TU WIEN Informatics

Function Clone Detection Evaluation

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software und Information Engineering

eingereicht von

Burkhard Otwin Hampl
Matrikelnummer 11776165

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Mitwirkung: Univ.Lektor Dipl.-Ing. Dr.techn. Georg Merzdovnik, BSc
Michael Pucher, BSc

Wien, 24. September 2021

Burkhard Otwin Hampl   Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Function Clone Detection Evaluation

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software and Information Engineering

by

Burkhard Otwin Hampl
Registration Number 11776165

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Assistance: Univ.Lektor Dipl.-Ing. Dr.techn. Georg Merzdovnik, BSc
Michael Pucher, BSc

Vienna, 24th September, 2021

Burkhard Otwin Hampl   Edgar Weippl

Erklärung zur Verfassung der Arbeit

Burkhard Otwin Hampl

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 24. September 2021

Burkhard Otwin Hampl

Danksagung

Ich möchte mich gerne bei der TU Wien und SBA Research für die Möglichkeit und Hilfe während dieser Arbeit bedanken, insbesondere bei Georg Merzdovnik und Michael Pucher. Außerdem möchte ich mich bei meiner Familie und meinen Freunden bedanken, die mich unterstützt und angetrieben hat.

Acknowledgements

I would like to thank the TU Wien and SBA Research for the opportunity and help during this thesis, especially Georg Merzdovnik and Michael Pucher. Furthermore, thanks to my family and friends who supported me and kept me going.

Kurzfassung

Beim Reverse Engineering von unbekannten ausführbaren Binärdateien stößt man oft auf bereits bekannte oder duplizierte Funktionen. Diese Funktionen der unbekannten Binärdateien sind ähnlich oder gleich zu bekannten Funktionen anderer bekannter Binärdateien. Der Vergleich und die Erkennung dieser kann natürlich nur auf der Ebene des Assembler-Codes durchgeführt werden, da wir keinen Zugriff auf den ursprünglichen Quellcode haben, aus dem die Binärdatei kompiliert wurde, was die Erkennung wiederum schwierig macht, da der Assembler-Code sich oft und stark verändern kann. Mit dem Aufkommen von Machine Learning wurden viele neue vielversprechende Ansätze zur Erkennung dieser Funktionsklone veröffentlicht, die die manuelle Arbeit bei der Binäranalyse erleichtern und beschleunigen. Für viele dieser Ansätze gibt es jedoch keine öffentlich zugänglichen Implementierungen und sie werden auch nicht von den gängigen Reverse-Engineering- und Binäranalyse-Tools genutzt.

In dieser Arbeit implementiere und vergleiche ich vier aktuelle Ansätze zur Erkennung von Funktionsklonen, die aus vielen kürzlich vorgeschlagenen Ansätze ausgewählt wurden, und implementiere den Besten in Open-Source Software-Tools, sodass er praktisch für die Reverse Engineering Arbeit eingesetzt werden kann.

Abstract

While reverse engineering unknown binaries, one often finds already known or duplicate binary functions. These functions from the unknown binaries are similar or equal to known functions of other known binaries. This comparison and detection can of course only be done on assembly code level as we do not have access to the original source code of which the binary was compiled from, which in return makes the detection difficult as the assembly code can change often and heavily. With the rise of machine learning, many new promising approaches are published that detect these function clones, which help and speed up the manual work during binary analysis. But many of these approaches do not have public available implementations and are not utilized by commonly used reverse engineering and binary analysis tools.

In this work I implement and compare four state-of-the-art function clone detection approaches, which are selected out of many recently proposed ones, and implement the best one in open-source software (OSS) tools, so it can be used practically for reverse engineering.

Contents

Kurzfassung xi

Abstract xiii

Contents xv

1 Introduction 1

2 Background 3
2.1 Control-flow graphs 4
2.2 Obfuscation 6
2.3 Optimization 7
2.4 Machine learning 7
2.5 Open-source software tools 9

3 Related Work 11
3.1 Approaches 11
3.2 Comparison 15
3.3 Tools 15

4 Implementation 19
4.1 Approaches 19
4.2 Implementation 21
4.3 Open-source software tools implementation 25

5 Evaluation 29
5.1 Methodology 29
5.2 Preparation 29
5.3 Repository 30
5.4 Results 31
5.5 Time 35
5.6 Hyperparameters 37

6 Conclusion 41

List of Figures 43

List of Tables 45

List of Listings 47

Acronyms 49

Bibliography 53

CHAPTER 1

Introduction

When developing software, one usually does not start from zero, often it involves copying and reusing code from other projects and other places like the internet.[18, 46] In the end, however, all code is executed on the same computer hardware and results in the same behavior or outcome. Because of this we can, via the low level assembly language, detect similarities and code/function clones. The three main goals of function clone detection are:

- To identify functions or function behavior of functions from a repository in unknown binaries. This is used to filter out and name known functions out of the functions from unknown binaries and to quickly see the purpose of an unknown function, which makes reverse engineering faster and easier. An additional goal would be to be able to search if a binary contains a specific function, which would make it possible to identify vulnerabilities directly from the assembly code.
- To identify same or very similar (with minimal difference) methods in the same binary. This is used to detect function clones that were generated by simply copy & paste or as an obfuscation method.
- To create "searchable" representations of functions, called embeddings, which can be stored and compared easier than the original representation. The difficulty is to create embeddings that are both representative of the original function and fast to search trough/compare.

The challenge is, not only to detect simple assembly clones that are almost one-to-one the same, but also to find the similarities over different computer architectures and obfuscation methods, that make it hard to find these clones. To do this, we not only need to match the syntax (the assembly code) itself, but also extract the semantic

meaning of functions, their dependencies and the critical patterns, that are also used by an experienced reverse engineer, who does the process manually.

There are many different approaches that try to solve this problem with varying success. The big problem is the public availability of implementations of the suggested algorithms and therefore the wide adoption.[20] While some approaches have implementations publicly available, they are not implemented in widely used (OSS) tools or there are no, that I know of, plugins for the tools that implement the suggested approaches. The result is, that they are not used in the day-to-day workflows of binary analysts. Another problem with most of the implementations, that are provided by existing research, is that they are not maintained and stopped working on current up-to-date systems.

The goal of this thesis is to find the best state-of-the-art function clone detection approach, out of a selection of recent work. This method will then be used to build plugins for interaction with OSS tools, so that they can be integrated into the binary analysis workflow. To do this, first, function clone detection approaches of the last few years are searched and an overview is obtained. After that the most promising approaches are selected and implemented. In the end they are evaluated and the best one is integrated with the OSS tools.

CHAPTER 2

Background

Computer programs are usually written in high-level programming languages that are compiled and assembled or interpreted into a low-level assembly language which the central processing unit (CPU) can understand and execute. With different architectures come different assembly languages that produce different instructions from the same original source code.[24, 70] Additionally, different compilers and compilation settings vary the outputted assembly instructions under certain circumstances quite a lot.[34] The human-readable representation of these assembly instructions are abbreviated or displayed as so-called *mnemonics*, which for instance represent the addition as `add` or the memory move instruction as `mov`. These instructions can have from zero up to three operands which indicate registers, memory locations or constants, where some architectures support more instructions than others.[24, 39] But because programs consist of more than a simple juxtaposition of instructions, CPUs and their assembly languages also support control structures like conditional and unconditional jumps/branches. With them control structures like `if` statements and loops can be represented.[70] An example

```c
int even_odd(int num) {
    int odd = num % 2;
    if(odd == 0) {
        printf("%d is even.\n", num);
    } else {
        printf("%d is odd.\n", num);
    }
    return odd;
}
```

Listing 1: C code of a simple functions that displays if a number is even or odd and returns 0 or 1 accordingly.

```
<even_odd>:
    push   %rbp
    mov    %rsp,%rbp
    sub    $0x20,%rsp
    mov    %edi,-0x14(%rbp)
    mov    -0x14(%rbp),%eax
    cltd
    shr    $0x1f,%edx
    add    %edx,%eax
    and    $0x1,%eax
    sub    %edx,%eax
    mov    %eax,-0x4(%rbp)
    cmpl   $0x0,-0x4(%rbp)
    jne    <even_odd+0x3d>
    mov    -0x14(%rbp),%eax
    mov    %eax,%esi
    lea    0xe7d(%rip),%rax   # "%d is even.\n"
    mov    %rax,%rdi
    mov    $0x0,%eax
    call   <printf@plt>
    jmp    <even_odd+0x56>
    mov    -0x14(%rbp),%eax
    mov    %eax,%esi
    lea    0xe6f(%rip),%rax   # "%d is odd.\n"
    mov    %rax,%rdi
    mov    $0x0,%eax
    call   <printf@plt>
    mov    -0x4(%rbp),%eax
    leave
    ret
```

Listing 2: The assembly code outputted from the *objdump* utility of the Listing 1 compiled as a x86–64 binary.

of such a translation can be found in Listing 1 and Listing 2, where the second is a direct result of the compilation process of the first.

2.1 Control-flow graphs

However, this brings problems, since there are now multiple paths instructions can be executed in and that can drastically change the behavior of the running program. To represent these flows a control-flow graph (CFG) is used, which is a directed graph that is constructed from an assembly function/procedure. The nodes (or vertices) of this graph are the *basic blocks*, that are the consecutive assembly instructions that are not branch, call or return instructions and the edges are these control flow transitions between basic blocks.[42, 77] One such CFG can be found in Figure 2.1, which has five basic blocks (the nodes with the round corners that are titled by their address and offset of the function

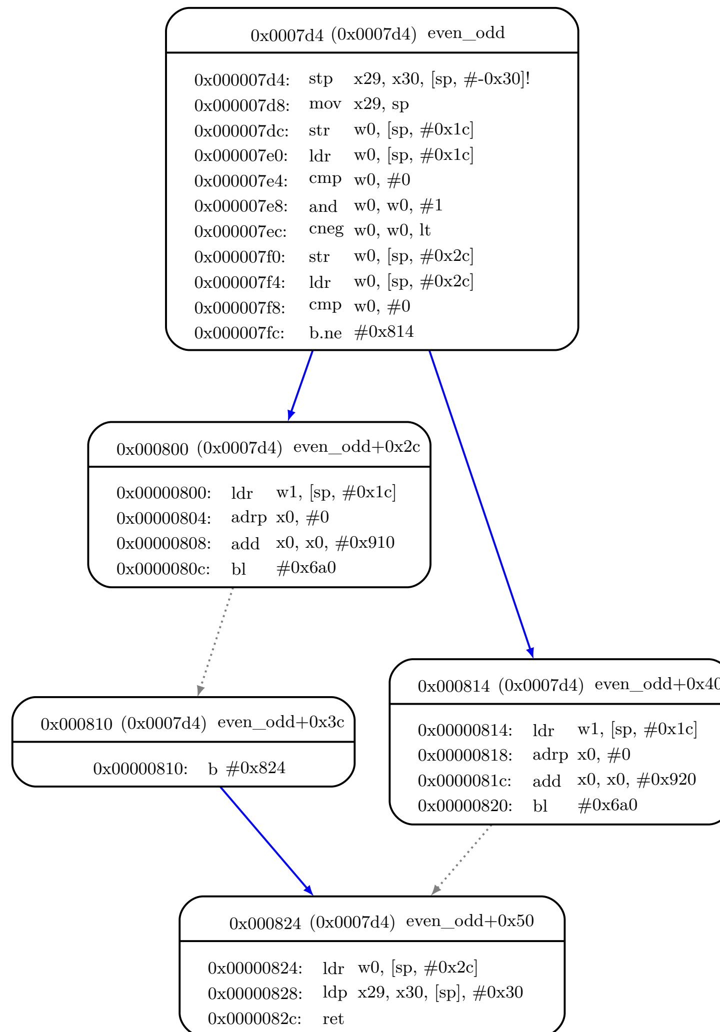


Figure 2.1: The code from Listing 1 compiled as AArch64 binary displayed as CFG. It has five basic blocks, two function calls and no loops.

start), that are split by the different branches of the `if` statement, indicated by the three solid lines, two function calls, indicated by the two dashed lines, and has no loops, as there are no directed cycles in the graph. To generate these CFGs one needs to detect the functions in binaries. This is not a simple task, as functions, which are defined at source code level, can be hard to detect in a binary.[6, 7] This is however not subject of this thesis as this would happen before a function clone analysis.

Every function can have multiple *callers* and *callees*, where the callers are all the functions that call a particular function and the callees are all the functions a certain function calls. These can also be visualised by a *call graph*, that is similar to a CFG but on function level, meaning the nodes are functions and the edges are calls. Compilers sometimes *inline*

callees, meaning they copy the code from the callee and replace the call in the caller, to improve performance of the binary, as function calls can be expensive operations. This will change the CFG and the involved basic blocks, and therefore must be taken into consideration when searching for function clones.[46]

An extension to CFGs are attributed control-flow graphs (ACFGs), where each CFG node is labeled with attributes.[76] Because CFGs are generated from the basic blocks of each function they capture only information about one function each. A graph that stores basic block information over multiple functions is called inter-procedural control-flow graph (ICFG), which is the combination of a CFG and call graph.[19] One related type of graph is the data flow graph (DFG), which instead of tracking control flow transfers on the edges, tracks the data flow. Meaning it stores and displays information on the edges of which variable/register/memory address/etc. is used by the connected vertices.[30]

Because the CPU executes a concrete path of the CFG, one can use *random walks* to serialize the CFG. Here random paths are selected, meaning starting from a basic block, that can also be randomly selected or deterministic, an edge is selected that is connected to that basic block and the connecting basic block is now our new starting basic block. This process is repeated until either there is no new basic block/edge discoverable or until a specific (path) length is reached. One can generate multiple random walks to get different execution sequences.[19, 18, 4]

2.2 Obfuscation

It is not easy to recognise known/familiar source code functions or vulnerabilities in compiled and assembled binaries even if the original source code is available, mainly because of the many variables that influence the compile process. This is why we want to find similarities between different compiled functions that originate from the same or very similar source code.[64] To make this step harder one can use obfuscation methods that make it so simple logic looks complex or that it is not simple to understand a function's purpose and how/why it is called/used. There are obfuscation techniques like control flow flattening (FLA) and bogus control flow (BCF) that can change the CFG significantly. BCF splits basic blocks apart and adds random branches and basic blocks. FLA introduces a new basic block that consists out of a `switch` statement, which aims to make a simple basic block chain more complex. The CFG and the remaining basic blocks are changed so that they now all lead to this select basic block, the resulting CFG is now flat and the basic blocks are changed to include a state variable which is used by the `switch` statement.[75, 38, 18]

An additional obfuscation method is instruction substitution (SUB), which replaces instructions with more complex logically equivalent ones and possibly adds new constants.[18, 75] Furthermore, function clones themselves can be used as an obfuscation method. This is done to disguise multiple calls to the same function as different function calls and to introduce fake dependencies.[56, 40]

2.3 Optimization

Compilers usually have different binary output depending on the compilation settings, some of these settings depend on the optimization level that is used. These compiler optimization levels determine what and how much the resulting binary is optimized. Generally a binary can be optimized for speed or size, which has different effects on what type of optimizations are applied. These optimization techniques include memory alignment, dead code elimination and loop unrolling. All of these things can, under certain circumstances, heavily change the resulting binary and therefore also things like the CFG. The particular optimization types heavily depend on the optimization level as higher levels apply more optimizations and usually include optimizations from previous levels.[12, 19, 6, 58]

A different compiler setting that affects a binary reverse engineer's efforts is binary *stripping*. Here function names, debug symbols and other source-level information is removed. This is done to decrease the file size of the resulting binaries, but also makes it harder to reverse engineer.[57] Another setting is to produce statically linked binaries, which combines the binary and all its libraries, that are normally dynamically loaded, into one huge binary, which can prevent library version mismatch. This combined with stripped binaries makes it hard to quickly identify and understand binaries, as now even common library functions, that are named and known under a non-static context, are now unnamed and unknown.[6]

It does not even have to be an obfuscation method that modifies the compiled code or the CFG. To speed up a function/program, the compiler can for example decide to inline function calls, as mentioned before. Now, to match these different CFGs, some approaches mimic the compiler's behavior and merge CFGs manually.[10] Additionally, because of different compilers and compiler settings, the same code sequences can result in almost identical assembly code that only differs in its registers or memory references. To handle these cases, one can normalize the instructions by generalizing the memory locations, registers and constants by replacing them with general constants/references that are the same for all kinds. For example, a constant is replaced by the literal `C` or a memory location by `MEM`. There are many levels on which these generalizations can happen, therefore also on registers.[4, 26, 11]

2.4 Machine learning

In the last few years artificial intelligence (AI) and machine learning (ML) gained massive traction and therefore can also be found in the field of binary function clone detection. Even though the field of AI is very broad, the subfield of ML is particularly interesting for clone detection. The different ML approaches and algorithms have a common structure in that they first extract features from the function of a target binary to be analysed and use that information to train a machine learning algorithm, so that it will "remember" the function and recall it correctly in the future. When a new function is analysed now

and fed into the trained ML model, it will return how similar they are.[77]

2.4.1 Neural networks

To make that possible one can use artificial neural networks (ANNs), often just called neural networks (NNs), that try to simulate some of the learning processes of the human brain. A typical NN consists out of nodes/units/neurons that are connected by directed edges/links. The links are weighted and the units have so called *activation functions* that act on the inputs and produce outputs. Usually, the units are arranged in layers where each unit in a layer only gets input from units from the previous layer.[66] There are typically three types of layers: input, output and hidden, where the input layer receives the input, the output layer generates the output and the hidden layer(s) can be used to store data.[44, 77] One type of NN is the recurrent neural network (RNN), where the output of a unit is fed back into its input.[66] A deep neural network (DNN) is a more complex NN that has a large number of hidden layers.[77, 44] Another special kind of NN, that was original designed for handling image classification, is a convolutional neural network (CNN), which is designed to handle data that has a known, grid-like topology.[45, 32] While a multilayer perceptron (MLP) also has multiple layers, it is different from a CNN in that it has non-linear activation functions and can therefore handle not linearly separable data.[32, 77]

2.4.2 Natural language processing

Due to the changing output of the same compiled source code, it is hard to search for function clones only on syntactical basis. This is why the focus shifted away from matching syntactical to semantic similarities. One technique, that extracts the semantic meaning from text of different languages, can be borrowed from the field of natural language processing (NLP).[64, 80, 65] While it does not seem like the field of NLP and the analysis of binary code have much in common, on a second look one can see that both want to accomplish similar goals.[80] NLP uses so-called *word embeddings*, which are (multidimensional) vectors, to encode the semantics of words.[65, 64] The usual pattern for applying this technique on binaries is to interpret an assembly instruction as a word, which is why they are also often called *instruction embeddings* (analogous to *word embeddings*), and a basic block as sentence.[34, 64] Functions can not be simply represented as sentences because they can be executed in many different ways.[65] To work around this, for example, random walks can be used, as they transform the graph to one or more sequential path. A popular choice to generate word embeddings is the *word2vec* model. To use the words around a target word as context, one can use the continuous bag-of-words (CBOW) [49] method of word2vec.[19] It has an input layer where the surrounding words are fed into a projection layer that averages all words and an output layer that predicts the target word. An approach that can also be used to obtain embeddings is Bidirectional Encoder Representations from Transformers (BERT) [16] which features model pre-training and incorporates the context of words. Pre-training

is a method of not using random initial values but instead getting the values from the training dataset directly, which improves the general performance of many models.[14]

2.5 Open-source software tools

To make the life of binary reverse engineering easier, there are tools that perform the disassembling step and basic analysis on top. While there is commercial off-the-shelf (COTS) software available that does the job, the focus of this work is on OSS tools. This is because everyone has access to them and generally the tutorial and plugin/contribution ecosystem is better.

2.5.1 Ghidra

Ghidra [53] is an open-source software reverse engineering (SRE) framework, which is developed by the National Security Agency (NSA) of the United States of America. While Ghidra provides many different tools, the main features are the included disassembler and decompiler, which try to reconstruct the C code from the assembly. The decompiler makes it very easy to pick up and accessible even for reverse engineering novices.[23]

2.5.2 Radare2

Radare2 [60] (short r2) is also an open-source reverse engineering framework. The framework consists out of many different command-line interface (CLI) utilities, where the main tool is the radare2 executable. It offers a disassembler, debugger, binary patcher, analyzer and visualizer. The biggest disadvantage and the reason that hinders wide adoption is the steep learning curve, as the CLI is not very intuitive and needs time to get used to.[47]

CHAPTER 3

Related Work

In last few years the amount of research into the field of binary code similarity and function clone detection increased greatly.[34] With the rise of AI and ML, it also became popular among binary code analysis approaches.[77] This is why I looked mostly into approaches leveraging ML.

3.1 Approaches

There are different types of approaches to tackle the function clone detection problem, some are ML based or NLP based, others are for instance pure CFG based. NLP is also a subfield of AI and makes use of *embeddings*.[34]

3.1.1 Zeek

Zeek [67] uses the self developed algorithm *proc2vec* to transform instructions to vectors which then are fed into a NN. Proc2vec splits the basic blocks into so-called *strands*, the instructions needed to calculate a particular variable, where each instruction in a basic block can be used in multiple strands, and uses a MD5 hash of the textual representation of the strands to generate an index for the output vector. The NN consists out of 4 layers, takes two vectors (the functions to compare) and outputs the similarity as probability.

According to the authors the implementation is faster than GitZ [15] and also outperforms it in terms of accuracy. Sadly it is only compared to GitZ and has no public implementation available.

3.1.2 InnerEye

InnerEye [80] is based on neural machine translation (NMT) and uses long short-term memory (LSTM) (a type of RNN) to convert basic blocks into embeddings, that are

stored in a locality-sensitive hashing (LSH) database. Additionally, a longest common subsequence (LCS) algorithm on paths of the CFG is used to find code that is contained in other code from a different architecture.[34]

The evaluation dataset has many functions/basic blocks (1.2 million) and their results show good accuracy and efficiency, compared to symbolic execution.[34] But for each assembly language, it needs its own separate model and it is only evaluated against a support vector machine (SVM) classifier that uses the same basic block attributes as Gemini [76], so not a real comparison against the full approach.[64] There is also no information given on what LSH algorithm is used.[34] They (partially) released their datasets, models and code[1].[34, 65]

3.1.3 RLZ18

In this work [65] an approach is suggested that uses a word2vec CBOW model and clusters similar instructions, even across architectures. Meaning the instruction embeddings are clustered by their semantic across the same and other architectures.[13, 34]

While they use the same dataset as InnerEye [80], they do not use the same amount of functions (only 200 thousands). Their evaluation models and code was published[2].[34]

3.1.4 Gemini

Gemini [76] uses the extracted ACFG to train a graph embedding DNN. To build the ACFG they manually select two inter-block and six block-level features/attributes that include the number of instruction and calls or different constants. After transforming the information into vectors a LSH based hash table is used for fast lookup and similarity comparison.[34, 48]

While the approach has a rather large dataset (over 420 million functions) and outperforms both Genius [28] and a bipartite graph matching (BGM) algorithm regarding accuracy and performance, the used LSH algorithm is not disclosed. The evaluation sources are publicly available[3].[34, 48]

3.1.5 DeepBinDiff

DeepBinDiff [19] begins by generating an ICFG and feature vectors, where the vectors are produced from random walks of ICFG via a word2vec CBOW model. The result of this is then used to generate graph embeddings with the text-associated DeepWalk (TADW) graph learning technique. Then it uses a k-hop greedy algorithm to compare the graphs and the basic blocks.

[1] `https://nmt4binaries.github.io/` (last accessed 30th July 2021)

[2] `https://github.com/nlp-code-analysis/cross-arch-instr-model` (last accessed 30th July 2021)

[3] `https://github.com/xiaojunxu/dnn-binary-code-similarity` (last accessed 30th July 2021)

This outperforms Asm2Vec [18] and was tested against a few binaries in many different versions and optimization levels, but not across architectures. It is susceptible to optimization/obfuscation that heavily changes the CFG and no evaluation is done with obfuscation techniques. The source code is available online[4].

3.1.6 YCT$^+$20

This approach [78] processes the CFG in two ways. First it uses BERT [16] to extract semantic information by pre-training the block and token embeddings. The result of this is then fed into a message passing neural networks (MPNN) [31], which is a graph prediction framework original developed for the chemistry field and here used to extract the structure of the CFG, where the graph embeddings are calculated. The second way to process the CFG is to learn the order of the graph nodes by passing it through a CNN. The result of both ways is then concatenated and transformed by a MLP to get the final vector that can be used for comparison.[31]

The proposed model is compared against and outperforms Gemini [76], Word2Vec [50], and BERT [16]. It should also work cross-compiler, but they do not evaluate against it, furthermore they do not measure and compare training or run time. The implementation and dataset is not publicly accessible.

3.1.7 BinGo

BinGo [10] uses the CFG to select and inline some functions, which can also be library functions, via a recursive algorithm. To reduce the search space a filtering algorithm is utilised that selects the functions that are the most similar to the target (here called *signature*) function. For the selected functions *partial traces*, code sequences that are generated out of the CFG (comparable with random walks with random start nodes), are generated and the one that are to compiler specific or “infeasible to reach” are eliminated again immediately. To now extract the semantic information of the traces, states that are true before and after the execution of these traces are defined. These are then solved by generating random input/output (I/O) samples and solving the symbolic expressions. The result of this is then used to generate a similarity score between the current and target function.

This approach outperforms other binary matching tools and was successful used for bug hunting in closed source COTS binaries, where it also was relative fast, in that it can find candidates in under 100 milliseconds for small and under 100 seconds for large (> 5000 functions) binaries. But when compared to approaches that work on assembly level it performs worse, because the random sampling can misidentify samples.[18] It also does not perform well when changing optimization or other compilation settings and the used intermediate representation (REIL [21]) is limited.[75] There is no public implementation available.

[4] `https://github.com/yueduan/DeepBinDiff` (last accessed 30th July 2021)

3.1.8 BinSequence

BinSequence [39] works by disassembling the binaries and normalizing the operands of instructions, meaning addresses and registers are replaced with constants while constants stay the same. These normalized instructions are then compared via a self-developed algorithm that returns a similarity score, which is calculated by comparing the mnemonics and operands. This is then used in combination with a LCS algorithm that returns the similarity score between two basic blocks. After that the longest path of the CFG is found and a similarity score between the target function and the current function is calculated, which is also based on the basic block one. To improve the result a neighborhood expansion algorithm is used that expands certain neighborhoods that have similar characteristics. In order to reduce the set of matching functions, the functions are filtered before they are matched with the target function. There are two metrics for that, similar number of basic blocks and similar fingerprint of the normalized function content, both defined by thresholds.

The proposed approach outperforms a few other similar tools in terms of accuracy and performance. While they also test against obfuscation they do not compare the results against other approaches. It also can not deal with FLA and some compiler optimizations without other software and can produce false positives rather easily, because thresholds have to be defined manually. Additionally, it also runs only on Microsoft Windows and the implementation is not publicly available.[34]

3.1.9 BinSign

BinSign [55] is a fingerprint based approach. To generate the fingerprint of a function the CFG is taken and partial traces, here called *tracelets*, with two basic blocks are generated. The tracelets are then used to extract block-level features like the number of constants or function calls. Additionally, a hash is calculated via the *min-hashing* [5] technique from the normalized instructions which is later used to find matching candidates via LSH and the number of basic blocks. To rank different candidates both the tracelets and global features like the number of arguments or tracelets are used.

The approach is scalable/distributed across many machines and tested against obfuscation and different compiler optimizations. But it only runs on Microsoft Windows and can not identify functions when they are obfuscated with FLA or BCF. There is no open source implementation available.[34]

3.1.10 FOSSIL

FOSSIL [4] begins by normalizing instruction, i.e. replacing memory references and constants with fixed values and removing concrete register information. Then, the opcodes are extracted and an opcode frequency distribution is calculated. Additionally, the CFG is used to generate random walk sequences. The opcodes and opcode frequencies are then put into a hidden Markov model (HMM) [72] which scores and classifies the

functions based on the opcodes of the function. To compare the CFGs of the current and target function the generated random walks are fed into a hash subgraph pairwise (HSP) [79] method. When the two previous metrics do not match a third is applied, which compares the z-score of the opcode frequency distribution. All of them are combined into a Bayesian network (BN) model, which is based on statistics and models the dependencies between the different outputs, that improves the efficiency.

It is tested against simple obfuscation techniques and with a real malware dataset. It is also evaluated against and outperforms, among others, BinClone [25] and Fast Library Identification and Recognition Technology (FLIRT) [35]. Though it does not work cross-architecture and does not detect or handle function inlining. There is no open source implementation available.[34]

3.2 Comparison

To get a better overview of the different approaches, I compiled Table 3.1, which is inspired by a survey [34]. The common metrics of the approaches from both section 3.1 and section 4.1 are included in the table. The common criteria include the evaluation methodology, the target/implementation architecture and the number of comparisons to other research. For the cross-compiler methodology only different compiler implementation are counted, i.e. only using different versions of the same compiler do not count as cross-compiler approaches.

3.3 Tools

There are many different binary analysis tools, both COTS and OSS. Widely used ones are Hex-Rays Interactive Disassembler Professional [37] (short IDA Pro or simply IDA), Ghidra from the NSA, radare2 and Binary Ninja [73] developed by Vector 35.[22, 3] They differ in their supported CPU architectures and features, where all of them have built-in disassemblers and some have decompilers. Some of these binary analysis tools also implement functionality to find and detect known or duplicate functions.

3.3.1 IDA FLIRT

Hex-Rays IDA FLIRT or F.L.I.R.T. [35] is the function recognition technology by IDA, a commercial disassembler. It is a signature-based algorithm that is used to match standard C/C++ library functions, that works by extracting the first 32 bytes of the compiled function plus the cyclic redundancy check (CRC) sum of the rest of the bytes of the function. To accommodate byte variation between different compile and linking processes FLIRT supports byte wildcards, that ignore these particular changing bytes. The process of generating the signature database limited and the databases need constant updating, to stay up-to-date with the newest versions of the binaries.[36, 22, 74, 55, 4]

Approach	Year	Architecture			Methodology				
		x86(–64)	ARM/AArch64	MIPS(32/64)	Cross-architecture	Cross-compiler	Cross-optimization	Obfuscation	No. of comparisons
BinGo [10]	2016	✓	✓	✗	✓	✓	✓	✗	4
Gemini [76]	2017	✓	✓	✓	✓	✗	✓	✗	2
BinSequence [39]	2017	✓	✗	✗	✗	✗	✗	✗	4
BinSign [55]	2017	✓	✗	✗	✗	✗	✓	✓	2
Zeek [67]	2018	✓	✓	✗	✓	✓	✓	✗	1
VulSeeker [30]	2018	✓	✓	✓	✓	✗	✓	✗	1
FOSSIL [4]	2018	✓	✗	✗	✗	✗	✗	✓	7
RLZ18 [65]	2018	✓	✓	✗	✓	✗	✓	✗	0
Asm2Vec [18]	2019	✓	✗	✗	✗	✓	✓	✓	12
InnerEye [80]	2019	✓	✓	✗	✓	✗	✓	✗	0
GeneDiff [46]	2019	✓	✓	✓	✓	✗	✓	✗	4
SAFE [48]	2019	✓	✓	✗	✓	✓	✓	✗	1
DeepBinDiff [19]	2020	✓	✗	✗	✗	✗	✓	✗	3
YCT$^+$20 [78]	2020	✓	✓	✗	✓	✗	✓	✗	7

Table 3.1: Table of comparison between different approaches. This table also includes the chosen approaches from section 4.1.[34]

3.3.2 Radare2 Zignatures

In addition to using FLIRT signature files, radare2 uses its own signature format called *zignatures* which is saved as string database (SDB) [63]. It is comparable to FLIRT as it can also store a byte pattern and bite masks. But it supports more options like graph metrics, function references, basic block hashes or variables as can be seen in the signatures generated in Listing 3.[33, 62] These two signatures are from two functions, which were compiled from the same source code, and while many signature values are the same, they do differ in their byte mask, variables, types and hashes.

3.3.3 Ghidra FunctionID

Ghidra also uses a similar function identification approach called *Function ID* (short FID) to FLIRT. It uses two 64-bit hashes, the *full* and *specific* hash, of the instructions of the function, where the *specific* hash also includes a constant operand, based on whether a heuristic classifies it as address or not. This allows accommodation of changes that are

```
sym.even_odd_x86-64:
  bytes: 554889e54883ec20897dec8b45ec99c1ea1f01d083e00129d08945fc837dfc0075
  ↪  1b8b45ec89c6488d057d0e00004889c7b800000000e8acfeffffeb198b45ec89c6488
  ↪  d056f0e00004889c7b800000000e891feffff8b45fcc9c3
  mask: ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff0
  ↪  0ffffffffffff000000000000ffffffffffffffffff00000000ff00ffffffffffff00
  ↪  0000000000ffffffffffffffffff00000000ffffffffff
  graph: cc=2 nbbs=4 edges=4 ebbs=1 bbsum=91
  addr: 0x00001159
  refs: sym.imp.printf, sym.imp.printf
  vars: b-28, b-12, r110, r119
  types: func.sym.even_odd.args=2, func.sym.even_odd.arg.0="int64_t,arg1",
  ↪  func.sym.even_odd.arg.1="int64_t,arg3"
  bbhash: 698b018c76f82adf954ce575a0ab6213a58f2bdd5737839f333a38e9e1f5325d
sym.even_odd_aarch64:
  bytes: fd7bbda9fd030091e01f00b9e01f40b91f0000710000001200a4805ae02f00b9e0
  ↪  2f40b91f000071c1000054e11f40b90000009000402491a5ffff9705000014e11f40b
  ↪  900000009000802491a0ffff97e02f40b9fd7bc3a8c0035fd6
  mask: fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  ↪  fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  ↪  ffffffffffffffffffffffffffffffffffffffffffffffff
  graph: cc=2 nbbs=4 edges=4 ebbs=1 bbsum=92
  addr: 0x000007d4
  refs: sym.imp.printf, sym.imp.printf
  vars: s28, s44, r74
  types: func.sym.even_odd.args=3, func.sym.even_odd.arg.0="int64_t,arg1",
  ↪  func.sym.even_odd.arg.1="int64_t,arg_1ch",
  ↪  func.sym.even_odd.arg.2="int64_t,arg_2ch"
  bbhash: ef744d293556eb96f3fdfb3ea7aad0fc4ed5ecf71314a0b0fa5e03c6a99489e9
```

Listing 3: Zignatures that are generated from the x86-64 and AArch64 compiled binaries from Listing 1.

introduced during the linking process, while still being able to tell different versions of a function apart. In addition to the hashes, caller and callee are considered when matching functions, however it does not support wildcards.[1, 52, 3, 23]

CHAPTER 4

Implementation

With the different approaches and research directions in mind, this chapter picks a number of promising candidates. This selection is then implemented and the most promising approach of the evaluation in chapter 5 is picked and implemented in the OSS tools.

4.1 Approaches

After reviewing all the approaches in chapter 3 and additionally comparing them in section 3.2, I picked four different approach. The main reasons for this choice were missing comparisons against each other and good performance when compared to previous approaches. Reference implementations of the approaches, when available, are adapted for the purpose of this work.

4.1.1 Asm2Vec

Asm2Vec [18] is a vector based approach, which measures the similarities of functions via the distance between their vector representation. Based on Paragraph Vector-Distributed Memory (PV-DM) [43], which is an extension to the original word2vec NLP model, a new model named Asm2Vec is proposed, that learns the semantic relationship of instructions within their surrounding context. To do that it extracts the CFG and applies selective callee expansion, to handle function inlining. The resulting graph is then used to generate random walks which are used as input for the training of the proposed model. The resulting vector is stored and used to compare to new function embeddings that are generated from the trained model via cosine similarity.

According to the authors and other research it can handle compiler optimizations and obfuscation. It also has high accuracy, does not require prior knowledge, meaning it does not use manually selected features, and is compared to many other approaches.[19, 46, 34] But it is designed only for a single architecture and can not handle dynamic jumps. An

implementation of the authors can be found online[1], there is also an unofficial Python implementation [51] available.

This approach was chosen because it is recent, looks promising and has many different evaluations. Additionally, it is often cited and compared by newer approaches, providing good baseline evaluation results.

4.1.2 GeneDiff

GeneDiff [46] is another NLP approach based on PV-DM. It consists out of three components that are called one after the other. The first one is the preprocessor stage that extracts some information/metadata from the binary and converts the binary instructions into the intermediate representation (IR), an architecture independent representation, of Valgrind [54]. It is called VEX and GeneDiff makes use of a particular Python implementation that is called PyVEX [68]. To reduce the number of low-frequency words introduced in this step, constants and strings are replaced by common tags. The second component is the *Semantic Representation Model*, that also includes the model training. In order to mitigate function inlining introduced by the compiler, GeneDiff first has to perform callee expansion to expand function calls. After that a path generation algorithm, that has the same goal as the random walks of Asm2Vec [18], is proposed that creates multiple paths out of the CFG of a function. The resulting paths are then used in the training process, which transforms them into embeddings. The last component is the *Detection Model* that is used to find similar functions from the previous learned functions via cosine distance.

This approach works cross-architecture via an IR, meaning it can also be applied to many different architectures. The authors claim the approach is both accurate and fast, and they also test it by searching real vulnerabilities in firmware images. But they do not evaluate against obfuscation and only compare to a limited number of approaches. There is no public implementation available.

GeneDiff was chosen because it uses an IR to achieve architecture independence and handles different special cases, like function inlining.

4.1.3 SAFE

SAFE [48] stands for Self-Attentive Function Embeddings and consists out of two phases. In phase one each instruction of a function is transformed into a word embedding via word2vec. To reduce the amount of different instructions, memory addresses and big constants (absolute value > 5000) are replaced by special symbols and functions are truncated to a fixed maximum length (150 instructions). In the second phase the instructions vectors of a function are combined into one by a custom NN. More precisely this Self-Attentive Network is a RNN, and the output is used to calculate the similarity of two function vectors measured by the cosine similarity.

[1] `https://github.com/McGill-DMaS/Kam1n0-Community` (last accessed 30th July 2021)

The approach is more accurate and faster (up to ten times) than what they compare it to (Gemini [76]). A large amount of instructions is needed for training the word embeddings of the first stage. Furthermore, it is only compared to the one approach and is not tested against obfuscation. The source code [17] is publicly available.

The main reason why SAFE was chosen is that it is a recent cross-architecture approach, that outperforms Gemini and its source code is publicly available.

4.1.4 VulSeeker

VulSeeker [30] is an ML based approach with its primary purpose being vulnerability search in binaries, that can also be applied for binary clone detection. It works by generating a labeled semantic flow graph (LSFG), a CFG graph where each edge is marked with either ones or zeros depending on whether the connected basic blocks accesses the same data or not, which is constructed out of the CFG and DFG, of the target function. After that eight features, the number of instructions of different groups like stack (`push`) or algorithm (`add`) instructions, are extracted from the basic blocks and captured in a numerical vector. Next, an embedding is generated from the vectors of the function via a DNN model. The resulting vector is then compared via cosine distance with the vector of functions in the database/repository.

VulSeeker supports different architectures and is evaluated in a vulnerability search context. It also outperforms Gemini [76] in terms of accuracy while maintaining similar time cost, but other approaches are not evaluated.[34] The source code [29] is publicly available.

The reason why this approach is in the selection, is because it is architecture agnostic, shows good results and has a reference implementation available. Because it is only evaluated against one approach, I wanted to see how it performs against the others.

4.2 Implementation

Because all the available implementations are written in Python, the implementations for this thesis also make use of Python 3, in order to reuse code and stay close to the original implementations. To make the training and evaluation part easier I adapted all implementations to use *angr* [69] for binary disassembling and CFG building. To get better debugging output, logging was also added and used instead of `print` and to make the implementations faster, optional concurrency was implemented that takes advantages from many CPU cores.

In order to manage the different Python dependencies of the implementations better *Pipenv* [59] is used. This makes it easy to get consistent and reliable Python package versions within the different implementations, as the default python tools do not offer a reproducible way to install dependencies for specific projects, and it offers a uniform interface which can be used to call the different implementation from external tools.

When parameter values in the implementation would differ from the one in the papers, I would replicate the one used in their respective papers when ever possible. The following sections explain the changes that were made and problems that occurred during the implementation and adaptation of the different approaches.

4.2.1 Asm2Vec

After using the unofficial Python implementation to get started, I adapted it by adding missing features, the selective callee expansion, which mirrors the parameters and the algorithm described in the paper. The Python implementation did not provide a way to read and extract executable binaries, only assembly source files. To fix that I added angr, which takes care of disassembling and additionally replaces the CFG construction. A missing feature from the Python implementation was also the possibility to extend an existing repository. The Python implementation had the learning rate logic disabled, so I re-enabled it and fixed it.

While the original research only evaluated against x86–64 binaries, I opted to also apply it to ARM/AArch64, to get a better comparison with the other evaluated approaches.[18, 34]

4.2.2 GeneDiff

Because this approach uses similar methods as Asm2Vec, the code is also based on the unofficial Python implementation of Asm2Vec and follows the general structure that is outlined in the GeneDiff paper.

The multi-path generation algorithm can run into corner cases that do not terminate. This happens when there are no source or sink nodes, which happens if there is a cycle in the CFG, or in case the source/sink nodes are not reachable from the current node in the path. Generally the algorithm does not terminate when the CFG has cycles, because it does only terminate when the current node does not have a previous node. To change that, during the node choice, it is checked if there are source/sink nodes and that they are reachable from the chosen node. Furthermore, the chosen node is directly added to the `blks` set, which stores the already selected basic blocks, from the Algorithm 2 in [46] to preferable not select it again, which breaks eventual cycles and makes the algorithm terminate. These changes to the original algorithm are also implemented in my implementation.

4.2.3 SAFE

The original source code did not fit all the needs and requirements of this thesis and needed to be adjusted. In order to have a uniform function detection and disassembly method, I added an angr implementation to the existing radare2 function analyzer. That means instead of calling the external radare2 executable and letting it generate the CFG, the SAFE implementation can use angr directly in python and extract the CFG. As additional functionality, I added the possibility to embed all functions in a binary, as it

was only possible to embed single functions by their address. Because the implementation uses deprecated dependencies, the *TensorFlow* [2] code had to be upgraded and updated to make it work in a current up-to-date environment.

To make it easier to train and build a repository of functions/binaries in one go, I added the option to train the word2vec model. The original implementation only referred to the pretrained word2vec model, the word2vec trainer was implemented in order to have the whole SAFE model be trained and evaluated. The adapted implementation needs multiple versions of functions with the same names. This allows the model to specifically learn the differences between compilers and different compiler optimizations. However, without multiple versions of the same binary, SAFE cannot be trained.

4.2.4 VulSeeker

The original source code had many Chinese comments that had to be translated first, in order to make the code more readable. Additionally, it was written in Python 2, so it first had to be ported it to Python 3, which was also used to refactor the code base. While adapting the code for use within this thesis, several performance improvements were made: Recursive function calls have been removed, directories have been used for indexing intermediate data and values, for example addresses of string references, are pre-calculated. To make the source code repository smaller I also removed all the unnecessary files like binaries or CFG files. Some parts had to be rewritten because it used the IDA Python application programming interface (API). The functionality of IDA, in this case the CFG extraction and disassembly, has been replaced by angr. As with SAFE, the implementation relied on deprecated dependencies, therefore the TensorFlow code also had to be updated/upgraded.

Originally it used a custom version of miasm2, which still relied on Python2 and was not up-to-date with mainline miasm. This custom version of miasm has been replaced for the implementation in this thesis and is using mainline miasm [9] now. VulSeeker uses the intermediate representation control-flow graph (IRCFG) of miasm, which is a CFG for IR, that can not be replaced by angr because it uses a different IR and has no IRCFG. VulSeeker quickly runs into performance issues, as it would take a small sized repository, with 1000 functions, about 35 days to run.

4.2.5 Common interface

For training and evaluation I designed a unified interface of scripts that is used by all implementations. This interface is called by one script that can train and evaluate all implementations at once. The arguments passed to this script are passed on to the interface and used there.

The common interface is called by a Bash script (`evaluate.sh`) that trains and extracts the results of all implementations. To make them work one needs to prepare the binaries first by collecting them in a directory, this is also done by Bash script (`prepare.sh`), which copies the different compiled binaries from the corresponding build directory into

a common binary directory. To speed up analysis the reading of the binary and the extraction of the CFG by angr can be pre-calculated and stored in binary format, so the implementations can read them and use the finished CFG. The first script of the common interface is the training script (`train.py`) which is individual to each implementation, but follows the same steps: First the directory that was given is searched for all the binaries that should be trained. Afterwards the individual implementations are called and the models/repositories are trained from the functions of the binaries. At the end the models or function repositories are saved to disk, so they can be used again later. The second script (`run.py`) is for evaluating the approaches, this is done by querying and comparing functions from binaries with the saved ones in the respective saved function repositories. It works similar to the first script, as it also first collects all binaries that should be evaluated, but then executes the non-training/query/search/estimation methods. The result of these comparisons is saved in a comma-separated values (CSV) file (`result.csv`) for further evaluation and result extraction. An illustration of this process can be seen in Figure 4.1, where interactions between the different components are displayed by arrows and different implementations are indicated with dashed line borders.

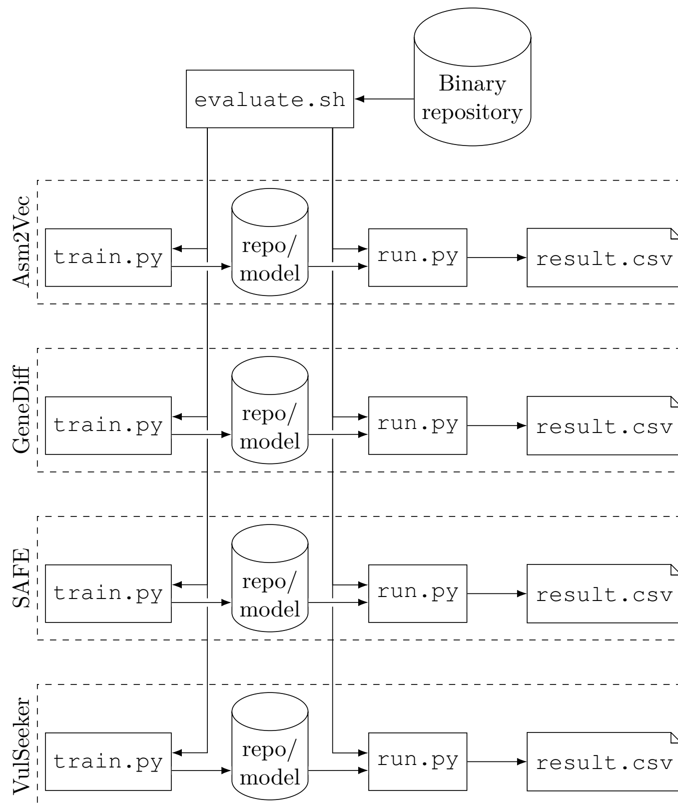


Figure 4.1: Graphical summery of the common interface and script workflow.

Both scripts also support various options like enabling or disabling debug outputs or

parallel processing for faster run times. The training script generally also skips already processed steps when the relevant files and directories are already exist. Similar to that when the second script extracts the results, it can resume when it was interrupted, by skipping the already processed paths of the binaries.

4.3 Open-source software tools implementation

Out of the selected approaches GeneDiff was integrated with the OSS tools. While there are many OSS tools out there the decision was made that the implementation is done with radare2 and Ghidra. The reason for this is that they are widely used and already have many plugins, which makes adopting and implementing easier.

The integrations are structured in a similar manner to the evaluation scripts in subsection 4.2.5. They call the same functions that are used for training and similarity checks. However, instead of loading binary files from a directory, the binary information is passed from the respective tool. If only the selected function should be trained or checked, the plugins ask the tool for the current function and pass that information to the implementation.

Because they both rely on the same implementation a trained repository can be used with both. Additionally, they support configuration files, meaning settings and changes can be saved. Changes can be made to log levels, i.e. to enable debugging, parallelism can be turned on and off, and others. In general, every parameter passable as command line flags in subsection 4.2.5 can be set via configuration files. To make it possible to save multiple binaries that contain the same functions or function addresses in one repository, the repository stores each binary separately and each binary contains its functions.

4.3.1 Radare2

To interact with radare2 a script was developed that extends the existing radare2 CLI to train and query repositories. It uses *rlang-python* [61] to integrate the script as a *core plugin* into the radare2 CLI, which makes it seamless for the user to use. To make this work the `genediff.py` script location is passed to the radare2 executable via the command line argument `-i`, which tells radare2 to run the script after the binary file was loaded. Radare2 then calls the script every time a command is sent to radare2, e.g. via the CLI, and the script checks if it is a relevant command for it, and performs the related action. While radare2 could als be used for generating the CFG, in the current implementation angr is still used for this because it makes the saved repository interoperable between tools. An example workflow of the working plugin can be seen in Listing 4, where a function of a binary is trained and evaluated against a function from a different binary. To train the function the binary has to be analysed (with the `aaa` command), else radare2 can not locate the functions (`s sym.even_odd`). With the next command (`Aa.`) the current selected function is trained, in this case the "even_odd" function. Because we want to reuse the repository, we need to save it to a file with the `As`

```
$ r2 -i ./genediff.py ./even_odd_amd64
[0x00001060]> A?
Usage: A[calsrwe] [file]
| A    show this help
| Ac    check function(s) against repository
| Aa    add function(s) to repository
| Al    load repository
| As    save repository
| Ad    reset current repository
| Ar    read configuration
| Aw    write configuration
| Ae    get/set configuration options
| A?    show this help
[0x00001060]> aaa
[...]
[0x00001060]> s sym.even_odd
[0x00001159]> Aa.
100%|████████████████████████████████████████| 1/1 [00:00<00:00, 81.85it/s]
[0x00001159]> As ./bin.repo
[0x00001159]> q
$ r2 -i ./genediff.py ./even_odd_aarch64
[0x000006c0]> aaa
[...]
[0x000006c0]> s sym.even_odd
[0x000007d4]> Al ./bin.repo
[0x000007d4]> Ac.
100%|████████████████████████████████████████| 1/1 [00:00<00:00, 267.90it/s]
even_odd_amd64:
    sym.even_odd (0x7d4):
        even_odd (0x1159): 0.9574687841851504
[0x000007d4]> q
```

Listing 4: Radare2 plugin in action on x86-64 and AArch64 compiled binaries from Listing 1.

command. The same steps are followed by the second radare2 invocation, but this time the stored binary repositories is loaded (`Al`) and the current function in check against it (`Ac.`), the resulting similarity score can be seen in the second to last line.

4.3.2 Ghidra

Ghidra plugins are using a Java API and are typically written in Java, but because all implemented approaches are written in Python, it would introduce additional maintenance overhead if the plugin was written in Java. Because of that, the *Ghidra Bridge* [41] project was used to integrate the Python implementation. The interaction with Ghidra works via the CLI similar to the radare2 script from above. For this to work the Ghidra Bridge needs to be started first as the script communicates with it via network sockets and in order to retrieve the necessary information from the running Ghidra instance. The

script itself is therefore independent from Ghidra and can be executed everywhere via the Python interpreter. To maintain compatibility between tools regarding the saved function repositories, angr is still used for generating and saving them. The script output of an example can be seen in Listing 5, that shows the training and evaluation of two functions out of Ghidra. To make this work one first needs to start a code browser instance in

```
$ python ./genediff.py
Welcome to the genediff shell.   Type help or ? to list commands.

> ?

Documented commands (type help <topic>):
========================================
add  check  delete  env  exit  help  load  quit  read  save  write

> add current
100%|████████████████████████████████████████| 1/1 [00:00<00:00, 81.01it/s]
> save ./bin.repo
> quit
$ python ./genediff.py
Welcome to the genediff shell.   Type help or ? to list commands.

> load ./bin.repo
> check current
100%|████████████████████████████████████████| 1/1 [00:00<00:00, 145.62it/s]
even_odd_amd64:
    even_odd (0x7d4):
        even_odd (0x1159): 0.9576054810967567
> quit
```

Listing 5: Ghidra plugin in action on x86-64 and AArch64 compiled binaries from Listing 1.

Ghidra and start the Ghidra Bridge, so the script can connect to the socket. After the script has connected to the running instance, the current in Ghidra selected function can be trained (`add current`) and the repository saved (`save`). When now a new binary is loaded into Ghidra and the Ghidra Bridge is restarted, the stored repository can be loaded (`load`) and the current selected function can be checked against it (`check`), which returns the similarity between the two functions.

CHAPTER 5

Evaluation

In order to get an overview of the four different implemented approaches, this chapter performs an evaluation of these approaches, based on metrics commonly used in the accompanying literature. Besides accuracy in function clone detection, training and execution speed are also metrics of interest.

5.1 Methodology

A widely used metric for comparing different approaches is the area under the curve (AUC) of the receiver operating characteristic (ROC). The ROC curve plots the true positive vs. the false positive rate regarding a certain threshold and the AUC is the area under the plotted line.[27, 8] It is used to quantify the quality of embeddings, by using the threshold to decide when two functions are similar regarding their cosine distance. Many of the evaluated approaches [46, 48] use and compare with this metric. But this is not the only metric that is used to evaluate accuracy, precision (the true positives divided by all positives) and recall (the true positives divided by false negatives plus the true positives) are also necessary, because we need it to measure the performance in a real repository context, where we want the implementations to return the correct function first.[27, 8] Unfortunately precision and recall can not be used directly in the evaluation, thus the *Precision at Position 1* (*Precision@1*) is used, where the precision value is calculated for function matches at the first/top position.[18] A different metric that is also important for practical approaches is the train and run time.

5.2 Preparation

The actual evaluation built upon the common interface that is described in subsection 4.2.5. To perform the evaluation, first the code sources, that are compiled and used for building the binary repository, must be selected, downloaded and put into the repository directory.

In the next step the provided sources are compiled via a combination of a Docker container and a shell script into Executable and Linking Format (ELF) files/binaries. This standardises and streamlines the compilation process so it can be replicated more easily. After the sources are compiled into the different ELF binaries they are collected into one common directory structure with a Bash script (`prepare.sh`). The directories are organized by compiled architecture, compiler version and compiler flags, which are typically optimization levels. As mentioned before to improve performance and reduce the run time of the implementations, the CFG of the compiled binaries is calculated beforehand and saved as cache.

Afterwards the evaluation script of subsection 4.2.5, with the build binary repository and cache passed on, is executed. The results of this stage are the CSV files that are going to be processed after processing finishes. Each line in the result files consist out of the path of the binary, the compared functions and the similarity value, which is a distance between the vectors of the functions. These files are now processed by the `extract_results.py` Python script and transformed into new CSV files (`out.csv`). The script reduces the CSV files to the lines where the similarity value is the highest for a particular path and function pair, which is later used by the Precision@1 script.

To plot and calculate the ROC curves and the area under them, the `auc.py` script is used. It reads the result CSV files (`result.csv`) and makes the calculations, by extracting the similarity scores and building a target score by comparing the function names. The last Python script (`precision.py`) is used to calculate the Precision@1 values. It does this by reading the previously created CSV files (`out.csv`) and extracting if, for a given function pair, it is a true positive (the functions are similar) or a false positive.

5.3 Repository

The large repository was filled with the source codes of OpenSSL 1.1.1i, curl 7.74.0, PostgreSQL 13.1 and glibc 2.32. During the development and evaluation of some implementations there were problems with the PostgreSQL binary, as analysis within the implementations ran into memory constraints. Therefore, PostgreSQL binaries are not included in the evaluation.

For the compilation process different compiler versions and architectures were used: different versions and architectures of the GNU Compiler Collection (GCC), the GCC version vom 7 to 10 and the CPU architectures of AArch64 (ARM64), ARM (32) armel & armhf, x86_64 and x86 (i686).

The small repository consists out the curl binaries from the large repository and that used the output of GCC version 10 for training and the output of GCC version 8 evaluation. The exception to this is SAFE as it needs the same binaries twice for training so it was trained with version 9 and 10 and evaluated with version 8. The same was done to VulSeeker but here it was trained with 9 and 10 and evaluated with 8 and 9.

The repository of the optimization comparisons uses the binaries of the compiled curl 7.75.0 source code and compiles them with the GCC version 10 and the architectures mentioned before. Instead of compiling with different GCC version the sources are compiled with different optimization levels, in this case from no optimization (`O0`) to full optimization (`O3`) with all steps in between (`O1` and `O2`).

5.4 Results

The first criteria that is looked into are the ROC curves with their respective AUCs. The results for the small repository can be seen in Figure 5.1, where the ROC curves are plotted for all four tested approaches and the AUC is given for each. This shows us

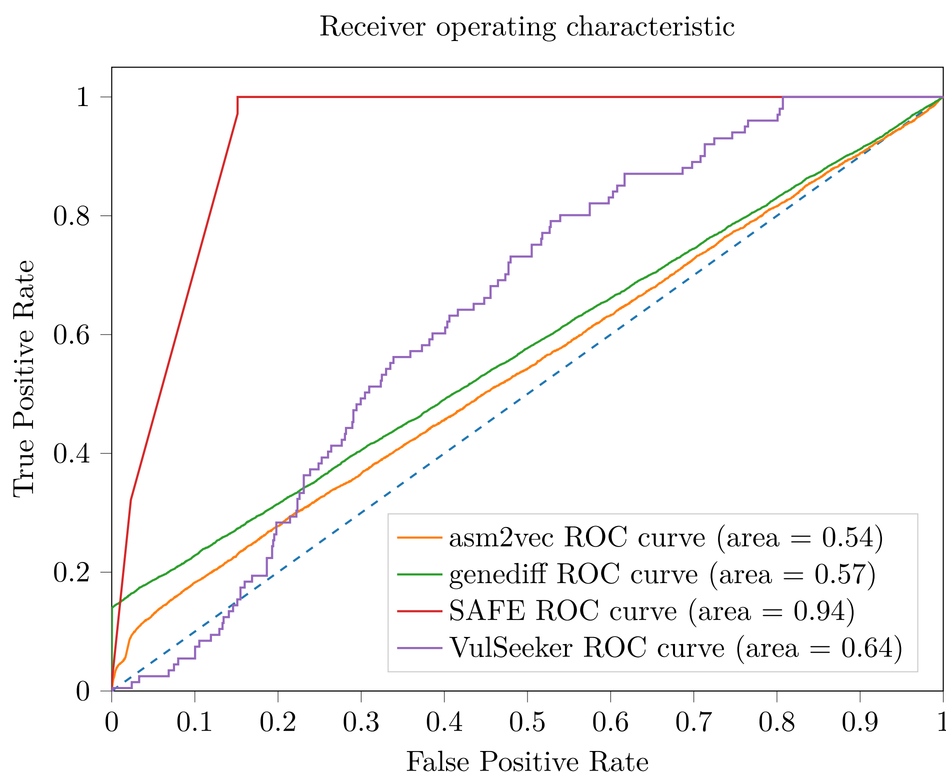


Figure 5.1: The ROC curves of all approaches with the small repository.

that the different approaches handle the given binaries differently and result in different outputs. The AUC value, that is stated in the graph as “area” in the legend, tells us the area under the respective curves, which can also be seen in the plot. When comparing the different approaches between each other we can see that SAFE outperforms VulSeeker which in return outperforms both GeneDiff and Asm2Vec. This tells us that, for all functions and binaries in the given repository, the similarity between the learned and the evaluated function, is the best in SAFE and the worst in Asm2Vec. The Table 5.1

however tells us that the Precision@1 value of VulSeeker is the best and of Asm2Vec the worst. What also stands out, is that they all do not perform as well here as they showed in their respective papers, but SAFE comes the closest with the AUC value that is 0.05 less than the original. A possible reason for this could be the chosen repository/binaries or the evaluation methodology, that could be implemented wrong.

With the ROC curve of Figure 5.2 the behavior of the different implementations on a large repository can be seen. This scenario mirrors the results of the small repository in

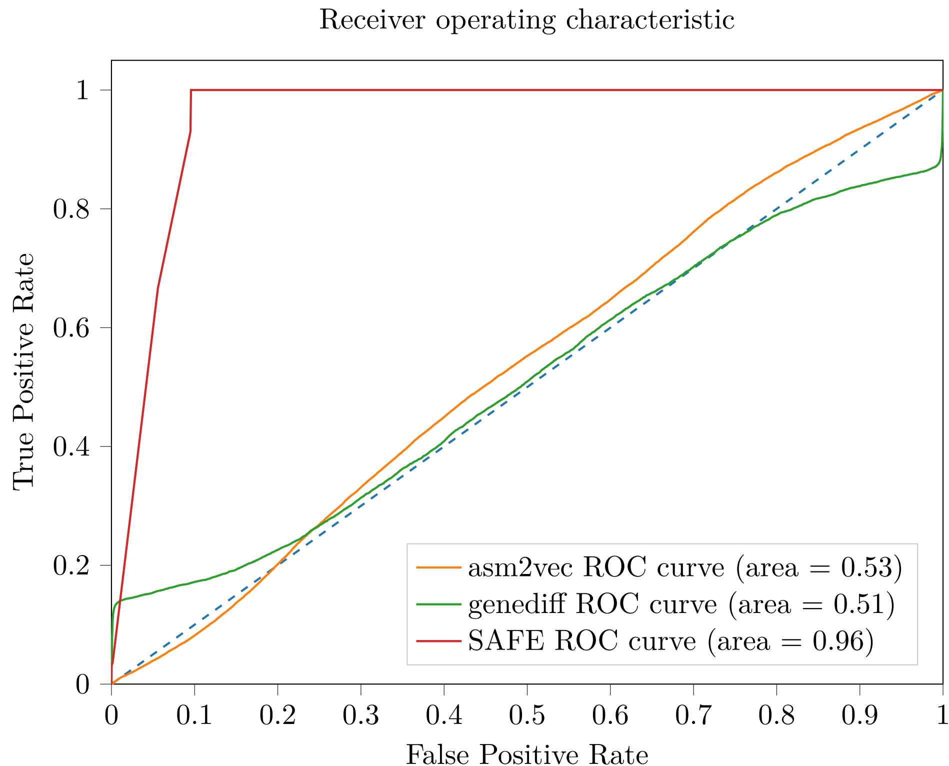


Figure 5.2: The ROC curves of the large repository.

terms of the AUC of SAFE and the other two implementations perform barely above a random classifier, that is displayed by the dashed diagonal line. Asm2Vec performs better than GeneDiff, which is the opposite of the small repository, but both do cross the random classifier line, which indicates that there is something wrong with either how they both handle the dataset/evaluation or the implementations are buggy/faulty. When looking at the Precision@1 values in Table 5.1, we can see that Asm2Vec performs well below the other two approaches and SAFE also outperforms GeneDiff.

SAFE in general seems to behave strangely as it only has very few points on the curve and therefore behaves very “angular”, except for the second optimization evaluation. This does not match with the original research [48] and can be an indication that the implementation or evaluation in this thesis is not correct. This is also indicated by

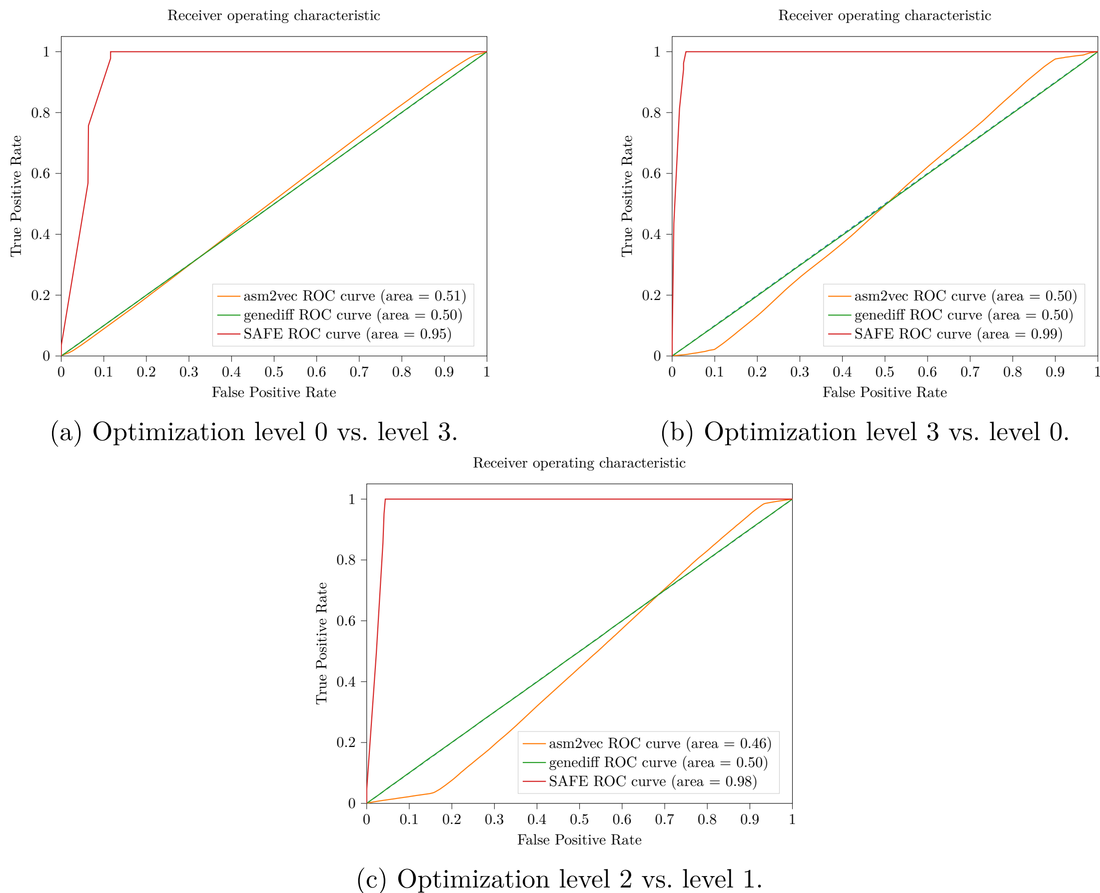
running the evaluation on the OpenSSL 1.1.1i binaries of the large repository with the original trained model, as they result in similar angular curves and results (AUC of 0.93 and Precision@1 of 0.195100) as the one in the small repository.

Repository	**Approach**	**Precision@1**
All approaches with the small repository	asm2vec	0.000703
	genediff	0.149039
	SAFE	0.181520
	VulSeeker	0.203235
The large repository	asm2vec	0.000067
	genediff	0.003827
	SAFE	0.015316
Optimization level 0 vs. level 3 from the first optimization evaluation	asm2vec	0.008557
	genediff	0.001716
	SAFE	0.339909
Optimization level 3 vs. level 0 from the first optimization evaluation	asm2vec	0.012167
	genediff	0.002638
	SAFE	0.616920
Optimization level 2 vs. level 1 from the first optimization evaluation	asm2vec	0.008788
	genediff	0.006861
	SAFE	0.581922
Optimization level 0 vs. level 3 from the second optimization evaluation	asm2vec	0.003060
	genediff	0.002547
	SAFE	0.045882
Optimization level 3 vs. level 0 from the second optimization evaluation	asm2vec	0.004557
	genediff	0.000851
	SAFE	0.052171
Optimization level 2 vs. level 1 from the second optimization evaluation	asm2vec	0.008100
	genediff	0.002098
	SAFE	0.107313

Table 5.1: Precision at Position 1 of the evaluations inspired by Table 3 in [48].[18]

5.4.1 Optimization

The ROC graphs of the first optimization evaluation can be seen in Figure 5.3, where the first optimization level was trained and the second one was evaluated. First of all SAFE outperforms the other two (VulSeeker was not tested as it was too slow to get results and therefore not feasible, as mentioned before) tested approaches clearly again, which do perform about the same as a random classifier (AUC of 0.5). Notable is that GeneDiff performs exactly like a random classifier and Asm2Vec a bit better (Figure 5.3a) and worse (Figure 5.3c), as it also dips under the "random" guess diagonal dashed line. When comparing the different plots between each other it seems that they all perform and look similar, especially the last two. It is also notable that when comparing Figure 5.3a



(a) Optimization level 0 vs. level 3.

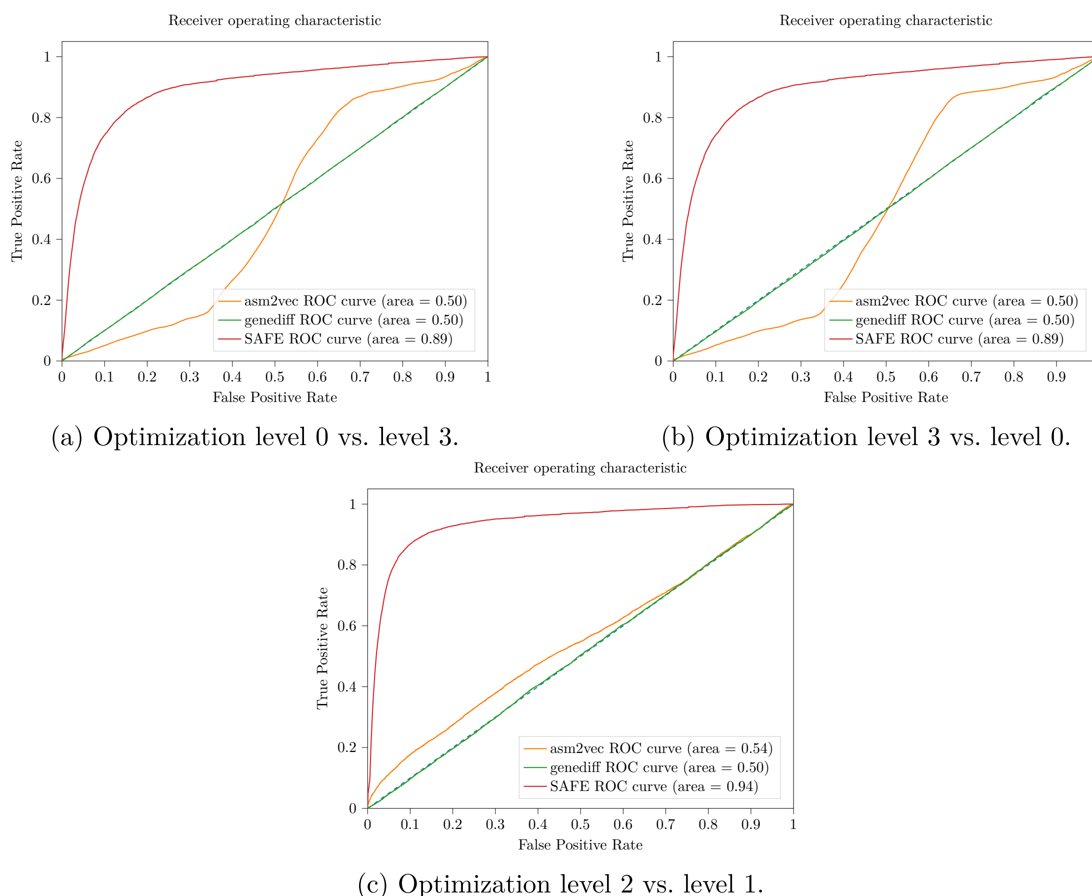
(b) Optimization level 3 vs. level 0.

(c) Optimization level 2 vs. level 1.

Figure 5.3: The ROC curves of different optimization levels from the first optimization evaluation.

and Figure 5.3b, which have the same binaries but are trained and evaluated with the respective other, in the second curves SAFE performs better.

The second optimization evaluation, which can be seen in Figure 5.4, is trained with a subset (here 70%) of all optimized binaries/functions, hence all optimization levels, and evaluated with the remaining subset of the mentioned optimization levels. When looking into the result of this we can see that they do look better, than the first evaluation, especially SAFE, but are still not what the respective papers showed. Here does SAFE still perform the best and looks way better, but does not perform as well when looking at the AUC of the ROC curve. Asm2Vec does perform a bit better, but does also dip below the random classifier more than in the first evaluation and GeneDiff is still the same as a random classifier. Comparing the first two graphs (Figure 5.4a and Figure 5.4b), which are the same optimization levels but reveres, one can see that they look the same and result in the same values. The last graph Figure 5.4c does show the best results out of the three.



(a) Optimization level 0 vs. level 3.

(b) Optimization level 3 vs. level 0.

(c) Optimization level 2 vs. level 1.

Figure 5.4: The ROC curves of different optimization levels from the second optimization evaluation.

That means when comparing binaries with different optimization levels, the more the levels are apart the less the approaches recognize the similarities, which makes sense as higher optimization levels usually add more optimization methods and so gradually change the resulting binaries.

Additionally, Table 5.1 shows that Precision@1 values of SAFE are better than those of Asm2Vec and GeneDiff in both evaluations. What is striking, however, is that Precision@1 values of SAFE are worse in the second evaluation, even though the AUC values are better.

5.5 Time

The (run-)time was tracked during the creation and training/evaluation of the optimization repository. The prepare-script, that collects the binaries, took 70 and the cache script, that extracts the CFG from the binaries to speed up the implementation, took

1163 seconds to complete. Both scripts run on the whole repository, i.e. do not distinguish between the different optimization levels.

5.5.1 First optimization evaluation

The scripts/implementations of the first optimization evaluation are performed on one optimization level each, where the results are the average of the three optimization runs.

Building of the Asm2Vec repository, that is used for training, takes about two minutes, which is accelerated by the cache that was built before, and the training itself 35 minutes. On the evaluation side of Asm2Vec, it takes about one hour and 40 minutes to generate the finished CSV, which includes the loading of the binary cache, but varies considerably, depending on the repository and the binaries in it.

In GeneDiff the function extraction, out of the cache, takes three minutes and the model training on average about seven and a half hours. That is the same time on average that it takes to generate the evaluation. Those two times fluctuate a lot.

With SAFE, the i2v implementation takes around three minute and the data extraction/dataset creation takes 24 minutes on average. The training process itself takes 14 minutes with the hyperparameters shown in Table 5.4. For the evaluation the implementation takes over two hours on average, which varies a lot.

5.5.2 Second optimization evaluation

For the second optimization evaluation, the first step was to train the three implementations with the subset of all optimization level binaries/functions. To do this all binaries were read and the functions split into the training and evaluation set, which took about two minutes. After that the training was done once for the implementations and the evaluation was done with the different levels.

The data extraction for the training process of Asm2Vec took five and a half minutes and the training itself took one hour and 22 minutes. The evaluation process took without optimizing it about ten hours and with the optimized evaluation the time was reduced to about one hour.

For GeneDiff the data extraction, including callee expansion and multi path generation, took 16 minutes and the training process 18 hours. Without the optimized evaluation implemented it took GeneDiff almost 35 hours to complete one evaluation, with the optimized implantation it was cut down to three and a half hour.

SAFE took 51 minutes for the data extraction and dataset creation, for the training it took 37 minutes. The evaluation took about 50 minutes.

VulSeeker was not evaluated with any optimization, because it already took over 6 hours for the small subset of binaries, even with lowered/small parameters, to complete during the previous evaluations. The only evaluation was done on the small repository (see Figure 5.1) and there not on the full compiler version as that one would have taken over

a month to complete. A smaller subset was evaluated, of about one and a half percent, and even that took over six hours to complete.

5.6 Hyperparameters

In this section the used hyperparameters for the training and evaluation process are stated.

5.6.1 Asm2Vec

The parameters for Asm2Vec can be seen in Table 5.2

Parameter Name	Meaning	Value
`d`	The dimension of the vectors for tokens.	`100`
`initial_alpha`	The initial learning rate.	`0.0025`
`alpha_update_interval`	How many tokens can be processed before changing the learning rate?	`10000`
`rnd_walks`	How many random walks to perform to sequence a function?	`10`
`neg_samples`	How many samples to take during negative sampling?	`25`
`iteration`	How many iterations to perform?	`1`

Table 5.2: Asm2Vec hyperparameters adapted from the Asm2Vec repository.[51, 18]

5.6.2 GeneDiff

The parameters for GeneDiff can be seen in Table 5.3

Parameter Name	Meaning	Value
`num_ins`	The number of instructions in the callee function to be considered for not expanding.	`10`
`threshold`	The threshold ratio between the callee and caller to be considered for not expanding.	`0.5`
`d`	the dimension of the instruction vector.	`100`
`starting_alpha`	The starting alpha for the learning rate.	`0.025`
`ALPHA_UPDATE_RATE`	The alpha update interval.	`10000`
`negative_samples`	Number of negative samples.	`5`
`iteration`	The number of iterations.	`50`
`window_size`	The window size.	`3`

Table 5.3: GeneDiff hyperparameters adapted from the Table 5.2.[46, 50]

5.6.3 SAFE

The parameters for SAFE can be seen in Table 5.4

Parameter Name	Meaning	Value
	i2v model	
min_frequency	The minimal world frequency.	8
batch_size	The training batch size.	128
embedding_size	The embedding vector dimension.	100
skip_window	The window size to one size.	4
num_skips	The number of times an input should be reused to create a label.	2
num_sampled	The number of negative samples.	16
iterations_factor	A factor that specifies how many iterations are made.	50
	Self-Attentive Network model	
batch_size	The training batch size.	250
num_epochs	The number of training epochs.	50
embedding_size	The function embedding dimension.	100
learning_rate	The initial learning rate.	0.001
l2_reg_lambda	A regularization coefficient.	0
rnn_state_size	The RNN state dimension.	50
rnn_depth	The RNN depth.	1
max_instructions	The maximum amount of instructions in a function (truncated when to long).	150
attention_hops	The attention hops number (r).	10
attention_depth	The attention depth (d_a).	250
dense_layer_size	The size of the dense layer (e).	2000
seed	Fixed seed to initialize the random generators.	2

Table 5.4: SAFE hyperparameters adapted from the Table 5.2.[17, 48, 71]

5.6.4 VulSeeker

The parameters for VulSeeker can be seen in Table 5.5

Parameter Name	**Meaning**	**Value**
`P`	The embedding size.	`64`
`D`	The dimension of the input vector.	`8`
`B`	The batch size.	`10`
`lr`	The learning rate.	`0.0001`
`max_iter`	The number of iterations.	`100`
`decay_steps`	The decay step of the learning rate.	`10`
`decay_rate`	The decay rate of the learning rate.	`0.0001`

Table 5.5: VulSeeker hyperparameters adapted from the Table 5.2.[29, 30]

CHAPTER 6

Conclusion

Binary function clone detection is an important topic for binary analysis and reverse engineering. While there is research done in this field, practical and open source implementations are missing and the functionalities that widely used analysis tools provide are lacking in this area.

In this thesis, four function clone detection approaches are selected, implemented and evaluated, with one of them being integrated into commonly used OSS tool. These four are selected out of approaches from the last few years. The implementations are either based on existing open source implementations or complete reimplementations of the approaches. They are all trained and evaluated together by a combination of common interfaces and scripts.

For the OSS tool implementation GeneDiff was chosen, because it initially performed the best and most consistent of all approaches and was outperformed only after a reevaluation, that happened after the implementation was done already. Additionally, it has the advantage that it was easy to adopt for the use case, as saving and restoring repositories was straightforward to implement. The developed common interface makes it convenient for all implementation to be adopted for a OSS tool.

During the evaluation it became clear that not all implementation matched the original research claims, which can result from the selected repository and methodology for evaluation or potential errors in implementing the approaches in this thesis. Nonetheless, does this thesis show that it is often not so simple to verify and adapt the research for real world workflows.

There is still active research and development in the field, so the goal would be, that in the future there will be approaches and open source implementations that make it into OSS tools and into the binary analysis and reverse engineering workflows of people.

List of Figures

2.1 The code from Listing 1 compiled as AArch64 binary displayed as CFG. It has five basic blocks, two function calls and no loops. 5

4.1 Graphical summery of the common interface and script workflow. 24

5.1 The ROC curves of all approaches with the small repository. 31
5.2 The ROC curves of the large repository. 32
5.3 The ROC curves of different optimization levels from the first optimization evaluation. 34
5.4 The ROC curves of different optimization levels from the second optimization evaluation. 35

List of Tables

3.1 Table of comparison between different approaches. This table also includes the chosen approaches from section 4.1.[34] 16

5.1 Precision at Position 1 of the evaluations inspired by Table 3 in [48].[18] . 33
5.2 Asm2Vec hyperparameters adapted from the Asm2Vec repository.[51, 18] 37
5.3 GeneDiff hyperparameters adapted from the Table 5.2.[46, 50] 37
5.4 SAFE hyperparameters adapted from the Table 5.2.[17, 48, 71] 38
5.5 VulSeeker hyperparameters adapted from the Table 5.2.[29, 30] 39

List of Listings

1 C code of a simple functions that displays if a number is even or odd and returns 0 or 1 accordingly. 3
2 The assembly code outputted from the *objdump* utility of the Listing 1 compiled as a x86–64 binary. 4
3 Zignatures that are generated from the x86-64 and AArch64 compiled binaries from Listing 1. 17
4 Radare2 plugin in action on x86-64 and AArch64 compiled binaries from Listing 1. 26
5 Ghidra plugin in action on x86-64 and AArch64 compiled binaries from Listing 1. 27

Acronyms

ACFG attributed control-flow graph. 6, 12

AI artificial intelligence. 7, 11

ANN artificial neural network. 8

API application programming interface. 23, 26

AUC area under the curve. 29, 31–35

BCF bogus control flow. 6, 14

BERT Bidirectional Encoder Representations from Transformers. 8, 13

BGM bipartite graph matching. 12

BN Bayesian network. 15

CBOW continuous bag-of-words. 8, 12

CFG control-flow graph. 4–7, 11–15, 19–25, 30, 35, 43

CLI command-line interface. 9, 25, 26

CNN convolutional neural network. 8, 13

COTS commercial off-the-shelf. 9, 13, 15

CPU central processing unit. 3, 6, 15, 21, 30

CRC cyclic redundancy check. 15

CSV comma-separated values. 24, 30, 36

DFG data flow graph. 6, 21

DNN deep neural network. 8, 12, 21

ELF Executable and Linking Format. 30

FLA control flow flattening. 6, 14

FLIRT Fast Library Identification and Recognition Technology. 15, 16

GCC GNU Compiler Collection. 30, 31

HMM hidden Markov model. 14

HSP hash subgraph pairwise. 15

I/O input/output. 13

ICFG inter-procedural control-flow graph. 6, 12

IR intermediate representation. 20, 23

IRCFG intermediate representation control-flow graph. 23

LCS longest common subsequence. 12, 14

LSFG labeled semantic flow graph. 21

LSH locality-sensitive hashing. 12, 14

LSTM long short-term memory. 11

ML machine learning. 7, 8, 11, 21

MLP multilayer perceptron. 8, 13

MPNN message passing neural networks. 13

NLP natural language processing. 8, 11, 19, 20

NMT neural machine translation. 11

NN neural network. 8, 11, 20

NSA National Security Agency. 9, 15

OSS open-source software. xiii, 2, 9, 15, 19, 25, 41

PV-DM Paragraph Vector-Distributed Memory. 19, 20

RNN recurrent neural network. 8, 11, 20, 38

ROC receiver operating characteristic. 29–35, 43

SDB string database. 16

SRE software reverse engineering. 9

SUB instruction substitution. 6

SVM support vector machine. 12

TADW text-associated DeepWalk. 12

Bibliography

[1] 0x6d696368. Ghidra FID generation. `https://blog.threatrack.de/2019/09/20/ghidra-fid-generator/`, 9 2019. last accessed 7th June 2021.

[2] Martín Abadi, P. Barham, Jianmin Chen, Z. Chen, Andy Davis, J. Dean, M. Devin, S. Ghemawat, Geoffrey Irving, M. Isard, M. Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, D. Murray, Benoit Steiner, P. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqian Zhang. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, OSDI'16, pages 265–283, USA, 2016. USENIX Association.

[3] Toufique Ahmed, Premkumar Devanbu, and Anand Ashok Sawant. Finding inlined functions in optimized binaries. *CoRR*, abs/2103.05221, 2021.

[4] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. FOSSIL: A resilient and efficient system for identifying foss functions in malware binaries. *ACM Transactions on Privacy and Security*, 21(2):1–34, 01 2018.

[5] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 1 2008.

[6] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 583–600. USENIX Association, 8 2016.

[7] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 177–189, 4 2017.

[8] Paula Branco, Luís Torgo, and Rita P. Ribeiro. A survey of predictive modeling on imbalanced domains. *ACM Comput. Surv.*, 49(2):1–50, 8 2016.

[9] CEA IT Security. Miasm. `https://github.com/cea-sec/miasm`. last accessed 4th August 2021.

[10] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. BinGo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 678–689, New York, NY, USA, 11 2016. Association for Computing Machinery.

[11] Philippe Charland, Benjamin C. M. Fung, and Mohammad Reza Farhadi. Clone search for malicious code correlation. In *NATO RTO Symposium on Information Assurance and Cyber Defense (IST-111), Koblenz*, 2012.

[12] Hui Chen. The influences of compiler optimization on binary files similarity detection. In *Proceedings of the 2013 the International Conference on Education Technology and Information System (ICETIS 2013)*, pages 971–975. Atlantis Press, 06 2013.

[13] Zimin Chen and Martin Monperrus. A literature study of embeddings on source code, 04 2019.

[14] Andrew M. Dai and Quoc V. Le. Semi-supervised sequence learning. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[15] Yaniv David, Nimrod Partush, and Eran Yahav. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 79–94, New York, NY, USA, 06 2017. Association for Computing Machinery.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding, 10 2018.

[17] Giuseppe Antonio Di Luna. Safe. `https://github.com/gadiluna/SAFE`. last accessed 21th August 2021.

[18] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charlan. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489, 5 2019.

[19] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. DeepBinDiff: Learning program-wide code representations for binary diffing. In *Network and Distributed System Security Symposium*, 01 2020.

[20] Thomas Dullien. Searching statically-linked vulnerable library functions in executable code. `https://googleprojectzero.blogspot.com/2018/12/searching-statically-linked-vulnerable.html`, 12 2018. last accessed 16th July 2021.

[21] Thomas Dullien and Sebastian Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In *Proceeding of CanSecWest*. Citeseer, 2009.

[22] Chris Eagle. *The IDA Pro Book, 2nd Edition*. No Starch Press, 2 edition, 2011.

[23] Chris Eagle and Kara Nance. *The Ghidra Book*. No Starch Press, Incorporated, 8 2020.

[24] Ata Elahi. *Computer Systems*. Springer International Publishing, Cham, 2018.

[25] Mohammad Reza Farhadi, Benjamin C. M. Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *Proceedings of the 2014 Eighth International Conference on Software Security and Reliability*, SERE '14, pages 78–87, USA, 06 2014. IEEE Computer Society.

[26] Mohammad Reza Farhadi, Benjamin C. M. Fung, Yin Bun Fung, Philippe Charland, Stere Preda, and Mourad Debbabi. Scalable code clone search for malware analysis. *Digital Investigation*, 15(C):46–60, 07 2015. Special Issue: Big Data and Intelligent Data Analysis.

[27] Tom Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 27(8):861–874, June 2006.

[28] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 480–491, New York, NY, USA, 10 2016. Association for Computing Machinery.

[29] Jian Gao. Vulseeker. `https://github.com/buptsseGJ/VulSeeker`. last accessed 22th August 2021.

[30] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 896–899, New York, NY, USA, 9 2018. Association for Computing Machinery.

[31] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1263–1272. PMLR, PMLR, 06–11 Aug 2017.

[32] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[33] Dennis Goodlett. Reverse engineer faster with radare2 signatures. https://hurricanelabs.com/blog/reverse-engineer-faster-with-radare2-signatures/, 5 2020. last accessed 6th June 2021.

[34] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *ArXiv*, abs/1909.11424, 2019.

[35] Hex-Rays. F.L.I.R.T. https://hex-rays.com/products/ida/tech/flirt/. last accessed 3rd June 2021.

[36] Hex-Rays. IDA F.L.I.R.T. Technology: In-Depth. https://hex-rays.com/products/ida/tech/flirt/in_depth/. last accessed 3rd June 2021.

[37] Hex-Rays. IDA Pro. https://hex-rays.com/ida-pro/. last accessed 30th July 2021.

[38] Yikun Hu, Yuanyuan Zhang, Juanru Li, Hui Wang, Bodong Li, and Dawu Gu. Binmatch: A semantics-based hybrid approach on binary code clone analysis. In *ICSME*, pages 104–114. IEEE Computer Society, 08 2018.

[39] He Huang, Amr M. Youssef, and Mourad Debbabi. BinSequence: Fast, accurate and scalable binary code reuse detection. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 155–166, New York, NY, USA, 04 2017. Association for Computing Machinery.

[40] Ivannikov Institute for System Programming of the RAS. ISP Obfuscator. Code obfuscation to protect against vulnerability exploitation. https://www.ispras.ru/en/technologies/isp_obfuscator/. last accessed 3rd September 2021.

[41] justfoxing. Ghidra Bridge. https://github.com/justfoxing/ghidra_bridge. last accessed 14th July 2021.

[42] Wei Khoo, Alan Mycroft, and Ross Anderson. Rendezvous: A search engine for binary code. In *2013 10th IEEE Working Conference on Mining Software Repositories (MSR 2013)*, pages 329–338, Los Alamitos, CA, USA, 05 2013. IEEE Computer Society.

[43] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 Issue 2 of *Proceedings of Machine Learning Research*, pages 1188–1196, Bejing, China, 22–24 Jun 2014. PMLR, PMLR.

[44] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. CCLearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260, 09 2017.

[45] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αDiff: cross-version binary code similarity detection with DNN. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 667–678, New York, NY, USA, 09 2018. Association for Computing Machinery.

[46] Zhenhao Luo, Baosheng Wang, Yong Tang, and Wei Xie. Semantic-based representation binary clone detection for cross-architectures in the internet of things. *Applied Sciences*, 9(16):3283, 8 2019.

[47] maijin and pancake. *The Official Radare2 Book*.

[48] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. Safe: Self-attentive function embeddings for binary similarity. In *Proceedings of 16th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 309–329, 6 2019.

[49] Tomas Mikolov, Kai Chen, Greg S. Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *Proceedings of ICLR Workshop*, volume abs/1301.3781, pages 1–12, 01 2013.

[50] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, volume 26 of *NIPS'13*, page 3111–3119, Red Hook, NY, USA, 10 2013. Curran Associates Inc.

[51] Sirui Mu. asm2vec. `https://github.com/Lancern/asm2vec`. last accessed 20th August 2021.

[52] National Security Agency. Function ID. `https://github.com/NationalSecurityAgency/ghidra/blob/b6ba209ed796343880327bc3337355c303b760cd/Ghidra/Features/FunctionID/src/main/help/help/topics/FunctionID/FunctionID.html`. last accessed 7th June 2021.

[53] National Security Agency. Ghidra. `https://ghidra-sre.org/`. last accessed 15th July 2021.

[54] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. Association for Computing Machinery.

[55] Lina Nouh, Ashkan Rahimian, Djedjiga Mouheb, Mourad Debbabi, and Aiman Hanna. BinSign: Fingerprinting binary functions to support automated analysis of code executables. In Sabrina De Capitani di Vimercati and Fabio Martinelli, editors,

ICT Systems Security and Privacy Protection - 32nd IFIP TC 11 International Conference, SEC 2017, Rome, Italy, May 29-31, 2017, Proceedings, volume 502 of *IFIP Advances in Information and Communication Technology*, pages 341–355, Cham, 05 2017. Springer International Publishing.

[56] Paladion. Code Obfuscation Part 3 - Hiding Control Flows. `https://www.paladion.net/blogs/code-obfuscation-part-3-hiding-control-flows`, 10 2005. last accessed 3rd September 2021.

[57] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. Probabilistic naming of functions in stripped binaries. In *Annual Computer Security Applications Conference*, ACSAC '20, page 373–385, New York, NY, USA, 12 2020. Association for Computing Machinery.

[58] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724. IEEE, 5 2015.

[59] Python Packaging Authority. Pipenv. `https://pipenv.pypa.io/en/latest/`. last accessed 5th August 2021.

[60] radare org. radare2. `https://www.radare.org/n/radare2.html`. last accessed 14th July 2021.

[61] radare org. radare2-rlang. `https://github.com/radareorg/radare2-rlang`. last accessed 14th July 2021.

[62] radare org. radare2/cmd_zign.c. `https://github.com/radareorg/radare2/blob/247b509edcc48007eee4b695f2438c527ebf5197/libr/core/cmd_zign.c`. last accessed 6th June 2021.

[63] radare org. sdb: Simple and fast string based key-value database with support for arrays and json. `https://github.com/radareorg/sdb`. last accessed 6th June 2021.

[64] Kimberly Redmond. An instruction embedding model for binary code analysis. 2019.

[65] Kimberly Redmond, Lannan Luo, and Qiang Zeng. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis, 2018.

[66] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3 edition, 2009.

[67] Noam Shalev and Nimrod Partush. Binary similarity detection using machine learning. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, PLAS '18, page 42–47, New York, NY, USA, 2018. Association for Computing Machinery.

[68] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings 2015 Network and Distributed System Security Symposium*, volume 1, pages 1–1. Internet Society, 01 2015.

[69] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 05 2016.

[70] James T. Streib. *Guide to Assembly Language*. Undergraduate Topics in Computer Science. Springer International Publishing AG, Cham, 2 edition, 2020.

[71] TensorFlow. word2vec_basic.py. `https://github.com/tensorflow/tensorflow/blob/876fc8dc4f40c75914dbfcb0a809feaf81be7412/tensorflow/examples/tutorials/word2vec/word2vec_basic.py`. last accessed 20th August 2021.

[72] Annie H. Toderici and Mark Stamp. Chi-squared distance and metamorphic virus detection. *Journal of Computer Virology and Hacking Techniques*, 9(1):1–14, 2 2013.

[73] Vector 35. Binary Ninja. `https://binary.ninja/`. last accessed 30th July 2021.

[74] Maximilian von Tschirschnitz. Library and function identification by optimized pattern matching on compressed databases: A close to perfect identification of known code snippets. In *Proceedings of the 2nd Reversing and Offensive-Oriented Trends Symposium*, ROOTS '18, pages 1–12, New York, NY, USA, 11 2018. Association for Computing Machinery.

[75] Shuai Wang and Dinghao Wu. In-memory fuzzing for binary code similarity analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 319–330. IEEE Press, 10 2017.

[76] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 363–376, New York, NY, USA, 10 2017. Association for Computing Machinery.

[77] Hongfa Xue, Shaowen Sun, Guru Venkataramani, and Tian Lan. Machine learning-based analysis of program binaries: A comprehensive study. *IEEE Access*, 7:65889–65912, 05 2019.

[78] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. In *The*

Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020, volume 34 Issue 1, pages 1145–1152. Association for the Advancement of Artificial Intelligence (AAAI), 4 2020.

[79] Yijia Zhang, Hongfei Lin, Zhihao Yang, Jian Wang, and Yanpeng Li. Hash subgraph pairwise kernel for protein-protein interaction extraction. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 9(4):1190–1202, 7 2012.

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