successfully used to detect similarities between windows binaries. Locality-sensitive hashes are created from a set of features extracted from executable functions which can be compared by using hamming distance to determine how similar the function pair is. In the research are laid out the basics of how to utilise simhash effectively on a binary similarity analysis task. Also, Dullien demonstrated how to effectively create weighted simhash values based on a set of features and how they could be used in a similarity score. The system utilised CFGs and n-grams of instruction mnemonics as function features. The research showed promising results that the simhash can be used to compare and measure the similarity of binaries.

Control flow graphs (CFG) and call graphs (CG) can provide an accurate metric for the structural similarity between binaries, but the systems that mainly utilise graphs for similarity measure have to compare graphs extracted from binaries against each other. Subgraph isomorphism is known to be an NP-complete or NP-hard problem [30, 28] and is often a limiting factor for systems that utilise graphs as a similarity metric. Due to this, systems that utilise only structural and graph-based similarity have to make different optimisations like K-subgraph matching or filtering of matching candidates to further improve the performance of the system [1, 20]. Alternatively, the dimensionality of graphs comparison can be scaled down using embeddings or locality-sensitive hashing to improve matching performance. Like Nouh et al. [14] showed locality-sensitive hashing algorithms combined with function features can be used to detect similarities between function pairs. CFG of a function can be divided into partial traces of execution called tracelets which consist of two basic blocks. Typically basic blocks can be considered to be nodes in a CFG. This reduces the required space for fingerprints, but also provides information about the execution traces of a function. Hashed and normalized instruction mnemonic groups can also be effectively used to reduce the size of fingerprint sets. Similarity can be computed using extracted tracelets, tracelet features and global features of a function. Global function features and tracelet features are sets of features that can be visible in the scope of a function.

Ding et al. [33] proposed Kam1n0 to detect code clones from a set of binary files. The system is tailored for clone detection in data mining processes. Kam1n0 leverages adaptive locality-sensitive hashing (ALSH) combined with map-reduced sub-graph-based similarity searches to further enhance the accuracy of inexact ALSH searches. ALSH is constructed from a reduced k-nearest neighbour set of hashed basic block-level features converted into a prefix tree. This allows prefix trees to be dynamically indexed regardless of the used order. This also allows the creation of partial locality-sensitive hashes. Basic block-level semantic similarity can be then compared using the cosine similarity of ALSH. The results suggest that locality-sensitive hashing with map-reduced sub-graph search can yield accurate results relatively efficiently and quickly. BinSequence [34] proposed a locality-sensitive hashing method based on fuzzy matching using path and neighbourhood exploration techniques with filtering using Minhash. Functions extracted from a binary are filtered based on the number of basic blocks they contain and Minhash signatures. Functions that have similarity scores under the threshold are filtered from the pairwise path and neighbourhood analysis. Function pairs are compared against each other based on their longest common sequence (LCS) of basic blocks. Furthermore, exploration of the longest paths extracted from control flow graphs using depth-first search and neighbourhood search of basic blocks are used. Results of the research show that fuzzy matching with path exploration and neighbourhood analysis can be used to detect patched functions from a large set of binaries. Furthermore, the performance issue can be addressed by reducing the number of query candidates and fingerprint searches based on filtering based on basic block number and fingerprint similarity thresholds. Overall the path exploration techniques are relatively expensive operations even with optimisations. Therefore, they are not suitable for most systems with strict performance limitations.

Hu et al. [26] in their research address the problem of differences in control flow graphs between architectures and instruction sets by utilizing semantic similarity of binary functions. The system leverage an intermediate representation (IR) to minimize differences between instruction sets. The system proposed in their research uses several different steps to detect similarities in functions. In the first step, CFG is created. From a binary function argument information and indirect switch statements with their target addresses are extracted during the CFG traversal stage. Once the CFG traversal phase is completed semantic signatures are created utilising Minhash or longest common subsequence (LCS) of normalised semantic signature sequences collected from emulating a function execution with random input values. Semantic signatures are created by collecting information about function input and output values, comparison operands, opcodes and library calls. Semantic signatures are normalised as they might include binary-specific offsets and values. After normalisation semantic signature sequences are converted into locality-sensitive hash or LCS when the sequence length is less than a threshold. The system seemed to achieve great performance for a system utilising symbolic execution in function matching.

David et al. [28] demonstrated an approach to detect vulnerable code and executable files from firmware images. Vulnerable code blocks and functions are detected using strands created from canonical procedure fragments and by observing procedures and neighbouring procedures that are found from executables utilising back-and-forth games. Researchers used strands derived from CFG procedures at a basic block level to detect semantic similarities between binaries. In procedure strands, basic blocks extracted from a CFG are decomposed into single units of executions containing single instructions needed to compute a single output value. Procedure strands are normalised into an IR and further optimised with LLVM optimiser to keep similarities between strands across different compilations. This is achieved by implementing methods like variable name normalisation and register folding. Furthermore, the matching of similar procedure pairs extracted from executable files pairwise similarity. Pairwise matching is enhanced with Ehrenfeucht-Fraïssè games also known as back-and-forth games [76] on binary level procedures to enhance matching accuracy when using executable-centric procedure matching.

Shirani et al. [25] proposed BinShape a binary similarity detection method based on function features to create robust signatures. Function features consist of semantic features, statistical features, structural features derived from graphs like CFG or CG and instruction-level features extracted from a binary to detect standard library functions. The proposed method is targeted mainly at C/C++ standard library functions and aims to detect them from a larger set of executable binaries. The system utilises top-ranked features selected via mutual information (MI) and the best features are selected by a decision tree for signatures creation for each function. Due to a large number of function signatures in the function matching phase B++ tree structure has been implemented to ensure efficient function matching. B++ tree indexes feature vectors of a library by the best feature nodes to reduce the time complexity of the system. The results show that the optimisation levels do not significantly affect the proposed system, but compilers do affect the results. The feature selection was also utilised to reduce the size of feature sets and to exclude functions from the feature set that do not belong to that library. Results show that the best feature and top-ranked feature selection had an impact on accuracy in the evaluation set, however the improvement was not significant.

Feng et al. [23] proposed Genius that utilises both machine learning techniques and structural similarity derived from CFG analysis. Genius utilises attributed control flow graphs (ACFG) to which an unsupervised machine learning process is applied to learn categorisations from extracted ACFG. Codebooks are generated from ACFG using feature similarity using graph-based similarity and clustering using machine learning methods. Created codebooks are then encoded into a point in high dimensional vector space that can be indexed and searched. LSH can be then utilised to efficiently match encoded function vectors to a high dimensional function vector space. To improve performance bipartite graph matching was used for the ACFGs in the codebook generation with structural and statistical features that are less complex to extract. Once similarity between different has been established unsupervised machine learning can be used to cluster similar ACFGs to create codebooks in an offline process to allow real-time bug searches. Feature encoding is used to transform a function codebook into a point in a vector space to allow faster searches to match similar functions in a bug-search process. The results show that traditional graph analysis enhanced with machine learning methods can be accurately used for bug searches for IoT systems.

The performance of a bug search system can be improved with methods proposed by Pewny et al. [32] which included an adapted MinHash algorithm which allows fast comparison of normalised function semantic I/O pairs achieved via sampling. Semantic hashes can be then used to match bugs in a binary executable. The adapted MinHash algorithm constructs multiple hash values for each input value, thus creating hash buckets and storing the k smallest values that can be utilised in real-time. Once a partial match is made by using bug signatures, affected basic blocks can be further matched using the Best-Hit-Broadening algorithm (BHB) on a CFG and basic block MinHash signatures to fully match the corresponding bug signature. BHB algorithm compares CFGs found from bug signatures and matching candidates. It explores neighbourhoods of both matching candidate and bug signatures while preserving direction information. Once the optimal sub-graph neighbourhood is found the basic block with the highest similarity is returned. The results show that the proposed method achieved relatively high accuracy for near and exact signature matches. However, graph-based matching is even with optimisations quite an expensive operation and reduces the performance of the system.

To enhance the accuracy of matching in systems that do not rely on machine learning in the matching process often have to improve the accuracy of the results in some ways. This can be achieved for example using a staged matching process. Kim et al. [27] proposed a combined matching system with function names, attributed n-tuples of statically extracted function features and n-grams of function basic blocks. To minimise the effects of binary obfuscation traces stored from dynamic execution of malware functions can be utilised. Matching is done in three different stages and if unmatched functions remain matching process moves to the latter more computationally complex stage. With this, the similarity of binary executables can be computed and malware samples can be then clustered into malware families. However, this type of matching process introduces problems especially in malware analysis if obfuscation detection methods are used as they tend to modify the structure of CFGs.

Cesare et al. [31] proposed a malware variant detection system based on CFG signatures. The CFG k-subgraphs and n-grams are extracted from CFGs. Features can be extracted either from CFG n-grams or k-subgraphs and then converted into a vector representation. With the string representation of graphs well established string similarity metrics can be utilised for the similarity metric. Strings derived from the k-subgraphs are constructed from a labelled adjacency matrix. Then the n-grams are constructed via a sliding window over the control flow graph to construct a string representation of the graph with length N. In the classification stage of the proposed system, normalized compression distance (NCD) metric with edit distance is used to search similar feature vector pairs from the signature base in the filtering and classification stages. However, it was stated that the vector-based signatures had a better performance compared to the string-based approach in pairwise function feature comparison in the filtering stage. With the metric access method and tree data structures like DBM-Tree [77, 78] similarity search can be performed to match similar malware samples. The results indicate that signature-based detection methods can effectively be utilised for malware detection systems.

5.1.2. Machine Learning Aided Binary Similarity Research

Systems leveraging natural language processing to detect similarities in binary executable files face a common problem of varying instruction sets. All the major processor architectures have their own instruction sets that are optimised for that specific architecture. Many solutions have been proposed to solve this problem, like Redmond et al. [4] proposed that a continuous bag of words (CBOW) model can be used to learn instruction semantics based on the context in which they are used. This information can be then applied to semantically similar basic blocks with joint objective functions. This allows instructions to be clustered based on their semantic groups. The similarity between two instructions can be measured using set similarity methods like cosine similarity [79] to compare one or more different instruction sets. This allows grouping semantically related instructions over different instruction sets even on a basic block level. However, the abstraction of the operands loses some structural information of instructions.

Deep learning models have provided powerful tools for researchers in the field of binary similarity. Yang et al. [38] proposed a system leveraging information about executable files from an abstract syntax tree. The abstract syntax tree (AST) can be used as a semantic similarity metric combined with the Long Short-Term Memory (Tree-LSTM) data encoding structure to capture the semantic meaning of the instructions. The similarity of these encoded data structures can be computed from AST vectors produced by Siamese and Tree-LSTM networks. Similarly, TREX proposed by Pei et al. [21] demonstrated that execution semantics of a binary can be successfully used to detect vulnerable functions from a large code base. Machine learning methods and masked language modelling (LM) can be utilized to learn the semantics of instructions from the dynamic traces of instructions extracted using micro-execution. These micro-traces can be used to train the model to learn the semantics of instructions, which can be utilised to evaluate the semantic similarity of binary executable files. With masked LM machine learning models can learn instruction semantics based on context by masking out different parts of input sequences. The model is trained by executing instructions and gathering semantics from features like memory access or control transitions during the execution. A pre-trained model can provide good accuracy between different compilers, optimisation levels and even when obfuscations are applied to binary executables. Although TREX provides good matching accuracy, the scalability of the system is limited. This is caused by the mandatory model training that requires extensive preparation to gather training data set with multiple instruction sets and the gained accuracy over other methods is limited.

Liu et al. [6] demonstrated in their research that neural networks can be used to reduce the need for expert knowledge to conduct a feature selection process for similarity comparison for binary executable files. The proposed system utilises embeddings extracted from raw bytes as a semantic similarity metric enhanced with structural features like CGs and libraries exported into a function. With the convolutional neural network and Siamese Network, researchers omitted a need for expert knowledge when comparing the semantic similarity of function pairs. Moreover, natural language processing methods can be used to achieve similar goals. Instructions used to create binary executable files have similar features to spoken language. Therefore, natural language processing techniques can be successfully utilised in semantic similarity analysis and binary similarity detection [18].

Basic blocks can often be the smallest feasible building block for binary similarity systems. However, Shalev et al. [41] demonstrated that basic blocks can be divided into smaller pieces called strands. A strand derived from a basic block is a sequence of instructions that are needed to compute the value of a certain variable. This causes overlapping of instructions between strands in such a way that one instruction may belong to several different strands. Instructions inside a strand can be lifted into an intermediate presentation like VEX to normalise mnemonics and opcodes and hashed to convert the strand into a numeric value. From a set of hashed strands, vectors can be created. The similarity between vectors can be computed using a neural network consisting of two hidden layers and 2-neuron output representing the similarity between vector sets. The research showed that strands alongside neural networks can be successfully and quickly used to detect similar code blocks from open source libraries.

Code obfuscation and different compiler optimisation levels are commonly acknowledged problems for many binary similarity, clone detection and vulnerability detection systems. Traditionally text-based models like the paragraph vector distributed memory (PV-DM) model with CFGs can be used to learn the semantics of instructions. CFGs can be divided into sequences of execution traces where a list of operands and instructions are used as tokens for the PV-DM model. To enhance matching selective callee expansion for inlined function calls is implemented alongside random walks and edge coverage for binary functions. Edge coverage is done by sampling all edges from the callee expanded CFG and random walks over the assembly function to detect loop and branch conditions. This method can be successfully used to detect functions and vulnerabilities even from a large sample corpus. [19]

Machine learning classification methods can yield high accuracy and precision on a binary similarity matching task. However, these methods can have high time and space complexity which makes them unusable for real-time matching systems. Zhao et al. [40] proposed a staged hybrid system that utilises k-nearest neighbour (kNN) algorithm [80] and support vector machine (SVM) [81] to detect possibly vulnerable functions for bipartite ACFG matching. From a set of functions, dissimilar functions can be filtered based on the kNN algorithm and passed to the more accurate, but slower SVM for further filtering. Filtered vulnerable function candidates are then matched using ACFGs which contain a weighted set of basic block-level features. These ACFGs are matched based on the minimum distance threshold between basic block feature vectors. Research shows that the kNN-SVM filtering improved the performance of the matching system without losing too much accuracy on vulnerability detection tasks across different architectures in firmware functions. However, graph-based matching is a complex task which might introduce bottlenecks when the system is scaled up.

Learning the semantics of instructions can be used to filter dissimilar functions in vulnerability search. For the filtered set of similar functions more costly dynamic execution can be conducted that provides more accurate results compared to statistical and structural analysis. Dissimilar functions can be filtered by first filtering them based on a set of features and then by using a neural network to learn semantic similarity between instruction sets. This can be then used to create embedding of certain vulnerabilities for function similarity filtering. This allows dissimilar functions to be filtered, thus reducing the number of function candidates for dynamic execution. Dynamic execution allows more accurate matching based on the semantic signatures of functions that are more complex to extract without dynamic execution. This enhances the accuracy of the system in vulnerability detection task accuracy across all optimisations levels. [35] Even though, the accuracy is increased relatively significantly compared to other methods the time consumed per function is doubled. This makes the function matching for large-scale binaries unscalable compared to other methods that do not utilise dynamic execution traces.

Binary similarity detection has been studied in many instances using various approaches. This is due to the complexity and computationally expensive nature of the topic. Methods proposed in the literature often rely on features extracted from binaries like CFGs or mnemonic n-grams or machine learning algorithms that learn to match similar binaries. In many cases, methods have to balance matching accuracy and performance. Therefore, often systems include one or more performance optimisations like matching candidate pre-filtering and kNN-SVM filtering to improve performance while maintaining matching accuracy. The focus of research has been on machine learning based solutions for past years, but an increasing amount of solutions focusing on fast binary similarity matching have been proposed.

5.2. Future Work

In future research, the usability of the proposed system between different versions of compilers could be conducted as the different versions of compilers could yield structurally different binary executable files from the same source code. Furthermore, the accuracy of the proposed method could be enhanced across different library versions as they might contain varying amounts of changes between them. In the current research, these aspects were intentionally left out to reduce the scope of the research. Also, the effects of obfuscation techniques on matching speed and matching accuracy could be researched further. Patch detection was also left out of the scope of this research. The usability of simhash in patch detection could prove to be a fruitful area of research. If the granularity of the detection is changed from the function level to the basic block level minor changes between versions or patched functions compared to the upstream versions might become detectable. This would allow further analysis of patches or even license violations occurring in analysed applications or firmware. Research could also be expanded towards version detection. Most of the current implementations proposed in the literature focus on detecting libraries, clones or certain behaviour in malware samples, but the versions of used libraries are almost completely ignored. Future research could focus on the detection of differences between library versions. Open source libraries change a lot between major releases and the code base may change significantly. Further research is needed to make the detection more resilient to those changes.

In the field of IoT and embedded applications size of the executable has to be optimised due to limitations set by the used hardware. To overcome these limitations several different optimisations have been applied to the executable file formats to reduce consumed memory and executable's physical size smaller. These optimisations to the executable file formats may include methods like copying only certain parts of the original executable into the resulting binary files. Due to these optimisations, IoT executable binaries can quite often be in a non-standard ELF format, which may reduce matching precision by obfuscating the CFG structure and making disassembly slower and more complex. In future research accuracy and usefulness of simhash-based binary similarity matching could be studied on these size and memory-optimised binary executable formats.

Current implementation heavily leverages the semantic similarity of functions and binaries in similarity matching, but structural similarity like n-grams could be utilised in prepossessing stages to filter vastly different functions from similarity matching to improve matching performance and accuracy. The feature set chosen to represent a function could be also improved further by excluding overlapping features or features that are similar between semantically dissimilar function pairs.

6. SUMMARY

In this research, a fast and scalable binary similarity matching method using feature-based locality-sensitive hashing is proposed. Locality-sensitive hashing allows faster binary similarity matching compared to other available methods like graph-based similarity by decreasing the dimensionality of the similarity comparison problem. Furthermore, locality-sensitive hashing along with instruction disassembly normalisation introduces fuzziness to similarity matching which reduces the effects of different instruction set architectures and used compiler flags. This allows fast and robust binary similarity matching even if semantically similar binaries are structurally different. The system uses carefully selected features to represent a binary function. Features are chosen based on literature and inspection of similar binaries and are chosen in such a way that they are as consistent as possible across different compilations of the same source function. Machine learning is utilised to optimise feature weights for locality-sensitive hash to make sure that similar functions cluster closer together. To further reduce the effects of compilers, especially between different optimisation levels, selective function inlining is conducted. Hamming distance between simhash pairs alongside Tversky Index similarity between arbitrary binary samples and a database of known open source library hashes can be measured in a fast and scalable manner.

Our binary similarity analysis platform is created to overcome issues that most out-of-the-shelf solutions like Angr have like complex analysis tasks performed that are not mandatory for the method proposed in this research. The platform only performs tasks mandatory for the binary similarity method proposed in this research to improve performance. Moreover, our minimal IR is created for binary disassembly based on semantic groups of instructions and indented register usage documented on ISA-specific calling conventions.

The proposed method achieved a bit lower, but comparable AUC of around 75 percent. There might be several reasons that cause lower matching accuracy compared to other methods represented in the literature. These reasons can vary from overlapping features to failure to correctly classify semantically similar functions. To our understanding non-optimal function features and standard library function inlining cause the lower matching accuracy.

Matching performance achieved increased linearly when function count increases. The average matching time achieved for binary that contained 1000 functions was about 40 seconds and binary containing 1200 functions was about 120 seconds. The proposed methods achieved on average matching times of around 0.35 seconds with a signature database containing in total about 20000 hashes.

7. REFERENCES

[1] Haq I.U. & Caballero J. (2019) A survey of binary code similarity URL: `http://arxiv.org/abs/1909.11424`.

[2] Kim D., Kim E., Cha S.K., Son S. & Kim Y. (2020) Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned URL: `http://arxiv.org/abs/2011.10749`.

[3] Bourquin M., King A. & Robbins E. (2013) Binslayer: Accurate comparison of binary executables. In: Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, PPREW '13, Association for Computing Machinery, New York, NY, USA. URL: `https://doi-org.pc124152.oulu.fi:9443/10.1145/2430553.2430557`.

[4] Redmond K., Luo L. & Zeng Q. (2018) A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis URL: `http://arxiv.org/abs/1812.09652`.

[5] Wang Y., Wang R., Jing J. & Wang H. (2021) Implementing a high-efficiency similarity analysis approach for firmware code. PLoS ONE 16.

[6] Liu B., Li W., Huo W., Li F., Zou W., Zhang C. & Piao A. (2018) Adiff: Cross-version binary code similarity detection with dnn. Association for Computing Machinery, Inc, pp. 667–678.

[7] Slaney M. & Casey M. (2008), Lecture notes: Locality-sensitive hashing for finding nearest neighbors.

[8] Shirani P., Collard L., Agba B.L., Lebel B., Debbabi M., Wang L. & Hanna A., Binarm: Scalable and ecient detection of vulnerabilities in firmware images of intelligent electronic devices.

[9] Duan R., Bijlani A., Xu M., Kim T. & Lee W. (2017) Identifying open-source license violation and 1-day security risk at large scale. Association for Computing Machinery, pp. 2169–2185.

[10] Kontogiannis K., Society I.C., of Western Ontario U., on Software Engineering I.C.S.T.C., of Electrical I. & Engineers E. SANER '20 : proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution, and Reengineering : February 18-21, 2020, London, ON, Canada. 671 p.

[11] Saini V., Farmahinifarahani F., Lu Y., Baldi P. & Lopes C. (2018) Oreo: Detection of clones in the twilight zone URL: `http://arxiv.org/abs/1806.05837http://dx.doi.org/10.1145/3236024.3236026`.

[12] Bowman B. & Huang H.H., Vgraph: A robust vulnerable code clone detection system using code property triplets.

[13] Alrabaee S., Debbabi M., Shirani P., Wang L., Youssef A., Rahimian A., Nouh L., Mouheb D., Huang H. & Hanna A. (2020) Binary Analysis Overview, Springer International Publishing, Cham. pp. 7–44. URL: `https://doi.org/10.1007/978-3-030-34238-8_2`.

[14] Nouh L. (2017), Binsign: Fingerprinting binary functions to support automated analysis of code executables.

[15] Xiao Y., Cao S., Cao Z., Wang F., Lin F., Wu J. & Bi H. (2016) Matching similar functions in different versions of a malware; matching similar functions in different versions of a malware URL: `https://www.hex-rays.com/index.shtml`.

[16] Kargén U. & Shahmehri N. (2017) Towards Robust Instruction-Level Trace Alignment of Binary Code.

[17] Feng Q., Wang M., Zhang M., Zhou R., Henderson A. & Yin H. (2017) Extracting conditional formulas for cross-platform bug search. Association for Computing Machinery, Inc, pp. 346–359.

[18] Zuo F., Li X., Young P., Luo L., Zeng Q. & Zhang Z. (2019) Neural machine translation inspired binary code similarity comparison beyond function pairs. Internet Society.

[19] Ding S.H., Fung B.C. & Charland P. (2019) Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. Institute of Electrical and Electronics Engineers Inc., vol. 2019-May, pp. 472–489.

[20] Eschweiler S., Yakdan K. & Gerhards-Padilla E. (2017) discovre: Efficient cross-architecture identification of bugs in binary code. Internet Society.

[21] Pei K., Xuan Z., Yang J., Jana S. & Ray B. (2020) Trex: Learning execution semantics from micro-traces for binary similarity URL: `http://arxiv.org/abs/2012.08680`.

[22] Lakhotia A., Preda M.D. & Giacobazzi R. (2013) Fast location of similar code fragments using semantic 'juice'. In: Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, PPREW '13, Association for Computing Machinery, New York, NY, USA. URL: `https://doi-org.pc124152.oulu.fi:9443/10.1145/2430553.2430558`.

[23] Feng Q., Zhou R., Xu C., Cheng Y., Testa B. & Yin H. (2016) Scalable graph-based bug search for firmware images. Association for Computing Machinery, vol. 24-28-October-2016, pp. 480–491.

[24] Dullien T. (2018), Searching statically-linked vulnerable library functions in executable code. `https://googleprojectzero.blogspot.com/2018/12/searching-statically-linked-vulnerable.html`. Accessed: 2022-01-22.

[25] Shirani P., Wang L. & Debbabi M. (2017) Binshape: Scalable and robust binary library function identification using function shape. In: M. Polychronakis & M. Meier (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment, Springer International Publishing, Cham, pp. 301–324.

[26] Hu Y., Zhang Y., Li J. & Gu D. (2017) Binary code clone detection across architectures and compiling configurations. IEEE Computer Society, pp. 88–98.

[27] Kim T.G., Lee Y.R., Kang B.J. & Im E.G. (2019) Binary executable file similarity calculation using function matching. Journal of Supercomputing 75, pp. 607–622.

[28] David Y., Partush N. & Yahav E. (2018) Firmup: Precise static detection of common vulnerabilities in firmware. Association for Computing Machinery, vol. 53, pp. 392–404.

[29] Karande V., Caballero J., Chandra S., Khan L., Lin Z. & Hamlen K. (2018) Bcd: Decomposing binary code into components using graph-based clustering. Association for Computing Machinery, Inc, pp. 393–398.

[30] David Y., Partush N. & Yahav E. (2016) Statistical similarity of binaries. ACM SIGPLAN Notices 51, pp. 266–280.

[31] Cesare S., Xiang Y. & Zhou W. (2014) Control flow-based malware variant detection. IEEE Transactions on Dependable and Secure Computing 11, pp. 307–317.

[32] Pewny J., Garmany B., Gawlik R., Rossow C. & Holz T. (2015) Cross-architecture bug search in binary executables. Institute of Electrical and Electronics Engineers Inc., vol. 2015-July, pp. 709–724.

[33] Ding S.H., Fung B.C. & Charland P. (2016) Kam1n0: Mapreduce-based assembly clone search for reverse engineering. Association for Computing Machinery, vol. 13-17-August-2016, pp. 461–470.

[34] Huang H., Youssef A.M. & Debbabi M. (2017) Binsequence: Fast, accurate and scalable binary code reuse detection. Association for Computing Machinery, Inc, pp. 155–166.

[35] Gao J., Yang X., Fu Y., Jiang Y., Shi H. & Sun J. (2018) Vulseeker-pro: Enhanced semantic learning based binary vulnerability seeker with emulation. Association for Computing Machinery, Inc, pp. 803–808.

[36] Liu B., Li W., Huo W., Li F., Zou W., Zhang C. & Piao A. (2018) diff: Cross-version binary code similarity detection with dnn. Association for Computing Machinery, Inc, pp. 667–678.

[37] Peng D., Zheng S., Li Y., Ke G., He D. & Liu T.Y. (2021) How could neural networks understand programs? URL: `http://arxiv.org/abs/2105.04297`.

[38] Yang S., Cheng L., Zeng Y., Lang Z., Zhu H. & Shi Z. (2021) Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection URL: `http://arxiv.org/abs/2108.06082http://dx.doi.org/10.1109/DSN48987.2021.00036`.

[39] Massarelli L., Luna G.A.D., Petroni F., Querzoni L. & Baldoni R. (2018) Safe: Self-attentive function embeddings for binary similarity URL: `http://arxiv.org/abs/1811.05296`.

[40] Zhao D., Lin H., Ran L., Han M., Tian J., Lu L., Xiong S. & Xiang J. (2019) Cvsksa: cross-architecture vulnerability search in firmware based on knn-svm and attributed control flow graph. Software Quality Journal 27, pp. 1045–1068.

[41] Shalev N. & Partush N. (2018) Binary similarity detection using machine learning. Association for Computing Machinery (ACM), pp. 42–47.

[42] Ombredanne P. (2020), Free and open source software license compliance: Tools for software composition analysis.

[43] Ji Y., Cui L. & Huang H.H. (2021) Buggraph: Differentiating source-binary code similarity with graph triplet-loss network. Association for Computing Machinery, Inc, pp. 702–715.

[44] David Y., Partush N. & Yahav E. (2017) Similarity of binaries through re-optimization. ACM SIGPLAN Notices 52, pp. 79–94.

[45] Marcelli A., Graziano M., Ugarte-Pedrero X., Fratantonio Y., Mansouri M. & EURECOM D.B., How machine learning is solving the binary function similarity problem. URL: `https://github.com/Cisco-Talos/binary_function_similarity,`.

[46] Jafari O., Maurya P., Nagarkar P., Islam K.M. & Crushev C. (2021) A survey on locality sensitive hashing algorithms and their applications URL: `http://arxiv.org/abs/2102.08942`.

[47] Wang J., Shen H.T., Song J. & Ji J. (2014) Hashing for similarity search: A survey URL: `http://arxiv.org/abs/1408.2927`.

[48] Charikar M.S., Similarity estimation techniques from rounding algorithms.

[49] Henzinger M. (2006), Finding near-duplicate web pages: A large-scale evaluation of algorithms. URL: `www.cs.berkeley.edu/index.html`.

[50] Broder A.Z., On the resemblance and containment of documents.

[51] Williamson C., Zurko M.E.M.E. & Library. A.D. (2007) Proceedings of the 16th International Conference on World Wide Web 2007 : Banff, Alberta, Canada, May 08-12, 2007. ACM Press.

[52] Broder A.Z., Glassman S.C., Manasse M.S. & Zweig G. (1997), Syntactic clustering of the web. URL: `http://www.research.digital.com/SRC/`.

[53] Tversky A. (1977), Features of similarity.

[54] Jimenez S., Becerra C., Bátiz A.G.C.I.A.J.D. & esq Av Mendizábal, Softcardinality-core: Improving text overlap with distributional measures for semantic textual similarity. URL: `www.gelbukh.com`.

[55] Minhash lsh ensemble. `http://ekzhu.com/datasketch/lshensemble.html`. Accessed: 2022-06-28.

[56] M.K V. & K K. (2016) A survey on similarity measures in text mining. Machine Learning and Applications: An International Journal 3, pp. 19–28.

[57] Andriesse D., Slowinska A. & Bos H. (2017) Compiler-agnostic function detection in binaries. Institute of Electrical and Electronics Engineers Inc., pp. 177–189.

[58] Bao T., Burket J., Woo M., Turner R. & Brumley D. (2014) BYTEWEIGHT: Learning to recognize functions in binary code. In: 23rd USENIX Security Symposium (USENIX Security 14), USENIX Association, San Diego, CA, pp. 845–860. URL: `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/bao`.

[59] Committee T. (1995), Tool interface standard (tis) executable and linking format (elf) specification version 1.2.

[60] Shoshitaishvili Y., Wang R., Salls C., Stephens N., Polino M., Dutcher A., Grosen J., Feng S., Hauser C., Kruegel C. & Vigna G. (2016) SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: IEEE Symposium on Security and Privacy.

[61] Tai Z., Washizaki H., Fukazawa Y., Fujimatsu Y. & Kanai J. (2020) Binary similarity analysis for vulnerability detection. Institute of Electrical and Electronics Engineers Inc., pp. 1121–1122.

[62] Grune D., van Reeuwijk K., Bal H.E., Jacobs C.J.H. & Langendoen K. (2012) Optimization Techniques, Springer New York, New York, NY. pp. 385–459. URL: `https://doi.org/10.1007/978-1-4614-4699-6_9`.

[63] Xu Z., Chen B., Chandramohan M., Liu Y. & Song F. (2017) Spain: Security patch analysis for binaries towards understanding the pain and pills. Institute of Electrical and Electronics Engineers Inc., pp. 462–472.

[64] Eppsteint D., Subgraph isomorphism in planar graphs and related problems*.

[65] Alrabaee S., Shirani P., Wang L. & Debbabi M. (2018) Fossil: A resilient and efficient system for identifying foss functions in malware binaries. ACM Transactions on Privacy and Security 21.

[66] Owen A.B. (2006), A robust hybrid of lasso and ridge regression.

[67] Bottou L., Stochastic gradient descent tricks. URL: `http://leon.bottou.org`.

[68] Kira K. & Rendell L.A. (1992) The feature selection problem: Traditional methods and a new algorithm. .

[69] Kononenko I., Estimating attributes: Analysis and extensions of relief.

[70] Granizo-Mackenzie D. & Moore J.H. (2013) Multiple threshold spatially uniform relieff for the genetic analysis of complex human diseases. In: L. Vanneschi, W.S. Bush & M. Giacobini (eds.) Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–10.

[71] Chandramohan M., Xue Y., Xu Z., Liu Y., Cho C.Y. & Kuan T.H.B. (2016) Bingo: Cross-architecture cross-os binary search. Association for Computing Machinery, vol. 13-18-November-2016, pp. 678–689.

[72] Ming J., Pan M. & Gao D. (2012), ibinhunt: Binary hunting with inter-procedural control flow ibinhunt: Binary hunting with inter-procedural control flow citation citation ibinhunt: Binary hunting with inter-procedural control flow. URL: `https://ink.library.smu.edu.sg/sis_research`.

[73] Liu Z. (2021) Binary code similarity detection; binary code similarity detection .

[74] Flach P. (2016), Roc analysis motivation and background.

[75] Roy C.K. & Cordy J.R. (2018) Benchmarks for software clone detection: A ten-year retrospective. Institute of Electrical and Electronics Engineers Inc., vol. 2018-March, pp. 26–37.

[76] Thomas W. (1993) On the ehrenfeucht-fraïssé game in theoretical computer science .

[77] Vieira M.R., Traina C., Chino F.J.T. & Traina A.J.M., Dbm-tree: A dynamic metric access method sensitive to local density data.

[78] Souza J.A.D., Razente H.L., Camila M. & Barioni N. (2014), Optimizing metric access methods for querying and mining complex data types. URL: `http://www.journal-bcs.com/content/20/1/17`.

[79] Han J., Kamber M. & Pei J. (2012) Getting to Know Your Data. Elsevier, 39-82 p.

[80] Guo G., Wang H., Bell D., Bi Y. & Greer K. (2003) Knn model-based approach in classification. In: R. Meersman, Z. Tari & D.C. Schmidt (eds.) On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 986–996.

[81] Shmilovici A. (2005) Support Vector Machines, Springer US, Boston, MA. pp. 257–276. URL: `https://doi.org/10.1007/0-387-25465-X_12`.