Scalable Clone Detection on Low-Level Codebases

Submitted to
Graduate School of Information Science and Technology
Osaka University

December 2022

Davide PIZZOLOTTO

Abstract

In recent years, the adoption of open source software and its inclusion into closed source projects greatly increased. This does not only include open source libraries, but also code snippets copied and pasted from Stack Overflow. Aside from the potential license violation, this code has been proven often outdated or containing vulnerabilities that have long been patched in the original code.

While most clone detection tools are focused on detecting copied code in source code, little work exists that can be applied to binary code or lower-than-source code. This may be a problem in case the original source code is not available or in case of huge codebases to analyze. In fact, existing binary clone detectors are usually limited to pairwise analysis.

In this dissertation, we discuss the detection of cloned snippets in low-level code. The dissertation is divided into two main parts: a first part highlighting techniques to improve clone detection in source code by transforming code into its low-level counterpart, and a second part focusing on scalable binary clone detection.

The first part provides two different approaches to improve clone detection by means of low-level code. In the first approach we take programs written in the Rust language, a particular language with a compilation pipeline composed by multiple steps, and we apply regular clone detection after performing some compilation transformations. While this approach is specific to a particular language, the second one is more generic and consist of implementing transformations similar to the one of a compiler in order o provide code normalization and improve clone detection.

The second part of this dissertation is instead based exclusively on binary code detection and is subdivided into two sub-parts. First, we discuss a novel approach for detecting clones in a binary file in a scalable way. This novel approach uses methods commonly found in decompilation and builds a representation of functions that can be compared in linear time. We show, however, that this method is highly sensitive of optimization options and, for this reason, we close this second part of the dissertation with a study on optimization flags detection using learning approaches.

Ultimately, we provide different methods for detecting clones in low-level languages using scalable approaches: in particular, in binary code clone detection we show that it is possible to reach the same precision of existing learning approaches while keeping the speed of a traditional one that is an order of magnitude faster.

Acknowledgments

I want to express my most sincere gratitude to my advisor Professor Katsuro Inoue, for his guidance, continuous support and patience during these years. Without his help, this work would have not been completed. I would also like to thank Professor Yoshiki Higo, for his help during the final stages of this work.

Besides my supervisors, I would like to express my gratitude also to the members of my committee: Professor Shinji Kusumoto and Professor Fumihiko Ino for their advice and comments in writing this thesis. I would also like to thank Associate Professor Makoto Matsushita for assistance and discussion in daily research and laboratory activities.

Additional thanks go to my former research mates: Kaoru Ito at Japan Research Institute, Assistant Professor Kazumasa Shimari at Nara Institute of Technology and Assistant Professor Tetsuya Kanda for the continuous and active research and non-research discussions.

I wish to thank all the members of the laboratory, and, in particular, special thanks to our lab secretary Ms. Mizuho Karube, for her assistance in absolutely every single problem I had to face here in Japan. My life in this country would have not been so smooth without her.

Finally, I would like to thank my parents Angelo Pizzolotto and Silvana Prosdocimo, and my sister Alessandra Pizzolotto for supporting me throughout my entire life.

List of Publications

Major Publications

1. **Code Clone Detection in Rust Intermediate Representation**.
   Davide Pizzolotto and Makoto Matsushita and Katsuro Inoue.
   In IPSJ/SIGSE, 2022-SE-211(26), 1-7 (2022-07-21), 2188-8825.

2. **Blanker: A Refactor-Oriented Cloned Source Code Normalizer**.
   Davide Pizzolotto and Katsuro Inoue.
   In Proceedings of the 14th IEEE International Workshop on Software Clones, IWSC 2020, London, ON, Canada, February 18, 2020.

3. **BinCC: Scalable Function Similarity Detection in Multiple Cross-Architectural Binaries**.
   Davide Pizzolotto and Katsuro Inoue.
   IEEE Access, 2022, Volume 10, Pages 124491-124506.

4. **Identifying Compiler and Optimization Level in Binary Code From Multiple Architectures**.
   Davide Pizzolotto and Katsuro Inoue.
   IEEE Access, 2021, Volume 9, Pages 163461-163475.

5. **Identifying Compiler and Optimization Options from Binary Code using Deep Learning Approaches**.
   Davide Pizzolotto and Katsuro Inoue.
   In Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020.

Related Publications

1. **OBLIVE: Seamless Code Obfuscation for Java Programs and Android Apps**.
   Davide Pizzolotto, Roberto Fellin, and Mariano Ceccato.
   In Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019.

2. **Obfuscating Java Programs by Translating Selected Portions of Bytecode to Native Libraries**.
   Davide Pizzolotto and Mariano Ceccato.

In Proceedings of the 18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Clone Detection

In recent years, copying and reusing portions of code has become a common practice. This is often done by taking an existing snippet of code, and by integrating it into an existing codebase while performing the necessary modifications [65]. However, aside from the potential license violation, this procedure introduces Code Clones. Code Clones, as the name implies, are portions of codes that exhibit some kind of similarities between them [82]. However, while the introduction of code clones may or may not be intentional [54], it has been proved that the presence of clones may introduce an additional maintenance burden [50]. The main problem of clones is related to the correctness of software, in a problem usually called *inconsistent bug fix*: if a fault is found in a cloned portion of code and not fixed in all the cloned instances, the program will still exhibit the incorrect behaviour [50, 80].

However, not every clone is equal. In fact, code clones can be classified into four categories, showing the degree of similarity between the represented code [82]. The first category, *type-1*, includes code that is identical, except for white spaces and comments. The second category, *type-2*, contains clones that are syntactically identical, except for variations in identifiers, literals, types, whitespace, layout and comments. The third category, *type-3* comprises clones with further modifications such as changed, added or removed statements, in addition to every change that can fit into the type-2 categorization. Finally, *type-4* clones, contains two fragments that performs the same operation but with a completely different syntax.

1.2 Scalable Clone Detection in Source Code

Over the course of the years, several techniques for detecting the various type of clones have been developed. These include mainly text-based [48, 81], token-based [51,91,94] and tree-based [8,47] approaches. In this dissertation, however, we focus mainly on token-based approach, given their higher precision compared to other type of approaches [91].

Token-based clone detectors work on the premise of transforming the input source code into a sequence of tokens and then detecting the various clones over

this sequence rather than the original text. While this was originally done to properly account for identifier renaming typical of type-2 clones [51], it has been proved to scale reliably also for type-3 clones [91, 112].

The main problem of token-based approaches, however, is the scalability of the approach for complex (type-3 and type-4) clone types. In fact, a traditional token-based clone detection has to compare pairs of code snippets representing potential clones. This has the downside of being an operation with complexity of $\mathcal{O}(n^2)$ as opposed to the $\mathcal{O}(n)$ that can be achieved using hashing techniques. In this case, n represents the potential clone snippet, and could be either a code block, a function or an entire file. While some scalable token-based techniques exist, for example by using token-based hashing, these usually do not support type-3 clones [43, 45]. Despite this problem being not particularly relevant in small projects, while analyzing big codebases or determining the evolution of clones, it may hamper the actual analysis. For this reason, Sajnani et al. developed *SourcererCC* [91], with a specific focus on scalability and the ability to analyze type-3 clones using a token-based approach while being able to tackle even huge codebases.

SourcererCC works by early rejecting potential clone pairs by using some heuristics: in particular, using clone size and analyzing the amount of overlapping code in a smaller portion of the potential clones, it is capable of greatly filtering the amount of comparisons to be performed. While this approach is still $\mathcal{O}(n^2)$, the filtering performed reduces the problem size to allow tractability even in huge codebases.

1.3 Code Normalization and Low-Level Code

While SourcererCC proved a high accuracy in detecting source code clones of type-3, the detection of type-4 is still on its early stages, with most tools relying on graph property matching [57] or machine learning [96]. To increase the detection of not only type-4, but any kind of clone with substantially different syntax, Ragkhitwetsagul et al. [76] noticed how, using a compiler increased the detection rate and quality. This is due to the amount of transformations done by the compiler over the original source code. In addition, further studies were conducted by Caldeira et al. on Intermediate Representation (IR) [11], a type of representation language-agnostic used internally by a compiler.

In this dissertation, and in particular in Part I, we investigate this normalization and its effects on improving code clone results. We use two studies, reported in Chapter 2 and Chapter 3. Both these studies uses existing code clone detector tools and retain their original scalability, providing scalable compiler transformations on top of that.

1.3.1 Code Normalization in existing compiler pipelines

The first study, reported in Chapter 2, investigates the code transformations provided by an existing compiler pipeline, in particular the one of the Rust language. The Rust language was chosen because its compilation pipeline is composed of multiple compilation steps applied in succession. These steps are used to check the language complex features with different code representations, and progressively refines and normalize the code: these representations include

normal source code, source code with expanded macros, High-level Intermediate Representation (HIR), Typed High-level Intermediate Representation (THIR), Mid-level Intermediate Representation (MIR), LLVM IR and, finally, binary code. Having different representations readily available from the compiler, coming from the same source, allows us to check for a possible increase in clone quality due to code normalization without implementing the code normalization by ourselves. Implementing these normalizations, instead, will be the subject of Chapter 3.

In the study, we force the Rust compiler to emit IR, in particular HIR, and then apply an existing Code Clone detection tool, namely *CCFinderSW* [94] on the generated code. We analyze the reported clone in the IR version and the original version, and conclude that, despite the increase in reported clones, this type of detection suffers from some impracticalities. In particular, it is notably hard to map the compiler IR back to the original source code.

1.3.2 Manually Implementing Code Normalization

Given the harder task of mapping the compiler IR back to the original source code, due to the fact that compilation is a process that is not under our control, we investigated the possibility of manually implementing these transformations manually.

Chapter 3 presents our study and analysis of this idea. We implemented several text transformations to effectively reduce the syntactic sugar for the C and Java language. Although not language agnostic, most of these transformations are quite simple and can be easily ported to other languages. Moreover, the transformations are applied only once to each source file, retaining the original scalability of the Code Clone detector.

After applying the transformation, we run clone detection limiting to type-2 clones. Our objective is to detect some type-3 and type-4 clones that were converted to type-2 using our text normalization. The main difference with respect to using an existing type-3 or type-4 detector lies in the fact that type-2 clones are easier to refactor, sometimes an "extract-method" action is sufficient, and with this analysis we can detect clones that are type-3 and type-4 before our normalization, but can be easily converted into type-2 and refactored.

The results we obtained shows that it is actually possible to increase the amount of detected clones by implementing the same transformation of a compiler, however, additional studies are needed to verify if the effort required to implement these transformations is worth the time necessary to implement and maintain them.

1.4 Scalable Clone Detection in Binary Code

While in Part I we investigated how it is actually possible to increase the accuracy and the amount of reported clones by transforming the source code into low-level code, in Part II we analyze a scalable way of detecting code clones directly in binary code.

Detecting clones in binary code would be of particular use in case of vulnerability detection and propagation: in particular, detecting the presence of a vulnerable library inside an already compiled, and possibly already-shipped,

executable. Not only this would help in cases where the source code is not available, but also when it is difficult to detect which version of the source code was associated with a particular binary. In these cases, binary clone detection could be used to detect the presence of vulnerable snippets and their propagation (e.g. through library linking) across a codebase.

However, despite the similarities between source and binary clone detection, several technical problems arise when dealing with binary code, and these problems usually hampers the scalability of the solution. In particular, the compiler transformation from source code into binary code, progressively removes information not needed at runtime, and transforms a code easier to read for humans into a type of code easier to execute for a machine. This involves the removal of any sort of structure from the code, and the usage of jumps in the code, that have been long associated with unmaintainable code and are no longer used in humanly-written source code [19].

In addition to technical problems [4], detecting clones in binary files also suffers from a taxonomy problem: unlike source code, binary code is not manually written[1] but only generated by a compiler. For this reason, there is no real distinction between *type-1* and *type-2* clones, given the lack of identifier naming, and using a similar taxonomy for register allocation would be useless for all practical purposes. Moreover, even the *type-1* is not clearly defined: two portions from two different binary files may use the exact same instructions, but have different jump offsets due to their positioning in the binary file. In this case, it is not clear if the two code portions should be treated as identical or not, and it may depend on the use case.

A starting point for a binary code clone detector is the transformation of the binary code into a text form, usually using architecture-dependent instructions. After that, while it may be tempting to use existing source-based approaches, our study shows that these are suboptimal, and better results can be achieved using ad-hoc methods.

The problem with ad-hoc methods, however, resides in their scalability: while accurate tools exist [20, 21], these usually work only on pairs of executables and scale poorly. This problem may be particularly important in the use case of vulnerability propagation in large-scale codebases, for example an entire Windows system or the LLVM toolchain.

In this second Part of the dissertation, we present two studies for detecting clones directly on binary code. The first study is reported in Chapter 4 and presents the actual clone detector, while the second study, reported in Chapter 5 investigates if it is possible to detect the compiler and optimization levels used in a given binary file.

1.4.1 Clone Detector

Despite several clone detector tools focused on binary code exist [20, 21, 68, 89], no tool can actually perform a scalable analysis on multiple binaries together. Existing tools are either focused on interactive analysis [27], require long trainig time [21] or have to deal with complex Control Flow Graph (CFG) analysis [24, 71]. In fact, in traditional approaches, analyzing a binary involves dealing with complex graph matching, that requires exponential time algorithms [102].

[1] Although assembly code can be written by a human, the final result is usually assembled by a compiler

In Chapter 4, we present our solution to solve this problem with a novel approach at a binary clone detector. In particular, instead of performing the analysis on the CFG of each function directly, we detect the various high-level structures and simplify CFG nodes, effectively transforming it into a rooted tree (an example of this transformation can be seen in Chapter 4 on Figure 4.1). Then, a Locality-Sensitive Hashing (LSH) function is used on the resulting trees and subtrees, reporting potential clones in linear time on the number of total subtrees in the analyzed executables. To reduce the number of false positives due to functions with similar structure but different semantics, we also use cosine similarity and rejects functions that have similar structure but with opcodes too dissimilar.

We show that this approach has not only the same accuracy of existing learning approaches, but it is three orders of magnitude faster, and can work with cross-architectural binaries.

1.4.2 Compiler and Optimizations Detection

While the binary clone detector presented in Chapter 4 has an accuracy comparable with state-of-the-art approaches, it still suffers from some problem originally highlighted by Sæbjørnsen et al.: if the compiler or the optimization flags used in the analyzed binaries are different, results are unreliable [89]. This is not a problem of the approach per se, but it stems from the fact that different compilers translate different source patterns in different ways. While the results in comparing binaries from different optimization levels and compilers are not dramatically inaccurate, there is a noticeable drop in accuracy.

For this reason, in Chapter 5, we present a study aimed at detecting the compiler and the optimization level of a given executable. The resulting approach is supposed to be used in the input files for any binary clone detector: if the binaries do not agree, it is likely that results may be inaccurate.

The optimization detector itself uses a machine learning approach: manually studying and learning the different optimization patterns for each compiler and each optimization level would be a task on the limit of feasibility. In particular, we trained two models, a faster and less accurate CNN and a slower and mode accurate LSTM. Both models have an accuracy between than 92% and 98%, and are in line with the time complexity of the clone detector presented in Chapter 4, thus retaining its scalability.

1.5 Outline

The rest of the dissertation is outlined as follows: in Part I we present approaches for increasing source code clone detection rates by means of using low-level code. We present Chapter 2 showing the results when applying existing compilation pipelines and performing clone detection with a state-of-the-art detector on IR. We then improve the technique in Chapter 3, by providing manual normalizations instead of using existing IR. In Part II instead, we focus solely on scalable binary clone detection: we present a detector in Chapter 4 and shows that not only has an accuracy compared to existing state-of-the-art, but it is also order of magnitudes faster. We then present some techniques to detect the compiler and optimization level in Chapter 5, that can be used to ensures the binary

analysis is consistent between multiple executables. Finally, in Chapter 6 we summarize the dissertation and describe some future works.

Part I

Improving Clone Detection with Low-Level Code

Chapter 2

Clone Detection in IR

2.1 Introduction

The practice of copying and pasting source code is frequently done by programmers, as this allows them to reuse the same source code multiple times. This leads to the creation of Code Clones: identical fragments of code scattered amongst a codebase. As harmless as these clones may seem, they instead present a burden for code maintainability and may become a huge technical debt. In fact, if a bug is found in a cloned fragment, it must be fixed in every instance of the clone, but this requires first the identification of every potential clone.

To alleviate this problem, during the years several code clone detection tools have been presented. These tools employ a variegate set of detection techniques: from token-based approaches of the *CCFinder* family [51] and *NiCad* [81], to the tree-based approach of *Deckard* [47], passing from graph based approaches [38] to the recent techniques employing deep learning [90]. As the detection performance in *type-1* clones (i.e. identical except for white spaces and layout) and *type-2* clones (i.e. identical except for identifiers, literals and white spaces) reached almost perfection, in recent years efforts shifted toward the detection of harder clones: *type-3* clones (i.e. identical up to a certain percentage) and *type-4* clones (i.e. different code but with the same functionality).

In order to better detect this type of clones, sometimes also the binary code has been analyzed, in particular by Ragkhitwetsagul et al. that applied a compilation and decompilation step to normalize differences between pieces of code with similar functionality [76]. Similar studies analyzed the possibility of applying normalization techniques directly to source code [72] and using LLVM Intermediate Representation to detect clones [11].

However, all these tools targets the same two languages: C/C++ and Java. Although some detectors provide a limited amount of additional languages, the study and the major focus is usually the Java language. This is done mainly due to the presence of a standardized clone benchmark, *BigCloneBench* [100] that contains clones in Java language only.

In this chapter, we evaluate the detection of clones in a modern and new language: Rust. We want to evaluate how the detection of clones is impacted upon transforming the original source code into a lower lever IR by means of a compiler. This particular choice of language is motivated by two main rea-

sons. Unlike other programming languages, Rust guarantees the memory-safety, thread-safety and null-safety of its programs at the price of increased compilers checks and language restrictions. In order to validate statically these safety guarantees, several transformations are done during compilation. In this study, we evaluate the refactorability of each clone with respect to the additional language restrictions. The second reason involves the Rust's compilation pipeline: we compare the clones detected both at source code level and at a lower intermediate representation, and checks whether the detected clones' quality increases or not.

The rest of the chapter is structured as follows: Section 2.2 opens the chapter by introducing related projects in the field of code clone detection. Section 2.3 explain some details about the Rust programming language in order to ease the understanding of this chapter. After that, Section 2.4 explains our approach in performing the current study and Section 2.5 shows our research questions and the relative experimental results. Section 2.6 discuss the limitations of our approach. Finally, Section 2.7 closes the chapter.

2.2 Related Work

Several code clone detectors have been developed by the research community in the recent years. These spans different types of approaches. Some detectors converts the source code into a stream of tokens and perform analysis on these tokens. These detectors comprises *NiCad* [81], *CCFinder* [51], *CP-Miner* [61], *iClones* [32], and many others [82]. More recently, additional tools have been developed to include more variety of languages, these includes *SourcererCC* [91], *CCFinderSW* [94] and *MSCCD* [112]. Several recent techniques also involves clone detector that does not operate on tokens or the Abstract Syntax Tree (AST): for example Amme et al. presented a clone detector for the Java language analysing the code dominator trees [3], while Saini et al. improved *SourcererCC* with deep learning capabilities in their *Oreo* clone detector [90].

Although a small amount of studies targeted specifically intermediate code generated by compilers [11] [72] [76], most studies targeting low level code focus on Java Bytecode. Recent works were performed by Yu et al. [109] that analysed fragments extracted from the bytecode and by Keivanloo et al. that used code fingerprint on the Java Bytecode [53].

Going even lower, some researcher even tried to find code clones directly on the binary layer. The most famous work is surely the one of Sæbjørnsen et al. [89] that analysed the similarity in assembly instructions. However, analysis at binary level is usually performed in order to retrieve information about software license violation, like in the work of Hemel et al. [37].

2.3 Background

Before describing the approach we adopted to run our experiments, in this section we are going to explain the unique characteristics of the Rust language. The Rust language is designed around safety, in fact it is guaranteed by the compiler the prevention of data races, memory safety and type safety [86].

In order to satisfy these guarantees, the compiler relies on the ownership

system [88]. This, in turn, contribute to two major differences from other languages:

Lifetimes
When a reference to an owned variable is passed around the code, the compiler must ensure that the owned variable remains valid for the entire life the reference. This allows the compiler to safely drop a variable immediately as it goes out of scope without needing a garbage collector or a reference counter. Lifetimes **must** be explicitly assigned by the programmer if the compiler can not infer them.

Mutability
Data races occurs when at least a pointer writes the data while another one can read the data, without any synchronization feature. In order to avoid this problem, Rust forbids the same reference to be hold mutably more than once, or to be hold mutably and immutably at the same time. The mutability for each variable **must** be declared by the programmer, otherwise the variable is considered immutable after the first assignment.

These two contraints may require additional keywords (e.g. `mut`, `'static`, ...) that might prevent the refactorability of otherwise identical snippets of code.

Moreover, in order to enforce these two constraints, the compiler employs a series of different IRs before emitting the final binary code. In order to understand this chapter we are interested in particular in two types of IR: expanded code and HIR.

2.3.1 Macro expansion

The Rust programming language, unlike C or C++ does not have a preprocessor. However, the compiler still provides a macro engine in order to ensure better maintainability. In particular, several "common" implementation of methods (e.g. `clone`, `cmp`, `default`) can be created with builtin macros. These macros are then expanded into their respective code during the first phase of compilation [84].

Consider as an example the code in Figure 2.1. We can see that, despite the struct A and B having two completely different types, the implementation of the `clone` method performed by the macro `#[derive(Clone)]` is absolutely identical. In order to perform code clone detection in Intermediate Code is thus important to account for duplicated code generated by these kind of macros.

2.3.2 HIR and THIR

After creating the AST, the Rust compiler, converts this tree into a desugared version called High-level Intermediate Representation (HIR) [83]. The HIR differs from the normal AST for the following reasons:

- parenthesis are removed, as they are not necessary anymore with the AST structure.
- The `if let` syntax is normalized into the `match` syntax.
- The `for` loop syntax is normalized into the `loop` syntax.

- The `while` loop syntax is normalized into the `loop` syntax.
- Additional constraints are relaxed and converted into generics.

After these changes we expect the HIR to contains different clones than the original source due to normalizations performed by the compiler. It is the scope of this chapter to understand the evolution of original clones after the transformations in the HIR.

The HIR is then used to infer data types by the compiler, and transformed into THIR [87]. However, THIR does not add useful information for clone detection and as such we limit our study at the HIR level.

2.3.3 MIR and IR

Although not used in our study, the step after THIR is the MIR [85]. The MIR code is more similar to a CFG rather than the original source code and is used mainly to checks the constraints of the ownership system [88]. If the ownership constraints are satisfied, the code can be lowered once again into a Codegen IR and the binary code finally generated. We plan to investigate the effects of MIR in code evolution as future works, as explained in Section 2.6.

2.4 Approach

An overview of the study is shown in Figure 2.2.

Firstly, for each case study, we gather each clone pair with the use of a Code Clone detector. Then, the original code is run through the Rust compiler and instructed to emit the HIR. At this point, we run the same code clone detector and collect the clone pairs for the HIR code. We then manually analyze each clone pair for every project in both the original code and HIR code, and report if the clone pair is really a clone or a false positive and compare it with the same clone in the HIR code. Note that we limit the analysis to type-1 and type-2 clones, given that most type-3 and type-4 detectors target the C and Java language only.

2.4.1 Case Studies

We select 15 popular (more than 2M downloads) and decently sized (more than 2K Line of Code) Rust projects as our case study. These projects are listed in Table 2.1 along with the amount of Line of Code and their popularity. All these projects are publicly available in the Rust Package Registry[1]. The amount of Line of Code are determined by the tool `cloc`[2] and excludes comments and blank lines.

The scope of these project is greatly diverse: it ranges from byte manipulation (*bytemuck* and *bytes*), to data structures (*generic-array*, *smallvec*, *hashbrown*), including async computation (*crossbeam-channel*[3] and *dashmap*).

[1] `https://crates.io`

[2] `https://github.com/AlDanial/cloc`

[3] *crossbeam-channel* is part of the *crossbeam* package

Table 2.1: Case studies evaluated, with version, number of lines of code excluding comments and blanks, and number of downloads.

Software	Version	SLOC	# of Downl. (10^6)
slab	0.4.6	1311	54
smallvec	1.8.0	2204	78
generic-array	0.14.5	2595	65
num-rational	0.4.1	3064	25
dashmap	5.3.4	2866	10
bytemuck	1.9.1	3026	8
bytes	1.1.0	4601	69
cgmath	0.18.0	8878	2
bstr	0.2.17	6227	26
hashbrown	0.12.1	10104	56
crossbeam-channel	0.8.1	15652	13
petgraph	0.6.2	20308	23
bitvec	1.0.0	26996	13
ndarray	0.15.4	29281	3

2.4.2 Compiler and Code Clone Detector

In our study we used an existing compiler to generate the HIR code and an existing Code Clone Detector to find the clone pairs. The compiler used to generate the HIR code is `rustc`, the compiler provided by the Rust project itself. This compiler enables emitting all the intermediate code such as HIR and MIR using the `-Z unpretty` flag. In particular, we used the option `-Z unpretty=expanded` to check the macro-expansion result, and `-Z unpretty=hir` to emit the HIR code. The compiler version we used is the 1.60.

Concerning the Code Clone detector, our options are pretty limited. Famous detectors like *CCFinderX* [51] and *NiCad* [81] do not support Rust. The modern *SourcererCC* tool [91], although not officially supporting Rust, can be easily extended, while an even newer tool called *MSCCD* [112] has builtin support. These two tools can support up to type-3 clones. However they greatly lack in the clone reporting capabilities. For this reason, in this analysis we used the *CCFinderSW* tool [94] that, despite supporting only up to type-2 clones, provides a more efficient reporting tool, allowing us to manually investigate all the clones in the 15 projects. The tool was run with a parameter `t`, the minimum number of tokens requires to signal a clone, equal to 65. This value is more conservative than the default one of the clone detector (90) and was chosen in order to detect even small functions composed by a couple of lines.

Note that it is not the scope of this chapter to compare the difference between the various tools, rather than to analyze the code evolution in the Rust Intermediate Representation.

2.4.3 Filtering

As explained in Section 2.3.1, before generating the HIR, the compiler expands macros. These macros are designed to avoid clones in source code for repetitive tasks (i.e. implementing the `clone` method on a struct). However, given that

the expansion is performed before the HIR generation, these macros will result in a lot of false positives: the macro implementation can be found multiple times in the HIR but only once in the original source. In Figure 2.2 we can note a step called "Filtering" that refers exactly to this step: cleaning up the HIR code from obvious macro expansion.

The filtering step is actually performed in two different ways depending on the type of macro targeted. In case of `#[derive(...)]` macros, the one explained in Section 2.3.1, the expanded method will have the statement `#[automatically_derived]` prepended to it. This can be seen also in Figure 2.1 in the expanded code. The result of these macros can be easily removed by checking the AST, emitted by rustc, and by stripping away the entire block of code following the `#[automatically_derived]` keyword.

The second type of macros are the one defined by the `macro_rules!` syntax. The expansion of these macros is usually simpler than the one performed with the `#[derive(...)]` attribute, as they have a fixed set of parameters. We implemented a filter for the most common ones provided by the Rust language (e.g. `vec!`, `write!`, `println!`...) by using only regular expressions.

In both cases, however, our filter does not cover custom macros defined per-crate.

2.5 Evaluation

In order to determine the clone evolution in the Rust Intermediate Representation, we want to answer the following three Research Questions:

- **RQ1**: *type*. What type of clones can be usually found in a Rust project? Can the clones be easily refactored?
  RQ_{type} is a study on the type of clones that can be found in the Rust ecosystem. Rust is fundamentally different from other languages, as explained in Section 2.3, given its stricter compiler and limited usage of variables. In this research question we want to investigate if these differences implies additional clones that can not be refactored as one would do in a canonical language, without violating Rust's constraints of mutability and lifetimes.

- **RQ2**: *agreement*. How different are the clones between original code and HIR? What type of clones are detected only by one method?
  $RQ_{agreement}$ is meant to investigate the usefulness of the HIR representation in detecting clones. To answer this question we match all the clones reported in the original source code and all the clones reported in the HIR code and check for differences. We then report the principal causes of divergences between clones in the original code and clones in HIR code.

- **RQ3**: *accuracy*. How accurate is the clones detection in both original code and HIR? How many false positives are generated by the code? $RQ_{accuracy}$ is meant to investigate the accuracy of the clone detection in both original code and HIR code. This means reporting the amount of actual code clones and false positive code clones for each of the two code categories.

We ran this analysis on a Mac mini with M1 processor and 16GB of RAM, running macOS 12.3.1 and manually analyzed every single clone reported using

the Gemini tool provided with *CCFinderSW*.

2.5.1 RQ1: type

To answer RQ1 we run the Code Clone detector on every project listed in Table 2.1 and obtain the results shown in Table 2.2.

Table 2.2: Number of clones detected for each project in the original source code.

Software	#Clones
slab	11
smallvec	16
generic-array	N.A.
num-rational	14
dashmap	18
bytemuck	14
bytes	43
cgmath	328
bstr	179
hashbrown	159
crossbeam-channel	807
petgraph	N.A.
bitvec	561
ndarray	396

We then manually check these 3033 clone and categorize them. Among them, not a single clone is due to a limitation of Rust (e.g. the necessity of putting a variable `mut` instead of immutable). This is not surprising, given that declaring different lifetimes or different mutability requires additional keywords, that go beyond the type-2 clones category. Note that the projects *generic-array* and *petgraph* do not report the number of clones: in fact for these projects *CCFinderSW* failed to generate a report, despite running for more than 24 hours. Moreover, these two projects are considerably smaller than others in our case study, and we can only assume the parser failed to correctly tokenize the codebase.

The highest amount of clones involves the manual implementation of "common" method such as `len` or `fmt` (debug print) that are copy pasted for each structure. This category is so ubiquitous that every project contains at least a clone of this type. In some cases, the definition varies only in the type of generics targeted, with the implementation copy-pasted several time. All these clones can be easily replaced by a macro, increasing the maintainability of the code.

Another common category, present in at least half the projects, is the variation of a method's parameters. Most methods can require a different number of parameters and most projects just copy-paste the method implementation. This problem is again, easily refactorable by having a generic method accepting all the possible parameters and specialized methods that calls the former after setting default values for the parameters.

Finally, a surprisingly high amount of clones can be found marked as *tests* or *benchmarks*. Rust allows mixing test code with implementation code by prepending the test function with a `#[test]` statement, and in our evaluation these tests mixed with implementation code have not been filtered out. A notable example of this is the *crossbeam-channel*, where a greater majority of clones were reported amongst these tests mixed with the implementation code.

We can conclude RQ1 with the following statement:

No Rust-specific features can be found in the original type-2 clones detected. Most clones involve manual implementation of common methods and traits or implementation of the same methods with variations in the number of parameters.

2.5.2 RQ2: agreement

After determining the type of clones that can be found in the original Rust projects, we perform the same clone analysis on the HIR code. Table 2.3 reports the variation in SLOC for each project after running the macro expansion step and the HIR generation step. We can note how, in every project, the number

Table 2.3: Lines of code in the original source (SLOC_{orig}), after macro expansion (SLOC_{exp}), and after HIR generation (SLOC_{HIR}).

Software	SLOC_{orig}	SLOC_{exp}	SLOC_{HIR}
slab	1311	671	838
smallvec	2204	1253	1783
generic-array	2595	2261	2970
num-rational	3064	2690	3680
dashmap	2866	1752	2189
bytemuck	3026	1206	1482
bytes	4601	3069	3837
cgmath	8878	17727	23843
bstr	6227	4453	5826
hashbrown	10104	4032	5043
crossbeam-channel	15652	4699	5480
petgraph	20308	15371	19348
bitvec	26996	15233	18772
ndarray	29281	19947	25886

of effective lines of code is decreased compared to the original project. This is due to the fact that, in order to obtain the macro expanded version or the HIR version, the compiler needs to be invoked. After invocation, all the boilerplate code is removed, the imports resolved and the eventual conditional compilations (e.g. tests) removed. This effectively reduces the amount of non-interesting benchmarks and tests clones reported in Table 2.2 and cited in Section 2.5.1.

After running the clone detection and collecting the results we obtain the number of clones show in Table 2.4, along with the results of the original code detection results.

Table 2.4: Number of clones detected for each project in the original and HIR code.

Software	#Clones (original)	#Clones (HIR)
slab	12	53
smallvec	14	24
generic-array	N.A.	18
num-rational	16	215
dashmap	14	35
bytemuck	18	55
bytes	37	287
cgmath	328	165834
bstr	179	311
hashbrown	159	151
crossbeam-channel	807	351
petgraph	N.A.	2991
bitvec	561	3909
ndarray	396	30669

The most impactful result we can notice is the immense increase in the number of clones for certain projects, in particular *cgmath* and *ndarray*. The gargantuan amount of clones is manually intractable, but we ensured to analyse at least a clone for each clone class reported by the detector. The reason of this immense increase in clones is the high amount of procedural code involved in these two projects. *cgmath*, a linear algebra library, requires the implementation of basic mathematical operations for different vectors of different size. This is done in the project by using macros that were not intercepted by our filtering process. This is evident also by the distribution of clones, as shown in Figure 2.3 and Figure 2.4.

We can see how most of the clones are clustered in big areas. Figure 2.3 shows clearly repeated structures in the clones. Upon further inspection those cluster of clones are the implementations of identical mathematical operations for different dimensionality of linear algebra tools (e.g. `Vector2`, `Vector3`, `Matrix3`, `Matrix4`, ...), accepting different generic arguments. A similar reason can be given to the *ndarray* project, represented in Figure 2.4. Also in this case, we can note a big concentration of clones clustered together, due to the usage of procedural macros.

On the other side of the spectrum, instead, we can see how the clones for *crossbeam-channel* have been decimated, going from 807 to a meager 351. This result, however, is uninteresting, as we already discussed in Section 2.5.1 how these clones were mainly due to test code mixed with implementation code. After generating the HIR code, these tests are evicted by the compiler, and thus all the relative clones are not reported.

Smaller projects, on the other hand, turned out to be more interesting: on the *smallvec* project, for example, every additional clone is due to normalization into the `match` statement. However, these clones detected only in the HIR are usually present in a small amount.

We can summarize RQ2 as follow:

For big projects, highly dependent on procedural macro, analysing the HIR is detrimental as most of the clones are the result of macro invocation. In projects with less macro usage, however, HIR clones can highlights interesting similarities between methods due to code normalization.

2.5.3 RQ3: accuracy

In the last question, we want to investigate the accuracy of the clone detection system and the usefulness of the HIR code for clone detection. After thoroughly analysing all the original source code clones and most of the HIR clones, we can assess that the type-2 clones reported by *CCFinderSW* were all genuine. However, this does not mean they are useful.

In fact, unlike the original clones, the HIR ones require a more careful detection threshold in order to provide useful insights. Consider for example the statement shown in Figure 2.5. The original code, line 905 in the file `src/lib.rs` from *smallvec*, is a one line invocation of a single function. However, when expanded, its length increases considerably, surpassing the threshold and being detected as a clone. We determined that, despite all clones being true positives, most of the HIR clones fall in this class: true positive but useless for refactoring purposes. Furthermore, only a handful of clones, usually a single digit number, can be efficiently refactored after being detected only in HIR code. In order to solve this problem, a different threshold must be used for HIR clones, given that most of the changed code in HIR increases considerably in size. This happens despite the HIR code having an overall less amount of line of code, as shown in Table 2.3.

We can thus conclude RQ3 as follow:

Despite the accuracy of the clone detection being 100%, most of the HIR clones are unusable due to the fact that the original code was already simple enough. Additional care must be taken in order to set a higher threshold for HIR clones.

2.6 Threats to validity and Future Works

In this study we used 15 projects of diverse size and scope but this is only a small fraction of the ecosystem and may diverge substantially from the real distribution of clones in the Rust language. We assumed the correctness of the rustc compiler in emitting the HIR code and the correctness of the *CCFinderSW* code clone detector in detecting the clones for the Rust language, however, if one or both these fact do not hold, the entire analysis may not be valid.

Moreover, we focused and drew conclusion based only on type-2 clones, where most of the Rust-specific features will likely require clones with minor difference in the used reserved words (i.e. type-3 clones). For this reason we plan to conduct further studies on the ecosystem focusing on type-3 clones and the transformations introduced by further lowering the code. This additional lowering targets the MIR that is more similar to a PDG style analysis [38] rather than a token based clone detector.

Finally, we did not have a comprehensive database of clones like in the *BigCloneBench* project [100], for this reason we can't now the real number of true positive and false negative clones present in our case studies. We can only count the number of false positives based on the performed manual evaluation.

2.7 Conclusion

In this chapter we investigated the code clone detection in a language different from the commonly targeted C or Java. We analyzed 15 projects and all their reported type-2 clones manually and determined the common causes of clones in the Rust language. We then compiled these clones, emitted their High-level Intermediate Representation (HIR), and ran again the code clone detection over this representation. Finally, we compared the original code and the Intermediate Representation (IR).

We determined that, although the code clone detector being capable of recognizing genuine clones in both the original code and the HIR code, only a small amount of code from the HIR code can actually be used. Most of them in fact are simple enough even in the original code and most of the complexity is added by the HIR generation step. For this reason the HIR code should be used as an extension of the original code, rather than completely replacing it.

```rust
#[derive(Clone)]
struct A {
    inner_string: String
}
#[derive(Clone)]
struct B {
    inner_number: u32
}
```

(a) Original Rust code

```rust
struct A {
    inner_string: String,
}

#[automatically_derived]
#[allow(unused_qualifications)]
impl ::core::clone::Clone for A {
  #[inline]
  fn clone(&self) -> A {
    match *self {
      Self { inner_string: ref __self_0_0 } => A {
        inner_string:::core::clone::Clone::clone(&(*__self_0_0)),
      },
}}}

struct B {
    inner_number: u32,
}

#[automatically_derived]
#[allow(unused_qualifications)]
impl ::core::clone::Clone for B {
  #[inline]
  fn clone(&self) -> B {
    match *self {
      Self { inner_number: ref __self_0_0 } => B {
        inner_number:::core::clone::Clone::clone(&(*__self_0_0)),
      },
}}}
```

(b) Rust code after macro expansion

Figure 2.1: Automatic implementation of the `clone` method by the compiler, via a procedural macro.

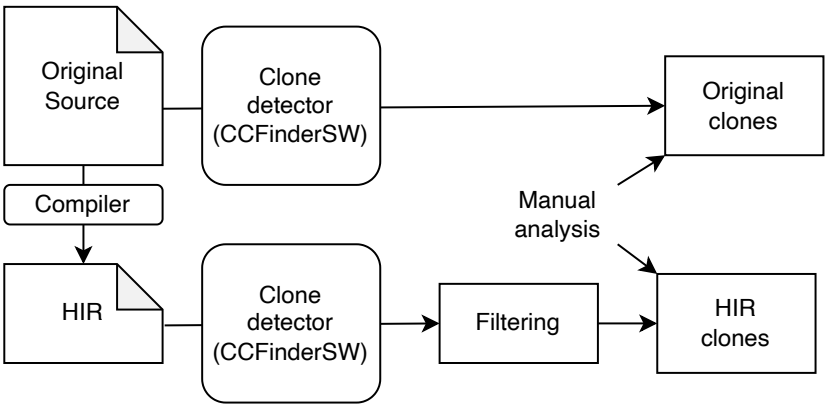


Figure 2.2: Overview of the study.

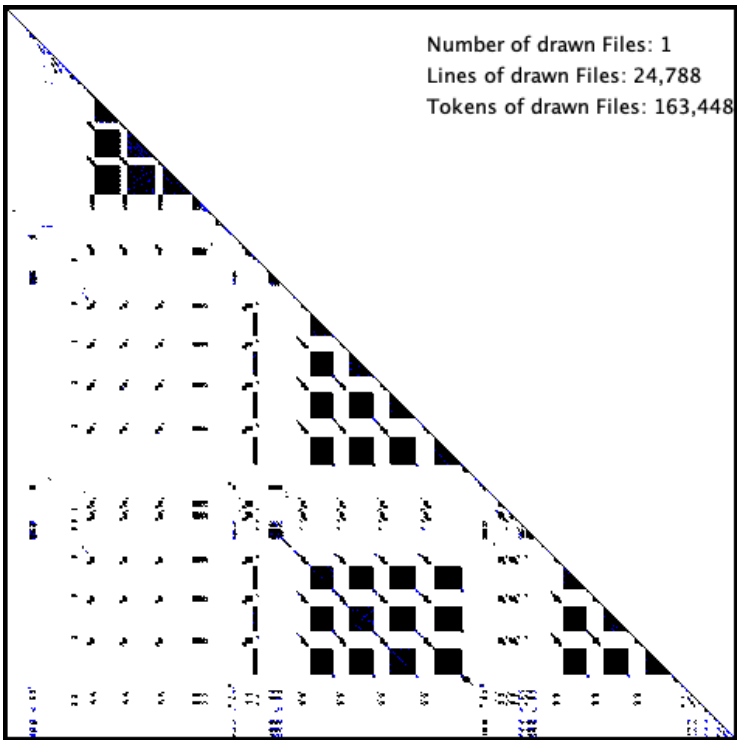


Figure 2.3: Clone scatterplot for *cgmath* HIR code.

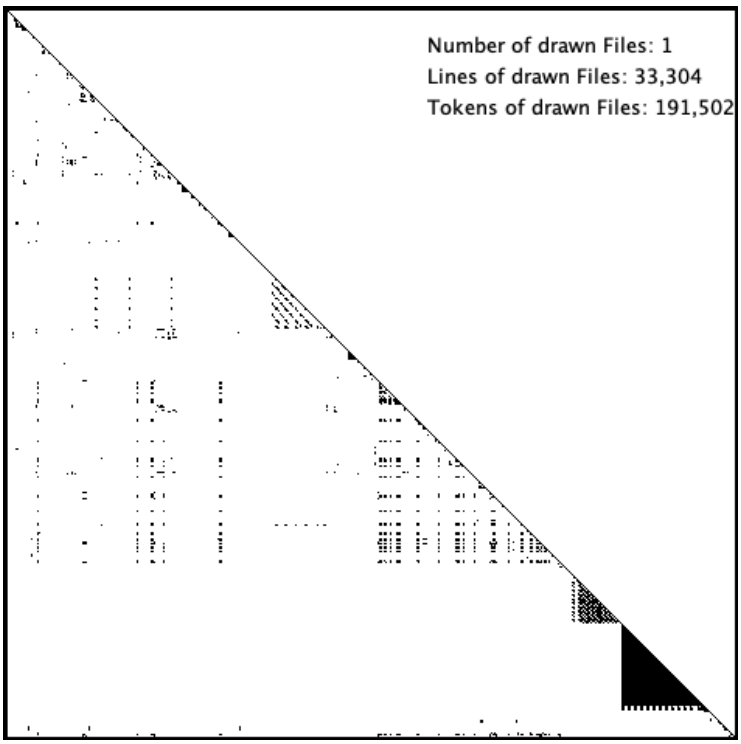


Figure 2.4: Clone scatterplot for *ndarray* HIR code.

```
let layout = layout_array::<A::Item>(new_cap)?;
```

(a) Original Rust code

```
let layout =
match #[lang = "branch"](layout_array::<A::Item>(new_cap))
{
#[lang = "Break"] { 0: residual } =>
#[allow(unreachable_code)]
  return #[lang = "from_residual"](residual),
#[lang = "Continue"] { 0: val } =>
#[allow(unreachable_code)]
  val,
};
```

(b) Rust code after macro expansion

Figure 2.5: Expansion of a simple statement into a more complex one triggering unrefactorable clones. This example is taken from the *smallvec* project, file `src/lib.rs`.

Chapter 3

Transforming Source Code

3.1 Introduction

Refactoring is the process of changing the code structure without changing its behavior. One of the most widely performed refactoring activity is the Extract Method, that simplifies methods by moving existing portions of code into a new method that can be reused [29]. This allows a developer to resolve a detected code clone by extracting it into a method and reusing it different times. For this reason, code clone detectors invest an important role in this scenario, given their ability to quickly identify a code portion that may be a refactoring candidate.

However, not always the output of a code clone detector can be used. In fact, a code clone detector usually categorize clones into four categories: *type-1* which are portions of code completely identical, *type-2* which are portions identical except for the variable naming, *type-3* almost identical except for a few statements and *type-4* which are different portions of code but with the same behavior [82]. Considering these categories, only the *type-1* and *type-2* can be easily and, almost automatically, refactored. Instead, *type-3* clones need to be manually checked and require an effort by the programmer that needs to check if the extra statements can be safely removed or not. This is true also for *type-4* clones that require even more work by the programmer, in addition to being extremely difficult to detect [82]. Our tool aims at normalizing some features provided by the programming language without changing the semantic, so a clone that would normally be categorized as a *type-3* or *type-4* can be categorized as *type-2* and thus easily refactored. As an example, Figure 3.1 shows a snippet taken from the class `XYBlockRenderer.java`[1] of JFreeChart: we can note the missing `else` keyword on the transformed version, that has been removed for consistency while retaining the same semantic.

This study follows the work of Ragkhitwetsagul and Krinke that showed how compiling and decompiling a source file can greatly increase the amount of detected clones, given that the compilation acts as a sort of "normalization step" [76]. However, the compilation/decompilation routine is not always applicable and can introduce errors in languages like C or could lead to false positives also in the Java language, as the aforementioned authors already discovered (i.e. by replicating some methods of the parent class of an inner class). To solve the

[1] `org/jfree/chart/renderer/xy/XYBlockRenderer.java`

```
if (r == null) {
    return null;
}
else {
    return new Range(r.getLowerBound() + this.yOffset,
            r.getUpperBound() + this.blockHeight + this.yOffset);
}
```

(a) Original Code

```
if (r == null) {
    return null;
}

    return new Range(r.getLowerBound() + this.yOffset,
            r.getUpperBound() + this.blockHeight + this.yOffset);
```

(b) Transformed Code

Figure 3.1: Example of transformation taken from JFreeChart.

problem we propose Blanker, that aims at replacing this compilation/decompilation routine with a plain code transformation. We replicated the setup and analyzed the various discoveries of Ragkhitwetsagul et al., then we manually implemented the transformations performed by the compilation step. Despite working with Java, in order to replicate the original study, we also ensured that our tool can perform the transformation on C source files.

Throughout the chapter we will refer to easy-to-refactor clones. With this name we intend type-2 clones that can be solved by a simple Extract Method process, instead of requiring an in-depth analysis of the extra statements like the type-3 clones.

To evaluate Blanker we replicated their exact same scenario by using the same three open source projects, namely *JUnit*, *JFreeChart* and *Apache Tomcat*, and *NiCad* [81] as code clone detector. We demonstrated that despite our tool missing a small amount of clones, compared to the compilation/decompilation approach, it has virtually no false positives and provides the added flexibility of being applicable to languages where the decompilation step may introduce errors.

In the rest of the chapter, Section 3.2 presents related state-of-the-art works. Section 3.3 shows our approach at implementing Blanker along with the engineering aspects, and Section 3.4 describes the evaluation of the tool. Section 3.5 closes the chapter.

3.2 Related Works

Clone detection is an active area in Software Engineering and several approaches have been proposed, ranging from token-based techniques such as NiCad [81], SourcererCC [91] and CCFinder [51] to structure analysis tools such as Deckard [47].

Multiple studies have been conducted on the compiled version of a software:

Selim et al. worked with an Intermediate Representation of Java (Jimple) by adapting a token based clone detector [93], Kononenko et al. used another adaptation to work at bytecode level [56], Davis and Godfrey analized the compiled assembly of a program using string matching [17].

Finally, Ragkhitwetsagul and Krinke used the compilation to achieve normalization and decompilation to avoid adapting existing clone detectors to work at a lower level [76].

3.3 Approach

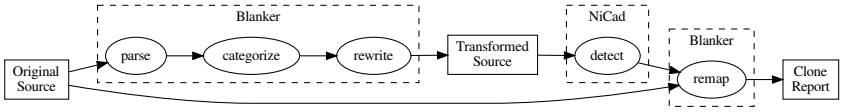


Figure 3.2: Overall structure of Blanker.

Blanker follows a linear workflow depicted in Figure 3.2 and composed by the following phases:

Parse: The original source file is parsed and the position of every semantic structure is recorded.

Categorize: The parsing result is analyzed in order to find possible refactoring candidates.

Rewrite The possible normalizations highlighted in the previous step are applied to a new file.

Detect: The code clone detector is applied to the rewritten file.

Remap: Possible discrepancies between the transformed file and the original file are addressed in this step.

The following subsections explains every component in detail. However, in order to better understand the following phases, it is worth precising that we built Blanker upon the results of Ragkhitwetsagul et al. [76]. In their previous results they showed that most of the normalizations performed by the compilation/decompilation process are applied to if-else statements, so, naturally, throughout our entire analysis, we focused greatly on those structures.

3.3.1 Parse

In order to transform a source code file, the first step requires identifying the semantic structures composing it. We built our tool aiming at negligible speed, so a single-pass token parser using *flex*[2] as lexer and *bison*[3] as parser was built.

[2]https://github.com/westes/flex
[3]https://www.gnu.org/software/bison/

The great challenge imposed by this phase was the creation of a grammar expressive enough to recognize the required structures, namely `if` and `else` blocks, without having to implement the entire java grammar. In order to solve this problem we chose each semicolon as a statement representation and recorded the position of `if`, `else` and `return` keywords only. Additionally we used curly braces to represent list of statements and let the lexer consume the condition following every `if` keyword. This grammar assumes the input file being a valid java file, a reasonable assumption for our tool, and works fine without modifications also for the C language, while avoiding having to deal with things such as parentheses, assignments and operators.

3.3.2 Categorize

This phase is used to analyze the parsed data and discover actual structures that could be refactored. After analyzing the results of Ragkhitwetsagul et al. and replicating their experiments we determined that the most prominent normalizations performed by the compiler/decompiler combo were:

1. removal of `else` keyword after an `if` block terminating with a `return`. An example of this can be seen in Figure 3.3 where the else keyword can be omitted and the same logic is kept. It is worth noting that this particular check is also part of LLVM's coding standard recommendations under the name of *readability-else-after-return*

```
if (r == null) {
    return null;
} else {
    return new Range(...);
}
```

Figure 3.3: Categorization 1). In this case the else keyword is redundant.

2. Returning an equality or inequality between two variables is transformed into an if block with the equality as the condition and return true or return false as body

3. Returning a conjunctive boolean formula is splitted into an "explicit short circuit evaluation". In order words, every variable is checked by itself in an if block and if the variable does not hold, false is returned. An example of the transformed code for `return a && b;` is shown in Figure 3.4.

4. `final` keywords lacking consistency. Sometimes a variable could be final but this keyword is not present, but it is present in a cloned snippet somewhere else.

Another common normalization that, however, we did not address was declaration and assignment of a variable in a single line or in different lines, possibly interleaved by other statements.

```
if (!a) {
    return false;
if (!b) {
    return false;
}
return true;
```

Figure 3.4: Categorization 3). This could be written as `return a && b;`.

3.3.3 Rewrite

This phase is the actual transformation of the structure described in Section 3.3.2. A key requirement of this phase was keeping as much as possible the file similar to the original one, the reason being explained in Section 3.3.4. In order to accomplish this, we exploited several facts: firstly the whitespaces and newlines being meaningless in both C and Java. Moreover, these are ignored also by the code clone detector of our choice. Given this we could easily transform categories 1) and 4) described in Section 3.3.2 just by patching the redundant parts with whitespaces. Additionally we exploited the fact that also newlines are meaningless in both C and Java, allowing us to write multiple statements in one line. Categories 2) and 3) in Section 3.3.2 requires replacing a single statement with multiple ones, and this language feature allows us to maintain a one to one mapping between the original cloned lines and the transformed ones.

3.3.4 Detect and Remap

At this point, the Detect phase applies the code clone detector to our transformed files. In our implementation, however, the normalized files are not overwritten, so the report generated by the code clone detector will present wrong paths. This is a major problem nonetheless, given that the entire normalization process should be invisible to the user, and thus the remap phase is used to map every reference of the normalized files in the code clone detector report back to the original files. This also gives an indication why in Section 3.3.3 was important to keep the file as similar as possible to the original one: replacing a statement with one spanning more lines or less lines means having to remap also every line reported by the code clone detector.

3.4 Evaluation

Being this study based on the results of the paper of Ragkhitwetsagul et al., we replicated their evaluation setup [76]. The same three open source project were considered for this evaluation, namely *JUnit v4.13*, *JFreeChart v1.5.0* and *Apache Tomcat v9.0*. These projects are listed ordered by size, spanning from 9.7k LoC of JUnit to 241.9k LoC of Tomcat.

The code clone detector used for this evaluation was NiCad v5.2. Our tool can be used with any code clone detector, but we decided to use NiCad in order to compare with the previous study. Studying the effect on other code clone detectors, especially the ones not using a token-based approach, will be consid-

ered as future work. Unlike the previous study, however, we did not analyze type-3 clones: our goal is providing an easy-to-refactor clone, thus limiting the evaluation to type-1 and type-2. The type-3 clones with the noise generated by the additional statements are considered as out-of-scope for our purpose. The configurations used are thus the default of NiCad named *type1* and *type2c*, the latter being type 2 clones with consistent naming.

The Research Questions we wanted to answer are the following, similar to the ones of the original study:

- **RQ1**$_{agreement}$: How many clone pairs are reported by both approaches? How many are exclusive to the plain file and to the normalized file?
- **RQ2**$_{accuracy}$: How is the detection performance of our tool compared to the compilation/decompilation approach presented in the study of Ragkhitwetsagul et al.?

Despite not conducting a real performance study, we also want to highlight that the processing time impact our tool is negligible: in the three project we measured this time to be, on average, half a millisecond per file, with every file being independent of the others thus enabling multithread processing. Also for big projects this translates in a normalization step order of magnitude faster than the clone detection process which usually requires several seconds.

3.4.1 Agreement

In order to answer RQ1 we ran NiCad with both *type1* and *type2c* configurations on the testing repositories, firstly without any normalization and then with normalizations applied. Table 3.1 depicts the results for *type2c*, despite,

Table 3.1: Comparison of the amount of detected clones by NiCad without and with our normalization applied.

Project	Original clones	Normalized clones	Variation
JUnit	6	6	+0.0%
JFreeChart	373	397	+6.43%
Apache Tomcat	242	275	+13.64%

by nature, these vary greatly depending on the considered project and the coding style. *type1* clones are not reported given that the results are absolutely identical. This is expected, given that is really unlikely that an user writes some code semantically different and keeps the same variable naming. By analyzing the *type2c* results instead, we can notice that the normalized version reports more clones. We manually analyzed the normalized clones reported and confirmed that they are a superset of the non-normalized version. Every clone pair reported is actually refactorable, however, despite this, we have to precise that the category 4) explained in Section 3.3.2 could in practice produce unrefactorable results, given that we simply removed every `final` keyword instead of performing a full constness propagation analysis to ensure semantic preservation. However, this problem was not highlighted in the three projects and requires further analyses. We can thus answer RQ1 as follows:

Processing the files with Blanker helps the code clone detector to find up to 10% more clones. In the case studies no false positives and no false negatives were detected compared to the original source detection.

3.4.2 Accuracy

In order to answer RQ2 we also ran the compilation/decompilation method and compared the original and normalized clones against that. Our normalized method provides no false positives, but, although not providing false negatives compared to the original source code, the compilation/decompilation provided some clone reports that both the original and our normalized version failed to detect. In the case of JFreeChart these comprise assignment and declaration of variables interleaved by a different amounts of statements, swapped if-else branches and a loop converted from while to for (done by the decompiler normalization step). Although each of these appearing only once, they are a clear sign that in order to discover even more clones a flow analysis may be required. On the other hand, the compilation/decompilation suffers from false positives, in particular related to the duplication of inner class methods that happens during compile time and not in the code written by the user. We can thus answer RQ2 as follows:

Our tool provides virtually no false positives at the cost of missing some clones. The compilation/decompilation approach suffers both false positives and negatives but provides a better spectrum of results if compared in conjunction with the original source code

3.5 Conclusion

In this chapter we presented Blanker, a code normalization tool useful to change some structures in order to detect more easy-to-refactor clones. We implemented some compiler normalizations, similar to the one we detected in Chapter 2, and implemented them manually, so they can be run on a different language, detached from their original compiler.

Our tool, provides an increased number of clones with no drawbacks, false positives or false negatives compared to running a code clone detector on the original files. If a low false negative ratio is required, the compilation/decompilation approach could provide more results due to the flow analysis performed by the compiler, even though a much higher effort is required to actually intersect and analyze the results.

As future work we plan to conduct an extensive analysis on the various categorization performed, and implement extra ones in addition to flow analysis.

Part II

Clone Detection in Binary Code

Chapter 4

Function Detection in Binary Code

4.1 Introduction

Free software has grown in popularity in recent years, and with that also the adoption of this kind of software by companies and its integration inside closed source projects. This popularity is mainly driven by the ease of customization and flexibility rather than economic reasons [65]: code is usually modified, adapted or simply reused and then re-released with a compatible license. With a higher availability of code, another common practice has become copy-pasting and reusing source code in form of small code snippets from online discussion platforms such as Stack Overflow. Aside from the potential license violation, reused code snippets have been proven to be usually harmful [26] or with security flaws that in the original repository have long been patched [77]. Although code cloning has been successful at determining copied function reuse both for legal implications [31] and plagiarism detection [75], its main scope is usually code evolution and vulnerability propagation [44, 108, 111].

Nonetheless, in recent years, several clone detection tools targeting lower-than-source-code have been developed. These are mainly driven by a normalization step performed by the compiler, rather than the lack of availability of the source code. These works have been targeting Java Bytecode [53, 109] or LLVM IR [11].

Different motivations can be found when working exclusively with binary code: we already cited the license violation in closed source software [46, 110]. Additional scenarios may involve detecting the presence of a vulnerable function in proprietary software [24, 71], analyzing the evolution of closed source software or the evolution of compiler transformations, and even aiding the disassembly when a known function snippet is detected.

While existing works in the field are mainly focused on malware detection, the main motivation for our work is software evolution and propagation in binary files, even in case of security-related issues (e.g. bug propagation in release of proprietary software). For this reason, the focus points of our research are the ability to analyze multiple executables at the same time and detect the propagation of a given function, the ability to perform such analysis in a reasonable time

for any amount of given executables, and the ability to work amongst different architectures, even lesser known.

In order to address these problems, our approach is based on the idea of generating a common structure between the compiled code in different architectures in order to leverage compilation differences, similar to how decompilation helps normalize source code differences [76]. In particular, starting from a CFG, we generate a higher level structure using structural analysis. This structure allows efficient comparison in linear time, as opposed to the exponential time required by a subgraph comparison. This structure has the additional property of normalizing the various architectural differences, allowing cross-architectural comparison. Thanks to its linear time scalability, our approach can be used to efficiently analyze multiple binaries altogether, outlining the evolution and propagation of code among huge codebases such as the LLVM project or the GNU toolchain, while existing works are usually limited to a pairwise comparison [27, 68].

The main novelty of our work is the following:

- A novel structural analysis designed specifically for clone detection.

  Unlike previous work in decompilation (e.g. [95]), our analysis can not emit `goto`s and thus must sacrifice some correctness. For this reason, we had to develop a novel analysis with different detection rules for converting a CFG. This new analysis is described in detail in Section 4.3.4.

- A novel comparison method to efficiently detect structural clones among multiple binaries.

  Comparing two CFGs requires an exponential algorithm [102] and even the average function length of 40 nodes is intractable. While previous works have based their comparisons on statistical properties [24] or dominator trees [3] we based our comparison on tree hashing, allowed by our structural analysis.

- An evaluation of clone detection across different CPU architectures.

  Previous work using structural or semantic analysis always targeted trivial binaries such as Java Bytecode [3, 76] or has been limited to the same architecture [89]. Whereas cross-architecture clones have been researched with deep learning methods, we propose a structural analysis algorithm that works in binary code and does not require a training step.

The chapter is structured as follows: Section 4.2 presents the State of the Art in code cloning for binaries and binary analysis. Section 4.3 explains in detail how the entire analysis is done, including the structural analysis algorithm and the final comparison. Section 4.4 evaluates our approach across varying architectures. Section 4.5 describes some limitations and threats to the validity of our work, and finally Section 4.6 closes the chapter.

4.2 Related Works

Given the intrinsic difficulty at analyzing binary files and the high amount of information lost during the compilation process, the literature in clone detection dealing with compiled executables is more scarce compared to the one dealing

with plain source code. The most complete and, to our knowledge, only existing analysis aimed at finding clones in binary files is the one performed by Sæbjørnsen et al. which uses a semantic approach based on vectors containing the instruction sequences [89]. This analysis is itself an extension of the one developed by A. Schulman [92].

In the matter of license violation, on the other hand, Hemel et al. performed three different types of analyses based on string, data compression and binary deltas analysis [37]. Both these approaches have been tested only on executables or libraries within the same architecture, although the string analysis of Hemel et al. could technically be applied also to different architectures.

The structural reconstruction, instead, has been studied in-depth by Engel et al. [23] and later refined by Brumley et al. [10] and Yakdan et al. [107]. The latter two, however, studied an approach more akin to decompilation and semantic preservation while we care more about having the same reconstruction in different binaries or architectures rather than a correct reconstruction.

In the software security field, several tools are available to check for similarities between binaries. In particular, BinDiff [27] and DarunGrim [68] present a structural analysis that use CFG isomorphism and basic block matching. BinSlayer [9] and discovRE [24] improve the existing work by providing faster CFG matching. However, these studies are aimed at finding bugs by analyzing CFG properties, while we refined even further the analysis in order to transform the CFG into a tree and have linear time comparison. This allows us to process thousands of functions in seconds. In contrast to structural analysis, some tools present a detection based on semantic properties. It is the case of BinHunt [30] using symbolic execution and BinJuice [59] that extracts the semantic representation of basic blocks. While all these works are essentially single-architecture, Pewny at al. provided a cross-architectural one by using a semantic representation of basic blocks [71], while, in contrast, our tool uses structural similarity. In addition to these tools, BinSim [64] can support obfuscated binaries.

In recent years, with the prevalence of IoT devices, cross-architectural analysis of binary files has been the scope of work such as BinGo [12] and CACompare [41] that use the signature of a function to perform comparisons. Hu et al., in particular, investigates also the impact of different compilers and optimization levels [42].

Recent works have also started introducing Deep Learning approaches in order to detect similarities between binaries [62, 106]. While some of these were focused just on graph properties [25], recent approaches are more similar to Natural Language Processing [20, 21, 105, 113].

The main difference between our approach and Machine Learning-based ones is the lack of requirement of a training step: building and retrieving a training dataset is already challenging for some dominant CPU architectures [105, 113], and it may be almost impossible in case of obscure or proprietary architectures. Our approach implementation, instead, currently supports 20 architectures, and additional ones can be added by writing a few lines of code, provided a disassembler is available. An additional downside of the learning based approaches is the runtime: several works such as InnerEye [113] and DeepBinDiff [21] can reach high precision, but require an enourmous amount of time to run, that usually hampers scalability. In our approach, instead, the runtime is completely dominated by the disassembly time and is orders of magnitude faster than the aforementioned approaches.

4.3 Approach

This section presents our approach for detecting function clones in a binary file. Our idea consists of generating a tree-like representation of each function, starting from a CFG, that may represent the original source code structure. Then, efficiently checking for clones in the subtrees of these representations. This come from the idea that, even for different architectures, the duplicated code once compiled should retain the same structure in both architectures.

However, directly comparing a graph as the CFG is not trivial, as it is known to be a NP-complete problem [102]. Comparing two CFGs with an exponential algorithm, for every pair of function we want to process, is thus unfeasible.

For this reason we use a structural analysis step, to reduce a CFG into a tree, more suitable for analysis. This step is akin to a decompilation, but it differs from it in the sense that semantic correctness is secondary as opposed to reconstruction completeness, allowing us to remove CFG edges.

4.3.1 Overview

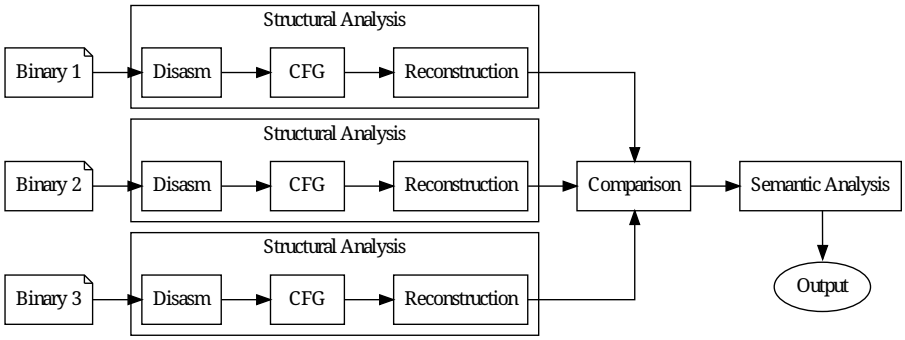


Figure 4.1: Overview of the binary analysis.

The overview of our approach can be seen in Figure 4.1. In this figure, several binaries labeled as *Binary 1*, *Binary 2* and *Binary 3* are analyzed in order to find cloned functions between them. We can see from the figure that our approach requires one or more binary files as input, and produces a single output: this output contains clone classes (clone sets) [82] which indicate cloned functions. Each clone class represents functions implementing the same behavior: two or more functions are reported as belonging to a clone class along with their name, original binary name and cloned basic blocks. Our approach compares all functions of all binaries together and is based on three main steps namely Structural Analysis, Comparison and Semantic Analysis.

Structural Analysis is used to retrieve a structural representation of every binary function. With those representations, the Comparison step generates the clone classes, and the Semantic Analysis further filters those clone classes from false positives. A detailed description of the various steps in Figure 2.2 is the following:

Disasm

In this step, the original binary code is disassembled and, for every function, a list of tuples `<offset, mnemonic, arguments>` is obtained. This step is performed once for every binary.

CFG
In this step, the CFG of each function is retrieved, starting from the list of statements. Additionally, these CFGs are refined and slightly modified as explained in Section 4.3.3. This step is performed for every function retrieved during disassembly.

Reconstruction
This step is the core of our approach: here each CFG is transformed into a representation of the original function using nested high-level structures (e.g. If, While, Do-While). These nested structures can represent the function in the form of a tree thus allowing constant time comparison, unlike a CFG. Our analysis, however, does not completely preserve the program structure. In fact, our approach modifies the CFG in case the analyzed function does not have structured control flow (e.g. loops with `break` and `continue` statements). For this reason, the algorithm is explained in detail in Section 4.3.4 along with reconstruction rules for each CFG pattern.

Comparison
The results of the reconstruction are compared altogether and if any duplication or match is found it is reported as output. Comparison is done using a hash-based approach, by searching for hash collisions while linearly processing all functions. Further details on this step are explained in Section 4.3.5.

Semantic Analysis
Finally, in this step potential clones reported by the comparison step are checked for semantic consistency. A different approach is used based on whether the potential clones belong to the same architecture or not. This step is described in detail in Section 4.3.6.

4.3.2 Disassembly

The first step of the entire analysis is the disassembly, requiring as input the original binary file, either as executable, library, or object code. Given that a binary file is just a collection of bytes interpretable as machine code, data, or comments, the purpose of this step is taking as input the aforementioned binary file and providing as output assembly instructions required by the subsequent steps. An assembly instruction is a pair `<mnemonic, arguments>` where *mnemonic* represents a particular instruction to be executed by a CPU, taken from a set of instructions composing the CPU Instruction Set Architecture (ISA), and *arguments* is a sequence of elements passed as arguments to the *mnemonic*. Additionally, ensuing steps of our analysis require assigning a unique *offset* to each instruction. This *offset* is generally calculated as the number of bytes from the beginning of the file until the instruction, and must be coherent with the jump targets: if an instruction performs a jump to another one, the argument of the jump must be the offset of the target instruction.

We can thus say that, for the purpose of our analysis, an assembly instruction is a tuple `<offset, mnemonic, arguments>`, and an example of this can be seen in the following code:

```
0x601 cmp dword [var_4h], 0
0x605 je 0x60E
0x607 mov eax, 0
0x60C jmp 0x613
0x60E mov eax, 1
0x613 pop rbp
0x614 ret
```

In this code, each line represents an instruction. The first number of each line is the offset, followed by a space and the mnemonic. Everything else on each line are the mnemonic arguments.

The expected output from this step is a list of function names, and for each function a list of assembly instructions. In our implementation we relied on external tools in order to perform the disassembly, as such, in order to keep generality, subsequent steps in our analysis assumes this list of assembly statements as plain strings in Intel syntax, as presented in the previous example.

4.3.3 Control Flow Graph

The purpose of this step is the transformation of the list of instructions obtained in the Disasm step into a suitable representation in form of a CFG, required for the structural analysis. This step is performed for every function obtained in the disassembly step. A CFG is a directed graph $\mathcal{G} = (V, E)$ where every node $v_k \in V$ represents a basic block and every edge $e_{i,j} = (v_i, v_j) \in E \wedge v_i, v_j \in V^2$ represents a possible movement from the basic block v_i to the basic block v_j. This kind of graph is effective at presenting the flow of the program and is the starting point for our structural analysis.

Given the relatively easy task of transforming a list of statements into a CFG, we are not going to present its implementation here. The only requirement is to have the list of conditional and unconditional mnemonics for each ISA.

Additionally, after retrieving the CFG, a refinement step is performed, consisting of the following actions:

Single exit
: If multiple exits for the analyzed function are found, a new single exit is created. This step is required to preserve consistency with some patterns defined later in Section 4.3.4.

Dead nodes removal
: Dead basic blocks, unreachable from the root, are removed from the resulting CFG. These basic blocks are resulting from indirect jumps, i.e. jumps to a dynamically known address which are unsolvable at static time and thus ignored during the CFG reconstruction [66] [33].

4.3.4 Reconstruction

The refined CFG with a single entry and a single exit obtained in the previous step is refined in this step. In particular, we aim at iteratively reducing CFG

portions into high-level structures until a single node is left. This node will represent the entire function, and recursively contains the various structures composing it. This representation two major advantages over a plain CFG:

- Loops are removed, and the resulting output is a rooted tree. The resulting tree nodes are labeled by type, allowing faster comparison through hashing. Comparing a CFG instead requires exponential time [102].
- Minor differences between the `x86_64` and `aarch64` CFGs are leveraged.

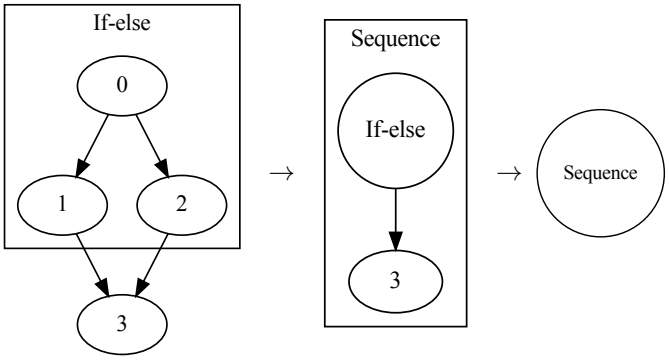


Figure 4.2: Example of the Reconstruction step. The nodes 0, 1 and 2 of the CFG are removed and replaced with a node labeled as If-else structure. The new If-else structure and node 3 can then be replaced with a Sequence structure. When only one node is left the algorithm terminates.

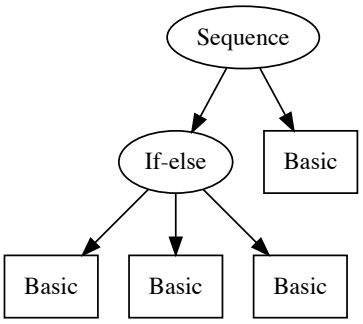


Figure 4.3: The CFG of Figure 4.2 represented as tree after reconstruction.

An example of the reconstruction can be seen in Figure 4.2, with its interpretation in tree form on Figure 4.3. On the left of the Figure, the basic blocks 0, 1 and 2 forming an *If-else* structure have been reduced. Then, in the center, the newly created *If-else* structure and basic block 3 are reduced into a *Sequence*, thus terminating the algorithm, as shown on the right. This analysis reduces CFG portions to the following structural types: *Sequence*, *Self-loop*, *If-then*, *If-else*, *While*, *Do-While*, *Switch*, *Optimized If*.

Algorithm 1 presents the reconstruction procedure. In the *Reconstruction* function, the first three lines are used to modify Natural Loops, loops with

Algorithm 1: Reconstruction main loop

```
inputs : A CFG G = (V, E)
output: A single node representing the nested structures. Nil if the
        procedure failed
Function Reconstruction(G):
    sccs ← FindStronglyConnectedComponent(G);
    foreach scc ∈ sccs do
        TransformNaturalLoop(G, scc);
    while |G| > 1 do
        list ← postorderDFS(G);
        while list ≠ ∅ do
            node ← pop(list);
            R ← reduce(node);
            if |R| > 0 then
                Let r be a single node containing R G ← (G \ R) ∪ r;
                foreach (v_i, v_j) ∈ E ∧ v_i ∉ R do
                    if v_j ∈ R then
                        v_j ← r;
                break;
        if not modified G then
            return nil;
    return G;
```

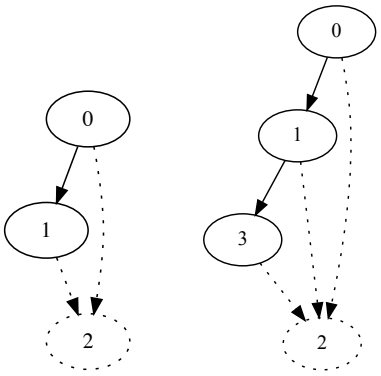
more than one exit, and are explained in Section 4.3.4. In the remaining part, we can see that the outer `while` is the iterative step, running until the graph is composed solely by one node. For each iterative step, a post-order depth-first visit is performed. The reason for the post-order visit lies in the fact that while processing a node, every possible descendant of it has already been given the possibility to be reduced beforehand. Then, the inner `while` attempts to reduce each node of the post-order visit to a known structure by calling the *reduce* function on each node.

The *reduce* function takes a node as input and returns a set, called $\mathcal{R}$, containing the original nodes composing a particular structure. Reductions are tried in the following order: *Self loop*, *While* and *Do-while* altogether, *If-then*, *If-else*, *Sequence*, *Switch* and *Optimized If*. If the reduction succeeds, firstly every node composing the region is removed from the CFG and a new structural node named r is added. Then, for each edge $e_{i,j} = (v_i, v_j)$ where v_i does not belong to the reduced region, the target v_j mapping to a component of the region itself is remapped to r. The edge becomes $e = (v_i, r)$, effectively replacing a region with a single node.

Lastly, the inner loop is terminated, given that the post-order visit is now invalidated having the CFG been modified. The outer `while` generates a new post order visit and the process is repeated until a single node is left. If, instead, the list is processed in its entirety without any modification to the CFG the procedure terminates with a failure state. We now explain how every region has

been reduced, excluding self-loops which are trivial.

If-then resolution



(a) If-then (b) Short circuit If-then

Figure 4.4: If-then and Short circuit If-then. A sequence of nodes composed of conditional jumps with the target being fixed (2) can be considered as If-then structure.

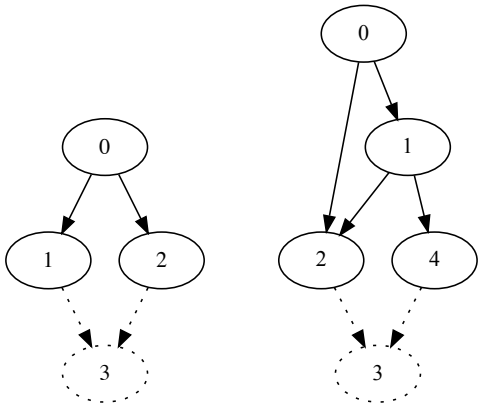
We can see in Figure 4.4, on the left, the representation of a minimal *If-then* structure as it would appear on a CFG. Solid lines represent the nodes that will be reduced. Note that in case an *If-then* structure has more complex logic in the *then* node, this has already been reduced in previous iterations. In order to reduce a node as an *If-then* region, the following conditions must hold:

1. The current node has two children, the *then* node and the *next* node.

2. The *then* node is a node with a single predecessor and a single successor.

3. The successor of the *then* node is the *next* node.

If all these conditions are satisfied, the current node and the *then* node are transformed into an *If-then* region, with the *next* node as its successor. In Figure 4.4, on the right, we can instead see the CFG for an *If-then* with a short-circuit evaluation. Recall that short-circuit evaluation is the semantic of a boolean expression where some arguments are not evaluated if the truth value of the expression has already been established, represented in this example by the edge $e_{0,3}$. This kind of *If-then* is automatically resolved iteratively using the previously defined rules, generating a nested *If-then* region where the *then* node is another *If-then* region. In our implementation we flattened these nested nodes into a single *If-then* keeping the first $n-1$ nodes as the various conditions and the last node as the actual *then*. Moreover, the number of conditions are not considered in the comparison phase, so a short circuit *If-then* is reconstructed identical to a normal one.



(a) If-else (b) Short circuit If-else

Figure 4.5: If-else and Short circuit If-else. Two different paths sharing a node (0) and a sink (3) can be reduced as an If-else structure. In principle, the two paths must be disjoint, except in the case where every node of a path connects to the same node of the other, indicating a short circuit If-else.

If-else resolution

In Figure 4.5 on the left, the minimal structure of an *If-else* is presented. Exactly like the *If-then* resolution, the following set of conditions must be verified in order to reduce a node as an *If-else* node:

1. The current node has two children, the *then* node and the *else* node.
2. The *then* node has a single predecessor and a single successor.
3. The *else* node has a single predecessor and a single successor.
4. The successor of the *then* node is the same of the *else* node.

If all these conditions are satisfied, the current node, *then*, and the *else* node are merged into an *If-else* region with the successor of *then* as successor. It is important to note that deciding which node is *then* and which node is *else* is not trivial and has some consequences, however, this will be discussed in Section 4.3.5 when dealing with the comparison. Additionally, unlike the *If-then* structure, the short-circuit version of the *If-else* presented on the right in Figure 4.5 is not resolved automatically using the previous rules, given that the rule number 3 is violated by the short-circuit. It is thus required to implement a routine that tries to descend into the *then* subtree, asserting that every successor is either another *then* node or the *else* node. The new rule 3 instead will ensure that every predecessor of the *else* node is either the current node or one of the *then* nodes. Also in this case, short-circuit version and normal one are treated equally in the comparison phase.

An interesting case can be seen in Figure 4.6. The figure represents a particular *If-else* where the *then* branch can arbitrarily jump to the *else* one. Although

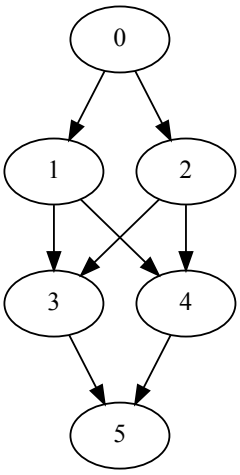


Figure 4.6: Optimized If structure. This structure is similar to an If-else, but the two paths are not disjoint.

impossible to realize in code without `goto`s, this construct is often emitted by the compiler to handle exit conditions and failures. Unfortunately, a compiler can generate arbitrarily long *then* and *else* branches, resulting in endless different between-branch jump possibilities. This means that either we ignore some of these structures, increasing the failure rate for the structural analysis, or we use the same label for structures that are not the same, sacrificing comparison correctness. In our study, we chose the first approach, as we further discuss in Section 4.4.1. In particular, we detected only the most basic compiler-generated patterns like the one in Figure 4.6 and labeled them as *Optimized If*.

Switch resolution

A more complex variant of the *If-else* is the *Switch*. These structures, after compilation, are usually implemented with a jump table and thus require particular care to be detected statically. Fortunately, several algorithms for recovering statically these tables exist in literature, and the task is thus delegated to the disassembler [15].

In the CFG, switch thus appear as in Figure 4.7 and can be easily detected as its root node has a number of edges higher than 2. If all these nodes point to the same successor, they are reduced as *Switch*, otherwise the successors should be refined first.

Sequence resolution

Sequences do not present particular cases, so it is sufficient that the current node and its successor satisfy these conditions to merge them into a 2-nodes sequence:

1. current node has a single exit
2. successor has a single entry and a single exit

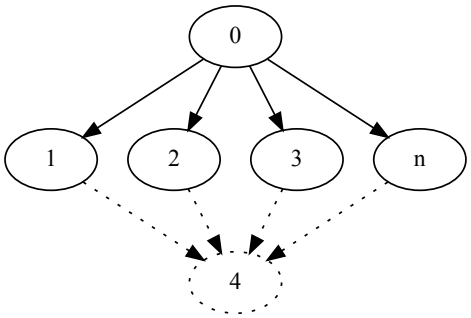
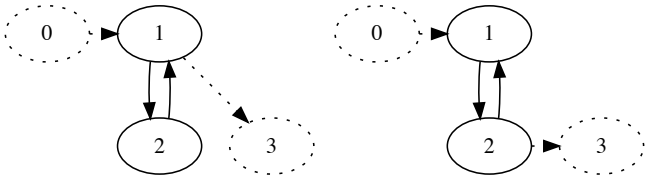


Figure 4.7: Switch structure. Any sequence of disjoint paths from a node (0) to a sink (4) with more than two paths can be reduced to a Switch structure.

Additionally, if the current node or the successor is already a sequence, the newly created one is not nested but appended to the existing ones.

While and Do-While resolution

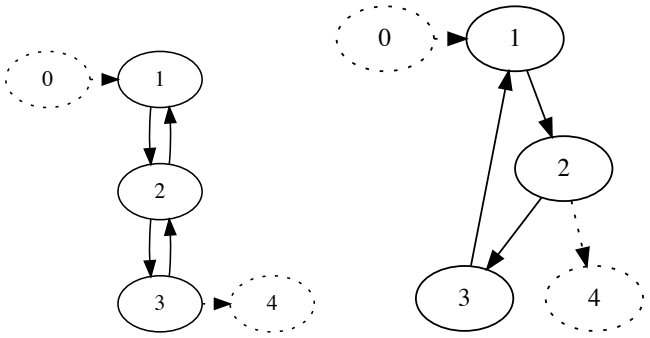


(a) While (b) Do-While

Figure 4.8: While and Do-While loops structures. A path of length 2 starting from and ending in the same node can be considered a loop. If the loop entry and exit are from the same node, this loop can be reduced as a While structure, otherwise as a Do-While structure.

While and *Do-While* loops are conceptually similar, as can be seen in Figure 4.8 with the former on the left and the latter on the right, the only difference being the exit node. While these loops can be found quite easily using the Tarjan's Strongly Connected Component (SCC) algorithm, particular care must be taken when dealing with nested loops, where the head of a loop is also the tail of the other one, as shown in Figure 4.9 on the left. Specifically, the SCC cannot be blindly used to determine the exit of a loop, given that in case of nested loops both the inner and outer loop have the same SCC. Additionally, unlike *While* loops, *Do-While* constructs can have a minimal form composed of three nodes that can not be reduced to two nodes using *Sequence* rules, and requires an ad-hoc reduction. This form is shown in Figure 4.9 on the right.

Transform Natural loop



(a) Nested Do-While (b) Do-While with 3 nodes

Figure 4.9: Nested Do-While and minimal 3-nodes Do-While. These are two particular cases of Do-While structure that require additional care: (a) for the correct identification of loop entry and exit and (b) because the combination of Sequence and Do-While detection rules can not detect this particular case.

In case the original source code had a loop with *break* or *return* statements, in the CFG this loop could potentially have multiple exits. In order to effectively reduce a loop to a minimal 2-nodes variant, a single exit is required and the first three lines in Algorithm 1 are used to detect and remove these edges from the CFG. In fact, our analysis aims at reconstructing only the original structure and thus these loop-breaking statements are redundant. Given a CFG $\mathcal{G} = (V, E)$ and a particular set of exit edges $EX(k)$ from a SCC k as

$$\begin{aligned} EX(k) = \{\forall e_{i,j} \in E \mid v_i \in V \wedge \\ v_j \in V \wedge \\ SCC(v_i) = k \wedge \\ SCC(v_j) \neq k\} \end{aligned}$$

where $SCC(k)$ is the SCC index, the natural loop resolution is run for the SCC if $|EX(k)| > 1$, meaning that more than one exit edge exists for that SCC. From here on, given $e_{i,j} = (v_i, v_j)$ such that $v_i \in EX(k)$ is an exit edge, we call exit node the node v_i and target the node v_j. Asserted that the number of exit nodes is always greater than one, two possible types of natural loop exist:

Single Target
This type of natural loop is generated by *break* statements without any kind of cleanup, like `if(condition)break;`. If an exit node has a higher number of predecessors than the others it is kept as the real exit, with the assumption that this loop would be a *While* loop. Otherwise, the first exit node encountered in a Depth-First Search (DFS) is kept. Although this could be the wrong one, recall that we are focusing on consistency between different programs rather than decompilation correctness. Every edge not belonging to this exit is removed from the CFG.

Multiple Targets
In addition to the single target, *if* constructs with additional logic before the *break* keyword generate a different exit target. Moreover, also *return*

statements generate an additional target. While the latter can be easily spotted by searching a jump directly to the function exit, we decided to keep only the target having the longest path $d(v_j, v_k)$ where v_j is the target, v_k the function return node and $d(v_j, v_k)$ the minimum distance between the two nodes. Every edge pointing to the wrong target is removed, then the natural loop is resolved in the same way as the Single Target one. Note that unlike the Single Target, this approach could generate orphan nodes.

After the removal of these edges from the CFG, the loop can be reduced to the minimal form of the *While* or the *Do-While* presented in Section 4.3.4 with the iterative step of Algorithm 1.

4.3.5 Comparison

After completing the reconstruction step, the output graph can be represented as a tree of structures, similar to the example shown in Figure 4.10. In order to

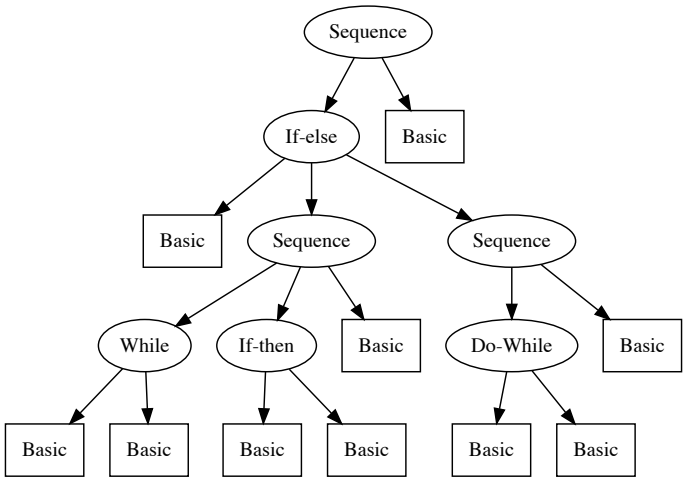


Figure 4.10: Tree resulting from the reconstruction.

efficiently compare all the possible subtrees generated by the reconstruction of every function, we use LSH. Specifically, we choose a hash function such that the collisions between trees with the same structure are maximized. It is of vital importance to carefully design this function: we want to ensure a hash collision for structures that are *similar*, not *identical*. In fact, the functionality and high level structure of two functions may be the same, but they may use basic blocks or jumps with different offsets. For this reason we hash only the node type, avoiding the basic block offsets, recursively.

In our implementation we used as hashing function f_h the public domain FNV-1a [28] and implemented the function $h(t)$, in order to hash a subtree, shown in Equation 4.1.

$$\begin{aligned} h(t) = & f_h(type(t)) \circ h(c_0) \circ h(c_1) \cdots \circ h(c_n) \\ & \forall c_i \in C_t \\ & \text{where } C_t \text{ is the set containing the children of } t \end{aligned} \tag{4.1}$$

We can see that the hash of a node t is calculated by composing the hash of the current node type with the hash of all its children. The node type is a unique value assigned to each structure type. If two nodes t_1 and t_2 are both the same structure (e.g. Sequence, If-Else, etc.) $type(t_1)$ and $type(t_2)$ have the same value.

We apply this function to every reconstructed function of each binary. If the tree depth of the current node is higher than a threshold θ, it is written into a hash table data structure, with $h(t)$ as key and t as value. Upon iterating all the keys of this hash table, if a key contains more than one value, the key itself represents a clone class and all its values represent the various clone snippets belonging to this class.

Considering that f_h is a constant time operation, the complexity of $h(root)$, where $root$ is the root of each reconstructed tree, is linear in the number of tree nodes. According to our reconstruction rules, a reconstructed tree can not have more nodes than basic blocks, implying that our approach has a comparison step with linear complexity in the total number of basic blocks retrieved from all functions.

We can see that hashing is not based on the statements contained inside a specific structure, but only on the structural shape itself. This enables the comparison between different architectures at the cost of increased false positives ratio, in case two pieces of code are structurally similar but perform different actions. Being thus the hashing distance based solely on the structural shape, we use the threshold θ as variable to control the ratio between false positives and negatives: a lower θ will match structures with few nested nodes that may be shared by code performing different operations, while a higher θ will require a more unique structure but may miss some matches.

We do, however, account for children's order when calculating the hash; it is thus easy to see why in Section 4.3.4 it was important to clearly determine the *then* node and *else* node deterministically.

4.3.6 Semantic Analysis

The comparison presented in Section 4.3.5 is sufficient for finding clones, however, its calculations are entirely based on the structure of binary code. This can present a situation where two binary functions are reported as clones because they share the same structure, albeit having different opcodes thus resulting in different functionality. For this reason, in this section we present an additional refinement step to our approach, that should run on the comparison results, in order to filter some false positives that happen to have the same structure, but different opcodes.

This refinement is based on the idea that to every instruction we can assign a number, representing the amount of times that particular opcode appears in the function over the total number of opcodes in the function, thus creating a frequency vector. Then, a function to measure the similarity between two vectors, can be used to measure the degree of similarity between two frequency vectors. If this value is greater than a threshold, the two basic blocks can be considered the same. We run this comparison on the (unordered) list of opcodes composing the basic blocks of two matching reconstructed tree, similar to the one in Figure 4.10. Moreover we do not consider the opcode parameters: doing so would overspecialize our comparison and match only identical functions. In

our implementation we used cosine similarity as similarity function: given two frequency vectors A and B, their similarity is given by $S_{AB} = \frac{A \cdot B}{\|A\| \|B\|}$

Naturally, if the architecture between the potential clones is different, their opcodes would not match and their similarity will always be zero. For this reason, in case of different architectures, instead of using the opcodes directly we assign them to a "family of operation" and determine the frequency of each opcode family instead of the opcode itself. For example, the `jmp` opcode for `x86_64` and `b` opcode for `aarch64` are both assigned to the `JUMP` family, and the similarity calculated on the frequency of the `JUMP` family, instead of the frequency of `jmp` or `b`. In total we divided opcodes into 30 different families. For space reason, we do not report them here, but the full list can be found in the repository referenced in Section 4.7.

Note that, unlike the comparison approach presented in Section 4.3.5, the semantic analysis presented in this section is performed by comparing two potential clones at a time. This means that, to compare n functions, $\mathcal{O}(n^2)$ operations have to be performed, compared to the $\mathcal{O}(n)$ required by the structural analysis. This explains why, in our approach, the semantic analysis is used only after gathering a list of potential clones from the structural one, and its higher run time is confirmed by the experimental results we provide in Section 4.4.

4.4 Evaluation

We implemented our analysis in Rust under Linux, however, we are able to analyze the binaries for multiple operating systems and 20 different architectures. The implementation of our approach is named BinCC, and is openly available on GitHub [1]. In this evaluation, we target mainly the `x86_64` and `aarch64` architectures, being the dominant architecture for the Desktop and Mobile market respectively.

As mentioned in Section 4.3.2 we depend on an external disassembler to retrieve the list of functions and statements. In our implementation we used the open source tool *radare2* [2], version 5.7.4, whereas every other step is independent of external tools. Despite using *radare2*, we do not depend exclusively on it: in fact any disassembler could be used, provided that a list of statements like the one in Section 4.3.2 is returned as a string.

In order to evaluate the effectiveness of our approach, we want to address the following research questions:

- **RQ1**$_{completeness}$: How many functions are successfully converted to a single node representation?
- **RQ2**$_{correctness}$: How precise is our tool at detecting function clones across different architectures?
- **RQ3**$_{use\text{-}case}$: Is our tool able to find function reuse from libraries in a real-case application?
- **RQ4**$_{performance}$: How performant is our tool, varying the input executable size?

[1] `http://github.com/davidepi/bincc`
[2] `https://rada.re/r/`

The first research question, **RQ1**, is meant to determine the amount of functions that can actually be reconstructed, across different architectures and optimization levels. This can be seen as a measure of the goodness of our structural analysis algorithm, given that functions that are not reconstructed can not be compared with hashing. The second research question, **RQ2**, is meant to measure the precision of our detection when checking for clones in binaries within the same architecture or different architectures. Additionally, the threshold parameter θ for the LSH function defined in Section 4.3.5 is evaluated here and our approach is compared against existing state-of-the-art works. The third research question, **RQ3**, is meant to be a use-case evaluation, measuring the precision at detecting function reuse with binaries and libraries resembling more a real-case scenario. Finally, the last research question, **RQ4**, is used to measure the time required by the analysis and the scalability of our approach.

In order to ensure an evaluation as close as possible to the final use-case, while still guaranteeing a correct evaluation, in this study a different dataset was used for each research question. A summary of the various datasets can be found in Table 4.1, while details on their creation are listed in the respective Research Questions.

RQ	Binaries	Functions	Stripped	Optimization
RQ1	11211	23953469	yes	O0, O2, Os
RQ2	216	40907	no	O2
RQ3	800	276485	Unknown	Unknown
RQ4	1934	1434790	yes	O2

Table 4.1: Statistics and features of the dataset used for each Research Question. Because RQ3 uses publicly available real-case binaries, the optimization level and strip status are not known.

4.4.1 RQ1: Completeness

In order to answer RQ1 we used as dataset a collection of binaries publicly available on Zenodo [74]. These binaries come from multiple open source projects of different scopes and are compiled using different optimization levels. As reported in Table 4.1, these binaries are stripped. We used binaries compiled using GCC for the `x86_64` and `aarch64` architectures, divided into `O0`, `O2`, `Os` optimization levels.

For the evaluation, we ran the reconstruction on every function and checked if the output was effectively a single node. Input functions already composed of a single node are not considered in this evaluation.

Figure 4.11 shows the percentage of correctly reconstructed functions over the total, for a given CFG input size. The immediate result we can note is that the higher the optimization level, the lower the chance our algorithm is able to correctly perform the reconstruction. This is somewhat expected, as higher optimization levels introduce additional compilation patterns that are more difficult to map to the structures we defined in Section 4.3.4. In particular, we analyzed the failed reconstructions and determined that most failures are due to variations of the *Optimized If* defined in Section 4.3.4. With high optimiza-

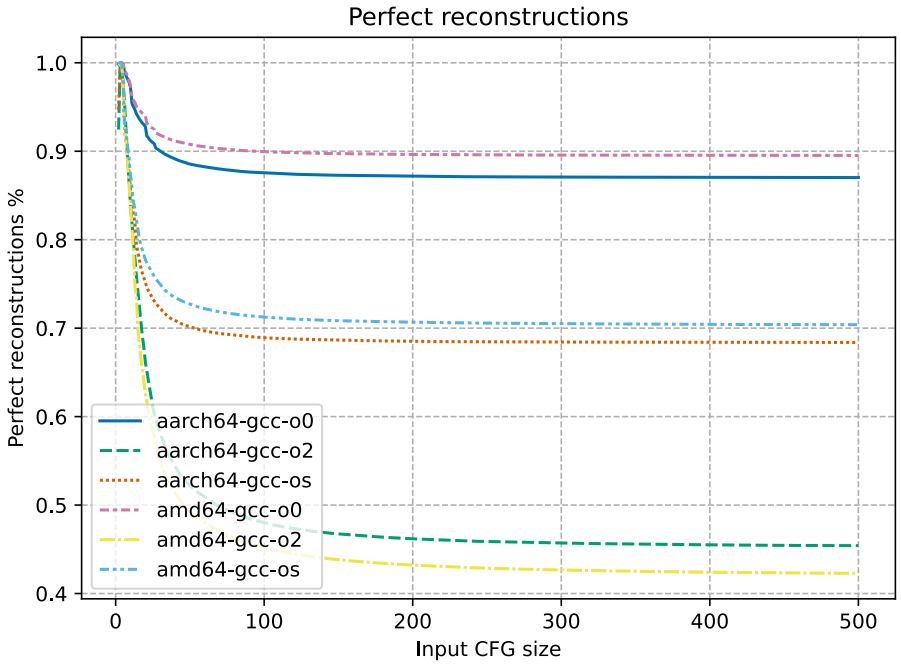


Figure 4.11: Perfectly reconstructed functions on stripped binaries, excluding trivial ones.

tion levels, the compiler enables jumps between the various if-else branches, generating complex branching structures that cannot be easily categorized.

Differences between the two architectures instead are minor: the reconstruction in `aarch64` is slightly less precise compared to `x86_64` in both `O0` and `Os`, but not in `O2`. We could not determine the reason of this difference by looking at the functions, and we can only assume it is due to the disassembler being more proficient with `x86_64`.

Figure 4.12 and Figure 4.13 show the reduction percentage in all functions and the reduction percentage in failed reconstructions only. These are meant to represent how much smaller a reduced function is, compared to the original counterpart, considering the number of nodes before and after a reduction. This is very important, as in case of failed reductions the comparison may still be possible if the number of nodes is sufficiently small.

In fact, the maximum amount of nodes we found is 4202 for the `aarch64` architecture and 9424 for `x86_64` with an average of 40 nodes per function. In these cases, the CFG analysis without our reconstruction is absolutely intractable with an $\mathcal{O}(2^n)$ algorithm. However, as we can see from Figure 4.13 even in case of failed reconstructions our analysis reduces the number of nodes by 30-40% in any optimization level, and may allow tractability in the average case, going from 2^{40} comparisons to less than 2^{30}.

Given the results, we can answer RQ1 as follows:

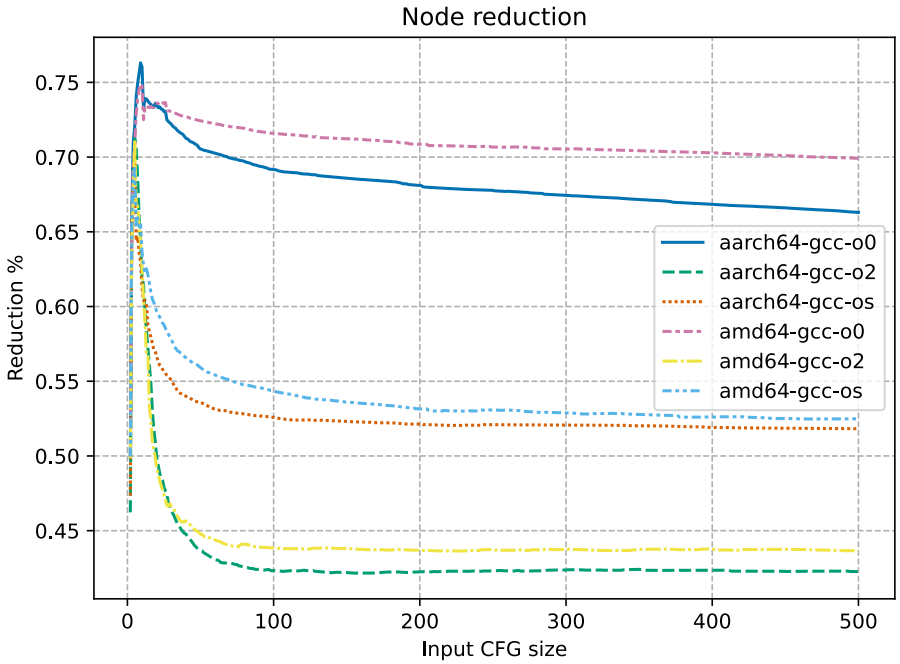


Figure 4.12: Amount of reduced nodes based on the original CFG length.

The average percentage of correctly reconstructed functions is around 90% for the O0 optimization level, 70% for Os and 45% for O2. Even in the case of failed reconstructions, the number of nodes is reduced by 35-45%, and enables tractability in the average case.

4.4.2 RQ2: Correctness

For the second research question, we want to analyze how accurate is the detection rate of cloned functions in our approach, while also estimating the change in accuracy varying the approach parameters. Specifically, we first analyze the clone detection using structural analysis only, then semantic analysis only and, finally, a combination of the two. We close this analysis by comparing our tool with two existing state-of-the-art products: BinDiff [27] and DeepBinDiff [21].

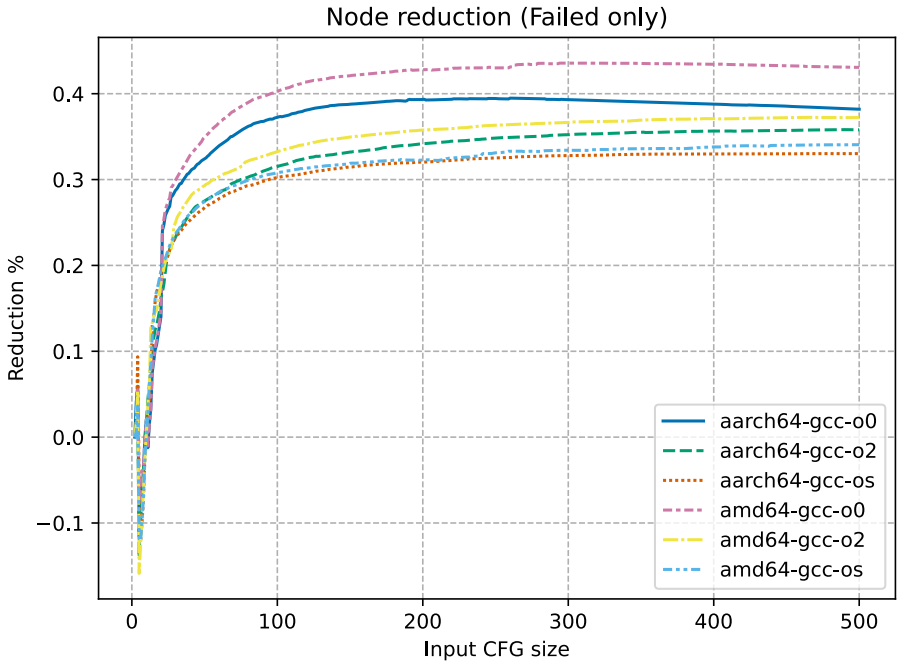


Figure 4.13: Amount of reduced nodes, considering only failed reconstructions.

Table 4.2: True positives, False Positives, False Negatives, Precision and Recall in clone detection using structural analysis alone, varying the minimum tree depth threshold θ. Results obtained comparing all the coreutils binaries together.

	Same architecture (x86_64)					Cross architecture (x86_64, aarch64)				
θ	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall
2	3337	3774	48	0.4693	0.9858	4826	8966	2406	0.3499	0.6673
3	2150	581	309	0.7873	0.8743	3733	2313	2101	0.6174	0.6399
4	1540	291	162	0.8411	0.9048	2888	1213	1452	0.7042	0.6654
5	687	68	129	0.9099	0.8419	1167	579	833	0.6684	0.5835
6	375	34	91	0.9169	0.8047	663	293	576	0.6935	0.5351

Table 4.3: True positives, False Positives, False Negatives, Precision and Recall in clone detection using semantic analysis alone, varying the minimum cosine similarity threshold. Results obtained comparing all the coreutils binaries together.

	Same architecture (x86_64)					Cross architecture (x86_64, aarch64)				
Min.Cosine Sim.	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall
0.95	93287	252978	7360	0.2694	0.9269	218512	867087	209742	0.2013	0.5102
0.98	12256	14320	2090	0.4612	0.8543	39171	41092	41092	0.3710	0.4880
0.99	7928	2648	307	0.7496	0.9627	16626	11555	17958	0.5900	0.4807
0.999	8922	1163	3724	0.8847	0.7055	17406	3701	28343	0.8247	0.3805

For this Research Question we compiled GNU coreutils [3] tagged `v.9.31` on Linux for the `x86_64` and `aarch64` architectures with optimization level `O2`. We chose *coreutils* for the reason that, being composed by several programs in a single codebase, we expect to find more clones than by choosing two completely random binaries.

In the "same architecture" analysis we ran our tool comparing all functions belonging to the 108 *coreutils* binaries built for `x86_64`, reporting both intra-project and inter-project clones. Similarly for the "cross architecture" analysis we added also the functions belonging to the same binaries built for `aarch64`. As highlighted in Table 4.1, this time we did not strip the binaries: in fact, using non-stripped binaries allows us to investigate false negatives by matching the various function names. Note that our approach supports also sub-function granularity, as presented in Section 4.3.5, however, to reduce the amount of manual analysis to be done, we limit the study to function granularity.

Unlike most tools in literature [27, 64, 68], ours compares not only pairs, but entire sets of similar functions. For this reason, the metrics were defined as follows: for each reported clone class A, the most similar clone class B was retrieved from the ground truth using Jaccard Similarity. With these two sets, True Positives (TP) are defined as $\#(A \cap B)$, False Positives (FP) as $\#(A \setminus B)$ and False Negatives (FN) as $\#(B \setminus A)$. Additionally, we added to the False Negatives count the elements of each unmatched clone class in the ground truth. The ground truth set was built with a combination of similar function name matching, existing state-of-the-art tools and manual analysis.

Table 4.2 shows the results using Structural Analysis only. When operating within the same architecture, for sufficiently complex functions (higher θ), this approach is capable of reaching a precision up to 91%. In this type of analysis, the False Positives are exclusively determined by functions with the same structure but different opcodes. For example, in the binary `pr`, the functions `hold_file` and `tzfree` are reported as clones despite being composed of completely different opcodes. This happens because their structure is essentially the same. Naturally, with a low threshold these false positives increase in number, but even with a low value as $\theta = 3$ we are capable of reaching a fairly high precision of almost 80%.

The same cannot be said for the cross architectural Structural Analysis. In this case the precision never goes above 70%. The main problem we identified by analyzing the results is the fact that our approach can correctly report big clone classes in their respective architectures but these classes have slightly different structures, mostly sequences with a different number of basic blocks, that result in different structural hashes. For this reason these clone classes are not merged into a single cross-architectural clone class. Naturally, we report these clones as False Positives/Negatives: despite being correctly identified as clones we are interested in cross-architectural clones only.

Table 4.3, instead, shows the results using Semantic Analysis only. Unlike similar approaches in source code clone detection [91], the minimum similarity threshold for our frequency vector is much higher. Even in CISC architectures such as `x86_64` that potentially can use thousands of opcodes, in practice most functions use a small subset of common opcodes. This problem is exacerbated in the cross architectural detection by our "opcode family" described in Sec-

[3] `https://www.gnu.org/software/coreutils/`

tion 4.3.6 that further reduces variance between functions, resulting in 80% precision with functions that have a similarity higher than 0.999. In addition to this problem, the semantic analysis has complexity $\mathcal{O}(n^2)$ in the number of functions, compared to the $\mathcal{O}(n)$ of the structural analysis, limiting its scalability.

Table 4.4: Precision and Recall in clone detection within the same architecture, using structural analysis and semantic analysis combined, varying their input thresholds.

	Same architecture (x86_64) Min. Cosine Sim.		
θ	0.98	0.99	0.999
2	0.8494 / 0.9746	0.8702 / 0.9708	0.8752 / 0.7690
3	0.9148 / 0.9114	0.9280 / 0.9084	0.9306 / 0.9235
4	0.9178 / 0.9143	0.9345 / 0.9145	0.9363 / 0.9039
5	0.9627 / 0.8593	0.9876 / 0.8646	0.9901 / 0.7964
6	0.9658 / 0.8905	0.9888 / 0.7599	0.9907 / 0.8004

While discussing Table 4.2 we explained how most of the False Positives in the structural analysis are determined by clones with the same structure but different opcodes. For this reason, by using the results of the structural analysis and applying semantic analysis on them, we overcome this problem and obtain the results shown in Table 4.4. The Table reports only precision and recall, but we can clearly see how dramatic the improvement in precision is: even for simpler functions composed of three nested structures both precision and recall are now above 91%. For complex functions the precision can reach up to 99% with our best result being 474 True Positives and only 4 False Positives in one configuration. The remaining false positives are due to the granularity of our approach returning block-level clones but being treated as function-level clones by our experimental settings, in particular for $\theta = 2$.

Table 4.5: Precision and Recall in clone detection across different architectures, using structural analysis and semantic analysis combined, varying their input thresholds.

	Cross architecture (x86_64, aarch64) Min. Cosine Sim.		
θ	0.98	0.99	0.999
2	0.7311 / 0.5071	0.7334 / 0.5087	0.7516 / 0.4645
3	0.7241 / 0.5402	0.7368 / 0.5476	0.7449 / 0.5250
4	0.7691 / 0.5504	0.7679 / 0.5647	0.7720 / 0.5402
5	0.7204 / 0.5398	0.7240 / 0.5145	0.7320 / 0.5263
6	0.7410 / 0.4922	0.7430 / 0.4908	0.7500 / 0.4888

This solution, however, does not work for cross architectural clones, as can be seen in Table 4.5. Here the main problem lies in the structural analysis not correctly mixing the various clone classes and this problem can not be fixed by the semantic analysis.

Table 4.6: Time required to perform the structural analysis, semantic analysis and combined analysis in both same architecture and cross architecture using $\theta = 3$ and min similarity of 0.99. Results obtained analyzing all coreutils binaries together.

Analysis	Time same arch.(μs)	Time cross arch.(μs)
Structural only	5066	8922
Semantic only	16111921	46399417
Combined	543672	497643

Finally, Table 4.6 shows the time required for each approach. We can clearly see how the semantic analysis is 1000 times slower than the structural one, while the combination of the two still manages to reach an acceptable time, due to the initial filtering performed by the structural analysis.

Table 4.7: Precision and number of detected clones in pairwise function detection of several state-of-the-art approaches. The listed programs have been compared against `du`.

bin	BinCC (ours)	BinDiff	DeepBinDiff
dir	0.9593 (172)	0.9333 (105)	0.9368 (95)
ls	0.9593 (172)	0.9333 (105)	0.9167 (96)
mv	0.9704 (169)	0.9739 (115)	0.9245 (106)
cp	0.9652 (144)	0.9783 (92)	0.8295 (88)
sort	0.9923 (131)	0.9670 (91)	0.9157 (83)
du	0.9574 (188)	0.9937 (159)	0.9933 (150)
csplit	0.9574 (94)	0.9254 (67)	0.9194 (62)
expr	0.9489 (98)	0.9677 (62)	0.9062 (64)
nl	0.9444 (90)	0.9672 (61)	0.9828 (58)
ptx	0.9266 (109)	0.9444 (72)	0.9254 (67)
split	0.9375 (96)	0.9538 (65)	0.9831 (59)
mean	0.9562	0.9580	0.9303

In Table 4.7 we compare our approach against BinDiff [27], a commercial tool for binary diffing, and DeepBinDiff [21], a research tool performing the same task using deep learning. Given that both BinDiff and DeepBinDiff support only pairwise comparison, and none of them support cross-architectural comparison, we compared eleven binaries in the coreutils package of different size against `du`. In particular we used our tool combining structural and semantic analysis with a low threshold in order to match as many clones as possible ($\theta = 2$). For BinDiff and DeepBinDiff, instead, we report as clone a pair of functions having at least half their basic blocks matching. Using higher threshold for all tools would result in perfect precision, but would miss most function pairs.

We can see that BinCC, our tool, even with low thresholds, can emit more clones compared to the state-of-the-art. This is due to our tool being capable of emitting intra-project clones that, depending on the use case, may be desirable or not. The precision of the detected clones is similar with respect to the other state-of-the-art tools. We determined that our tool, similarly to DeepBinDiff,

Table 4.8: Time required to compare `du` with the binary listed in the column bin, in seconds. The size column contains the combined size of `du` and the target binary. Times have been counted from program invocation until program termination.

bin	size (KiB)	BinCC (ours)	DeepBinDiff
dir	1081	3.40s	1409s
ls	1081	3.18s	1432s
mv	1045	3.43s	1999s
cp	978	3.22s	1740s
sort	952	2.90s	1216s
du	921	2.80s	2043s
csplit	682	2.55s	659s
expr	673	2.44s	652s
nl	642	2.43s	527s
ptx	749	2.58s	771s
split	701	2.43s	702s

fails in cases where functions have identical implementation modulo a different function call. For example, they both detect `xcalloc` and `xmalloc` as clones, being these two functions different only in the allocation function used. The results obtained from DeepBinDiff are comparable with the results presented in its original paper [21], while for BinDiff we obtained slightly more accurate results compared to previous experiments [64].

A huge difference instead, can be noted in the speed. Table 4.8 shows a comparison between our tool and DeepBinDiff. BinDiff using Ghidra requires to manually disassemble the file, and for this reason it is not listed. The Table clearly shows that our tool is order of magnitudes faster than the competition, in some cases reaching difference of a factor 10^3 (e.g. the comparison `du-du`). In addition, DeepBinDiff can compare only two binaries together, requiring to perform the analysis for any combination of executables, whereas our tool can report the clones for all 108 coreutils binaries together in less than a minute.

For RQ2 we can thus conclude:

> *Even without using semantic analysis our approach is capable of reaching more than 90% precision if the function is complex enough. For simpler functions, this precision can be reached by combining the structural analysis with a semantic one, sacrificing some detection speed. Nonetheless, our tool is still order of magnitude faster than the competition and can achieve similar precision.*

4.4.3 RQ3: Use-case

After performing the evaluation on a controlled case with unstripped binaries in our possession, in this section we want to simulate a use-case by checking the detection rate on stripped-only, real-case binaries.

Given that after the strip phase every information about the function name is lost, building a ground truth set requires manually checking every function. For this reason, we adopted an approach similar to the one of Hemel et al. [37]:

we compiled several binaries with the default options and static linking. After that, we ran the comparison between the binary itself and the libraries reported to be used at link time. Additionally, we performed the check for a limited set of extra libraries unrelated to the project. In this case, being more of a use-case, we wanted to check more consistent functions and thus used higher thresholds. In the dataset summary of Table 4.1, we reported the optimization for this Research Question as "unknown": being this a real-case scenario, we used binaries without knowing the original compilation options. We used a value θ of 7 and a minimum number of nodes of 7 for the reconstruction and 15 for the CFG.

Table 4.9: Results of the library and function usage detection for busybox. The column "used" refers to functions in the library that were used inside busybox. "checked" refers to the amount of functions checked from that library.

library name	used	checked
libresolv.so.2	1	2
libm.so.6	2	40
libjpeg.so.8.1.2	0	8
libtiff.so.5.3.0	0	7
libpng16.so.16.34.0	0	8
libMagick++-6.Q16.so.7.0.0	0	4
libFLAC.so.8.3.0	0	1
libsamba-util.so.0.0.1	0	21
libXext.so.6.4.0	0	20

Table 4.9 shows the results for *busybox* tag 1_31_0 which uses *libm* and *libresolv*. Our tool correctly reported function reuse only in the libraries statically linked with the binary. We then ran the same analysis against those libraries compiled for `aarch64` and obtained similar results except for *libm* that was not detected: of the 40 functions checked in `x86_64`, 0 were analyzed in `aarch64`. The reason for this is that multiple functions were under the threshold and thus not analyzed, probably due to the smaller CFG in the second architecture. The number of analyzed functions for the other libraries were similar instead.

However, in order to not bias the evaluation given that we purposely used libraries unrelated to the project, we also reran the evaluation with 800 libraries usually present in an Ubuntu Linux install. In this specific case, we found 407 libraries that had function reuse and 393 that did not. We did not check if all the reports were false positives or negatives, because this would require a prohibitively high amount of time having to manually understand and analyze over 8000 functions in assembly. However, we found that most of the reported libraries are dependent upon *libc*, that is statically linked with the executable we analyzed.

For this reason, we plan to conduct a more controlled test, analyzing also the dependencies between the various libraries.

We can then answer RQ3 as follows:

By analyzing a stripped real-case binary executable our approach detects the libraries statically linked with it and also the dependencies of those libraries. Libraries unrelated with them are not detected

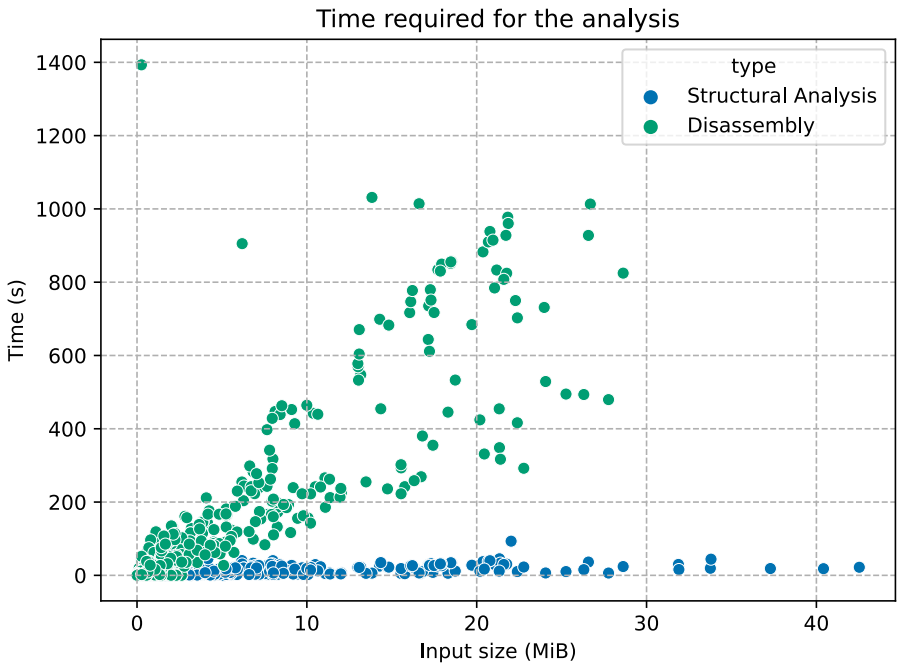


Figure 4.14: Time required for disassembly and reconstruction on x86_64 O2.

4.4.4 RQ4: Performance

The last research question evaluates the performance and the scalability of our analysis. In order to do so, we used a subset of the same dataset used for RQ1, limited to `x86_64` and `O2`. These times were recorded on a machine mounting an Intel Xeon E5-2620 @ 24x 2.5GHz with 64GB of RAM, limited to a single core and with hyper-threading disabled. Figure 4.14 shows the time required for the disassembly of the executables. We can observe that the time scales linearly with respect to the executable size, and it is in the order of hundredth of seconds.

However, by observing Figure 4.15, showing the time required for the structural analysis only, we can note that the time required is in the order of tenth of seconds. The entire procedure is thus completely dominated by the time required by the disassembly on which we depend, rather than the time required by our analysis. This can be seen clearly also in Figure 4.16 showing the composition of time required to perform the combined analysis. In the Figure we can note how the disassembly operation takes half the total time of the entire analysis, represented by the line. Moreover we can note that our approach working with a 158MiB executable requires the same time as existing approaches on a 400KiB, as shown in Table 4.8.

We experimented with the disassembler, and managed to perform a faster disassembly by analyzing only the function calls instead of performing a full binary analysis. However, the reconstruction accuracy suffered greatly: the fast disassembly took around the same time as our structural analysis, but the reconstruction accuracy presented in RQ1 dropped by a flat 20% in every

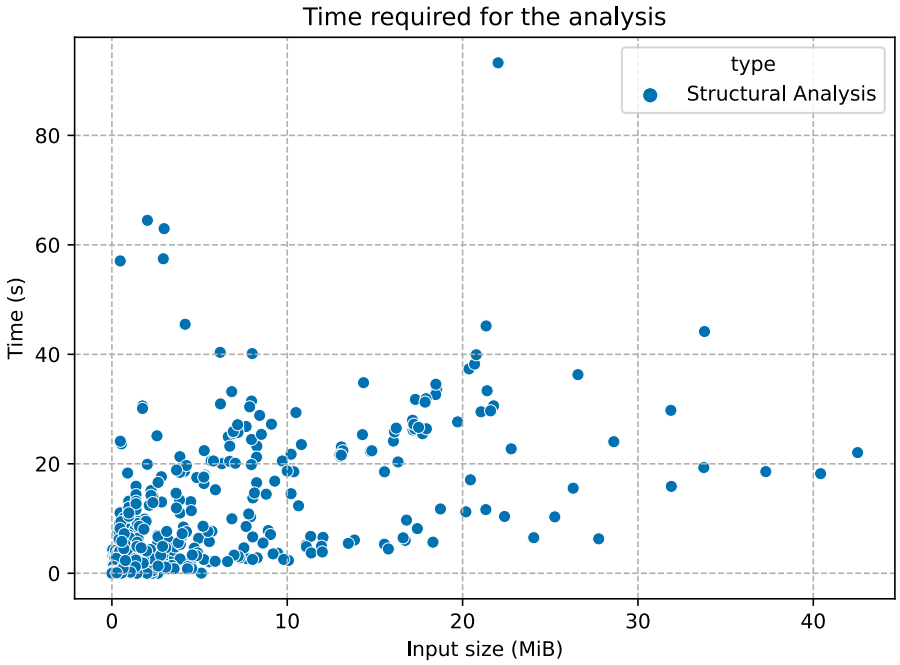


Figure 4.15: Time required for the CFG reconstruction step.

optimization level. For this reason, we decided not to report this analysis with the fast disassembly option.

For RQ4 we can thus conclude:

Our analysis scales linearly and is completely dominated by the time required to perform the disassembly.

4.5 Limitations and Threats to Validity

4.5.1 Predication

In some architectures, such as `ARM`, predication is used as an alternative to branching by converting flow dependency to data dependency. Predication works by associating instructions with a *predicate*, a boolean value, usually a CPU flag, used to indicate if the instruction is allowed to execute. If this boolean value does not hold at the time the instruction should be executed, that instruction does not modify the architectural state, with the advantage of not breaking the CPU pipeline. As an example, consider the following code:

```
0x410 cmp r0, 0
0x414 cmpne r1, 0
0x41C movne r0, 5
0x420 moveq r0, 6
0x424 bx lr
```

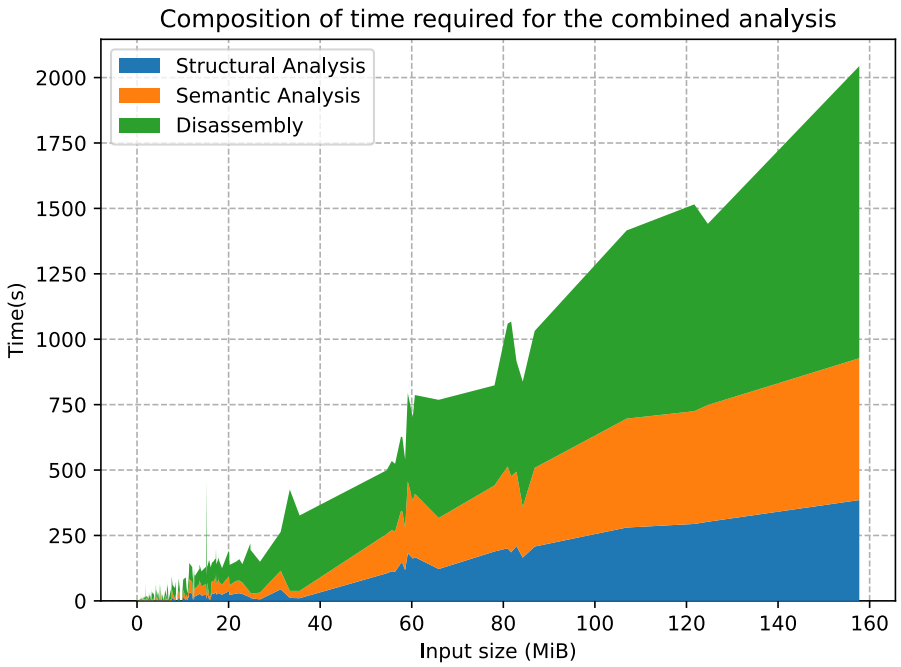


Figure 4.16: Time required for the combined analysis. The line represents the total amount of time required, while the colors represent the portion of time taken by each analysis step.

The first instruction executes a comparison and sets the CPU flags according to the result. If the flags are set as true, the second and third instructions do nothing, and the fourth is executed, otherwise the second is executed which changes the flags values again for when the third will be executed. The problem with predication thus lies in the fact that a CFG reconstructed just by looking at jumps will miss the flow hidden beneath these instructions. In our implementation we decided to ignore predication given that only two mnemonics, namely `CMOV` and `SETcc`, supports these operations in `x86_64`, while in `aarch64` these are mostly deprecated given the recent advancements with branch predictors [35].

4.5.2 Mismatched compiler configurations

In addition to the results showed in Section 4.4.2, we briefly analyzed potential clones between binaries compiled with different compilers or optimization settings.

Results show that our algorithm fails at finding potential clones when the compiler or the optimization flags are different: these results are due to the fact that code is rearranged during the data flow analysis performed by the compiler, and thus the output dramatically changes depending on the analyses performed at compilation time.

This was originally highlighted by Sæbjørnsen et al. [89] and we can confirm that not only the compiled binary is semantically different at various optimization levels but also structurally different, and we expect every CFG-based

approach to suffer from this limitation. In order to overcome this problem, Chapter 5 is dedicated to preventively detecting if there is a mismatch between the executables' optimization levels, to known if the results from binary clone detection will be reliable or not.

4.5.3 Disassembler dependency

In this work, the starting point of our analysis is the disassembler. We relied on an external tool as there is no novelty in implementing our own and deemed the process as out-of-scope for this work. However, this means the goodness of our structural analysis is dependent on the disassembly quality: using a different disassembler may result in a different qualitative result in terms of structural analysis ability and clone detection precision. We highlighted this fact in Section 4.4.4 by citing the fast analysis: using a faster but less accurate disassembly resulted in a flat drop of 20% accuracy in the structural analysis alone.

4.6 Conclusion

In this chapter, we presented our approach for finding function reuse and clones using a high-level refined CFG. We implemented this analysis and conducted several tests in 24 million functions in the `x86_64` and `aarch64` architectures.

Results show that our analysis is faster than existing approaches, and the time required is dominated by the disassembly step on which it depends. Our technique can be applied without any training step in 20 different architectures, analyzing any given number of executables at the same time.

When performing the comparison, our work showed to be capable of detecting function clones with precision ranging from 91% to 99% when comparing binaries from the same architecture and 75% when comparing binaries in different architectures. We showed that this precision is enough to detect the presence of functions coming from a particular library, that partially answers our motivation of detecting the presence and propagation of a vulnerable function in compiled software. However, more studies are required to determine if this can work efficiently also at basic block granularity.

We also showed that the weak point of binary clone detection in general, as determined previously by Sæbjørnsen et al. [89] is the presence of different compilers and optimization levels in the analyzed binaries. For this reason, Chapter 5 is focused on detecting this information on a binary.

4.7 Replication

The dataset used in our study can be found on Zenodo at the following URL [74], while source code can be found publicly on GitHub [4]. On GitHub, the branch `experiments` contains all the experimental data and results used in this paper.

[4] `http://github.com/davidepi/bincc`

Chapter 5

Compiler Detection

5.1 Introduction

During the software development life-cycle of a natively compiled application, the process of converting source code to binary code performed by a compiler occurs frequently. During this transformation, the compiler is passed several flags via the "build" commands and settings contained within a Makefile or equivalent. This informs the compiler of the developer's intent to retain or omit some information or to modify the original code in an optimized version.

These flags can be used to optimize faster executions, smaller sizes, and lower energy consumption [40]. However, the flags are not explicitly recorded in the binary itself, as they are completely unnecessary for the machine to execute the binary code.

Moreover, the compiler itself is not easily identifiable. There is no standard way to record this information, and although some compilers write a comment in the binary itself, it is easily skipped or duplicated and not guaranteed to be parsable. For example, if a file compiled with the Clang compiler is linked with a library compiled with GNU compiler collection (GCC), this comment will contain both signatures.

However, this information is extremely valuable for various applications, such as categorizing an older build, finding vulnerabilities [18], finding similarities in binaries [16], rewriting a binary [103], or providing accurate bug reports in case the compilation environment cannot be controlled [79]. A simple example of the latter case could be a library that has incompatibilities only with a specific compiler, in a product published by a different vendor than the library developer. Pallister et al. have shown that different optimization options can have a significant impact on final energy consumption [69]. The use of precompiled libraries could be problematic in embedded applications where energy consumption is a concern.

Knowing the compilation flags could even be helpful when performing binary analysis. In their work, Pewny et al. reported that applying their analysis to different compilation flags significantly affects accuracy [71]. In this case, our work will help to provide confidence in the analysis results, because the differences between the optimization levels of O2 and O3 are much less pronounced than those of O0 and O3.

Although there are several papers on detecting the compiler [79] and toolchain [78] used, these methods do not rely on automatic learning approaches. Therefore, a significant effort is required to detect the above information in different architectures. The new architecture must be studied and understood to check if and how the information can be retrieved. In contrast, with a machine learning-based approach, it is sufficient to provide new data and re-run the training to detect a new compiler or flag. With the automated dataset generation provided in our work, the time required to generate this data is a couple of hours for each optimization level that we want to classify.

In this study, we present our approach for recognizing both compiler and optimization levels using a long-short term memory network (LSTM) [39] and a convolutional neural network (CNN) [58] within different architectures. We analyzed common optimization levels on Linux binaries, compiled with either GCC or Clang in two different architectures, and with GCC only in seven different architectures. We want to identify optimization levels ranging from non-optimized code (O0), code that is optimized for speed with different levels of aggressiveness (O1, O2, O3) or code optimized for small binary size (Os). Although we are not the first to tackle this problem [13], the novelty of our research can be summarized as follows:

- The creation of a huge dataset with automated replication scripts consisting of 76630 compiled files. These files come from seven different CPU architectures and two different compilers, with a combined total of 123 GB of stripped binary data.
- The implementation and tuning of a neural network structure that outperforms existing work in flag detection.
- An analysis that examines the minimum possible number of raw bytes that are required to obtain accurate predictions.

This study is an extended version of our previous work [73]. The main differences between our previous study and the current study are as follows.

- The number of optimization levels test was increased from {O0, O2} to {O0, O1, O2, O3, Os}.
- The number of architectures tested have increased from {`x86_64`} to {`x86_64`, `AArch64`, `RISC-V`, `SPARC`, `PowerPC`, `MIPS`, `ARM32`}. These additional architectures require a completely different cross-compilation approach for generating the dataset, as described in Section IV-A. We also made considerable efforts to automate the generation of the dataset. In our previous work, compilation was a manual task, while now we provide several scripts to automate the process. The main advantage of this automated approach is the ability to generate a different dataset with different compilation flags without the user having to do anything. Although this does not work for additional architectures, it still provides an effortless way to generate the dataset with additional flags.
- The evaluation has been completely reworked, considering the expanded category set. Moreover, in this extended version, we tested the feasibility of our study in a scenario with function-level granularity. We also

examined the distribution of flags in the Ubuntu and MacOS operating systems.

The rest of this chapter proceeds as follows. Section 5.2 presents the motivation for our work. Section 5.3 discusses related work in the field of binary file analysis with machine learning. Section 5.4 presents the problem and our approach and Section 5.5 presents the empirical evaluation. Section 5.6 compares our choices with another similar work in the field and discusses them in terms of the results obtained. Section 5.7 describes the limitations of our study and Section 5.8 concludes the chapter. In addition, Section 5.9 provides instructions to download our dataset and source code.

5.2 Motivation

After briefly introducing the motivation behind our work in Section 3.1, we provide an example of the reasons behind our work in this section. Although there are several reverse engineering tools such as IDA[1] or Ghidra[2], these usually do not detect the flags used during compilation or at link time. This is because a user is interested in extracting and collecting information from single binary files as part of reverse engineering. Despite this, in some applications knowing the original binary flags is almost mandatory. For example, Wang et al. showed how in binary rewriting most of the failures are due to compiler optimizations [103].

The decompilation scenario is also equally interesting. According to Katz et al. [52], the presence of optimizations reduces the success rate in correctly decompiling an executable. Moreover, their results are based on recurrent neural networks (RNNs), so we expect traditional approaches with hand-crafted rules to be more susceptible to underlying optimizations. With a higher level of optimization, we expect the decompiler to output a higher amount of `goto` statements. If this is the case, knowing the optimization flags may give an indication of the expected accuracy of a decompiler, knowing that a heavily optimized code may decompile into source code that is more difficult to analyze.

Nevertheless, the focus of our study was on code analysis. When comparing multiple binaries, knowledge of the compilation flags becomes much more useful, almost mandatory. As reported in Section 3.1, Pewny et al. in their analysis of cross-architectural binary code devoted an entire research question investigating the presence of false/true positives at different optimization levels [71]. Ultimately, they concluded that comparing non-optimized to optimized code results in significantly lower accuracy. In our internal study, we came to a similar conclusion. Comparing control flow graphs (CFGs) of functions, even within the same architecture but with different compilers or optimization flags, is dominated by false positives. The impact is so great that the comparison is usually impractical when dealing with different compilers or O0 versus O2, where it is perfectly fine when comparing O1 with O2 or O3. Thus, performing analyses or comparisons between binaries may lead to results that are either good or completely unusable, depending only on the compilation flags used. These two contrasting results have led us to look for a way to detect in advance

[1] `https://hex-rays.com/`
[2] `https://ghidra-sre.org/`

the presence of optimizations. This is a way to determine whether a comparison or analysis between binaries is feasible, and if the results will be reliable.

In our previous study, we focused only on the presence/absence of optimizations, the most influential factor in determining a poor comparison between two binaries. In this extended version, we also tried to detect the degree of optimization. In fact, we can expect a lesser loss of accuracy when comparing, for example, an O1 optimized code to an O3 optimized code.

5.3 Previous Works

Binary file analysis is widely used in the field of security. Recently, machine learning techniques have been used to aid malware detection. Pascanu et al. used recurrent neural networks (RNNs) [70] to extract malicious features from a binary file in an unsupervised manner, which was extended with convolutional neural networks by Athiwaratkun et al. [6].

Related works dealing with compiler flags, instead, focuses on the effects of these flags rather than their detection. Work performed by Triantafyllis et al. focused on exploring optimal compiler flags [101] as did the work of Hoste et al. [40]. In recent years, several machine learning-based techniques have been developed [104] [5]. Older techniques focus on the use of machine learning to reduce the number of iterative compilations required to obtain a good set of flags [1] and to help approximate NP-hard problem efficiently, like phase ordering [99]. Instead, more recent techniques use deep learning to detect function boundaries, a work by Bao et al. [7], which was then extended by Shin et al. [97]. Chua et al. instead detected function types using RNNs [14] whereas He et al. attempted to recover the debug symbols from a stripped binary [36].

To the best of our knowledge, the only work attempting to detect flags in an existing binary, rather than optimizing them, is that of Chen et al. [13]. The main differences between their study and our work are as follows:

- We investigate the detection of not only flags but also compilers.
- We investigate the detection in seven different architectures instead of only one.
- Our analysis aims not only to maximize the accuracy but also to minimize the required input.
- Our dataset is more than 100 times larger [74], disproving some claims of previous research.

5.4 Approach

The problem we are trying to solve is to identify the optimization level and possibly the original compiler used to compile from source code to binary code, when only a portion of the binary code is available. Specifically, given a sequence of bytes v of arbitrary length coming from a binary, we want to train a classifying function $\mathcal{M}_{flags}$ capable of predicting the compilation flags and a classifying function $\mathcal{M}_{compiler}$ predicting the compiler used.

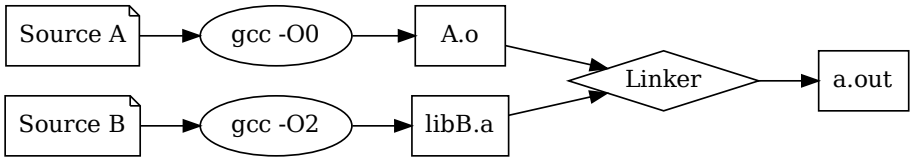


Figure 5.1: Linking an executable may include binary data compiled with different flags.

With $\mathcal{M}_{flags}$, we try to classify the optimization level used in the input binary. In our study, we target the commonly used optimization levels $\{O0, O1, O2, O3, Os\}$ [3]. We trained different $\mathcal{M}_{flags}$ for each of the seven architectures we studied, namely `x86_64`, `AArch64`, `ARM32`, `MIPS`, `PowerPC`, `RISC-V`, and `SPARC`, and expect a user to select the prediction model according to the input architecture. The architecture of a binary is easily recognizable by tools such as `file`, so this fact is not a limitation and simplifies training.

Similarly, with $\mathcal{M}_{compiler}$ we classify the compiler between `gcc` and `clang`, which means that the compiler analysis is a binary classification. This is done only for the natively generated architectures, such as `x86_64` and `AArch64`, for reasons explained in Section 5.4.1. Thus, we have trained two different $\mathcal{M}_{compiler}$.

Our goal is not only to maximize accuracy, but also to keep the sequence of bytes v as small as possible; as such, we dedicate Section Section 5.4.2 to explain how the binary code is transformed into v (or several vs), the input expected by our learning network. To compare the performance of different models, we trained all the aforementioned configurations using a feed-forward Convolutional Neural Network $\mathcal{M}^{CNN}$ and a Long-Short Term Memory Network $\mathcal{M}^{LSTM}$, producing in total $7\mathcal{M}^{CNN}_{flags}$, 7 $\mathcal{M}^{LSTM}_{flags}$, 2 $\mathcal{M}^{CNN}_{compiler}$ and 2 $\mathcal{M}^{LSTM}_{compiler}$. These networks are trained in several different datasets, explained in detail in Section 5.4.1, and their prediction results are compared.

More details about the network models can be found in Section 5.4.4.

5.4.1 Dataset

To train our networks, we must first collect the data. Our networks perform supervised learning, so it is necessary to partition the binary code by optimization level and compiler used.

Although this task may seem trivial, as there are plenty of open-source software programs that can be compiled with the desired flags, several precautions are required during the linking phase. This is because although we can choose both the optimization level and compilers, we have no guarantees about the environment that performs the compilation. There are several libraries available to be statically linked, and we know nothing about the compilation settings used for these libraries. This problem is highlighted in Figure 5.1.

When a library is statically linked, its binary code is copied inside the final executable during the linking phase. As such, in most build systems, preexisting libraries could be linked while generating the dataset. In Figure 5.1,

[3] Note that some applications might use additional "hand-picked" flags. This limitation is discussed in Section 5.7

this is shown in the *Source B* path, where, a source originally compiled with an optimization level of O2 is linked within the same binary with an optimization level of O0. This implies that these libraries could irreversibly contaminate our build, because we lack information about their creation.

Possible options for solving this problem would be:

1. Use a dataset composed of object files prior to linking.

2. Edit the build script for the generated binary to exclude static linking.

3. Create a system with only shared libraries. Then use this system to build the dataset.

In our study, we chose options 3 for the following reasons: Option number 1 would not provide a realistic case study, as several optimizations can also be performed at link time [2]. Option 2, could be "too risky" as it involves manually checking various compilation settings written in different languages and styles in order to ensure dynamic linking and has an high risk of error. For example, in our experiments, we found that some build scripts use hard-coded parameters, others use environment variables, and others recursively integrate files. Checking, understanding, and modifying all these build scripts correctly is entirely possible but carries a high risk of error.

Option 3 can guarantee the absence of contamination, but requires some care, as one needs to isolate from the existing environment. While generating our dataset, in order to ensure this constraint was respected, we used two different approaches depending on whether the build architecture was matching the target one or not. The final dataset is publicly available on Zenodo at the following link [74].

Dataset: Native compilation

The dataset we use for the native compilation contains all software listed in the *Linux from Scratch* book[4], version 9.1-systemd, published on March 1, 2020. In addition, we added to every dataset the LLVM suite version 10.0.0, which is required for building Clang[5]. To solve the static linking problem within a native build system we performed the following steps:

1. We built a toolchain with no particular compiler and optimization level from the host machine. We ensured that only shared libraries were built in this toolchain.

2. We created a chrooted environment containing only the toolchain to isolate it from the original build system.

3. We built the actual dataset, with the desired compiler and optimization level.

After building all software binaries, we then strip and use each ELF file.

[4] `http://www.linuxfromscratch.org/lfs/index.html`
[5] `https://releases.llvm.org/10.0.0/`

Dataset: Cross compilation

The cross-compilation approach we use is based on the presence of readily available toolchains in the Ubuntu Package repository[6]. For the following study, we need the *gcc*, *g++* and *binutils* packages for each architecture we want to target. For cross-compiled architectures, we did not build a Clang dataset and limit our analysis to GCC only.

Unlike in the native compilation presented in Subsection 5.4.1, the linker is not capable of linking binaries from the build system because they are compiled for a different architecture. However, existing toolchains may still ship static libraries, requiring some care to ensure either these libraries are built with the correct flags or not present at all. Aside from that, chrooting is not neeeded, meaning that we can fully automate the entire building process, which normally ends with the execution of the chroot command. Pointers to the script used to perform this automated building can be found in Section 5.9.

However, not all software used in the native building supports cross-compiling. Despite heavily editing most scripts, some software, such as Perl, fails to cross-compile. For this reason, when cross-compiling, we use a slightly different dataset and limit our study to the GCC compiler. This new dataset comprises all the software from LFS that supports cross-compilation as well as some software coming from *Beyond Linux from Scratch*[7]. Due to space limitations, we cannot list every piece of software used in this building process. For a comprehensive list see the `resources/scripts` folder in the repository listed in Section 5.9.

As with native compilation, each ELF file generated is stripped and used in the dataset.

5.4.2 Preprocessing

In this section, we explain the additional preprocessing before the input vector is fed into the networks. In the evaluation, we prove that advanced encoding is unnecessary, but we report it here anyway because previous research has used it.

The goal of this preprocessing is to transform a binary file, usually composed of a sequence of data and instructions, into one or more input vectors for the automated learning step. We are naturally interested in correctly classifying the smallest possible input, with the finest grain being the function grain. We compare two approaches: one using a stream of bytes without prior knowledge of the underlying data and one that requires disassembly and precise function boundaries. The effectiveness of these two approaches is analyzed in Section 5.5.3.

The first approach, referred to as "raw bytes" throughout the chapter, has been shown to be effective for learning functions boundaries in previous research [7].

To generate this representation, we use `readelf` to dump the `.text` section of the executable and divide it into fixed-size chunks. We chose a-priori 2048 bytes as the maximum size of this chunk of bytes v. However, we also evaluated the precision of the networks in detecting the compilation settings for these chunks as their size varied, in order to simulate a case where we want to detect the optimization of a single function.

[6] `https://packages.ubuntu.com`

[7] `http://www.linuxfromscratch.org/blfs/index.html`

```
4889442418  mov qword [rsp+0x18], rax
31C0        xor eax, eax
4885FF      test rdi, rdi
7423        je 0x25
488B4208    mov rax, qword [rdx+0x8]
48893424    mov qword [rsp], rsi
4889E6      mov rsi, rsp
4889442408  mov qword [rsp+0x8], rax
488B02      mov rax, qword [rdx]
4889442410  mov qword [rsp+0x10], rax
E85A0E0000  call 0xE5F
4885C0      test rax, rax
0F95C0      setne al
```

Figure 5.2: Portion of a disassembled function.

One drawback of this representation is that we do not know whether the raw data represents instructions or stack data. In contrast, if we want to classify an entire executable, disassembly is not required, a step that usually requires several minutes. Even when we want to classify at function granularity, based on the research of Bao et al. [7], we can extract function boundaries and perform classification without disassembling the executable, which is not only slow, but is also inherently difficult and prone to error in some architectures [4].

In the second representation, the one that requires disassembly, *radare2*[8], is used to extract each function from the executable. The results are shown in Figure 5.2.

In the figure, the left column represents the raw bytes written in the binary and the right column their translations in Intel Assembly syntax. The example is taken from `x86_64` disassembled code. We can see many bytes specifying that the registers to be used should have a length of 64 bits, represented by the red bytes in figure, preceding every instruction involving `rax`, `rsi`, `rsp` registers. This is a problem, because real functions can be of arbitrary length, but our networks support fixed-length vectors as input. We expect these extra bytes to contribute almost nothing to the final result and thus decide to remove them and keep only the byte(s) representing the operations to be performed, without parameters. Unlike the previous research, we also remove the operands in our representation, in order to save more space and accomodate even more "valuable" instructions inside the limited length vector [13]. Finally, only the blue bytes in Figure 5.2 were encoded in our representation. Extra data are pre-truncated, because we expect the most useful operations are at the end of a function and not at the beginning, which contains the initialization. Insufficiently long functions are pre-padded with zeroes, as pre-padding has been proved to be better for LSTMs [22]. From now on, we will refer to this representation as "encoded".

In both representations, we feed our data to the networks as a time series, where each point in time is actually a byte of data from the binary file. For example, the first two instructions of Figure 5.2 would have this vector in the raw byte approach: `[0x48,0x89,0x44,0x24,0x18,0x31,0xC0]`. In the encoded

[8] `https://rada.re/`

```
f0 25 14 de af 8c 85 c3    00 f0 25 14 de af 8c 85
85 bf 5b cf e0 f2 63 0b    00 00 00 00 85 bf 5b cf
92 af 97 0b 06 84 1d 5d    00 00 00 92 af 97 0b 06
e3 14 bc ac a8 de 21 e7    00 00 00 00 00 00 e3 14
73 11 27 9a ff 4f d9 73    00 00 00 00 00 73 11 27
03 d6 ce de 8b 0d af 46    00 00 03 d6 ce de 8b 0d
74 37 35 f2 49 c3 e5 69    00 00 00 74 37 35 f2 49
8c 47 4a 57 d2 cf 7e 46    00 8c 47 4a 57 d2 cf 7e
```

Figure 5.3: Truncation of input sequences on the left and subsequent padding on the right. The amount of truncated bytes in this Figure is symbolic.

representation instead they would be `[0x89,0x31]`.

5.4.3 Padding

For the encoded representation presented in Section 5.4.2, the input vector length is equal to the function length. This makes it necessary to pad the data, as the function length is always different. However, the raw data approach, requires a fixed amount of sequential data from the binary. As any data in any part of the `.text` binary section can be used as an input, in principle, no padding is strictly required.

However, we determined that by always providing unpadded vectors, both CNN and LSTM are unable to deal with padded data during evaluation. The experimental data in Section 5.5.4 shows that when training with unpadded inputs, the inference of a vector padded with zeroes by more than 60% of its length results in a 10% accuracy drop. This can be detrimental in a real case, as it would be necessary to train different models for different input sizes if we want to infer a smaller amount of data or just a portion of the executable.

To solve this problem, we truncate a random number of bytes in the interval $[0, \alpha]$ where

$$\alpha = len(v) - 32$$

The value 32 has been chosen so that the input chunk v to be still classifiable: if we pad too much, we might get chunks where the classification is impossible due to lack of enough information. This number is also extremely conservative: consider that in x86_64, for example, just calling an imported function requires at least 11 bytes of data, and translates to a single opcode.

The random amount is defined by an exponential distribution. Our intention is to use a distribution where 99% of the values fall within the above interval, while clamping the outliers to 32. In this case, the network would predominately receive low-padded vectors, while occasionally encountering a mostly-padded vector. With the exponential cumulative distribution function as $y = 1 - e^{-\lambda x}$ we fix y to 0.99 and x to α to obtain the λ shown in Equation 5.1.

$$\lambda = \frac{2 \ln 10}{\alpha} \tag{5.1}$$

We then use this λ in the exponential distribution generating the random number of bytes that should be truncated for each input. After truncating the input, we prepad it by adding zeroes.

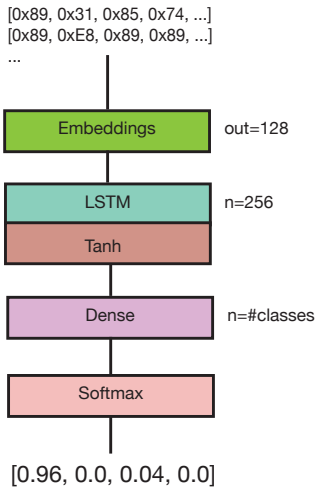


Figure 5.4: LSTM model structure.

An example of this can be seen in Figure 5.3, where each line represents an input sequence before padding on the left block and after padding on the right block. The red part represents the amount of input data that will be truncated. The length of this part is decided randomly within the interval bounds previously mentioned. On the right block we can see that the same amount is replaced by prepending zeroes.

Evaluation of this padding is provided in Section 5.5.4.

5.4.4 Networks

For our analysis, we used two different networks: a LSTM [39] and a feed-forward CNN [58]. These networks have been chosen due to their successful applications in natural language processing or image recognition. In fact, we model our optimization recognition problem as a pattern recognition problem: a particular optimization can be recognized by a network as a pattern of opcodes in the input sequence of bytes.

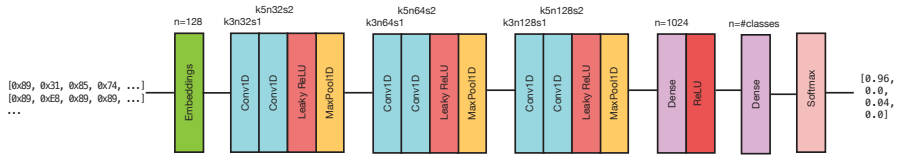


Figure 5.5: CNN model structure.

The first model is shown in Figure 5.4. This model depicts a simple LSTM, given that we encoded our sequence of bytes as a time series and the ability of LSTMs to perform well on this type of problem. LSTMs, in fact, have special "memory" cells, that allows them to memorize a particular input or pattern

even in long sequences [39]. Our core idea is to train this kind of model into memorizing a particular pattern, representing the compiler or the optimization level, over a long sequence of bytes belonging to the binary.

As we can see from the figure the model is pretty straightforward. It is composed of an embedding layer with 256 as vocabulary size (because we use bytes in range 0x0 to 0xFF) and 128 as dimension for the dense embedding. This layer encodes positive integers into a dense vector of fixed size, understandable by the LSTM. Then, the LSTM layer with 256 units is used for the actual learning. This layer uses a hyperbolic tangent (tanh) as the activation function. The kernel is initialized by drawing samples from a uniform distribution in $[-64^{-\frac{1}{2}}, 64^{-\frac{1}{2}}]$. The last part of the network is a dense layer with 1 output and Sigmoid activation for the binary case, dense layer with 5 outputs and Softmax activation for the multiclass case. The optimizer is Adam [55] with learning rate of 10^{-3}.

The second model, is based on the trend in image recognition and categorization [98] and is based on a convolutional neural network. The idea is that a series of convolutions is used to extract highly dimensional information from the sequence of raw bytes passed as input. The Structure is shown in Figure 5.5.

The first layer is identical to that of the LSTM version, because its utility is the same. Then, three blocks of convolution, convolution, and pooling, with increasing number of filters, were used. In the Figure, the label k3n32s1 for a convolutional layer indicates a kernel size of 3, a number of filters of 32, and a stride of 1. In these blocks, the convolutions are used to extract features from the sequence of bytes, and the pooling is used to make these features independent of their position in the sequence.

The leaky ReLU [63] is used in place of the ReLU [67], because the latter suffers from the vanishing gradient problem. Although the ReLU function returns 0 for values less than 0, the leaky variant returns an ϵ, in our case 0.01, to keep the neuron alive. Before output, the final fully connected layer composed of 1024 neurons is used, followed by a ReLU activation and the canonical dense and sigmoid for binary classification or dense and softmax for multiclass classification. Also in this case the optimizer is Adam with learning rate of 10^{-3}.

All the models use binary cross-entropy as loss function for the binary classification and categorical cross-entropy for the multiclass classification [34].

The presented hyperparameters of both the LSTM and the CNN were estimated using the Hyperband algorithm [60]. We used powers of two as space search in the interval $[32, 1024]$ for most features, except kernel size and strides. For the kernel size, the space search was the set $\{3, 5, 7\}$. Instead, for the stride, the space search was $\{1, 2\}$.

5.5 Evaluation

We performed experiments using an Nvidia Quadro RTX5000 on the data presented in Section 5.4.1 after the preprocessing explained in Section 5.4.2. However, given the high amount of binary data, we obtained a number of input vectors in the order of millions. Thus, we could safely split the data into disjointed sets, while maintaining a high number of samples for each set. Thus, we set training, validation, and testing with split ratios of 50%, 25% and 25%

respectively.

No augmentation was performed and no overlapping sequences were collected, with each sample being absolutely unique between training validation and testing. Duplicate samples were removed from each set. These are common especially in the opcode-encoded datasets. Training was performed for 40 epochs using a batch size of 512 samples. An early stopper was employed, which stopped the learning after three epochs if the loss function in the validation dataset did not improve by at least a factor 10^{-3}. In total, we trained 36 models in approximately 550h.

In this section we want to answer the following research questions:

- **RQ**$_{accuracy}$: Is it better to use a CNN or a LSTM? What results can be expected from each network?
- **RQ**$_{min}$: What is the minimum number of bytes required to have accurate predictions?
- **RQ**$_{encoding}$: Does extracting data with a disassembler increase the accuracy of the predictions?
- **RQ**$_{pad}$: Does padding during training improve the performance of the networks?
- **RQ**$_{occurrence}$: What are the most common optimization levels in real-case distributions?

RQ$_{accuracy}$ aims to investigate the advantages and disadvantages of using one network over the other for both compiler detection and optimization level detection. This question is then expanded in RQ$_{min}$ to investigate how much the input size can be reduced while still maintaining a sufficiently high accuracy. RQ$_{encoding}$ was investigated to explain our choice of training with raw data. In particular, this contradicts the claim of previous work conducted by us and Chen et al. [13]. RQ$_{pad}$, instead, serves the purpose of justifying why in Section 5.4.3, we claim that padding is necessary while training with raw data. Finally, RQ$_{occurrence}$ concludes our study by running our models in real-case code to obtain statistics about the most used flags, proving the assumption of O2 being the most popular flag as not always correct.

5.5.1 Accuracy

To evaluate the accuracy of both the CNN and LSTM we divided our dataset by architecture. The number of samples we used for each architecture is listed in Table 5.1. It should be noted that in the worst case we trained with at least 10^6 samples and tested on at least $6 \cdot 10^5$ samples.

The number of features for each sample are 2048 sequential bytes from the specified architecture, categorized by optimization level. In addition, `x86_64` and `AArch64` contains samples compiled with both GCC and Clang. We trained a CNN and a LSTM model for each dataset and obtained the results shown in Table 5.2 regarding the optimization level detection. Note that all results were obtained with raw encoding and padded data unless otherwise stated.

This table represents the categorical accuracy achieved while performing supervised evaluation of our trained models in the testing data. Being the

Table 5.1: Number of training and testing samples for each architecture. Each sample is composed of 2048 bytes.

Dataset	Training	Testing
$\mathcal{D}_{x86\_64}$	$2.4 \cdot 10^6$	$1.2 \cdot 10^6$
$\mathcal{D}_{aarch64}$	$2.3 \cdot 10^6$	$1.1 \cdot 10^6$
$\mathcal{D}_{riscv64}$	$1.3 \cdot 10^6$	$7.0 \cdot 10^5$
$\mathcal{D}_{sparc64}$	$1.9 \cdot 10^6$	$9.8 \cdot 10^5$
$\mathcal{D}_{powerpc}$	$2.0 \cdot 10^6$	$1.1 \cdot 10^6$
$\mathcal{D}_{mips}$	$1.6 \cdot 10^6$	$8.2 \cdot 10^5$
$\mathcal{D}_{arm}$	$1.2 \cdot 10^6$	$6.2 \cdot 10^5$

Table 5.2: Accuracy for each architecture while detecting the optimization level.

Architecture	CNN Accuracy	LSTM Accuracy
`x86_64`	0.8781	0.9291
`AArch64`	0.9181	0.9687
`RISC-V`	0.8427	0.9209
`SPARC`	0.9364	0.9682
`PowerPC`	0.8702	0.9227
`MIPS`	0.9596	0.9837
`ARM`	0.9380	0.9588

accuracy a metric defined for binary classification, every time we use this term in a multiclass context, we refer to the following formula in Equation 5.2, where tp, tn, fp, fn are true positives, true negatives, false positives, false negatives and k is the number of classes.

$$accuracy = \frac{\sum_i^k \frac{tp_i + tn_i}{tp_i + tn_i + fp_i + fn_i}}{k} \tag{5.2}$$

We can note how that the LSTM is always better than the CNN, with the worst accuracy being recorded for `x86_64`, `RISC-V`, and `PowerPC` in both networks. However, the downside of using the LSTM is its extensive training time. We can see this in Figure 5.6.

The figure shows the times obtained from the MIPS dataset, one of the datasets with the fastest training times owing to its size. It takes approximately seven epochs for the LSTM to reach the same accuracy as the CNN at the end of the first epoch. In addition, the CNN can complete 10 epochs in the time the LSTM is able to complete only 3. We also measured similar speeds during inference, with a CNN two to three times faster than the LSTM. This is not limited to the single architecture we presented in Figure 5.6. As we can see in Table 5.3, it applies to any architecture, with the worst case being `x86_64` having an LSTM requiring more than five times the corresponding CNN training time. Note, however, that these times were collected only once. As such, variations, even significant ones, are expected.

To further investigate the accuracy, Figure 5.7 shows all the confusion matrices for each model trained with the CNN.

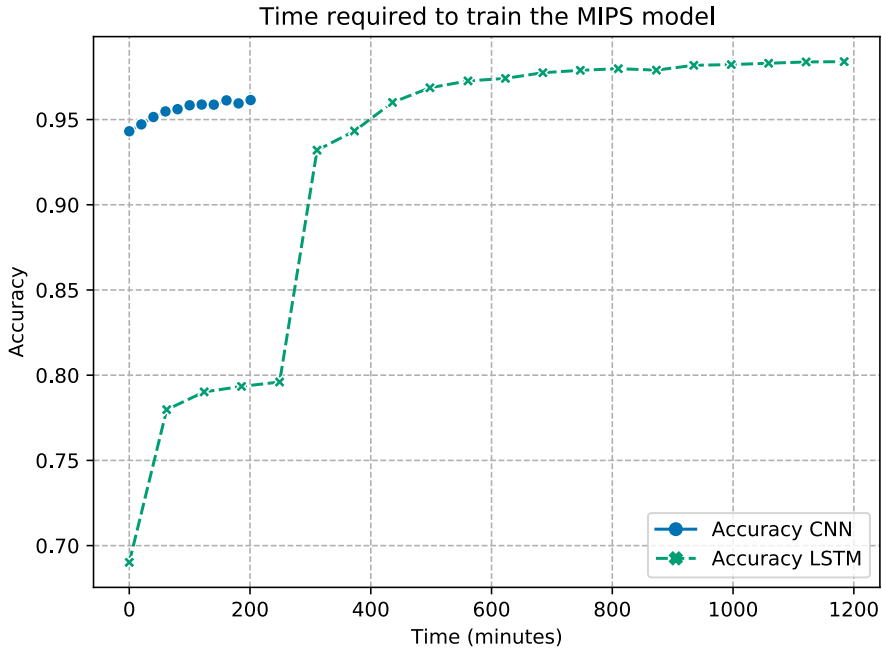


Figure 5.6: Accuracy obtained in the validation dataset at the end of each training epoch.

The problematic part, as seen from the Figure, is the distinction between O2 and O3. In fact, O0 and O1 can be detected with 99% accuracy in any architecture and Os is never below 96%. O3, however, in the worst case has more wrong classifications than correct ones, as we can see in `PowerPC` and `RISC-V`. This situation is slightly better when an LSTM is used. The results are shown in Figure 5.8

In this Figure, we can see how the LSTM achieves high accuracy in some architectures, namely `AArch64`, `SPARC`, `MIPS`, and `ARM`. The architectures problematic for the CNN remain problematic also for the LSTM, but to a much lesser extent. In fact, no optimization level reports more wrong classification than correct ones, and the worst case is a 70% accuracy for `PowerPC` O2.

Table 5.3: Training time, in minutes, required for each network and architecture.

Architecture	CNN Time	LSTM Time
`x86_64`	361 min	1845 min
`AArch64`	298 min	1764 min
`RISC-V`	220 min	1034 min
`SPARC`	346 min	1397 min
`PowerPC`	398 min	1379 min
`MIPS`	257 min	1362 min
`ARM`	407 min	629 min

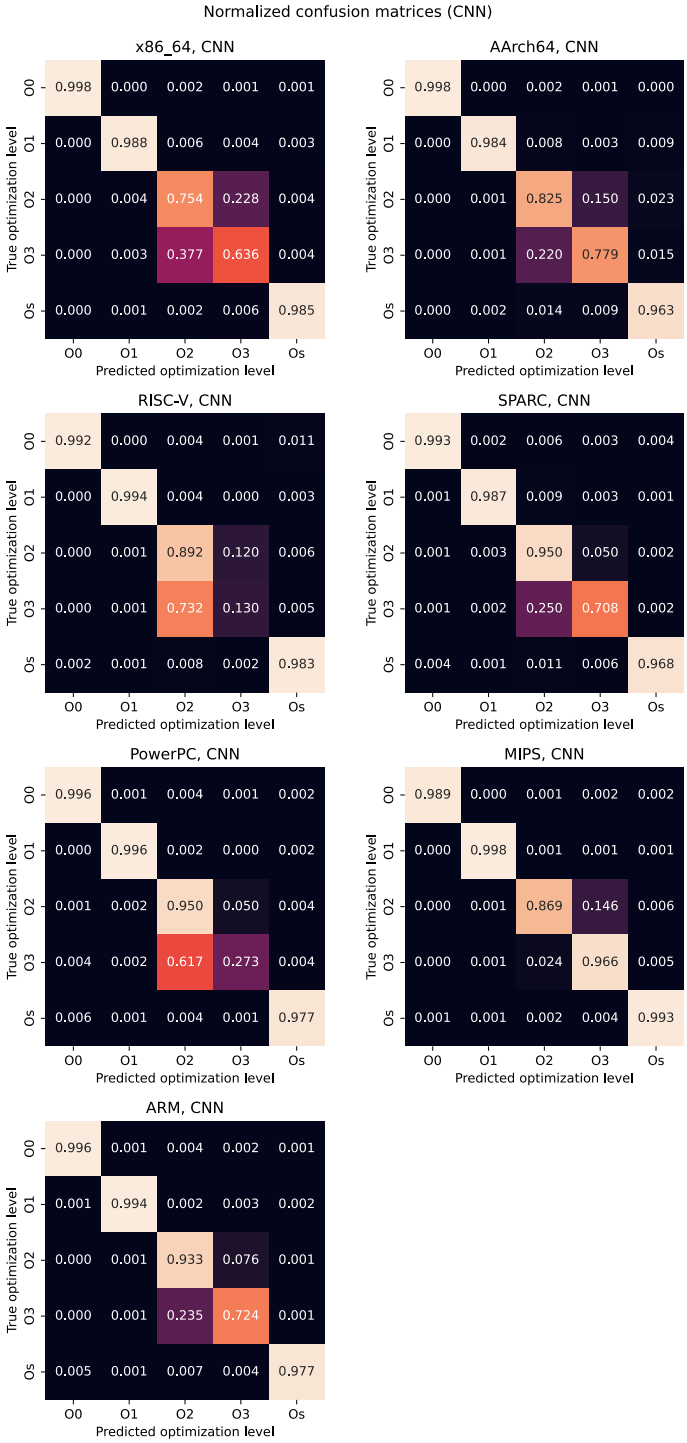


Figure 5.7: Confusion matrices while detecting optimization level for each architecture. Results obtained with the CNN.

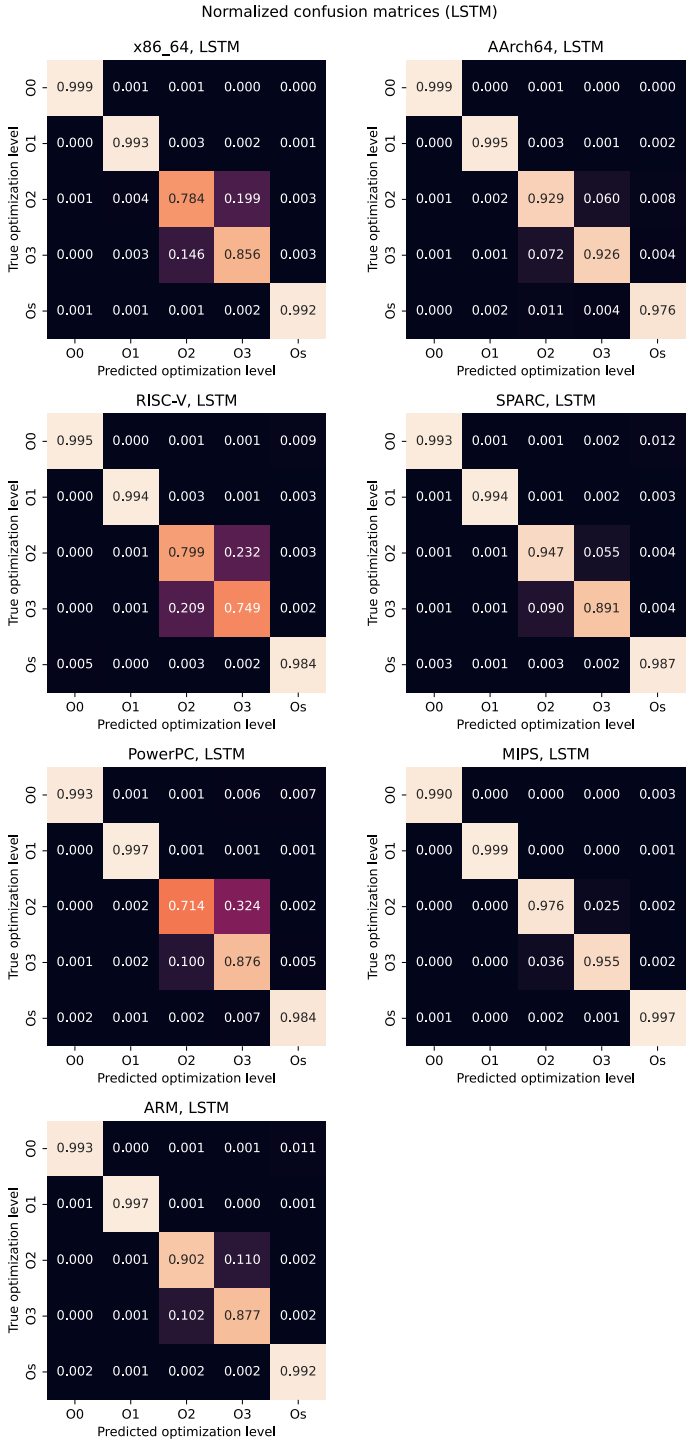


Figure 5.8: Confusion matrices while detecting optimization level for each architecture. Results obtained with the LSTM.

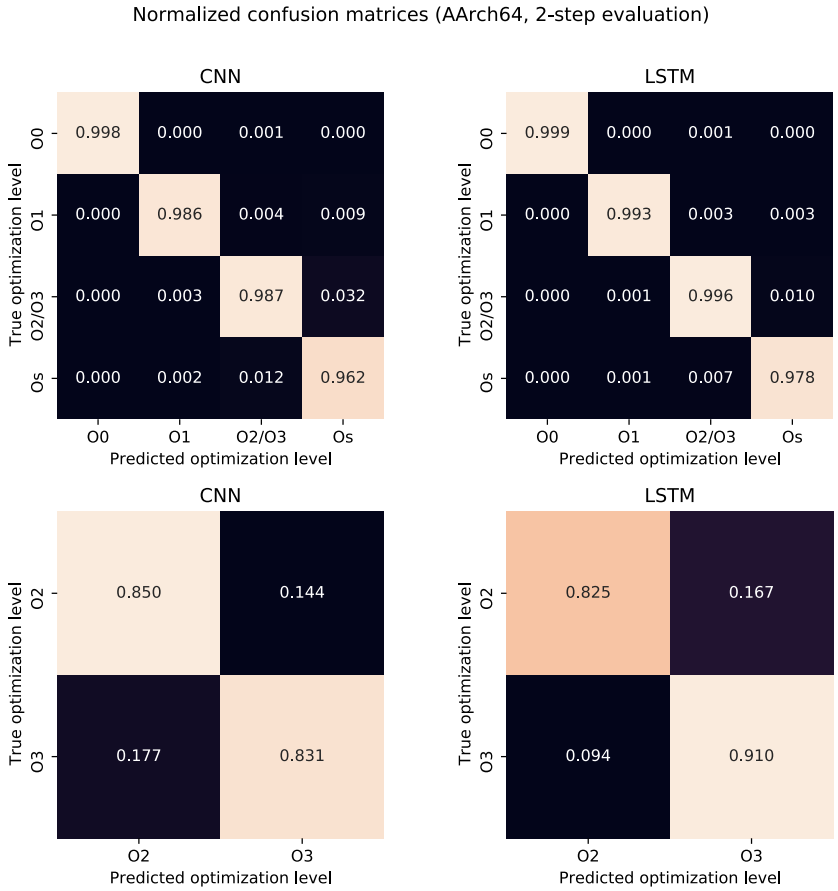


Figure 5.9: Confusion matrices whith split dataset. CNN on left, LSTM on right.

To mitigate this problem, we trained two additional datasets: $\mathcal{D}_{merged}$ and $\mathcal{D}_{splitted}$: the first containing all optimization flags but with O2 and O3 merged together, the second containing only O2 and O3.

Figure 5.9 reports this split dataset situation. We can notice how the CNN network performs slightly better after separating O2 and O3 from the rest of the dataset, whereas the LSTM performs slightly worse compared to the results without separation.

Concerning the compiler detection, results are reported in Table 5.4.

The table shows how both networks in both architectures perform excellently. Even in the `x86_64` with CNN case, that performed quite poorly in the optimization detection, the incorrect classifications were 587 compared to 1225822 correct classifications. Given the high accuracy of the CNN and its faster speed compared to an LSTM, it should be the preferred choice for compiler detection.

We can thus answer $\text{RQ}_{accuracy}$ as follows:

Table 5.4: Accuracy for each architecture while detecting the compiler.

Architecture	CNN Acc.	LSTM Acc.
`x86_64`	0.9995	0.9932
`AArch64`	0.9996	0.9996

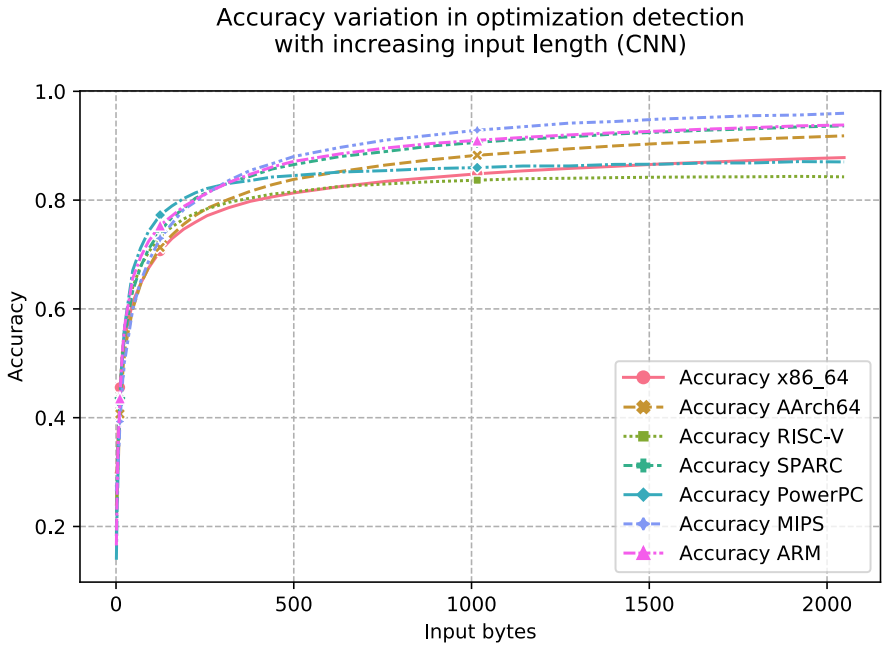


Figure 5.10: Accuracy for the CNN in the optimization detection.

While detecting the optimization level, the LSTM can offer higher accuracy at the price of slower train and inference. The accuracy range from a minimum of 92% to a maximum of 98% depending on the architecture. While detecting the compiler, however, both networks perform well. In this case the CNN is the preferred choice due to its speed advantage and an accuracy of 99.95%.

5.5.2 Minimum bytes

This section investigates the possibility of detecting the optimization level and compiler with function granularity. To this end, we performed our evaluation for each model while feeding a progressively increasing number of bytes. We thus performed the initial evaluation with only 1 byte for each sample; then, we performed a second evaluation with 2 bytes and so on until we used the full vector length of 2048 bytes.

Figure 5.10 shows the results obtained using the CNN network. Figure 5.11, shows the same results but with the LSTM network.

Note that every architecture follows the same detection trend in the LSTM

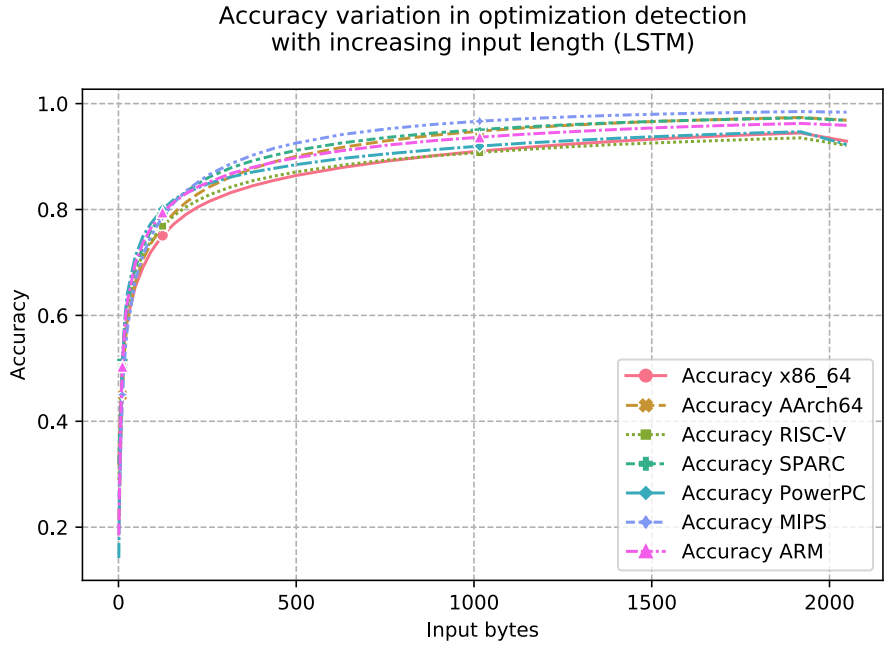


Figure 5.11: Accuracy for the LSTM in the optimization detection.

network. In the CNN one, however, the accuracy for `x86_64`, `PowerPC`, and `RISC-V` stops increasing, unlike in the other architectures. These three architectures are the same architecture we classified as "problematic" in Section 5.5.1. In addition, this does not happen for the LSTM network. We can assume the CNN failed to learn how to properly recognize O2 and O3 in these architectures, given the strong similarity between these two optimization levels. As such, additional bytes do not help the network at all, in contrast to the LSTM case.

Additionally, we can note how, with any number of bytes, the LSTM performs definitely better than the CNN.

To highlight this, Figure 5.12 shows a direct comparison between LSTM and CNN in a single architecture, `x86_64`. This figure, shows how there is always approximately 5% more accuracy in the predictions of an LSTM compared to the predictions of a CNN.

Having analyzed how the overall accuracy varies when the input length changes, we now want to check whether the average function length is sufficient to achieve good accuracy. To do this, however, we need to gather statistical data on binary files. The idea is to calculate the average function length at each optimization level and check the accuracy of the network for that particular level at that particular length.

We disassembled every binary for every optimization level and compiler and counted the number of bytes that compose each function. The results are shown in Figure 5.13.

The reported number is the median, calculated over $47 \cdot 10^6$ functions across

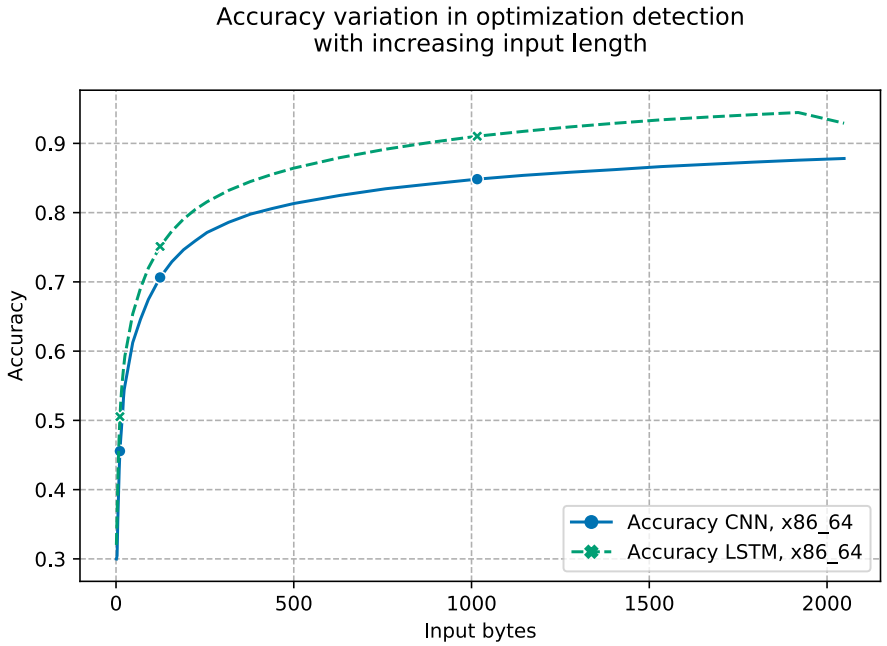


Figure 5.12: Comparison between CNN and LSTM in the x86_64 optimization detection.

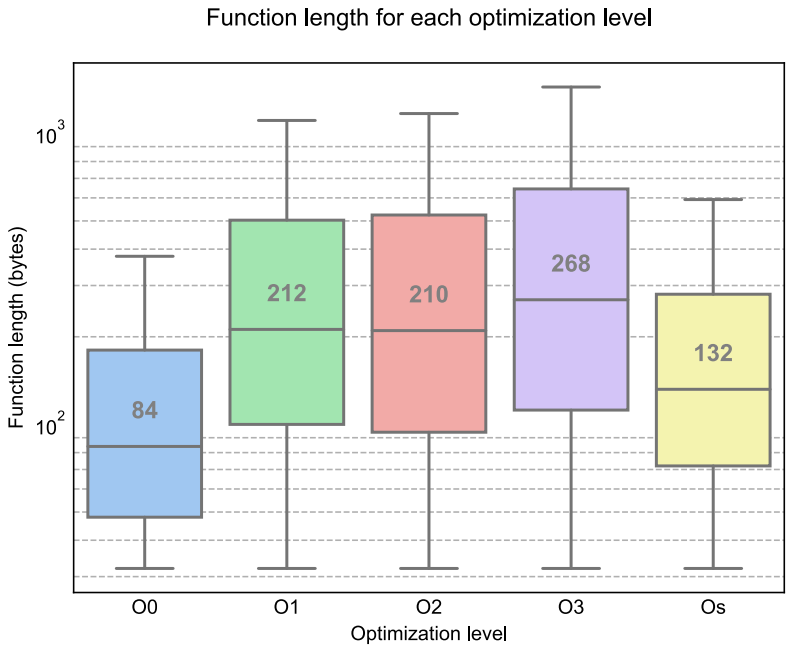


Figure 5.13: Statistics about the length of $47 \cdot 10^6$ functions. The value inside each box refers to the median.

all architectures. We chose to show this average metric, as opposed to the mean, as we want to draw conclusions based on the "typical" function length. This is also a much more conservative approach, given that the mean is influenced by some outliers with a very high number of bytes, on the order of 10^6. Although Figure 5.13 shows the medians for all the architectures together, more precise results for each architecture are reported in Table 5.5.

Table 5.5: Median number of bytes per function for each optimization level in each architecture. Results collected over a total of $47 \cdot 10^6$ functions.

Arch.	O0	O1	O2	O3	Os
x86_64	65	223	220	257	125
AArch64	68	232	220	260	132
RISC-V	84	160	166	218	113
SPARC	104	192	220	268	132
PowerPC	136	224	264	292	148
MIPS	184	276	284	376	188
ARM	72	188	174	232	106

At this point, we calculated the accuracy for each of the listed medians. Figure 5.10 and 5.11 show the overall accuracy of correctly predicting each optimization level. Instead, we want to consider the accuracy of predicting each optimization level at its own statistical median, that represents the typical length of a function at that optimization level. As an example, for architecture `x86_64` we evaluate the accuracy for O0 with an input of 65 bytes, for O1 with an input of 223 bytes. Table 5.6 shows the accuracy at these input lengths.

Table 5.6: Accuracy of each optimization level limiting input to the median number of bytes per function at that optimization level. Results obtained with the LSTM.

Arch.	O0	O1	O2	O3	Os
x86_64	0.992	0.935	0.568	0.579	0.761
AArch64	0.984	0.948	0.719	0.590	0.694
RISC-V	0.987	0.863	0.729	0.344	0.791
SPARC	0.985	0.899	0.829	0.512	0.808
PowerPC	0.988	0.968	0.701	0.527	0.884
MIPS	0.985	0.986	0.786	0.771	0.909
ARM	0.984	0.951	0.658	0.529	0.807

The Table confirms the results we obtained in Section 5.5.1. Optimization levels O0 and O1 are easy to detect even at function granularity. The same goes for Os, although with a lower accuracy. The problem is, again, distinguishing between O2 and O3, as the median length of each function at that optimization level is not enough for an accurate prediction.

Regarding the compiler detection, the accuracy plot for increasing number of bytes can be seen in Figure 5.14. Unlike the CNN and LSTM comparison for optimization level detection, in Figure 5.12, we can see very few differences

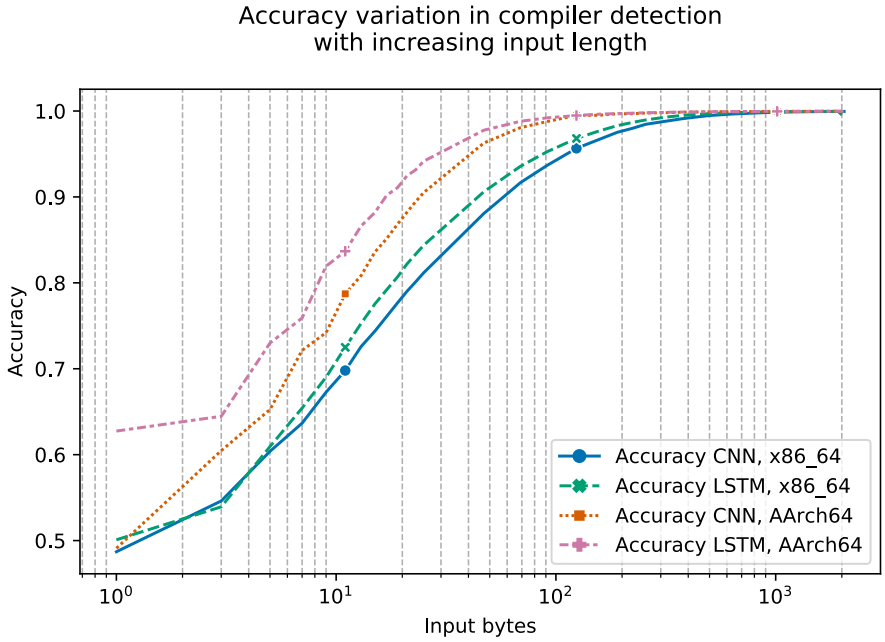


Figure 5.14: Accuracy in the compiler detection.

between the two networks, even with shorter inputs. Moreover, the accuracy is high even when there is not much data available; for example, with only 100 bytes, it is possible to have more than 90% accuracy. This means that we can correctly predict the compiler, even with function granularity.

Given these results, we can answer RQ_{min} as follows:

> *When performing a function grained analysis, with a short input, it is generally possible to detect O0, O1 and Os optimization level. O2 and O3, instead, requires as much bytes as possible, given their subtle differences. In contrast, compiler detection does not suffer this problem, achieving great accuracy even with* 10^2 *input bytes.*

5.5.3 Encoding

After showing how function-grained analysis is possible for some flags in Section 5.5.2, we want to explore a possible improvement by removing redundant information from the input array. This stems from the conclusion of Chen et al. who found removing x86_64 prefixes increases the accuracy [13]. In this section, we compare the raw input with the encoded variant which is explained in Section 5.4.2. The result of this analysis is shown in Figure 5.15, depicting only the x86_64 architecture.

We can note how the encoded variant reflects the same difference between LSTM and CNN previously highlighted in Section 5.5.2, in particular in Figure 5.12. More interestingly, the encoded variant reaches its maximum accuracy with an input length of approximately 250 bytes. The raw input variant, in-

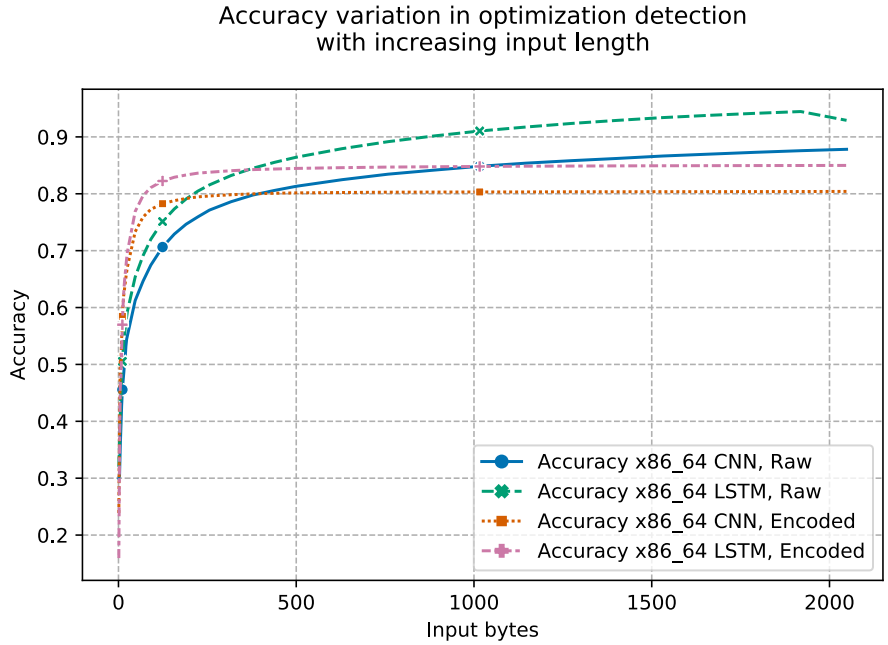


Figure 5.15: Accuracy in the optimization detection with encoded input data.

stead, as more bytes are supplied to it, steadily improves in accuracy beyond this limit.

Moreover, the comparison we used in the Figure is biased towards the encoded variant, especially when we use a small number of bytes. In fact, for any given number of bytes in the encoded variant, we have more bytes available in the corresponding raw one. We calculated this difference to be an average of 186 bytes per function.

A similar situation can be seen when analyzing compiler detection, as depicted in Figure 5.16. In this case, we can see that the encoded variant reaches maximum accuracy at approximately 100 bytes without further improvements. In contrast, the raw data accuracy continues to increase, outperforming the encoded variant at 150 bytes and peaking at 1000 bytes.

We decided, however, against extending this analysis to all seven architectures. In fact, in Section 3.1, one of the motivations of our study was to have an automated way of detecting optimization flags that does not require deep knowledge of the underlying architecture. To generate the encoded variant, however, it is necessary to possess a basic knowledge of the target architecture, which contradicts our original motivation. This fact, in addition to the poor performance and the need for accurate disassembly prompted us to abandon the encoded variant study.

Before concluding this section, it is worth noting that the study of Chen et al. used a dataset 100 times smaller and determined the encoding variant was remarkably better [13]. In our previous study, we used a dataset 10 times smaller than our current dataset and determined the encoding variant to be on

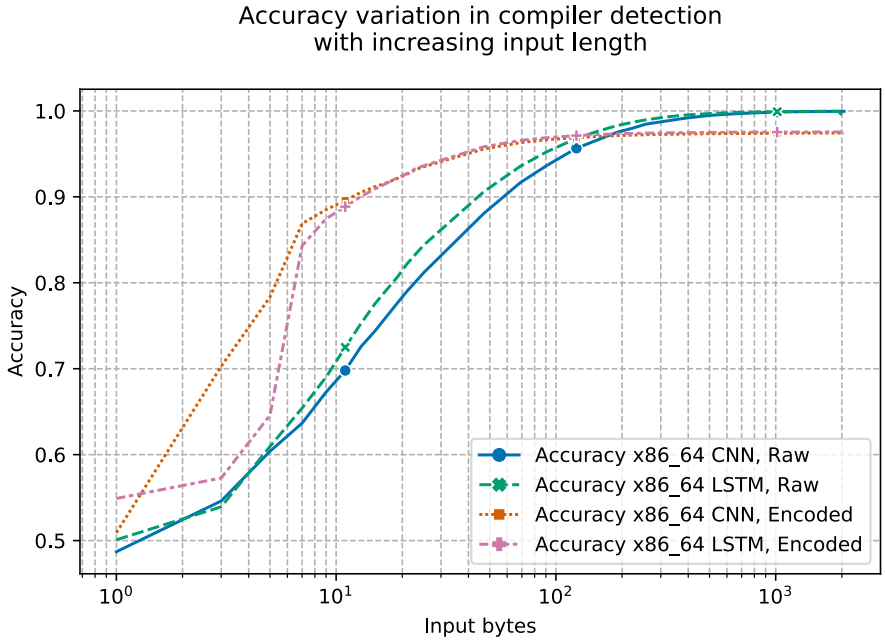


Figure 5.16: Accuracy in the compiler detection with encoded input data.

par with the raw variant [73]. We can easily assume that with a smaller dataset, the network is less capable of learning which information is useful and what is not in the raw data. This would explain why in previous studies the encoded variant, which provides data without useless prefixes, was more competitive. However, with a sufficiently large dataset, the encoded variant does not offer any advantages.

We can thus conclude $\text{RQ}_{encoding}$ as follows:

Disassembling and encoding the data does not provide additional benefits, requiring knowledge of the underlying architecture and function disassembly for an overall lower accuracy.

5.5.4 Padding

In Section 5.4.2 we assert that our networks perform worse if, during training, raw byte sequences are never padded during training, and then padded sequences are predicted. In this section we present RQ_{pad}, and investigate the difference between padding during training and not padding. In this experiment, we trained two networks with the same dataset, seed, and samples ordered in the same way. However, in one case; the training values were always of 2048 bytes, in the other case, they were randomly cut in the interval $[32, 2048]$, following the distribution explained in Section 5.4.3.

We evaluated both CNN and LSTM over the MIPS architecture, which achieved the best results. The differences between the padded and unpadded variants are shown in Figure 5.17

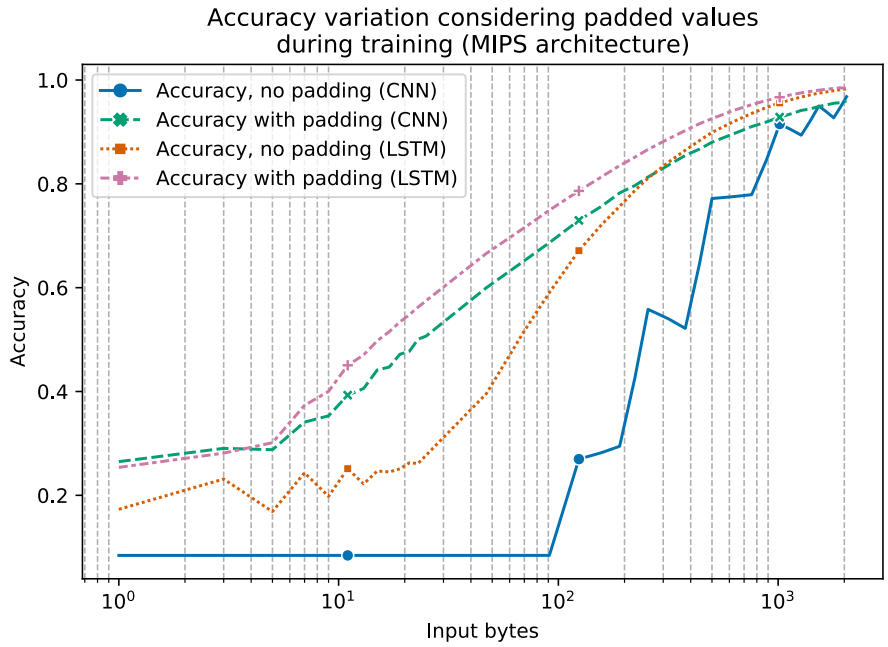


Figure 5.17: Accuracy variation in the optimization detection evaluation when including padding data during training.

From the figure, we can see how the absence of padding during training is a problem when evaluating small input vectors. For example, the LSTM is approximately 10% less accurate and reaches its counterpart trained with padding only when the input vectors are longer than 1000 bytes. The CNN results are even more extreme: with less than 100 bytes the network trained without padding always predicts the same output, and even at 125 bytes, there is a 60% accuracy gap between the two variants.

Although not presented, we performed this analysis also in the x86_64 architecture, and obtained similar results.

We can thus conclude RQ_{pad} as follow:

> *If inputs are never padded during training, networks will have significantly lower accuracy while predicting shorter sequences.*

5.5.5 Occurrence

To conclude the evaluation, we would like to provide some data on the optimization flag distribution in a real scenario. To do so, we took every binary and library inside an *Ubuntu Linux 20.04 server* and *macOS 10.15.7 Catalina*, both unmodified. For each binary, we predicted the optimization level using our LSTM model for `x86_64`, reporting the results.

To analyze each binary, we divided the binary into several chunks of 2048 bytes, which is the same as our max input length. We then performed the inference for each chunk, and calculated an average between all the chunks. For

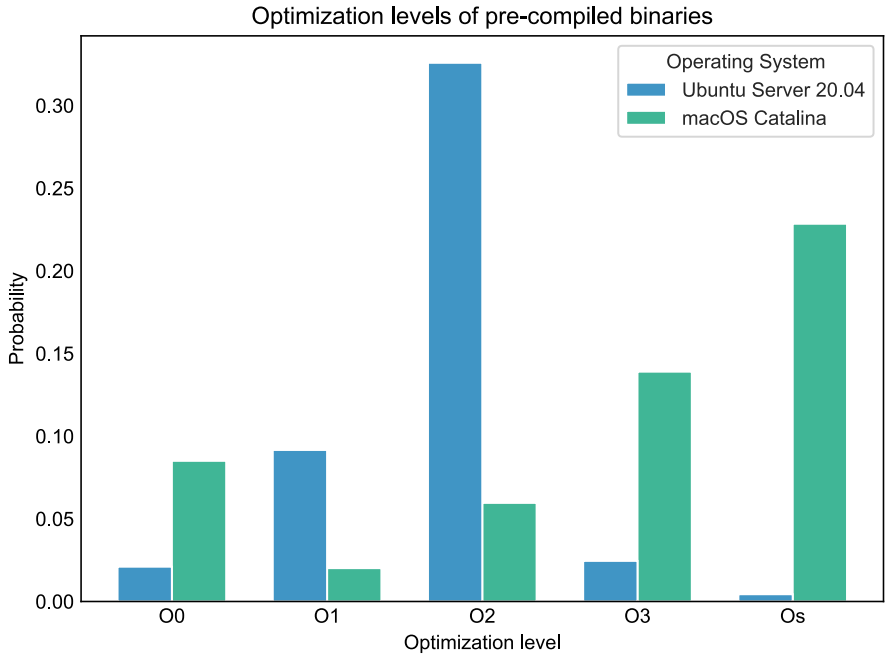


Figure 5.18: Optimization level in pre-compiled binaries shipped with Ubuntu server 20.04 and macOS Catalina.

each chunk, we weighted its contribution to the average by the network accuracy achieved at the chunk's input length.

Figure 5.18 shows the results of this analysis, performed over 10254 and 1216 files, respectively, for Ubuntu and macOS. Given the highly imbalanced number of input files, the histogram was normalized.

We can note how the distribution of files in the Linux system tends towards the O2 optimization level. This is not surprising, as the O2 optimization level provides the highest optimization without increasing the code size to the same extent as O3. The latter, in fact, could generate a larger code that does not fit in the instruction cache, resulting in overall slower execution [49]. Therefore, O2 is the suggested optimization level in some distributions, such as Gentoo Linux[9]. The macOS result, despite being more diverse, shows how most of its core programs are optimized for code size. As this is rather uncommon, we verified this results by manually inspecting the publicly available build scripts for Apple software[10]. As a confirmation, in most Makefiles we can find Os as the default optimization flag, explaining the histogram results. However, no reason for this choice is given in the build scripts.

Nonetheless, this final analysis is useful to prove the point that targeting binary analyses at the O2 optimization level assuming it to be the default one [71] may be a completely wrong assumption in some cases.

With this data, we can answer $\mathrm{RQ}_{occurrence}$ as follows:

[9] `https://wiki.gentoo.org/wiki/GCC_optimization`
[10] `http://opensource.apple.com`

The most common optimization found in Ubuntu Linux is O2. In contrast, on macOs, Os is more common although not as dominant as O2 is for the Linux case. This proves that expecting O2 to always be the most common optimization level may be a false expectation.

5.6 Discussion

The results obtained in Section 5.5 provide a fast way of detecting the optimization level with multiple granularities.

One major change from previous studies is the lack of disassembly in our approach. Using a disassembler in order to retrieve function boundaries can be very time consuming. For large binaries, if the function grain is not required, previous approaches still require functions beginning and function ending, where ours can select a random 2048 bytes from the `.text` section and be dominated by the inference time. Unlike disassembly, dumping raw bytes from an executable or library is very fast, because it involves just reading the file itself.

If a function grain is necessary, the function headers may be retrieved by other means (i.e. Deep Learning). Given that our method does not require to preprocessing of the input data, we can skip disassembly also in this case, avoiding again the slowest part of binary analysis. This allows our tool to be used to check the compilation flags even at runtime, without any noticeable performance impact.

This lack of disassembly is the result of our evaluation in Section 5.5.3, which contradicts the claim of Chen et al. that the encoded variant is better [13]. As mentioned in the section, this could be due to our larger dataset, which allowed the network to automatically learn which data is useful in the input architecture and which is not, rendering manual encoding useless.

In our study, given the larger size of our analysis, we also focused on producing an automated script to generate the dataset. In fact, manually compiling a matrix of five optimization levels using seven architectures would have required an unmanageable amount of time. In addition, with this automated generation, we can extend the study to additional flags with small changes in the scripts parameters. In previous approaches, including ours, the entire dataset had to be manually regenerated to add new flags, the most tedious part of this entire study.

In addition, thanks to the small number of bytes required by our method, we can target very small portions of code, and thus, it can be used to check which portions of the binary match the used compilation flags. This allows for better categorization of the binary content and can help in binary analysis. In fact, if a small portion of the file is found with different flags or compiler than the rest of the file, there is a high probability that this portion belongs to a static library or a different compilation unit.

5.7 Limitations and Future Works

The analysis we performed was limited to a pair of compilers and the most common optimization levels. In this study, we have shown that detection results can vary greatly between different architectures, and we have no guarantee that this analysis can be extended to more architectures without sacrificing accuracy.

This is true even in the case of different compilers. The main difficulty in our study was distinguishing between O2 and O3 given their similarities. However, optimization levels are compiler specific and we cannot assume compilers other than GCC or Clang provide the same set of optimization levels.

Moreover, in this study, we focused entirely on optimization levels instead of specific flags. Although it would be easy to consider optimization flags, given our automated dataset generation, the classification should probably change from multiclass to multilabel. Furthermore, some flags would be challenging if not impossible to detect, the "dead code elimination" flag being one example.

Future work will involve assessing the feasibility of this multilabel classification, especially in compilers other than GCC or Clang.

5.8 Conclusion

In this chapter, we have described two deep learning networks, one based on a long short-term memory model and the other based on a convolutional neural network model. We evaluated them in seven different architectures and showed that they can achieve between 92% and 98% accuracy while detecting between five different optimization levels and over 99.99% accuracy while detecting two different compilers.

We also provided an evaluation of the minimum number of bytes needed for accurate predictions, combined with statistical data about the different architectures and their median function length for each optimization level. Ultimately we proved that function grained optimization level detection is possible unless we are not aiming to distinguish between O2 and O3.

The results obtained are consistent with the initial motivation for our study: when comparing the structure of different binaries we reported the highest accuracy drop emerging in the case of different compilers or O0 compared with any other optimization level. These are also the values with the highest detection accuracy in our study, suggesting that our approach may be useful in detecting accuracy drop when comparing different binaries.

The speed required for this detection is minimal, compared to deep learning approaches presented in Chapter 4, allowing us to use this approach along with techniques presented in the aforementioned chapter, while retaining the same scalability.

5.9 Replication

The dataset used in our study can be found on Zenodo at the following url [74]. This dataset contains each binary, divided by architecture, optimization level and compiler. Source code and pre-trained models can be found publicly on GitHub [11].

[11] `http://github.com/inoueke-n/optimization-detector`

Chapter 6

Conclusion

6.1 Conclusion

This dissertation presents our studies in detecting cloned portions of code in lower-than-source level.

Firstly, we studied if it is possible to increment the accuracy of the reported clones by using compilation transformations and clones detection at IR level. Our first study was focused on combining existing tools, in particular we used the Rust compiler `rustc` to generate the IR and *CCFinderSW* as code clone detector in order to efficiently detect the clones. We detected the Type-2 clones in the original codebases and their respective IR representation. We showed that, despite an increase in the number of clones reported, most of these are due to macro expansion. Moreover, a noticeable effort is required to map the IR representation clone to the original source version in order to be displayed to the user.

For this reason, in the second study, we improved this approach by manually implementing the compiler normalization instead of using IR. This allows the implementor to control the code generation step and ease the mapping of the original source code with the transformed one. This second study proved to be more general and applicable to multiple languages, even non-compiled ones. The reported clones have no drawbacks compared to the original ones, but a non-trivial effort is required to implement this normalization step.

In the third study, we focused entirely on binary clone detection and provided our approach for detecting clones exclusively in binary code. We showed that our approach has the same accuracy of existing ones and is three order of magnitude faster, while being capable of accurately detecting clones even across architectural boundaries. This accuracy ranges between 91% and 99% while detecting clones in the same architecture, and is around 75% when detecting clones in different architectures.

Despite that, we had to dedicate a fourth study at detecting compiler and optimization options. In fact, despite the goodness of our approach in binary code clone detection, if the input files differ in the compiler or optimization flags used, the final result may not be accurate. This was a common problem highlighted even in the earliest work in binary code clone detection. This fourth study showed that it is possible to detect the optimization level with an accuracy

between 92% and 98% and the compiler with 99.95% accuracy. Despite the implementation being a learning approach, its runtime is consistent with the binary code clone detector, and can be used alongside it without hampering its scalability.

6.2 Future Work

In the second part of the dissertation, we presented a novel binary clone detection algorithm that can analyze multiple binaries altogether in a fast and scalable way. However, in the original study, we focused only on the performance of the approach and did not perform a large-scale study using it. Future works will use the presented tool to compile a list of vulnerabilities propagation across different architectures and versions.

Moreover, it would be interesting to apply the scalable approach presented in Chapter 4 to low-level code (e.g. Bytecode and IR) and see if it provides additional benefits over existing source code detectors. While we have determined that existing source code detector do not work particularly well in binary code, we can not exclude that binary-level clone detectors may work better than source-level one in these contexts, given the degree of similarity between binary and low-level code.

Acronyms

AST Abstract Syntax Tree. 10, 11, 14

CFG Control Flow Graph. 4, 5, 12, 34–41, 43, 45, 46, 49, 50, 59, 62, 63

DFS Depth-First Search. 45

HIR High-level Intermediate Representation. 3, 11–14, 16–19

IR Intermediate Representation. 2, 3, 5, 9, 11, 12, 19, 93, 94

ISA Instruction Set Architecture. 37, 38

LSH Locality-Sensitive Hashing. 5, 46

MIR Mid-level Intermediate Representation. 3, 12, 13, 18

SCC Strongly Connected Component. 44, 45

THIR Typed High-level Intermediate Representation. 3, 12

Bibliography

[1] Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F., Thomson, J., Toussaint, M., and Williams, C. K. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization (CGO'06)* (2006), IEEE, pp. 11–pp.

[2] Aho, A. V., Sethi, R., and Ullman, J. D. Compilers, principles, techniques. *Addison wesley 7*, 8 (1986), 9.

[3] Amme, W., Heinze, T. S., and Schäfer, A. You look so different: Finding structural clones and subclones in java source code. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2021), IEEE, pp. 70–80.

[4] Andriesse, D., Chen, X., Van Der Veen, V., Slowinska, A., and Bos, H. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th {USENIX} Security Symposium ({USENIX} Security 16)* (2016), pp. 583–600.

[5] Ashouri, A. H., Killian, W., Cavazos, J., Palermo, G., and Silvano, C. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR) 51*, 5 (2018), 1–42.

[6] Athiwaratkun, B., and Stokes, J. W. Malware classification with lstm and gru language models and a character-level cnn. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2017), IEEE, pp. 2482–2486.

[7] Bao, T., Burket, J., Woo, M., Turner, R., and Brumley, D. {BYTEWEIGHT}: Learning to recognize functions in binary code. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)* (2014), pp. 845–860.

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

[9] Bourquin, M., King, A., and Robbins, E. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop* (2013), pp. 1–10.

[10] Brumley, D., Lee, J., Schwartz, E. J., and Woo, M. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)* (2013), pp. 353–368.

[11] Caldeira, P. M., Sakamoto, K., Washizaki, H., Fukazawa, Y., and Shimada, T. Improving syntactical clone detection methods through the use of an intermediate representation. In *2020 IEEE 14th international workshop on software clones (IWSC)* (2020), IEEE, pp. 8–14.

[12] Chandramohan, M., Xue, Y., Xu, Z., Liu, Y., Cho, C. Y., and Tan, H. B. K. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2016), pp. 678–689.

[13] Chen, Y., Shi, Z., Li, H., Zhao, W., Liu, Y., and Qiao, Y. Himalia: Recovering compiler optimization levels from binaries by deep learning. In *Intelligent Systems and Applications* (Cham, 2019), K. Arai, S. Kapoor, and R. Bhatia, Eds., Springer International Publishing, pp. 35–47.

[14] Chua, Z. L., Shen, S., Saxena, P., and Liang, Z. Neural nets can learn function type signatures from binaries. In *26th {USENIX} Security Symposium ({USENIX} Security 17)* (2017), pp. 99–116.

[15] Cifuentes, C., and Van Emmerik, M. Recovery of jump table case statements from binary code. *Science of Computer Programming 40*, 2-3 (2001), 171–188.

[16] David, Y., Partush, N., and Yahav, E. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2017), pp. 79–94.

[17] Davis, I. J., and Godfrey, M. W. From whence it came: Detecting source code clones by analyzing assembler. In *2010 17th Working Conference on Reverse Engineering* (2010), IEEE, pp. 242–246.

[18] Demertzi, M., Annavaram, M., and Hall, M. Analyzing the effects of compiler optimizations on application reliability. In *2011 IEEE International Symposium on Workload Characterization (IISWC)* (2011), pp. 184–193.

[19] Dijkstra, E. W. Letters to the editor: go to statement considered harmful. *Communications of the ACM 11*, 3 (1968), 147–148.

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

[21] Duan, Y., Li, X., Wang, J., and Yin, H. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and Distributed System Security Symposium* (2020).

[22] Dwarampudi, M., and Reddy, N. V. S. Effects of padding on lstms and cnns. *CoRR abs/1903.07288* (2019).

[23] Engel, F., Leupers, R., Ascheid, G., Ferger, M., and Beemster, M. Enhanced structural analysis for c code reconstruction from ir code. In *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems* (2011), ACM, pp. 21–27.

[24] Eschweiler, S., Yakdan, K., and Gerhards-Padilla, E. discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS* (2016), vol. 52, pp. 58–79.

[25] Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B., and Yin, H. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), pp. 480–491.

[26] Fischer, F., Böttinger, K., Xiao, H., Stransky, C., Acar, Y., Backes, M., and Fahl, S. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), IEEE, pp. 121–136.

[27] Flake, H. Structural comparison of executable objects. In *Detection of intrusions and malware & vulnerability assessment, GI SIG SIDAR workshop, DIMVA 2004* (2004), Gesellschaft für Informatik eV.

[28] Fowler, G., and Vo, P. Fowler/noll/vo (fnv) hash (1991). *URL: http://isthe. com/chongo/tech/comp/fnv 5* (1991).

[29] Fowler, M. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[30] Gao, D., Reiter, M. K., and Song, D. Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security* (2008), Springer, pp. 238–255.

[31] German, D. M., Di Penta, M., Gueheneuc, Y.-G., and Antoniol, G. Code siblings: Technical and legal implications of copying code between applications. In *2009 6th IEEE International Working Conference on Mining Software Repositories* (2009), IEEE, pp. 81–90.

[32] Göde, N., and Koschke, R. Incremental clone detection. In *2009 13th European conference on software maintenance and reengineering* (2009), IEEE, pp. 219–228.

[33] Gonzalvez, A., and Lashermes, R. A case against indirect jumps for secure programs. In *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering* (2019), pp. 1–10.

[34] Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[35] Group, A., et al. Armv8 instruction set overview. *vol. PRD03-GENC-010197 15*, 11 (2011), 10.

[36] He, J., Ivanov, P., Tsankov, P., Raychev, V., and Vechev, M. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), pp. 1667–1680.

[37] Hemel, A., Kalleberg, K. T., Vermaas, R., and Dolstra, E. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories* (2011), ACM, pp. 63–72.

[38] Higo, Y., Yasushi, U., Nishino, M., and Kusumoto, S. Incremental code clone detection: A pdg-based approach. In *2011 18th Working Conference on Reverse Engineering* (2011), pp. 3–12.

[39] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural computation 9* (12 1997), 1735–80.

[40] Hoste, K., and Eeckhout, L. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization* (2008), pp. 165–174.

[41] Hu, Y., Zhang, Y., Li, J., and Gu, D. Binary code clone detection across architectures and compiling configurations. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)* (2017), IEEE, pp. 88–98.

[42] Hu, Y., Zhang, Y., Li, J., Wang, H., Li, B., and Gu, D. Binmatch: A semantics-based hybrid approach on binary code clone analysis. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2018), IEEE, pp. 104–114.

[43] Hummel, B., Juergens, E., Heinemann, L., and Conradt, M. Index-based code clone detection: incremental, distributed, scalable. In *2010 IEEE International Conference on Software Maintenance* (2010), IEEE, pp. 1–9.

[44] Inoue, K., and Roy, C. K. *Code Clone Analysis*. Springer, 2021.

[45] Ishihara, T., Hotta, K., Higo, Y., Igaki, H., and Kusumoto, S. Inter-project functional clone detection toward building libraries-an empirical study on 13,000 projects. In *2012 19th Working Conference on Reverse Engineering* (2012), IEEE, pp. 387–391.

[46] Jhi, Y.-C., Wang, X., Jia, X., Zhu, S., Liu, P., and Wu, D. Value-based program characterization and its application to software plagiarism detection. In *Proceedings of the 33rd international conference on software engineering* (2011), pp. 756–765.

[47] Jiang, L., Misherghi, G., Su, Z., and Glondu, S. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering* (2007), IEEE Computer Society, pp. 96–105.

[48] Johnson, J. H. Substring matching for clone detection and change tracking. In *ICSM* (1994), vol. 94, pp. 120–126.

[49] Jones, M. T. Optimization in gcc. *Linux journal 2005*, 131 (2005), 11.

[50] Juergens, E., Deissenboeck, F., Hummel, B., and Wagner, S. Do code clones matter? In *2009 IEEE 31st International Conference on Software Engineering* (2009), IEEE, pp. 485–495.

[51] Kamiya, T., Kusumoto, S., and Inoue, K. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering 28*, 7 (2002), 654–670.

[52] Katz, O., Olshaker, Y., Goldberg, Y., and Yahav, E. Towards neural decompilation. *arXiv preprint arXiv:1905.08325* (2019).

[53] Keivanloo, I., Roy, C. K., and Rilling, J. Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In *2012 6th International Workshop on Software Clones (IWSC)* (2012), IEEE, pp. 36–42.

[54] Kim, M., Sazawal, V., Notkin, D., and Murphy, G. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (2005), pp. 187–196.

[55] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[56] Kononenko, O., Zhang, C., and Godfrey, M. W. Compiling clones: What happens? In *2014 IEEE International Conference on Software Maintenance and Evolution* (2014), IEEE, pp. 481–485.

[57] Krinke, J. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering* (2001), IEEE, pp. 301–309.

[58] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.

[59] Lakhotia, A., Preda, M. D., and Giacobazzi, R. Fast location of similar code fragments using semantic'juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop* (2013), pp. 1–6.

[60] Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research 18*, 1 (2017), 6765–6816.

[61] Li, Z., Lu, S., Myagmar, S., and Zhou, Y. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering 32*, 3 (2006), 176–192.

[62] Ma, Z., Ge, H., Liu, Y., Zhao, M., and Ma, J. A combination method for android malware detection based on control flow graphs and machine learning algorithms. *IEEE Access 7* (2019), 21235–21245.

[63] Maas, A. L., Hannun, A. Y., and Ng, A. Y. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml* (2013), vol. 30, p. 3.

[64] Ming, J., Xu, D., Jiang, Y., and Wu, D. {BinSim}: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *26th USENIX Security Symposium (USENIX Security 17)* (2017), pp. 253–270.

[65] Morasca, S. Why do developers adopt open source software? past, present and future. In *Open Source Systems: 15th IFIP WG 2.13 International Conference, OSS 2019, Montreal, QC, Canada, May 26–27, 2019, Proceedings* (2019), Springer, p. 104.

[66] Moser, A., Kruegel, C., and Kirda, E. Limits of static analysis for malware detection. In *Twenty-third annual computer security applications conference (ACSAC 2007)* (2007), IEEE, pp. 421–430.

[67] Nair, V., and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)* (2010), pp. 807–814.

[68] Oh, J. Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries. In *Blackhat technical security conference* (2009).

[69] Pallister, J., Hollis, S. J., and Bennett, J. Identifying compiler options to minimize energy consumption for embedded platforms. *The Computer Journal 58*, 1 (2015), 95–109.

[70] Pascanu, R., Stokes, J. W., Sanossian, H., Marinescu, M., and Thomas, A. Malware classification with recurrent networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2015), IEEE, pp. 1916–1920.

[71] Pewny, J., Garmany, B., Gawlik, R., Rossow, C., and Holz, T. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 709–724.

[72] Pizzolotto, D., and Inoue, K. Blanker: A refactor-oriented cloned source code normalizer. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)* (2020), IEEE, pp. 22–25.

[73] Pizzolotto, D., and Inoue, K. Identifying compiler and optimization options from binary code using deep learning approaches. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2020), IEEE, pp. 232–242.

[74] Pizzolotto, D., and Inoue, K. Binary software compiled for different architectures with different optimization levels, Apr. 2021.

[75] Prechelt, L., Malpohl, G., Philippsen, M., et al. Finding plagiarisms among a set of programs with jplag. *J. UCS 8*, 11 (2002), 1016.

[76] Ragkhitwetsagul, C., and Krinke, J. Using compilation/decompilation to enhance clone detection. In *2017 IEEE 11th International Workshop on Software Clones (IWSC)* (2017), IEEE, pp. 1–7.

[77] Ragkhitwetsagul, C., Krinke, J., Paixao, M., Bianco, G., and Oliveto, R. Toxic code snippets on stack overflow. *IEEE Transactions on Software Engineering* (2019).

[78] Rosenblum, N., Miller, B. P., and Zhu, X. Recovering the toolchain provenance of binary code. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (2011), pp. 100–110.

[79] Rosenblum, N. E., Miller, B. P., and Zhu, X. Extracting compiler provenance from program binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (2010), pp. 21–28.

[80] Roy, C. K., and Cordy, J. R. A survey on software clone detection research. *Queen's School of computing TR 541*, 115 (2007), 64–68.

[81] Roy, C. K., and Cordy, J. R. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th iEEE international conference on program comprehension* (2008), IEEE, pp. 172–181.

[82] Roy, C. K., Cordy, J. R., and Koschke, R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming 74*, 7 (2009), 470–495.

[83] Rust Foundation. The hir, Accessed 2022-06. `https://rustc-dev-guide.rust-lang.org/hir.html`.

[84] Rust Foundation. Macro expansion, Accessed 2022-06. `https://rustc-dev-guide.rust-lang.org/macro-expansion.html`.

[85] Rust Foundation. The mir (mid-level ir), Accessed 2022-06. `https://rustc-dev-guide.rust-lang.org/mir/index.html`.

[86] Rust Foundation. The rustonomicon - meet safe and unsafe, Accessed 2022-06. `https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html`.

[87] Rust Foundation. The thir, Accessed 2022-06. `https://rustc-dev-guide.rust-lang.org/thir.html`.

[88] Rust Foundation. What is ownership?, Accessed 2022-06. `https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html`.

[89] Sæbjørnsen, A., Willcock, J., Panas, T., Quinlan, D., and Su, Z. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis* (2009), ACM, pp. 117–128.

[90] Saini, V., Farmahinifarahani, F., Lu, Y., Baldi, P., and Lopes, C. V. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2018), pp. 354–365.

[91] Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K., and Lopes, C. V. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering* (2016), pp. 1157–1168.

[92] Schulman, A. Finding binary clones with opstrings function digests: Part iii. *Dr. Dobb's Journal 30*, 9 (2005), 64.

[93] Selim, G. M., Foo, K. C., and Zou, Y. Enhancing source-based clone detection using intermediate representation. In *2010 17th Working Conference on Reverse Engineering* (2010), IEEE, pp. 227–236.

[94] Semura, Y., Yoshida, N., Choi, E., and Inoue, K. Ccfindersw: Clone detection tool with flexible multilingual tokenization. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)* (2017), IEEE, pp. 654–659.

[95] Sharir, M. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages 5*, 3-4 (1980), 141–153.

[96] Sheneamer, A., and Kalita, J. Semantic clone detection using machine learning. In *2016 15th IEEE international conference on machine learning and applications (ICMLA)* (2016), IEEE, pp. 1024–1028.

[97] Shin, E. C. R., Song, D., and Moazzezi, R. Recognizing functions in binaries with neural networks. In *24th {USENIX} Security Symposium ({USENIX} Security 15)* (2015), pp. 611–626.

[98] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[99] Stephenson, M., Amarasinghe, S., Martin, M., and O'Reilly, U.-M. Meta optimization: Improving compiler heuristics with machine learning. *ACM sigplan notices 38*, 5 (2003), 77–90.

[100] Svajlenko, J., and Roy, C. K. Evaluating clone detection tools with bigclonebench. In *2015 IEEE international conference on software maintenance and evolution (ICSME)* (2015), IEEE, pp. 131–140.

[101] Triantafyllis, S., Vachharajani, M., Vachharajani, N., and August, D. I. Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* (2003), IEEE, pp. 204–215.

[102] Ullmann, J. R. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM) 23*, 1 (1976), 31–42.

[103] Wang, R., Shoshitaishvili, Y., Bianchi, A., Machiry, A., Grosen, J., Grosen, P., Kruegel, C., and Vigna, G. Ramblr: Making reassembly great again. In *NDSS* (2017).

[104] Wang, Z., and O'Boyle, M. Machine learning in compiler optimization. *Proceedings of the IEEE 106*, 11 (2018), 1879–1901.

[105] Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., and Song, D. 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* (2017), pp. 363–376.

[106] Xue, H., Venkataramani, G., and Lan, T. Clone-slicer: Detecting domain specific binary code clones through program slicing. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation* (2018), pp. 27–33.

[107] Yakdan, K., Eschweiler, S., Gerhards-Padilla, E., and Smith, M. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS* (2015).

[108] Yoshida, N., and Choi, E. Clone evolution and management. In *Code Clone Analysis*. Springer, 2021, pp. 197–208.

[109] Yu, D., Wang, J., Wu, Q., Yang, J., Wang, J., Yang, W., and Yan, W. Detecting java code clones with multi-granularities based on bytecode. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)* (2017), vol. 1, IEEE, pp. 317–326.

[110] Zhang, F., Jhi, Y.-C., Wu, D., Liu, P., and Zhu, S. A first step towards algorithm plagiarism detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (2012), pp. 111–121.

[111] Zhang, H., and Sakurai, K. A survey of software clone detection from security perspective. *IEEE Access 9* (2021), 48157–48173.

[112] Zhu, W., Yoshida, N., Kamiya, T., Choi, E., and Takada, H. Msccd: Grammar pluggable clone detection based on antlr parser generation. *arXiv preprint arXiv:2204.01028* (2022).

[113] Zuo, F., Li, X., Young, P., Luo, L., Zeng, Q., and Zhang, Z. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint arXiv:1808.04706* (2018).