TU WIEN Informatics

Identification of Obfuscated Function Clones in Binaries using Machine Learning

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Michael Pucher, BSc
Matrikelnummer 01425215

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Mitwirkung: Dipl.-Ing. Dr.techn. Georg Merzdovnik

Wien, 7. Dezember 2021

Michael Pucher	Edgar Weippl

TU WIEN Informatics

Identification of Obfuscated Function Clones in Binaries using Machine Learning

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Michael Pucher, BSc
Registration Number 01425215

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Assistance: Dipl.-Ing. Dr.techn. Georg Merzdovnik

Vienna, 7th December, 2021

Michael Pucher	Edgar Weippl

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

Erklärung zur Verfassung der Arbeit

Michael Pucher, BSc

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, 7. Dezember 2021

Michael Pucher

Danksagung

Danke an Kexin Pei, Hauptautor von TREX[1], für das Beantworten meiner Anfragen und die Bereitstellung von Benchmark-Ergebnissen. Danke an Rudolf Mayer, dafür, dass ich eine Woche lang GPU-Cycles auf eurem Server verbrennen durfte, sowie für Input in Bezug auf den Machine Learning Teil der Arbeit. Danke an Georg Merzdovnik für deine Geduld mit mir.

Acknowledgements

Thanks to Kexin Pei, first author of TREX[1], for answering some questions about their paper and providing me with benchmark results. Thanks to Rudolf Mayer for letting me burn a week worth of GPU hours on their server, as well as providing input and discussion on the machine learning approaches and metrics. Thanks to Georg Merzdovnik for always putting up with me.

Kurzfassung

Mit steigender Beliebtheit von höheren Sprachen für Malware-Entwicklung, wie C# unter Windows, oder Go für Plattform-übergreifende Malware, steigt auch deren Komplexität und Funktionsumfang. Zusätzlich wird Obfuscation eingesetzt, um die bösartigen Absichten vor Virus-Scannern zu verbergen und den nötigen Reverse Engineering Aufwand für menschliche Analysten zu erhöhen. Eine Möglichkeit, diesen Aufwand gering zu halten, ist Function Clone Detection.

Wie bei jedem anderen Software Projekt wird auch von Malware Code wiederverwendet und bestehender Code leicht abgeändert. Kann eine Binärfunktion als bereits bekannt, oder ähnlich zu bereits bekannten Funktionen betrachtet werden, verringert das die Zeit, die für die Analyse benötigt wird. Abseits von Malware kann derselbe Function Clone Detection Mechanismus dazu verwendet werden, Varianten von Funktionen zu finden, die bereits bekannte Sicherheitslücken aufweisen, wodurch sich diese Technik als besonders nützlich erweist. In dieser Arbeit wird ein Ansatz für das auffinden von Obfuscated Function Clones unter dem Namen Ofci vorgestellt. Dieser baut auf jüngsten Erkenntnissen im Bereich von Function Clone Detection, mithilfe von Machine Learning, auf.

Mit Hilfe des Albert Transformers, einer auf Sparsamkeit optimierten Variante des Bert Sprachverarbeitungs-Modells, kann die Assembler-Sprache wie natürliche Sprache behandelt werden. Das führt dazu, dass Ofci trotz einer Reduktion der Modell-Parameter um 83% nur einen durchschnittlichen Abfall von 7% in Bezug auf ROC-AUC Scores hinnehmen muss. Um sich speziell dem Problem von Obfuscation anzunehmen, analysiert Ofci die Effekte von Funktionsaufrufen auf die Funktions-Suche und wagt sich an Code, der mittels Virtualisierung verschleiert wurde, heran. Dazu verfolgt Ofci einen dynamischen Ansatz, der mittels Instruction Traces versucht Funktionen in virtuellem Code zu erkennen. Allerdings muss Ofci auch Rückschläge im Hinblick auf die Präzision der Funktions-Suche hinnehmen und erörtert systematisch die Gründe dafür. Unabhängig vom Machine Learning Ansatz stellt Ofci auch ein eigenes Framework zur Extraktion und Verarbeitung von Binärfunktionen dar. Durch die Implementierung dieser Funktionen als Plug-In für Ghidra, einer frei verfügbaren Reverse Engineering Umgebung, kann Ofci einen durchgehenden Ansatz für die Analyse von Binärfunktionen anbieten, der an jeder Stelle nur Open-Source Software benützt. Durch den Headless Analyse-Modus kann dieses Framework Massen an Binärdateien parallel verarbeiten.

Abstract

With widespread use of higher-level languages for malware, such as C# on Windows or Go for cross-platform malware, the complexity and functionality of malware is ever-increasing. Additionally, obfuscation is used to hide the malicious intent from virus scanners and increase the time it takes for a human analyst to reverse engineer the binary file. One way to minimize this effort is function clone detection.

Like any other software engineering project, malware reuses code and modifies already existing code. Detecting whether a binary function is already known, or similar to an existing function, can reduce the time needed to analyze it. Outside of malware, the same function clone detection mechanism can be used to find vulnerable versions of functions in binaries, making it a powerful technique. This thesis introduces an approach for the detection of obfuscated function clones, called Ofci, building on recent advances in machine learning based function clone detection.

Using the Albert transformer, a size-optimized version of the Bert natural language processing model, on textual disassembly instead of language, Ofci can achieve an 83% model size reduction in comparison to state-of-the-art approaches, while only causing an average 7% decrease in the ROC-AUC scores of function pair similarity classification. To additionally tackle the issue of obfuscation, Ofci analyzes the effect of known function calls on function similarity and applies function similarity classification on code obfuscated through virtualization. Instead of trying to match virtualized function pairs statically, Ofci tries to perform function clone detection based on traces of the virtualized function, as a cheap form of dynamic analysis. However, the reduced model size comes at the cost of precision for function clone search and the evaluation of Ofci discusses the reasons for this and other pitfalls of building function similarity detection tooling.

Besides evaluating the machine learning approach, Ofci also establishes a new framework for the extraction and processing of binary functions. By implementing this functionality as a Ghidra plugin, Ofci offers an end-to-end approach for binary function analysis where every part of the pipeline is open-source. Through headless analysis, this framework can scale to analyzing large quantities of binary executables in parallel.

Contents

CHAPTER 1

Introduction

In order for any programming language to be executed by a CPU, it has to be translated, through possibly multiple steps of compilers and interpreters, into the machine code of the respective CPU architecture. This ranges from high-level languages, like JavaScript or Haskell, to low-level languages like C, and the more layers of abstractions are involved, the less information from the original code is preserved in machine code. However, sometimes it can become necessary to traverse this translation in the other direction, through reverse engineering, to determine the original intention of the programmer. An example for such an occasion would be a domain name generation algorithm in a malware binary; in order to stop the malware from spreading, the algorithm has to be reverse engineered.

```c
void hello() {
    printf("Hello World!\n");
}

int main() {
    hello();
    return 0;
}
```

Listing 1: A basic "Hello World" example in C.

But reverse engineering is not a simple task and the time to success depends on just how much information about the program has been preserved in its machine code form. Luckily, operating systems do not directly execute native machine code from a file, but have their own file formats for containing code, such as ELF on Linux, which can store additional meta-information needed for the code to run, or needed for a developer to efficiently debug the program. Listing 1 contains a simple "Hello World" program written

in C. Just how much information from this program is retained in the executable binary file? During development or for finding issues in production a programmer would usually compile the C source code to binary using debug information. The result of the generated binary can be seen in Listing 2.

```
1 sym.hello ();
2     /*55           */  push rbp                ; main.c:4 void print_hello() {
3     /*4889e5       */  mov rbp, rsp
4     /*488d3dc00e00.*/  lea rdi, str.Hello_World ; main.c:5 printf("Hello World!");
5     /*e8e7feffff   */  call sym.imp.puts       ; int puts(const char *s)
6     /*90           */  nop                     ; main.c:6 }
7     /*5d           */  pop rbp
8     /*c3           */  ret
```

Listing 2: Disassembled ELF file with debug information.

The listing shows how the C code has been translated into `x86_64` assembly code, and the corresponding raw bytes in hexadecimal. The debug information adds some additional hints to the disassembly: The name of the function, `hello()`, is preserved and comments on the side point the disassembler to the source code file, allowing the matching between generated instructions and original source code.

```
1 sym.hello ();
2     /*55           */  push rbp
3     /*4889e5       */  mov rbp, rsp
4     /*488d3dc00e00.*/  lea rdi, str.Hello_World
5     /*e8e7feffff   */  call sym.imp.puts
6     /*90           */  nop
7     /*5d           */  pop rbp
8     /*c3           */  ret
```

Listing 3: Disassembled ELF file without debug information.

Realistically, no production binary will be distributed with debug information and it will be missing in any task that requires reverse engineering to begin with (using debug information when working with malicious binaries can even be harmful [2]). In the best realistic case, a program will look like Listing 4 when disassembled. There is no more reference to the original source file, but the functions and global variables/constants, like the "Hello World!" string, will still be named. Proprietary binaries will not have this information, as a minimum effort to protect intellectual property. Through the process of *stripping*, as much information regarding names is removed from the executable file format, to the point where the program is still able to function. Besides function and global variable names, a lot of information regarding the memory mapped sections can be stripped from an ELF file and still have it function as a working executable. The resulting disassembly after the binary file has been stripped can be seen in Listing 4.

The disassembler now no longer knows the function name and has to generate or infer a name, usually based on the address where the function is located. The name of the global string constant has been removed as well, and only the address remains; it is now no longer visible what the behavior of the program is at first glance.

```
1  fcn.00001139 ();
2      /*55              */  push rbp
3      /*4889e5          */  mov rbp, rsp
4      /*488d3dc00e00.*/  lea rdi, [0x00002004]
5      /*e8e7feffff      */  call sym.imp.puts
6      /*90              */  nop
7      /*5d              */  pop rbp
8      /*c3              */  ret
```

Listing 4: Disassembled stripped ELF file.

This is the least amount of information left in a normal binary, and what most deployed binary programs look like. To take this one step further, the program has been modified in Listing 5. To make it harder to reverse engineer, a simple obfuscation technique to thwart linear disassemblers has been added.

```
1  fcn.00001139 ();
2      /*55              */  push rbp
3      /*4889e5          */  mov rbp, rsp
4      /*488d3dc00e00.   */  lea rdi, [0x00002004]
5      /*eb02            */  jmp 0x1148
6      /*4889e8          */  mov rax, rbp
7      /*e7f0            */  out 0xfe, eax
8      /*ff              */  invalid
9      /*ff              */  invalid
10     /*90              */  nop
11     /*5d              */  pop rbp
12     /*c3              */  ret
```

Listing 5: Disassembled ELF file, obfuscated to confuse linear disassemblers.

The output in the listing is what a linear disassembler would produce, but does not mirror what is actually executed during runtime. The bytes highlighted in green correspond to the original `call sym.imp.puts()` instruction, but the disassembler can't see this, because garbage bytes have been inserted to make the disassembler assume a different instruction. The inserted yellow highlighted jump causes the execution to jump into the middle of the garbage instruction, which will execute the original call again. This is only a simple obfuscation trick, but it already takes a bit of thinking to see the original program when not using the correct disassembler. If the number of obfuscation is increased and the complexity of these obfuscations rises, so will the time it takes to analyze it and find

the original program. This brings up the question of how a program with thousands of heavily obfuscated functions can still be reverse engineered in a reasonable amount of time and one of many answers to this is *Function Clone Detection*, the topic of this thesis.

1.1 Motivation

The amount of Go [3] malware in active use by advanced persistent threats and other online crime groups has increased by a staggering amount. [4] Go allows straightforward cross-compilation and is significantly harder to reverse engineer than, e.g., C programs by default, while still providing low-level access to the system and a versatile ecosystem. The greatest factor in what makes Go hard to reverse engineer is the default static compilation; instead of relying on system libraries to be installed, every dependency in Go is compiled into one large binary. A version of the "Hello, World!" example from earlier in Go can be seen in Listing 6.

```
1  package main
2
3  import "fmt"
4
5  func hello() {
6      fmt.Println("Hello World!")
7  }
8
9  func main() {
10     hello()
11 }
```

Listing 6: Same version of the "Hello, World!" program in Go.

While the corresponding C program produces, without any minification, an executable binary with a size of 16KB, the Go program produces a 1.6MB binary, due to static linking. Analyzing the generated binary with Ghidra [5] shows 1471 function symbols listed in the binary. Go allows generating stripped binaries with a compiler switch; the difference between the function listings of the unstripped and stripped version can be seen in Figure 1.1. Not only is Ghidra not able to assign *any* function names, because there are no names in the binary, it is also not able to even detect all the functions, only reporting 1099 identified functions. If only one of these functions performed malicious actions and all functions have been obfuscated, finding the function in question becomes a seemingly impossible task.

To reduce the manual analysis load for reverse engineers, parts of an executable binary, which have already been identified or analysed in previous iterations, or are known from different binaries, should ideally be excluded from analysis. This is made possible by the practice of code reuse for similar tasks, which is also prevalent among malware. [6, 7]

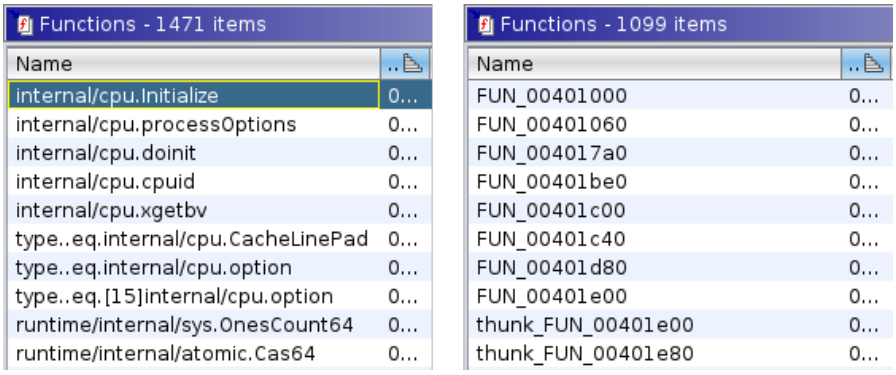

Figure 1.1: Function listing of Go "Hello, World!" in Ghidra, before and after stripping.

Additionally, as in the case of Go, large parts of an application are part of a standard library or are utility functions. These can already be analyzed before analyzing the malware itself, because standard library functions are freely available in different binary formats, versions and in source code. Ideally, a classifier would be trained on this data and then just be able to detect the standard library functions in malware.

1.2 Problem Statement

Methods for finding sections of code reuse in binaries can be distinguished by granularity, with one approach being function level granularity. This approach is called the *function clone identification* problem, which can be defined as such: Given a binary without symbol/debug information and a repository of extracted functions with symbol/debug information, identify similar functions in the binary and assign symbols to them.

In its simplest form, function clone identification assigns names to functions in a binary without symbol information, thus allowing a reverse engineer to recognize certain functions as e.g. library functions that have been statically linked into the binary. However, predicting debug information in stripped binaries has also been shown to be feasible. [8]

The function clone identification problem can further be split into subproblems. The detection of function boundaries in a binary is a hard problem on current binary architectures and is as such a prerequisite for identification. As promising solutions for this exist, e.g. [9], the focus of function clone identification lies on the following subproblems:

- Generate/maintain a database of function signatures.
- Looking up unidentified functions in the database, to find likely candidates.

Numerous approaches for this exist, with machine learning techniques gaining popularity in more recent approaches. [10, 11, 12] Machine learning based on binary features is well suited for this task, as it is inherently *fuzzy* in its nature. As function clone identification

is intended to find functions that are not exact matches, learning similarity of functions allows matching functions that share certain features. This is backed up by a recent survey on binary similarity [13], showing that approaches based on machine learning perform well in comparison to others. The same survey also discusses the shortcomings of current function clone identification approaches when dealing with obfuscation. The goal of this thesis is therefore to focus on improving function clone identification in the presence of obfuscation.

1.3 Contributions

The aim of this work is to improve upon the existing capabilities of function clone detection when dealing with heavily obfuscated executable binaries. The focus is solely on function clone detection and algorithms for extracting the function data itself are not explored. Extracting the function data and the data processing pipeline requires a significant engineering effort, and the needed tools are built as part of this thesis. The main contributions of this thesis are:

- **Development of an end-to-end function clone identification framework.** Existing approaches have not published their whole data processing pipeline, or rely on commercial tools, like IDA Pro [14], for feature extraction. This thesis presents Ofci, an end-to-end framework, built from publicly accessible and open source tools. Data is exported from Ghidra [5], processed in the machine learning pipeline and finally imported again.

- **A reduced and improved machine learning model for function clone identification.** As recent related models come with high computational complexity, Ofci aims to reproduce similar performance at a student budget. In addition, Ofci uses API calls, like system or library calls, as an additional feature vector that can be applied iteratively, i.e. detection of a function clone can improve the identification of other functions calling this clone.

- **Matching state-of-the-art performance of function similarity in the presence of basic obfuscation.** Ofci can match state-of-the-art solutions for function similarity classification when obfuscations like bogus control-flow, control-flow flattening, instruction substitution etc. are present.

- **Analysis of identifying functions in the presence of virtual machine obfuscation** Ofci presents the *first* approach to function clone detection, when functions are obfuscated using virtualized code. This is implemented by detecting the function clones in recorded execution traces of the VM. While the results themselves are not adequate, a detailed analysis of possible pitfalls and research directions is given. This helps future work from repeating the same mistakes and tackle the underlying issues.

- **Analysis of reduced function clone search performance.** While the ROC-AUC scores reported by Ofci are comparable to the state of the art, the function clone search shows reduced performance. This thesis gives an analysis into possible issues and the consequences for other function clone search implementations.

CHAPTER 2

Background

In order to understand contributions to the field of binary code similarity or function clone detection, background knowledge of different fields is required. In the context of this thesis, obfuscation and modern machine learning techniques play a crucial role as well. This chapter will cover the basic concepts of these fields and introduces some terminology that will be required in the following chapters.

2.1 Function Clone Identification

The central part of this thesis discusses a contribution to the field of binary code similarity, i.e. finding similarities between different segments of executable binary code, usually machine code instructions. To find such similarities, a definition of what constitutes *similarity* between two segments of binary code is required. However, there is no one specific definition for binary code similarity, as this definition might change in relation to the problem to be solved. For example, a program might be syntactically similar, but might show different semantic behavior during runtime and vice-versa. For the purpose of this thesis, as is also the case in related works, [10, 11, 1, 15] we define executable binary code to be similar if it is semantically similar, i.e.

$$B_i \sim B_j$$

means B_i is similar to B_j, if the code segments show similar runtime behavior. It is worth noting that similarity is still somewhat vague: In general the problem of proving two binary code segments to be semantically equivalent is undecidable, and equivalence might not be desirable. One use case of binary similarity is to find known vulnerable code in programs, but as the code directly surrounding the vulnerability might have changed, the segment used to compare against might not be semantically equivalent anymore. Therefore, the similarity relation should be fine-grained enough to distinguish

two segments of binary code, but coarse enough to capture a property of interest. To address this issue, the binary similarity problem can be constrained to what is called function clone identification. In virtually all modern instruction set architectures, segments of binary code can be grouped into functions to allow code reuse. These functions have input/output parameters and a context containing local variables, usually on the call stack, making them a prime target for studying execution semantics in a restricted segment of binary code. Function clone identification is the binary similarity problem applied to functions:

$$\begin{aligned}
&\mathbb{L} = \text{Domain of possible labels} \\
&\mathbb{B} = \text{Domain of arbitrary length executable binary code} \\
&F = \{(l, b) | l \in \mathbb{L}, b \in \mathbb{B}\}
\end{aligned}$$

$$\begin{aligned}
\forall F_i, F_j : b_i \sim b_j &\implies F_i \sim F_j \\
\forall F_i, F_j : F_i \sim F_j &\implies l_i = l_j \\
\forall F_i, F_j : F_i \sim F_j &\implies F_j \text{is a clone of} \sim F_j
\end{aligned}$$

Given the functions F_i and F_j, determine whether these are similar, i.e. F_i similar to F_j. As per the formula above, functions are defined as similar if their binary code is similar, i.e. semantically equivalent. If this condition holds, F_i is called a *clone* of F_j, and the functions have the same label/name. Besides providing a natural code boundary for comparison, restricting binary similarity to the scope of functions reduces it to an information retrieval problem: Given a repository R_F, find an ordered range of functions F_i to F_j that are similar to a query function F_q. Treating binary code as a language then opens the possibility of applying modern information retrieval algorithms, which have been designed for, e.g., full text search and machine translation.

While concentrating on function clone detection offers some simplifications to the binary code similarity problem, it also comes with the pitfall of detecting the existence of functions in the first place. Detecting the boundaries of a function in stripped binaries is in general undecidable, but recent research [9, 16] has shown that in practice, a large portion of function boundaries can be recovered. As the detection of function boundaries is a separate research topic only marginally related to function clone detection, this thesis will not discuss it and assume knowledge of all relevant function boundaries.

2.2 Obfuscation

In terms of binary code, obfuscation means transforming code in such a manner that it becomes hard to analyze, while still preserving the intended execution semantics. There are many possible use cases, benevolent or malicious: Companies can deploy obfuscation

as an additional layer of security (i.e. *security through obscurity*), or to protect intellectual property, or it can be used in malware to make analysis/attribution harder for an analyst. Especially in the latter case, finding ways to improve on deobfuscation manifests as an important task. As obfuscation increases the code size and the runtime of the code, it not only poses a challenge to human analysts, but also slows down analysis algorithms. As program analysis problems have a tendency to be undecidable as soon as the analysis concerns non-trivial program properties, it is easy for an obfuscation method to trick program analysis heuristics, while it is hard for new heuristics to take obfuscation methods into account.

With obfuscation techniques being an active research field in itself, there are countless obfuscation methods to choose from. To restrict the scope of this thesis, the obfuscators being used to evaluate the presented techniques are Tigress [17], Obfuscator-LLVM [18], and by extension Hikari [19], which is a port of Obfuscator-LLVM. The last two have been chosen due to references in recent works, allowing the comparison of evaluation results. [10, 1] Tigress has been chosen due to it being a well-known academic project, referenced by a large number of publications. The following sections give an overview of these tools and the obfuscations used in this thesis.

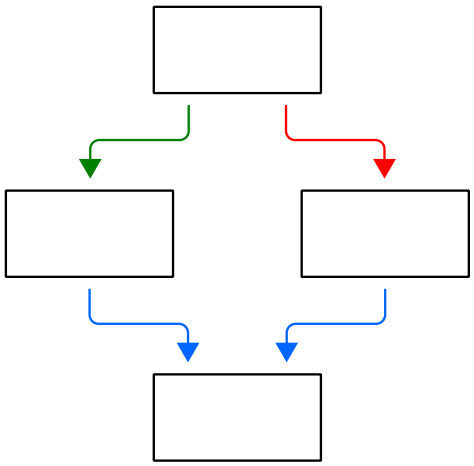

Figure 2.1: Minimal example of a Control-Flow Graph (CFG).

2.2.1 Obfuscator-LLVM

Built on top of LLVM, Obfuscator-LLVM can be used as a drop-in replacement for other compilers, without having to adjust the build process. This allows studying obfuscation of full software projects instead of hand-crafted examples. Unfortunately, Obfuscator-LLVM has been last updated 2017 on LLVM 4.0; to modernize the project, Hikari has been forked off Obfuscator-LLVM and added additional obfuscation passes, while also

improving the existing obfuscation passes. In accordance with related work [10, 1] a number of passes has been selected for evaluation. To highlight the changes to the program made by these passes, Figure 2.1 shows a minimal example of a control flow graph (CFG) for a simple function. The nodes of a CFG are called basic blocks: It contains all instructions up to the first instruction that initiates a control-flow transfer, e.g. a conditional jump instruction, with the exception of function calls. In the figure, the top basic block transfers control through a true and a false branch, indicated by the green and red arrow, as it would be created by a simple condition check. The following two basic blocks both return to the bottom block, where control-flow is transferred either through fall-through or unconditional jumps.

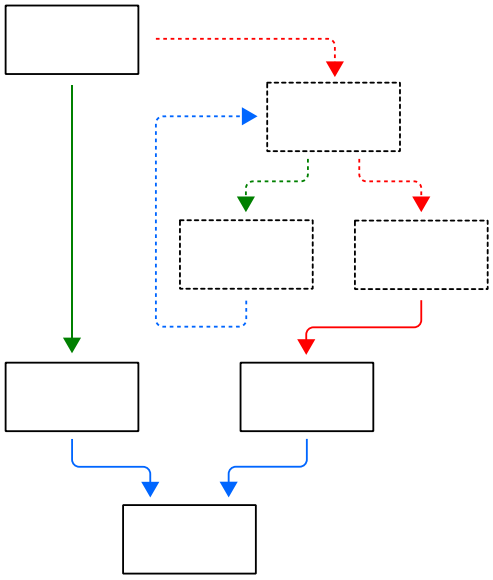

Figure 2.2: A CFG with inserted bogus control flow.

When applying **Bogus Control Flow (BCF)**, the CFG is changed by inserting new basic blocks and constructs that make it harder to find the relevant basic blocks of the original function. This can be seen in Figure 2.2, where the added control flow is highlighted with dotted lines. In this case, the added construct is a simple loop with one path leading back to the original control flow. As pictured in the example it could also be introduced by a compiler optimization or a minor code change, therefore the BCF pass has to produce larger and more complex constructs in order to obfuscate the original control flow. To keep the runtime overhead of this obfuscation minimal, the added control flow is ideally not executed and the full graph is only visible to static analysis tools. This is achieved by hiding the jump conditions of the BCF behind *opaque predicates*, complex expressions that always evaluate to the same outcome during runtime, but are hard to analyze statically. **Basic Block Splitting (BBS)** works in a similar manner to BCF,

but while BCF leaves the original CFG intact and only adds new control flow, BBS breaks apart larger basic blocks. The goal of BBS is to create a large number of new basic blocks, which can then be separated using bogus control flow again, slowing down analysis.

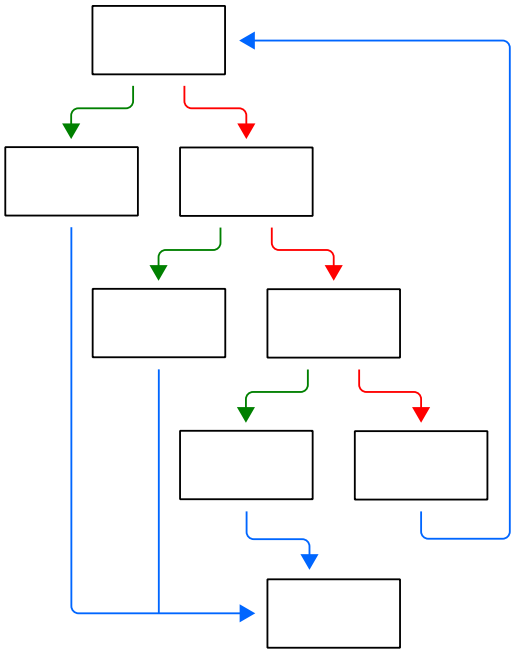

Figure 2.3: A CFG with flattened control flow.

Obfuscation methods like BCF or BBS expand the base CFG, but leave the ordering of the original basic blocks intact, i.e. if the CFG has no loops and can be sorted using topological sort, the ordering of the original basic blocks in the output will be the same. This is also necessary to preserve program semantics, as basic blocks cannot be reordered without performing analysis on whether the reordering of contained instructions is possible. However, the order and structure of the original CFG can be obfuscated from static analysis by abstracting the control flow of the CFG into data flow. This method of obfuscation is called **Control Flow Flattening (CFF)**, as it flattens the control flow into a structure that is just a case distinction on a single variable. In the first step, every basic block or subsection of a basic block is assigned a random number. One variable is declared in the function and used to store the random number assigned to a basic block. In the case of CFF, usually a chain of conditional jumps is used to select the correct basic block based on the assigned random number, turning the original CFG into something looking like Figure 2.3. As seen in the figure, the chain repeats until it either loops back to the starting block or leaves the function at the exit block. Original control flow transitions are not visible: Instead of performing jumps at the end of a basic

block, the actual transition is made by assigning the random number of the next basic block to the function variable. The static CFG is now no longer useful for heuristics or human analysts, as the CFG of every function will have the same structure, with the length of the main condition chain varying.

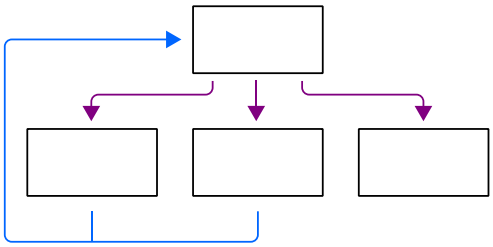

Figure 2.4: A CFG showing register-Based indirect branching.

With the original control flow completely obfuscated, CFF is already a potent obfuscation method on its own. However, the original basic blocks and code is still discovered and present in the CFG, and the long condition chains are not ideal for performance, as it takes $\mathcal{O}(n)$ to find the target block in the worst case. So far, the depicted CFGs have only shown basic blocks with one, i.e. the fallthrough or unconditional jump, or two, i.e. a conditional jump, outgoing edges. Additionally it is possible to have an arbitrary number of outgoing edges: The target address for a jump can be stored in a lookup table and loaded into a register at runtime, with the **Register-Based Indirect Branching (RBIB)** obfuscation pass making use of this. Instead of assigning random numbers to basic blocks, the addresses of the blocks are stored in a potentially randomized lookup table. Then, control-flow is transferred by passing the index into the lookup table and loading the basic block address into a register, performing an indirect branch using the register value. This obfuscation method is not part of the original Obfuscator-LLVM and has been added as an improvement in Hikari. [19] If a static analysis program is able to determine the jump targets, it will look like the CFG shown in Figure 2.4, with one basic block as *dispatcher* passing control to the others. Due to Hikari jumping back to the dispatcher through pushing the address and performing a return, e.g. Ghidra [5] is not able to show the basic blocks as part of the CFG, but will only show the dispatcher block; it can however still locate and disassemble the basic blocks. Within the context of this thesis, files obfuscated with the **Instruction Substitution** pass are used as well. This pass is different from the others, as it does not change the underlying CFG, but only performs changes on the instruction level. Instead of loading constants or performing simple operations directly, the instructions are replaced with a more complex chain of instructions. The newly generated expression does not necessarily have to follow the same semantics as the original version, it only has to produce the same output for all expected inputs. It is also possible to use vector registers, e.g. AVX registers, since analysis tools as well as human analysts struggle with simplifying vector operations.

2.2.2 Tigress

Tigress [17] is a source-to-source obfuscator originally created by Christian Collberg at the University of Arizona. It is widely used in academic publications to evaluate the effectiveness of binary analysis techniques on obfuscated code, and comes with a variety of different obfuscation passes. Unfortunately, compared to Obfuscator-LLVM, Tigress cannot be used as a drop-in replacement, making it difficult to apply on larger projects. Tigress supports all of the less potent obfuscations, which are already covered by Obfuscator-LLVM, e.g. bogus control flow, and these are therefore not used again in Tigress obfuscation experiments. Additionally, obfuscations that split or merge entire functions are avoided, as the focus of this thesis is on function similarity, and self-modifying code is avoided, due to requiring extensive dynamic analysis.

```
1 uint64_t fn(uint64_t a, uint64_t b, uint64_t c) {
2     return a + 2 * b + 3 * c;
3 }
```

Listing 7: A simple C function before virtualization.

Current commercially viable obfuscators [20, 21] rely on multiple levels of **virtualized code**. Instead of running the originally intended assembly code, code is transformed into bytecode that is run in a virtual machine (VM). This form of obfuscation has been proven robust against analysis, as nothing of the original code remains and finding/understanding the generated bytecode requires analysis of the virtual machine first. However, obfuscation techniques can be combined, allowing the operation handlers of the virtual machine to be obfuscated as well. While there exist different approaches to VM deobfuscation, there appears to be no prior work regarding binary code similarity of code executed in the VM; a discussion of related work can be found in section 3.3. As an example for how invasive code virtualization is, Listing 7 shows a very simple C function taking three parameters and performing arithmetic operations. There is no pre-existing control-flow, meaning O-LLVM will have a hard time inserting bogus control flow or performing CFF; a small function like this will likely not trigger an obfuscation heuristic or is going to be inlined. Tigress takes a list of functions to be obfuscated and when specifying this toy function, it will generate the code shown in Listing 8. Looking at the generated code, there is no trace of the original toy function, with the exception of the function signature. In place of the formerly simple arithmetic expression there is now a *while* loop with a series of switch statements, a characteristic construct of code obfuscated through virtualization. This code is a small virtual machine (hence the term virtualized code): It contains a stack, a stack pointer, storage for local variables, a program counter and a number of different opcodes. The opcodes are random and generated in a way that tries to hide the original intention of the code. The code that actually controls this function, i.e. the bytecode, is stored at a different location in memory and the control-flow of the virtual machine cannot be observed from static analysis.

```c
uint64_t fn(uint64_t a , uint64_t b , uint64_t c )
{
  char _1_fn_$locals[32] ;
  union _1_fn_$node _1_fn_$stack[1][32] ;
  union _1_fn_$node *_1_fn_$sp[1] ;
  unsigned char *_1_fn_$pc[1] ;

  {
  _1_fn_$sp[0] = _1_fn_$stack[0];
  _1_fn_$pc[0] = _1_fn_$array[0];
  while (1) {
    switch (*(_1_fn_$pc[0])) {
    case _1_fn__store_unsigned_long$left_STA_0$right_STA_1:
    (_1_fn_$pc[0]) ++;
    *((unsigned long *)(_1_fn_$sp[0] + 0)->_void_star) = (_1_fn_$sp[0] + -1)->_unsigned_long;
    _1_fn_$sp[0] += -2;
    break;
    case _1_fn__constant_unsigned_long$result_STA_0$value_LIT_0:
    (_1_fn_$pc[0]) ++;
    (_1_fn_$sp[0] + 1)->_unsigned_long = *((unsigned long *)_1_fn_$pc[0]);
    (_1_fn_$sp[0]) ++;
    _1_fn_$pc[0] += 8;
    break;
    /* ... */
    }
  }
}
```

Listing 8: C function after virtualization.

Another form of potent obfuscation provided by Tigress is **EncodeArithmetic**. This obfuscation mode strengthens arithmetic expressions by generating what is known as a mixed boolean-arithmetic (MBA) expression. The code in Listing 9 shows how the previous toy example has been transformed into a complex MBA expression. While this does appear to be simpler than virtualized code, it is far from trivial to derive the original intent and expression from the MBA expression. Deciphering MBA expressions is an active field of research, with approaches like MBA-Blast [22] claiming the analysis of MBA expression to still only be at the starting point.

2.3 Machine Learning

In many cases of hard or undecidable problems, machine learning is often leveraged to achieve approximate solutions. As shown in section 2.1, function clone identification can be formulated as an information retrieval problem and modern information retrieval techniques rely on complex machine learning model architectures, such as Bert. [23] Variations of Bert have been used in related work [1, 16] and are adopted for the solution presented in this thesis as well. As machine learning is its own discipline, this

```
uint64_t fn(uint64_t a , uint64_t b , uint64_t c ) {
    return ((((a ^ ((2UL & b) * (2UL | b) + (2UL & ~ b)
            * (~ 2UL & b))) + ((a & ((2UL & b) * (2UL | b)
            + (2UL & ~ b) * (~ 2UL & b))) + (a & ((2UL & b)
            * (2UL | b) + (2UL & ~ b) * (~ 2UL & b)))))
            - ~ ((3UL & c) * (3UL | c) + (3UL & ~ c)
            * (~ 3UL & c))) - 1UL);
}
```

Listing 9: MBA expression produced by EncodeArithmetic.

background section focuses on the models and advancements that lead up to the current state of the art of natural language processing (NLP). While classical feed-forward neural networks and a large number of applications for modern deep neural networks derive outputs from one or multiple independent inputs, this has limited applicability to language processing. When dealing with languages, text is a sequence of sentences, and sentences are a sequence of words. Depending on the language, the ordering of words within a sentence can be important, and the ordering of sentences in a text is needed to convey meaning. These traits make natural language processing a driver for more sequence-oriented concepts in machine learning.

One approach to working with sequences is to focus on one element, and calculate a value based on the surrounding elements, similar to how downsampling works in convolutional neural networks. [24] In the context of language processing, words are distributed across a text in a specific manner, and certain words are more likely to appear in certain parts of a sentence or surrounded by other specific words, making the distribution of words across contexts one possible filter mechanism to work with sequences. This idea has been brought up in the 1950s under the name of distributional structure [25], contemplating whether the distribution of a word is linked with its semantics. Over the years, this has been refined and became known as word embeddings, where words are "embedded" in all their contexts and the aforementioned distribution represented as a vector. The release of WORD2VEC [26] was a milestone that caused a plethora of research to be published using embeddings. This is due to the impressive results of word2vec, showcasing what seems to be a semantic understanding between words, allowing simple semantic concepts to be represented by arithmetic operations on vectors. [27] The success of WORD2VEC caused the embedding procedure to be expanded onto other fields and other objects, and embeddings are now a fixed concept in modern models such as BERT. For the intents and purposes of this thesis, another aspect of embeddings is important: Since embeddings are vectors, the cosine similarity shows how similar the embeddings are in vector space, making it possible to implement a search algorithm where the cosine similarity of the vector is calculated and the most similar vector is chosen. Due to cosine similarity being an efficient operation, even a brute-force k-nearest neighbour search can perform reasonably well.

The word2vec embedding vectors can be used as representation that is further processed in another model, e.g. when using IR models like Conv-KNRM [24], or models can integrate the embedding training and inference directly. While embeddings already map simple semantic concepts, they still only concern single words and their contexts, but not actual sequences. In order to process sequences, a neural network model needs to incorporate some sort of memory, keeping state from, e.g. the previous word or words in a sentence, and letting that influence the training pass for the current word. To model such processes, recurrent neural networks (RNNs) came up relatively early in the history of neural networks. Generally, when compared to feed-forward neural networks, recurrent neural networks have weights connecting outputs of neurons back to their input or to the input of neurons in a layer before them, allowing the modelling of historical state. In theory, RNNs can be built with an arbitrary amount of layers and feedback loops, in practice however, this causes the training error in Backpropagation Through Time to either go towards zero in a way that prevents learning progress, or towards infinity. [28] To prevent this from happening, the Long Short-Term Memory (LSTM) [28] architecture was introduced. While the original model has been updated over time, the main idea is to induce a special structure onto the repeating elements of an RNN, giving more fine grained control to stop the exponential effects on error propagation. This is achieved by constructing an LSTM in such a way that each cell guarantees a constant error in itself.

Being able to process sequences, as compared to static inputs, opens up the possibility of generating an output sequence from an input sequence, i.e. a sequence-to-sequence model, or seq2seq model. A seq2seq model consists of an encoder, converting an input sequence into an internal representation, and a decoder, using that internal representation and transforming it into a sequence in the domain of the output vocabulary. Therefore, seq2seq models are the preferred design for neural machine translations, and language processing revolves around improving these architectures. When working with seq2seq models, these will have to store hidden state per element in the input sequence, and this combined hidden state is then passed along to the decoder. It has been shown that this context vector is prohibiting RNN and LSTM seq2seq models from performing on large sequences or long sentences in the specific context of neural machine translation. [29] To work around this limitation, the concept of attention [29, 30] has been introduced: When analyzing a sentence, not all words carry the same weight of conveying meaning and certain parts of a sentence might need more care/state to analyze than others. To take this into account, the hidden states for a sequence can be given scores, depending on their importance in the sequence. When aggregating the hidden states, the states with higher scores are weighted higher, paying more *attention* to these inputs.

Due to the inherent sequential nature of RNNs and LSTM models, training cannot make effective use of modern architectures heavily relying on parallel computing and GPU processing, culminating in the introduction of the Transformer architectures. [31] A transformer is a seq2seq model that has been designed to work with a fixed maximum input length in order to optimize the training and inference on modern GPU hardware. It consists of a stack of encoders and decoders, which consist of a self-attention layer

and a feed-forward network, with the decoder containing an additional attention layer. Self-attention works in a similar manner to the previously described attention through hidden states and uses efficient matrix operations and the softmax function to score the influence other words have on a certain word. The transformer additionally introduces the concept of multiple attention heads: Instead of shifting the focus of attention to one position, multiple attention heads allow the attention training of multiple positions at the same time.

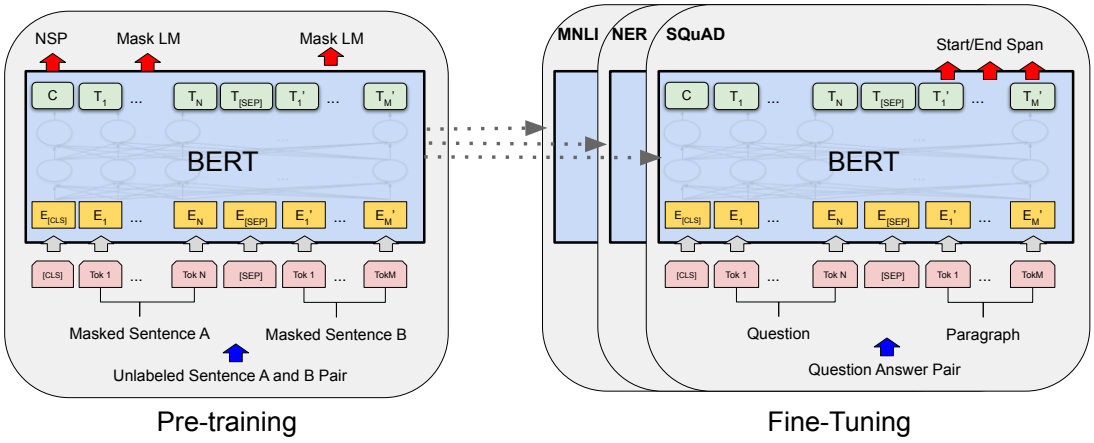

Figure 2.5: Architecture of the BERT network. [23]

Eventually, Google AI language researchers introduced BERT [23], which removes the decoder half from the transformer and instead just stacks several encoder layers, breaking with the seq2seq design of a transformer. Instead, BERT splits up its training process into a pre-training and a fine-tuning session, as shown in Figure 2.5. On the right side of the diagram, different language processing benchmarks are depicted, e.g. MNLI, which are usually supervised learning tasks on a labelled dataset. However, on the pre-training side BERT makes use of masked language modelling (LM), which is a semi-supervised task: An sequence is put into the model, but before processing, BERT masks out random tokens in the input sequence. The same input sequence is then used as a label on the output side, but without the masked tokens. This effectively teaches the model to predict the correct token at the masked location, making it learn the structure of the input corpus. By splitting its training process into two stages, the BERT architecture is one of the prime examples for transfer learning. Through training the model on the input corpus without any manual labels, the model can gain a basic understanding of the input language. This basic understanding is then transferred to a specific task, and refined by the fine-tuning process. To this end, a new task-specific neural network is placed on top of BERT after the pre-training is done. This thesis makes use of BERT-like architectures and applies it to assembly instructions instead of natural language.

CHAPTER 3

Related Work

This section gives an overview of the current state of the art regarding function clone identification and deobfuscation. As function clone identification is not always wanted or possible, most research is done in the broader field of binary similarity, which is not necessarily limited to the scope of functions.

3.1 Classical Approaches to Binary Similarity

In the context of this thesis, all approaches that do not rely on or use machine learning techniques to estimate similarity between binaries are referred to as classical approaches. These approaches do not model binaries as a natural language processing or information retrieval problem, but rather rely on meticulous feature engineering. The general idea is to tackle the problem from the view of an experienced reverse engineer: When analyzing a binary file they will be able to spot similar code by recognizing patterns gained through experience. Due to the specific structure of the control flow and conditional branches, it becomes easy to discern e.g. a function acting as `strlen`, `memcpy` or even `printf`. In addition to this, magic constants, such as certain scalar values in address calculations or specific offsets, can identify a function on first glance.

A prime example for the classical approach is BinDiff, which has been publicly released by Google in 2016. [32] It works on the function and basic block level, trying to match similar structures in two binaries to compare them and find how these binaries evolved. To perform the matching, BinDiff provides a large list of heuristics for functions and basic blocks. [33] These include simple attributes, such as loop counting, string references, sequence of calls, and more advanced techniques like fuzzy hashing and heuristics for call graphs in functions, and control-flow graphs for basic blocks. BinDiff's limitations lie in the nature of these heuristics; most of them rely on counting of different attributes and are designed to execute efficiently, making it fast to use on real-world applications, but with concerns regarding the quality of the function matching.

Different forms of fuzzy hashing are in general widely used for traditional binary similarity approaches, the most common form being locality-sensitive hashing (LSH). [34, 35] As the similarity problem can be represented as finding the nearest neighbour in a similarity vector space, nearest neighbour appears to be a viable solution. However, nearest neighbour algorithms perform poorly when facing high-dimensional data, and LSH tries to solve this issue by providing a way to approximate nearest neighbour search through hashing. According to a survey on binary similarity [13], a large number of approaches uses LSH as some part of their pipeline, such as Kam1n0 [36], BinHash [37], Multi-MH [38], BinSequence [39], CACompare [40] and others. [41, 42, 43, 44] Kam1n0 also introduces a custom, new form of locality-sensitive hashing. One reason for the popularity of LSH is that it is rarely used on its own, but as an additional step in a number of different heuristics to achieve better results, like e.g. BinDiff.

One of these additional heuristics can be n-gram analysis of instruction or byte sequences. For this heuristic, sequences, be it byte, instruction or other tokens found in binaries, of length n are generated and then compared with sequences in the target binary. The first use of n-grams on instruction opcodes was described by Myles and Collberg [45] as a way of generating software birthmarks in order to identify code theft. As n-grams are simplistic on their own, there have not been significant changes to the core idea, however, n-grams are still used in other approaches as one part of a larger pipeline. An example for this is Rendezvous [46], which performs a number of different tasks, such as n-gram analysis, n-perm analysis [47] and CFG subgraph [48] extraction and comparison. In comparison to n-grams, n-perms do not just capture one sequence, but their permutations, as it might be legal to reorder some instructions without breaking semantics of the original program. However, results of experiments conducted by the authors of Rendezvous suggest that this might not be wanted, as n-grams still outperform n-perms on all measurements. Furthermore, n-gram on its own appears to perform better than k-graph subgraph, but the overall best precision/recall performance is achieved when combining all techniques, with the exception of n-perms.

Structural comparisons like these make up a large portion of static binary similarity analyzers, usually including some form of graph matching, like BinDiff. One of the more recent examples relying on graph based similarity comparison is discovRE [49]. This framework performs the function comparison based on a number of manually selected features, extracting both numeric and structural features. The numeric features are used to pre-filter potential candidate matches to avoid performing prohibitively many structural comparisons of CFGs. The authors performed a robustness evaluation on the numeric features they initially envisioned, and chose to ignore some of the original features, e.g. the number of strongly connected components in a CFG for estimation of loop counts; the remaining features include the number of function calls, counts for different instruction groups, basic blocks, basic block edges and incoming calls. A k-Nearest Neighbor search is performed using these counts, and the selected candidates are then compared by the structure of their CFG. The structural distance comparison is implemented by using the maximum common subgraph isomorphism problem, using the McGregor al-

gorithm. [50] In the evaluation, the authors present the capabilities of DISCOVRE on cross-architecture bug search, comparing and slightly improving upon MULTI-K-MH [38]; however, DISCOVRE is faster than MULTI-K-MH by several magnitudes.

Two interesting approaches to structural similarity by the same authors are TRACY [51] and GITZ [52]. TRACY splits a function into *k-tracelets*, recording partial traces up to length k, with the intention of capturing the execution trace of one basic block. The main goal of tracelets is to record the transition from one basic block to another, capturing the first few instructions of the next basic block, to avoid dealing with possibly imprecise jump addresses. To match the recorded tracelets, TRACY defines a number of edit operations, which are applied to convert one tracelet into a similar one. The similarity is then counted as the edit distance between the two tracelets, i.e. the number of edit operations performed. To further improve on results, TRACY also performs a custom rewrite algorithm to counter the effects of compiler optimizations; this allows the authors to outperform n-gram and k-graph CFG matching. The second tool, GITZ, picks up ideas from the rewrite algorithm in TRACY, aiming to perform reoptimization on the code extracted from functions. GITZ starts by retrieving the CFG of a function and splitting up basic blocks to produce strands [53], sequences of instructions within basic blocks that are data-flow dependent. The strands are then lifted to VEX-IR [54] and subsequently to LLVM-IR; according to the authors, the step from VEX-IR to LLVM-IR is added to provide better support for different architectures. The LLVM-IR is normalized and then optimized using LLVM [55], with the final reoptimized strands being checked for similarity using hashing.

The general consensus among binary similarity researchers is the inability of structural comparison common in static analysis tools, making static approaches brittle to code obfuscations with the intention to specifically destroy these structures. In addition to static approaches, a plethora of dynamic techniques exists to complement them, with some projects, e.g. MULTI-MH [38], making use of both, static and dynamic features. MULTI-MH first lifts binary code into VEX-IR, to abstract the underlying architecture away and allow for cross-architecture bug search, and then simplifies the generated expressions using a theorem prover. To extract the semantics of the lifted expressions, the IR code is emulated and fed with inputs according to a sampling strategy, a technique that is also common in blackbox deobfuscation, as described in section 3.3. As the analysis presented in MULTI-MH works on the basic block level, the sampling produces a number of input/output vectors which need to be compared in order to establish similarity. Because this comparison is infeasibly slow in practice, due to the large number of I/O pairs, the authors repurpose LSH, which has been widely used in various static analysis approaches, as shown before.

BINGO [56] is another dynamic analysis approach, which combines several of the presented ideas, such as the tracelets from TRACY and the I/O pair sampling strategy to model semantic similarity, and improves upon them. The main contribution of BINGO lies in the extension of the tracelet concept: By selectively inlining function calls, a trace is not only able to capture a control-flow transition within the CFG, but also the

function call graph. This concept would later be picked up by asm2vec [10], as described in section 3.2. The selective inlining process works by inlining library calls and selectively choosing user-defined functions for inlining by measuring their function coupling score. The authors remark that by inlining functions with a low function coupling score, they are more likely to inline utility functions that don't rely on other user-defined functionality.

While BinGo, Tracy and Multi-MH rely on sampling of I/O pairs, Blex [57] proposes a different primitive for determining similarity via dynamic analysis. This primitive, called blanket execution, takes an environment and a function, executing the function and recording its changes to the environment. In order to cover all instructions of a function, Blex simply starts execution at instructions that haven't been executed in the next iteration. In order to calculate similarity, Blex records some features observed during execution into feature vectors. These features include memory access to locations on the heap, memory access to locations on the stack, calls to dynamically imported functions, system calls and return values. In their evaluation, the authors show that stack read/writes and writes to heap memory provide the highest accuracy during function similarity comparison.

Similar to Blex, Tinbergen [58] builds upon the idea of monitoring the environment while executing a function, and introduces IOVecs, storing input program state and the state expected after execution. The idea behind IOVecs are, if a function accepts certain inputs, a similar function is likely to accept the same inputs, i.e. a similar function should produce the same/similar output on the same/similar input. To find IOVecs in the first place, Tinbergen fuzzes the functions and if the function terminates, the generated inputs and collected outputs are combined into IOVecs. A particular problem with IOVecs is the detection of pointer arguments as input parameters for functions; to mitigate this, the authors apply taint tracing to determine whether a crash during fuzzing was caused due to an invalid pointer argument.

In general, dynamic analysis approaches suffer from long runtimes when compared to static analysis, but try to improve robustness and accuracy. A prime example for this is BinHunt [59], which uses taint tracing to compare binaries, and can take hours to match basic blocks. A combination of techniques, as used by Expose [60], where functions are prefiltered using static techniques and then dynamic analysis is used to compare the rest, has the potential to circumvent the high cost of dynamic analysis, providing a tradeoff between accuracy and performance.

3.2 Approaches Based on Machine Learning

The rise of computation power and the development of modern machine learning techniques opened up new possibilities in the field of information retrieval and natural language processing. Function clone identification no longer has to rely on static signature heuristics or simple stochastic models, but can use these machine learning techniques to model functions as complex feature sets. Especially approaches based on embeddings,

as described in section 2.3, have been shown to work well on the problem of function clone identification. One of the earlier approaches to use a technique akin to graph embeddings based on machine learning in contrast to manual feature extraction is Genius [61]. It uses attributed control-flow graphs (ACFGs) as raw features extracted from the binaries. In addition to the normal CFG, the ACFG in Genius' case contains a number of different indicators as attributes, split into statistical and structural features. The statistical features include the number of certain instruction types, string constants and numeric constants, while the structural features count the number of child nodes of a basic block and calculate the centrality of the basic block in the CFG. In the next step, called codebook generation by the authors, Genius calculates what is practically a graph embedding. It performs a bipartite matching of ACFGs, assigning a cost value, which relies on the attributes in the CFG, to each matched edge and then performs a learning process using a genetic algorithm, to optimize the attribute weights. Afterwards, an unsupervised learning algorithm is used to cluster ACFGs with the previously learned similarity metric, creating a set of n clusters. Raw ACFGs are then encoded into a vector of dimension n, by comparing the distance to the nearest centroid neighbour in the codebook. The actual similarity lookup is performed by using LSH on the generated feature vectors, as done by a majority of classical approaches. The evaluation compares Genius to discovRE and Multi-k-MH, outperforming them in terms of result quality and query time.

Gemini [15] is approaching the binary code similarity problem with embeddings of the ACFG, in a similar manner to Genius. Compared to Genius however, Gemini uses neural networks to calculate the embeddings of an ACFG. The basic building blocks for this approach have been laid by structure2vec [62], a network designed to create embeddings for structured data. However, the authors of Gemini note that structure2vec has been designed for classification tasks and that generating the graph embeddings is in itself not a classification task. To overcome this limitation, Gemini uses a Siamese architecture: It contains two identical structure2vec networks and combines them to produce a similarity score. Using a pair of two functions and a score for describing their similarity, the model can then be trained to match this similarity score. This training process requires a large amount of function pairs and their similarity, Gemini uses source code information to create these function pairs with identical/equivalent functions.

While the neural network used by Gemini comes with performance benefits compared to Genius, it also allows for specializing the trained model further for a specific task. This task-specific re-training, as described in their paper, allows human feedback on classification to be integrated into the model. This can be seen as a simple version of transfer learning, where a model is pre-trained on a ground truth dataset, and then fine-tuned for a specific task. The evaluation results of Gemini show a speedup and accuracy increase over Genius, on their fabricated and real-world examples, showing the usefulness of embeddings generated through machine learning. Gemini's approach of embedding the control-flow graph is however not without pitfalls, as the authors describe that when only relying on the ACFG for embeddings, changes that don't affect

the ACFG also do not affect the similarity. In one specific evaluation case, Gemini was not able to determine whether a software version was vulnerable or not, due to the patch only affecting a small number of instructions and no control-flow transitions.

Increasing the complexity from Gemini in terms of model design, *a*Diff [63] tries to remove the need for feature modelling altogether, by introducing a convolutional neural network, modelled after image processing, which works on raw bytes instead of the CFG or disassembled instructions. Not having to model features is the theoretical ideal solution, as there would be no need for third-party tooling like disassemblers, IR lifters or CFG reconstruction tools, also significantly speeding up the matching process. *a*Diff outperforms BinDiff [32] and BinGo [56], interestingly also showing better performance than BinGo on cross-architecture, despite being trained on raw bytes of one architecture. The results seem questionable, as other recent work is cited multiple times in the paper, but comparisons only use older approaches like BinGo or BinDiff, instead of comparing with, e.g. Gemini.

An improvement to Gemini is shown by VulSeeker [12]. Instead of relying on the ACFG alone, VulSeeker also tries to incorporate the data-flow graph into its analysis. To this end, a construct called labeled semantic flow graph is generated by taking the CFG and then adding data-flow edges, if it can be derived from LLVM IR that two basic blocks are data-flow dependent. On the basic block level, VulSeeker uses a similar set of statistical features as Gemini, combining the semantic flow graph and these features using a neural network modeled after structure2vec. While VulSeeker on its own is already able to outperform Gemini, the authors released an updated version called VulSeeker-Pro [64], which builds on VulSeeker and improves the matching of the top candidate functions by performing emulation.

Asm2vec [10] uses an approach closer to the original word2vec: Instead of calculating embeddings based on the control-flow graph, the embeddings are calculated directly through the instructions. This marks a paradigm-shift towards treating assembly language, or raw bytes of machine code instructions as similar to natural language. In terms of architecture, the authors do not use a specific pre-existing network architecture, but use PV-DM/doc2vec [65] as base architecture and construct their own network to fit the assembly domain. As word2vec only marks the embeddings of the words itself, it is limited when applied to function clone identification on assembly-level. Embeddings of the assembly instructions themselves are not helpful on their own, but when aggregated to create an embedding of the function as a whole, the embedding vectors of two functions can then be used to calculate the similarity via cosine vector similarity. The extension of word2vec necessary for this to work is covered in the PV-DM approach; in terms of natural language, the assembly instructions would be words and a function would be a paragraph or document.

As the authors did not just copy the PV-DM approach, some intricacies revolving around specifics of assembly language are introduced. Asm2vec does not just linearly disassemble a function from start to end and feed the instructions to the model for training, but performs random walks on the control-flow graph of the function. This means, there is

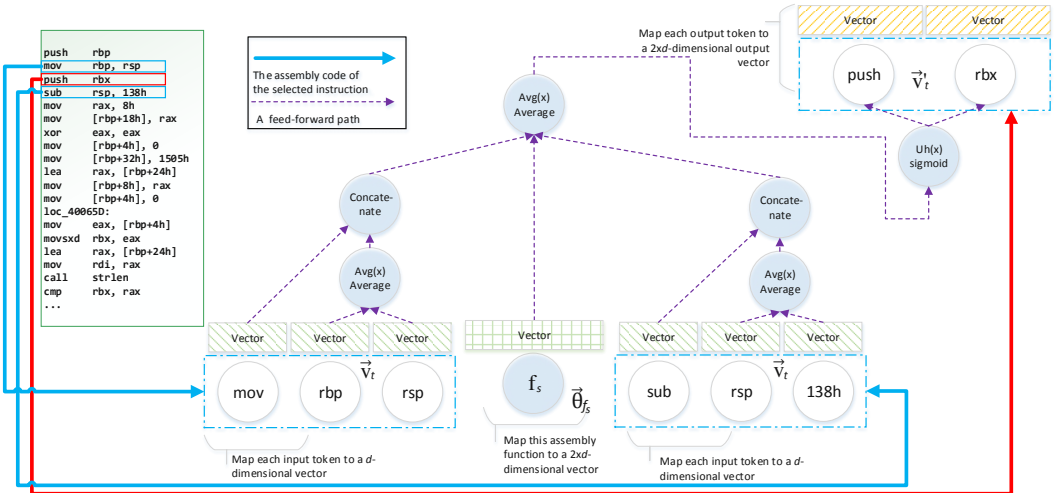

Figure 3.1: Asm2Vec extension of the PV-DM model, as seen in [10].

not a single list of instructions, but a set of multiple instruction sequences of the same function. These sequences are generated by taking the branch targets of control-flow transitions into account: Random walks are being performed on the control-flow graph and called functions are selectively inlined into the instruction sequences of the caller, as suggested by BinGo [56]. To reduce overhead, function calls are not recursively expanded, but only the first level of function calls is considered.

Another assembly language specific detail is the handling of operands. As assembly language instructions are small parameterizable units, they lexically consist of multiple tokens, but semantically act like simple functions. This can be seen in the graphical description of the Asm2Vec model, shown in Figure 3.1; instead of treating it like natural language, the instruction operands are handled separately from the instruction opcode. All parts of an assembly instruction are tokenized and turned into vectors with two parts, the embedding vector for the opcode, and the average of the embedding vectors for the operands. These two parts are concatenated to form the embedding vector for the instruction and averaged with the surrounding instructions. Finally, these vectors are updated with the gradients from the backpropagation training process. The authors do not specify how numerical operands, such as constant addresses, are treated during the tokenization, making these operands normal lexical tokens that can cause potentially large vocabularies. In their evaluation, they also take binaries obfuscated by O-LLVM [18] into account, making it one of the first machine learning based approaches to function clone identification that also consider obfuscated inputs. An approach similar to Asm2Vec is instruction2vec [66], which was published at around the same time; however the description of the approach is opaque and due to the different data set and evaluation goal, the performance cannot be directly compared to Asm2Vec.

Released around the same time as Asm2Vec, Reymond et al. [67] try to tackle the embedding of instructions across architectures. Instead of building on PV-DM, their approach uses word2vec directly, making it limited to predicting instruction similarity. In order to learn embeddings for multiple architectures, they adapt the word2vec continuous bag of words training approach to predict the following instruction in a different architecture instead of its own architecture. While the authors provide a ROC curve for basic block matching with simple summation of instruction embeddings that seems promising, it ultimately can't replace the overall better designed approach of Asm2Vec; however, it poses a starting point for making Asm2Vec aware of multiple architectures, as the presented multi-architecture approach could be introduced into Asm2Vec with little effort.

InnerEye [44] works on addressing the cross-architecture problems of previous approaches, as well as going a step lower to the basic block level instead of working with function pairs as most other approaches do. Compared to Asm2Vec, instructions are therefore treated as words and basic blocks as sentences/paragraphs. Instead of combining the embedding training into one model, like Asm2Vec does, instruction embeddings are generated first for each architecture. However, when generating the basic block embeddings, every architecture to be trained has its own LSTM model. Training these LSTM models happens via a Siamese architecture, making sure that the models for the different architectures are trained concurrently.

Similarly to InnerEye, Safe [11] takes a two-step approach to function embeddings. Compared to Asm2Vec, where PV-DM was used to generate the function embedding in one pass, Safe first generates embeddings for the instructions themselves and then subsequently uses the instruction embeddings of a function to train a bidirectional recurrent neural network utilizing self-attention, building upon previous work from natural language processing. [68] The respective models of these steps are trained separately, showing a slow trend towards transfer learning, which has been hinted at with Gemini before. Comparing the approach to InnerEye, Safe only requires one RNN with self-attention, instead of separate LSTM models for each architecture.

For generating the instruction embeddings, called instruction2vec [1], a standard word2vec approach is used, and the model is trained using the skip-gram technique. This means, the embeddings are trained by predicting the neighbouring instructions of the current instruction. Compared to Asm2Vec, constant values in disassembly instructions are not just treated as additional lexical tokens, but if it concerns a memory offset or an immediate value, the constant is being replaced by an abstract token, e.g. `IMM` for intermediate values or `MEM` for addresses. The authors believe this to improve the quality of the embeddings, but have not provided evidence to back this claim.

The function embedding part of Safe takes the sequence of instruction embedding vectors of a function as input. The authors do not state how this sequence of instructions is generated, but they mention not needing to generate a CFG as a speedup over Gemini;

[1] This model is not related to the instruction2vec [66] model in the paper of the same name.

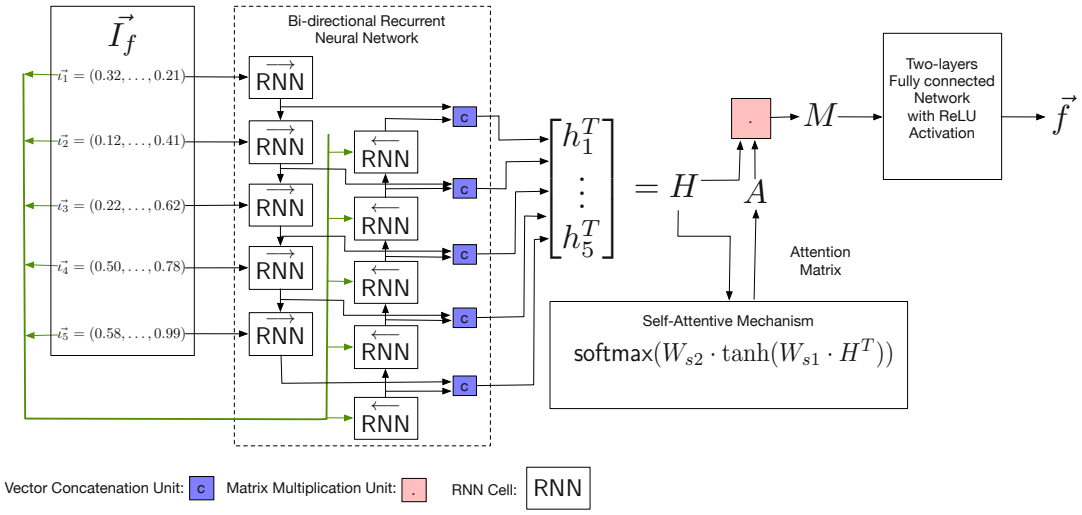

Figure 3.2: Architecture of the SAFE function embedding RNN. [11]

this likely means that the instruction sequence is generated through linear disassembly, making it not robust in the presence of obfuscations, such as data insertion between instructions. This is in stark contrast to the meticulously designed random CFG walk and call target inlining strategy implemented by ASM2VEC. Ater feeding the instruction embedding sequence to the network, it first passes the bidirectional RNN, shown in Figure 3.2. The hidden states of both directions are then concatenated and passed through the self-attention mechanism, producing weighting in the form of an attention matrix that is multiplied with the hidden state vector. This result is then finally passed through a 2-layer feed-forward network to generate the final function embedding. In order to train this model, SAFE relies on the same mechanism as GEMINI and uses a Siamese architecture to learn the similarity of function pairs.

Despite not reconstructing the CFG to perform random walks and call target inlining, SAFE shows good performance in the authors' benchmarks. The results have been evaluated on one dataset on amd64 with multiple compilers, and a second dataset showing the capabilities of cross-architecture function similarity. While acknowledging the existence of ASM2VEC in the introduction, the authors only compared SAFE to GEMINI in their benchmarks, where SAFE outperforms GEMINI in accuracy and evaluation performance. The support for multiple architectures in SAFE is mentioned as an improvement over ASM2VEC, however SAFE does not handle architectures in a special way and there is no obvious reason why ASM2VEC should not be able to be trained on multiple architectures, besides it not being shown in the paper evaluation. As an additional evaluation task, SAFE performs a semantic classification of the function embedding vectors, categorizing them in encryption, math, sorting and string manipulation functions. Visualizing the categories on the t-SNE [69] visualization of the embedding vector space, highlights that semantically similar functions form clusters.

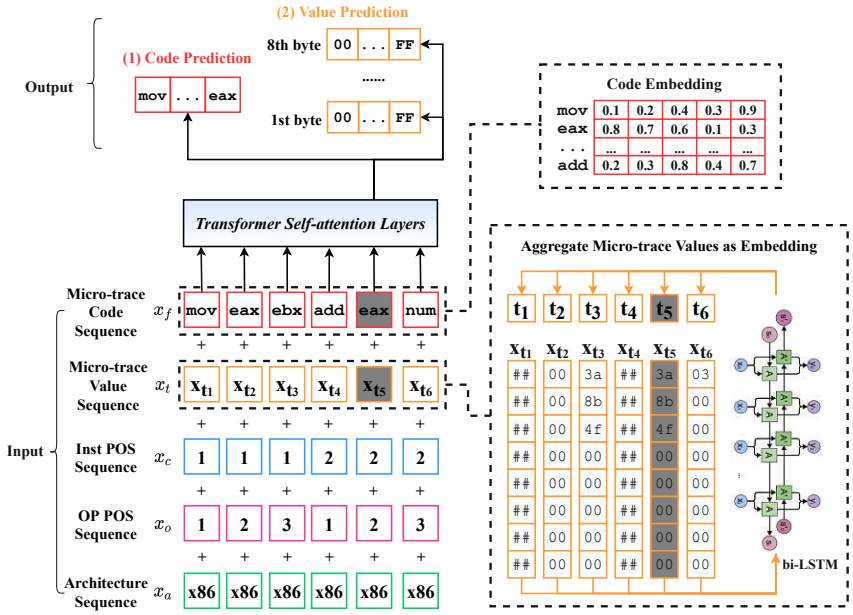

Figure 3.3: Masked LM of the Trex model. [1]

As of the time of writing, the most current addition to machine learning approaches for function clone identification is Trex [1], adding and refining several new concepts. Trex model is based upon RoBERTa [70], providing the basics for transfer learning that has been hinted at in Gemini and Safe by different means. Transfer learning allows to shift more work into the pre-training of the model, as ideally the pre-training only happens once and the model can be specialized for a specific task afterwards. With the specialized task being the calculation of function embeddings, done through a simple feed-forward network, the focus in this paper is on the pre-training approach. Here, the authors decided to capture the execution semantics of instructions, as compared to working with static instruction sequences. Observing the instruction execution through microtraces, a form of microexecution [71] and pre-training with the dynamic values pulled from registers and memory related to an instruction, is supposed to teach the model the semantics of these instructions.

The RoBERTa model is trained with information from the microtraces by using a masked language model objective. In this case, if a dynamic value retrieved from a microtrace happens to be masked, the model is taught to predict this value. During the evaluation phase of the model, only static information is used as input and it will internally use the learned prediction on the static instruction sequence. Due to the existence of these dynamic values, the Trex authors had to come up with an encoding scheme for constants and concrete values, as using an abstract token for these values,

like SAFE does, would remove the additional information gained by using microtraces. To keep this information without using dynamic values as tokens, a bidirectional LSTM model is used for generating what is effectively a simple form of embeddings. Because the pre-training approach is meant to capture the overall semantics of the instructions, this part of the training can be done on a completely different dataset than the one needed for the function similarity finetuning. Ideally, the pre-training set would cover a wide range of different instructions and on different architectures.

In contrast to the other approaches, the TREX authors also include a thorough evaluation and comparison to the most recent approaches listed here. TREX is able to show similar performance to ASM2VEC when applying it to one step of compiler optimizations, and is able to outperform ASM2VEC when doing cross-optimization search across the respective lowest and highest optimization levels. Performance across obfuscations is slightly improved when compared to ASM2VEC, and performance in general is better than reported by SAFE. TREX also contains an ablation study, showing how the pre-training impacts the performance of the model. When pre-trained with microtraces, there is close to no noticeable difference when trained with 100%, 66% or 33% percent of the original microtrace dataset. When not pre-trained, the AUC scores drop on average 15.7% and when pre-trained with static data instead of microtraces, the AUC scores drop by 7%, putting the AUC scores on average below the reported values of SAFE.

3.3 Deobfuscation

The usage of machine learning for deobfuscation is an interesting case, as the goal for code deobfuscation is in general to retrieve the original code or something close to it. This implies that the deobfuscated code needs to have the same exact semantics as the original code, and machine learning techniques usually do not provide exact results. However, there are a multitude of topics in deobfuscation where approximate outcomes are applicable, as deobfuscation can also involve a human reverse engineer and approximate information can be valuable for human thought processes. This applies to names in particular; in the worst case, an executable binary has no function labels, no symbol information or no imported library functions. If a heuristic provided a list of imported functions that is only 80% correct, this can already improve the time needed for a human analyst to understand the overall structure of obfuscated code, turning e.g. detection of obfuscated function clone into a problem solvable by applying machine learning algorithms.

A prime example for this is DEBIN [8]. Instead of restricting itself to function clone identification, DEBIN tries to recreate debug information for a binary, i.e. it tries to assign the correct function names, as well as variable names and type information. In order to facilitate this goal, DEBIN uses several different models for different elements to be recovered: To determine whether a register or memory offset is a variable, Extremely randomized Trees [72] are used. Afterwards, conditional Conditional Random Fields are used to assign the most likely names and types to these variables and functions. Instead

of working directly on bytes or disassembly, Debin extracts the features needed for training the statistical models by lifting the binary into BAP-IR first, the intermediate language of the binary analysis platform. [73] This allows Debin to function independent of architecture and compilers and make use of pre-existing analysis modes in BAP, such as detection of function boundaries.

Deguard [74] and Macneto [75] are approaches that work on deobfuscating Android applications. When compiling and packaging Android applications, a lot of information is retained in the final application, including names and identifiers, in contrast to binary applications. While it is still possible to significantly obfuscate Java bytecode using control-flow transformations [76], Deguard focuses on reversing layout obfuscations done by ProGuard [77], shortening and renaming identifiers, such as class, variable, or function names, to reduce size of the produced bytecode file and obfuscate the meaning behind these variables. Deguard tries to work around these obfuscations, by using a similar approach to Debin, where a dependency graph of related features in an Android application is constructed, feature functions are assigned and combined into a Conditional Random Fields; Deguard appears to be a precursor to what Debin would eventually become on arbitrary binaries.

Macneto [75] instead uses an approach more in line with the recent advances in function clone identification. [1, 10, 11] While focusing on the obfuscations provided by ProGuard, the authors also assume control-flow and data transformations to be present in the code, but exclude dynamic obfuscation, such as overuse of reflection in Java. As the first step in its pipeline, Macneto performs instruction distribution analysis, to build a typical index as used for information retrieval. In the second step, the authors apply topic modelling [78] to machine code instructions, where functions are treated as documents and instructions as terms, in order to uncover hidden topics in the disassembled instructions of a function. The authors of Macneto arbitrarily chose 35 machine topics, and used Latent Dirichlet Allocation [78] to create a topic vector for each function, based on their instructions, mimicking instruction embeddings. Finally, the instruction distribution and topic vector is combined and used to train a recurrent neural network, predicting the topic vector of an unknown function, and cosine similarity is used to find the original function.

While Debin, Deguard and Macneto work by reconstructing features necessary for a human analyst, approaching deobfuscation of code constructs directly using machine learning is possible. Tofighi-Shirazi et al. [79] present a technique for analyzing opaque predicates, applying machine learning to this specific type of obfuscation. Through automated binary analysis, using symbolic execution, they build a labeled dataset of raw binary data for supervised learning. The raw binary data is lifted to an intermediate representation using Miasm [80] and then turned into a normalized version of a static single assignment (SSA) form. Based on this data, the authors seek machine learning solutions for two questions to be answered: One model for determining whether an expression is an opaque predicate and one for determining the value an opaque predicate evaluates to. As not every predicate in a binary should be an opaque predicate, an

obfuscator tries to make an opaque predicate look like a legitimate predicate; the opaque predicate always evaluates to one certain value, and identifying a normal predicate as opaque might lead to the wrong result when trying to calculate its value. Therefore the first model is a safeguard to prevent the second model from trying to determine the fixed opaque value. The authors tested several different simple machine learning approaches, e.g. k-nearest neighbor or support vector machines, and found out that decision trees produce the best results on their dataset.

Syntia [81] takes the application of machine learning for deobfuscation in a different direction and does not directly use the output of a model for deobfuscation, as Tofighi-Shirazi et al. did, but uses machine learning as a way to guide search space exploration. In order to find the original code from code that has been obfuscated as a virtual machine, the authors of Syntia use program synthesis, a technique where an optimal piece of code is generated according to some formal specification of semantics. They mainly build on work by Gulwani et al. [82], which introduced program synthesis based on specification in off-the-shelf SMT solvers for synthesis of loop-free programs, and work by Jha et al. [83], using I/O value pairs as an oracle to guide program synthesis. In order to deobfuscate virtualized code, Syntia first records traces of a program under execution. These traces are then dissected into trace windows, which split the trace into logical units, with the authors mentioning that selection of the window boundary is critical to the analysis; in the case of VM obfuscation, they chose to split the trace on indirect branches, indicating processing of the next opcode. Afterwards, they perform random sampling by finding I/O value pairs for the trace window, and finally perform program synthesis using these value pairs. The main contribution of the authors is the use of Monte-Carlo Tree Search (MCTS) [84] combined with simulated annealing [85] to make program synthesis scalable in this context. This combination allowed the authors to automatically learn the semantics of 98.9% of VM handlers in VMProtect [21] and 94.4% in Themida [20] at the time of the publication.

The most recent addition to deobfuscation of virtualized code using machine learning approaches is Xyntia [86], which explores possible improvements of its namesake Syntia, and reflects on machine learning based blackbox code deobfuscation in general. The authors generalize the approach taken by Syntia and define a blackbox obfuscator as a tool, which can generate arbitrary input, trigger the obfuscated code with this input, observe the output and then generate code approximating the observed semantics. Treating the obfuscated code as a blackbox that is queried rather than analyzed allows to completely bypass obfuscations that rely on syntactic complexity alone in theory; the approach of querying and e.g. comparing it with previously observed semantics for similarity especially encourages stochastic heuristics or heuristics based on machine learning. Instead of performing MCTS like Syntia, Xyntia uses S-metaheuristics [87], with Iterated Local Search (ILS) [88] being the specific heuristic used. The authors mention ILS being especially suitable for unstable search spaces and the ability to restart the search from the previously best found solution. While the MCTS approach used by Syntia may create new subtrees in an AST when synthesizing an expression, the ILS

optimization strategy used by Xyntia only mutates terminal nodes in the AST. Xyntia again uses an I/O oracle as sampling strategy and stops the ILS when the synthesized expression is valid on all I/O samples, or when the search algorithm takes longer than a set amount of time. When performing distance measurements, ILS shows a greater structural distance from the starting expression than MCTS, as MCTS acts like enumeration in this case, while ILS is actively guided by an objective function, allowing for mutations that would not be found by MCTS as they are too deep down in the search space. The reported results of Xyntia across datasets used by QSynth [89] shows better performance than Syntia and other state-of-the-art program synthesizers [90, 91], while also outperforming QSynth when dealing with heavy obfuscation; the authors credit the increased syntactic complexity on the heavy obfuscation dataset for this, as QSynth uses a greybox deobfuscation approach.

Syntia and Xyntia demonstrate the viability of machine learning methods, even when transforming code that requires exact results, and Xyntia shows that uninformed search through a blackbox using learning strategies relying on a guiding function can outperform white- and greybox solutions like QSynth. While a trend towards learning approaches can be identified, there is still more potential for the introduction of modern machine learning techniques, i.e. current approaches rely on stochastic search techniques and do not make use of architectures like deep neural networks. Additionally, with the rise of machine learning techniques comes the possibility of adversarial models targeting these techniques. The authors of Xyntia briefly discuss how blackbox analysis methods can be countered, namely using VM opcode handlers with complex semantics or complex nested conditions, showing the fragility of blackbox analysis in the face of an adversary.

CHAPTER 4

Obfuscated Function Clone Identification

Review of related approaches has shown that binary similarity has profited from machine learning and that usage of machine learning is viable for deobfuscation tasks. Out of recent state-of-the-art function clone detection approaches, only two [1, 10] have dealt with analyzing binary code which had its control flow and data obfuscated. Neither of these existing approaches has tried to apply binary code similarity algorithms to counter obfuscation by code virtualization. In order to improve the state of function clone detection when faced with heavily obfuscated code, this thesis introduces Obfuscated Function Clone Identification, or OFCI for short.

4.1 Assumptions and Threat Model

Before discussing the general architecture, it is necessary to highlight the assumptions and constraints under which OFCI was designed to operate. In order for OFCI to detect function clones, the knowledge of function entry points and boundaries is required upfront, i.e. the function start detection problem is assumed to be solved by existing tools/libraries. Since OFCI uses textual disassembly for comparing functions, as opposed to raw bytes, a working disassembler is required; however, compared to other solutions (e.g. ASM2VEC, OFCI does not need to reconstruct the CFG. In order to keep the amount of training data to a manageable size, OFCI is only evaluated on the x86_64 architecture. TREX [1] and SAFE [11], the previous works OFCI is based upon, have shown good performance of cross-architecture clone search, making it likely that OFCI can generalize across architectures as well, when presented with an adequate amount of training data. Furthermore, when training OFCI from scratch, a consumer GPU with at least 6 GB of VRAM is required to perform the training in a feasible amount of time (cf. chapter 6). Using a trained version of OFCI for function clone identification does not re-

quire a GPU, but using a GPU can speed up the inference process. Only Linux binaries have been used for evaluation/validation, but there is no inherent obstacle preventing the application of Ofci on executables for different operating systems or raw firmware binaries.

The threat model of Ofci assumes the author of the binaries under analysis to be an attacker, who makes use of obfuscation techniques to hide information contained in the binary. It is not relevant, whether the binary executable is benign or malicious. In this threat model, the attacker is assumed to only apply the obfuscations discussed in section 2.2, i.e. only obfuscations that do not split or merge functions, as the goal of Ofci is function clone identification. The attacker is generally not assumed to use packed code, with the exception of code that can be traced or unpacked by existing tools, using Ofci to statically analyse the unpacked code. As the first tool of its kind, Ofci also allows an attacker to use code virtualization as an obfuscation technique. This is limited to the function level, meaning virtualization is applied to individual functions and not the whole program, as implemented by Tigress [17].

4.2 Architecture Overview

The architecture of Ofci consists of several moving parts and pipelines, designed to be integrated into a reverse engineering environment. Every part of these pipelines can be replaced by concept: It does not matter which reverse engineering environments are used, as long as the environment provides a plugin interface for data import and export. The machine learning libraries can be replaced by different implementations in a similar manner, as long as the implementation provides the same model interface and general architecture.

A coarse overview of the Ofci concept is shown in Figure 4.1. The diagram shows the lifecycle of Ofci and its three basic pipelines: Pretraining, fine-tuning and inference within the reverse engineering environment. The training step being split up into pre-training and fine-tuning is the result of the underlying machine learning model architecture (see section 4.4). Trex was the first approach to adopt a recent trend in natural language processing, where a transformer model is pre-trained with a generic task on the input language, initializing the network parameters by learning an "understanding" of the language. This knowledge is then *transferred* to a specific task, in this case sentence similarity, where functions extracted from a binary are interpreted as sentences. While Trex used the pre-training to additionally include data gathered by dynamic analysis in the input language [1], the authors also showed that pre-training without dynamic data still increases the performance of the function similarity task. Ofci makes use of this insight to build a simpler model that does not include the additional number of model parameters required to accomodate dynamic information, removing the dynamic value concept altogether. In the pre-training step of the pipeline, Ofci extracts a large number of functions from a set of different open source tools, using the binaries published

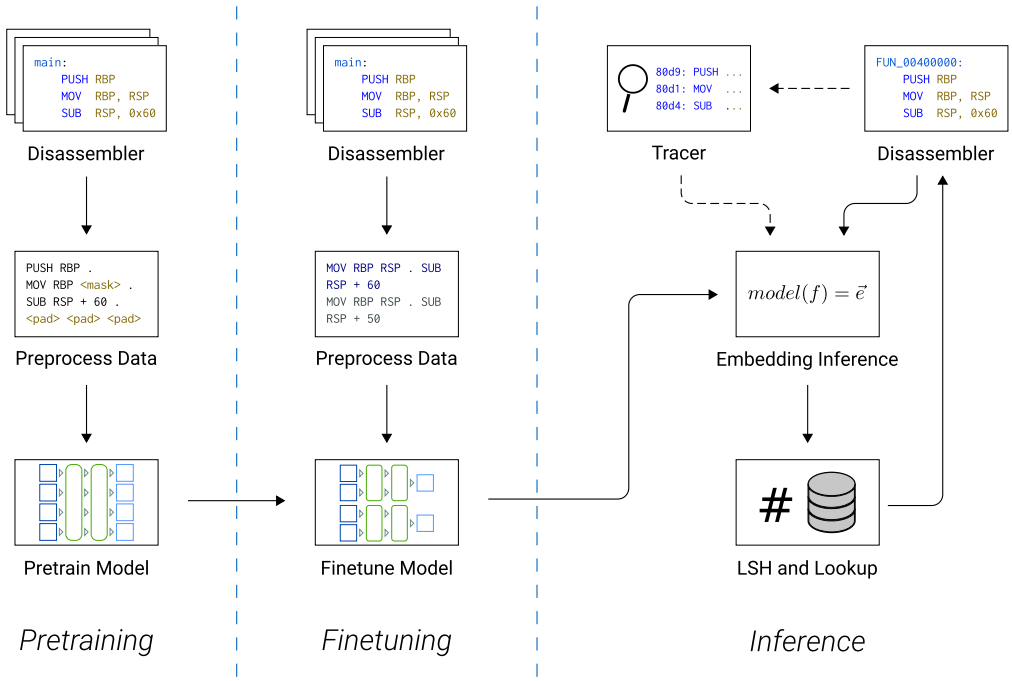

Figure 4.1: Overview of the Ofci architecture and pipeline.

by the authors of Trex along with their code [1]. After disassembling the functions, the generated assembly code is preprocessed and stored in a database. The model is then pre-trained on the complete set of functions using the *Masked Language Modelling* objective: Individual instructions within a function are replaced with a mask token and the objective is to predict the original token at this position. The result of this pre-training is a model which can already produce embeddings for functions, albeit embeddings of poor quality for the similarity task. This step in the pipeline is the most time-consuming, as the pre-training should make use of an adequately large dataset, and should ideally be a one-time effort.

Making use of the pre-trained model, the second step in the pipeline uses the model as starting point and applies a fine-tuning training for the task of function similarity. In this step, a pair of functions is used as inputs for the model and the model outputs the embedding vectors for each of them. The cosine similarity of the model outputs is calculated and the difference to a specified label is used as the training loss. This means that the fine-tuning step is supervised, specifying a pair of function and whether they are similar (1.0) or dissimilar (-1.0), as opposed to the self-supervised pre-training step.

[1] The dataset is provided in the description of `https://github.com/CUMLSec/trex` (Accessed 2021-12-06).

A consequence of this is the need for function pairs in the dataset, through different versions, compiler optimizations, or obfuscations. While the performance of the fine-tuned model should not significantly degrade when used on previously unseen data, this step is less time intensive than the pre-training task and can be repeated in a shorter amount of time. In the final step of the Ofci pipeline, the fine-tuned model is being used to generate embeddings for the functions to be analyzed. First, the function repository, ideally the same functions from which the fine-tuning dataset has been generated, is annotated with an embedding for each function in the database. When trying to find a similar function for an unknown function, the disassembler can pass the assembly to the model, generate the embedding and use locality-sensitive hashing to find a correlating function in the repository. When compared to existing approaches [11, 10, 1], Ofci also allows to extract the assembly for embedding generation from an instruction trace, making it possible to analyze functions obfuscated through virtualization, where different functions could reuse the exact same virtual machine code, but their bytecode behavior only becomes apparent during dynamic analysis.

The different parts of the pipeline come with their own intricacies. Concluding this high-level architecture overview, the following sections elaborate on the theoretical concepts and considerations for each part of the pipeline, while the practical details and software dependencies are covered in chapter 5.

4.3 Feature Modelling

While it is not an explicit pipeline step in itself, the feature extraction happens at every part of the pipeline, providing the necessary input data for the embedding model. When comparing the task of function similarity with sentence similarity in natural language processing, the idea of treating disassembly as text and using it for further processing directly seems to be the obvious choice. However, existing work has shown that there is a broad number of approaches on how to model features based on executable binary code: Some approaches use numeric features of basic blocks in the CFG [49, 61], other approaches work directly on the byte level [16] and others perform random walks on the call graph/CFG and use these instruction sequences for training [10, 92]. This highlights the importance of feature modelling for the various approaches.

Ofci takes inspiration from Trex [1] and Safe [11], using raw disassembly as a starting point for the feature extraction. It does not rely on being able to reconstruct a CFG and only assumes knowledge of function entry points and that functions are layed out linearly in memory. While raw disassembly can be used as text input for natural language processing (NLP) tasks without further modification, numeric constants in the disassembly present a challenge in typical word processing settings. These constants can carry specific meaning, e.g. relative address offsets or special constants used for bitwise operations, which should be preserved to aid identification of a function. The flipside of preserving constants unmodified is increasing the size of the vocabulary used by the tokenizer, which turns the words into numeric identifiers that can be passed to the model

as input. On a 64-bit architecture this would entail the theoretical possibility of having a vocabulary of size 2^{64}, making the vocabulary prohibitively large. SAFE and ASM2VEC solve this issue by limiting the range of the constants: If a value is smaller than a fixed limit, it is interpreted as its own token, otherwise it is replaced with a token corresponding to the type of the constant, e.g. `NUM` for arbitrary numeric constants or `ADDR` when the constant is an address. TREX uses a bi-LSTM network to create embeddings for numeric values instead, as they also work with dynamic values obtained through micro-tracing. These dynamic values and constants are provided within a second input field besides static code, also including other fields like absolute positions, relative positions and architecture. OFCI takes an approach closer to the one used by TREX, but without using multiple input fields and bi-LSTM encoding for numeric values.

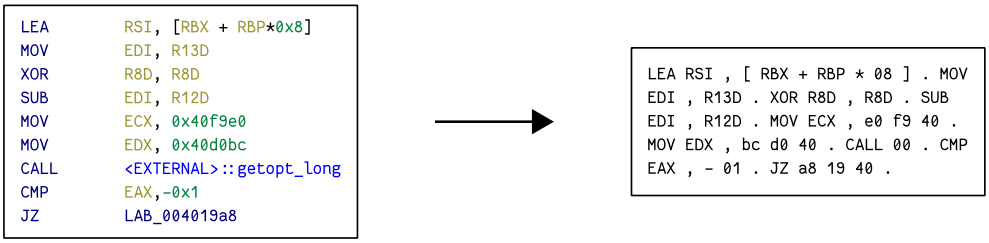

Figure 4.2: Extraction of a normalized function from disassembly.

An example of how OFCI normalizes a disassembled function for tokenization can be seen in Figure 4.2. In general, the information present in the disassembled code is largely preserved: Memory accesses are still delimited by square brackets and calculation operators are kept as separate tokens. All symbols and words are separated by whitespace, instruction operands are separated by comma and the instructions itself are separated with a dot to eliminate line endings. Register names and opcodes are passed through normalization unprocessed, but scalar values and addresses are treated separately. For both, scalars and address offsets, values are first treated as 8-byte integer values in little endian byte order. In order to prevent an arbitrarily large vocabulary, this integer is then added as a new word to the normalized disassembly byte by byte, omitting trailing zeros. This can be seen in the example, e.g. when `0x40f9e0` is turned into `e0 f9 40`. As each byte is added separately, the vocabulary can only increase by the fixed size of 256 entries.

Trailing zeros are removed in order to keep the actual amount of needed tokens for the number small, as they do not convey additional meaning. In a similar manner, whenever a scalar or address offset is negative, the absolute value is taken if possible and the negative sign is prepended to this value. In the example, this happens with -1, which would otherwise be normalized as `ff ff ff ff ff ff ff ff`. In case the negative scalar is part of an address calculation, a preceding addition operator is being

replaced with the subtraction operator. Another special case for the normalization is represented by addresses, which correspond to the entry points of known functions in the binary, or external function stubs. Some previous approaches [10, 92] try to improve function matching performance by expanding the called function and inlining parts of it into the caller. This allows the establishment of interprocedural relationships, but has so far only been used as additional information for training embeddings. It also comes with the pitfall of having to reconstruct the control flow to a certain degree, as the mentioned approaches rely on random walks across the CFG.

Ofci tries to incorporate the basic idea of function inlining, without actually having to reconstruct the CFG or inline the function into the caller. Instead, address references to the entrypoint of a function are recorded and replaced in the normalization phase. When building a database of function embeddings, unique IDs are assigned to the function names. Afterwards, whenever a function is called and the address of the called function can be derived from static disassembly, the address is used to retrieve the name of the function from the symbol table. Instead of keeping and normalizing the function address, the unique name ID of the function is inserted into the normalized disassembly. If no ID for a specific name is stored in the database, the address reference is replaced with 0, which can be seen at the `CALL` instruction in Figure 4.2, as the normalization was generated without a backing database. If the function `getopt_long` was recorded in the database with, e.g. name ID 1, the corresponding instruction would be normalized to `CALL 01`. This procedure allows iterative generation of embeddings, as it highlights the calls to other functions as part of what makes a function similar. When trying to assign function name labels to a stripped binary, the embeddings for all functions having no calls are generated first. After assigning labels to these functions, embeddings for all functions making one call are generated next, the matching labels are assigned, and so on. At the basic level, this also encodes the API calls to the operating system or external libraries, which are known statically [2] and can be factored into embeddings right at the start of the analysis. If no functions from other libraries are used, this approach also works on the syscall level, if the number of the syscall can be resolved statically.

Concluding the normalization of the disassembly, Figure 4.3 shows the rest of the feature extraction process and the subsequent generation of data for various steps of model training. After the disassembly is normalized, it is stored in a database, together with function metadata that is required for generating labeled training data and the iterative embedding generation. The normalized disassembly cannot be processed for model training directly, it still has to be tokenized, i.e. turned from words into an integer list. For tokenization, a method normally used with the chosen neural network architecture, see section 4.4, has been selected. This tokenizer is based on the Byte Pair Encoding (BPE) compression algorithm [94] and has been invented to allow handling of rare words, without having to mark the entire word as unknown. [95] Other state-of-the-art tokeniz-

[2] There are trivial and sophisticated obfuscation techniques aiming to specifically prevent the static lookup of imported functions. This does not fall under the threat model of Ofci, but promising approaches utilizing dynamic analysis exist. [93]

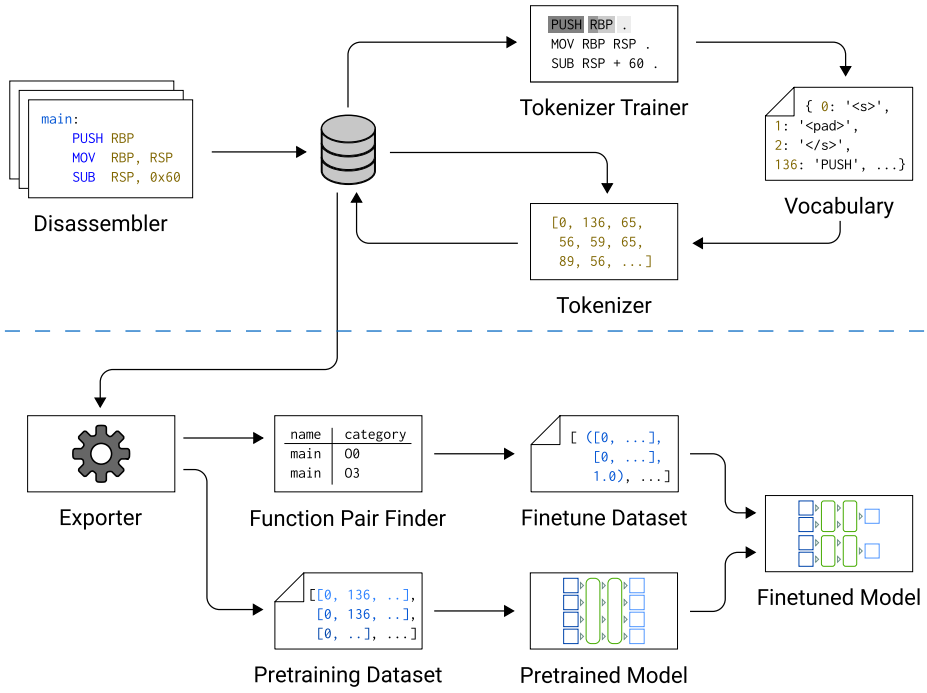

Figure 4.3: Feature extraction and training data generation.

ers such as Unigram [96] or Byte-Level BPE [97] build on the basic ideas of the BPE tokenizer and refine them for natural language; as assembly language is already very structured in itself, the improvements made by e.g., Unigram, are not necessary for the use case of this thesis. The general idea of a BPE tokenizer, or other recent tokenizers, is to prevent rare words from unnecessarily increasing the vocabulary size. In order to do that, BPE tokenizers split rare words into more common substrings and define a set of merge rules, which define when subsequent characters or substrings should be grouped together instead of interpreted as individual token. As an example, `MOV` is the most common instruction in most x86 programs, but with various different suffixes, different actions can be performed. One might encounter more common suffixes like `MOVSX` for sign extension, or rare opcodes like `MOVLPS` used with SSE instructions. Rather than adding all of these variants to the vocabulary, the opcode can be split up and have `MOV` as token and then treat e.g. `S` and `X` as separate tokens. Since `S` and `X` are single characters that can show up in suffixes of other instructions, treating these characters as tokens can decrease the base vocabulary size. The vocabulary and these merge rules are not fixed, which results in the tokenizer having to be trained on the dataset first. This step can be seen in Figure 4.3, where the BPE tokenizer trainer pulls the normalized disassembly from the database and learns vocabulary. Since the tokenizer does not inherently

know when to classify a word as rare, the desired vocabulary size has to be specified for training; the training process is short and is finished within a few seconds. When the training is done, the files containing the vocabulary and the merge rules are generated, which can now be used by the BPE tokenizer to convert the normalized disassembly into a list of numbers that can be used as input vector for a language model. The tokens for each function are stored in the database and are then used for the process in the lower half of Figure 4.3, the generation of training and evaluation data.

Two kinds of datasets are required for training Ofci: The pre-training and the fine-tuning dataset. The pre-training dataset does not require special handling, as the model is pre-trained using the Masked LM objective and masking words is handled by the corresponding data loader during training. Ideally, the pre-training dataset consists of a large number of functions that do not have to be similar to the functions used for the fine-tuning dataset. The exporter responsible for generating the pre-training set has to fetch all tokens for each function and make sure to split token lists longer than the input dimension of the model. Generating datasets for fine-tuning is more involved, as this step performs contrastive learning through a Siamese architecture. This means the model is trained on pairs of functions, rather than working with inputs from just one function, with each pair having a cosine similarity score assigned, where -1 corresponds to non-equal and 1 to equal. This not only results in the selection of function pairs, but also introduces a need for thorough dataset creation, as the fine-tuning is a supervised training and thus prone to overfitting and unlearning of the pre-trained knowledge. Another issue in training data generation is tied to the parameters of the underlying model, namely the maximum length of the accepted input, which is especially relevant in the approaches of Trex and Safe. Transformers have fixed-size inputs, and a maximum token count of e.g. 512 is fine when dealing with sentences, but function lengths can be vastly different. From the released source code of Safe it appears that functions are cut off after 128 tokens. This trivially implies that all functions that are only different after the maximum token count will be classified as the same. On inquiry, the author of Trex stated that they split the functions on the maximum token count border and classify every pair from the cross product between two function splits as similar. Ofci instead tries to adopt a different approach, with details being discussed in chapter 5.

4.4 Neural Network Architecture

Ofci is mainly inspired by Trex concerning the architecture of the main neural network. In comparison to Safe, which uses an LSTM network for language modelling, Trex builds on more recent work and uses a modified version of RoBERTa [70] to learn function embeddings. RoBERTa is an optimized version of Bert [23], a model based on the transformer architecture whose variants still achieve state-of-the-art performance on language processing tasks. When compared to an actual transformer, Bert only uses the encoder parts of the transformer, layering a certain number of transformer encoders on top of each other. However, the main idea behind Bert is not to just

layer transformers, as the typical transformer would need a decoder to perform tasks like translation and BERT only uses the encoder parts of the transformer. Instead of adding a decoder, BERT expects a task-specific neural network to be placed on top, e.g. a simple feed-forward network for classification tasks. This network does not have to be present for the complete training: BERT is first trained on a general task, such as masked language modelling, without the task-specific network; afterwards the task-specific network is added on top and the model is fine-tuned on a specific task via more training.

This process is called transfer learning and BERT is one of the current main architectures, which demonstrate the feasibility of transfer learning on practical tasks. Transfer learning in the case of language processing has the advantage of allowing training on a large corpus of data with subsequent re-use of the same trained network for different specific tasks. This makes it possible to deal with a large amount of model parameters and input data, since models are becoming more complex and are trained with increasingly large datasets. Additionally, the pre-training process is semi-supervised, e.g. when training with masked language modelling, which allows the handling of large amounts of data without labelling/preprocessing and is not prone to overfitting. On the other hand, the specific task that is trained during fine-tuning is usually supervised and therefore needs a labelled dataset and measures to prevent overfitting. While the transfer learning approach makes pre-training a one-time effort, pre-training is still expensive. This is largely not an issue for language processing, as tools and research that require existing pre-trained models have access to a large selection of pre-trained models [3] or can distill an existing pre-trained model onto their own architecture. [98] However, applying models of these dimensions to binary and assembly language processing is a recent development and pre-trained models are therefore rare. Even if a pre-trained model for a special use case exists, there is no agreed-upon representation of binary instructions, e.g. one could use raw bytes or a certain abstraction of assembly language. In addition, even modestly sized models using BERT architectures can run out of memory on consumer graphics cards, as the base version of the original BERT consists of roughly 110 million model parameters. The pre-trained model provided by TREX uses 60 million model parameters and the training checkpoint archive is about 700MB in size. While this reduces the size of the original ROBERTA and BERT to half the parameters, a reduction that is achieved by lowering the layer count, number of attention heads, and a smaller vocabulary, this is still not adequate for handling the model on consumer devices. Therefore, one goal of OFCI is to reduce the model size even further, while still producing results that are comparable to what TREX and SAFE can achieve.

One possibility for decreasing the model size is distillation, where a pre-trained large model is used as a teacher for training a student model. [98] However, the distillation process is only useful after training/pre-training the large model, a process that can still be prohibitively expensive. Instead, OFCI adopts a different architecture altogether, while still being based on BERT. This architecture is called ALBERT [99], designed specif-

[3] E.g. `https://huggingface.co/models` (Accessed 2021-12-06)

ically to reduce the model size of Bert. In order to limit the parameter count of Bert significantly, Albert introduces two optimization strategies: Factorization of the vocabulary embedding matrix and sharing of parameters between layers. While approaches like Safe and neural re-ranking techniques like Conv-KNRM [24] rely on existing embeddings or learn vocabulary embeddings as the first part of the process, the vocabulary embeddings are learned as part of the model in Bert-like architectures. A growing vocabulary size increases the size of this embedding matrix, which is in turn influenced by the size of the hidden layers. Albert factorizes the embedding matrix into smaller matrices to accommodate bigger vocabularies and hidden sizes; as the vocabularies of Ofci and Trex are small compared to natural language processing tasks to begin with, this optimization does not have a large impact on the number of parameters. However, sharing parameters between hidden layers does pay off, as a similarly sized Ofci model has 8 million parameters, compared to 60 million parameters in the corresponding Trex model; the saved model archive of Ofci is 30MB in size. While the memory savings are obvious, Albert also offers a slight speedup according to its authors, but the speedup in training/evaluation time is not within the same order of magnitude of the memory savings. The computations for the additional hidden layers still have to be performed, even if they do not need additional memory. Additionally, Albert allows grouping of hidden layers, in order to make sure parameters are only shared within a group; this is not used by Ofci, all its layers share the parameters.

In the context of Ofci, Albert is pre-trained using masked language modelling, with the disassembly features that have been previously extracted. After pre-training, a simple feed forward network, a "head" for the base network, has to be added on top of Albert. The similarity head is constructed as follows: First, mean pooling of the embeddings is calculated, then passed through a dropout layer, a single feed forward layer with the *tanh* activation function, through another dropout layer and then through normalization. This follows roughly the architecture used by Bert classification tasks, or the similarity heads implemented in Sentence-Bert [100] and Trex. While the structure of the similarity head itself looks similar to common classification heads, the difference lies in how the heads are trained. Classification relies on given labels and is trained using mean squared error (MSE) loss, but for similarity a contrastive learning approach is used. In literature, the Siamese architecture is commonly shown as two models with shared parameters, but in practice this means the two inputs are passed through the same network one after the other and fed into a loss function. The goal of the training is to adjust the embeddings learned during pre-training, so that the embeddings of similar functions have a cosine similarity score close to 1. This means the loss function has to be a cosine embedding loss, which can be derived by calculating the cosine between two embedding vectors and comparing it to the similarity label using MSE loss, but this type of loss is common enough for typical libraries to provide it out-of-the-box.[4] The network is then trained with the function pairs provided by the extractor discussed in section 4.3. An alternative to the contrastive cosine embedding loss would be a triplet loss, taking a

[4] `https://pytorch.org/docs/stable/generated/torch.nn.CosineEmbeddingLoss.html` (Accessed: 2021-12-06)

function, a similar function and a dissimilar function. Ofci has chosen the contrastive cosine embedding loss due to easier handling of function pairs compared to triplets, and pairs being prevalent in related works. [1, 11]

4.5 Virtual Machine Analysis

Analysis of obfuscation through virtualization is still an active research field, as discussed in section 3.3. While there is literature on deobfuscating virtualized code, research dealing with virtualization in the context of function clone detection is scarce. Another recent survey [101] highlights the need for covering interprocedural virtualization obfuscators like Themida [20], or VMProtect [21], as the obfuscations applied by Obfuscator-LLVM [18] do not appear to be harder to solve than cross-optimization binary similarity. Ofci does not solve interprocedural obfuscation either, but is intended as a stepping stone to expand research in this area. To this end, Ofci aims to perform binary similarity on functions virtualized with Tigress [17], which works by virtualizing functions separately. Previously, Tigress has only been discussed in Asm2Vec [10], but the obfuscated code has only been analyzed *statically*, whereas Ofci analyzes virtualized code *dynamically*.

When virtualizing code, the original code of a function is translated into a new language, i.e. bytecode, which is intended to obfuscate the original intention of the instructions. When two functions share the same bytecode, it is no longer possible to distinguish the two functions based on their instructions. Since static analysis cannot make sense of the attached bytecode, it is not able to derive the differences between the functions. If the program is virtualized interprocedurally, as with Themida, there are no function boundaries that can be discovered statically altogether. Therefore, it is necessary to incorporate some form of dynamic analysis into the similarity detection process. While there are approaches that use dynamic information to facilitate code clone detection, as discussed in chapter 3, at the time of writing, no approaches use dynamic analysis to detect function clones across virtualized code. Ofci fills this gap, by monitoring the virtualized code dynamically: The program is run and Ofci collects traces for the given inputs. Afterwards, the traces are fed into the training/inference machinery in place of the static disassembly, and the similarity score is calculated in relation to the trace.

The traces used for this process are plain instruction traces, without additional information, such as register values or memory accesses. This additional information could be used to make the analysis more precise, as shown in a similar manner with microtraces by Trex. Since Ofci does not use additional data, the instruction traces do not have to track every instruction, only the entrance and exit of basic blocks is tracked and written to a log file. This solves an additional problem, as the produced disassembly needs to be in the exact same format as the one by the disassembler. Due to slight differences in disassembly output, the disassembly is not generated within the tracer, but the log file from the tracer is loaded by the disassembler. The same disassembler plugin taking care of the dataset generation for the training process is then also processing the list of basic block addresses, going through every basic block and determining the function it origi-

nated from. In this basic version, calls from functions under analysis are not handled, i.e. if the virtualized code called into another virtualized function, it is not represented correctly in the generated function tokenization; in order to track control-flow transfers to different functions, the call stack needs to be simulated during analysis time.

4.6 Evaluation

For comparison with other approaches and as preparation for practical application, the database containing the training data has to be processed first. This process can be seen in Figure 4.4, which starts with an embedding generation program. The embedding generator takes all functions stored in the repository database and uses the fine-tuned model to generate embeddings for them, which are stored in the database along with the metadata. When encountering a function that is longer than the maximum input dimension of the model, the tokenized function is split into chunks at the maximum input length border. Since embeddings are vectors and similar vectors point in the same direction, the embeddings of all chunks are averaged to represent the overall function vector.

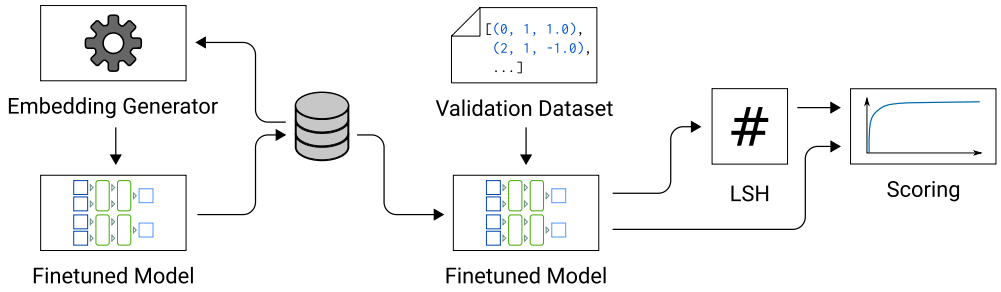

Figure 4.4: Embedding generation and validation scoring.

After all embeddings have been generated, Ofci can be used for inference in production. Additional steps are taken for evaluating the performance of Ofci: A validation set of function pairs that have not been used for fine-tuning the model is required. This set can be the same one described in section 4.3, but any set of function pairs can be used, as long as the functions have not been used during the fine-tuning step of the Ofci pipeline; the exact composition of the validation sets is being described in the respective sections of chapter 6. The metadata and embeddings for every function pair in the validation set are pulled from the database and then processed and scored according to the used metrics. The main metrics used for evaluation are the *Receiver operating characteristic* (ROC) and *Precision at rank 1* (P@1). As function clone detection is a knowledge retrieval problem, the main metrics are precision, i.e. the ratio of relevant functions in search results, and recall, i.e. the ratio of relevant functions out of all relevant functions.

4.7 Inference and Application

When applying Ofci in production use, i.e. when reverse engineering an unknown binary, the process is different to the evaluation, as not all information can be pulled from the database. Instead, the binary under analysis has to go through all steps of the pipeline, as shown in Figure 4.5. With its basic building blocks, the inference pipeline consists of four separate steps: Disassembly, tokenization, inference and lookup. In the first step, a disassembler has to provide the disassembly text of a function. In the case of simple obfuscations or compiler optimizations, the static disassembly alone is enough, but when virtualized code is present, a trace of the function is required. Ofci does not currently support automatic trace creation; if a trace is required for analysis, it has to be created manually using the tracing tool provided by Ofci. The reason why this is not done automatically is the nature of dynamic analysis, as a function under analysis might not be seen by a trace, if the wrong input is provided to a program. This could be solved using, e.g. coverage guided fuzzing or symbolic tracing from the function entrypoint, but is out of scope for this thesis.

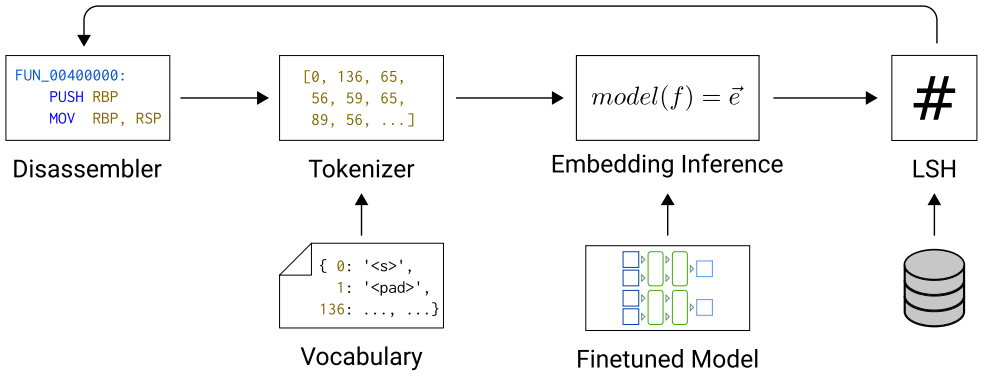

Figure 4.5: Resolving a function through embedding inference and lookup.

After the disassembly has been extracted, using either the disassembler's inbuilt facilities or the Ofci tracing tool, it has to be processed using the tokenizer. The tokenizer only requires a vocabulary file in production and works the same as when generating training data. The tokenized disassembly is then used as input for the embedding inference, which requires a snapshot of the fine-tuned model. Most machine learning frameworks provide means for packaging and integrating trained models into production environments and Ofci makes use of this to integrate the inference process into the disassembler plugin, without having to ship large dependencies. Lastly, the generated embeddings are used as queries for the LSH lookup into the database, filled with known functions. The lookup will always return a function from the database, therefore the cosine similarity

is calculated and a threshold applied, in order to avoid false positives when the cosine similarity is not conclusive. The throughput of the inference pipeline can easily be increased with parallel processing, as tokenization and LSH lookup is independent per function. The embedding inference is a bottleneck, as fast inference requires the use of a graphics card and only one inference step of the model can be executed per GPU. To combat this issue, the GPU memory usage has to be maximized, by not just moving a single tokenized chunk to GPU memory for inference, but a whole batch of tokenized chunks. The GPU can then perform the inference step on every chunk in the batch at the same time.

CHAPTER 5

Implementation

While recent approaches for function clone detection perform well on paper, code is rarely published or relies on closed-source software, such as IDA Pro [14], making independent evaluation hard or impossible. Safe [11] is fully open-source, Asm2vec [10] relies on IDA Pro, Gemini [15] and TREX [1] publish code for their respective machine learning models, but don't provide code for the feature extraction process. From an engineering perspective, the goal of this thesis is to build an end-to-end solution called Ofci, a set of tools covering feature extraction, model training and verification/application. The main requirement for Ofci is to only rely on recent and publicly available open-source software and to publicly release all parts of its own pipeline. This allows for independent verification of evaluation results and practical usage outside of academic applications. The following sections cover all major parts of the Ofci framework and discuss their implementations in detail.

5.1 Technology Stack Overview

At the core of the Ofci framework is Ghidra [5], which is used as disassembler and evaluation tool. Released publicly in 2019, Ghidra [5] is an environment for software reverse engineering, offering a similar set of features as IDA Pro [14], but at no cost and with its source code being publicly available. As development now continues in public and with Ghidra offering an API, good extensibility through its plugin architecture, and it being user-friendly, it has been selected for realization of the Ofci architecture. This choice also potentially allows the approach to be used in real world day to day reverse engineering work. Ghidra offers both a Python [102] and a Java [103] API, with Ofci making use of the Java API, because Ghidra interprets Python using Jython [104] within Java. This allows simple script execution within Ghidra, but Jython is limited, does not offer full Python compatibility, and adding additional packages for, e.g., database connectivity can only be done through the Java API. Using Java also comes with the

benefit of packaging as an extension, which allows code reuse among different parts of the Ofci pipeline, providing a single interface for training data generation and function clone detection application. Building the extension creates a *jar* package that can be distributed and installed cross-platform directly from the Ghidra user interface.

The Ghidra plugin takes care of extracting data from the disassembly and program database and stores the normalized disassembly together with function metadata in a PostgreSQL [105] database. The choice of PostgreSQL and SQL in general was made in consideration of training data generation: The fine-tuning training requires the matching of functions with the same name within the same tool, which is straightforward to do within SQL. When keeping application of Ofci by a user in mind, packaging or requiring a database server is not ideal and when Ofci grows beyond a research prototype, a use-case specific data storage format is required. For processing a large dataset in preparation of pre-training or fine-tuning, Docker [106] is leveraged to create a PostgreSQL container and spawn several Ghidra containers that can process a large dataset in parallel. For data processing that is not handled in the Ghidra plugin Rust [107] is used. The data processing tasks covered by the Rust tooling involve generating the datasets used for model training and querying/or modifying the extracted data in the PostgreSQL database. This has to be fast for quick iteration in prototype development and is desirable in production use as well; Rust fits the requirements as a safe, low-level language without garbage collection.

An additional factor for selecting Rust is the existing tooling for natural language processing (NLP): The Transformers library [108] built by the Hugging Face community uses Rust for implementing fast and parallel tokenizers, while Rust-Bert [109] provides a Rust port of the Transformers library. Transformers also is the library at the core of all machine learning operations in the Ofci pipeline. In a previous prototype fairseq [110], which is maintained by the Facebook machine learning team and has been used with Trex, was used, but Transformers was chosen due to the larger range of supported models and a cleaner API for providing extensions. Despite the existence of Rust-Bert as a port of Transformers, the original Python library is used for the training and evaluation of the models. This allows quick fixes and changes of parameters without having to recompile the program every time, as well as providing the means to plot and visualize the results. Python is also used at several different stages to automate simple tasks like sorting the original dataset into a folder structure that can be used by the Ghidra extractor. For training and evaluation, the PyTorch [111] backend is used and the Rust code also makes use of PyTorch, as Rust-Bert links to the Rust bindings of libtorch. For embedding lookup, Faiss [112] by Facebook Research is used. Besides providing exact embedding lookup, Faiss also allows speeding up queries at the cost of precision.

Because the Python API of Transformers is used for training the datasets processed and generated by the Rust data processing tool, a file format for exchange of large amounts of floating point data was needed. Python pickling, the standard serialization of Python, was too slow and custom raw binary formats would produce large files; the HDF5 file format [113] has been chosen, as it is designed specifically for exchange of

scientific data and therefore well suited for numerical data. It also has implementations for the relevant languages in the OFCI pipeline and loading the datasets used in chapter 6 finishes within a matter of seconds. Finally, in order to perform the tracing required for detecting virtualized function clones, a performant binary instrumentation framework is needed. Most binary instrumentation frameworks do not run the full binary from the start, as the interaction with the operating system creates overhead and additional complexity. This also requires setting up a harness for the binary, allowing it to be traced within the framework. As tracing the binaries should be as straightforward as possible, OFCI aims to avoid the additional harness and traces the full binary, including the operating system interactions. Intel Pin [114] was selected for creating traces, as it fulfills the mentioned criteria.

5.2 Feature Extraction via Ghidra

The OFCI plugin to Ghidra provides an API for extracting the necessary features from the disassembly text and a service for connecting to a PostgreSQL database in order to store the feature information. It follows the same directory and code structure as the sample plugin provided by Ghidra and implements several tasks that make use of the OFCI API as Ghidra scripts. The core part of the plugin is a script called *OFCIExtractor*. This plugin takes care of generating the feature data needed for training of the OFCI model and can be executed for one specific binary from the Ghidra GUI, or can be run in batch mode to process a large number of files at once. The feature extraction process consists of the following steps:

1. **Extraction of Function Symbols**. In the first step, a list of function symbols is extracted through the API. Special care has to be taken of external symbols, as these are being listed twice, for their respective PLT and GOT addresses.

2. **Extraction of Assembly Instructions**. As instructions are the features that we will embed, the next step is to extract all assembly instructions that belong to a function.

3. **Normalization of Opcodes and Arguments**. Before a tokenizer can create numeric vectors necessary for training the embeddings, the Ghidra plugin first has to normalize the instruction disassembly to deal with information that cannot easily be tokenized, such as numeric values or too many details in opcodes and arguments.

4. **Export of Extracted Features**. After the disassembly instructions have been normalized, they are stored in a database together with the necessary metadata needed for training set generation.

Collecting all function symbols in the binary serves multiple purposes: Providing the function entry point for disassembly of the function, as well as the function names

for labelling and for identifying call targets, since the call targets are used as special features with respect to the embeddings. Calls to external functions, e.g. syscalls or libc functions, are relevant for the call ID feature of Ofci: When not directly obfuscated, calls to library and operating system functionality can help to identify an otherwise obfuscated function. Ghidra handles these external functions as two separate function symbols: The GOT entry, which is being marked with an *external* flag, and the PLT address, which is the one actually being called. When iterating over all function symbols the extractor has to remember functions marked as *external*, but then only keep track of the address of the function with the same name that is not marked external, as this corresponds to the PLT entry. When later looking up a called function address, it will yield the correct function name. While assembling the list of relevant function symbols, *thunked* functions are discarded and not further analyzed. These functions are small code segments, e.g. trampolines, that redirect the caller to the actual function. The name and address of these function stubs are kept for the call ID feature, but their disassembly is not exported.

The list of function symbols is stored in the PostgreSQL database through batch inserts, assigning every function name a unique ID, or retrieving the ID if the function name is already present in the database. The retrieved IDs are then stored in a function lookup map, where functions can be looked up using either name or address of the function entry point, for making the call ID feature available later on. As the function names alone are not a unique identifier, the category, project and tool name are stored in the database as well. This structure is not ultimately necessary, but arose from the way the Trex dataset is built. Category corresponds to the process through which a binary was compiled, e.g. with *O3* compiler flags or optimizations. The project name is necessary, as there are several projects like *coreutils*, which are included in the dataset but consist of multiple binaries. Lastly, the tool name corresponds to the name of the actual binary file, e.g. *ls* or *mkdir* from *coreutils*.

After storing the binary and function name metadata in the database, the list of functions is processed by the extractor. As functions are not always laid out linearly in memory, the Ghidra API marks the contents of a function with a set of address ranges. The correct identification of what code ranges belong to a function can only be identified when enabling the *Decompiler Switch Analysis* in the analysis overview. Ofci takes this for granted; while it does not rely on the full CFG of a function being reconstructed, it does require an approximate range of which instructions belong to a function. The address ranges provided by Ghidra are iterated in forward order, sorted by the starting address of the range block. The interface provided by the Ofci plugin accepts these address ranges for disassembly and normalization, which is the same interface used by the trace analysis in section 5.5, as basic block information provided by traces directly translates to an address range set in Ghidra.

In order to train embeddings on the disassembly instructions, these have to be tokenized into numeric tokens. While the Ghidra plugin itself does not take care of the tokenization, it has to pre-process and normalize the text from the disassembly according to

uniform rules, which can then later be tokenized. As x86 disassembly is not defined in a precise way, e.g. there is AT&T and Intel syntax, and Ghidra also introduces variable naming for memory offsets instead of showing the actual offsets, OFCI has to process the disassembly to remove these discrepancies. This is done within the Ghidra plugin, because the API provides a way to analyze and modify the generated disassembly from scratch, instead of having to parse it from the text itself. Another benefit is that missing information can be pulled through Ghidra's API directly from the original data source without additional steps. An example for this are call and jump targets, where Ghidra can perform additional analysis, when the instruction text is either referring to an unknown address or the jump is register-based. Normalizing the disassembled instruction also requires design decisions in regards to which part of the instruction should actually be treated as a separate token and which parts should be combined. This becomes apparent when taking `LEA` instructions, or any instructions using the advanced x86 memory addressing as can be seen in Listing 10. In this case, the part of the instruction responsible for calculating the memory address can be either split up into its respective smaller tokens, or interpreted as a group. Ghidra fortunately represents complex memory addressing operands as operand groups and provides means of iterating the representation of an operand before it gets converted into a string. Each element of the iterator is either a register, scalar, address or a character. Due to words like *qword ptr* being presented to the iterator character by character, parsing the iterator is complicated. OFCI works around this by stripping the pointer size qualifications altogether and only accept characters from the iterator if they correspond to a separator or arithmetic character like `+`, `-` and so forth.

```
1  MOV RDI, qword ptr [RBP + -0x30]      ; 488b7dd0
2  MOV qword ptr [RBP + -0x6d0], RAX     ; 48898530f9ffff
3  CALL <EXTERNAL>::strlen               ; e809c8ffff
4  MOV RDI, qword ptr [RBP + -0x6d0]     ; 488bbd
5  LEA RAX, [RDI + RAX*0x1 + 0x11]       ; 488d440711
6  AND RAX, -0x10                        ; 4883e0f0
7  MOV RCX, RSP                          ; 4889e1
```

Listing 10: Disassembly produced by Ghidra showing complex memory addressing arguments.

A similar design decision is required when dealing with scalar operands or memory addresses, as scalar operands have been handled with care in previous work [11, 1] and memory addresses, such as call addresses, are an integral part of the way OFCI handles call IDs, as described in chapter 4. As earlier Ghidra presents operands and parts of a complex operand group as iterator again, but this time the focus is not on literal characters. While registers are passed through normalization unchanged, a distinction is made between scalars and memory addresses. If a scalar is encountered, it is encoded in the way described in section 4.3. The same goes for addresses, but in case the address was found in the function lookup map, the call ID stored in this map is inserted instead

of the address value. While thunked functions are not passed to the normalization step, their names are still stored in the database and therefore they also have an assigned ID. In practice this means that even if a function is external and only a stub is present, i.e. there only is the PLT jump code, the name of the external function still has a valid ID and can be referenced from a memory address operand. Additionally, the normalizer tracks the number of valid references made to functions in the function lookup map; this information is later used for iterative function similarity detection. Similar to function call references, syscalls can be interpreted as call references as well; in case of Linux, syscalls can be mapped to the same names exported by libc. On x86, syscalls do not have their respective syscall number as operand to the syscall instruction, which leads to additional effort required when this information is used for analysis purposes. Fortunately, Ghidra itself contains a script for deriving the syscall number when it stumbles upon a syscall instruction. Ghidra makes use of inbuilt emulation capabilities and provides a script called *ResolveX86orX64LinuxSyscallsScript*, which tries to resolve the syscall number based on a simple constant propagation algorithm. In order to make use of this information in the exporter, the script has to be called before using the Ofci exporter.

This normalization process is handled by a number of different interfaces in the Ofci API, which can be reimplemented in order to achieve a different normalization. At the bottom of the hierarchy is the *Tokenizer* interface, which contains a string buffer for storing the normalized disassembly while processing a function, and which takes care of adding integers to the string buffer as described in section 4.3. The *InstructionParser* interface takes care of normalizing one instruction and its operands/operand groups, while *FunctionParser* is the equivalent for normalizing a whole function, based on an address range set. The *FunctionParser* can make use of *FunctionMetadata* to retrieve the data about the current function from the Ghidra API, as well as *GlobalFunctionInfo*, which stores the function lookup table required for the call ID feature. After all address ranges of the function have been parsed, it can be passed to the *Dao* if the plugin is connected to the PostgreSQL database. The *Dao* will either remember the normalized function for batch insertion, or directly insert the function into the database. Besides storing the function name, category name, project name and tool name IDs, the database also contains a hash of the normalized disassembly, allowing quick identification of exact function clones.

Ghidra allows for batch processing of input files, using a directory of files as input, instead of a single file. While this allows processing of a large number of files, some limitations apply, as Ghidra is generally slower when compared to other disassemblers. [115] However, Ghidra allows deactivation of certain analysis options and as decompiled code is not required, the number of needed analysis passes can be reduced to a small set. In headless mode, Ghidra provides statistics for runtime of the individual analysis passes, allowing profiling; the pass consuming the most time is *Decompiler Switch Analysis*. Unfortunately, testing has shown that disabling this analysis prevents instructions from being discovered, as the basic blocks of a jumptable are no longer discovered, even

if they are directly next to the function and would be discovered by a linear sweep disassembler. This does not only affect obfuscation, but already happens in switch statements of normally compiled programs. Therefore, the switch analysis pass has been kept enabled, resulting in long analysis times as shown in chapter 6. In general, every analysis pass that does not impact the extracted features, has been disabled.

Static code archives on Linux present somewhat of a challenge to Ghidra in headless mode. As they are used by the dataset collected for the evaluation of TREX [1], they are crucial for the evaluation of Ofci, as this dataset is used to compare the base performance in chapter 6. The issue for Ghidra is the archive nature of these static libraries: Instead of being one contiguous binary file to analyze, a static library is an archive of object files that can't be directly opened with Ghidra. In the graphical user interface, Ghidra presents several options on how to handle the static library on import, but none of these appear to be available in the headless analyzer. [116] As a workaround, all static libraries are unpacked in a new subdirectory and all object files contained in the static library are treated as a separate binary to be analyzed. Through naming conventions the metadata of the extracted functions is then later merged when inserting it into the database, making it appear as one single project again.

In order to make better use of Ghidra's batch processing, Ofci uses Ghidra within Docker containers as an easy way to run it on an otherwise empty virtual machine. By default, Ghidra runs the analysis in a single thread and while it does support multi-threading, this functionality has not been used while developing Ofci. Rather than running analysis multithreaded, Ofci uses the structure of the Trex dataset and splits binaries into their categories, such as *O0* and *O3* for compiler optimizations. Then a new analysis container is started for every binary category, distributing the analysis load. During development of Ofci many iterations of the extractor plugin were necessary, which resulted in splitting the Ghidra extraction into two steps: The analysis step and the extraction step. For the analysis step, the Docker container would just load the binaries into a Ghidra project, perform the analysis, and save the project. The extractor container can then just open the project file without performing the analysis and immediately start with the extraction after the project has been loaded. Running the analysis with the help of Docker containers comes with another advantage: Hosting the required PostgreSQL server is simplified to just downloading the official container image and starting it. The extraction container then has to be created within the same Docker network the PostgreSQL container is running in.

5.3 Processing Exported Feature Data

After the normalized disassembly and metadata has been stored in the database, further processing is necessary. The disassembly still has to be tokenized, i.e. turned into a numeric vector instead of text, and datasets for model training need to be generated for tokenized data. As the database reaches multiple gigabytes of data on the Trex dataset alone, efficient processing is needed, which is why the data processing tool is written

in Rust, allowing fast iteration times when re-tokenization of data is required. All processing steps from the Ofci pipeline are handled within a single Rust library, having multiple different command line tools as frontends. These processing tools perform the following steps:

- **ofci-genvocab**. Before the database can be tokenized, a vocabulary for the tokenization process has to be generated. As outlined in chapter 4 a BPE tokenizer is trained and used, specifically the BPE tokenizer as implemented in the Rust tokenizers of the Transformers library [108]. As the vocabulary size is already very small compared to natural language vocabularies (roughly 800 entries), training the tokenizer is fast and produces a vocabulary file and the merges list typical for BPE tokenizers.

- **ofci-tokenize**. With an existing vocabulary, the tokenize tool simply fetches the normalized disassembly and stores the tokenized disassembly in the database. The length of the tokenized disassembly is not chunked or cut in this step, the full tokenization of the function is stored.

- **ofci-generate-pretrain**. For generating the pre-training dataset, the whole function database is fetched, the tokenized disassembly is chunked and written to a dataset file.

- **ofci-generate-finetune**. Generating the dataset for fine-tuning requires more effort than pre-training dataset generation, as function pairs need to be matched. Additionally, the dataset needs to be split into training and validation sets, because the fine-tuning is a supervised learning process.

All of these utilities require access to the database, which is managed by the Diesel ORM Mapper [117]. The schema for the database layout is managed with Diesel, allowing the migration management to initialize the PostgreSQL database in the first place. The database layout is small, as the database stores nothing but the names of functions, categories, projects and tools, with one table storing all function related data. The queries needed to process the data are simple select and update queries; therefore, and due to heavy reliance on compile-time code generation, the used Diesel ORM functionality do not incur significant overhead and provide convenience. Diesel does not currently provide iterators on query results based on cursors, which means in the current implementation of Ofci, queries fetching all functions will fetch all data at the same time. This did not matter on the machines used to generate the datasets, but for larger datasets a less memory-heavy solution is required. Initializing the database is currently not done automatically: The PostgreSQL server is started as a container with the help of *docker-compose* and it has to be manually initialized by executing the Diesel migration command.

After the database has been filled by the Ghidra extraction plugin, the **ofci-genvocab** utility has to be executed. As the name implies, this will start the vocabulary generation process, meaning all the instructions that are going to be tokenized have to be in the database by then; otherwise opcodes or arguments will be interpreted as out-of-vocabulary tokens later on. First, the utility fetches all functions from the database into memory and the tokenizer trainer will pre-process the disassembly text by stripping whitespace from the beginning and the end of the text, as well as performing Unicode normalization. The latter is not strictly necessary, as all text generated by the Ghidra plugin should be ASCII text anyways, but the tokenizer was carefully initialized in the exact same manner the Transformers library initializes the RoBERTa tokenizer when called from Python, to avoid introducing discrepancies between the tokenizer implementations. The BPE trainer is initialized to a maximum vocabulary size of 1000 and trained on the functions retrieved from the database, with the training being finished in a matter of seconds, while pulling the functions from the database and preprocessing the functions can take up to 40 minutes. After the BPE training finishes, the files containing the vocabulary and merges are stored at the location specified on the command line.

The **ofci-tokenize** utility initializes the tokenizer with the generated vocabulary file and pulls all functions from the database again. The tokenization can be performed in an *embarrassingly parallel* manner and every token list is stored in the database through an update of the function's row. The list of tokens is additionally serialized as JSON data before being stored in the database. Some form of serialization was needed to store the token list and JSON was chosen due to straightforward visual inspection when performing sanity checks on the dataset. Surprisingly, the overhead of storing the token lists, a list of small integers, as JSON instead of a binary format is negligible within PostgreSQL. Due to the vocabulary being small and the most common tokens being composed of one or two digits in JSON format, the JSON of a token list is only marginally bigger than the equivalent representation as an array of 16-bit integers. Additionally, PostgreSQL appears to be efficient at storing text data, compressing the full dataset including disassembly text and token list JSON down to 4GB, while a binary dump of 16-bit integers with padded chunks for pre-training results in a 7GB dataset file.

For creating the pre-training dataset, the **ofci-generate-pretrain** utility performs a mapping of the full function database to a hdf5 dataset that can be imported in a Python script for model training. As the pre-training of the model is handled in a self-supervised manner, the creation of labels is not needed and the tokens can largely be passed onto the dataset file unmodified. However, because the model's input size is limited to 512 tokens the token list of a function has to be processed accordingly, i.e. functions from the database have to be broken up in chunks with a maximum of 512 tokens each. The JSON token list of a function is first deserialized into a numeric array, and then cut into chunks. The memory for chunks is allocated upfront and every chunk has one pre-defined location in the allocated memory, based on the function and chunk index. This allows for simple parallel processing, as only one thread will access a specific chunk location at a time, speeding up the JSON token list decoding and copying to the

chunk location. In order to speed up loading and processing times, all chunks are padded to the maximum size of 512, using the padding token of the vocabulary, allowing the full dataset to be stored as a fixed-size tensor that can be loaded efficiently in the Python scripts. To save memory and loading times, the tokens are stored as 16-bit integers, which is the smallest integer that can store the vocabularies used by Ofci.

The step performed by **ofci-generate-finetune** is the most elaborate one within the Ofci data processing library. In comparison to generating the pre-training data, the fine-tuning data requires function pairs, due to the contrastive learning approach. Additional care is required due to the nature of this training approach: While the pre-training is self-supervised, fine-tuning is supervised and requires a labelled dataset. Supervised training is susceptible to overfitting and the used dataset has to be balanced, making sure bias is not introduced intentionally or unintentionally. In the first step of the process that aims to achieve this, the metadata of all functions is pulled from the database. The x86 dataset from Trex together with the newly introduced binaries and traces by Ofci adds up to roughly 180000 unique function names. This number does not seem large on its own, but for every function name a number of different versions and versions in different categories exist, allowing for roughly 180000^8 possible function pairings. Not all functions are included in this pairing; the excluded functions can be seen in Listing 11. The reason for leaving these functions out is that they exist in almost every binary, do not contain a lot of information in itself, or can be found trivially by most static analysis tools. With all the metadata available, the goal of the utility is to now select a random subset from the amount of all possible pairings.

```
1 _init
2 frame_dummy
3 __do_global_dtors_aux
4 register_tm_clones
5 deregister_tm_clones
6 __libc_csu_fini
7 _start
8 __libc_csu_init
```

Listing 11: Functions that have been excluded from fine-tuning dataset generation.

The selection of function pairings is limited by several bounds. The main upper bound is the number of chunks to be generated by the function pairing algorithm. The algorithm limits the number of chunks instead of the number of functions, as even functions with the same name may produce a vastly different amount of chunks based on the length of their token lists. The best example for this is the pairing of functions with a trace, where a trace can be several times longer than the actual function. The number of chunks is also not an exact limit; the pairing loop is simply stopped when the number of chunks exceeds the defined limit. Another global limit to function pairing is the amount of negative pairs that follows on a positive pair. Trex generates 5, Ofci generates 2 and Safe generates one negative function pair on one positive pair. Trex cites practice in

contrastive learning [118] as reason for selecting a 1:5 ratio, referring to the distribution of dissimilar functions when applying function clone detection. [1] However, the ratio of dissimilar functions is larger than 1:5 in practice, and SAFE produces good results with simple triplet learning, i.e. one similar and one dissimilar function, therefore OFCI chose the middle ground. Besides the ratio of similar and dissimilar functions, OFCI tries to distribute the selected functions fairly across different obfuscations. The categories of the dataset, i.e. *O0*, *O1*, *O2*, *BCF*, *CFF* and so on, are used to create a frequency table. Whenever a function pair is picked, the chunk counts of the respective categories are increased in that frequency table. The frequency table is used to calculate probabilities for selecting the next obfuscation category, using 0.9999^n as formula with n being the number of fragments already processed in the corresponding obfuscation category. The goal is to create roughly 50000 chunks for each obfuscation category, the formula makes sure that the probability for selecting a certain category is below 1% when this goal is reached. Some special handling is required for the cases where there is no function clone in the other category; in this case further processing is withdrawn and another two obfuscation categories and functions are picked. This also prevents issues when whole categories do not share similar functions, as is the case with *virt* and *bcf*. In general functions can be selected multiple times to achieve a large number of pairings, while function pairings themselves are unique within the dataset.

When the function pairs are selected, only their metadata is selected and stored in a list. To turn the list of function pairs into the final dataset, the same process for chunk splitting from the pre-training data generation is used. The difference in this case is that one row in a dataset now consists of three parts: A chunk from the first function, a chunk from the second function and a label indicating whether the functions are similar or not. As the number of chunks is likely different for functions in different categories, OFCI takes the minimum number of chunks shared within the function pair. This means, if one function is 2 chunks long and another version of the function consists of 5 chunks, only 2 chunks are selected from each function. The chunks are matched up linearly, i.e. the first chunk from the first function is matched up with the second chunk from the second function. After all chunks have been extracted, the dataset is stored *column-wise* in the exported hdf5 file, with the two function's chunks being stored in two different tensors and the labels stored in an additional tensor. This allows for easier processing on the Python side and for storing the labels as 8-bit integers, the smallest possible datatype in hdf5. Additionally, a list of functions that have been selected is stored alongside with the dataset, so the evaluation scripts can later determine which functions to use for performing validation metrics. For validation during the model training, chunks of the same functions can still be used if they haven't been used for training, for the evaluation afterwards, only chunks from functions that have not appeared in the dataset are used.

5.4 Model Training

The aim of the model implementation is to make sure, to reuse as much code from Transformers and PyTorch as possible, as the architectures in Transformers are constantly

evaluated and used by researchers. By only making minimal changes to the architecture, underlying models can be swapped out when needed, as long as they follow the same architecture and objectives as Albert, Bert or RoBERTa. All code for model training is written in Python and consists of several main parts: The data loader, taking care of loading the datasets from the hdf5 files and serving it to the trainer, the model implementation and the configuration and training process. While the Python scripts for the training do not contain a lot of code, development was not straightforward, as even deviating slightly from the usual tasks performed with Transformers requires reading up on the code of the library itself, because a lot of convenience features are not directly documented. These features seem to rely on an engineer just assuming some Python keyword arguments are interpreted correctly; whether the relevant data fields are passed along correctly requires studying the source code and testing.

```python
class OFCIPretrainDataset(Dataset):
  def __init__(self, data_tensor):
    self.elements = data_tensor
    self.amask = (data_tensor != PADDING_TOKEN).float()

  def __len__(self):
    return self.elements.size()[0]

  def __getitem__(self, idx):
    return { 'input_ids': self.elements[idx], 'attention_mask': self.amask[idx] }

def load_data(file):
    with h5py.File(file) as f:
        idx = torch.randperm(f['data'].shape[0])
        data_shuffled = torch.tensor(np.array(f['data']))[idx][:1_500_000].long()
        return data_shuffled[:10_000], data_shuffled[10_000:]
```

Listing 12: Data loader for pre-training.

First, the vocabulary and dataset files are loaded, with the dataset being fully loaded into memory. Custom datasets in Transformers are handled by inheriting from the PyTorch *Dataset* class. In documentation examples, tokenization is usually performed directly within the data loader, where the existing tokenizers in the Transformers library generate data exactly in the way it is needed for the model. As the tokenization is already taken care of during the data processing phase in Ofci, the data loader just has to pass on the data in the correct format. This happens in the implementation of `__getitem__` within the data loader, which accepts an index and expects an element in return. However, Ofci needs to pass additional parameters to the model; how this works is hinted at in the documentation of Transformers, but not explained. If the `__getitem__` implementation returns a Python dictionary instead of one single element, the dictionary entries are passed as keyword arguments to the model. In the case of a labelled dataset, i.e. the dataset for fine-tuning, the data loader provides the labels as `labels` entry in the dictionary. Because the dictionary is directly passed to the main

model function internally, it does not contain single elements, but tensors equal in size to the specified batch size for training. In order to select the correct elements from the dataset, Transformers passes an index tensor to `__getitem__` instead of a single index, allowing the efficient selection of multiple dataset elements.

Besides providing labels in the dictionary, not using the tokenizer in the data loader also requires OFCI to pass on an attention mask. This attention mask is passed on to the transformer model, limiting the self-attention mechanism on certain unwanted tokens. In the case of OFCI, this means the attention mask needs to mask all tokens corresponding to the padding token, which can be done easily through making use of inbuilt PyTorch tensor operations, as shown in Listing 12. As hinted by the `float` call at the end of the attention mask assignment, the data passed on to the model also needs to be converted to a specific data type. This is also done when loading the dataset from the hdf5 file, i.e. in `load_data`, where the token tensor has to be converted to the `long` datatype. Transformers requires this for it's masked LM data collator; the reasoning behind this is unclear as using a 64-bit integer for representing tokens appears to be a waste of memory, even for natural language vocabularies. Another peculiarity of Transformers is the slow import of hdf5 datasets when directly passed to `torch.tensor`[1], which is why `f['data']` is passed to `np.array` first in the code snippet.

```
class AlbertSimilarityHead(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.dense = nn.Linear(config.hidden_size, config.hidden_size)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.out_proj = nn.Linear(config.hidden_size, config.hidden_size)

    def forward(self, features):
        x = torch.mean(features, dim=1)
        x = self.dropout(x)
        x = self.dense(x)
        x = torch.tanh(x)
        x = self.dropout(x)
        x = self.out_proj(x)
        x = F.normalize(x)
        return x
```

Listing 13: Similarity head for an ALBERT transformer network.

For pre-training, OFCI uses the unmodified ALBERT model as provided by Transformers, specifically the inbuilt version with a masked LM head. The masking of the tokens necessary for masked LM is performed by the Transformers data collator, with a probability of 0.2. When it comes to fine-tuning, some manual effort is required. While the Transformers library does provide several fine-tuning heads, there is no head for contrastive learning or sentence similarity. The structure of the head network is roughly the same as

[1] https://github.com/pytorch/pytorch/issues/28761 (Accessed: 2021-12-06)

in the code provided by TREX or Sentence-Bert [100] and can be seen in Listing 13. First the outputs of the Bert part of Albert have to be mean-pooled before processing. Afterwards, the outputs are passed through a dropout layer, through a layer using *tanh* as activation function, and then finally through another dropout layer before being normalized. Dropout layers randomly and intentionally discard inputs with a certain probability, which trains the network to generalize better and to prevent overfitting.

```python
class AlbertForSequenceSimilarity(AlbertPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.config = config
        self.albert = AlbertModel(config)
        self.similarity = AlbertSimilarityHead(config)
        self.init_weights()

    def forward(
        self,
        input0: torch.LongTensor = None,
        input1: torch.LongTensor = None,
        attention_mask0: torch.LongTensor = None,
        attention_mask1: torch.LongTensor = None,
        labels: torch.FloatTensor = None,
        ...
    ) -> SequenceSimilarityOutput:
        # Feed first function fragment to model
        outputs = self.albert(input0, attention_mask=attention_mask0, ... )[0]
        logits0 = self.similarity(outputs[0])

        # Feed second function fragment to model
        outputs = self.albert(input1, attention_mask=attention_mask1, ... )[0]
        logits1 = self.similarity(outputs[0])

        # Calculate cosine similarity loss
        loss = F.cosine_embedding_loss(logits0, logits1, labels, margin=0.1)
        return SequenceSimilarityOutput(loss=loss, logits0=logits0, logits1=logits1)
```

Listing 14: Sequence similarity task implemented on top of an Albert transformer network.

The similarity head only describes the additional structure of the neural network added on top of the base transformer model, but it does not describe the task itself. The implementation of the corresponding task can be seen in Listing 14 and is based on other implementations within the Transformers library. The fine-tuning tasks inherit from the class of the pre-trained model, in this case `AlbertPretrainedModel`, and follow the same interface as the base model. This means the only important function that needs to be implemented is the `forward` method, taking the relevant inputs and calculating the loss. The `forward` method expects a number of keyword arguments with the additional arguments passed by default hidden in the listing. In order to implement contrastive

learning, the forward pass has to accept two inputs, `input0` and `input1`, corresponding to a batch of function chunks. Additionally, for every input the attention mask is needed for the underlying model and since fine-tuning is supervised, the needed labels have to be provided as well. The inputs are passed to the base model one after the other, then passed through the similarity head and finally combined using the loss function. As training through cosine similarity is a common operation when performing contrastive learning, PyTorch provides an inbuilt embedding loss function. If this loss function was not present, a similar effect can be achieved by performing cosine similarity and using the result with mean squared error (MSE) loss, calculating the error with respect to the label. The loss is returned with a custom data class for model outputs and is the only attribute that has to be present, as otherwise the trainer could not train the model; the output follows the same sparsely documented rules as the keyword arguments for the forward pass. In addition to the loss variable, the results of the similarity head output, i.e. the embedding vectors, are returned in the output as well. Every additional attribute of the model output data class can be used within an evaluation function provided to the model trainer.

The evaluation function is called `compute_metrics` within Transformers and allows performing custom evaluation and metric computation in addition to the metrics calculated by the trainer itself. The metric computation is performed at certain steps of the training and the frequency of it can be specified. Ofci uses the function to evaluate how well the model performs on a part of the validation dataset, mainly to detect overfitting; as the default metric only logs loss on the training dataset, overfitting is not detected per default. In addition to logging the evaluation loss, Ofci also calculates the area under the curve (AUC) of the receiver operating characteristic (ROC) on the validation set, together with the numbers of how many functions were classified as similar and dissimilar. The ROC-AUC and other metrics necessary for the evaluation are being discussed in detail in section 6.1. The data returned from the `compute_metrics` function is conveniently logged by the Transformers trainer implementation and stored within the data directory of every checkpoint that is created during the training. The point in time of checkpoint creation together with its frequency is one of several configuration options provided by the default trainer of Transformers. Other notable configuration settings are the number of epochs, the training batch size, the evaluation batch size, gradient accumulation and support for 16-bit floating point operations.

The batch size specifies how many function chunks can be presented to the network at the same time; this does not only speed up training, but larger batch sizes are also connected with the learning rate of the training and small batch sizes can lead to faster overfitting; e.g. the fairseq developers provide specific recommendations for batch sizes and learning rates when pre-training RoBERTa[2]. When developing Ofci it became apparent that transformer models use up most memory on consumer graphics cards, and the small on-device batch sizes quickly led to overfitting when trying to fine-tune a model. The

[2] `https://github.com/pytorch/fairseq/blob/main/examples/roberta/README.pretraining.md` (Accessed: 2021-12-06)

solution for this is gradient accumulation, allowing the decoupling of the speed and the training quality benefits of batches. The configuration options allow specifying a number of steps for accumulating the gradient, before performing the backpropagation pass, allowing to "simulate" larger batch sizes even when the GPU can not fit the full batch in memory. During evaluation, gradient accumulation is not needed and not having to store the gradient information in GPU memory allows a slight increase in on-device batch size. Another useful feature natively provided by PyTorch is mixed precision training [119], which can be enabled using the *fp16* option. Mixed precision training allows training with half-precision floating point numbers, i.e. 16-bit floats, without losing accuracy compared to performing the whole training with single-precision floating point numbers. During the development of Ofci, training with fp16 has shown an approximate 2x speedup in training time, when training on consumer GPUs by Nvidia starting with the 2000 RTX generation. Older cards have fp16 support, but tests did not result in reduced training time. In general, during all Ofci experiments fp16 has never resulted in reduced memory consumption, which presented as another advantage of mixed precision training.

```
1  VOID StoreBBLRange(ADDRINT end_ptr, ADDRINT start_ptr) {
2      ADDRINT start = start_ptr - base_address;
3      ADDRINT end = end_ptr - base_address;
4      trace_file.write(reinterpret_cast<char *>(&start), sizeof(ADDRINT));
5      trace_file.write(reinterpret_cast<char *>(&end), sizeof(ADDRINT));
6  }
7
8  VOID Trace(TRACE trace, VOID* v) {
9      for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl)) {
10         // Only trace instructions of the current main binary
11         ADDRINT addr = BBL_Address(bbl);
12         if (addr >= IMG_HighAddress(main_binary) || addr < IMG_LowAddress(main_binary))
13             continue;
14
15         INS_InsertCall(
16             BBL_InsTail(bbl),
17             IPOINT_BEFORE,
18             AFUNPTR(StoreBBLRange),
19             IARG_INST_PTR,
20             IARG_ADDRINT,
21             addr,
22             IARG_END
23         );
24     }
25 }
```

Listing 15: Hooking and logging basic blocks within a simple pintool.

5.5 Trace Generation and Analysis

The Ofci trace generation is not handled by Ghidra, as Ghidra did not possess a functional tracing API fulfilling the previously mentioned criteria at the time of writing. Instead, tracing currently has to be done manually with the help of Intel Pin and a small Ofci pintool. Pintools are plugins for the main Pin engine and are compiled as shared library files within the Pin project structure. The Ofci pintool, *OFCITracer*, is provided as an out-of-tree buildable module, which can be built in a straightforward manner, given standard Linux build utilities are installed and the path to the Pin distribution folder is passed as `PIN_ROOT` variable to the makefile. The pintool performs a single task: Hook all basic blocks during execution and log their start and end addresses to a trace file; this is covered by the code snippet shown in Listing 15.

As a means to keep the trace file small and prevent library calls from being logged, the pintool identifies the main binary image on startup. When hooking the basic blocks it is then checked, whether the basic blocks in question are actually located within the main image. This comes with another benefit: By subtracting the image address from the basic block start and end address, the trace file is position independent per design. The binaries generated for the evaluation of the trace tooling are position independent and have a different base address when analyzed in Ghidra; the Ghidra plugin can load the trace file and simply add its own image base address to the basic block addresses. Within the trace file, the addresses are just stored with two unsigned 64-bit integers, as shown in the code snippet. The endianness is ignored and as the Ofci prototype only supports the x86 architecture, little endian is used on both the tracing and analysis architecture.

CHAPTER 6

Evaluation

After discussing the methods and implementations in the previous chapters, this chapter evaluates the performance of OFCI with respect to the research questions posed in section 1.3. To put the achieved results in perspective, OFCI is compared to the approaches it mainly builds upon, i.e. TREX [1] and SAFE [11], as well as recent approaches with the same goal, such as ASM2VEC [10]. Beyond the main research questions other implications that are rarely covered in existing work, e.g. practical applicability and training time, are discussed.

6.1 Metrics

In order to analyse the performance of OFCI and compare it to the state of the art, the used metrics have to be introduced. Work directly related to function clone detection mainly evaluates two things: The quality of the produced embeddings, if the approach uses embeddings, and the quality of the returned results when searching for a function clone in the known database. Quality of embeddings in the case of similarity based approaches, such as OFCI, can be determined by evaluating the cosine similarity of similar and dissimilar functions. The value produced by the cosine is in $[-1, +1]$, with -1 being definitely dissimilar and $+1$ being definitely similar. The in-between values are up for interpretation and for distinguishing similar and dissimilar function pairs, a threshold has to be introduced, e.g. 0. To make sure the similarity classification performance holds up under the threshold, related work relies on the *Receiver Operating Characteristic* (ROC) curve, which is a common measure in classification problems. For drawing the ROC curve, every encountered similarity score is added into a threshold list. This list is sorted in decreasing order and for every threshold in the list the true positive rate, as well as the false positive rate is calculated. The false positive rate is then used as x-coordinate and the true positive rate used as y-coordinate. To start the line at $(0, 0)$, an additional value higher than the maximum threshold is added; since there are no

positive predictions at this threshold, the true positive and false positive rate are both 0. In the upper right corner of a ROC curve plot, the threshold reaches the minimum and all predictions are classified as positives, putting the true positive and false positive rate at 1. When decreasing the threshold, a perfect classifier would not generate any false positives before all possible true positives have been generated, creating a curve that moves straight from the lower left corner to the upper left corner and then to the right. A random classifier would result in a straight line from the lower left corner to the upper right corner, as both positive rates steadily increase. The goal of similarity classifiers like Ofci is to have the ROC curve bending towards the corner of the ideal classifier. To compare the performance of different models more easily, the area under the ROC curve can be computed (ROC-Area Under Curve, or ROC-AUC). Following the description of perfect/random classifier, the perfect classifier would result in a ROC-AUC score of 1, while the random classifier results in a ROC-AUC score of 0.5. The AUC is not a perfect measure [120] and hides the actual distribution of true-positive/false-positive rates on the ROC curve, but Trex provides a large number of reference AUC scores with different approaches on their dataset and thus offers a decent reference point to compare against. Additionally, the curves reported by Safe and Gemini [15] are largely symmetrical around the median threshold, minimizing loss of information through the AUC score. While ROC and ROC-AUC are used to evaluate the quality of the produced embeddings with respect to similarity, a different metric is needed for the actual task of Ofci: Querying a database for function clones. As this is an information retrieval problem, the corresponding metrics also come from the field of information retrieval and are used as defined by Manning et al. [121]. After calculating the embedding of a function, it can be used as a query into the database to look for similar functions. The embedding lookup algorithm will then return at most k similar functions, ordered by decreasing cosine similarity score. This makes function clone detection a ranked retrieval problem and as such, the main metrics are *Precision/Recall at k documents* (P@k, R@k). For one function similarity query, the precision at k corresponds to the ratio of actually similar functions out of all returned functions k; recall at k corresponds to the ratio of similar functions in the top k query results out of all similar functions in the dataset. For ranked retrieval problems, recall eventually becomes meaningless as k gets larger, because with a sufficiently large k recall will eventually reach 1. To compare with Asm2Vec [10], P@1 is used within the evaluation. In this setting, P@1 assumes that there is exactly one similar function, i.e. the correct function has the same name and/or version, making the precision for one query either 0 or 1, and equal to R@1. The values reported by Asm2Vec are averaged above all queries, making the reported P@1 the ratio of all functions that have been identified correctly; unless otherwise mentioned, this definition is used for the evaluation when reporting P@1.

6.2 Experiment Setup

For evaluation of Ofci, the model is pre-trained and fine-tuned on the dataset provided by Trex, with additional data to accommodate analysis of virtualization-based

obfuscation. Extraction of data from the dataset, fine-tuning, and all other tasks unless specifically mentioned otherwise, are performed on a developer machine using Arch Linux with a 5.14 kernel, an AMD Ryzen 1800x, 32GB of RAM, and an NVIDIA RTX 2070 Super. This is a realistic mid-range setup for a reverse engineer regularly using VMs for malware analysis, as well as a mid-range consumer graphics card. For pre-training the model, a server running Ubuntu 18.04.5 LTS with an NVIDIA GTX Titan X, 94GB of RAM and an Intel Xeon X6580 CPU has been used. This machine has been used exclusively for pre-training of the model and despite offering almost twice as much graphics memory, it does not offer better training performance than the local developer machine; both machines run current NVIDIA graphics drivers and CUDA version 11, but the fp16 performance of the Titan X is worse than the performance of the 2070. The reason for choosing to run the pre-training on this server nonetheless was concern regarding the stability of the local developer machine, due to the pre-training requiring a time-span of several days.

For training ALBERT at the core of OFCI, the hyperparameters shown in Table 6.1 have been chosen. For parameters that are not shown in the table, the default values of the mentioned Transformers version are used. The batch sizes used for pre-training and fine-tuning are directly derived from the available graphics memory and the amount of gradient accumulation steps; the number of gradient accumulation steps was chosen in order for the overall batch size to be at least 512, thus resulting in "odd-looking" batch sizes when using the maximum available batch size in GPU memory. After pre-training and fine-tuning, the embeddings for all functions within the embedding dataset are generated and stored in a separate database for evaluation. The evaluation scripts work directly on this database and do not go through the process of extracting the data from the raw binaries through Ghidra again. As the validation dataset is quite large, data for the graphs relies on sampling from the database, with the sample size specified in the respective evaluation section.

hidden_size	768
intermediate_size	3072
num_attention_heads	12
max_position_embeddings	514
num_hidden_layers	8
vocab_size	868
pretrain_batch_size	520
finetune_batch_size	522
peak_learning_rate	0.00005
transformers_version	4.11.3

Table 6.1: OFCI Hyperparameters for ALBERT

6.3 Processing and Exploration of the Dataset

The base evaluation of Ofci is being done on the dataset of the Trex authors [1] , as the authors already compared a number of different approaches and they are the only ones who released their full, raw dataset. This dataset covers several architectures, but as Ofci is only implemented on amd64 for now, the other architectures are not taken into account for this evaluation. The uncompressed dataset for amd64 contains 1.5GB of ELF files and is split into two sets: One unobfuscated part, containing the same binaries for different compiler optimization levels and one part containing binaries which have been obfuscated using various methods. It is noteworthy that not the complete dataset has been used for benchmarking in [1], and there are additional obfuscations in the dataset; the reason for this seems to be that the additional obfuscations are not applied to every binary in the dataset. As the datasets for these obfuscations are too small and some of the obfuscations violate Ofci assumptions, e.g. not breaking up functions into multiple functions, these binaries are not considered for evaluation.

Project	O0	O1	O2	O3	BCF	CFF	IBR	SPL	SUB	Total
Binutils	57.527	43.901	42.166	39.096	62.981	62.981	62.981	62.981	62.981	497.595
Busybox	3.321	2.108	1.831	1.854	3.160	3.271	3.483	3.282	3.282	25.592
Coreutils	96.696	74.505	73.013	69.207	17.331	17.331	17.343	17.331	17.331	400.088
Curl	5.390	742	727	661	1.002	1.002	1.002	1.002	1.002	12.530
Diffutils	3.959	2.500	2.829	2.614	848	848	850	848	848	16.144
Findutils	5.055	2.671	3.625	3.286	1.400	1.400	1.404	1.400	1.400	21.641
GMP	789	698	690	665	782	782	782	782	782	6.752
ImageMagick	4.456	2.389	2.380	2.308	4.447	4.447	4.447	4.447	4.447	33.768
Libmicrohttpd	200	176	171	161	200	200	204	200	200	1.712
LibTomCrypt	794	749	743	726	794	794	795	794	794	6.983
OpenSSL	12.178	11.197	11.077	10.755	10.745	10.871	12.176	10.749	11.476	101.224
PuTTy	8.104	5.741	5.666	5.387	8.154	8.154	8.154	8.154	8.154	65.668
SQLite	2.192	1.525	1.367	1.181	2.183	2.183	2.183	2.183	2.183	17.180
Zlib	154	139	124	115	154	154	154	154	154	1.302

Table 6.2: Extracted functions from the Trex dataset.

The number of functions extracted through the Ghidra exporter can be seen in Table 6.2. These numbers show some discrepancies with the numbers reported by Trex and includes Busybox as a project, which is not reported by the Trex authors at all. Minor discrepancies can be explained by the way the functions are exported from Ghidra: Even if there is no explicit function symbol at a given location, Ghidra will still export a function with a generated name, i.e. `FUN_` followed by the address. In Ofci, these functions are not used during fine-tuning, as the generated function names do not allow for meaningful function pair comparison, but are still left in the pre-training dataset. In general, the numbers reported in Table 6.2 have not been pre-filtered in any way and Trex does not mention whether the numbers they reported had filters applied or which filters they applied. For several listed projects, e.g. Binutils and Coreutils, the numbers are off by an amount that cannot be explained by the Ghidra export strategy alone. The Trex dataset contains multiple versions of several projects, but the paper states only

certain versions that have been used for training the model. In comparison, Ofci uses all available versions in the dataset; when dividing the number of extracted functions by the number of versions present, the function coins more accurately depict the number of functions reported by Trex. The reported counts for functions in obfuscated binaries are in line with the Ofci requirements that prohibit functions from being split into new versions and with minor differences where Ghidra detects new functions, the function counts stay the same across obfuscations.

This dataset covers the basic set to compare it with existing approaches, without taking obfuscation into account. The additional methodology of Ofci for analyzing functions not only obfuscated by O-LLVM but through Tigress virtualization, requires an addition to the existing dataset. Since Tigress is not a drop-in replacement for other compilers, it takes considerable effort to create a dataset large and diverse enough for training machine learning classifiers. To this end, an approach similar to the one used by Syntia [81] and QSynth [89] has been used: A script generates functions from a simple grammar, calculating variable values through arithmetic and bitwise operators, based on some function input parameters. As this results in very short functions, the generation script for Ofci also adds variable assignments and conditional expressions, in an attempt to represent real functions more closely. The script makes sure that these conditionals are always taken when certain arguments are supplied, simplifying the dynamic analysis of the functions later on, as there exists only one program path and the tracer can follow this exact path. In total, 70.000 functions are generated this way and split across 70 source files, to make them manageable with Tigress. As with the Trex dataset, the source files are compiled on all optimization levels, but not with O-LLVM, due to Hikari not building on the target system. From the Tigress obfuscations, encode arithmetic (EA) and virtualization (VIRT) have been chosen and applied to all functions. For the final step, EA and VIRT have been combined, producing three sets of binaries obfuscated with Tigress. While the binaries obfuscated with EA are treated just like O-LLVM obfuscated binaries, the ones using virtualization have to be analyzed by the Ofci tracer, which is done manually. All generated binaries are then imported into the complete database, using the usual Ghidra exporter or the specific trace exporter. As the virtualization dataset overlaps the Trex dataset in the O0 to O3 categories, it is not included in the evaluations focussing on the comparison with existing approaches, as this would falsify results, due to additional function pairs being present.

Functions that have no corresponding similar functions cannot be used for fine-tuning and are therefore left out when generating the fine-tuning dataset. These functions are retained in the overall dataset, as they can still be used for pre-training. In addition, some form of basic filtering is imposed on the training dataset generation: For pre-training, only fragments of a function containing 60 tokens or more are selected, with 60 tokens being the minimum length of the smallest function in the virtualization dataset. This rule has been introduced to reduce the number of function fragments available for pre-training, as Trex has already shown that reducing the pre-training dataset does not have a large impact on performance later on. [1] Selecting fragments with 60 tokens or

more also makes sure that the pre-training is able to learn something from a fragment, as the masked language-modelling performs better if the fragments are not mostly empty. Even with the filters in place, the pre-training dataset contains still roughly 3M function fragments. To reduce training time, the dataset is shuffled randomly and 1.5M fragments are selected, 10.000 of which are used for validating pre-training performance after every epoch; the validation set for this step is small, as there is no risk of overfitting. For generating the fine-tuning dataset, the function pairing strategy and filters as described in section 5.3 are used. Out of all unique functions in the database, 30% have been randomly selected and used for function pair generation. From the fragment pairs of the selected function pairs, 300.000 have been randomly selected to reduce training time and of these, 100.000 are used for validation during the training. Therefore, out of the 30% training function selection, the training will not see all functions/all function fragments.

6.4 Feature Extraction and Training Performance

Category	Analysis	Export
O0	88	8
O1	95	6
O2	90	6
O3	103	7
BCF	102	6
CFF	36	7
IBR	410	5
SPL	124	7
SUB	57	6
EA	42	11
VIRT	248	17
VIRT-EA	271	17

Table 6.3: Processing times of the dataset within Ghidra, in minutes.

The largest part of the feature extraction pipeline in terms of time spent is the analysis and export of the binaries in Ghidra. While Ghidra offers comfort from a user standpoint, analysis is comparatively slow, with the analysis times for all different categories of the dataset being shown in Table 6.3. The listed times have been split in analysis time and export time, with the latter being the time Ofci took to extract the normalized disassembly. In a one-off setting, analysis and extraction can be performed as a single step, but for the purpose of developing the exporter, the dataset categories have been analyzed upfront and stored as Ghidra projects. The space required by the Ghidra projects is roughly 10x larger than the original binary dataset; this does not matter, as the Ghidra part of the pipeline should only be executed once, unless the exporter is changed, and the database of each analyzed file can be discarded after analysis/export. The listed duration measurements offer some insight into the expected analysis times on

different program categories within Ghidra. The difference in time between optimizations is not big and the duration tends to increase with more complex optimizations. When taking a look at obfuscations, it becomes clear that indirect branching and virtualization is the most time intensive w.r.t analysis in Ghidra, as Ghidra tries to resolve target locations of indirect jumps. Interesting outliers also exist on the opposite side of the spectrum: Control-flow flattening and instruction substitution result in shorter analysis durations, with control-flow flattening taking less than half the time of the binaries compiled with O0. Since CFF and SUB should be in theory harder to analyze than O0, this gives cause for suspicion. However, the function counts (as previously shown in Table 6.2) show that all functions have been extracted, the export logs of the OFCI plugin do not show abnormal behavior and the exported data in the database appears to be complete. The export duration measurements, which scale linearly with the amount of instructions exported, do not show stark differences in CFF and SUB either, leaving the cause of the shorter analysis times unknown. The timings have been replicated three times and the best guess is that Ghidra fails to identify a large number of cross-references due to these obfuscations; OFCI is indifferent to this information and only requires the disassembly. Compared to CFF and SUB, EA is not an outlier, as this obfuscation is only performed on the smaller virtualization dataset. Export times across the categories are expectedly similar, with the exception of the virtualization categories: The exporter does not parse functions here, but traces and the instruction traces are longer than functions.

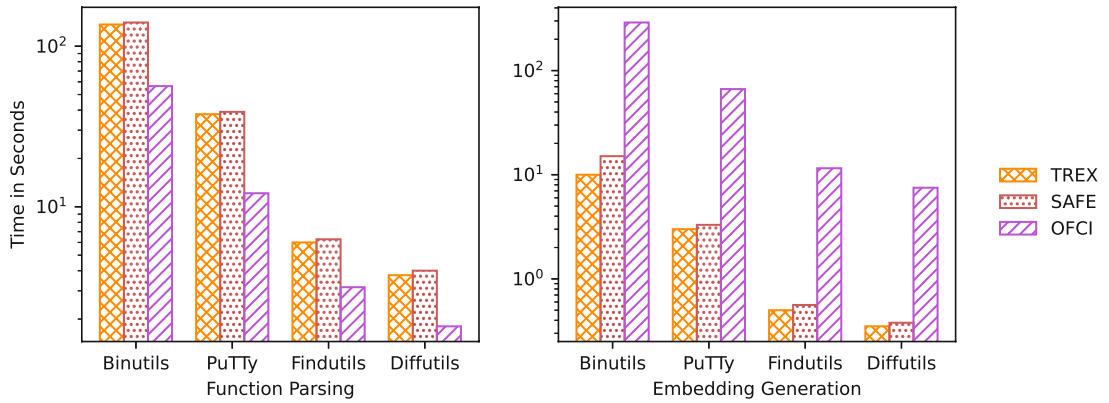

Figure 6.1: Function parsing and embedding generation performance.

After the normalized disassembly has been exported from Ghidra, a vocabulary has to be trained (in case it is not present) and the disassembly has to be tokenized. The process for vocabulary generation took 10 minutes and the tokenization took 50 minutes, both of which have been executed across the complete dataset. A large part of the tokenization process is pulling the disassembly from the database, serializing the generated tokens and storing them in the database again; this is currently handled in a naive way that performs a query for every function and could be optimized to form batches when reading from/writing to the database. After selecting which functions to use for fine-tuning, the

pre-training and fine-tuning datasets are generated, which happens by dumping the tokens from the database into HDF5 files. The pre-training on the remote server was run for exactly one week, during which it achieved 8 epochs on the pre-training dataset. Due to validation runs in between epochs, the total training time was hard to predict, as the base goal was to fit around 10 epochs of pre-training into a week of training. The fine-tuning was executed on the local developer machine and ran for 30 epochs within 20 hours.

To compare the performance of the framework with other approaches, the performance benchmarks published by Trex are used. The authors argue that only function parsing and embedding generation can be meaningfully compared, as training time can be neglected over time and embedding lookup depends on the embedding lookup approach being used. [1] When comparing the function parsing performance, Ofci is faster than both Safe and Trex; as Trex specifically mentions function parsing, the Ghidra analysis time is not taken into account. Exporting the disassembly and normalizing it in the Ofci Ghidra plugin takes 16s, 4s, 1.5s and 0.9s for Binutils, PuTTy, Findutils and Diffutils respectively. Additional 40s, 8s, 1.6s and 0.9s are required for tokenization; adding these measurements results in the function parsing performance shown in Figure 6.1. It is unclear how the Trex authors measured the function parsing performance, especially w.r.t. Safe. When trying to replicate the performance numbers on the published code of Safe the measured function parsing duration is a lot higher than reported (by the Trex authors). For example: Diffutils should be parsed in roughly 4s according to the figure, but actually takes 236s to parse when replicated. The difference is large enough to correspond to a different unit of time, i.e. the values are close enough to be interpreted as *minutes* instead of *seconds*, but sanity checking with the embedding generation measurements confirms that the plots are indeed correctly labelled with seconds. In the embedding generation, Safe is very fast compared to Ofci and Trex seemingly only requires a fraction of Ofci's embedding generation time. The main contributing factor for this stark difference is the measurement of embedding generation on the GPU: The reported values are measured on a system with 8 Nvidia RTX 2080-TI GPUs, while Ofci is evaluated on a machine with only one RTX 2070 Super. Even when taking the difference in hardware into account the differences appear odd, as measuring Safe embedding generation performance results in the same numbers as reported by Trex, but only on a single GPU. Without knowing the exact benchmarking setup of the Trex authors, it appears as if only Trex is taking advantage of the 8 GPUS, while Safe does not. As Trex does not have fully functional code and models published, the exact amount of time it takes to generate embeddings cannot be verified; since Trex is based on RoBERTa, while Ofci is based on the much smaller Albert, Ofci should outperform Trex in terms of embedding generation throughput.

6.5 Comparison of Model Size

One goal in the creation of Ofci is to decrease the size of the model when compared to other recent approaches, cutting down on model complexity. The model size in terms of

on-disk size and number of trainable model parameters is shown in Table 6.4, compared to the most recent and best performing approaches, TREX and SAFE. While the table shows OFCI in a good position, namely lowest model size on disk and lowest number of parameters in comparison to TREX and SAFE, it does not show the full picture and requires additional context. First, other approaches like ASM2VEC or GEMINI are not shown, as TREX and SAFE outperform them. GEMINI is significantly smaller, with about 10.000 trainable parameters, but does not work on instructions, but manually selected statistical features. [15] The amount of trainable parameters in ASM2VEC depends on how many functions it is currently trained with, making it hard to compare against the more recent approaches. In general, neither ASM2VEC nor GEMINI provide models trained on their full dataset and exact numbers regarding model size on disk and parameters in the case of ASM2VEC can therefore not be established. Another issue presented when comparing the size of the different models is the different ways of storing the trained model on disk. OFCI has 9M trainable parameters, multiplied by 4 (for 32-bit floating point numbers) results in 36MB of model data, meaning the parameters are stored almost entirely without meta-data or transformation, with the exception of compression. Performing the same calculation does not hold up for TREX, which is also using PyTorch to store its model, but reaches a size of almost 700MB, where it should only have 240MB according to the same calculation.

Approach	Size on Disk	Number of Parameters
TREX	696MB	60.606.229
SAFETORCH	210MB	55.043.500
OFCI	**35MB**	**9.136.000**

Table 6.4: Comparison of model size of different approaches.

The case of SAFE is more complex than shown in the table, as to allow a better comparison, SAFETORCH[1], a reimplementation of SAFE with the same parameters used for the paper, for PyTorch by the Facebook research team is used. The original SAFE implementation used Tensorflow instead of PyTorch, but the reason for the difficult comparison can be found elsewhere: In their original implementation, SAFE is split up into two parts, namely the instruction embedding and the function embedding part. The instruction embedding part is implemented using word2vec [26], training instruction embeddings based on the functions they appear in. The result is stored in a large matrix, 192MB compressed, 403MB uncompressed, and as this is not part of the SAFE Tensorflow model, it would also not be counted as a trainable parameter. The actual model, a Siamese architecture for contrastive learning using RNNs instead of the transformers used by OFCI, weighs in at a 211MB file and 4.5M trainable parameters. This would put SAFE at half the trainable parameters of OFCI, but would only consider half of its training process, as it still has to train the instruction embeddings before training the Siamese network. The use of SAFETORCH for comparison is valid, as the reimplementation keeps

[1] `https://github.com/facebookresearch/SAFEtorch` (Accessed: 2021-12-06)

the parameters of the actual Safe model the same, but PyTorch allows for easier integration of the instruction embedding into the overall training process. This provides one complete model, trained together with the embedding matrix, stored as a 210MB file on disk. This is surprisingly close to the model stored by the Tensorflow version of Safe and would make sense if this version also stored the embedding matrix together with the model; following the code this does not appear to be the case, making the similar disk sizes a coincidence. Integrating the instruction embedding training into the overall training is valid and does not break with the actual architecture of Safe: Just as in a separate training, the instruction embeddings can still be used on their own, extracted from the full model, and used for training the model on different data, without training the embeddings again. The same is the case for Trex and Ofci, as the underlying models also store embedding matrices, which could be reused or repurposed. All conditions considered, this does put Ofci as the leanest approach in comparison to Safe and Trex, at worst requiring only 17% of the disk space and trainable parameters.

6.6 Performance on Unobfuscated Data

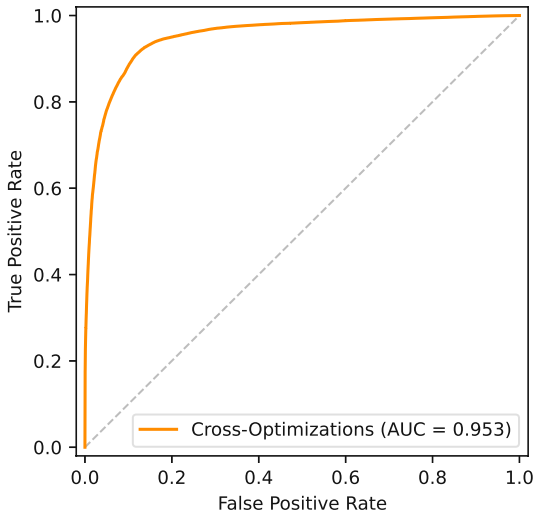

Figure 6.2: ROC of function pair similarity across all optimization levels.

While the focus of Ofci lies on obfuscated data, the first set of experiments is being performed on the unobfuscated part of the dataset and establishes baseline results for comparison with other approaches. In the unobfuscated dataset, the performance of Ofci's function clone detection capabilities is evaluated on functions with different optimization levels. When mentioning results of other approaches, these mirror the results reported by the authors of Trex or other related work, unless otherwise specified. This is due to the fact of other approaches not providing code/trained models and Trex having baseline results on their own dataset. The robustness of Ofci embeddings on function pairs sampled from different optimization levels can be seen through the ROC

curve in Figure 6.2. On its own, the curve shows good results for the embeddings generated by Ofci, with a high ROC-AUC of 0.95 and no obvious skewing towards true/false positive rates.

To put the ROC-AUC score into perspective, Table 6.5 compares the AUC scores against Trex. The authors of Trex provide ROC-AUC values for the respective projects in the dataset and for this comparison the cross-optimization scores have been selected. To calculate the ROC-AUC for one project, Ofci samples 1.000 similar and the same amount of dissimilar pairs from the database, with the standard deviation across 10 runs of sampling being smaller than 0.01 in all cases, while it is unclear how Trex selected/averaged the values, as it is not feasible to calculate the ROC of all possible function pairs in the dataset, depending on the project. The AUC scores of Ofci are worse, as the number of trainable parameters has been significantly reduced and it does not make use of the microtraces introduced by Trex. The numbers do not follow a specific pattern when comparing the two approaches, as the AUC scores reported by Trex are already very high; only the slight drop in performance of the Zlib project is seen in both approaches. Ofci performs the worst at LibTomCrypt, with an AUC score of 0.849. In this case, the drop in AUC is roughly 15%, while the average decrease is at 7%, which is minor when compared to the 85% reduction of model parameters in Ofci, when compared to Trex.

Project	Trex	Ofci
Binutils	0.993	0.933
Coreutils	0.992	0.968
Curl	0.993	0.929
Diffutils	0.992	0.951
Findutils	0.992	0.939
GMP	0.993	0.929
ImageMagick	0.993	0.924
Libmicrohttpd	0.994	0.901
LibTomCrypt	0.994	0.849
OpenSSL	0.992	0.952
PuTTY	0.995	0.938
SQLite	0.994	0.916
Zlib	0.991	0.892
Average	0.990	0.925

Table 6.5: Comparison of AUC scores across unobfuscated projects.

To analyze the performance of Ofci when comparing function pairs of different complexity in detail, the ROC curves of unoptimized functions and their optimized counterparts in different optimization levels are shown in Figure 6.3. When splitting up into different optimization levels, the ROC-AUC score of the function pair classification starts to decline with higher optimization levels. This is expected and provides some additional

insight into the dataset: The highest ROC-AUC score when comparing with unoptimized functions is produced by pairing with functions optimized at O1. When pairing with functions in O2, the ROC-AUC score drops slightly, highlighting how O2 produces code more different to O0 than O1. Finally, there is a larger drop when comparing with O3; this is expected as well, since O3 can introduce the most complex optimizations, significantly breaking up the code structure.

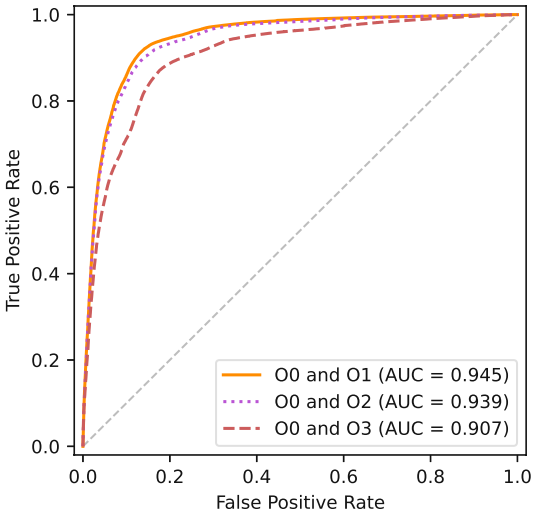

Figure 6.3: ROC of unoptimized functions and optimized counterparts.

Similar conclusions can be reached when interpreting the ROC curves in Figure 6.4. Instead of comparing every optimization level to the unoptimized base version, the function pairs are sampled from O1-O2 and O2-O3, showing how the performance of Ofci changes when incrementally increasing the optimization level. Starting from the levels O0-01, which is shown on the previous figure, this group achieves a ROC-AUC score of 0.945. The next group, O1-O2, achieves a ROC-AUC of 0.969, showing how the increase from O0 to O1 has a higher complexity than the increase from O1 to O2. When analyzing the ROC curve of O2-O3, the AUC can be seen decreasing to 0.95 again. However, while the AUC drops due to increased complexity of O2-O3 when compared with O0-O1, the ROC-AUC of O0-O1 is the lowest, marking this step as the highest change in function complexity.

As the ROC-AUC only shows the performance of classifying functions as similar, it does not show how well Ofci performs on function search. To this end, Table 6.6 shows the comparison of Precision@1 reported by Trex and Asm2Vec to Ofci. The reference values are taken directly from the Trex paper and contain the function lookup performance across different optimization levels. While the function similarity classification performance of Ofci is on par with other approaches, this is not the case when it comes to function search. When searching similar functions from binaries with optimization levels O2 and O3, the average performance of Ofci is well below Trex and Asm2Vec,

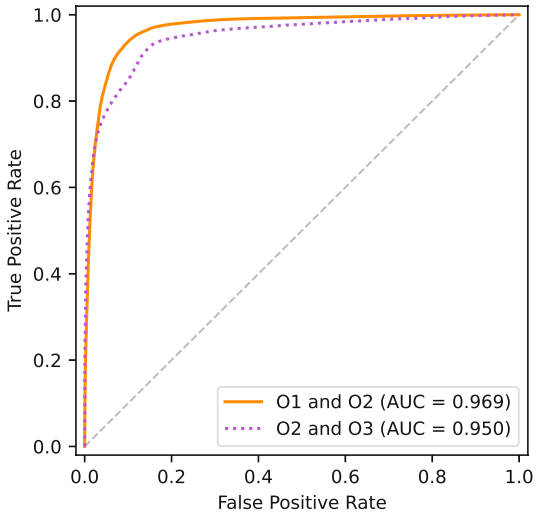

Figure 6.4: ROC of function pairs between different optimization levels.

at around 50%. This gap further widens when searching functions across O0 and O3, where differences in the binaries increase, only achieving a Precision@1 of 0.121. While Ofci does not perform well in function search, the precision values are still well above the performance of random embeddings. For example, the Precision@1 for Coreutils is at 0.025, but there are roughly 28k unique functions in the evaluation set, putting the chance of matching one function correctly at 0.0003 when accounting for 8 different versions of Coreutils; a Precision@1 of 0.025 implies that about 700 functions have been matched correctly, making it unlikely to be caused by random chance based on the probability of matching one function. Another observation is the missing correlation between performance drops when compared to Trex and Asm2Vec. While both of these approaches show decreased performance in the same projects, the Ofci performance drops do not occur in the same projects and appear to rather depend on the number of unique function names in the project evaluation sets.

A drop in performance is expected, as Ofci reduces model complexity and the number of trainable parameters, but the reduction margin should be closer to the results on ROC-AUC scores. Due to the long duration of a full model training cycle, evaluating all possible factors for the decreased function search performance is not in scope of this thesis and outside of a student budget for GPU hardware/hours of computation time. Therefore, the discussion of possible reasons is limited to what can be observed from the available results and surrounding decisions made in the implementation phase. The starting point for this discussion is the aforementioned **reduction of model complexity**. With the initial intention of building on Trex, a model too computation heavy to train on a student budget, the focus of Ofci was set to transformer-based models. Here, Ofci decided to work with Albert [99] in comparison to Trex, which made use of RoBERTa [70] at its core. Within the Albert paper, the authors devise certain

	O2 and O3			O0 and O3		
Project	Trex	Asm2Vec	Ofci	Trex	Asm2Vec	Ofci
Coreutils	0.955	0.929	0.137	0.913	0.781	0.025
Curl	0.961	0.951	0.680	0.894	0.850	0.159
GMP	0.974	0.973	0.748	0.886	0.763	0.219
ImageMagick	0.971	0.971	0.457	0.891	0.837	0.066
LibTomCrypt	0.991	0.991	0.611	0.923	0.921	0.040
OpenSSL	0.982	0.931	0.469	0.914	0.792	0.082
PuTTy	0.956	0.891	0.248	0.926	0.788	0.049
SQLite	0.931	0.926	0.551	0.911	0.776	0.117
Zlib	0.890	0.885	0.465	0.902	0.722	0.329
Average	0.957	0.939	0.485	0.907	0.803	0.121

Table 6.6: Comparison of Precision@1 across optimizations.

measures to reduce the model complexity (discussed in section 4.4), and can achieve on-par performance with Bert [23] on several widely used natural language processing benchmarks, including semantic textual similarity (STS). RoBERTa performs better than simple Bert and thus better than Albert in theory. The performance differences discussed in these papers however are in line with the ROC-AUC score differences discussed here, and not with the drastic differences in precision scores. In fact, Albert does appear to even outperform smaller variants of RoBERTa in textual similarity benchmarks.[2] Based on this data, it does not appear that the bad precision scores of Ofci are caused by this choice of neural network architecture and the reduced complexity. In order to rule out this possibility definitely, a reimplementation of Trex with the full dataset and same methodology is required; besides requiring an extensive amount of time, not all relevant code for generating the data from scratch is published by the authors.

This ties in with the **reporting of metrics** and **description of evaluation sets** in related work. While there are a variety of popular benchmarks for natural language processing, these benchmarks do not exist for function clone detection and most fields of binary analysis in general, making it hard to objectively compare different approaches on a certain task or validate claims. This is highlighted by the authors of Trex in a striking manner: For every approach they compare against, i.e. Asm2Vec [10], Safe [11], Gemini [15] and Blex [57], they compare results with different metrics, because with the exception of Safe, none of the other approaches provides the full source-code or trained model. Additionally, the datasets are also not exactly the same, ruling out a fair and objective comparison. Another issue in comparing the metrics is omitting the details on how exactly these metrics are calculated; this does not relate to the established definition of the metrics, but on how the data for the metrics is selected. For example, due to the large number of possible function pairs, it is not possible to calculate the ROC curve

[2] https://paperswithcode.com/sota/semantic-textual-similarity-on-mrpc (Accessed: 2021-12-04)

for all possible pairs. To solve this, the function pairs have to be sampled from the base dataset, which opens up several possibilities for influencing the outcome: The dataset can be partitioned in favor of the benchmarks, the number of sampled function pairs can be varied and a different ratio of similar/dissimilar function pairs can be selected in order to steer the results into a better direction. This information could readily be omitted from a paper, if the accompanying evaluation code was released as well, since this would clear up any missing details. Unfortunately, most papers do not release the code for their approaches and even when releasing all of their relevant implementation code, like SAFE did, they might not release their evaluation code. Large, publicly available and well designed benchmarks could alleviate this problem; as chapter 3 shows, binary code similarity is no longer a niche problem and agreed upon benchmarks are long overdue.

Finally, the most important factor in the subpar function search performance of OFCI is the **definition of what constitutes a similar function**. Per the definition in section 2.1, two functions are similar when they exhibit the same semantics. However, the dataset and the corresponding labels are not structured this way: In the context of related work, a function is similar if they have the same name within the same project, as the ultimate goal is to identify an existing function and assign a name. When compared to TREX, OFCI made the decision to not only treat functions with the same name in a project as similar, but also if they are from different versions of the same project. This is possible, as the TREX dataset provides different versions for some projects, e.g. 8 different versions of Coreutils, of which all functions with the same name are considered similar. While this does not account for all of the missing function search performance in OFCI, it does make the results different to the ones reported by TREX. Not only are there more overall functions per project, but now the different versions are treated as similar during training, when it is also possible that functions have changed in large parts. Combined with the issues of the code virtualization similarity approach in section 6.10 (which has been trained together with the rest in one fine-tuning session) and the small amount of training data selected to keep training times feasible, this leads to possible issues with the composition of the training set. One possibility to counteract this effect is to reduce the scope of the training to one project: A reverse engineer will normally have a good guess about which libraries are being used in a program and just has difficulties identifying the specific functions; they could therefore select a model that has been fine-tuned on this specific project. The fine-tuning process would be faster, as less training data needs to be processed, making it possible to have multiple fine-tuning training processes running. Alternatively, functions could be grouped into overall functionality categories, e.g. cryptographic functions, which is what SAFE implements on top of its embeddings. [11]

Following the definition of similar functions, another issue that is present in all approaches is the unexpected similarity of functions with different names. When manually inspecting retrieval results at a certain rank k for OFCI, two distinct variants of negative results (i.e. the correct function is not at rank 1) appear. In the first case, functions with different names perform exactly the same or very similar operations and because

they are so similar, they push the correct function out of rank 1. If the functions are exactly the same, this can be detected by hashing, but if some addresses are different or a handful of instructions are added, this is not easy to detect. In the second case, the correct function does not appear within rank 10-20; when comparing the disassembled original function, the rank 1 function and the correct function, the visual representation (disassembly, CFG) of the rank 1 function did indeed appear more similar to the original function. As Trex has not published precision values for their models without microtraces, it is unclear whether adding more semantic information in the form of microtraces helps reducing the impact of this issue and there are still going to be cases where, for example, the optimized versions of two functions are similar, reducing the problem to the first case. The first case is interesting, as it can be somewhat seen as a chicken-and-egg problem: In order to create a good dataset, function similarity of the dataset needs to be known beforehand. While natural language processing can use humans to annotate the similarity of two text snippets, this is difficult to apply to reverse engineering, where assessing the similarity can take a significant amount of time, even for obvious function clones. Related work does not mention whether they sanitized their dataset w.r.t. unexpected similarity; Asm2Vec mentions the edit distance between function pairs, but does not go into detail whether this affected the labeling of the dataset. [10] This remains an open issue that could potentially be solved by applying classical similarity approaches to find obvious similar functions.

6.7 Performance on O-LLVM Obfuscated Binaries

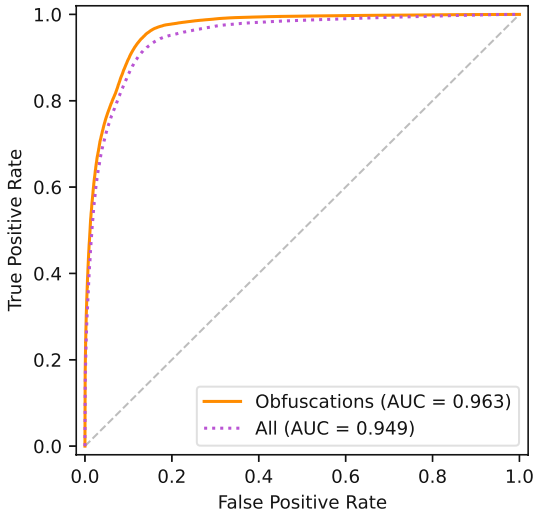

Figure 6.5: ROC of obfuscated function pairs and all function pairs.

To start off the evaluation of Ofci on obfuscated binaries, Figure 6.5 shows the ROC curves of classifying similar function pairs between obfuscations and between all functions in the dataset, including the different optimization levels. The AUC score on the

obfuscated function pairs is 0.963, which puts the classification performance above the previously discussed classifying of function pairs across optimization levels, which resulted in an AUC of 0.953. In general, this means that OFCI performs slightly better on the obfuscation part of the dataset, due to two reasons: The obfuscations provided by O-LLVM appear to be less drastic than difference between O0 and O3, and the AUC score is calculated between obfuscated function pairs only, meaning there are less differences between the different obfuscations itself.

	Obfuscated Pairs		All Pairs			
Project	TREX	OFCI	TREX	(w/o microt.)	SAFE	OFCI
Binutils	**0.991**	0.947	**0.952**	0.871	0.918	0.942
Coreutils	**0.991**	0.966	0.951	0.900	0.910	**0.965**
Curl	**0.991**	0.940	**0.953**	0.919	0.931	0.924
Diffutils	**0.990**	0.959	**0.952**	0.931	0.918	0.945
Findutils	**0.990**	0.967	**0.961**	0.889	0.910	0.934
GMP	**0.990**	0.869	**0.953**	0.931	0.930	0.852
ImageMagick	**0.989**	0.910	**0.962**	0.889	0.935	0.902
Libmicrohttpd	**0.991**	0.905	**0.951**	0.910	0.917	0.874
LibTomCrypt	**0.991**	0.836	**0.953**	0.900	0.911	0.844
OpenSSL	**0.989**	0.946	**0.952**	0.858	0.925	0.941
PuTTy	**0.990**	0.976	0.941	0.840	0.900	**0.951**
SQLite	**0.993**	0.956	**0.953**	0.850	0.929	0.947
Zlib	**0.990**	0.911	**0.960**	0.810	0.931	0.888
Average	**0.990**	0.929	**0.953**	0.884	0.920	0.916

Table 6.7: Comparison of AUC scores across obfuscated projects.

The performance on the obfuscated data when compared to other approaches can be seen in Table 6.7. The comparison is split into two categories, with the first category being the ROC-AUC across obfuscated function pairs only. OFCI is here directly compared to TREX, as the results come from the same dataset; the ROC-AUC calculation for OFCI is again sampling 1.000 similar and dissimilar function pairs, and calculating the mean AUC across 10 runs of sampling, while the TREX and SAFE results are taken from the TREX paper or have been provided by the authors on request. When comparing the AUC scores of obfuscated pairs, OFCI is expectedly outperformed by TREX and again shows a maximum drop of 15% in AUC score, which is similar to the unobfuscated results presented in the previous section. This is again an acceptable tradeoff for an 85% reduction in the number of trainable model parameters. The second category, the ROC-AUC of all function pairs in the dataset, is taken from the ablation study performed by TREX. It includes the AUC scores calculated for SAFE on the TREX dataset and the TREX results when the model is not pre-trained with microtraces, but only with static data, similar to OFCI. This comparison is not entirely objective, as these numbers also include cross-architecture function pairs that are not present in OFCI;

however, the comparison is presented here to contextualize the results produced by Ofci. Keeping this in mind, Ofci manages to achieve a higher AUC score than Trex on two projects, manages to generally outperform the version of Trex that does not make use of microtraces and produces results in the same range as Safe. Even keeping the difference in dataset composition in mind, Ofci can therefore achieve good results when only taking the ROC-AUC into account.

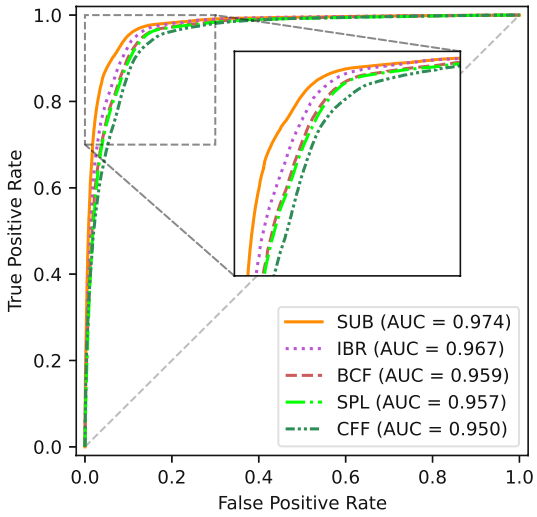

Figure 6.6: ROC of unobfuscated functions and obfuscated counterparts.

To analyze the performance of Ofci on the different obfuscation methods, Figure 6.6 shows the ROC curves when sampling function pairs from O0 and the obfuscations, with the obfuscations sorted by decreasing AUC score. While CFF, one of the most complex obfuscations offered by O-LLVM, shows the worst AUC score as expected, the similarly potent IBR obfuscation shows the second highest ROC-AUC. The SUB obfuscation performs best, as it only adds minor changes by substituting instructions and constants, while the methods adding to the control-flow, BCF and SPL, end up in between IBR and CFF. Inspecting the different obfuscation with Ghidra explains the reason for the bad performance of CFF and the comparatively good performance of IBR: The IBR obfuscation does not flatten the control-flow like CFF does, but only adds indirect branches where there already have been direct branches in the code. Due to Ghidra's indirect branch analysis, it still detects the targets behind the indirect jumps and since the code is not reordered, looking at it from the disassembly view does not produce extensive changes. Analyzing the same function in a binary obfuscated with CFF, not only existing control-flow is replaced, but new basic blocks are added, and what remains of existing functionality does not appear in order when looking at the disassembly. Analyzing the function search Precision@1 results in the values presented in Table 6.8, again compared to the reported values of Trex and Asm2Vec. The use case of function search for an obfuscated function is the determination of the original unobfuscated function. To

this end, the embeddings of the unobfuscated functions are stored in an index, and the index is queried with the obfuscated functions. This is done for the BCF, CFF and SUB obfuscations, as these are the ones listed by Asm2Vec and Trex. The original table reported by Asm2Vec [10] contains another row for combining the obfuscations, but the Trex dataset only contains the separate obfuscations and Ofci can therefore not report on this category. The results are slightly worse than the numbers shown for Ofci in the previous section when searching functions across O0 and O3, highlighting that O-LLVM obfuscations are still able to increase the complexity of the function beyond the typical capabilities of standard compiler optimizations. The Precision@1 values are also in line with the ROC-AUC scores in Figure 6.6: Instruction substitution is among the simpler obfuscations with an average Precision@1 of 0.229, while CFF and BCF are on the more complicated end with 0.136 and 0.149 average Precision@1 respectively. Similar to the precision values of the unobfuscated function search, the results are however far behind the reported values of Trex and Asm2Vec; the argumentation for these results is the same as for the unobfuscated version and is not repeated at this point (c.f. section 6.6).

Obf.	Approach	GMP	LibTomCrypt	ImageMagick	OpenSSL	Average
bcf	Trex	0.926	0.938	0.934	0.898	0.924
	Asm2Vec	0.802	0.920	0.933	0.883	0.885
	Ofci	0.158	0.121	0.224	0.093	0.149
cff	Trex	0.943	0.931	0.936	0.940	0.930
	Asm2Vec	0.772	0.920	0.890	0.795	0.844
	Ofci	0.169	0.178	0.156	0.043	0.136
sub	Trex	0.949	0.962	0.981	0.980	0.968
	Asm2Vec	0.940	0.960	0.981	0.961	0.961
	Ofci	0.249	0.214	0.283	0.169	0.229

Table 6.8: Comparison of Precision@1 scores across obfuscated projects.

6.8 Performance of Fragmented Functions

While recent approaches [11, 1] limit the input size of their solutions, or have to limit the input size due to limitations of the underlying model architecture, there is no comprehensive evaluation of the effects on larger functions. This does not matter as much in natural language processing, as sentences and paragraphs will more easily fit within a 512 token boundary, or 4096 when taking recent large models into account. [122, 123] These models require a large amount of memory and GPU resources and do not solve the underlying problem in function clone detection: The length of a function can range anywhere from 10 to thousands of tokens, making it hard to fit functions into a fixed-size window for classification. The ROC curve in Figure 6.7 shows the performance of Ofci on unobfuscated functions, selected over a different number of fragments per function pair; in this case, the function pairs have been sampled from functions having the same amount of fragments. On unobfuscated functions, Ofci performs best on functions

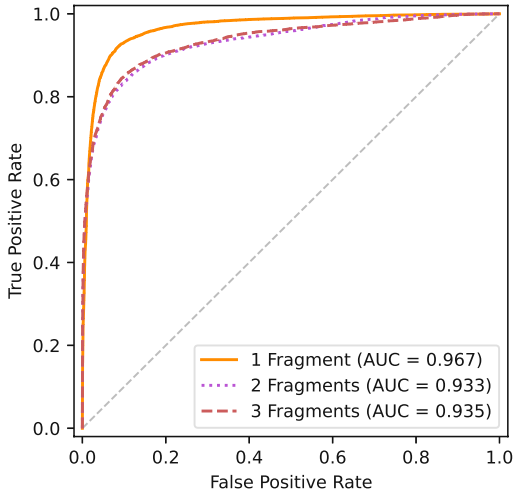

Figure 6.7: ROC of unobfuscated function pairs categorized by fragment count.

that are not longer than one fragment. There is a noticeable drop in the ROC-AUC of function pairs with two or three fragments, showing how the embeddings become less robust as soon as functions become longer. It can also be seen that the quality of the embeddings does not further decline after the initial drop, as function pairs that are three fragments long perform slightly better according to the ROC-AUC. In general, the ROC-AUC does not further decline after more than two fragments, oscillating roughly around the 0.93 mark, some of this being due to the fact that the pool to sample function pairs from becomes smaller with growing length and the ROC becoming less precise.

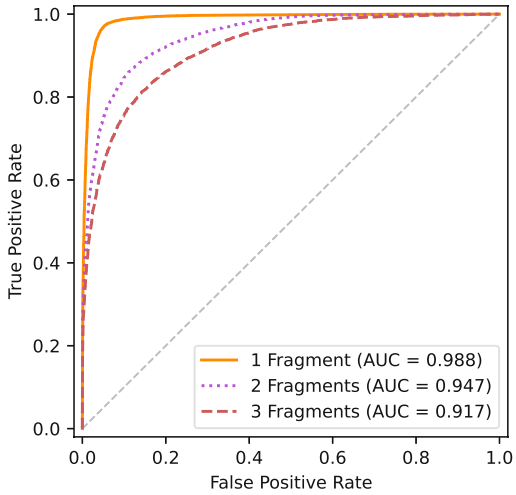

Figure 6.8: ROC of obfuscated function pairs categorized by fragment count.

Taking the same viewpoint on the set of obfuscated function pairs reveals a similar picture

in Figure 6.8. Again, function pairs that are no longer than one fragment perform best, with a ROC-AUC 0.98 that is even higher than it is for unobfuscated function pairs. However, for obfuscated functions, the ROC-AUC continues to drop, even after having function pairs with more than two fragments, before it eventually starts to converge against 0.91. The reason for this is not clear from the dataset; one contributing factor is the fact that the performance for functions with one fragment is already higher than the corresponding ROC-AUC for unobfuscated functions, spreading the performance drops over a wider range.

Project	1 Frag.	2 Frag.	3 Frag.
Binutils	0.102	0.009	0.006
Coreutils	0.032	0.017	0.019
Curl	0.305	0.013	0.000
Diffutils	0.380	0.071	0.113
Findutils	0.296	0.087	0.079
GMP	0.500	0.150	0.032
ImageMagick	0.269	0.072	0.076
Libmicrohttpd	0.800	0.167	0.136
LibTomCrypt	0.237	0.114	0.100
OpenSSL	0.055	0.016	0.053
PuTTY	0.111	0.035	0.045
SQLite	0.314	0.011	0.052
Zlib	0.750	0.111	0.160
Average	0.319	0.067	0.067

Table 6.9: Comparison of Precision@1 of fragmented functions.

The difference in AUC scores also translates to Precision@1 measurements across all projects. The measurements have been taken through function search of the original function from binaries that have been obfuscated with CFF, in order to highlight the impact on obfuscated function search. With two exceptions, Coreutils and OpenSSL, the Precision@1 drops if the function is longer than one fragment, and in some cases continues to drop when looking at functions with three fragments. Limiting the search to all functions that are not longer than one fragment also reveals better performance on obfuscated function search than the results of the previous section have shown. This implies that long functions drag the overall Precision@1 score down.

To summarize, the performance of Ofci is still acceptable for functions that cannot be represented within 512 tokens, showing the robustness of the generated embedding vectors. However, as expected, there is a noticeable drop when looking at embeddings consisting of two or more fragments, showing room for improvement when dealing with longer functions. Unfortunately, research into this direction, e.g. the transformer models in [122, 123], tries to push the boundaries by increasing the input size of transformer models and despite successful reduction of complexity, the memory consumed by these

models is still overwhelming for consumer graphics cards. Without a breakthrough w.r.t. transformer models, strategies for longer inputs have to shift towards redesigning the training process: Similar to gradient accumulation for achieving certain batch sizes, accumulating results specific for the contrastive learning task across fragments could improve results at the cost of a much more complicated and less efficient training task.

6.9 Ablation Study

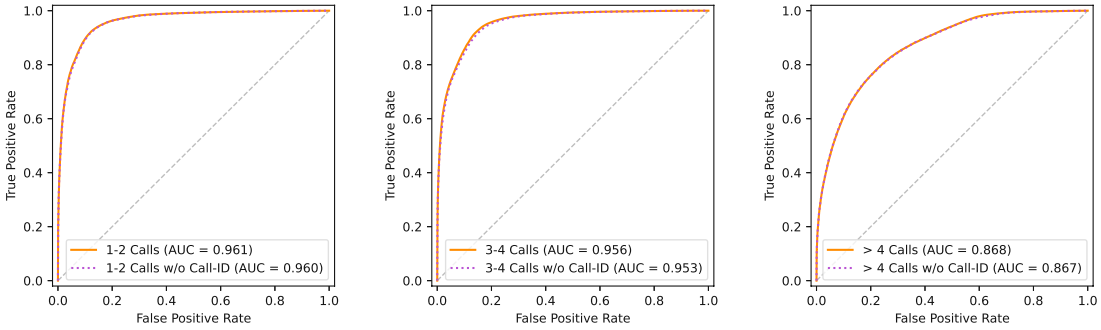

Figure 6.9: ROC of all function pairs restricted to a different number of function calls.

One feature Ofci adds on top of the machine learning model is the *Call-ID*, where references to another call are added to the call instruction, with the intention of improving the results of the embedding. Within this ablation experiment, the performance of Ofci with and without the Call-ID feature is tested, in order to see whether the call reference offer an improvement of the embeddings and identification results. The first experiment can be seen in Figure 6.9, showing the ROC curves of function pairs selected from the full dataset, grouped by the number of calls these functions execute. Functions that do not perform any calls are excluded, as the Call-ID feature will not have any impact on them. Furthermore, as the set of available function pairs decreases with a growing number of function calls, there is no further distinction among functions that perform more than 4 function calls. To build the necessary dataset for comparing the results of function pairs with and without Call-IDs, the dataset is exported from Ghidra twice, one export containing the full Call-IDs to be tokenized and the other export having all Call-IDs set to 0, treating it as if all calls are made in reference to unknown functions. Embeddings are generated for both of these datasets, each function being indexed by an ID that is the same across datasets; when sampling function pairs, the functions are selected based on their attributes and a function pair consists of two function IDs. When looking up the embeddings of the functions in the pair, the same ID can be used to retrieve the embedding from either the dataset with or without Call-IDs. The ROC curves in Figure 6.9 show no visible difference, the robustness of the embeddings appears to be just as good as without the Call-ID feature. However, there is a very small difference in the ROC-AUC score of function pairs in favor of having the Call-ID feature. For the groups of 1-2 calls and the functions with more than 4 calls, this difference is as low as 0.001

difference in AUC, and does not become bigger the more calls a function has, which does not appear to be intuitive at first.

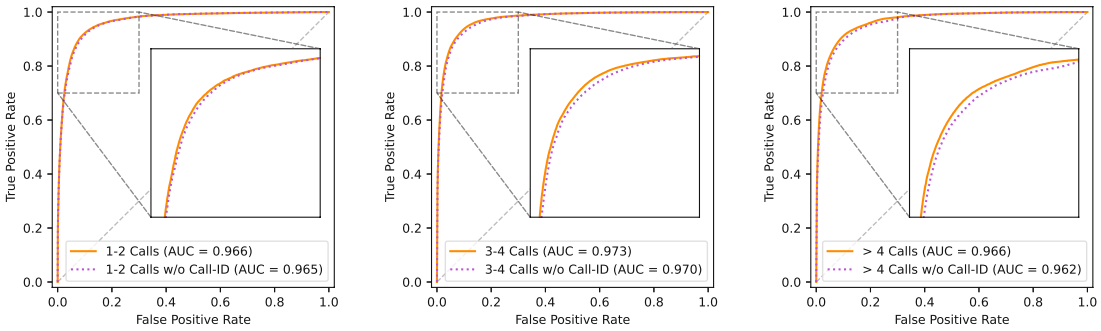

Figure 6.10: ROC of all function pairs by call count and corrected for fragmentation.

One issue here is that the functions sampled from are inspected without regard for length and the more calls a function has, the more tokens it will usually contain. This effect also manifests in the decline of the general ROC-AUC score across the three curves, where the AUC is generally lower if function pairs with higher call counts are selected. To compensate for this, Figure 6.11 shows the ROC curve corrected w.r.t. the length of a function, by only picking functions that fit within one fragment. The results are similar to the uncorrected curves, showing that the Call-ID feature does improve the robustness of the embeddings in all cases, albeit by a small margin. In comparison to the uncorrected ROC curves, these results do show an improvement if a larger number of function calls is involved, going from a ROC-AUC difference of 0.001, to 0.003 for function pairs with 3-4 function calls, and 0.004 for function pairs with more than 4 calls.

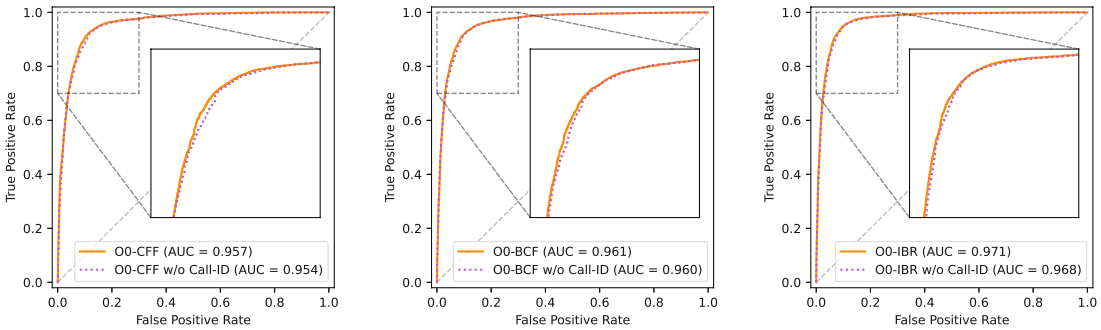

Figure 6.11: ROC of various obfuscated function pairs with and without Call-ID.

As the previous results are sampled from the full dataset and Call-ID is intended to improve the performance on obfuscated function clones, Figure 6.11 shows the performance of Call-ID on three selected obfuscations, with CFF and IBR being the most complex obfuscations provided by O-LLVM, and BCF being one of the less invasive operations.

In order to produce a real-world comparison, where the an obfuscated binary is under analysis and the unobfuscated version is nown, the function pairs are sampled from the specific obfuscation method and the corresponding function in O0. The results show an improvement in embedding robustness across the listed obfuscations, the range being of similar margin as the previous results. The curves also show expected behavior w.r.t. the complexity of the applied obfuscations: CFF and IBR, the most complex out of the O-LLVM obfuscations, benefit more from Call-ID with a difference in ROC-AUC of 0.003, while a simple obfuscation like BCF is hardly affected by Call-ID at all. Virtualized examples are not considered here, as Ofci does not perform analysis on virtualized code that is also interprocedural. While the differences in the ROC-AUC scores are very small, the effect of Call-ID can be observed in the Precision@1 scores as well, shown in Table 6.10 for all projects when searching O0 from CFF. The dataset is the same subset used for the ROC curve O0-CFF in Figure 6.11. While the differences are again small, they are more distinct than the difference in ROC-AUC scores and with the exception of Curl/OpenSSL they are showing that Call-ID can indeed improve the function clone search.

Project	Call-ID	w/o Call-ID
Binutils	0.131	0.116
Coreutils	0.059	0.057
Curl	0.462	0.500
Diffutils	0.471	0.462
Findutils	0.360	0.326
GMP	0.857	0.714
ImageMagick	0.377	0.333
Libmicrohttpd	0.857	0.714
LibTomCrypt	0.296	0.296
OpenSSL	0.049	0.052
PuTTY	0.108	0.100
SQLite	0.372	0.283
Zlib	0.857	0.857
Average	0.404	0.370

Table 6.10: Comparison of Precision@1 with and without Call-ID.

To summarize, the ablation testing shows how the Call-ID feature does appear to be beneficial, with the shortcoming of the small margin. A difference in the third decimal place of the ROC-AUC is not a significant improvement, but this difference is consistently positive across all tests. In its current form a tradeoff has to be made: Using Call-ID is cheap, but not free when exporting disassembly from Ghidra. As the classification of other functions might change when new functions are identified, some functions have to be put through embedding generation again. Processing functions sorted by the number of calls minimizes this effect, but the problem still persists; as the benefit of using Call-ID is that small, repeated unnecessary embedding generations might be per-

formed. For future work, a strategy on how to increase the benefit of Call-ID is needed. With the current implementation, the Call-IDs are added as tokens into the normalized disassembly, in hopes of the transformer models recognizing these tokens as important. Through ablation it becomes clear that the significance of the Call-ID tokens is not increased in relation to other tokens, making it necessary for future work to more tightly integrate this with a custom model, where Call-IDs can be handled separately. This can happen through extending a transformer model, or adding a separate model in the likes of Gemini [15], which generates embeddings based on manually selected features.

6.10 Performance on Tigress Virtualized Examples

As the second major new addition on top of existing work, Ofci tries to approach analysis of code obfuscated through virtualization. At the time of writing there are only two references to virtualized code in related work regarding function clone detection: A survey [13] citing virtualized code as an unsolved problem, and Asm2Vec [10] performing analysis on the static code generated by Tigress. The Asm2Vec authors showed that they were still able to detect vulnerabilities from the static code generated by Tigress virtualization, with a true positive rate of 35.8% on a small dataset, but do not discuss function clone detection across virtualization. To the best of my knowledge, Ofci is the first approach trying to tackle function clone detection across virtualization through dynamic analysis, i.e. through generating instruction traces of the virtualized code and comparing with these traces; the results can therefore not directly be compared with existing approaches.

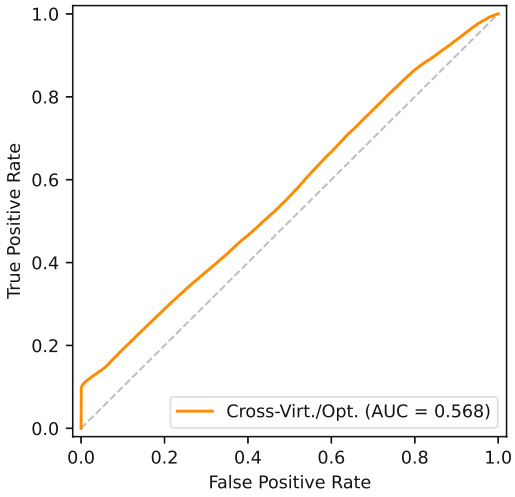

Figure 6.12: ROC of all function pairs across optimizations and virtualization.

The dataset of the experiments on virtualized code is distinct from the dataset used in the previous evaluations, however the model has always been trained with both to be as general as possible. The main difference between the two datasets is the source of ground

truth: While the previous results have been shown on real-world applications, the Tigress dataset has been synthetically crafted by following a few grammar rules. Tigress makes it hard to create large-scale datasets on real applications, as it requires the program to be easily combined into a single C file, does source-to-source transformations and outputs the compiled binary itself. As mentioned in section 6.2, the dataset contains the binaries across different optimization levels, EA, VIRT and VIRT-EA. The ROC curve for function pair embeddings across this whole dataset can be seen in Figure 6.12. The results show that the classification of function pairs across virtualized code is not good, but also not completely random, with the random classifier being represented by the line in the middle of the plot. There is an initial climb up to a certain threshold, meaning no false positives have been detected below this threshold, but also not all positives. In the second experiment, only the robustness of function pairs across virtualizations has been measured, with the corresponding ROC curve shown in Figure 6.13.

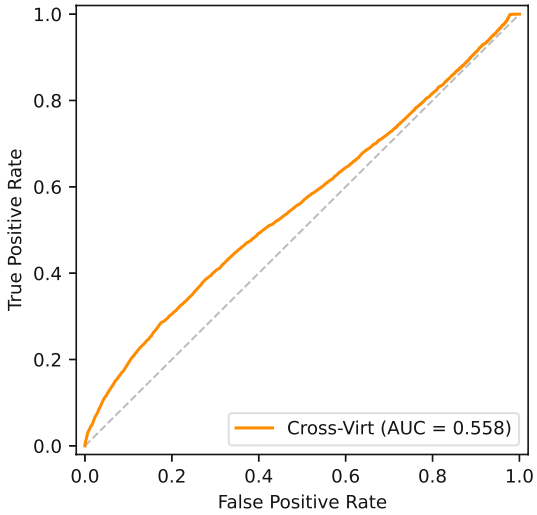

Figure 6.13: ROC of all function pairs across virtualization.

The curve is similarly close to the curve across all function pairs in the dataset, but missing the initial climb, leaning more towards the random classifier as the thresholds become larger. Otherwise, it is showing a more curve-like behavior, hinting that the true positive/false positive rates are leaning more slightly to what is expected. Precision@1 scores are omitted here, as they show results similarly close to the performance of a random classifier. Only in the case of searching virtualized functions against obfuscated or other virtualized functions does Ofci produce results that are slightly above random performance. Therefore, a possible solution to produce better results would be to obfuscate known functions with the same virtualization technique and only then compare them with unknown virtualized function traces. The generally bad ROC-AUC scores lead to the question, why the model is working well on the non-virtualized code, but producing embeddings that can hardly be classified correctly for virtualized code. One

reason for this is the length of the virtualized code: As shown previously, the robustness of the generated embeddings drops when functions larger than one fragment are compared and tends to drop as the number of fragments increases for obfuscated code. When comparing the traces of virtualized code with normal functions, the length of the traces is several magnitudes higher than the average length of normal functions; the traces in the dataset are long enough to make up 1/3 of the fragments in the pre-training dataset. While the traces are therefore a special case and the evaluation of fragmented functions has shown that there is a drop in performance, it has also shown that the drop is not this big and can therefore not fully explain the near-random classification performance of virtualized functions.

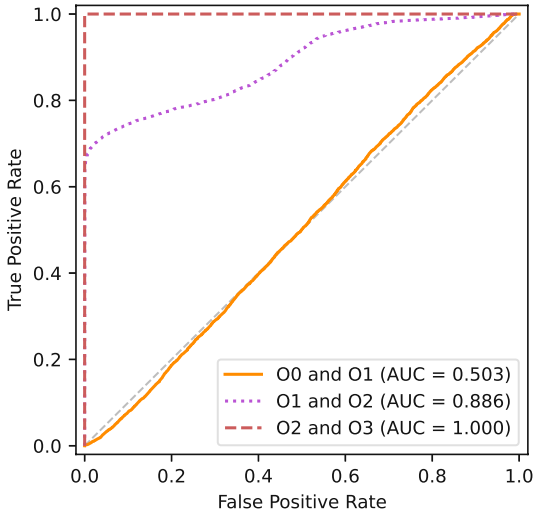

Figure 6.14: ROC of unobfuscated functions in the virtualization dataset.

A more likely culprit is the composition of the OFCI dataset itself. While the approach of generating functions according to a simple grammar works well when trying to retrieve an exact result through deobfuscation [89, 81], it comes with issues when trying to apply this to a machine learning algorithm that works through pattern recognition. The ROC curve in Figure 6.14 highlights some issues w.r.t to missing code complexity and composition of the synthetic dataset. Embeddings generated for function pairs in the O0 and O1 categories correspond to a random classifier. However, there is a significant jump in classification performance from O1 to O2, and when sampling function pairs from O2 and O3, the ROC curve corresponds to the other extreme end, i.e. the perfect classifier. While this seems nonsensical at first, it gives some insight into the complexity of the dataset: The functions are considered very similar and O1 effectively optimizes every function in this dataset. When the optimization level is increased, not every function can be optimized further; looking at the ROC curve suggests that only half of the functions in the dataset can be optimized further and the changes introduced by O2 are not as drastic as O1. Finally, due to the low overall complexity of the functions, O3 does not

introduce any other optimizations and all functions in O3 are exactly the same as in O2. Since every positive function pair sampled from O3 and O2 will be exactly the same function, this corresponds to the perfect classifier. This also shows how the complexity of the synthetic dataset differs w.r.t. a dataset consisting of real-world applications. When looking at the performance of Ofci on functions with different optimization levels in section 6.6 and the ROC curves in Figure 6.3 and Figure 6.4, a real-world dataset shows the biggest change in ROC-AUC score when looking at optimization level O3, while the synthetic dataset shows no change between O2 and O3.

CHAPTER 7

Conclusion

Function clone detection remains a hard problem and surrounded by ever more complex growing machine learning approaches, this thesis introduced Ofci. With the initial intention of building on Trex [1] and focusing on obfuscated function clones, it became apparent that these modern approaches are too computation-heavy for production use. Therefore, Ofci trimmed down the model into a slimmer form of state-of-the-art architectures and implemented the approach as an efficient framework. Every part of this framework makes use of open-source technologies, with Ghidra as core reversing engineering environment and PyTorch as central machine learning library. This makes it possible to have an open end-to-end pipeline for function clone detection, whereas existing approaches rely on proprietary disassemblers like IDA. In addition to openness, the evaluation of Ofci has shown that the framework is able to scale to hundreds of thousands of functions, facilitated by Ghidra's headless analysis modes, outperforming existing approaches in terms of function processing.

Ofci has significantly reduced the number of trainable parameters when compared to the most recent and promising related approaches. This does not heavily affect the ability to classify function pairs based on their similarity, which is highlighted in the ROC-AUC scores of the evaluation. Unfortunately, not all aspects of Ofci have been as successful and the precision of function clone search leads to mixed results in comparison to other approaches. The reasons for these results have been discussed, but further scrutiny and trials are required. Similar issues can be found in the Call-ID and virtualized clone detection features of Ofci. The evaluation showed that Call-ID, i.e. adding function call identification info into the tokenized disassembly, can slightly improve performance function search performance, while virtualized clone detection through traces does not work in the current form as implemented by Ofci. By extensively analyzing these mixed results, Ofci was able to highlight some general issues in the way related work tackles the function clone detection problem. As future approaches need to tackle these issues, Ofci provides a mature framework for rapid prototyping and testing new models.

7.1 Future Work

The development of Ofci highlighted the difficulty of working with modern machine learning models: Research institutions have large amounts of computation resources available and these resources are used to squeeze out additional performance. This comes at the cost of approaches not being reproducible on consumer hardware or not being usable in production for reverse engineering. Therefore, one important issue is to reduce the complexity of new function clone search approaches, while keeping the search performance at the same level. Ofci has already reduced the number of trainable parameters by a large margin, but is still a computation heavy transformer model.

The feasibility of approaches ties in with the input length limitations. Ofci has shown that function pairs that are longer than the model input length generally perform worse in similarity classification and function search. However, just increasing the model input length is not enough, as this makes models even more expensive to train and use. As the length of functions in binaries varies greatly, a more elastic approach is needed. A model like this would ideally accumulate an embedding by moving a sliding window over a stream of instructions. This would also allow models to make better use of instruction traces and other dynamic information.

Another issue is the generation of datasets for training the models. While this is largely trivial for cross-architecture, cross-optimization and cross-obfuscation (in the case of O-LLVM) binaries, more interesting targets require more effort to create large enough datasets. Ofci ran into the issue of not being able to create a large and diverse enough dataset to train its model for cross-virtualization function clone detection. This issue is largely caused by Tigress not being able to function as drop-in replacement and working as a very limited source-to-source compiler. Other virtualizing obfuscators like Themida [20] and VMProtect [21] offer graphical user interfaces, which are not helpful when the goal is to build a large dataset. The best course of action for future work is to use Tigress for building a dataset based on real, open-source programs; due to the nature of Tigress, this does unfortunately require an enormous amount of manual work.

Another point that has not been touched by Ofci is the explainability of machine learning models. So far, no related approach has tried to analyze the inner workings of their models in order to explain *why* certain functions are classified as similar. As transformer models are now widely used, new recent research [124] tries to expand the possibilities for explainability. Lastly, there are several minor things that can be integrated into Ofci. While existing function clone detection approaches already support multiple architectures, Ofci only supports amd64, as its focus is on obfuscated function clones; adding new architectures should be as simple as adding additional binaries to the dataset, but the effects on function search performance need to be evaluated. Since Ofci also supports instruction traces through Intel Pin, another interesting prospect is the integration of additional information from dynamic traces, similar to the microtraces used by Trex. And finally, it is worth investigating whether using intermediate representations or raw bytes instead of disassembly is viable as input for binary code similarity models.

List of Figures

List of Tables

List of Listings

Bibliography

[1] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Trex: Learning execution semantics from micro-traces for binary similarity. *CoRR*, abs/2012.08680, 2020.

[2] James Oakley and Sergey Bratus. Exploiting the hard-working dwarf: Trojan and exploit techniques with no native executable code. *WOOT*, 10:2028052–2028063, 2011.

[3] The Go Development Team. The go project. `https://golang.org/project/`. Accessed: 2021-05-15.

[4] Catalin Cimpanu. Go malware is now common, having been adopted by both apts and e-crime groups. `https://www.zdnet.com/article/go-malware-is-now-common-having-been-adopted-by-both-apts-and-e-crime-groups/`. Accessed: 2021-05-15.

[5] National Security Agency. Ghidra. `https://ghidra-sre.org/`. Accessed: 2021-04-04.

[6] Trend Micro. Ursnif, emotet, dridex and bitpayme linked by loader. `https://www.trendmicro.com/en_us/research/18/l/ursnif-emotet-dridex-and-bitpaymer-gangs-linked-by-a-similar-loader.html`. Accessed: 2021-01-15.

[7] J. Upchurch and X. Zhou. Malware provenance: code reuse detection in malicious software at scale. In *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 1–9, 2016.

[8] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1667–1680, New York, NY, USA, 2018. Association for Computing Machinery.

[9] D. Andriesse, A. Slowinska, and H. Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 177–189, 2017.

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

[11] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. Safe: Self-attentive function embeddings for binary similarity. In Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 309–329, Cham, 2019. Springer International Publishing.

[12] 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, 2018. Association for Computing Machinery.

[13] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *ACM Comput. Surv.*, 54(3), April 2021.

[14] Hex-Rays. Ida pro. `https://www.hex-rays.com/products/ida/`. Accessed: 2021-04-04.

[15] 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, pages 363–376, New York, NY, USA, 2017. Association for Computing Machinery.

[16] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. XDA: accurate, robust disassembly with transfer learning. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[17] Christian Collberg. The tigress c obfuscator. `https://tigress.wtf`. Accessed: 2021-05-13.

[18] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-llvm – software protection for the masses. In *2015 IEEE/ACM 1st International Workshop on Software Protection*, pages 3–9, 2015.

[19] Naville Zhang. Hikari obfuscator. `https://github.com/HikariObfuscator/Hikari`. Accessed: 2021-10-05.

[20] Oreans Technologies. Themida - advanced windows software protection system. `https://oreans.com/themida.php`. Accessed: 2021-05-13.

[21] VMProtect Software. Vmprotect software protection. `http://vmpsoft.com/`. Accessed: 2021-05-13.

[22] Binbin Liu, Junfu Shen, Jiang Ming, Qilong Zheng, Jing Li, and Dongpeng Xu. MBA-Blast: Unveiling and simplifying mixed Boolean-Arithmetic obfuscation. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1701–1718. USENIX Association, August 2021.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[24] Zhuyun Dai, Chenyan Xiong, Jamie Callan, and Zhiyuan Liu. Convolutional neural networks for soft-matching n-grams in ad-hoc search. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, WSDM '18, pages 126–134, New York, NY, USA, 2018. Association for Computing Machinery.

[25] Zellig S Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.

[26] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.

[27] Sanjeev Arora, Yuanzhi Li, Yingyu Liang, Tengyu Ma, and Andrej Risteski. A latent variable model approach to pmi-based word embeddings. *Transactions of the Association for Computational Linguistics*, 4:385–399, 2016.

[28] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[29] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[30] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2015.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[32] Christian Blichmann. Bindiff now available for free. https://security.googleblog.com/2016/03/bindiff-now-available-for-free.html. Accessed: 2021-05-12.

[33] Google LLC. Bindiff manual. https://www.zynamics.com/bindiff/manual/index.html. Accessed: 2021-05-12.

[34] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. Association for Computing Machinery.

[35] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[36] Steven H.H. Ding, Benjamin C.M. Fung, and Philippe Charland. Kam1n0: Mapreduce-based assembly clone search for reverse engineering. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 461–470, New York, NY, USA, 2016. Association for Computing Machinery.

[37] Wesley Jin, Sagar Chaki, Cory Cohen, Arie Gurfinkel, Jeffrey Havrilla, Charles Hines, and Priya Narasimhan. Binary function clustering using semantic hashes. In *2012 11th International Conference on Machine Learning and Applications*, volume 1, pages 386–391, 2012.

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

[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, 2017. Association for Computing Machinery.

[40] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. Binary code clone detection across architectures and compiling configurations. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 88–98, 2017.

[41] Jonathan Crussell, Clint Gibler, and Hao Chen. Andarwin: Scalable detection of semantically similar android applications. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security – ESORICS 2013*, pages 182–199, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[42] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting code clones in binary executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 117–128, New York, NY, USA, 2009. Association for Computing Machinery.

[43] Paria Shirani, Lingyu Wang, and Mourad Debbabi. Binshape: Scalable and robust binary library function identification using function shape. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 301–324, Cham, 2017. Springer International Publishing.

[44] 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)*, 2019.

[45] Ginger Myles and Christian Collberg. K-gram based software birthmarks. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, pages 314–318, New York, NY, USA, 2005. Association for Computing Machinery.

[46] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. Rendezvous: A search engine for binary code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 329–338, 2013.

[47] Md Enamul Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1-2):13–23, 2005.

[48] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *International Workshop on Recent Advances in Intrusion Detection*, pages 207–226. Springer, 2005.

[49] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS*, volume 52, pages 58–79, 2016.

[50] James J McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience*, 12(1):23–34, 1982.

[51] Yaniv David and Eran Yahav. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 349–360, New York, NY, USA, 2014. Association for Computing Machinery.

[52] 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, 2017. Association for Computing Machinery.

[53] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. *ACM SIGPLAN Notices*, 51(6):266–280, 2016.

[54] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[55] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[56] 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, 2016. Association for Computing Machinery.

[57] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 303–317, 2014.

[58] Software ethology: An accurate and resilient semantic binary analysis framework. *CoRR*, abs/1906.02928, 2019. Withdrawn.

[59] Jiang Ming, Meng Pan, and Debin Gao. ibinhunt: Binary hunting with interprocedural control flow. In *International Conference on Information Security and Cryptology*, pages 92–109. Springer, 2012.

[60] Beng Heng Ng and Atul Prakash. Expose: Discovering potential binary code reuse. In *2013 IEEE 37th Annual Computer Software and Applications Conference*, pages 492–501, 2013.

[61] 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, 2016. Association for Computing Machinery.

[62] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 2702–2711. JMLR.org, 2016.

[63] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. diff: 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, 2018. Association for Computing Machinery.

[64] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, Heyuan Shi, and Jiaguang Sun. Vulseeker-pro: Enhanced semantic learning based binary vulnerability seeker with emulation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 803–808, New York, NY, USA, 2018. Association for Computing Machinery.

[65] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages 1188–1196. JMLR.org, 2014.

[66] Yongjun Lee, Hyun Kwon, Sang-Hoon Choi, Seung-Ho Lim, Sung Hoon Baek, and Ki-Woong Park. Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with cnn. *Applied Sciences*, 9(19), 2019.

[67] Kimberly Redmond, Lannan Luo, and Qiang Zeng. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. In *NDSS Workshop on Binary Analysis Research (BAR)*, 2019.

[68] Zhouhan Lin, Minwei Feng, Cícero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding. *CoRR*, abs/1703.03130, 2017.

[69] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[70] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[71] Patrice Godefroid. Micro execution. In *Proceedings of the 36th International Conference on Software Engineering*, pages 539–549, 2014.

[72] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[73] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 463–469, Berlin, Heidelberg, 2011. Springer-Verlag.

[74] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 343–355, New York, NY, USA, 2016. Association for Computing Machinery.

[75] Fang-Hsiang Su, Jonathan Bell, and Baishakhi Ray Gail Kaiser. Deobfuscating android applications through deep learning. https://mice.cs.columbia.edu/getTechreport.php?techreportID=1632. Accessed: 2021-01-15.

[76] Douglas Low. Protecting java code via code obfuscation. *XRDS*, 4(3):21–23, April 1998.

[77] Guardsquare. Proguard, java optimizer and obfuscator. https://github.com/Guardsquare/proguard. Accessed: 2021-05-12.

[78] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

[79] Ramtine Tofighi-Shirazi, Irina-Mariuca Asavoae, Philippe Elbaz-Vincent, and Thanh-Ha Le. Defeating opaque predicates statically through machine learning and binary analysis. In *Proceedings of the 3rd ACM Workshop on Software Protection*, SPRO'19, pages 3–14, New York, NY, USA, 2019. Association for Computing Machinery.

[80] Fabrice Desclaux. Miasm: Framework de reverse engineering. https://github.com/cea-sec/miasm. Accessed: 2021-05-13.

[81] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the semantics of obfuscated code. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, pages 643–659, USA, 2017. USENIX Association.

[82] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. *SIGPLAN Not.*, 46(6):62–73, June 2011.

[83] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, New York, NY, USA, 2010. Association for Computing Machinery.

[84] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[85] Ben Ruijl., Jos Vermaseren., Aske Plaat., and Jaap van den Herik. Combining simulated annealing and monte carlo tree search for expression simplification. In *Proceedings of the 6th International Conference on Agents and Artificial Intelligence - Volume 1: ICAART,*, pages 724–731. INSTICC, SciTePress, 2014.

[86] Grégoire Menguy, Sébastien Bardin, Richard Bonichon, and Cauim de Souza Lima. Ai-based blackbox code deobfuscation: Understand, improve and mitigate. *CoRR*, abs/2102.04805, 2021.

[87] El-Ghazali Talbi. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009.

[88] Helena Ramalhinho Lourenço, Olivier C Martin, and Thomas Stützle. Iterated local search: Framework and applications. In *Handbook of metaheuristics*, pages 129–168. Springer, 2019.

[89] Robin David, Luigi Coniglio, and Mariano Ceccato. Qsynth-a program synthesis based approach for binary code deobfuscation. In *Workshop on Binary Analysis Research*, 2020.

[90] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[91] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1):305–316, 2013.

[92] 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), 2019.

[93] Binlin Cheng, Jiang Ming, Erika A Leal, Haotian Zhang, Jianming Fu, Guojun Peng, and Jean-Yves Marion. Obfuscation-resilient executable payload extraction from packed malware. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3451–3468. USENIX Association, August 2021.

[94] Philip Gage. A new algorithm for data compression. *C Users J.*, 12(2):23–38, February 1994.

[95] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics.

[96] Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[97] Changhan Wang, Kyunghyun Cho, and Jiatao Gu. Neural machine translation with byte-level subwords. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9154–9160, 2020.

[98] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108, 2019.

[99] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. *CoRR*, abs/1909.11942, 2019.

[100] Nils Reimers, Iryna Gurevych, Nils Reimers, Iryna Gurevych, Nandan Thakur, Nils Reimers, Johannes Daxenberger, Iryna Gurevych, Nils Reimers, Iryna Gurevych, et al. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2019.

[101] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *CoRR*, abs/2011.10749, 2020.

[102] Python Software Foundation. The python programming language. `https://www.python.org/`. Accessed: 2021-10-06.

[103] Oracle. Openjdk. `https://openjdk.java.net/`. Accessed: 2021-12-06.

[104] Python Software Foundation. Jython. `https://www.jython.org/`. Accessed: 2021-12-06.

[105] The PostgreSQL Global Development Group. Postgresql. `https://www.postgresql.org/`. Accessed: 2021-12-06.

[106] Inc. Docker. Docker. `https://www.docker.com/`. Accessed: 2021-10-06.

[107] The Rust Community. The rust programming language. `https://rust-lang.org`. Accessed: 2021-10-06.

[108] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[109] Guillaume Becquin. End-to-end NLP pipelines in rust. In *Proceedings of Second Workshop for NLP Open Source Software (NLP-OSS)*, pages 20–25. Association for Computational Linguistics, 2020.

[110] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[111] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[112] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2021.

[113] The HDF Group. Hdf5 documentation. `https://portal.hdfgroup.org/display/HDF5/HDF5`. Accessed: 2021-10-06.

[114] Intel Corporation. Pin. `https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html`. Accessed: 2021-10-08.

[115] Quarkslab. An experimental study of different binary exporters. `https://blog.quarkslab.com/an-experimental-study-of-different-binary-exporters.html`. Accessed: 2021-04-04.

[116] Ryan Govostes. Batch import of object files from archive with analyzeheadless? `https://github.com/NationalSecurityAgency/ghidra/issues/823`. Accessed: 2021-05-10.

[117] The Diesel Core Team. Diesel is a safe, extensible orm and query builder for rust. `http://diesel.rs/`. Accessed: 2021-10-08.

[118] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020.

[119] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. In *International Conference on Learning Representations*, 2018.

[120] Jorge M. Lobo, Alberto Jiménez-Valverde, and Raimundo Real. Auc: a misleading measure of the performance of predictive distribution models. *Global Ecology and Biogeography*, 17(2):145–151, 2008.

[121] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, USA, 2008.

[122] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. In *NeurIPS*, 2020.

[123] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *CoRR*, abs/2004.05150, 2020.

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